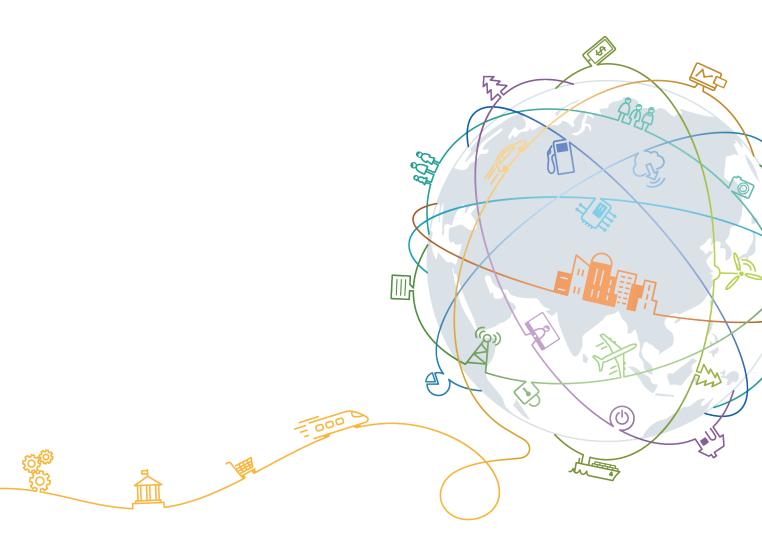
Atlas 500

Matrix API 参考

文档版本 01

发布日期 2019-09-30





版权所有 © 华为技术有限公司 2019。 保留一切权利。

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

商标声明



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

注意

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

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

华为技术有限公司

地址: 深圳市龙岗区坂田华为总部办公楼 邮编: 518129

网址: http://e.huawei.com

前言

读者对象

本文档主要适用于以下工程师:

- 华为技术支持工程师
- 渠道伙伴技术支持工程师
- ISV软件工程师

符号约定

在本文中可能出现下列标志,它们所代表的含义如下。

符号	说明	
▲ 危险	用于警示紧急的危险情形,若不避免,将会导致人员死亡 或严重的人身伤害。	
▲警告	用于警示潜在的危险情形,若不避免,可能会导致人员死 亡或严重的人身伤害。	
△ 注意	用于警示潜在的危险情形,若不避免,可能会导致中度或 轻微的人身伤害。	
注意	用于传递设备或环境安全警示信息,若不避免,可能会导致设备损坏、数据丢失、设备性能降低或其它不可预知的结果。 "注意"不涉及人身伤害。	
□ 说明	用于突出重要/关键信息、最佳实践和小窍门等。 "说明"不是安全警示信息,不涉及人身、设备及环境伤害。	

修改记录

文档版本	发布日期	修改说明
01	2019-09-30	第一次正式发布版本。

目录

前言	ii
1 简介	1
2.1 流程串接接口(C 语言)	
2.1.1 HIAI Init(与 C++共用)	
2.1.2 HIAI_CreateGraph.	
2.1.3 HIAI_DestroyGraph	
2.2 流程串接接口(C++语言)	
2.2.1 Graph::GetInstance	5
2.2.2 Graph::CreateGraph(根据配置文件创建 Graph)	6
2.2.3 Graph::CreateGraph(根据配置文件创建 Graph,并将生成的 Graph 回写到 list 中)	7
2.2.4 Graph::CreateGraph(根据 Protobuf 数据格式创建 Graph)	7
2.2.5 Graph::DestroyGraph	8
2.2.6 Graph::UpdateEngineConfig.	9
2.3 数据传输接口(C 语言)	10
2.3.1 HIAI_SetDataRecvFunctor	10
2.3.2 HIAI_C_SendData	11
2.4 数据传输接口(C++语言)	12
2.4.1 Graph::SetDataRecvFunctor	12
2.4.2 Graph::SendData.	13
2.4.3 HIAI_SendData.	14
2.5 接口使用示例	15
2.5.1 C 语言接口使用示例	15
2.5.2 C++接口使用示例(根据配置文件创建 Graph)	
2.5.3 C++接口使用示例(根据配置文件创建 Graph,并将生成的 Graph 回写到 list 中)	19
3 Engine 实现接口(C++语言)	22
3.1 Engine 的构造函数与析构函数	22
3.2 Engine::Init	22
3.3 宏:HIAI_DEFINE_PROCESS	24
3.4 宏:HIAI_IMPL_ENGINE_PROCESS	24
3.5 Engine::SetDataRecvFunctor	26
3.6 Engine::SendData	27

3.7 调用示例	28
4 模型管家接口(C++语言)	31
4.1 离线模型管家	
4.1.1 AIModelManager::Init	31
4.1.2 AIModelManager::SetListener	33
4.1.3 AIModelManager::Process	33
4.1.4 AIModelManager::CreateOutputTensor	34
4.1.5 AIModelManager::CreateInputTensor	35
4.1.6 AIModelManager::IsPreAllocateOutputMem	35
4.1.7 AIModelManager::GetModelIOTensorDim	36
4.1.8 AIModelManager::GetMaxUsedMemory	37
4.1.9 AISimpleTensor::SetBuffer	37
4.1.10 AITensorFactory::CreateTensor	38
4.1.11 调用示例	39
4.2 AIPP 配置接口	40
4.2.1 简介	40
4.2.2 SetDynamicInputIndex	41
4.2.3 SetDynamicInputEdgeIndex	41
4.2.4 SetInputFormat	42
4.2.5 SetCscParams (设置默认参数值)	43
4.2.6 SetCscParams (根据需求设置参数值)	45
4.2.7 SetRbuvSwapSwitch	47
4.2.8 SetAxSwapSwitch	47
4.2.9 SetSrcImageSize.	48
4.2.10 SetCropParams.	48
4.2.11 SetPaddingParams	49
4.2.12 SetDtcPixelMean.	51
4.2.13 SetDtcPixelMin	
4.2.14 SetPixelVarReci	53
4.2.15 SetInputDynamicAIPP	54
4.2.16 GetDynamicInputIndex	55
4.2.17 GetDynamicInputEdgeIndex	56
4.2.18 调用示例	56
4.3 数据类型	59
4.3.1 AIConfigItem	59
4.3.2 AIConfig	59
4.3.3 AITensorParaDescription.	59
4.3.4 AITensorDescription.	59
4.3.5 AITensorDescriptionList	59
4.3.6 AIModelDescription	60
4.3.7 AINNNodeDescription.	60
4.3.8 AINNNodeDescriptionList.	61

4.3.9 AIAPIDescription.	61
4.3.10 AIAPIDescriptionList.	61
4.3.11 AIOPDescription	
4.3.12 AIOPDescriptionList	61
4.3.13 NodeDesc.	62
4.3.14 EngineDesc.	62
4.3.15 GraphInitDesc.	62
4.3.16 GeneralFileBuffer	62
4.3.17 AIContext.	63
4.3.18 TensorDimension.	63
4.3.19 IAIListener	64
4.3.20 IAITensor	65
4.4 其他用于编译依赖的接口	65
4.4.1 AIAlgAPIFactory	66
4.4.2 AIAlgAPIRegisterar	66
4.4.3 REGISTER_ALG_API_UNIQUE	67
4.4.4 REGISTER_ALG_API	67
4.4.5 AIModelManager	
4.4.6 getModelInfo	68
4.4.7 IAINNNode	68
4.4.8 AINNNodeFactory	69
4.4.9 AINNNodeRegisterar	
4.4.10 REGISTER_NN_NODE	71
4.4.11 AITensorGetBytes	71
4.4.12 AITensorFactory	71
4.4.13 REGISTER_TENSOR_CREATER_UNIQUE	
4.4.14 REGISTER_TENSOR_CREATER	72
4.4.15 AISimpleTensor	72
4.4.16 AINeuralNetworkBuffer	74
4.4.17 AIImageBuffer	76
4.4.18 HIAILog	77
4.4.19 HIAI_ENGINE_LOG	78
4.5 异常处理	79
5 辅助接口	80
5.1 数据获取接口(C++语言)	
5.1.1 获取 Device 数目	
5.1.2 获取第一个 DeviceID	
5.1.3 获取第一个 GraphID	
5.1.4 获取下一个 DeviceID	
5.1.5 获取下一个 GraphID	
5.1.6 获取 Engine 指针	
5.1.7 获取 Graph 的 GraphId	
2.1.1 3/7-1/2 Οταρίτ μα Οταρίτια	

5.1.8 获取 Graph 的 DeviceID	84
5.1.9 获取 Engine 的 GraphId	85
5.1.10 获取 Engine 队列最大大小	85
5.1.11 获取 Engine 指定端口当前队列大小	85
5.1.12 解析 Matrix 配置文件	86
5.1.13 获取 PCIe 的 Info	86
5.1.14 获取版本号	87
5.1.15 获取 OamConfig 智能指针	88
5.2 数据类型序列化和反序列化(C++语言)	88
5.2.1 宏:HIAI_REGISTER_DATA_TYPE	88
5.2.2 宏:HIAI_REGISTER_TEMPLATE_DATA_TYPE	89
5.2.3 宏:HIAI_REGISTER_SERIALIZE_FUNC	90
5.2.4 Graph::ReleaseDataBuffer	92
5.2.5 接口使用示例	92
5.3 内存管理(C 语言)	95
5.3.1 HIAI_DMalloc	95
5.3.2 HIAI_DFree.	97
5.3.3 HIAI_DVPP_DMalloc	97
5.3.4 HIAI_DVPP_DFree	98
5.4 内存管理(C++语言)	99
5.4.1 HIAIMemory::HIAI_DMalloc	99
5.4.2 HIAIMemory::HIAI_DFree	102
5.4.3 HIAIMemory:: IsDMalloc	102
5.4.4 HIAIMemory::HIAI_DVPP_DMalloc	
5.4.5 HIAIMemory::HIAI_DVPP_DFree	104
5.5 日志(C++语言)	105
5.5.1 错误码注册	
5.5.1.1 宏 HIAI_DEF_ERROR_CODE	
5.5.1.2 接口使用示例	
5.5.2 日志打印	106
5.5.2.1 日志打印格式 1	106
5.5.2.2 日志打印格式 2	107
5.5.2.3 日志打印格式 3	107
5.5.2.4 日志打印格式 4	
5.5.2.5 日志打印格式 5	
5.5.2.6 日志打印格式 6	
5.5.2.7 日志打印格式 7	110
5.5.2.8 日志打印格式 8	110
5.6 队列管理 MultiTypeQueue 接口(C++语言)	111
5.6.1 MultiTypeQueue 构造函数	111
5.6.2 PushData	112
5.6.3 FrontData	112

5.6.4 PopData	112
5.6.5 PopAllData	113
5.6.6 接口使用示例	114
5.7 事件注册接口(C++语言)	114
5.7.1 Graph::RegisterEventHandle	114
5.8 其他(C++语言)	114
5.8.1 DataRecvInterface::RecvData	114
5.8.2 Graph::SetPublicKeyForSignatureEncryption.	115
6 低功耗接口(C++语言)	117
6.1 函数接口	117
6.1.1 PowerState::SubscribePowerStateEvent	117
6.1.2 PowerState::UnsubscribePowerStateEvent	118
6.2 数据类型	118
6.2.1 枚举: PowerState::DEVICE_POWER_STATE	118
6.2.2 PowerState::PowerStateNotifyCallbackT	
6.3 接口使用示例	119
7 附录	120
7.1 Matrix 数据类型	120
7.1.1 Protobuffer 数据类型	120
7.1.2 Matrix 自定义数据类型	124
7.2 Matrix 已经注册的数据结构	124
7.3 示例	140
7.3.1 编排配置示例	140
7.3.2 性能优化传输示例	141

1 简介

Matrix运行于操作系统之上,业务应用之下。屏蔽操作系统差异,为应用提供统一的标准化接口。Matrix具有多节点调度能力和多进程管理,可以根据配置文件完成业务流程的建立和运行,以及相关的统计信息汇总等。

Matrix总体逻辑包含3个主要的部分,Matrix Agent(运行在Host侧)、Matrix Daemon(运行在Device侧)和Matrix Service(运行在Device侧)。

∭说明

- Host指与Device相连接的X86服务器、ARM服务器或者WindowsPC, 会利用Device提供的NN (Neural-Network) 计算能力,完成业务。
- Device指安装了Ascend 310芯片的硬件设备,利用PCIe接口与Host侧链接,为Host提供NN计算能力。
- DVPP (Digital Vision Pre-Processing): 主要实现视频解码、视频编码、JPEG编解码、PNG解码、视觉预处理。
- Framework是深度学习框架,可以将caffe等开源框架模型转换成Mind Studio支持的模型。
- CCE (cube-based computing engine) 加速库通过API的方式,为上层应用(为机器学习提供的各种framework或者Application)提供加速。
- Runtime运行于APP进程空间,为APP提供了Ascend 310芯片的Memory管理、Device管理、Stream管理、Event管理、Kernel执行等功能。

Matrix为通用业务流程执行引擎,主要包含:

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

您可以在DDK包的安装目录下的"ddk/include/inc/hiaiengine"目录下查看接口的定义文件。如果通过引导安装的方式同时安装Mind Studio和DDK,您可以使用Mind Studio

安装用户登录Mind Studio服务器,在 "~/tools/che/ddk/ddk/include/inc/hiaiengine"路径下查看接口定义文件。每个接口对应的定义文件请参见具体的接口描述。

2 流程编排接口

- 2.1 流程串接接口(C语言)
- 2.2 流程串接接口(C++语言)
- 2.3 数据传输接口(C语言)
- 2.4 数据传输接口(C++语言)
- 2.5 接口使用示例

2.1 流程串接接口(C语言)

2.1.1 HIAI_Init(与C++共用)

初始化Matrix的HDC(Host Device Communication)通讯模块。如果Graph配置文件中已经指定运行Graph的设备编号,则以配置文件为准,否则用此处设置的deviceID为准。该接口在init.h中定义。

函数格式

HIAI_StatusT HIAI_Init(uint32_t deviceID)

□说明

HIAI StatusT结构体的定义请参见7.1.2 Matrix自定义数据类型。

参数说明

参数	说明	取值范围
deviceID	设备编号,默认值为0。	0~63

返回值

返回的部分错误码请参见"错误码示例"中的"错误码"列。

错误码示例

序号 错误码 1 HIAI_OK		错误码描述
		running ok
2	HIAI_INIT_FAIL	hiai init fail

2.1.2 HIAI_CreateGraph

创建并启动整个Graph,该接口支持单个配置文件中的graph之间的串联,不支持多个配置文件之间的graph串联。只允许在Host侧或Ascend 310 RC上被调用。该接口在c_graph.h中定义。

函数格式

HIAI StatusT HIAI CreateGraph(const char* configFile, size t len)

参数说明

参数	说明	取值范围
configFile	配置文件路径。 请确保传入的文件路径是 正确路径。	单个配置文件中最大支持 2048个graph。但受系统资 源限制,根据硬件配置不 同,以及graph大小不同, 实际最多可能只支持几个 或十几个graph。
len	配置文件名的长度。	1~255

返回值

返回的部分错误码请参见"错误码示例"中的"错误码"列。

错误码示例

序号	错误码	错误码描述
1	HIAI_OK	running ok
2	HIAI_GRAPH_PROTO_FILE_PA RSE_FAILED	ParseFromString return failed

2.1.3 HIAI_DestroyGraph

销毁指定的Graph, 只允许在Host侧或Ascend 310 RC上被调用。该接口在c_graph.h中定义。

函数格式

HIAI_StatusT HIAI_DestroyGraph (uint32_t graphID)

参数说明

参数	说明	取值范围
graphID	指定的Graph ID。	-

返回值

返回的部分错误码请参见"错误码示例"中的"错误码"列。

错误码示例

序号	错误码	错误码描述
1	HIAI_OK	running ok
2	HIAI_GRAPH_DESTROY_TIME OUT	destroy timeout
3	HIAI_GRAPH_ENGINE_DESTR OY_FAILED	engine destroy failed

2.2 流程串接接口(C++语言)

2.2.1 Graph::GetInstance

获取Graph实例,在创建Graph成功后可以正常使用该接口。该接口在graph.h中定义。

函数格式

static std::shared ptr<Graph> Graph::GetInstance(uint32 t graphID)

参数说明

参数	说明	取值范围
graphID	目标graph的id。	配置文件里配置的Graph编号。

返回值

Graph的智能指针。

错误码示例

序号	错误码	错误码描述
1	HIAI_OK	running ok
2	HIAI_GRAPH_PROTO_FILE_PA RSE_FAILED	ParseFromString return failed

2.2.2 Graph::CreateGraph(根据配置文件创建 Graph)

创建并启动整个Graph,该接口支持单个配置文件中的graph之间的串联,不支持多个配置文件之间的graph串联。该接口在graph.h中定义。

只允许在Host侧或Ascend 310 RC上被调用。

函数格式

static HIAI StatusT Graph::CreateGraph(const std::string& configFile)

参数说明

参数	说明	取值范围
configFile	整个Graph的配置文件路 径。 请确保传入的文件路径是 正确路径。	单个配置文件中最大支持 2048个graph。但受系统资 源限制,根据硬件配置不 同,以及graph大小不同, 实际最多可能只支持几个 或十几个graph。

返回值

返回的部分错误码请参见"错误码示例"中的"错误码"列。

错误码示例

序号	错误码	错误码描述
1	HIAI_OK	running ok
2	HIAI_GRAPH_GET_INSTANCE _NULL	get instance null

2.2.3 Graph::CreateGraph(根据配置文件创建 Graph,并将生成的Graph 回写到 list 中)

重载**2.2.2 Graph::CreateGraph(根据配置文件创建Graph)**,Matrix根据配置文件创建Graph,并将生成的Graph回写到list中,供用户使用。该接口在graph.h中定义。

只允许在Host侧或Ascend 310 RC上被调用。

函数格式

static HIAI_StatusT Graph::CreateGraph(const std::string& configFile,
std::list<std::shared_ptr<Graph>>& graphList)

参数说明

参数	说明	取值范围
configFile	整个Graph的配置文件路 径。 请确保传入的文件路径是 正确路径。	单个配置文件中最大支持 2048个graph。但受系统资 源限制,根据硬件配置不 同,以及graph大小不同, 实际最多可能只支持几个 或十几个graph。
graphList	用户定义的lsit,Matrix将 生成的Graph回写到list, 以供用户使用。	-

返回值

返回的部分错误码请参见"错误码示例"中的"错误码"列。

错误码示例

序号	错误码	错误码描述
1	HIAI_OK	running ok
2	HIAI_GRAPH_GET_INSTANCE _NULL	get instance null

2.2.4 Graph::CreateGraph (根据 Protobuf 数据格式创建 Graph)

创建并启动整个Graph,该接口支持根据Protobuf数据格式创建Graph。该接口在graph.h中定义。

只允许在Host侧或Ascend 310 RC上被调用。

函数格式

static HIAI_StatusT Graph::CreateGraph(const GraphConfigList& graphConfig)

参数说明

参数	说明	取值范围
graphConfig	Protobuf数据格式。 用户在编写代码时,可以选择以下定义 graphConfig 参数的方式: • 定义一个 GraphConfigList 类型的参数,作为 Graph::CreateGraph(const GraphConfig)接口的入参graphConfig)接口的入参graphConfig。 • 调用 5.1.12 解析Matrix 配置文件的接口后,获取graphConfigList参数值,作为Graph::CreateGraph(const GraphConfigList&graphConfigList&graphConfig)接口的入参graphConfig)接口的入参graphConfig)。	graph中的Engine数量必须 小于等于512。

返回值

返回的部分错误码请参见"错误码示例"中的"错误码"列。

错误码示例

序号	错误码	错误码描述
1	HIAI_OK	running ok
2	HIAI_GRAPH_GET_INSTANCE _NULL	get instance null

2.2.5 Graph::DestroyGraph

销毁整个Graph。该接口在graph.h中定义。 只允许在Host侧或Ascend 310 RC上被调用。

函数格式

static HIAI_StatusT Graph::DestroyGraph(uint32_t graphID)

参数说明

参数	说明	取值范围
graphID	需要销毁的graph的id	配置文件里配置的Graph编号

返回值

返回的部分错误码请参见"错误码示例"中的"错误码"列。

错误码示例

序号	错误码	错误码描述
1	HIAI_OK	running ok
2	HIAI_GRAPH_DESTROY_TIME OUT	destroy graph timeout
3	HIAI_GRAPH_ENGINE_DESTR OY_FAILED	destroy graph failed

2.2.6 Graph::UpdateEngineConfig

通过HiAI接口更新指定Engine的参数。该接口在graph.h中定义。

使用场景:

通过该接口, 更新流程运行过程中指定的运行参数。

例如:视频抓拍图场景,需要根据白天黑夜的变化更新运行流程,则需要通过UpdataEngineConfig函数将白天/黑夜的数值更新入Engine。

函数格式

static HIAI_StatusT Graph::UpdateEngineConfig(const uint32_t& graphId, const uint32_t& engineId, const AIConfig& aiConfig, const bool& syncFlag = false)

参数说明

参数	说明	取值范围
graphId	Graph ID	-
engineId	engine ID	-

参数	说明	取值范围
aiConfig	配置参数	-
syncFlag	同步/异步执行,同步则等 待返回结果	true/false

返回值

返回的部分错误码请参见"错误码示例"中的"错误码"列。

错误码示例

序号	错误码	错误码描述
1	HIAI_OK	running ok
2	HIAI_GRAPH_NOT_EXIST	graph not exist
3	HIAI_GRAPH_MEMORY_POOL _NOT_EXISTED	memory pool is not existed
4	HIAI_GRAPH_MALLOC_LARG ER	failed to malloc buffer due to the size larger than 128M
5	HIAI_MEMORY_POOL_UPDAT E_FAILED	failed to update memory pool
6	HIAI_GRAPH_SENDMSG_FAIL ED	hdc send msg failed
7	HIAI_GRAPH_MEMORY_POOL _INITED	memory pool has inited
8	HIAI_GRAPH_NO_MEMORY	no memory

2.3 数据传输接口(C语言)

2.3.1 HIAI_SetDataRecvFunctor

设置Graph中某Engine接收消息的回调函数。该接口在c_graph.h中定义。

∭说明

回调函数类型: typedef HIAI_StatusT (*HIAI_RecvDataCallbackT)(void*)

函数格式

HIAI_StatusT HIAI_SetDataRecvFunctor(HIAI_PortID_t targetPortConfig, HIAI RecvDataCallbackT recvCallback)

参数说明

参数	说明	取值范围
targetPortConfig	数据接收方的Graph ID、 Engine ID 和Port ID。	-
recvCallback	设置用户自定义的结果接 收方式。	-

返回值

返回的部分错误码请参见"错误码示例"中的"错误码"列。

错误码示例

序号	错误码	错误码描述
1	HIAI_OK	running ok
2	HIAI_PORT_ID_ERROR	port id error

2.3.2 HIAI_C_SendData

用户从外部向Matrix发送数据(或消息)。该接口在c_graph.h中定义。

川说明

- 用户自定义的消息类型需要调用5.2 数据类型序列化和反序列化(C++语言)先注册。
- 该消息指针dataPtr控制权为调用者所有, Matrix不管理该指针, 由调用者负责释放。
- SendData采用DMA传送方式,可能会影响CPU对中断请求的及时响应与处理,例如调用new或者malloc分配内存。

函数格式

HIAI_StatusT HIAI_C_SendData(HIAI_PortID_t targetPortConfig, const char* messageName, size t len, void* dataPtr)

参数说明

参数	说明	取值范围
targetPortConfig	数据接收方的Graph ID、Engine ID和Port ID。	-
messageName	当前发送的消息名。	-
len	消息名称的长度。	-

参数	说明	取值范围
dataPtr	指向具体的消息指针(该消息指针控制权为调用者所有,Matrix不管理该指针,由调用者负责释放)。	-

返回值

返回的部分错误码请参见"错误码示例"中的"错误码"列。

错误码示例

序号	错误码	错误码描述
1	HIAI_OK	running ok
2	HIAI_GRAPH_ENGINE_NOT_E XIST	engine not exist
3	HIAI_GRAPH_NOT_EXIST	graph not exist
4	HIAI_GRAPH_NO_USEFUL_ME MORY	no useful memory
5	HIAI_GRAPH_SENDMSG_FAIL ED	hdc send msg failed
6	HIAI_GRAPH_INVALID_VALUE	graph invalid value

2.4 数据传输接口(C++语言)

2.4.1 Graph::SetDataRecvFunctor

设置Graph中某Engine接收消息的回调函数。调用该接口的代码的运行环境必须与Engine一致,例如,Engine在Device侧运行,那么调用该函数的代码也必须运行在Device。该接口在graph.h中定义。

该接口需要与**DataRecvInterface::RecvData**接口配合使用,一般用于将推理结果返回给用户,详细原理如下:

- 1. 定义一个DataRecvInterface类的子类(例如DdkDataRecvInterface),初始化一个子类的对象。
- 2. 调用Graph::SetDataRecvFunctor接口将DdkDataRecvInterface类的对象设置成回调函数。
- 3. 调用Engine::SendData函数将对应的输入数据发送到对应的输出端口。
- 4. 调用DdkDataRecvInterface类的RecvData函数返回推理结果。

函数格式

static HIAI_StatusT Graph::SetDataRecvFunctor(const EnginePortID&
targetPortConfig, const std::shared_ptr<DataRecvInterface>& dataRecv);

参数说明

参数	说明	取值范围
targetPortConfig	数据接收方的Graph ID、 Engine ID 和Port ID。	-
dataRecv	用户自定义的数据接收回 调函数。	-

返回值

返回的部分错误码请参见"错误码示例"中的"错误码"列。

错误码示例

序号	错误码	错误码描述
1	HIAI_OK	running ok
2	HIAI_PORT_ID_ERROR	port id error

2.4.2 Graph::SendData

用户从外部向Matrix发送void类型的数据到指定的端口。该接口在graph.h中定义。

∭说明

SendData采用DMA传送方式,可能会影响CPU对中断请求的及时响应与处理,例如调用new或者 malloc分配内存。

函数格式

HIAI_StatusT Graph::SendData(const EnginePortID& targetPortConfig, const std::string& messageName, const std::shared_ptr<void>& dataPtr, const uint32_t timeOut = 500)

参数说明

参数	说明	取值范围
targetPortConfig	数据接收方的Graph ID、 Engine ID 和Port ID。	-
messageName	消息的名称。	-

参数	说明	取值范围
dataPtr	消息的指针。	-
timeOut	调用该接口发送数据时的一次超时时间,不指定timeout参数时,默认的一次超时时间为500ms。若发送数据超时,系统后台会再次尝试发送数据,最大尝试16次。	-

返回值

返回的部分错误码请参见"错误码示例"中的"错误码"列。

错误码示例

序号	错误码	错误码描述
1	HIAI_OK	running ok
2	HIAI_GRAPH_ENGINE_NOT_E XIST	engine not exist
3	HIAI_GRAPH_NOT_EXIST	graph not exist
4	HIAI_GRAPH_NO_USEFUL_ME MORY	no useful memory
5	HIAI_GRAPH_SENDMSG_FAIL ED	hdc send msg failed
6	HIAI_GRAPH_INVALID_VALUE	graph invalid value

2.4.3 HIAI_SendData

用户从外部向Matrix发送void类型的数据到指定的端口。该接口在c_graph.h中定义。

□说明

SendData采用DMA传送方式,可能会影响CPU对中断请求的及时响应与处理,例如调用new或者malloc分配内存。

函数格式

HIAI_StatusT HIAI_SendData (HIAI_PortID_t targetPortConfig, const std::string& messageName, std::shared_ptr<void>& dataPtr, uint32_t timeOut=0)

参数说明

参数	说明	取值范围
targetPortConfig	数据接收方的Graph ID、 Engine ID和Port ID。	-
messageName	消息的名称。	-
dataPtr	消息的指针。	-
timeOut	发送数据时用户可以添加 time_out作为时延,单位为 毫秒(ms)。如果出现发送 队列满或者内存池不足的 情况,根据时延作为阻 塞。默认为0,即不阻塞。	-

返回值

返回的部分错误码请参见"错误码示例"中的"错误码"列。

错误码示例

序号	错误码	错误码描述
1	HIAI_OK	running ok
2	HIAI_GRAPH_ENGINE_NOT_E XIST	engine not exist
3	HIAI_GRAPH_NOT_EXIST	graph not exist
4	HIAI_GRAPH_NO_USEFUL_ME MORY	no useful memory
5	HIAI_GRAPH_SENDMSG_FAIL ED	hdc send msg failed
6	HIAI_GRAPH_INVALID_VALUE	graph invalid value

2.5 接口使用示例

2.5.1 C 语言接口使用示例

```
/**

* @file graph_c_api_example.cpp

*

* Copyright(C), 2017 - 2017, Huawei Tech. Co., Ltd. ALL RIGHTS RESERVED.

*

* @Source Files for HIAI Graph Orchestration

*

* @version 1.0
```

```
*/
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include "hiaiengine/c_api.h"
#include "user_def_data_type.h"
* Normally the mandatory steps to setup the whole HIAI graph include:
* 1) call HIAI_Init() API to perform global system initialization;
  2) create HIAI CreateGraph to create and start graph
* 3) set profile config if necessary
* 4) call HIAI_DestoryGraph to destroy the graph finally
// Just define variables which are used in only this example
// and are not required in real user case.
#define MATRIX_USER_SPECIFY_ENGINE_PRIORITY (10)
static uint32_t g_graph_id = 100;
static uint32_t src_engine_id = 1000; // 定义在graph.prototxt static uint32_t dest_engine_id = 1002; // 定义在graph.prototxt static const char graph_config_proto_file[] = "llt/hiaiengine/st/examples/config/graph.prototxt";
static const char graph_config_len = sizeof(graph_config_proto_file);
// Define End
// 用户自定义数据接收 (recv_data 由框架自动释放)
HIAI_StatusT UserDefDataRecvCallBack(void* recv_data)
   // 处理接收到的数据
    return HIAI_OK;
/******************************
* HIAIEngine Example
* Graph:
* SrcEngine <--> Engine <--> DestEngine
HIAI StatusT HIAI InitAndStartGraph()
    // Step1: Global System Initialization before using Matrix
   HIAI_StatusT status = HIAI_Init(0);
   // Step2: Create and Start the Graph
    status = HIAI_CreateGraph(graph_config_proto_file, graph_config_len);
    if (status != HIAI_OK)
       return status;
    // Step3: 设置回调函数用于接收数据
   HIAI_PortID_t engine_id;
    engine_id.graph_id = g_graph_id;
   engine_id.engine_id = dest_engine_id;
    engine_id.port_id = 0;
   HIAI_SetDataRecvFunctor(engine_id, UserDefDataRecvCallBack);
   return HIAI_OK;
// User Application Main Example
```

```
int main()
   HIAI_StatusT ret = HIAI_OK;
   // 创建流程
   ret = HIAI_InitAndStartGraph();
   if (HIAI_OK != ret)
       return -1;
   // 读取数据,往流程灌输数据
   HIAI_PortID_t engine_id;
   engine_id.graph_id = g_graph_id;
   engine_id.engine_id = src_engine_id;
   engine_id.port_id = 0;
   int is_end_of_data = 0;
   while (!is end of data)
       UserDefDataTypeT* user_def_msg = (UserDefDataTypeT*) malloc(sizeof(UserDefDataTypeT));
       // 读取数据并填充 UserDefDataType, 此数据类型注册见数据类型注册章节
       const char * message_name = "UserDefDataType";
       size_t msg_len = strlen(message_name);
       if (HIAI_OK != HIAI_C_SendData(engine_id, message_name, msg_len, user_def_msg))
           // 队列满时返回发送数据失败,由业务逻辑处理是否重发或丢弃
       free(user_def_msg);
       user_def_msg = nullptr;
   // 处理结束时,删除整个Graph
   \label{eq:hial_DestroyGraph} \textbf{(} g\_\texttt{graph\_id) ;}
   return 0:
```

2.5.2 C++接口使用示例(根据配置文件创建 Graph)

```
* @file graph_cplusplus_api_example.cpp
* Copyright (C), 2017 - 2017, Huawei Tech. Co., Ltd. ALL RIGHTS RESERVED.
* @Source Files for HIAI Graph Orchestration
* @version 1.0
#include <unistd.h>
#include <thread>
#include <fstream>
#include <algorithm>
#include "hiaiengine/api.h"
#include "user_def_data_type.h"
* Normally the mandatory steps to setup the whole HIAI graph include:
* 1) call HIAI_Init() API to perform global system initialization;
* 2) create HIAI_CreateGraph or Graph::CreateGraph to create and start graph
* 3) set profile config if necessary
* 4) call HIAI_DestoryGraph or Graph::DestroyGraph to destroy the graph finally
```

```
***********************************
// Just define variables which are used in only this example
// and are not required in real user case.
#define MATRIX_USER_SPECIFY_ENGINE_PRIORITY (10)
static uint32_t g_graph_id = 100;
static uint32_t src_engine_id = 1000; // 定义在graph.prototxt
static uint32_t dest_engine_id = 1002; // 定义在graph.prototxt static std::string graph_config_proto_file = "./config/graph.prototxt";
// Define End
namespace hiai {
class GraphDataRecvInterface: public DataRecvInterface
public:
   GraphDataRecvInterface()
   /**
   * @ingroup hiaiengine
   * @brief 读取数据,保存
   * @param [in]输入数据
   * @return HIAI Status
   HIAI_StatusT RecvData(const std::shared_ptr<void>& message)
       // 转换成具体的消息类型,并处理相关消息, 例如:
       // shared_ptr<std::string> data =
       // std::static_pointer_cast<std::string>(message);
       return HIAI_OK;
private:
* HIAIEngine Example
* Graph:
* SrcEngine <--> Engine <--> DestEngine
HIAI StatusT HIAI InitAndStartGraph()
    // Step1: Global System Initialization before using Matrix
   HIAI_StatusT status = HIAI_Init(0);
   // Step2: Create and Start the Graph
   status = hiai::Graph::CreateGraph(graph_config_proto_file);
   if (status != HIAI_OK)
       HIAI_ENGINE_LOG(status, "Fail to start graph");
       return status;
   // Step3: 设置回调函数用于接收数据
   std::shared_ptr<hiai::Graph> graph = hiai::Graph::GetInstance(g_graph_id);
   if (nullptr == graph)
       HIAI_ENGINE_LOG("Fail to get the graph-%u", g_graph_id);
       return status;
    // Specify the port id (default to zero)
   hiai::EnginePortID target_port_config;
   target_port_config.graph_id = g_graph_id;
```

```
target_port_config.engine_id = dest_engine_id;
   target_port_config.port_id = 0;
   graph->SetDataRecvFunctor(target_port_config,
       std::shared_ptr<hiai::GraphDataRecvInterface>(
       new hiai::GraphDataRecvInterface()));
   return HIAI_OK;
// User Application Main Example
int main()
   HIAI_StatusT ret = HIAI_OK;
   // 创建流程
   ret = HIAI_InitAndStartGraph();
   if(HIAI OK != ret)
       HIAI_ENGINE_LOG("Fail to start graph");;
       return -1;
   // 读取数据,往流程灌输数据
   std::shared_ptr<hiai::Graph> graph = hiai::Graph::GetInstance(g_graph_id);
   if (nullptr == graph)
       HIAI ENGINE LOG("Fail to get the graph-%u", g graph id);
       return -1;
   hiai::EnginePortID engine_id;
   engine_id.graph_id = g_graph_id;
   engine_id.engine_id = src_engine_id;
   engine_id.port_id = 0;
   bool is_end_of_data = false;
   while (!is_end_of_data)
       std::shared_ptr<UserDefDataType> user_def_msg(new UserDefDataType);
       // 读取数据并填充 UserDefDataType, 此数据类型注册见数据类型注册章节
       // 往Graph灌输数据
       if (HIAI OK != graph->SendData(engine id, "UserDefDataType",
           \verb|std::static_pointer_cast| < \verb|void| > (user_def_msg)))|
           // 队列满时返回发送数据失败,由业务逻辑处理是否重发或丢弃
           HIAI_ENGINE_LOG("Fail to send data to the graph-%u", g_graph_id);
           break;
   // 处理结束时,删除整个Graph
   hiai::Graph::DestroyGraph(g_graph_id);
   return 0;
```

2.5.3 C++接口使用示例(根据配置文件创建 Graph, 并将生成的 Graph 回写到 list 中)

```
/**

* @file graph_cplusplus_api_example.cpp

*

* Copyright(C), 2017 - 2017, Huawei Tech. Co., Ltd. ALL RIGHTS RESERVED.

*
```

```
* @Source Files for HIAI Graph Orchestration
* @version 1.0
*/
#include <unistd.h>
#include <thread>
#include <fstream>
#include <algorithm>
#include "hiaiengine/api.h"
#include "user def data type.h"
* Normally the mandatory steps to setup the whole HIAI graph include:
* 1) call HIAI_Init() API to perform global system initialization;
* 2) create HIAI_CreateGraph or Graph::CreateGraph to create and start graph
* 3) set profile config if necessary
* 4) call HIAI_DestoryGraph or Graph::DestroyGraph to destroy the graph finally
// Just define variables which are used in only this example
// and are not required in real user case.
#define MATRIX_USER_SPECIFY_ENGINE_PRIORITY (10)
static uint32_t src_engine_id = 1000; // 定义在graph.prototxt
static uint32_t dest_engine_id = 1002; // 定义在graph.prototxt
static std::string graph_config_proto_file = "./config/graph.prototxt";
// Define End
namespace hiai {
class GraphDataRecvInterface: public DataRecvInterface
public:
   GraphDataRecvInterface()
   /**
   * @ingroup hiaiengine
   * @brief 读取数据,保存
   * @param [in]输入数据
   * @return HIAI Status
   \verb|HIAI_StatusT| RecvData(const std::shared_ptr<void>\& message)|\\
      // 转换成具体的消息类型,并处理相关消息, 例如:
      // shared_ptr<std::string> data =
      // std::static_pointer_cast<std::string>(message);
      return HIAI_OK;
private:
};
* Matrix Example
* Graph:
* SrcEngine <--> Engine <--> DestEngine
********************************
HIAI_StatusT HIAI_InitAndStartGraph(std::list<std::shared_ptr<hiai::Graph>>& graphList)
   // Step1: Global System Initialization before using Matrix
   HIAI_StatusT status = HIAI_Init(0);
   // Step2: Create and Start the Graph
   status = hiai::Graph::CreateGraph(graph_config_proto_file, graphList);
   if (status != HIAI OK)
      HIAI_ENGINE_LOG(status, "Fail to start graph");
      return status;
   // Step3: 设置回调函数用于接收数据
   // 如果在一个配置文件中配置了多个graph,可以遍历GraphList取相应graph执行期望操作
   std::list<std::shared_ptr<hiai::Graph>>::iterator iter = graphList.begin();
```

```
std::shared_ptr<hiai::Graph> graph = hiai::Graph::GetInstance((*iter)->GetGraphId());
    if (nullptr == graph)
        HIAI ENGINE LOG("Fail to get the graph");
        return status;
    // Specify the port id (default to zero)
    hiai::EnginePortID target_port_config;
    target_port_config.graph_id = (*iter)->GetGraphId();
    target_port_config.engine_id = dest_engine_id;
    target_port_config.port_id = 0;
    graph->SetDataRecvFunctor(target_port_config,
        std::shared ptr<hiai::GraphDataRecvInterface>(
        new hiai::GraphDataRecvInterface()));
    return HIAI_OK;
// User Application Main Example
int main()
    HIAI StatusT ret = HIAI OK;
    // 存储用户创建的graph
    std::list<std::shared_ptr<hiai::Graph>> graphList;
    // 创建流程
    ret = HIAI_InitAndStartGraph(graphList);
    if(HIAI OK != ret)
        HIAI_ENGINE_LOG("Fail to start graph");;
        return -1;
    // 读取数据, 往流程灌输数据
    std::list<std::shared_ptr<hiai::Graph>>::iterator iter = graphList.begin();
    std::shared_ptr<hiai::Graph> graph = hiai::Graph::GetInstance((*iter)->GetGraphId());
    if (nullptr == graph)
        \label{eq:hiai_engine_log} \verb|HIAI_ENGINE_LOG("Fail to get the graph"); \\
        return -1;
    hiai::EnginePortID engine_id;
    engine_id.graph_id = (*iter)->GetGraphId();
    engine_id.engine_id = src_engine_id;
    engine_id.port_id = 0;
    bool is_end_of_data = false;
    while (!is_end_of_data)
        std::shared_ptr<UserDefDataType> user_def_msg(new UserDefDataType);
        // 读取数据并填充 UserDefDataType, 此数据类型注册见数据类型注册章节
        // 往Graph灌输数据
        if (HIAI_OK != graph->SendData(engine_id, "UserDefDataType",
            \verb|std::static_pointer_cast| < \verb|void| > (user_def_msg)))|
            // 队列满时返回发送数据失败,由业务逻辑处理是否重发或丢弃
            \label{eq:hial_engine_log} \mbox{HIAI\_ENGINE\_LOG("Fail to send data to the graph-%u", (*iter)->GetGraphId());}
            break;
    // 处理结束时,删除所有Graph
    std::list \langle std::shared\_ptr \langle hiai::Graph \rangle \rangle ::iterator\ graphIter;
    for (graphIter = graphList.begin(); graphIter != graphList.end(); ++graphIter) {
        hiai::Graph::DestroyGraph((*graphIter)->GetGraphId());
    return 0;
```

3 Engine 实现接口(C++语言)

- 3.1 Engine的构造函数与析构函数
- 3.2 Engine::Init
- 3.3 宏:HIAI DEFINE PROCESS
- 3.4 宏:HIAI IMPL ENGINE PROCESS
- 3.5 Engine::SetDataRecvFunctor
- 3.6 Engine::SendData
- 3.7 调用示例

3.1 Engine 的构造函数与析构函数

该接口在engine.h中定义。

函数格式

Engine()

virtual ~Engine()

∭说明

该构造函数和析构函数接口是可选的,用户根据实际确定是否需要重载实现。

3.2 Engine::Init

初始化Engine实例的相关配置。该接口在engine.h中定义。

∭说明

该Init接口是可选的,用户根据实际情况决定是否需要重载实现。

函数格式

HIAI_StatusT Engine::Init(const AIConfig &config, const vector<AIModelDescription> &modelDesc)

参数说明

参数	说明	取值范围
config	Engine配置项。	-
modelDesc	模型描述。	-

返回值

返回的错误码是由用户提前注册的,具体注册方式见5.5.1 错误码注册。

错误码示例

本API(Application Programming Interface)的错误码由用户进行注册。

重载实现示例

初始化推理Engine,在初始化过程中通过模型管家(AIModelManager)的**4.1.1 AIModelManager::Init**接口加载模型。

```
HIAI_StatusT FrameworkerEngine::Init(const hiai::AIConfig& config,
   const std::vector<hiai::AIModelDescription>& model_desc)
    hiai::AIStatus ret = hiai::SUCCESS;
    // init ai model manager
    if (nullptr == ai_model_manager_)
        ai_model_manager_ = std::make_shared<hiai::AIModelManager>();
    std::cout<<"FrameworkerEngine Init"<<std::endl;</pre>
    HIAI_ENGINE_LOG("FrameworkerEngine Init");
    for (int index = 0; index < config.items_size(); ++index)</pre>
        const ::hiai::AIConfigItem& item = config.items(index);
        // loading model
        if(item.name() == "model_path")
            const char* model_path = item.value().data();
            std::vector<hiai::AIModelDescription> model_desc_vec;
            hiai::AIModelDescription model desc;
            model_desc_.set_path(model_path);
            model_desc_.set_key("");
            {\tt model\_desc\_vec.\,push\_back\,(model\_desc\_)\,;}
            ret = ai_model_manager_->Init(config, model_desc_vec);
            if (hiai::SUCCESS != ret)
                HIAI_ENGINE_LOG(this, HIAI_AI_MODEL_MANAGER_INIT_FAIL, "[DEBUG] fail to init
ai_model");
                return HIAI_AI_MODEL_MANAGER_INIT_FAIL;
    HIAI ENGINE LOG("FrameworkerEngine Init success");
    return HIAI_OK;
```

3.3 宏:HIAI_DEFINE_PROCESS

用户直接调用该宏定义Engine的输入与输出端口数。该宏在engine.h中定义。

本宏封装用到了以下函数:

HIAI_StatusT Engine::InitQueue(const uint32_t& in_port_num, const uint32_t& out_port_num);

相关宏:

在HIAI_IMPL_ENGINE_PROCESS(name, engineClass, inPortNum)之前调用此宏。

宏格式

HIAI DEFINE PROCESS (inputPortNum, outputPortNum)

参数说明

参数	说明	取值范围
inputPortNum	Engine的输入端口数。	-
outputPortNum	Engine的输出端口数。	-

调用示例

```
#define FRAMEWORK_ENGINE_INPUT_SIZE 1
#define FRAMEWORK_ENGINE_OUTPUT_SIZE 1
```

* @[in]: 定义一个输入端口, 一个输出端口*/

HIAI_DEFINE_PROCESS (FRAMEWORK_ENGINE_INPUT_SIZE, FRAMEWORK_ENGINE_OUTPUT_SIZE)

3.4 宏:HIAI_IMPL_ENGINE_PROCESS

用户需要重载实现该宏,用于定义Engine的具体实现。该宏在engine.h中定义。

本宏封装用到了以下函数:

```
static HIAIEngineFactory* GetInstance();
```

HIAI_StatusT HIAIEngineFactory::RegisterEngineCreator(const std::string&

engine_name, HIAI_ENGINE_FUNCTOR_CREATOR engineCreatorFunc);

HIAI_StatusT HIAIEngineFactory::UnRegisterEngineCreator(const std::string& engine_name);

相关宏:

在HIAI_DEFINE_PROCESS(inputPortNum, outputPortNum)之后调用本宏。

宏格式

HIAI_IMPL_ENGINE_PROCESS(name, engineClass, inPortNum)

参数说明

参数	说明	取值范围
name	Config的Engine名称。	-
engineClass	Engine的实现类名称。	-
inPortNum	输入的端口数。	-

返回值

返回的错误码由用户提前注册。

错误码示例

该API的错误码由用户进行注册。

重载实现样例

定义推理Engine的实现,在实现过程中通过模型管家(AIModelManager)的**4.1.3 AIModelManager::Process**接口执行模型推理。

```
HIAI_IMPL_ENGINE_PROCESS("FrameworkerEngine", FrameworkerEngine, FRAMEWORK_ENGINE_INPUT_SIZE)
   hiai::AIStatus ret = hiai::SUCCESS;
   HIAI StatusT hiai ret = HIAI OK;
    // receive data
   //arg0表示Engine的输入端口(编号为0),如果有多个输入端口,可依次通过arg1(编号为1的输入端
口)、arg2(编号为2的输入端口)等来对应输入端口。通过输入端口获取上一个Engine发送的数据。
    std::shared_ptr<std::string> input_arg =
       std::static_pointer_cast<std::string>(arg0);
    if (nullptr == input_arg)
       HIAI_ENGINE_LOG(this, HIAI_INVALID_INPUT_MSG, "[DEBUG] input arg is invalid");
       return HIAI INVALID INPUT MSG;
   std::cout<<"FrameworkerEngine Process"<<std::endl;</pre>
   // prapare for calling the process of ai_model_manager_
   std::vector<std::shared_ptr<hiai::IAITensor>> input_data_vec;
   uint32_t len = 75264;
   HIAI_ENGINE_LOG("FrameworkerEngine:Go to Process");
   std::cout << "HIAIAippOp::Go to process" << std::endl;</pre>
    std::shared ptr<hiai::AINeuralNetworkBuffer> neural buffer =
std::shared_ptr<hiai::AINeuralNetworkBuffer>(new hiai::AINeuralNetworkBuffer());//
std::static_pointer_cast<hiai::AINeuralNetworkBuffer>(input_data);
   neural_buffer->SetBuffer((void*)(input_arg->c_str()), (uint32_t)(len));
    std::shared_ptr<hiai::IAITensor> input_data =
std::static_pointer_cast<hiai::IAITensor>(neural_buffer);
    input_data_vec.push_back(input_data);
    // call Process and inference
   hiai::AIContext ai_context;
   std::vector<std::shared_ptr<hiai::IAITensor>> output_data_vec;
   \verb|ret = ai_model_manager_-\rangle CreateOutputTensor(input_data_vec, output_data_vec);|\\
    if (hiai::SUCCESS != ret)
       HIAI_ENGINE_LOG(this, HIAI_AI_MODEL_MANAGER_PROCESS_FAIL, "[DEBUG] fail to process
ai model");
```

```
return HIAI_AI_MODEL_MANAGER_PROCESS_FAIL;
    ret = ai_model_manager_->Process(ai_context, input_data_vec, output_data_vec, 0);
    if (hiai::SUCCESS != ret)
        HIAI_ENGINE_LOG(this, HIAI_AI_MODEL_MANAGER_PROCESS_FAIL, "[DEBUG] fail to process
ai_model");
        return HIAI_AI_MODEL_MANAGER_PROCESS_FAIL;
    \verb|std::cout| << \verb|"[DEBUG]| output_data_vec size is "<< output_data_vec.size() << std::endl; \\
    for (uint32_t index = 0; index < output_data_vec.size(); index++)</pre>
        // send data of inference to destEngine
        std::shared_ptr<hiai::AINeuralNetworkBuffer> output_data =
std::static_pointer_cast<hiai::AINeuralNetworkBuffer>(output_data_vec[index]);
        std::shared_ptr<std::string> output_string_ptr = std::shared_ptr<std::string> (new
std::string((char*)output_data->GetBuffer(), output_data->GetSize()));
        hiai_ret = SendData(0, "string", std::static_pointer_cast<void>(output_string_ptr));
        if (HIAI_OK != hiai_ret)
            HIAI_ENGINE_LOG(this, HIAI_SEND_DATA_FAIL, "fail to send data");
            return HIAI SEND DATA FAIL;
    return HIAI_OK;
```

3.5 Engine::SetDataRecvFunctor

设置Engine接收消息的回调函数。该接口在engine.h中定义。

该接口需要与DataRecvInterface::RecvData接口配合使用,详细原理如下:

- 1. 定义一个DataRecvInterface类的子类(例如DdkDataRecvInterface),初始化一个子类的对象。
- 2. 调用Engine::SetDataRecvFunctor接口将DdkDataRecvInterface类的对象设置成回调函数。
- 3. 调用Engine::SendData函数将对应的输入数据发送到对应的输出端口。
- 4. 调用DdkDataRecvInterface类的RecvData函数返回数据。

函数格式

HIAI_StatusT Engine::SetDataRecvFunctor(const uint32_t portId, const shared_ptr<DataRecvInterface>& userDefineDataRecv)

参数说明

参数	说明	取值范围
portId	端口ID。	-
userDefineDataRecv	用户自定义的数据接收回 调函数。	-

返回值

返回的部分错误码请参见"错误码示例"中的"错误码"列。

错误码示例

序号	错误码	错误码描述
1	HIAI_OK	running ok
2	HIAI_PORT_ID_ERROR	port id error

3.6 Engine::SendData

将数据从本Engine发送到指定port_id。该接口在engine.h中定义。

□说明

SendData采用DMA传送方式,可能会影响CPU对中断请求的及时响应与处理,例如调用new或者malloc分配内存。

函数格式

HIAI_StatusT Engine::SendData(uint32_t portId, const std::string& messageName, const shared_ptr<void>& dataPtr, uint32_t timeOut = TIME_OUT_VALUE);

参数说明

参数	说明	取值范围
portId	Engine的输出端口号。	-
messageName	当前发送的消息名(该消息必须已经调用HiAI提供的宏已经注册过)。	-
dataPtr	指向具体的消息指针。	-
timeOut	调用该接口发送数据时的一次超时时间,不指定timeout参数时,默认的一次超时时间为500ms。若发送数据超时,系统后台会再次尝试发送数据,最大尝试16次。	-

返回值

返回的部分错误码请参见"错误码示例"中的"错误码"列。

错误码示例

序号	错误码	错误码描述
1	HIAI_OK	running ok
2	HIAI_ENGINE_NULL_POINTER	null pointer
3	HIAI_GRAPH_ENGINE_NOT_E XIST	engine not exist
4	HIAI_GRAPH_SRC_PORT_NOT _EXIST	src port not exist

3.7 调用示例

```
* @file multi_input_output_engine_example.h
* Copyright(c) <2018>, <Huawei Technologies Co., Ltd>
* @version 1.0
* @date 2018-4-25
#ifndef MULTI INPUT OUTPUT ENGINE EXAMPLE H
#define MULTI_INPUT_OUTPUT_ENGINE_EXAMPLE_H_
#include "hiaiengine/engine.h"
#include "hiaiengine/data_type.h"
#include "hiaiengine/multitype_queue.h"
#define MULTI_INPUT_OUTPUT_ENGINE_EXAMPLE_INPUT_SIZE 3
#define MULTI_INPUT_OUTPUT_ENGINE_EXAMPLE_OUTPUT_SIZE 2
namespace hiai {
// Define New Engine
class HIAIMultiEngineExample : public Engine {
public:
   HIAIMultiEngineExample() :
       input_que_(MULTI_INPUT_OUTPUT_ENGINE_EXAMPLE_INPUT_SIZE) {}
    // 重载父类Engine的Init
   HIAI StatusT Init(const AIConfig& config,
        const std::vector<AIModelDescription>& model_desc)
        return HIAI_OK;
    /**
    * @ingroup hiaiengine
    * @brief HIAI_DEFINE_PROCESS
    * @[in]: 定义一个输入端口, 一个输出端口
   HIAI_DEFINE_PROCESS (MULTI_INPUT_OUTPUT_ENGINE_EXAMPLE_INPUT_SIZE,
MULTI_INPUT_OUTPUT_ENGINE_EXAMPLE_OUTPUT_SIZE)
private:
    // 私有实现一个成员变量,用来缓存输入队列
   hiai::MultiTypeQueue input que ;
```

```
};
#endif //MULTI INPUT OUTPUT ENGINE EXAMPLE H
/**
* @file multi_input_output_engine_example.h
* Copyright(c)<2018>, <Huawei Technologies Co., Ltd>
* @version 1.0
* @date 2018-4-25
*/
#include "multi_input_output_engine_example.h"
#include "use_def_data_type.h"
#include "use_def_errorcode.h"
namespace hiai {
/**
* @ingroup hiaiengine
* @brief HIAI_DEFINE_PROCESS: 实现多端口输入输出处理流程 * @[in]: 定义一个输入端口,一个输出端口,
         并该Engine注册,其名为"HIAIMultiEngineExample"
HIAI_IMPL_ENGINE_PROCESS("HIAIMultiEngineExample", HIAIMultiEngineExample,
MULTI_INPUT_OUTPUT_ENGINE_EXAMPLE_INPUT_SIZE)
    // This Engine has three input args and two output
    // 每个端口接收到的数据可能不是同一个时刻,因此某些输入端口可能是空的
    input_que_. PushData(0, arg0);
    input_que_. PushData(1, arg1);
    input_que_. PushData(2, arg2);
    std::shared_ptr<void> input_arg1;
    std::shared_ptr<void> input_arg2;
    std::shared_ptr<void> input_arg3;
    // 仅当三个端口都有输入数据时才继续后续处理,方式一
    if (!input_que_.PopAllData(input_arg1, input_arg2, input_arg3))
       HIAI_ENGINE_LOG(this, HIAI_INVALID_INPUT_MSG, "fail to process");
       return HIAI_INVALID_INPUT_MSG;
    // 仅当三个端口都有输入数据时才继续后续处理,方式二
    if (!(input_que_.FrontData(0, input_arg1) && input_que_.FrontData(1, input_arg2) &&
input_que_.FrontData(2, input_arg3)))
       HIAI ENGINE LOG(this, HIAI INVALID INPUT MSG, "fail to process");
       return HIAI_INVALID_INPUT_MSG;
    } else {
        input_que_.PopData(0, input_arg1);
        input\_que\_.PopData(1, input\_arg2);
         input_que_.PopData(2, input_arg3);
    // 中间业务逻辑可以直接调用 DVPP API或AIModelManger API 处理。
    // 分配内存用来保存结果
    std::shared_ptr<UseDefDataTypeT> output1 =
       std::make_shared<UseDefDataTypeT>();
```

```
std::shared_ptr<UseDefTemplateDataType<uint64_t, uint64_t, vint64_t>> output2 = std::make_shared<UseDefTemplateDataType<uint64_t, uint64_t, uint64_t>>();

// 填充输出数据结构,例如简单赋值
*output1 = *input_arg2;
*output2 = *input_arg3;

// 将输出端口发送到端口0
hiai::Engine::SendData(0, "UseDefDataTypeT", std::static_pointer_cast<void>(output1));

// 将输出端口发送到端口1
hiai::Engine::SendData(1, "UseDefTemplateDataType_uint64_t_uint64_t_uint64_t", std::static_pointer_cast<void>(output2));

return HIAI_OK;
}
```

4 模型管家接口(C++语言)

- 4.1 离线模型管家
- 4.2 AIPP配置接口
- 4.3 数据类型
- 4.4 其他用于编译依赖的接口
- 4.5 异常处理

4.1 离线模型管家

通过如下宏标识的接口,属于当前版本的未实现接口,用户不能直接调用:__ANDROID_、ANDROID、__LITE__。

4.1.1 AIModelManager::Init

初始化接口,主要完成模型加载。

函数格式

virtual AIStatus AIModelManager::Init(const AIConfig &config, const std::vector<AIModelDescription> &model_descs = {}) override;

参数	说明	取值范围
config	配置信息。	-
	关于AIConfig数据类型的 定义,请参见 4.3.2 AIConfig。	

参数	说明	取值范围
model_descs	模型描述信息列表,支持 从内存、文件加载模型, 内存加载模型优先级高于 从文件加载模型。	-
	关于AIModelDescription数 据类型的定义,请参见 4.3.6 AIModelDescription。	

SUCCESS 初始化成功/FAILED 初始化失败。

调用示例

请注意,MODEL_NAME仅支持大小写字母、数字、下划线和点,MODEL_PATH仅支持大小写字母、数字、下划线。

● 从文件加载模型:

```
AIModelManager model_mngr;
AIModelDescription model desc;
AIConfig config;
/* 校验输入文件路径是否合法
  路径部分: 支持大小写字母、数字、下划线
  文件名部分: 支持大小写字母、数字、下划线和点(.) */
model_desc.set_path(MODEL_PATH);
model_desc.set_name(MODEL_NAME);
model desc. set type(0);
vector<AIModelDescription> model_descs;
model_descs.push_back(model_desc);
// AIModelManager Init
AIStatus ret = model_mngr.Init(config, model_descs);
if (SUCCESS != ret)
   printf("AIModelManager Init failed. ret = %d\n", ret);
   return -1;
```

● 从内存加载模型:

```
AIModelManager model_mngr;

AIModelDescription model_desc;
vector<AIModelDescription> model_descs;

AIConfig config;
model_desc.set_name(MODEL_NAME);
model_desc.set_type(0);
char *model_data = nullptr;
uint32_t model_size = 0;
ASSERT_EQ(true, Utils::ReadFile(MODEL_PATH.c_str(), model_data, model_size));
model_desc.set_data(model_data, model_size);
model_desc.set_size(model_size);
AIStatus ret = model_mngr.Init(config, model_descs);
if (SUCCESS != ret)
{
    printf("AIModelManager Init failed. ret = %d\n", ret);
    return -1;
}
```

□说明

model size必须与模型实际大小保持一致。

4.1.2 AIModelManager::SetListener

设置模型管理回调函数。

川说明

如果不调用此接口或调用此接口时listener设置为nullptr,表示Process接口为同步调用,否则为异步。

函数格式

virtual AIStatus AIModelManager::SetListener(std::shared_ptr<IAIListener> listener) override;

参数说明

参数	说明	取值范围
listener	回调函数。 关于IAIListener数据类型 的定义,请参见 4.3.19 IAIListener。	-

返回值

SUCCESS初始化成功/FAILED初始化失败。

4.1.3 AIModelManager::Process

计算接口。单模型推理时,process的缓冲队列长度限制为2048。推理队列长度超过2048,将返回失败。

函数格式

virtual AIStatus AIModelManager::Process(AIContext &context, const

std::vector<std::shared_ptr<IAITensor>> &in_data,

std::vector<std::shared_ptr<IAITensor>> &out_data, uint32_t timeout);

参数	说明	取值范围
context	context运行时上下文信息,包含engine运行时的一些可变参数配置。 关于AIContext数据类型的定义,请参见4.3.17 AIContext。	-

参数	说明	取值范围
in_data	模型输入tensor列表。 关于IAITensor数据类型的 定义,请参见 4.3.20 IAITensor。	-
out_data	模型输出tensor列表。 关于IAITensor数据类型的 定义,请参见 4.3.20 IAITensor。	-
timeout	计算超时时间,预留参数,默认值为0,配置无效。	-

SUCCESS初始化成功/FAILED初始化失败。

4.1.4 AIModelManager::CreateOutputTensor

创建输出的Tensor列表。

□说明

如果用户使用该接口来创建tensor,该接口返回的内存首地址满足512对齐要求。建议用户使用AITensorFactory::CreateTensor接口来创建tensor,详见4.1.10 AITensorFactory::CreateTensor。

函数格式

 $virtual\ AIS tatus\ AIM odel Manager:: Create Output Tensor (const$

std::vector<std::shared_ptr<IAITensor>> &in_data,

std::vector<std::shared_ptr<IAITensor>> &out_data) override;

参数	说明	取值范围
in_data	输入tensor列表。 关于IAITensor数据类型的 定义,请参见 4.3.20 IAITensor。	-
out_data	输出tensor列表。 关于IAITensor数据类型的 定义,请参见 4.3.20 IAITensor。	-

SUCCESS初始化成功/FAILED初始化失败。

4.1.5 AIModelManager::CreateInputTensor

创建输入的Tensor列表。

∭说明

如果用户使用该接口来创建tensor, 该接口返回的内存首地址满足512对齐要求。建议用户使用AITensorFactory::CreateTensor接口来创建tensor, 详见4.1.10 AITensorFactory::CreateTensor。

函数格式

virtual AIStatus

AIModelManager::CreateInputTensor(std::vector<std::shared_ptr<IAITensor>> &in data);

参数说明

参数	说明	取值范围
in_data	输入tensor列表。 关于IAITensor数据类型的 定义,请参见 4.3.20 IAITensor。	-

返回值

SUCCESS初始化成功/FAILED初始化失败。

4.1.6 AIModelManager::IsPreAllocateOutputMem

是否可以预分配输出内存。

相关函数:

AIStatus AIModelManager::CreateOutputTensor(const std::vector<std::shared_ptr<IAITensor>> &in_data, std::vector<std::shared_ptr<IAITensor>> &out_data);

函数格式

virtual bool AIModelManager::IsPreAllocateOutputMem() override;

参数说明

无。

返回值

● True: 可调用CreateOutputTensor申请输出内存申请。

● False: 不可调用CreateOutputTensor。

∭说明

只有模型管家加载一个模型时才返回true。

4.1.7 AIModelManager::GetModelIOTensorDim

获取已加载模型的输入输出尺寸。

函数格式

virtual AIStatus AIModelManager::GetModelIOTensorDim(const std::string&model_name, std::vector<TensorDimension>& input_tensor, std::vector<TensorDimension>& output_tensor);

参数说明

参数	说明	取值范围
model_name	模型名称。	1
input_tensor	模型输入尺寸列表。 关于TensorDimension数据 类型的定义,请参见 4.3.18 TensorDimension。	
output_tensor	模型输出尺寸列表。 关于TensorDimension数据 类型的定义,请参见 4.3.18 TensorDimension。	-

返回值

SUCCESS初始化成功/FAILED初始化失败。

示例

例如获取resnet50模型的输入输出Tensor描述如下。

若output_tensor的size不为0,则获取到的output_tensor数据会追加在原output_tensor数据之后。

```
input_tensor
name = "data"
                       #输入层的name
                       #预留数据类型,暂时不用
data_type = 0
size = 20
                       #内存大小,单位字节
                       #预留Tensor排布格式,暂时不用
format = 0
dims = {1, 3, 224, 224}
output_tensor
name = "output_0_prob_0"
                       #输出Tensor的name,格式为: output_{数字}_{输出节点name}_{输出节点输出
索引}
                        #预留数据类型,暂时不用
data_type = 0
                       #内存大小,单位字节
size = 20
```

4.1.8 AIModelManager::GetMaxUsedMemory

根据模型名字查询模型使用内存大小。

函数格式

int32_t AIModelManager::GetMaxUsedMemory(std::string model_name);

参数说明

参数	说明	取值范围
model_name	模型名称。	-

返回值

模型使用内存大小。

4.1.9 AISimpleTensor::SetBuffer

设置tensor数据地址。

函数格式

void SetBuffer(void *data, const int32_t size, bool isown=false);

参数	说明	取值范围
data	数据地址。	-
size	数据长度。	-
	说明 单位为字节。size必须与数据实际大小一致。	
isown	是否在tensor生命周期结束后,由tensor释 放data地址的内存。	-
	● false:默认值,data地址的内存在 tensor生命周期结束后,由用户进行 free操作。	
	● true: 在tensor生命周期结束后tensor释 放该内存,用户不能进行free操作,否 则导致重复释放。	

无。

4.1.10 AITensorFactory::CreateTensor

创建模型tensor。该接口在ai_tensor.h中定义。

函数格式

std::shared_ptr<IAITensor> CreateTensor(const AITensorDescription &tensor_desc, void *buffer, int32 t size);

std::shared_ptr<IAITensor> CreateTensor(const AITensorDescription &tensor_desc);
std::shared_ptr<IAITensor> CreateTensor(const std::string &type);

参数说明

参数	说明	取值范围
tensor_desc	tensor描述。 关于AITensorDescription数据类型的定义,请参见 4.3.4 AITensorDescription 。	-
buffer	数据地址。 说明 • 建议输入数据及输出数据都通过 HIAI_DMalloc接口申请,这样就能够使能算法推理的零拷贝机制,优化Process时间。 • 由客户申请内存地址,使用结束后,由客户释放内存地址。	-
size	数据长度。 说明 单位为字节, size必须与数据实际大小一致。	-
type	注册Tensor时的类型。 说明: 该接口不分配内存,在非预分配内存情况 下使用。请参见 4.3.4 AITensorDescription 中type描述。	-

返回值

创建模型tensor成功,则返回tensor指针。创建模型tensor失败,则返回空指针。

4.1.11 调用示例

示例1同步调用示例

当前示例是单模型情况下推理Engine的实现代码。

在多模型情况下,如果需要参考该示例,您需要注意以下几点:

- 声明preOutBuffer变量时,不能带有static关键字,您需要按如下方式定义: bool preOutBuffer = false;

示例代码如下:

```
// 推理Engine Process函数实现
HIAI_IMPL_ENGINE_PROCESS("ClassifyNetEngine", ClassifyNetEngine, CLASSIFYNET_ENGINE_INPUT_SIZE)
   HIAI_ENGINE_LOG(this, HIAI_OK, "ClassifyNetEngine Process");
   HIAI StatusT ret = HIAI OK;
   static bool preOutBuffer = false;
   std::vector<std::shared_ptr<hiai::IAITensor>> inDataVec;
   // 获取从上个Engine传入的数据
   std::shared ptr<EngineTransNewT> input arg =
       std::static_pointer_cast<EngineTransNewT>(arg0);
    // 如果传入数据为空指针,直接返回
   if (nullptr == input_arg)
       HIAI_ENGINE_LOG(this, HIAI_INVALID_INPUT_MSG,
           "fail to process invalid message");
       return HIAI INVALID INPUT MSG;
    // 准备输出数据,输出数据使用HIAI_DMalloc接口分配内存, 并通过CreateTensor给到算法推理,如果使
用同步的机制
   // 调用推理的Process,则只需要分配一次输出内存
   if (preOutBuffer == false) {
       std::vector<hiai::TensorDimension> inputTensorVec;
       std::vector<hiai::TensorDimension> outputTensorVec;
       ret = ai_model_manager_->GetModelIOTensorDim(modelName, inputTensorVec, outputTensorVec);
       if (ret != hiai::SUCCESS)
           HIAI ENGINE LOG(this, HIAI AI MODEL MANAGER INIT FAIL,
               "hiai ai model manager init fail");
           return HIAI_AI_MODEL_MANAGER_INIT_FAIL;
        // allocate OutData in advance
       HIAI StatusT hiai ret = HIAI OK:
       for (uint32_t index = 0; index < outputTensorVec.size(); index++) {</pre>
           hiai::AITensorDescription outputTensorDesc =
hiai::AINeuralNetworkBuffer::GetDescription();
           uint8_t* buffer = nullptr;
           // HIAI_Dmalloc分配内存,该内存主要是给算法进行推理,需要调用HIAI_DFree释放内存
           // Engine析构时释放
           hiai ret = hiai::HIAIMemory::HIAI DMalloc(outputTensorVec[index].size, (void*&)buffer,
1000);
           if (hiai ret != HIAI OK || buffer == nullptr)
               std::cout<<"HIAI_DMalloc failed"<< std::endl;
               continue:
           outData_.push_back(buffer);
           shared ptr<hiai::IAITensor> outputTensor =
               hiai::AITensorFactory::GetInstance()->CreateTensor(outputTensorDesc, buffer,
outputTensorVec[index].size);
```

```
outDataVec_.push_back(outputTensor);
       preOutBuffer = true;
    // Transfer buffer to Framework directly, only one inputsize
   hiai::AITensorDescription inputTensorDesc =
       hiai::AINeuralNetworkBuffer::GetDescription();
    shared_ptr<hiai::IAITensor> inputTensor =
       hiai::AITensorFactory::GetInstance()->CreateTensor(inputTensorDesc,
        input_arg->trans_buff.get(), input_arg->buffer_size);
    // AIModelManager. fill in the input data.
    inDataVec.push_back(inputTensor);
   hiai::AIContext ai_context;
    // Process work
    ret = ai_model_manager_->Process(ai_context,
       inDataVec, outDataVec_, 0);
    if (hiai::SUCCESS != ret)
       HIAI_ENGINE_LOG(this, HIAI_AI_MODEL_MANAGER_PROCESS_FAIL,
            "Fail to process ai model manager");
       return HIAI_AI_MODEL_MANAGER_PROCESS_FAIL;
    // Convert the generated data to the buffer of the string type and send the data.
   for (uint32_t index = 0; index < outDataVec_.size(); index++)</pre>
       HIAI_ENGINE_LOG(this, HIAI_OK, "ClassifyNetEngine SendData");
       std::shared_ptr<hiai::AINeuralNetworkBuffer> output data =
std::static_pointer_cast<hiai::AINeuralNetworkBuffer>(outDataVec_[index]);
       std::shared_ptr<std::string> output_string_ptr =
            std::shared_ptr\std::string\(new std::string((char*)output_data-\)GetBuffer(),
output_data->GetSize()));
       hiai::Engine::SendData(0, "string",
            std::static_pointer_cast(void)(output_string_ptr));
   inDataVec.clear();
   return HIAI_OK;
ClassifyNetEngine::~ClassifyNetEngine() {
    // 释放outData预分配内存
   HIAI_StatusT ret = HIAI_OK;
    for (auto buffer : outData_) {
       if (buffer != nullptr) {
            ret = hiai::HIAIMemory::HIAI_DFree(buffer);
           buffer = nullptr;
```

示例 2 异步调用示例

请参见DDK样例中的"ddk安装目录/ddk/sample/customop/customop app/main.cpp"。

4.2 AIPP 配置接口

4.2.1 简介

AIPP(AI Preprocessing)用于在AI Core上完成图像预处理,包括改变图像尺寸(Crop 裁剪或Padding补边)、色域转换(转换图像格式)、减均值/乘系数(改变图像像素)。

对于动态AIPP,模型转换时仅设置AIPP模式为动态,每次模型推理前需要在推理Engine的代码中设置动态AIPP参数值,然后在模型推理时可使用不同的AIPP参数。在推理Engine的代码中设置动态AIPP参数值,就需要调用本章提供的AIPP配置相关接口,这部分接口在ai_tensor.h中定义。

4.2.2 SetDynamicInputIndex

函数格式

void SetDynamicInputIndex(uint32_t dynamicInputIndex = 0);

功能说明

指定对于模型的第几个原始输入做AIPP处理。

参数说明

参数名	输入/ 输出	类型	描述
dynamicInputIndex	输入	uint32_t	模型原始输入的下标,从0开始,默认为0。 例如模型有两个输入,需要对第2个 输入做AIPP,则将dynamicInputIndex 配置为1。

返回值

无。

异常处理

无。

约束说明

无。

4.2.3 SetDynamicInputEdgeIndex

函数格式

void SetDynamicInputEdgeIndex(uint32 t dynamicInputEdgeIndex = 0);

功能说明

如果一个模型输入为多个算子共有,即Data算子后面跟着多个算子,通过调用该接口,对Data算子的不同的输出边做不同的AIPP处理。

参数说明

参数名	输入/ 输出	类型	描述
dynamicInputEdgeInde x	输入	uint32_t	某个模型输入对应的输出边的下标,从0开始,默认为0。

返回值

无。

异常处理

无。

约束说明

无。

4.2.4 SetInputFormat

函数格式

AIStatus SetInputFormat(AippInputFormat inputFormat);

功能说明

用于设置模型的原始输入的类型。

参数名	输入/ 输出	类型	描述
inputFormat	输入	AippInputFormat	模型原始输入类型。
			enum AippInputFormat
			{
			YUV420SP_U8 = 1,
			XRGB8888_U8,
			RGB888_U8,
			YUV400_U8,
			RESERVED
			};

参数名	类型	描述
-	AIStatus	设置成功返回0,如果输入非法,返回其它 值。
		AIStatus类型的定义如下:
		AIStatus = uint32_t

异常处理

无。

约束说明

无。

4.2.5 SetCscParams(设置默认参数值)

函数格式

AIStatus SetCscParams(AippInputFormat srcFormat,

AippModelFormat dstFormat,

ImageFormat imageFormat = BT_601NARROW);

功能说明

用户可以调用此接口,实现色域转换功能,根据入参处指定的原始输入类型、目标输入类型及图片类型,自动生成AIPP中CSC(color space convert)色域转换参数的一组默认值,同时根据原始输入类型、目标输入类型打开或关闭RB/UV通道交换开关。

参数名	输入/ 输出	类型	描述
srcFormat	输入	AippInputFormat	模型的原始输入类型。
			enum AippInputFormat
			{
			YUV420SP_U8 = 1,
			XRGB8888_U8,
			RGB888_U8,
			YUV400_U8,
			RESERVED
			};

参数名	输入/ 输出	类型	描述
dstFormat	输入	AippModelFormat	AIPP转换之后的目标输入类型。 enum AippModelFormat { MODEL_RGB888_U8 = 1, MODEL_BGR888_U8,
			MODEL_GRAY, MODEL_YUV444SP_U8, MODEL_YVU444SP_U8 };
imageFormat	输入	ImageFormat	图片类型,此接口之处的图片类型,当前仅支持配置为JPEG和BT_601NARROW。enum ImageFormat { BITMAP, PNG, JPEG, BT_601NARROW };

参数名	类型	描述(参数说明、取值范围等)
-	AIStatus	设置成功返回0,如果输入非法,返回其它 值。
		AIStatus类型的定义如下:
		AIStatus = uint32_t

异常处理

无。

约束说明

此接口支持对CSC色域转换相关参数进行快捷的配置,系统内置了一组默认的CSC配置参数,请参见《Atlas 500 模型转换指导》中的"AIPP配置"。

支持的AIPP转换前和转换后的图片输入格式如下:

AippInputFormat::YUV420SP_U8 到 AippModelFormat::MODEL_YVU444SP_U8

AippInputFormat::YUV420SP_U8 到 AippModelFormat::MODEL_RGB888_U8
AippInputFormat::YUV420SP_U8 到 AippModelFormat::MODEL_BGR888_U8
AippInputFormat::YUV420SP_U8 到 AippModelFormat::MODEL_GRAY
AippInputFormat::XRGB8888_U8 到 AippModelFormat::MODEL_YUV444SP_U8
AippInputFormat::XRGB8888_U8 到 AippModelFormat::MODEL_YVU444SP_U8
AippInputFormat::XRGB8888_U8 到 AippModelFormat::MODEL_GRAY
AippInputFormat::RGB888_U8 到 AippModelFormat::MODEL_BGR888_U8
AippInputFormat::RGB888_U8 到 AippModelFormat::MODEL_YUV444SP_U8
AippInputFormat::RGB888_U8 到 AippModelFormat::MODEL_YUV444SP_U8
AippInputFormat::RGB888_U8 到 AippModelFormat::MODEL_YVU444SP_U8
AippInputFormat::RGB888_U8 到 AippModelFormat::MODEL_GRAY
如果支持的图片类型,或者图片处理格式无法满足需求,请调用另一个4.2.5
SetCscParams(设置默认参数值)接口设置CSC色域转换相关参数。

4.2.6 SetCscParams(根据需求设置参数值)

函数格式

```
void SetCscParams(bool csc_switch = false,
int16 t cscMatrixR0C0 = 0,
int16_t cscMatrixR0C1 = 0,
int16 t cscMatrixR0C2 = 0,
int16_t cscMatrixR1C0 = 0,
int16 t cscMatrixR1C1 = 0,
int16 t cscMatrixR1C2 = 0,
int16 t cscMatrixR2C0 = 0,
int16 t cscMatrixR2C1 = 0,
int16 t cscMatrixR2C2 = 0,
uint8 t cscOutputBiasR0 = 0,
uint8 t cscOutputBiasR1 = 0,
uint8_t cscOutputBiasR2 = 0,
uint8 t cscInputBiasR0 = 0,
uint8_t cscInputBiasR1 = 0,
uint8 t cscInputBiasR2 = 0);
```

功能说明

用户可以调用此接口,根据需求对AIPP中CSC色域转换相关参数进行灵活的配置。

参数说明

参数名	输入/ 输出	类型	描述
csc_switch	输入	bool	色域转换开关,默认false。
cscMatrixR0C0	输入	int16_t	色域转换矩阵参数。
cscMatrixR0C1	输入	int16_t	色域转换矩阵参数。
cscMatrixR0C2	输入	int16_t	色域转换矩阵参数。
cscMatrixR1C0	输入	int16_t	色域转换矩阵参数。
cscMatrixR1C1	输入	int16_t	色域转换矩阵参数。
cscMatrixR1C2	输入	int16_t	色域转换矩阵参数。
cscMatrixR2C0	输入	int16_t	色域转换矩阵参数。
cscMatrixR2C1	输入	int16_t	色域转换矩阵参数。
cscMatrixR2C2	输入	int16_t	色域转换矩阵参数。
cscOutputBiasR0	输入	uint8_t	RGB转YUV时的输出偏移,默认值为0,可以只配置部分配置项。
cscOutputBiasR1	输入	uint8_t	RGB转YUV时的输出偏移,默认值为0,可以只配置部分配置项。
cscOutputBiasR2	输入	uint8_t	RGB转YUV时的输出偏移,默认值为0,可以只配置部分配置项。
cscInputBiasR0	输入	uint8_t	YUV转RGB时的输入偏移,默认值为0,可以只配置部分配置项。
cscInputBiasR1	输入	uint8_t	YUV转RGB时的输入偏移,默认值为0,可以只配置部分配置项。
cscInputBiasR2	输入	uint8_t	YUV转RGB时的输入偏移,默认值为0,可以只配置部分配置项。

返回值

无。

异常处理

无。

约束说明

4.2.7 SetRbuvSwapSwitch

函数格式

void SetRbuvSwapSwitch(bool rbuvSwapSwitch = false);

功能说明

设置CSC色域转换前,R通道与B通道交换开关,或者U通道与V通道交换开关。

参数说明

参数名	输入/ 输出	类型	描述
rbuvSwapSwitch	输入	bool	true表示支持交换,false表示不支持。

返回值

无。

异常处理

无。

约束说明

无。

4.2.8 SetAxSwapSwitch

函数格式

void SetAxSwapSwitch(bool axSwapSwitch = false);

功能说明

设置CSC色域转换前,GBA->ARGB或者YUVA->AYUV交换开关。

参数名	输入/ 输出	类型	描述
axSwapSwitch	输入	bool	true表示支持交换,false表示不支持。

无。

异常处理

无。

约束说明

无。

4.2.9 SetSrcImageSize

函数格式

void SetSrcImageSize(int32_t srcImageSizeW = 0, int32_t srcImageSizeH = 0);

功能说明

设置AIPP处理前,原始图片的宽和高信息。

参数说明

参数名	输入/ 输出	类型	描述
srcImageSizeW	输入	int32_t	原始图片的宽。
srcImageSizeH	输入	int32_t	原始图片的高。

返回值

无。

异常处理

无。

约束说明

无。

4.2.10 SetCropParams

函数格式

AIStatus SetCropParams(bool cropSwitch, int32_t cropStartPosW, int32_t cropStartPosH, int32_t cropSizeW, int32_t cropSizeH,

uint32_t batch_index = 0);

功能说明

设置Crop裁剪参数信息。动态AIPP支持为每个batch配置不同的Crop参数,batchIndex表示对第几个batch设置Crop参数,batchIndex取值范围[0, batchNum),如果超出这个范围,调用接口后会返回失败。

参数说明

参数名	输入/ 输出	类型	描述
cropSwitch	输入	Bool	true表示支持裁剪,false表示不支持。
cropStartPosW	输入	int32_t	裁剪时,坐标点起始位置在图中横向的坐 标。
cropStartPosH	输入	int32_t	裁剪时,坐标点起始位置在图中纵向的坐标。
cropSizeW	输入	int32_t	裁剪的宽度。
cropSizeH	输入	int32_t	裁剪的高度。
batchIndex	输入	uint32_t	对第几个batch做裁剪,默认为0,取值范围 [0, batchNum)。

返回值

参数名	类型	描述(参数说明、取值范围等)
-	AIStatus	设置成功返回0,如果输入非法,返回其它 值。
		AIStatus类型的定义如下:
		AIStatus = uint32_t

异常处理

无。

约束说明

无。

4.2.11 SetPaddingParams

函数格式

AIStatus SetPaddingParams(int8_t paddingSwitch,

int32_t paddingSizeTop, int32_t paddingSizeBottom,
int32_t paddingSizeLeft, int32_t paddingSizeRight,
uint32_t batch_index = 0);

功能说明

设置Padding补边参数。paddingSizeTop、paddingSizeBottom,paddingSizeLeft与paddingSizeRight四个参数分别表示在图片的上下左右填充的值。

参数说明

参数名	输入/ 输出	类型	描述
paddingSwitch	输入	Bool	true表示支持padding,false表示不支持。
paddingSizeTop	输入	int32_t	在图片上方填充的值。
paddingSizeBotto m	输入	int32_t	在图片下方填充的值。
paddingSizeLeft	输入	int32_t	在图片左方填充的值。
paddingSizeRight	输入	int32_t	在图片右方填充的值。
batchIndex	输入	uint32_t	对第几个batch做补边,默认为0,取值范围 [0, batchNum)。

返回值

参数名	类型	描述(参数说明、取值范围等)
-	AIStatus	设置成功返回0,如果输入非法,返回其它 值。
		AIStatus类型的定义如下:
		AIStatus = uint32_t

异常处理

无。

约束说明

4.2.12 SetDtcPixelMean

函数格式

AIStatus SetDtcPixelMean(int16_t dtcPixelMeanChn0 = 0,

int16_t dtcPixelMeanChn1 = 0,

int16_t dtcPixelMeanChn2 = 0,

int16_t dtcPixelMeanChn3 = 0,

uint32_t batch_index = 0);

功能说明

设置DTC通道均值参数。

参数说明

参数名	输入/ 输出	类型	描述
dtcPixelMeanChn	输入	int16_t	通道0均值。
dtcPixelMeanChn	输入	int16_t	通道1均值。
dtcPixelMeanChn 2	输入	int16_t	通道2均值。
dtcPixelMeanChn 3	输入	int16_t	通道3均值。
batchIndex	输入	uint32_t	对第几个batch设置DTC通道均值,默认为 0,取值范围[0, batchNum)。

返回值

参数名	类型	描述(参数说明、取值范围等)
-	AIStatus	设置成功返回0,如果输入非法,返回其它 值。
		AIStatus类型的定义如下:
		AIStatus = uint32_t

异常处理

约束说明

无。

4.2.13 SetDtcPixelMin

函数格式

AIStatus SetDtcPixelMin(float dtcPixelMinChn0 = 0,

float dtcPixelMinChn1 = 0,

float dtcPixelMinChn2 = 0,

float dtcPixelMinChn3 = 0,

uint32_t batch_index = 0);

功能说明

设置DTC通道最小值参数。

参数说明

参数名	输入/ 输出	类型	描述
dtcPixelMinChn0	输入	float	DTC通道0最小值。
dtcPixelMinChn1	输入	float	DTC通道1最小值。
dtcPixelMinChn2	输入	float	DTC通道2最小值。
dtcPixelMinChn3	输入	float	DTC通道3最小值。
batchIndex	输入	uint32_t	对第几个batch设置DTC通道最小值,默认为0,取值范围[0, batchNum)。

返回值

参数名	类型	描述(参数说明、取值范围等)
-	AIStatus	设置成功返回0,如果输入非法,返回其它 值。
		AIStatus类型的定义如下:
		AIStatus = uint32_t

异常处理

约束说明

无。

4.2.14 SetPixelVarReci

函数格式

AIStatus SetPixelVarReci(float dtcPixelVarReciChn0 = 0,

float dtcPixelVarReciChn1 = 0,

float dtcPixelVarReciChn2 = 0,

float dtcPixelVarReciChn3 = 0,

uint32_t batch_index = 0);

功能说明

设置DTC通道方差或(max-min)的倒数。

参数说明

参数名	输入/ 输出	类型	描述
dtcPixelVarReciChn0	输入	float	DTC通道0方差。
dtcPixelVarReciChn1	输入	float	DTC通道1方差。
dtcPixelVarReciChn2	输入	float	DTC通道2方差。
dtcPixelVarReciChn3	输入	float	DTC通道3方差。
batchIndex	输入	uint32_t	对第几个batch设置DTC通道方差,默认为0,取值范围[0, batchNum)。

返回值

参数名	类型	描述
-	AIStatus	设置成功返回0,如果输入非法,返回其它 值。
		AIStatus类型的定义如下:
		AIStatus = uint32_t

异常处理

约束说明

无。

4.2.15 SetInputDynamicAIPP

函数格式

AIStatus SetInputDynamicAIPP(std::vector<std::shared_ptr<IAITensor>>& inData, std::shared_ptr<AippDynamicParaTensor> aippParms);

功能说明

将生成的动态AIPP参数AippDynamicParaTensor插入到输入Tensor列表中。插入的位置由AIPP Tensor中的dynamicInputIndex和dynamicInputEdgeIndex两个属性决定,这两个属性由前面提到的4.2.2 SetDynamicInputIndex和4.2.15 SetInputDynamicAIPP两个接口设置。

AIPP Tensor将会被插入到原始输入的Tensor之后。

如果需要插入多个AippDynamicParaTensor,需要多次调用此接口,此时如果和Tensor列表中已有的AippDynamicParaTensor相比,新插入的AippDynamicParaTensor的dynamicInputIndex较小,或者dynamicInputIndex相同但dynamicInputEdgeIndex较小,则会被插入在已有的AippDynamicParaTensor前;如果dynamicInputIndex和dynamicInputEdgeIndex与列表中已有的AippDynamicParaTensor相同,则新插入的AippDynamicParaTensor覆盖已有的AippDynamicParaTensor。

注意,调用此接口时,并不会校验dynamicInputIndex和dynamicInputEdgeIndex是否合法,即dynamicInputIndex配置大于了模型的输出个数,或者dynamicInputEdgeIndex大于了对应输入的输出边的个数,接口不会返回错误。

参数名	输入/输 出	类型	描述
inData	输入	std::vector <std::shar ed_ptr<iaitensor>> &</iaitensor></std::shar 	输入Tensor列表
aippParms	输入	std::shared_ptr <aipp DynamicParaTensor ></aipp 	动态AIPP Tensor

参数名	类型	描述
-	AIStatus	设置成功返回0,如果输入非法,返回其它 值。
		AIStatus类型的定义如下:
		AIStatus = uint32_t

异常处理

无。

约束说明

无。

4.2.16 GetDynamicInputIndex

函数格式

uint32_t GetDynamicInputIndex() const;

功能说明

获取SetDynamicInputIndex设置的模型原始输入下标。

参数说明

无。

返回值

参数名	类型	描述
-	uint32_t	模型原始输入下标。

异常处理

无。

约束说明

4.2.17 GetDynamicInputEdgeIndex

函数格式

uint32_t GetDynamicInputEdgeIndex() const;

功能说明

获取SetDynamicInputEdgeIndex设置的模型输入对应的输出边的下标。

参数说明

无。

返回值

参数名	类型	描述
-	uint32_t	模型输入对应的输出边的下标。

异常处理

无。

约束说明

无。

4.2.18 调用示例

```
* @file mngr_sample.h
* Copyright(c) <2018>, <Huawei Technologies Co., Ltd>
* @version 1.0
* @date 2018-4-25
#include <stdio.h>
#include <string>
#include <vector>
#include <iostream>
#include <fstream>
#include <assert.h>
#include "hiaiengine/ai_model_manager.h"
using namespace std;
using namespace hiai;
// 图片数据路径
static const std::string IMAGE_FILE_PATH = "/data/input_zebra.bin";
static const char* MODEL_PATH = "/data/ResNet.davincimodel"; static const char* MODEL_NAME = "resnet18";
*** brief: 读取输入数据
char* ReadBinFile(const char *file_name, uint32_t *fileSize)
```

```
std::filebuf *pbuf;
   std::ifstream filestr;
   size_t size;
   filestr.open(file_name, std::ios::binary);
   if (!filestr)
       return nullptr;
   pbuf = filestr.rdbuf();
   size = pbuf->pubseekoff(0, std::ios::end, std::ios::in);
   pbuf->pubseekpos(0, std::ios::in);
   char * buffer = (char*)malloc(size);
   if (nullptr == buffer)
       return nullptr;
   pbuf->sgetn(buffer, size);
   *fileSize = size;
   filestr.close();
   return buffer;
int main(int argc, char* argv[])
   vector<shared_ptr<IAITensor>> model_input;
   vector<shared_ptr<IAITensor>> model_output;
   AIModelManager model_mngr;
   AIModelDescription model_desc;
   AIConfig config;
   AIContext context;
   {\tt model\_desc.\,set\_path\,(MODEL\_PATH)}\;;
   model_desc.set_name(MODEL_NAME);
   model_desc.set_type(0);
   vector<AIModelDescription> model_descs;
   model_descs.push_back(model_desc);
   // AIModelManager Init
   AIStatus ret = model_mngr.Init(config, model_descs);
   if (SUCCESS != ret)
       printf("AIModelManager Init failed. ret = %d\n", ret);
       return -1;
   // 输入tensor
   // 输入tensor会在读取图片数据后重新设置,这里的作用只是初始化
   AITensorDescription tensor_desc = AINeuralNetworkBuffer::GetDescription();
   shared_ptr<IAITensor> input_tensor = AITensorFactory::GetInstance()->CreateTensor(tensor_desc);
   if (nullptr == input tensor)
       printf("Create input_tensor failed.\n");
       return -1;
   // 读取图片数据
   uint32_t image_data_size = 0;
   float* image_data = (float*)ReadBinFile(IMAGE_FILE_PATH.c_str(), &image_data_size);
   if (nullptr == image_data)
       printf("ReadBinFile failed bin file path= %s \n", IMAGE_FILE_PATH.c_str());
       return -1;
   // 将图片数据地址指针及长度,设置给input_simple_tensor
   shared_ptr<AISimpleTensor> input_simple_tensor =
static_pointer_cast<AISimpleTensor>(input_tensor);
   input_simple_tensor->SetBuffer((void*)image_data, image_data_size);
   model_input.push_back(input_tensor);
   // 创建输出tensor
   if(model_mngr.IsPreAllocateOutputMem())
       ret = model_mngr.CreateOutputTensor(model_input, model_output);
       if (SUCCESS != ret)
```

```
printf("CreateOutputTensor failed.ret = %d\n", ret);
           delete image_data;
           return -1;
   else
       // 用户创建tensor
       ret = model_mngr.GetModelIOTensorDim(MODEL_NAME, input_tensor_dims, output_tensor_dims);
       std::vector<TensorDimension> input_tensor_dims;
       std::vector<TensorDimension> output_tensor_dims;
       for(TensorDimension & dims : output_tensor_dims)
           shared_ptr<IAITensor> output_tensor = AITensorFactory::GetInstance()-
>CreateTensor(tensor_desc);
           shared_ptr<AISimpleTensor> output_simple_tensor =
static_pointer_cast<AISimpleTensor>(output_tensor);
           output_simple_tensor->setBuffer((void*) new char[dims.size], dims.size);
           model_output.push_back(output_tensor)
   }
   bool dynamic_aipp_flag = true;
   if (dynamic_aipp_flag)
       const int batch_number = 2;
       AITensorDescription desc =
AippDynamicParaTensor::GetDescription(std::to_string(batch_number));
       shared_ptr<IAITensor> tensor = AITensorFactory::GetInstance()->CreateTensor(desc);
       shared_ptr<AippDynamicParaTensor> aipp_params_tensor =
static_pointer_cast<AippDynamicParaTensor>
       //设置模型的原始输入的类型。
       aipp_params_tensor->SetInputFormat(hiai::YUV420SP_U8);
       //设置色域矩阵参数值
       aipp_params_tensor=>SetCscParams(hiai::YUV420SP_U8, hiai::MODEL_BGR888_U8, hiai::JPEG);
       //设置原始图片的宽和高信息
       aipp_params_tensor->SetSrcImageSize(256, 224);
       //如果有多batch的情况下,为每个batch设置AIPP参数。
       for (int i = 0; i < batch_number; i++) {
           //设置抠图参数值
           aipp_params_tensor->SetCropParams(true, 0, 0, 200, 200);
           //设置Padding参数值
           aipp_params_tensor->SetPaddingParams(true, 12, 12, 12, 12);
           //设置DTC通道均值参数
           aipp_params_tensor->SetDtcPixelMean(104, 117, 123, 0, i);
           //设置DTC通道方差或 (max-min) 的倒数
           \verb|aipp_params_tensor-> SetPixelVarReci (1.0, 1.0, 1.0, 1.0, i); \\
   model_mngr.SetInputDynamicAIPP(model_input, aipp_params_tensor);
   // 启动模型推理
   printf("Start process.\n");
   ret = model_mngr.Process(context, model_input, model_output, 0);
   if (SUCCESS != ret)
       printf("Process failed.ret = %d\n", ret);
       return -1;
   // 因未设置监听器,所以模型推理为同步调用,可直接取出模型推理输出数据
   shared ptr<AISimpleTensor> result tensor =
static pointer cast<AISimpleTensor>(model output[0]);
   printf("Get Result, buffsize is %d", result_tensor=>GetSize());
   for(TensorDimension & dims : output_tensor_dims)
   printf("predict ok.\n");
```

```
return 0;
}
```

4.3 数据类型

4.3.1 AIConfigItem

AIConfig中配置项描述,详细实现参考ai_type.proto。

```
message AIConfigItem
{
    string name = 1;  // 配置项名称
    string value = 2;  // 配置项值
    repeated AIConfigItem sub_items = 3;  // 配置子项
};
```

4.3.2 AIConfig

调用模型管家Init接口时作为入参,详细实现参考ai types.proto。

```
message AIConfig
{
    repeated AIConfigItem items = 1; // 配置项列表
};
```

4.3.3 AITensorParaDescription

tensor参数描述,详细实现参考ai_types.proto。

4.3.4 AITensorDescription

tensor描述,主要用于描述模型的输入输出信息,详细实现参考ai types.proto。

```
// Tensor描述
message AITensorDescription
{
   string name = 1;  // Tensor名称
   string type = 2;  // Tensor类型
   repeated string compatible_type = 3;  // 指定可以兼容的所有父类类型
   repeated AITensorParaDescription paras = 4;  // 参数列表
};
```

4.3.5 AITensorDescriptionList

tensor描述列表,主要用于描述模型的输入输出信息列表,详细实现参考ai_types.proto。

```
message AITensorDescriptionList
{
    repeated AITensorDescription tensor_descs = 1; // tensor列表
}
```

4.3.6 AIModelDescription

调用Init接口时作为输入,用于描述模型,详细实现参考ai types.proto。

```
messAIPPDynaage AIModelDescription
   string name = 1; // 模型名称,支持大小写字母、数字、下划线和点(.)
   int32 type = 2; // 模型类型, 当前仅支持 DAVINCI_OFFLINE_MODEL类型, 值为0
                 // 模型管家已新增模型解析能力,因此该字段不设置亦无碍,为了保持向前兼容,特此
保留--2018/11/24
   string version = 3; // 模型版本
   int32 size = 4; // 模型大小
string path = 5; // 模型路径,支持大小写字母、数字、下划线
   repeated string sub_path = 6; // 辅助模型路径,用于模型文件为多个的情况,比如caffe在线模型
                           // 为了向前兼容,我们没有对path字段做修改(修改为repeated),而是
新增了sub_path字段
   string key = 7; // 模型秘钥
   repeated string sub_key = 8; // 辅助模型秘钥,用于模型秘钥为多个的情况,比如caffe在线模型
                           // 为了向前兼容,我们没有对key字段做修改(修改为repeated),而是新
增了sub key字段
   enum Frequency
      UNSET
            =0;
            =1;
      MEDIUM =2:
      HIGH
            =3;
   Frequency frequency = 9;
   enum DeviceType
      NPU = 0;
      IPU = 1;
      MLU = 2;
      CPU = 3;
      NONE = 255;
   DeviceType device_type = 10;
   enum Framework
      OFFLINE =0:
      CAFFE
               =1;
      TENSORFLOW =2;
   Framework framework = 11;
   bytes data = 100; // 模型数据
   repeated AITensorDescription inputs = 12; // 输入Tensor描述
   repeated AITensorDescription outputs = 13; // 输出Tensor描述
```

4.3.7 AINNNodeDescription

NN(neural network)Node描述,主要用于描述NN Node需要的模型、输入tensor、输出tensor等,详细实现参考ai_types.proto。

4.3.8 AINNNodeDescriptionList

NN(neural network) Node描述列表,主要用于描述NN Node需要的模型、输入tensor、输出tensor等,详细实现参考ai types.proto。

```
message AINNNodeDescriptionList
{
    repeated AINNNodeDescription nnnode_descs = 1; // nnnode列表
}
```

4.3.9 AIAPIDescription

API描述,主要用于描述API的名称、输入tensor、输出tensor等,详细实现请参考ai_types.proto。

4.3.10 AIAPIDescriptionList

API描述列表,主要用于描述API的名称、输入tensor、输出tensor等,详细实现请参考ai_types.proto。

```
message AIAPIDescriptionList
{
    repeated AIAPIDescription api_descs = 1; // API列表
}
```

4.3.11 AIOPDescription

该类型在ai types.proto中定义。

```
//AI Operation描述
message AIOPDescription
{
    enum OP_Type
    {
        AI_API = 0;
        AI_NNNODE = 1;
    }

    OP_Type type = 1;
    AINNNodeDescription nnnode_desc = 2;
    AIAPIDescription api_desc = 3;
};
```

4.3.12 AIOPDescriptionList

该类型在ai_types.proto中定义。

```
//AI Operation描述列表
message AIOPDescriptionList
{
```

```
repeated AIOPDescription op_descs = 1; // AI Operation列表 }
```

4.3.13 NodeDesc

该类型在ai_types.proto中定义。

```
message NodeDesc {
    string name=1; // IAINNNode或ALG_API 名字
    AIConfig config=2; //IAINNNode或ALG_API 需要的初始化参数
    repeated AIModelDescription mode_desc=3; //IAINNNode 需要的初始化参数
}
```

4.3.14 EngineDesc

该类型在ai_types.proto中定义。

```
message EngineDesc
   enum RunSide
       DEVICE=0;
       HOST=1;
   enum EngineType
       NORMAL=0;
       SOURCE=1;
       DEST=2;
   uint32 id =1; //Engine ID (节点)
   EngineType type=2;
   string name =3; //Engine 节点名字
   repeated string so_name=4; //需要的所有动态库so文件名列表
   RunSide side=5; //部署在Host侧还是Device侧
   int32 priority=6; //节点优先级
   uint32 instance_cnt=7; //实例个数(相当于线程个数)
   repeated uint32 next_node_id=8; //下一个节点列表
   bool user_input_cb=9; //IDE 可以忽略
   bool user_output_cb=10; //IDE 可以忽略
   repeated NodeDesc oper=11; //HIAIEngine Node 列表
```

4.3.15 GraphInitDesc

该类型在ai_types.proto中定义。

```
message GraphInitDesc
{
  int32 priority=1; //Graph 整个进程的优先级
}
```

4.3.16 GeneralFileBuffer

该类型在ai_types.proto中定义。

```
message GeneralFileBuffer
{
    bytes raw_data = 1;
    string file_name = 2;
}
```

4.3.17 AIContext

在使用异步调用模型管家Process接口时,用于保存process上下文,AIContext中保存string类型键值对。该数据类型在ai_types.h中定义。

```
class AIContext
public:
   * @brief 获取参数
   * @param [in] key 参数对应的key
   * @return sring key对应的值,如果不存在,则返回空字符串
   const std::string GetPara(const std::string &key) const;
   * @brief 设置参数
                      参数对应的key
   * @param [in] key
   * @param [in] value 参数对应的value
   void AddPara(const std::string &key, const std::string &value);
   * @brief 删除参数
   * @param [in] key 待删除参数对应的key
   void DeletePara(const std::string &key);
   * @brief 获取所有参数
   * @param [out] keys 所有已设置的参数key
   void GetAllKeys(std::vector<std::string> &keys);
#if defined( __ANDROID__) || defined(ANDROID)
      std::string Serialize();
       AIStatus Deserialize(std::string str);
#endif
private:
   std::map<std::string, std::string> paras_; /** 参数的名值对定义 */
```

4.3.18 TensorDimension

调用**AIModelManager::GetModelIOTensorDim**接口可获取模型输入/输出尺寸的信息,主要包含tensor的dim信息,数据类型,内存大小,tensor名字。该数据类型在ai_types.h中定义。

```
/*

* 描述Tensor尺寸

*/

struct TensorDimension

{

    uint32_t n; //批量大小Batch
    uint32_t c; //特征图通道Channels
    uint32_t h; //特征图高度Height
    uint32_t w; //特征图宽度Width
    uint32_t data_type;
    uint32_t size;
    std::string name;

};
```

● 若执行模型转换时,未配置AIPP(AI Preprocessing,用于在AI Core上完成图像预处理)相关参数,则通过AIModelManager::GetModelIOTensorDim接口获取到的模型输入/输出尺寸的信息如下:

- n、c、h、w就是转换前模型的n、c、h、w。
- data type枚举值的定义如下:

```
enum DataType
{

DT_UNDEFINED = 0; // Used to indicate a DataType field has not been set.

DT_FLOAT = 1; // float type, 数据大小为4字节

DT_FLOAT16 = 2; // fp16 type, 数据大小为2字节

DT_INT8 = 3; // int8 type, 数据大小为1字节

DT_UINT8 = 4; // uint8 type, 数据大小为1字节

DT_INT16 = 5; // int16 type, 数据大小为2字节

DT_INT32 = 7; // int32 type, 数据大小为32字节

DT_INT32 = 7; // int32 type, 数据大小为4字节

DT_INT64 = 8; // int64 type, 数据大小为8字节

DT_UINT64 = 10; // unsigned int32, 数据大小为4字节

DT_BOOL = 11; // bool type, 数据大小为1字节

DT_BOUBLE = 12; // double type, 数据大小为8字节
```

- size=n*c*h*w*各类型数据的字节数
- 若执行模型转换时,配置AIPP相关参数,则通过 AIModelManager::GetModelIOTensorDim接口获取到的模型输入/输出尺寸的信息如下:
 - n、c、h、w依然是转换前模型的n、c、h、w,并不是在AIPP处配置的h(对应src_image_size_h参数)、w(对应src_image_size_w参数)。
 - data_type值固定为3,表示DT_INT8类型。
 - size的值为AIPP处理后的图片数据的大小,不同输入格式对应的size不同,具体见表4-1。

表 4-1 size 公式

input_format	size
YUV400_U8	n * src_image_size_w * src_image_size_h
YUV420SP_U8	n * src_image_size_w * src_image_size_h * 1.5
XRGB8888_U8	n * src_image_size_w * src_image_size_h * 4
RGB888_U8	n * src_image_size_w * src_image_size_h * 3

∭说明

关于模型转换或AIPP的说明,请参见《Atlas 500 模型转换指导》。

4.3.19 IAIListener

异步调用Process时需要配置IAIListener,用于模型执行结束后,回调通知,具体实现如下。该类在ai_nn_node.h中定义。

```
/*
* 异步回调接口,由调用方实现
*/
class IAIListener
{
public:
    virtual ~IAIListener() {}
    /*
    * @brief 异步回调接口
    * @param [in] context 运行时上下文信息,包含nnnode运行时的一些可变参数配置
```

4.3.20 IAITensor

Porcess函数时使用IAITensor作为模型的输入输出。该类在ai_tensor.h中定义。

```
* Tensor, 用于表示输入输出的数据
*/
class IAITensor
public:
   IAITensor() {}
   virtual ~IAITensor() {};
   * @brief 通过AITensorDescription设置参数
    * @param [in] tensor_desc tensor描述
    * @return true: init成功
    * @return false: init失败,失败的可能原因比如tensor_desc与当前tensor不符
   */
   virtual bool Init(const AITensorDescription &tensor_desc) = 0;
   * @brief 获取类型名称
   virtual const char* const GetTypeName() = 0;
   /*
   * @brief 获取序列化后字节长度
   virtual uint32_t ByteSizeLong() = 0;
   virtual void SetBufferAttr(void *data, int32_t size, bool isowner, bool is_support_mem_share)
= 0;
   virtual bool IsSupportZerocpy() = 0;
#if defined( __ANDROID__) || defined(ANDROID)
      * @brief 序列化到buffer, 用于数据跨进程交互
      * @param [in] buffer 序列化输出的内存区域地址,由调用方分配
      * @param [in] size 输出内存区域的大小
      * @return SUCCESS: 成功
      *
               FAILED: 失败,如果该Tensor不支持跨进程,则不需要实现此接口,直接返回失败
      */
      virtual AIStatus SerializeTo(void* buffer, const uint32_t size) = 0;
      * @brief 从buffer反序列化回tensor,用于数据跨进程交互
      * @param [in] buffer 输入的内存区域地址
      * @param [in] size 输入内存区域的大小
      * @return SUCCESS: 成功
               FAILED: 失败,如果该Tensor不支持跨进程,则不需要实现此接口,直接返回失败
      *
      */
      virtual AIStatus DeserializeFrom(const void* buffer, const uint32_t size) = 0;
#endif
```

4.4 其他用于编译依赖的接口

以下接口属于编译依赖接口,不推荐直接调用。

4.4.1 AIAlgAPIFactory

ALG的API注册工厂类。该类在ai_alg_api.h中定义。

```
class AIAlgAPIFactory
public:
   static AIAlgAPIFactory* GetInstance();
   * @brief 获取API
   * @param [in] name api名称
   * @return API原型指针
   AI_ALG_API GetAPI(const std::string &name);
   * @brief 注册API
   * @param [in] desc api描述
   * @param [in] func api定义
   * @return SUCCESS 成功
   * @return 其他
   AIStatus RegisterAPI(const AIAPIDescription &desc, AI_ALG_API func);
   * @brief 获取所有API描述
   * @param [in] api_desc_list api描述列表
   void\ Get All APIDescription (AIAPIDescriptionList\ \&api\_desc\_list);
   * @brief 卸载注册API
   * @param [in] api_desc api描述
   AIStatus UnRegisterApi(const AIAPIDescription &api_desc);
   * @brief 获取API描述
   * @param [in] name api名称
   * @param [in] api_desc api描述
   * @return SUCCESS 成功
   * @return 其他
   AIStatus GetAPIDescription(const std::string &name, AIAPIDescription &api_desc);
   std::map<std::string, AI_ALG_API> func_map_;
   std::map<std::string, AIAPIDescription> desc_map_;
   std::mutex api_reg_lock_;
```

4.4.2 AIAlgAPIRegisterar

对AIAlgAPIFactory工厂类的注册封装。该类在ai_alg_api.h中定义。

```
class AIAlgAPIRegisterar
{
public:
    AIAlgAPIRegisterar(const AIAPIDescription &desc, AI_ALG_API func)
    {
        AIAlgAPIFactory::GetInstance()->RegisterAPI(desc, func);
        api_desc_ = desc;
}
    ~AIAlgAPIRegisterar()
    {
        AIAlgAPIFactory::GetInstance()->UnRegisterApi(api_desc_);
}
```

```
}
private:
    AIAPIDescription api_desc_;
};
```

4.4.3 REGISTER_ALG_API_UNIQUE

API注册宏,在API实现中使用。该宏在ai_alg_api.h中定义。

```
#define REGISTER_ALG_API_UNIQUE(desc, ctr, func) \
AIAlgAPIRegisterar g_##ctr##_api_registerar(desc, func)
```

4.4.4 REGISTER_ALG_API

对REGISTER_ALG_API_UNIQUE的封装宏。该宏在ai_alg_api.h中定义。

```
#define REGISTER_ALG_API(name, desc, func) \
REGISTER_ALG_API_UNIQUE(desc, name, func)
```

4.4.5 AIModelManager

模型管理类。该类在ai model manager.h中定义。

```
class AIModelManager : public IAINNNode
          public:
                     AIModelManager();
                     * @brief 设置动态batch接口
                     * @param [in] inputDim 模型输入尺寸
                     * @param [in] input 输入数据
                     * @param [out] inputIndex 需要设置动态batch输入的序号,从0开始
                      * @param [in] batchNumber 动态batch数
                     * @return SUCCESS 成功
                      * @return 其他 失败
                     AIStatus SetInputDynamicBatch(const vector<TensorDimension>& inputDim, std::vector
<std::shared_ptr <IAITensor> > &input,
                                uint32_t inputIndex, uint32_t batchNumber);
                     * @brief 是否可以预分配输出内存,该接口由业务NNNode实现,默认值为true.
                     virtual bool IsPreAllocateOutputMem() override;
                      * @brief 获取AINNNodeDescription对象.
                     static AINNNodeDescription GetDescription();
                     * @brief 获取已加载模型的输入输出尺寸
                     * @param [in] model_name 模型名字
                      * @param [out] input_tensor 模型输入尺寸
                     * @param [out] output_tensor 模型输出尺寸
                     * @return SUCCESS 成功
                      * @return 其他 失败
                     AIStatus GetModelIOTensorDim(const std::string& model name,
                                \verb|std::vector<TensorDimension>& input\_tensor, std::vector<TensorDimension>& input\_tensor<TensorDimension>& input\_tensor, std::vector<TensorDimension>& i
                     * @brief 设置线程推理请求ID
                     * @param [in] request_id 模型名字
                      * @return 无
```

```
*/
static void SetRequestId(uint64_t request_id);

-AIModelManager();

#ifdef __LITE__
/*
    * @brief 闲置超时调用,释放资源。实现时调用的是UnloadModels。
    */
    virtual AIStatus IdleRelease();
/*
    * @brief 超时后恢复,包含打开设备,加载模型等。实现时调用的是LoadModels。
    */
    virtual AIStatus IdleResume();
#endif // __LITE__

private:
    AIModelManagerImpl* impl_;
};
```

4.4.6 getModelInfo

解析一个模型文件,获取模型信息。该接口在ai model parser.h中定义。

4.4.7 IAINNNode

NN Node接口,业务提供方实现。IAINNNode类在ai_nn_node.h中定义。

```
class IAINNNode
public:
   virtual ~IAINNNode() {}
   * @brief 初始化接口,业务在该接口中实现模型加载或其他初始化动作
   * @param [in] model_desc 模型信息,如果不需要模型,则传入空的vector
   * @param [in] config 配置参数
   * @return SUCCESS 成功
   * @return 其他
                 失败
   */
   virtual AIStatus Init(const AIConfig &config,
      const std::vector<AIModelDescription> &model_descs = {}) = 0;
   * @brief 设置监听
   * @parm [in] 如果listener设置为nullptr,表示process接口为同步调用,否则为异步
   * @return SUCCESS 成功
   * @return 其他
   */
   virtual AIStatus SetListener(std::shared ptr<IAIListener> listener) = 0;
   * @brief 计算接口
   * @param [in] context 运行时上下文信息,包含nnnode运行时的一些可变参数配置
   * @param [in] in_data 输入数据
   * @param [out] out_data 输出数据
   * @param [in] timeout 超时时间,同步调用时无效
```

```
* @return SUCCESS 成功
      * @return 其他
      virtual AIStatus Process (AIContext &context,
          const std::vector<std::shared_ptr<IAITensor>> &in_data,
          std::vector<std::shared_ptr<IAITensor>> &out_data, uint32_t timeout) = 0;
      * @brief 创建输出的Tensor列表
      * @param [in] in_data 输入tensor列表, 计算输出时可能使用
      * @param [out] out_data 输出的tensor列表
      * @return SUCCESS 成功
      * @return 其他
      */
      virtual AIStatus CreateOutputTensor(
          const std::vector<std::shared_ptr<IAITensor>> &in_data,
          std::vector<std::shared_ptr<IAITensor>> &out_data) { (void)in_data;
(void)out data;return SUCCESS; }
      * @brief 是否可以预分配输出内存,该接口由业务NNNode实现,默认值为true.
      virtual bool IsPreAllocateOutputMem() { return true; }
      * @brief 判断nnnode是否有效
      virtual AIStatus IsValid() { return SUCCESS; }
      * @brief 查询node 支持的同步方式
                    支持同步异步
仅支持异步
      * @return BOTH
      * @return ASYNC
      * @return SYNC
                      仅支持同步
      virtual AI_NODE_EXEC_MODE GetSupportedExecMode() { return AI_NODE_EXEC_MODE::BOTH; }
#ifdef __LITE_
      * @brief nnnode闲置超时,需要释放的资源,由用户自定义实现,适用于lite场景。如果使用了
modelmanger,需要在实现该函数时调用相应modelmanager释放资源的方法IdleRelease
      virtual AIStatus IdleRelease() { return SUCCESS; }
      * @brief nnnode超时后恢复,包含打开设备,加载模型等,由用户自定义实现,适用于lite场景。如果
使用了modelmanger,需要在实现该函数时调用相应modelmanager的方法IdleResume
      */
      virtual AIStatus IdleResume() { return SUCCESS; }
      * @brief 设置最大闲置时间,如果超过该时间则自动销毁释放资源,未设置情况下,默认60s
              业务构造函数中调用
              在AIServiceBase中实现
      * @param [in] time 最大闲置时间,单位ms
      virtual void SetMaxIdleTime(const int32 t time) { (void)time;}
#endif // __LITE__
      * @brief 获取业务最大使用内存
      * @return 业务最大使用内存大小
      virtual uint32_t GetMaxUsedMemory() { return 0; }
```

4.4.8 AINNNodeFactory

支持业务引擎自注册,提供创建NN Node接口,提供已注册NN Node描述信息查询(按名称查询以及查询全部)。AINNNodeFactory类在ai nn node.h中定义。

```
class AINNNodeFactory
   public:
       static AINNNodeFactory* GetInstance();
       * @brief 根据nnnode name创建NNNode
       * @param [in] name nnnode名称
       *@return std::shared_ptr<IAINNNode> name对应的NNNode对象指针,如果返回nullptr,表示找不到
对应的nnnode
       std::shared_ptr<IAINNNode> CreateNNNode(const std::string &name);
       * @brief 获取所有已注册nnnode的描述信息
       * @param [out] nnnode_desc 所有已注册的nnnode描述信息
       * @return SUCCESS 成功
       * @return 其他
       void GetAllNNNodeDescription(AINNNodeDescriptionList &nnnode desc);
       * @brief 根据nnnode name获取nnnode的描述信息
       * @param [in] name nnnode名称
       * @param [out] engin_desc nnnode描述信息
       * @return SUCCESS 成功
       * @return 其他
       AIStatus GetNNNodeDescription(const std::string &name, AINNNodeDescription &engin_desc);
       * @brief 注册NNNode创建函数
       * @param [in] nnnode_desc nnnode描述信息
       * @param [in] create_func nnnode创建函数
       * @return SUCCESS 成功
       * @return 其他
       AIStatus RegisterNNNodeCreator(const AINNNodeDescription &nnnode_desc,
          AINNNODE_CREATE_FUN create_func);
       AIS tatus\ Register NNN ode Creator (const\ string\ nnnode\_str,
          AINNNODE_CREATE_FUN create_func);
       * @brief 注销NNNode
       * @param [in] name nnnode名称
       * @return SUCCESS 成功
       AIS tatus \ UnRegister NNNode (const \ AINNNode Description \ \&nnnode\_desc);\\
       AIStatus UnRegisterNNNode(const string nnnode_str);
       std::map<std::string, AINNNODE_CREATE_FUN> create_func_map_;
       std::map<std::string, AINNNodeDescription> nnnode_desc_map_;
       std::mutex map_lock_;
```

4.4.9 AINNNodeRegisterar

NN Node注册类。AINNNodeRegisterar类在ai_nn_node.h中定义。

```
class AINNNodeRegisterar
{
public:
    AINNNodeRegisterar(const AINNNodeDescription &nnnode_desc, AINNNODE_CREATE_FUN create_func)
{
    AINNNodeFactory::GetInstance()->RegisterNNNodeCreator(nnnode_desc, create_func);
    nnnode_desc_ = nnnode_desc;
    nnnode_str_.erase(nnnode_str_.begin(),nnnode_str_.end());
}
```

```
~AINNNodeRegisterar() {
		AINNNodeFactory::GetInstance()->UnRegisterNNNode(nnnode_desc_);
	}

private:
	AINNNodeDescription nnnode_desc_;
	string nnnode_str_;
}:
```

4.4.10 REGISTER NN NODE

NN Node注册宏。

```
/*
 * @brief NNNode 注册宏,业务NNNode在实现类中使用
 * 直接使用 REGISTER_ENGINE(desc, clazz)
 * @param [in] desc nnnode描述信息对象
 * @param [in] clazz nnnode类名
 */
#define REGISTER_NN_NODE(desc, name) \
 std::shared_ptr<IAINNNode> NNNode_##name##_Creator() \
 { \
 return std::make_shared<name>(); \
 } \
 AINNNodeRegisterar g_nnnode_##name##_creator(desc, NNNode_##name##_Creator)
```

4.4.11 AITensorGetBytes

获取Tensor字节大小。该接口在ai_tensor.h中定义。

extern uint32_t AITensorGetBytes(int32_t data_type);

4.4.12 AITensorFactory

Tensor工厂类,用来创建Tensor。AITensorFactory类在ai tensor.h中定义。

```
class AITensorFactory
   public:
      static AITensorFactory* GetInstance();
      * @brief 通过参数创建Tensor, 包含分配内存
      * @param [in] tensor_desc 包含Tensor参数的描述信息
      * @return shared_ptr<IAITensor> 创建完成的Tensor指针,如果创建失败,则返回nullptr
      std::shared_ptr<IAITensor> CreateTensor(const AITensorDescription &tensor_desc);
       * @brief 通过type创建Tensor,不分配内存,非预分配情况下使用
      * @param [in] type Tensor注册时的类型
      * @return shared_ptr<IAITensor> 创建完成的Tensor指针,如果创建失败,则返回nullptr
      std::shared_ptr<IAITensor> CreateTensor(const std::string &type);
#if defined( __ANDROID__) || defined(ANDROID)
      * @brief 通过参数、buffer创建Tensor,内容从buffer反序列化得到
       * @param [in] tensor desc 包含Tensor参数的描述信息
      * @param [in] buffer 已存在的数据缓存
       * @param [in] size buffer的大小
       * @return shared_ptr<IAITensor> 创建完成的Tensor指针,如果创建失败,则返回nullptr
      std::shared_ptr<IAITensor> CreateTensor(const AITensorDescription &tensor_desc,
          const void *buffer, const int32_t size);
#else
```

```
std::shared_ptr<IAITensor> CreateTensor(const AITensorDescription &tensor_desc,
           void *buffer, int32_t size);
#endif
       * @brief 注册Tensor
       * @param [in] tensor_desc tensor描述
       * @param [in] create_func tensor创建函数
       AIStatus RegisterTensor(const AITensorDescription &tensor_desc,
           CREATE_TENSOR_FUN create_func);
       AIStatus RegisterTensor(const string tensor_str, CREATE_TENSOR_FUN create_func)
           // 加锁, 防止多线程并发
           AITensorDescription tensor_desc;
           tensor_desc. set_type(tensor_str);
           return RegisterTensor(tensor_desc, create_func);
       * @brief 卸载注册Tensor
       * @param [in] tensor_desc tensor描述
       AIStatus UnRegisterTensor(const AITensorDescription &tensor_desc);
       AIStatus UnRegisterTensor(const string tensor_str)
           AITensorDescription tensor_desc;
           tensor_desc. set_type(tensor_str);
           return UnRegisterTensor(tensor_desc);
       * @brief 获取所有的Tensor列表
       * @param [out] tensor_desc_list 输出Tensor描述列表
       void GetAllTensorDescription(AITensorDescriptionList &tensor_desc_list);
       /*
       * @brief 获取tensor描述
       AIStatus GetTensorDescription(const std::string& tensor_type_name,
           AITensorDescription &tensor_desc);
       std::map<std::string, AITensorDescription> tensor_desc_map_;
       std::map<std::string, CREATE TENSOR FUN> create func map;
       std::mutex tensor_reg_lock_;
```

4.4.13 REGISTER_TENSOR_CREATER_UNIQUE

Tensor注册宏。该宏在ai_tensor.h中定义。

```
#define REGISTER_TENSOR_CREATER_UNIQUE(desc, name, func) \
AITensorRegisterar g_##name##_tensor_registerar(desc, func)
```

4.4.14 REGISTER_TENSOR_CREATER

对Tensor注册宏的封装宏。该宏在ai_tensor.h中定义。

```
#define REGISTER_TENSOR_CREATER(name, desc, func) \
REGISTER_TENSOR_CREATER_UNIQUE(desc, name, func)
```

4.4.15 AISimpleTensor

简单类型的数据Tensor。该类在ai tensor.h中定义。

```
class AISimpleTensor : public IAITensor
      AISimpleTensor();
      ~AISimpleTensor();
      * @brief 获取类型名称
      virtual const char* const GetTypeName() override;
     /*获取数据地址*/
      void* GetBuffer();
      @brief 设置数据地址
      @param [in] data 数据指针
      @param [in] size 大小
      @param [in] isown 是否在tensor生命周期结束后,由tensor释放data地址内存
      void SetBuffer(void *data, const int32_t size, bool isown=false);
      @brief 获取数据占用空间大小
      uint32_t GetSize();
      @brief 分配数据空间
      void* MallocDataBuffer(uint32_t size);
      * @brief 获取序列化后字节长度
      virtual uint32_t ByteSizeLong() override;
      * @brief 通过AITensorDescription设置参数
      * @param [in] tensor desc tensor描述
      * @return true: init成功
      * @return false: init失败,失败的可能原因比如tensor_desc与当前tensor不符
      virtual bool Init(const AITensorDescription &tensor_desc) override;
#if defined( __ANDROID__) || defined(ANDROID)
      * @brief 序列化到buffer, 用于数据跨进程交互
      * @param [in] buffer 序列化输出的内存区域地址,由调用方分配
      * @param [in] size 输出内存区域的大小
      * @return SUCCESS: 成功
               FAILED: 失败,如果该Tensor不支持跨进程,则不需要实现此接口,直接返回失败
      */
      virtual AIStatus SerializeTo(void* buffer, const uint32 t size) override;
      * @brief 从buffer反序列化回tensor,用于数据跨进程交互
      * @param [in] buffer 输入的内存区域地址
      * @param [in] size 输入内存区域的大小
      * @return SUCCESS: 成功
               FAILED: 失败,如果该Tensor不支持跨进程,则不需要实现此接口,直接返回失败
      virtual AIStatus DeserializeFrom(const void* buffer, const uint32_t size) override;
#endif
      @brief 构建tensor描述
```

```
static AITensorDescription BuildDescription(const std::string& size = "");
       @brief 根据描述创建tensor
       */
       static std::shared_ptr<IAITensor> CreateTensor(const AITensorDescription& tensor_desc);
       bool IsSupportZerocpy();
       @brief 设置数据地址
       @param [in] data 数据指针
       @param [in] size 大小
       @param [in] isown 是否在tensor生命周期结束后,由tensor释放data地址内存
       @param [in] is_data_support_mem_share_ 是否在tensor生命周期结束后,由tensor释放data地址内存
       void SetBufferAttr(void *data, int32_t size, bool isowner, bool is_support_mem_share)
override;
   private:
       void* data ;
       uint32_t size_;
       bool isowner_ ;
       static std::string type_name_;
```

4.4.16 AINeuralNetworkBuffer

NN通用Buffer定义。该类在ai tensor.h中定义。

```
class AINeuralNetworkBuffer : public AISimpleTensor
   public:
      AINeuralNetworkBuffer()
          data\_type\_ = 0;
          number_ = 1,
-1 = 1;
          height_
                    = 1;
                    = 1;
= "";
          width_
          name_
       };
       ~AINeuralNetworkBuffer() {}:
       * @brief 获取类型名称
       */
       const char* const GetTypeName();
       @brief 获取size字节大小
       uint32_t ByteSizeLong();
       @brief 初始化
       @param [in] tensor_desc tensor描述
       bool Init(const AITensorDescription &tensor_desc);
#if defined( __ANDROID__) || defined(ANDROID)
       * @brief 序列化到buffer,用于数据跨进程交互
       * @param [in] buffer 序列化输出的内存区域地址,由调用方分配
       * @param [in] size 输出内存区域的大小
       * @return SUCCESS: 成功
               FAILED: 失败,如果该Tensor不支持跨进程,则不需要实现此接口,直接返回失败
       */
       AIStatus SerializeTo(void* buffer, uint32 t size);
```

```
/*
        * @brief 从buffer反序列化回tensor, 用于数据跨进程交互
        * @param [in] buffer 输入的内存区域地址
        * @param [in] size 输入内存区域的大小
        * @return SUCCESS: 成功
                 FAILED: 失败,如果该Tensor不支持跨进程,则不需要实现此接口,直接返回失败
        AIStatus DeserializeFrom(const void* buffer, uint32_t size_);
#endif
        * @brief 获取描述信息
        static AITensorDescription GetDescription(
           const std::string &size = "0", const std::string &data_type="0", const std::string &number="0", const std::string &channel="0", const std::string &height="0", const std::string &width="0");
        @brief 创建tensor
        static std::shared_ptr<IAITensor> CreateTensor(const AITensorDescription& tensor_desc);
        @brief 获取number数量
        */
        int32_t GetNumber();
        @brief 设置number
        void SetNumber(int32_t number);
        @brief 获取channel数量
        int32_t GetChannel();
        @brief 设置channel
        void SetChannel(int32_t channel);
        @brief 获取height
        int32_t GetHeight();
        @brief 设置height
        void SetHeight(int32_t height);
        @brief 获取width
        int32\_t \ GetWidth();
        /*
        @brief 设置width
        void SetWidth(int32_t width);
        @brief 获取数据类型
        int32_t GetData_type();
        /*
```

```
@brief 设置数据类型
*/
void SetData_type(int32_t data_type);

/*
    @brief 获取数据类型
    */
    const std::string& GetName() const;

/*
    @brief 设置数据类型
    */
    void SetName(const std::string& value);

private:
    int32_t data_type_;
    int32_t number_;
    int32_t channel_;
    int32_t height_;
    int32_t width_;
    std::string name_;
};
```

4.4.17 AIImageBuffer

图像的通用Buffer定义。该类在ai tensor.h中定义。

```
class AIImageBuffer : public AISimpleTensor
   public:
      AIImageBuffer()
         format_ = JPEG;
         width_ = 0;
height_ = 0;
      * @brief 获取类型名称
      const char* const GetTypeName();
      @brief 获取size字节大小
      */
      uint32_t ByteSizeLong();
      @brief 初始化
      @param [in] tensor_desc tensor描述
      bool Init(const AITensorDescription &tensor_desc);
#if defined( __ANDROID__) || defined(ANDROID)
      * @brief 序列化到buffer, 用于数据跨进程交互
      * @param [in] buffer 序列化输出的内存区域地址,由调用方分配
      * @param [in] size 输出内存区域的大小
      * @return SUCCESS: 成功
              FAILED: 失败,如果该Tensor不支持跨进程,则不需要实现此接口,直接返回失败
      */
      AIStatus SerializeTo(void* buffer, uint32_t size_);
      * @brief 从buffer反序列化回tensor, 用于数据跨进程交互
      * @param [in] buffer 输入的内存区域地址
      * @param [in] size 输入内存区域的大小
      * @return SUCCESS: 成功
             FAILED: 失败,如果该Tensor不支持跨进程,则不需要实现此接口,直接返回失败
```

```
AIStatus DeserializeFrom(const void* buffer, uint32_t size_);
#endif
        * @brief 获取描述信息
        {\tt static}\ {\tt AITensorDescription}\ {\tt GetDescription}(
            const std::string &size = "0", const std::string &height="0",
const std::string &width="0", const std::string format="JPEG");
        /*
        @brief 创建tensor
        static std::shared_ptr<IAITensor> CreateTensor(const AITensorDescription& tensor_desc);
        @brief 获取图像格式
        ImageFormat GetFormat();
        @brief 设置图像格式
        void SetFormat(ImageFormat format);
        @brief 获取height
        int32_t GetHeight();
        @brief 设置height
        void SetHeight(int32_t height);
        @brief 获取width
        int32\_t \ GetWidth();
        @brief 设置width
        */
        void SetWidth(int32_t width);
    private:
        ImageFormat format_;
        int32 t width;
        int32_t height_;
```

4.4.18 HIAILog

HiAI的日志类。该类在log.h中定义。

```
* @ingroup hiaiengine
 * @brief 获取log输出级别
 * @return :log输出级别
 HIAI_LIB_INTERNAL uint32_t HIAIGetCurLogLevel();
 * @ingroup hiaiengine
 * @brief 判断该条log是否需要输出
 * @param [in]errorCode:消息的错误码
 * @return 是否能被输出
 HIAI LIB INTERNAL bool HIAILogOutputEnable(const uint32 t errorCode);
 * @ingroup hiaiengine
 * @brief log对应的级别名称
 * @param [in]logLevel:宏定义的log级别
 HIAI LIB INTERNAL std::string HIAILevelName(const uint32 t logLevel);
 * @ingroup hiaiengine
 * @brief 输出log
 * @param [in]moudleID:enum定义的组件id,
 * @param [in]logLevel:宏定义的log级别,
 * @param [in]strLog:输出的日志内容
 HIAI_LIB_INTERNAL void HIAISaveLog(const int32_t moudleID,
     const uint32_t logLevel, const char* strLog);
protected:
 /**
 * @ingroup hiaiengine
 * @brief HIAILog构造函数
 HIAI_LIB_INTERNAL HIAILog();
 HIAI_LIB_INTERNAL ~HIAILog() {}
private:
 /**
 * @ingroup hiaiengine
 * @brief HIAILog初始化函数
 HIAI_LIB_INTERNAL void Init();
 HIAI_LIB_INTERNAL HIAILog(const HIAILog&) = delete;
 HIAI_LIB_INTERNAL HIAILog(HIAILog&&) = delete;
 HIAI_LIB_INTERNAL HIAILog& operator=(const HIAILog&) = delete;
 HIAI_LIB_INTERNAL HIAILog& operator=(HIAILog&&) = delete;
private:
 \verb|HIAI_LIB_INTERNAL| static std::mutex mutexHandle;|\\
 uint32_t outputLogLevel;
 std::map<uint32_t, std::string> levelName;
 std::map<std::string, uint32_t> levelNum;
 static bool isInitFlag;
```

4.4.19 HIAI_ENGINE_LOG

日志封装宏,对HIAI_ENGINE_LOG_IMPL宏进行封装。该宏在log.h中定义。

```
#define HIAI_ENGINE_LOG(...) \
    HIAI_ENGINE_LOG_IMPL(_FUNCTION__, __FILE__, __LINE__, ##__VA_ARGS__)
```

4.5 异常处理

当调用方出现调用离线模型管家接口失败,也即调用该函数返回值为FAILED时,可通过Mind Studio界面的Log窗口查看日志。根据Time列的时间查看最新日志,并根据日志的提示排查异常调用错误。

示例:调用方在使用Init接口时,加载的模型不存在:

Init:hiaiengine/node/ai_model_manager_impl.cpp:163:"Load models failed!"

□ 说明

关于日志查看的详细操作,可参见《Ascend 310 Mind Studio开发辅助工具》中的"日志工具>基本操作>日志查看"章节。

5 辅助接口

- 5.1 数据获取接口(C++语言)
- 5.2 数据类型序列化和反序列化(C++语言)
- 5.3 内存管理(C语言)
- 5.4 内存管理(C++语言)
- 5.5 日志 (C++语言)
- 5.6 队列管理MultiTypeQueue接口(C++语言)
- 5.7 事件注册接口(C++语言)
- 5.8 其他 (C++语言)

5.1 数据获取接口(C++语言)

通过以下辅助接口可以获取相关参数。数据获取的接口大部分为C语言与C++通用的接口, C++专用的接口在函数格式中进行了说明。

5.1.1 获取 Device 数目

获取device个数。该接口在c graph.h中定义。

函数格式

HIAI_StatusT HIAI_GetDeviceNum(uint32_t *devCount);

参数说明

参数	说明	取值范围
devCount	输出device个数的指针。	-

返回值

返回的部分错误码请参见"错误码示例"中的"错误码"列。

错误码

序号	错误码级别	错误码	错误码描述
1	HIAI_INFO	HIAI_OK	running ok
2	HIAI_ERROR	HIAI_GRAPH_GET_DE VNUM_ERROR	et device number error

调用示例

uint32_t dev_count; HIAI_GetDeviceNum(&dev_count);

5.1.2 获取第一个 DeviceID

获取第一个device的id。该接口在c graph.h中定义。

函数格式

HIAI_StatusT HIAI_GetFirstDeviceId(uint32_t *firstDevID);

参数说明

参数	说明	取值范围
firstDevID	第一个device的id的指针。	-

返回值

返回的部分错误码请参见"错误码示例"中的"错误码"列。

错误码

序号	错误码级别	错误码	错误码描述
1	HIAI_INFO	HIAI_OK	running ok
2	HIAI_ERROR	HIAI_GRAPH_GET_DE VID_ERROR	get device id error

调用示例

uint32_t first_dev_id; HIAI_GetFirstDeviceId(&first_dev_id);

5.1.3 获取第一个 GraphID

获取GraphID。该接口在c_graph.h中定义。

函数格式

C语言与C++通用接口: HIAI_StatusT HIAI_GetFirstGraphId(uint32_t *firstGraphID);

参数说明

参数	说明	取值范围
firstGraphID	第一个graph的id指针。	-

返回值

C语言与C++通用接口: 错误码。

错误码

序号	错误码级别	错误码	错误码描述
1	HIAI_INFO	HIAI_OK	running ok
2	HIAI_ERROR	HIAI_GRAPH_GET_G RAPHID_ERROR	get graph id error

调用示例

uint32_t first_graph_id; HIAI_GetFirstGraphId(&first_graph_id);

5.1.4 获取下一个 DeviceID

获取下一个device的id。该接口在c_graph.h中定义。

函数格式

HIAI StatusT HIAI GetNextDeviceId(const uint32 t curDevID,uint32 t *nextDevID);

参数说明

参数	说明	取值范围
curDevID	当前device的id号。	-
nextDevID	下一个device的id号的指针。	-

返回值

返回的部分错误码请参见"错误码示例"中的"错误码"列。

错误码

序号	错误码级别	错误码	错误码描述
1	HIAI_INFO	HIAI_OK	running ok
2	HIAI_ERROR	HIAI_GRAPH_GET_DE VID_ERROR	get device id error

调用示例

uint32_t cur_dev_id = 1; uint32_t next_dev_id; HIAI_GetNextDeviceId(cur_dev_id, &next_dev_id);

5.1.5 获取下一个 GraphID

获取下一个GraphID。该接口在c_graph.h中定义。

函数格式

C语言与C++通用接口: HIAI_StatusT HIAI_GetNextGraphId(const uint32_t curGraphID,uint32_t *nextGraphID);

参数说明

参数	说明	取值范围
curGraphID	当前graph id。	-
nextGraphID	下一个graph的id指针。	-

返回值

返回的部分错误码请参见"错误码示例"中的"错误码"列。

错误码

序号	错误码级别	错误码	错误码描述
1	HIAI_INFO	HIAI_OK	running ok
2	HIAI_ERROR	HIAI_GRAPH_GET_G RAPHID_ERROR	get graph id error

调用示例

```
uint32_t cur_graph_id = 1;
uint32_t next_graph_id;
HIAI_GetNextGraphId(cur_graph_id, &next_graph_id);
```

5.1.6 获取 Engine 指针

根据engineID查找Engine对象指针。该接口在graph.h中定义。

函数格式

std::shared_ptr<Engine> Graph::GetEngine(uint32_t engineID);

参数说明

参数	说明	取值范围
engineID	目标Engine的id	-

返回值

Engine的智能指针。

调用示例

```
uint32_t engineID= 1000;
auto graphPtr = Graph::GetInstance(100);
auto enginePtr = graphPtr->GetEngine(engineID);
```

5.1.7 获取 Graph 的 GraphId

获取当前graph的id。该接口在graph.h中定义。

函数格式

uint32_t Graph::GetGraphId();

返回值

graph的id。

5.1.8 获取 Graph 的 DeviceID

获取当前graph在device侧的id。该接口在graph.h中定义。

函数格式

uint32_t Graph::GetDeviceID();

返回值

当前graph在device侧的id。

调用示例

```
auto graphPtr = Graph::GetInstance(100);
auto deviceId = graphPtr->GetDeviceID();
```

5.1.9 获取 Engine 的 GraphId

获取Engine的GraphId。该接口在engine.h中定义。

函数格式

uint32_t Engine::GetGraphId();

返回值

Engine的GraphId。

调用示例

```
uint32_t engineID= 1000;
auto graphPtr = Graph::GetInstance(100);
auto enginePtr = graphPtr->GetEngine(engineID);
auto graphID= enginePtr->GetGraphId();
```

5.1.10 获取 Engine 队列最大大小

获取Engine的Queue最大Size。该接口在engine.h中定义。

函数格式

const uint32_t Engine::GetQueueMaxSize();

返回值

该Engine的Queue最大Size。

调用示例

```
uint32_t engineID= 1000;
auto graphPtr = Graph::GetInstance(100);
auto enginePtr = graphPtr->GetEngine(engineID);
auto engineQueueSize = enginePtr->GetQueueMaxSize();
```

5.1.11 获取 Engine 指定端口当前队列大小

获取Engine的某端口号Queue的Size。该接口在engine.h中定义。

函数格式

const uint32_t Engine::GetQueueCurrentSize(const uint32_t portID);

参数说明

参数	说明	取值范围
portID	端口号	-

返回值

该Engine的portID端口号Queue的Size。

调用示例

```
uint32_t engineID= 1000;
uint32_t portID = 0;
auto graphPtr = Graph::GetInstance(100);
auto enginePtr = graphPtr->GetEngine(engineID);
auto engineQueueSize = enginePtr->GetQueueCurrentSize(portID);
```

5.1.12 解析 Matrix 配置文件

解析配置文件。Matrix解析配置文件,并将生成的Graph回写到list中,供用户使用。该接口在graph.h中定义。

函数格式

static HIAI_StatusT Graph::ParseConfigFile(const std::string& configFile, GraphConfigList& graphConfigList)

参数说明

参数	说明	取值范围
configFile	配置文件路径。 请确保传入的文件路径是正确路 径。	-
graphConfigList	Protobuf数据格式。	-

返回值

返回的部分错误码请参见"错误码示例"中的"错误码"列。

错误码

序号	错误码级别	错误码	错误码描述
1	-	HIAI_GRAPH_GET_IN STANCE_NULL	-

5.1.13 获取 PCIe 的 Info

获取PCIe(Peripheral Component Interconnect Express)的信息。该接口在c_graph.h中定义。

函数格式

static HIAI_StatusT HIAI_GetPCIeInfo(const uint32_t devId,int32_t* bus, int32_t* dev, int32_t* func);

参数说明

参数	说明	取值范围
devId	需要查询的devId	-
bus	返回PCIe总线号	-
dev	返回PCIe设备号	-
func	返回PCIe功能号	-

返回值

返回的部分错误码请参见"错误码示例"中的"错误码"列。

错误码

序号	错误码级别	错误码	错误码描述
1	HIAI_INFO	HIAI_OK	running ok
2	HIAI_ERROR	HIAI_GRAPH_GET_PC IEINFO_ERROR	bus,dev,func中有空指针, 或者get pcie info error

调用示例

uint32_t devId= 1; uint32_t bus; uint32_t dev; uint32_t func;

HIAI_GetPCIeInfo(devId, &bus, &dev, &func);

5.1.14 获取版本号

获取API版本信息。该接口在c_graph.h中定义。

函数格式

HIAI_API_VERSION HIAI_GetAPIVersion();

返回值

版本信息的enum信息。

调用示例

HIAI_API_VERSION HIAI_GetAPIVersion();

5.1.15 获取 OamConfig 智能指针

获取OamConfig智能指针,用于获取oam配置信息,包括模型名称、是否打开Dump开关、需要Dump模型中哪些层的算子信息。该接口在engine.h文件中定义。

函数格式

OAMConfigDef* GetOamConfig();

参数说明

无。

返回值

OamConfig智能指针。OAMConfigDef类型的定义请参见7.1.1 Protobuffer数据类型。

5.2 数据类型序列化和反序列化(C++语言)

为用户自定义的各种数据类型提供自动化序列化和反序列化机制。

其中相关宏封装用到了以下接口:

static HIAIDataTypeFactory* HIAIDataTypeFactory::GetInstance();

5.2.1 宏:HIAI REGISTER DATA TYPE

为用户自定义的数据结构类型提供自动化序列化和反序列化机制。该宏在 data_type_reg.h中定义。

□说明

- 该接口需要在host和Device端同时注册。
- 因为cereal的限制, 结构体中应避免使用long double类型, 为了保证跨平台(如Windows到 Linux)运行, 结构体成员尽量使用可移植的类型, 如 int32 t等;
- 因为cereal的限制, 结构体中需使用共享指针。

函数格式

HIAI_REGISTER_DATA_TYPE(name, type)

参数说明

参数	说明	取值范围
name	用户自定义的数据结构类型名字(不同的数据类型要保证名字唯一)。	-
type	用户自定义的数据结构类 型。	-

返回值

返回的部分错误码请参见"错误码示例"中的"错误码"列。

错误码示例

序号	错误码	错误码描述
1	HIAI_OK	running ok
2	HIAI_ENGINE_FUNCTOR_NUL L	hiai engine function is null
3	HIAI_ENGINE_FUNCTOR_EXIS T	hiai engine function is existed

5.2.2 宏:HIAI_REGISTER_TEMPLATE_DATA_TYPE

为用户自定义的模板的数据结构类型提供自动化序列化和反序列化机制。该宏在 data_type_reg.h中定义。

∭说明

该接口需要在host和Device端同时注册该类。

函数格式

HIAI REGISTER TEMPLATE DATA TYPE(name, type, basictype1, basictype2, ···)

参数说明

参数	说明	取值范围
name	用户自定义的数据结构类型名字(不同的数据结构类类型要保证名字唯一)。	-
type	用户自定义的模板的数据 结构类型。	-
basicType1	用户自定义的数据类型。	-
basicType2	用户自定义的数据类型。	-
		-

返回值

返回的部分错误码请参见"错误码示例"中的"错误码"列。

错误码示例

序号	错误码	错误码描述
1	HIAI_OK	running ok
2	HIAI_ENGINE_FUNCTOR_NUL L	hiai engine function is null
3	HIAI_ENGINE_FUNCTOR_EXIS T	hiai engine function is existed

5.2.3 宏:HIAI_REGISTER_SERIALIZE_FUNC

为用户自定义的数据类型提供自定义的序列化和反序列化机制。该宏在data_type_reg.h 中定义。

即发送时将用户传输的结构体指针转化为结构体buffer和数据buffer,接受数据时,将从框架获取的结构体buffer和数据buffer反转为结构体。

使用场景:

用于从host侧到Device侧快速搬运数据,如果用户需要提升传输性能,必须使用该接口注册函数。

∭说明

- 该接口用于从host侧到Device侧快速搬运数据使用。
- 如果用户想使用高性能传输接口,必须使用该注册函数注册序列化和反序列化函数。

函数格式

HIAI_REGISTER_SERIALIZE_FUNC(name, type, hiaiSerializeFunc, hiaiDeSerializeFunc)

参数说明

参数	说明	取值范围
name	注册的消息名字。	-
type	自定义数据结构的类型。	-

参数	说明	取值范围
hiaiSerializeFunc	序列化函数。函数形式为	-
	typedef void(*hiaiSerializeFunc) (void* inputPtr, std::string& ctrlStr,	
	uint8_t*& dataPtr, uint32_t& dataLen);	
	各参数说明如下:	
	● inputPtr: 结构体指针	
	● ctrlStr: 结构体buffer	
	● dataPtr: 结构体数据指 针buffer	
	● dataLen: 结构体数据大 小	
hiaiDeSerializeFunc	反序列化函数。函数形式 为	-
	typedef std::shared_ptr <void>(*hiai DeSerializeFunc)(</void>	
	const char* ctrlPtr, const uint32_t& ctrlLen,	
	const uint8_t* dataPtr, const uint32_t& dataLen);	
	各参数说明如下:	
	● ctrlPtr: 结构体指针	
	● ctrlLen:数据结构中控制信息大小	
	● dataPtr: 结构体数据指 针	
	● dataLen:结构中数据信息存储空间大小,仅用于校验,不表示原始数据信息大小	

返回值

返回的部分错误码请参见"错误码示例"中的"错误码"列。

错误码示例

序号	错误码	错误码描述
1	HIAI_OK	running ok

序号	错误码	错误码描述
2	HIAI_ENGINE_FUNCTOR_NUL L	hiai engine function is null
3	HIAI_ENGINE_FUNCTOR_EXIS T	hiai engine function is existed

5.2.4 Graph::ReleaseDataBuffer

在数据进行反序列化过程中,数据内存赋值给智能指针时,将本函数注册为删除器。 该接口在graph.h中定义。

函数格式

static void Graph::ReleaseDataBuffer(void* ptr)

参数说明

参数	说明	取值范围
ptr	内存指针	-

5.2.5 接口使用示例

```
CopyRight (C) Hisilicon Co., Ltd.
      Filename: \ user\_def\_datatype.\,h
    Description: User defined data type
       Version: 1.0
Created: 2018-01-08 10:15:18
       Author:
      Revision: initial draft;
#ifndef USE_DEF_DATA_TYPE_H_
#define USE_DEF_DATA_TYPE_H_
#ifdef __cplusplus
#include <vector>
#include <map>
#endif
// 用户自定义的数据类型 (C 语言风格)
typedef struct UseDefDataType
   int datal;
  float data2;
}UseDefDataTypeT;
#ifdef __cplusplus
// 用户自定义的数据类型 (带模板类型)
template<typename T1, typename T2, typename T3>
class UseDefTemplateDataType
```

```
public:
   std::vector<T1> data_vec_;
   std::map<T2, T3> data_map_;
#endif
#endif
CopyRight (C) Hisilicon Co., Ltd.
       Filename: \quad use\_def\_data\_type\_reg.\,cpp
*
     Description: User defined Data Type Register
        Version: 1.0
Created: 2018-01-08 10:15:18
         Author: h00384043
       Revision: initial draft:
#include "use_def_data_type.h"
#include "hiaiengine/data_type_reg.h"
// 用户自定义的数据类型必须先注册到Matrix,才能作为Engine之间的通信接口正常使用
template < class Archive>
void serialize(Archive& ar, UseDefDataTypeT& data)
   ar(data.data1, data.data2);
// 注册UseDefDataTypeT
HIAI_REGISTER_DATA_TYPE("UseDefDataTypeT", UseDefDataTypeT)
// 模板类型数据的注册
// 对于模板类型,对于所有需要使用的类型必须都注册
template <class Archive, typename T1, typename T2, typename T3>
void serialize(Archive& ar, UseDefTemplateDataType<T1, T2, T3>& data)
   ar(data.data_vec_, data.data_map_);
// 注册 UseDefTemplateDataType<int, float>
HIAI_REGISTER_TEMPLATE_DATA_TYPE("UseDefTemplateDataType_uint64_t_float_string",
UseDefTemplateDataType, uint64_t, float, std::string);
//... 注册其他类型
// 注册 UseDefTemplateDataType<int, int>
HIAI_REGISTER_TEMPLATE_DATA_TYPE("UseDefTemplateDataType_uint64_t_uint64_t",
UseDefTemplateDataType, uint64_t, uint64_t, uint64_t);
// 注册结构体的序列化和反序列化
typedef struct tagST_SC_IMAGE
   INT32 iWidth;
                               // image width
   INT32 iHeight;
                               // image height
   INT32 iWidthStep;
                              // Size of aligned image row in bytes
   INT32 iChannel;
                              // channels
   INT32 iDepth;
                               // depth in bits
   UINT32 uiTimeStampH;
   UINT32 uiTimeStampL;
   UINT32 iSize;
   UINT64 uiID;
   EN_SC_IMAGE_TYPE eType;
                                // image type
   ST_SC_IMAGE_ROI *roi;
                                \ensuremath{//} Image ROI. If NULL, the whole image is selected
   UINT8* pucImageData;
                                // Image data
   UINT64 uiPhyAddr;
}ST_SC_IMAGE;
```

```
std::shared_ptr\sT_SC_IMAGE\> st_image = std::make_shared\sT_SC_IMAGE\>();
st_image->iWidth = 1080;
st_image->iHeight = 1080;
st_image->iChannel = 1;
st image->iWidthStep = 1080;
st_image->iSize = 1080*1080*3/2 ;
st_image->eType = SC_IMAGE_YUV420SP;
st_image->roi = (ST_SC_IMAGE_ROI*)malloc(sizeof(ST_SC_IMAGE_ROI));
st_image->pucImageData = nullptr;
uint8_t* align_buffer = nullptr;
HIAI_StatusT get_ret = HIAIMemory::HIAI_DMalloc(st_image->iSize, (void*&) align_buffer, 10000);
hiai::EnginePortID engine_id;
engine id. graph id = 1;
engine_id.engine_id = 1;
engine_id.port_id = 0;
HIAI_SendData(engine_id, "ST_SC_IMAGE", std::static_pointer_cast<void>(st_image), 100000);
void GetStScImageSearPtr(void* inputPtr, std::string& ctrlStr, uint8_t*& dataPtr, uint32_t&
dataLen)
         // 如果结构体内有除了image之外的多个指针, 则进行拼接
        ST_SC_IMAGE* st_image = (ST_SC_IMAGE*)inputPtr;
        ctrlStr = std::string((char*)inputPtr, sizeof(ST_SC_IMAGE));
        if(nullptr != st_image->roi)
                std::string image_roi_str = std::string((char*)st_image->roi, sizeof(ST_SC_IMAGE_ROI));
                ctrlStr += image_roi_str;
        dataPtr = (UINT8*)st_image->pucImageData;
        dataLen= st_image->iSize;
std::shared_ptr<void> GetStScImageDearPtr(const char* ctrlPtr, const uint32_t& ctrlLen, const
uint8_t* dataPtr, const uint32_t& dataLen)
        ST_SC_IMAGE* st_sc_image = (ST_SC_IMAGE*)ctrlPtr;
        std:: shared\_ptr \\ \langle hiai:: Batch Image Para \\ \langle uint8\_t \rangle \\ Image Info (new hiai:: Batch Image Para \\ \langle uint8\_t \rangle); \\ (new hiai:: Batch Image Para \\ \langle uint8\_t \rangle); \\ (new hiai:: Batch Image Para \\ \langle uint8\_t \rangle); \\ (new hiai:: Batch Image Para \\ \langle uint8\_t \rangle); \\ (new hiai:: Batch Image Para \\ \langle uint8\_t \rangle); \\ (new hiai:: Batch Image Para \\ \langle uint8\_t \rangle); \\ (new hiai:: Batch Image Para \\ \langle uint8\_t \rangle); \\ (new hiai:: Batch Image Para \\ \langle uint8\_t \rangle); \\ (new hiai:: Batch Image Para \\ \langle uint8\_t \rangle); \\ (new hiai:: Batch Image Para \\ \langle uint8\_t \rangle); \\ (new hiai:: Batch Image Para \\ \langle uint8\_t \rangle); \\ (new hiai:: Batch Image Para \\ \langle uint8\_t \rangle); \\ (new hiai:: Batch Image Para \\ \langle uint8\_t \rangle); \\ (new hiai:: Batch Image Para \\ \langle uint8\_t \rangle); \\ (new hiai:: Batch Image Para \\ \langle uint8\_t \rangle); \\ (new hiai:: Batch Image Para \\ \langle uint8\_t \rangle); \\ (new hiai:: Batch Image Para \\ \langle uint8\_t \rangle); \\ (new hiai:: Batch Image Para \\ \langle uint8\_t \rangle); \\ (new hiai:: Batch Image Para \\ \langle uint8\_t \rangle); \\ (new hiai:: Batch Image Para \\ \langle uint8\_t \rangle); \\ (new hiai:: Batch Image Para \\ \langle uint8\_t \rangle); \\ (new hiai:: Batch Image Para \\ \langle uint8\_t \rangle); \\ (new hiai:: Batch Image Para \\ \langle uint8\_t \rangle); \\ (new hiai:: Batch Image Para \\ \langle uint8\_t \rangle); \\ (new hiai:: Batch Image Para \\ \langle uint8\_t \rangle); \\ (new hiai:: Batch Image Para \\ \langle uint8\_t \rangle); \\ (new hiai:: Batch Image Para \\ \langle uint8\_t \rangle); \\ (new hiai:: Batch Image Para \\ \langle uint8\_t \rangle); \\ (new hiai:: Batch Image Para \\ \langle uint8\_t \rangle); \\ (new hiai:: Batch Image Para \\ \langle uint8\_t \rangle); \\ (new hiai:: Batch Image Para \\ \langle uint8\_t \rangle); \\ (new hiai:: Batch Image Para \\ \langle uint8\_t \rangle); \\ (new hiai:: Batch Image Para \\ \langle uint8\_t \rangle); \\ (new hiai:: Batch Image Para \\ \langle uint8\_t \rangle); \\ (new hiai:: Batch Image Para \\ \langle uint8\_t \rangle); \\ (new hiai:: Batch Image Para \\ \langle uint8\_t \rangle); \\ (new hiai:: Batch Image Para \\ \langle uint8\_t \rangle); \\ (new hiai:: Batch Image Para \\ \langle uint8\_t \rangle); \\ (new hiai:: Batch Image Para \\ \langle uint8\_t \rangle); \\ (new hiai:: Batch Image Para \\ \langle uint8\_t \rangle); \\ (new hiai:: Batch Image Para \\ \langle uint8\_t \rangle); \\ (new hiai:: Batch Image Para \\ \langle uint8\_t \rangle); 
        hiai::ImageData<uint8_t> image_data;
        image\_data.width = st\_sc\_image->iWidth;
        image_data.height = st_sc_image->iHeight;
        image_data.channel = st_sc_image->iChannel;
        image_data.width_step = st_sc_image->iWidthStep;
        if (st sc image->eType == SC IMAGE U8C3PLANAR)
              image_data.size = st_sc_image->iWidth * st_sc_image->iHeight * 3;
              image_data.format = hiai::BGR888;
        else if (SC IMAGE U8C3PACKAGE == st sc image->eType)
               image_data.size = st_sc_image->iWidth * st_sc_image->iHeight * 3;
              image_data.format = hiai::BGR888;
        else if (st_sc_image->eType == SC_IMAGE_YUV420SP)
              image_data.size = st_sc_image->iSize;//st_sc_image->iWidth * st_sc_image->iHeight * 3 / 2;
              image_data.format = hiai::YUV420SP;
        image data. data. reset(dataPtr, hiai::Graph::ReleaseDataBuffer);
        ImageInfo->v_img.push_back(image_data);
        ImageInfo->b_info.frame_ID.push_back(0);
        ImageInfo->b_info.batch_size = ImageInfo->b_info.frame_ID.size();
        return std::static_pointer_cast<void>(ImageInfo);
HIAI_REGISTER_SERIALIZE_FUNC("ST_SC_IMAGE", ST_SC_IMAGE, GetStScImageSearPtr, GetStScImageDearPtr);
```

5.3 内存管理(C语言)

接口使用要点

通过HIAI DMalloc接口申请内存,关于内存管理,请注意以下要点:

- 申请自动释放内存,用于Host到Device或Device到Host的数据传输时,如果是智能指针,由于Matrix框架自动释放内存,所以智能指针指定的析构器必须是空的;如果非智能指针,则Matrix框架自动释放。
- 申请手动释放内存,用于Host到Device或Device到Host的数据传输时,如果是智能指针,则需要指定析构器为HIAI_DFree;如果非智能指针,则数据发送完成后需要调用HIAI DFree释放内存。
- 申请自动释放内存,对于该内存中的数据,不允许多次调用SendData接口发送数据。
- 申请手动释放内存时,如果用于Host到Device或Device到Host的数据传输时,在内存释放前,不可复用内存中的数据;如果用于Host到Host或Device到Device的数据传输时,在内存释放前,可以复用内存中的数据。
- 申请手动释放内存时,如果调用SendData接口异步传输数据,发送数据后,不允许修改内存中的数据。

如果调用HIAI_DVPP_MAlloc接口申请内存,用于Device到Host的数据传输时,由于HIAI_DVPP_MAlloc没有自动释放标签,所以一定需要调用HIAI_DVPP_DFree接口手动释放内存。如果使用智能指针存放申请的内存地址,必须指定析构器为HIAI_DVPP_DFree。

5.3.1 HIAI_DMalloc

通过Matrix接口申请Host侧或Device侧的内存块,配合高效数据传输使用。该接口在c graph.h中定义。

若要给Device端分配DVPP使用的内存,您需要参考5.3.3 HIAI DVPP DMalloc。

使用场景:

通过Matrix框架,将大数据从Host端搬运到Device端,该接口需要配合**5.2.3** 宏:HIAI REGISTER SERIALIZE FUNC使用。

例如:需要将1080P或者4K图像发送到Device端,如果需要提升传输性能,则必须通过HIAI_REGISTER_SERIALIZE_FUNC注册结构体转换和反转函数,另外,大数据块内存则通过HIAI DMalloc接口申请内存,使用该方式,传输性能将得到很大提升。

∭说明

该接口主要用于Host与Device的搬运大数据的性能问题,如无性能要求或非传输场景,不建议使用本接口。

函数格式

void* HIAI DMalloc (const uint32 t dataSize, const uint32 t timeOut, uint32 t flag)

参数说明

参数	说明	取值范围
dataSize	内存块大小。	● 如果调用该接口申请Host侧内存,则取值范围是: 0~256M Bytes,不包含0。推荐使用256K Bytes~256M Bytes。 ● 如果调用该接口申请Device侧内存,则取值范围是: 0~(256M Bytes-96 Bytes),不包含0。推荐使用(256K Bytes-96 Bytes)。Device侧模型管家会占用96 Bytes。
timeOut	当内存申请失败时,提供 时延进行阻塞等待有空余 内存,默认值为500毫秒。	-
flag	● MEMORY_ATTR_AUT O_FREE:默认值,如果设置了此值,表示如果分配了内存,且通据到对端,则无需调用HIAI_DFree,程序运行结束后,内存配了内存。但没有通过SendData接口发送数据到对端,则需要调用HIAI_DFree释放内存。 ● MEMORY_ATTR_MANUAL_FREE:如伊种情况,必须调用HIAI_DFree释放内存。 ● MEMORY_ATTR_NONE和MEMORY_ATTR_MAX当前未使用。	typedef enum { MEMORY_ATTR_NONE = 0, //框架自行释放DMalloc内存 MEMORY_ATTR_AUTO_FREE = (0x1 << 1), //需要手动调用DFree释放内存 MEMORY_ATTR_MANU AL_FREE = (0x1 << 2), MEMORY_ATTR_MAX } HIAI_MEMORY_ATTR;

返回值

使用HIAI_DMalloc接口申请到的内存地址。如果申请失败,则返回空指针。

5.3.2 HIAI_DFree

释放通过HiAI接口申请的内存。该接口在c graph.h中定义。

使用场景:

当用户通过**5.3.1 HIAI_DMalloc**申请内存后并且没有调用SendData接口发送数据,必须通过该接口释放内存,如果已经调用SendData接口,则不需要调用该接口释放内存。

函数格式

HIAI StatusT HIAI DFree (void* dataBuffer)

参数说明

参数	说明	取值范围
dataBuffer	被释放的内存指针。	-

返回值

返回的部分错误码请参见"错误码示例"中的"错误码"列。

错误码示例

序号	错误码	错误码描述
1	HIAI_OK	running ok
2	HIAI_GRAPH_NOT_EXIST	graph not exist
3	HIAI_GRAPH_INVALID_VALUE	graph invalid value
4	HIAI_GRAPH_NOT_FIND_MEM ORY	can't find the memory

5.3.3 HIAI DVPP DMalloc

用于分配内存,该接口主要用于分配内存给Device端的DVPP使用。该接口在c_graph.h 中定义。调用该接口申请内存后,必须使用HIAI_DVPP_DFree接口释放内存。

函数格式

void* HIAI_DVPP_DMalloc(const uint32_t dataSize)

参数说明

参数	说明	取值范围
dataSize	内存块大小。	0~(256M Bytes-96 Bytes), 不包含0。推荐使用(256K Bytes-96 Byte)~(256M Bytes-96 Bytes)。 Device侧模型管家会占用 96 Bytes。

返回值

使用HIAI DVPP DMalloc接口申请到的内存地址。如果申请失败,则返回空指针。

使用说明

HIAI_DVPP_DMalloc是专门为了给DVPP分配内存而新增的一个接口,通过该接口申请的内存能够满足DVPP对内存的各种限制及要求。

如果您在旧版本中使用HIAI_DMalloc(1000, buffer, 1000, HIAI_MEMORY_HUGE_PAGE)方式为DVPP申请内存,则需要修改为使用HIAI_DVPP_DMalloc接口申请内存。关于HIAI_DMalloc接口,请参见**5.3.1** HIAI_DMalloc。

使用范例:

```
// 使用HIAI_DVPP_DMalloc申请内存
uint32_t allocSize = 1000;
char* allocBuffer = (char*)HIAI_DVPP_DMalloc(allocSize);
if (allocBuffer == nullptr) {
    // 分配失败,进行异常处理
}
```

5.3.4 HIAI_DVPP_DFree

用于释放内存, 该接口主要用于释放 $HIAI_DVPP_DMalloc$ 接口分配的内存。该接口在 $c_graph.h$ 中定义。

函数格式

HIAI StatusT HIAI DVPP DFree(void* dataBuffer)

参数说明

参数	说明	取值范围
dataBuffer	被释放的内存地址。	-

返回值

返回的部分错误码请参见"错误码示例"中的"错误码"列。

错误码示例

序号	错误码	错误码描述
1	HIAI_OK	running ok
2	HIAI_GRAPH_NOT_EXIST	graph not exist
3	HIAI_GRAPH_INVALID_VALUE	graph invalid value
4	HIAI_GRAPH_NOT_FIND_MEM ORY	can't find the memory

5.4 内存管理(C++语言)

接口使用要点

通过HIAIMemory::HIAI DMalloc接口申请内存,关于内存管理,请注意以下要点:

- 申请自动释放内存,用于Host到Device或Device到Host的数据传输时,如果是智能指针,由于Matrix框架自动释放内存,所以智能指针指定的析构器必须是空的;如果非智能指针,则Matrix框架自动释放。
- 申请手动释放内存,用于Host到Device或Device到Host的数据传输时,如果是智能指针,则需要指定析构器为HIAIMemory::HIAI_DFree;如果非智能指针,则数据发送完成后需要调用HIAIMemory::HIAI_DFree释放内存。
- 申请自动释放内存,对于该内存中的数据,不允许多次调用SendData接口发送数据。
- 申请手动释放内存时,如果用于Host到Device或Device到Host的数据传输时,在内存释放前,不可复用内存中的数据;如果用于Host到Host或Device到Device的数据传输时,在内存释放前,可以复用内存中的数据。
- 申请手动释放内存时,如果调用SendData接口异步传输数据,发送数据后,不允许修改内存中的数据。

如果调用HIAIMemory::HIAI_DVPP_DMalloc接口申请内存,用于Device到Host的数据传输时,由于HIAIMemory::HIAI_DVPP_DMalloc没有自动释放标签,所以一定需要调用HIAIMemory::HIAI_DVPP_DFree接口手动释放内存。如果使用智能指针存放申请的内存地址,必须指定析构器为HIAIMemory::HIAI_DVPP_DFree。

5.4.1 HIAIMemory::HIAI_DMalloc

通过Matrix接口申请Host侧或Device侧的内存块,配合高效数据传输使用。该接口在ai_memory.h中定义。

若要给Device端分配DVPP使用的内存,您需要参考**5.4.4 HIAIMemory::HIAI_DVPP_DMalloc**。

使用场景:

通过Matrix框架,将大数据从Host端搬运到Device端,该接口需要配合**5.2.3 宏:HIAI REGISTER SERIALIZE FUNC**使用。

例如:需要将1080P或者4K图像发送到Device端,如果需要提升传输性能,则必须通过HIAI_REGISTER_SERIALIZE_FUNC注册结构体转换和反转函数,另外,大数据块内存则通过HIAI DMalloc接口申请内存,使用该方式,传输性能将得到很大提升。

□ 说明

- 该接口主要用于Host与Device的搬运大数据的性能问题,不推荐当做普通的malloc使用。
- 出于性能考虑, 该接口会预申请内存, 实际占用内存大小与申请内存大小存在一定差异, 该接口的运行时间也会存在波动。

函数格式

static HIAI_StatusT HIAIMemory::HIAI_DMalloc (const uint32_t dataSize, void*& dataBuffer, const uint32_t timeOut = MALLOC_DEFAULT_TIME_OUT, uint32_t flag = MEMORY ATTR AUTO FREE)

参数	说明	取值范围
dataSize	内存块大小。	● 如果调用该接口申请Host 侧内存,则取值范围是: 0~256M Bytes,不包含 0。推荐使用256K Bytes~256M Bytes。 ● 如果调用该接口申请 Device侧内存,则取值范围是: 0~(256M Bytes-96 Bytes),不包含0。推荐使用(256K Bytes-96 Bytes)。Device侧模型管家会占用96 Bytes。
dataBuffer	内存指针。	-
timeOut	当内存申请失败时,提供时延进 行阻塞等待有空余内存,默认值 为 MALLOC_DEFAULT_TIME_OU T(表示500毫秒)。	-

参数	说明	取值范围
flag	● MEMORY_ATTR_AUTO_FRE E:默认值,如果设置了此值,表示如果分配了内存,且通过SendData接口发送数据到对端,则无需调用HIAIMemory::HIAI_DFree,程序运行结束后,内存会自动释放;如果分配了内存,但没有通过SendData接口发送数据到对端,则需要调用HIAIMemory::HIAI_DFree释放内存。 ● MEMORY_ATTR_MANUAL_FREE:如设置此值,表示无论何种情况,必须调用HIAIMemory::HIAI_DFree释放内存。 ● MEMORY_ATTR_NONE和MEMORY_ATTR_NONE和MEMORY_ATTR_MAX当前未使用。	typedef enum { MEMORY_ATTR_NONE = 0, //框架自行释放DMalloc内存 MEMORY_ATTR_AUTO_FR EE = (0x1 << 1), //需要手动调用DFree释放内 存 MEMORY_ATTR_MANUAL _FREE = (0x1 << 2), MEMORY_ATTR_MAX } HIAI_MEMORY_ATTR;

返回值

返回的部分错误码请参见"错误码示例"中的"错误码"列。

错误码示例

序号	错误码	错误码描述
1	HIAI_OK	running ok
2	HIAI_GRAPH_NOT_EXIST	graph not exist
3	HIAI_GRAPH_MEMORY_POOL _NOT_EXISTED	memory pool is not existed
4	HIAI_GRAPH_MALLOC_LARG ER	failed to malloc buffer due to the size larger than 128M
5	HIAI_MEMORY_POOL_UPDAT E_FAILED	failed to update memory pool
6	HIAI_GRAPH_SENDMSG_FAIL ED	hdc send msg failed
7	HIAI_GRAPH_MEMORY_POOL _INITED	memory pool has inited
8	HIAI_GRAPH_NO_MEMORY	no memory

5.4.2 HIAIMemory::HIAI_DFree

释放通过HiAI接口申请的内存。该接口在ai_memory.h中定义。

使用场景:

当用户通过**5.4.1 HIAIMemory::HIAI_DMalloc**申请内存后并且没有调用SendData接口发送数据,必须通过该接口释放内存,如果已经调用SendData接口,则不需要调用该接口释放内存。

函数格式

static HIAI_StatusT HIAIMemory::HIAI_DFree (void* dataBuffer)

参数说明

参数	说明	取值范围
dataBuffer	需要释放的内存指针	-

返回值

返回的部分错误码请参见"错误码示例"中的"错误码"列。

错误码示例

序号	错误码	错误码描述
1	HIAI_OK	running ok
2	HIAI_GRAPH_NOT_EXIST	graph not exist
3	HIAI_GRAPH_INVALID_VALUE	graph invalid value
4	HIAI_GRAPH_NOT_FIND_MEM ORY	can't find the memory

5.4.3 HIAIMemory:: IsDMalloc

用户可调用该接口判断内存是否是由DMalloc申请。该接口在ai memory.h中定义。

函数格式

static bool IsDMalloc(const void* dataBuffer, const uint32 t& dataSize)

参数	说明	取值范围
dataBuffer	需要判断的内存指针	-

参数	说明	取值范围
dataSize	内存地址的大小,单位是 字节	-

返回值

true: 由DMalloc申请内存。 false: 不由DMalloc申请内存

5.4.4 HIAIMemory::HIAI_DVPP_DMalloc

用于分配内存,该接口主要用于分配内存给Device端的DVPP使用。该接口在ai_memory.h中定义。调用该接口申请内存后,必须使用HIAIMemory::HIAI_DVPP_DFree接口释放内存。

函数格式

HIAI_StatusT HIAI_DVPP_DMalloc(const uint32_t dataSize, void*& dataBuffer)

参数说明

参数	说明	取值范围
dataSize	内存块大小。	0~(256M Bytes-96 Bytes), 不包含0。推荐使用(256K Bytes-96 Byte)~(256M Bytes-96 Bytes)。 Device侧模型管家会占用 96 Bytes。
dataBuffer	分配出来的内存地址,如 果分配失败,为nullptr。	-

返回值

返回的部分错误码请参见"错误码示例"中的"错误码"列。

错误码示例

序号	错误码	错误码描述
1	HIAI_OK	running ok
2	HIAI_GRAPH_NOT_EXIST	graph not exist
3	HIAI_GRAPH_MEMORY_POOL _NOT_EXISTED	memory pool is not existed

序号	错误码	错误码描述
4	HIAI_GRAPH_MALLOC_LARG ER	failed to malloc buffer due to the size larger than 128M
5	HIAI_MEMORY_POOL_UPDAT E_FAILED	failed to update memory pool
6	HIAI_GRAPH_SENDMSG_FAIL ED	hdc send msg failed
7	HIAI_GRAPH_MEMORY_POOL _INITED	memory pool has inited
8	HIAI_GRAPH_NO_MEMORY	no memory

使用说明

HIAIMemory::HIAI_DVPP_DMalloc是专门为了给DVPP分配内存而新增的一个接口,通过该接口申请的内存能够满足DVPP对内存的各种限制及要求。

如果您在旧版本中使用HIAIMemory::HIAI_DMalloc(1000, buffer, 1000, HIAI_MEMORY_HUGE_PAGE)方式为DVPP申请内存,则需要修改为使用HIAIMemory::HIAI_DVPP_DMalloc接口申请内存。关于HIAIMemory::HIAI_DMalloc接口,请参见5.4.1 HIAIMemory::HIAI_DMalloc。

使用范例:

```
// 使用HIAI_DVPP_DMalloc申请内存
uint32_t allocSize = 1000;
char* allocBuffer = nullptr;
HIAI_StatusT ret = hiai::HIAIMemory::HIAI_DVPP_DMalloc(allocSize, (void*&)allocBuffer);
if (ret != HIAI_OK || allocBuffer == nullptr) {
    // 分配失败,进行异常处理
}
```

5.4.5 HIAIMemory::HIAI_DVPP_DFree

用于释放内存, 该接口主要用于释放HIAIMemory::HIAI_DVPP_DMalloc接口分配的内存。该接口在ai_memory.h中定义。

函数格式

HIAI_StatusT HIAI_DVPP_DFree(void* dataBuffer)

参数说明

参数	说明	取值范围
dataBuffer	被释放的内存地址。	-

返回值

返回的部分错误码请参见"错误码示例"中的"错误码"列。

错误码示例

序号	错误码	错误码描述
1	HIAI_OK	running ok
2	HIAI_GRAPH_NOT_EXIST	graph not exist
3	HIAI_GRAPH_INVALID_VALUE	graph invalid value
4	HIAI_GRAPH_NOT_FIND_MEM ORY	can't find the memory

5.5 日志 (C++语言)

5.5.1 错误码注册

5.5.1.1 宏 HIAI_DEF_ERROR_CODE

注册错误码。该宏在status.h中定义。

本宏封装用到了以下函数:

```
static StatusFactory* StatusFactory::GetInstance();
void RegisterErrorNo(const uint32_t err, const std::string& desc);
```

函数格式

HIAI_DEF_ERROR_CODE(moduleId, logLevel, codeName, codeDesc)

参数说明

参数	说明	取值范围
moduleId	模块ID。	-
logLevel	错误级别。	-
codeName	错误码的名称。	-
codeDesc	错误码的描述,字符串。	-

返回值

无。

5.5.1.2 接口使用示例

[*] ***********************************	
* CopyRight (C) Hisilicon Co., Ltd.	
*	

```
Filename: user_def_errorcode.h
    Description: User defined Error Code
       Version: 1.0
Created: 2018-01-08 10:15:18
        Author:
       Revision: initial draft;
#ifndef USE DEF ERROR CODE H
#define USE_DEF_ERROR_CODE_H_
#include "hiaiengine/status.h"
#define USE DEFINE ERROR 0x6001 // 自定义日志模块ID, 0x6001为自定义值, 不能跟status. h文件中的
MODID_GRAPH等重复即可
enum
   HIAI_INVALID_INPUT_MSG_CODE = 0 // 错误码的名称, 0为自定义值
HIAI DEF ERROR CODE(USE DEFINE ERROR, HIAI ERROR, HIAI INVALID INPUT MSG, "invalid input message
pointer");
#endif
```

5.5.2 日志打印

通过HIAI_ENGINE_LOG来根据传入的参数调用不同的HIAI_ENGINE_LOG_IMPL函数。在调用格式化函数时,format中参数的类型与个数必须与实际参数类型一致。日志打印相关的接口在log.h中定义。

调用HIAI_ENGINE_LOG后,系统将日志记录日志文件中,日志文件在Host侧或开发者板的"/var"目录下。Device侧的日志被记录在文件名称以device-*id*开头的日志文件中,Host侧的日志被记录在文件名称以host-0开头的日志文件中。

5.5.2.1 日志打印格式 1

函数格式

```
#define HIAI_ENGINE_LOG(...) \
HIAI_ENGINE_LOG_IMPL(__FUNCTION__, __FILE__, __LINE__,
## VA ARGS )
```

void HIAI_ENGINE_LOG_IMPL(const char* funcPointer, const char* filePath, int lineNumber,

hiai::Graph* graph, const uint32_t errorCode, const char* format, ...);

参数	说明	取值范围
graph	Graph对象指针	-
errorCode	错误码	-
format	log描述	-
	format中的可变参数,根 据日志内容添加	-

调用示例

```
auto graph = Graph::Getinstance(1);
HIAI_ENGINE_LOG (graph.get(), HIAI_OK, "RUNNING OK");
```

5.5.2.2 日志打印格式 2

函数格式

```
#define HIAI_ENGINE_LOG(...) \
HIAI_ENGINE_LOG_IMPL(__FUNCTION__, __FILE__, __LINE__, ##__VA_ARGS__)
```

void HIAI_ENGINE_LOG_IMPL(const char* funcPointer, const char* filePath, int lineNumber,

hiai::Engine* engine, const uint32_t errorCode, const char* format, ...);

参数说明

参数	说明	取值范围
engine	Engine对象指针	-
errorCode	错误码	-
format	log描述	-
	format中的可变参数,根 据日志内容添加	-

调用示例

```
// 在HIAI_IMPL_ENGINE_PROCESS里调用此接口
HIAI_ENGINE_LOG (this, HIAI_OK, "RUNNING OK");
```

5.5.2.3 日志打印格式 3

函数格式

```
#define HIAI_ENGINE_LOG(...) \
HIAI_ENGINE_LOG_IMPL(_FUNCTION__, __FILE__, __LINE__, 
##__VA_ARGS__)

void HIAI_ENGINE_LOG_IMPL(const char* funcPointer, const char* filePath, int lineNumber, 
const uint32_t errorCode, const char* format, ...);
```

参数说明

参数	说明	取值范围
errorCode	错误码	-
format	log描述	-
	format中的可变参数,根 据日志内容添加	-

调用示例

HIAI_ENGINE_LOG (HIAI_OK, "RUNNING OK");

5.5.2.4 日志打印格式 4

函数格式

```
#define HIAI_ENGINE_LOG(...) \
HIAI_ENGINE_LOG_IMPL(_FUNCTION__, __FILE__, __LINE__, 
##__VA_ARGS__)
```

void HIAI_ENGINE_LOG_IMPL(const char* funcPointer, const char* filePath, int lineNumber,

const char* format, ...);

参数说明

参数	说明	取值范围
format	log描述	-
	format中的可变参数,根 据日志内容添加	-

调用示例

HIAI_ENGINE_LOG ("RUNNING OK");

5.5.2.5 日志打印格式 5

函数格式

```
#define HIAI_ENGINE_LOG(...) \
HIAI_ENGINE_LOG_IMPL(__FUNCTION__, __FILE__, __LINE__, ##__VA_ARGS__)
```

void HIAI_ENGINE_LOG_IMPL(const char* funcPointer, const char* filePath, int lineNumber,

const int32_t moudleID, hiai::Graph* graph, const uint32_t errorCode, const char* format, ...);

参数说明

参数	说明	取值范围
moudleID	模块名枚举ID	-
graph	Graph对象指针	-
errorCode	错误码	-
format	log描述	-
	format中的可变参数,根 据日志内容添加	-

调用示例

```
auto graph = Graph::Getinstance(1);
HIAI_ENGINE_LOG (MODID_OTHER, graph.get(), HIAI_OK, "RUNNING OK");
```

5.5.2.6 日志打印格式 6

函数格式

```
#define HIAI_ENGINE_LOG(...) \
HIAI_ENGINE_LOG_IMPL(_FUNCTION__, __FILE__, __LINE__, 
## VA ARGS )
```

void HIAI_ENGINE_LOG_IMPL(const char* funcPointer, const char* filePath, int lineNumber,

const int32_t moudleID,hiai::Engine* engine, const uint32_t errorCode, const char* format, ...);

参数	说明	取值范围
moudleID	模块名枚举ID	-
engine	Engine对象指针	-
errorCode	错误码	-
format	log描述	-

参数	说明	取值范围
	format中的可变参数,根 据日志内容添加	-

调用示例

// 在HIAI_IMPL_ENGINE_PROCESS里调用此接口 HIAI ENGINE LOG (MODID OTHER, this, HIAI OK, "RUNNING OK");

5.5.2.7 日志打印格式 7

函数格式

```
#define HIAI_ENGINE_LOG(...) \
HIAI_ENGINE_LOG_IMPL(__FUNCTION__, __FILE__, __LINE__, 
##__VA_ARGS__)
```

void HIAI_ENGINE_LOG_IMPL(const char* funcPointer, const char* filePath, int lineNumber,

const int32_t moudleID, const char* format, ...);

参数说明

参数	说明	取值范围
moudleID	模块名枚举ID	-
format	log描述	-
	format中的可变参数,根 据日志内容添加	-

调用示例

HIAI_ENGINE_LOG (MODID_OTHER, "RUNNING OK");

5.5.2.8 日志打印格式 8

函数格式

```
#define HIAI_ENGINE_LOG(...) \
HIAI_ENGINE_LOG_IMPL(__FUNCTION__, __FILE__, __LINE__, 
##__VA_ARGS__)
```

void HIAI_ENGINE_LOG_IMPL(const char* funcPointer, const char* filePath, int lineNumber,

const int32 t moudleID, const uint32 t errorCode, const char* format, ...);

参数说明

参数	说明	取值范围
moudleID	模块名枚举ID	-
errorCode	错误码	-
format	log描述	-
	format中的可变参数,根 据日志内容添加	-

调用示例

HIAI_ENGINE_LOG (MODID_OTHER, HIAI_OK, "RUNNING OK");

5.6 队列管理 MultiTypeQueue 接口(C++语言)

支持多种类型的消息队列保存。队列管理的相关接口在multitype_queue.h中定义。

5.6.1 MultiTypeQueue 构造函数

MultiTypeQueue构造函数。

函数格式

MultiTypeQueue(uint32_t queNum, uint32_t maxQueLen = 1024, uint32_t durationMs = 0);

参数说明

参数	说明	取值范围
queNum	指定队列个数。	-
maxQueLen	指定每个队列的最大长度。	-
durationMs	指定队列成员从入队到出队最大时间间隔。当某个成员从入队(Push)起,超过duration_ms时间后还未出队(Pop),则该成员会被自动删除。	-

返回值

无。

5.6.2 PushData

往指定队列插入数据。

函数格式

bool MultiTypeQueue::PushData(uint32_t qIndex, const std::shared_ptr<void>& dataPtr);

参数说明

参数	说明	取值范围
qIndex	队列编号	0 ~ queNum - 1
dataPtr	队列函数指针	-

返回值

成功则返回true,否则返回false(例如队列满等)。

5.6.3 FrontData

读取指定队列头部数据(并不从队列中删除)

函数格式

bool MultiTypeQueue::FrontData(uint32_t qIndex, std::shared_ptr<void>& dataPtr);

参数说明

参数	说明	取值范围
qIndex	队列编号	0 ~ queNum - 1
dataPtr	队列函数指针	-

返回值

成功则返回true, 否则返回false (例如队列为空等)。

5.6.4 PopData

读取指定队列头部数据,并从队列中删除。

函数格式

bool MultiTypeQueue::PopData(uint32 t qIndex, std::shared ptr<void>& dataPtr);

参数说明

参数	说明	取值范围
qIndex	队列编号	0 ~ que_num-1
dataPtr	队列函数指针	-

返回值

成功则返回true, 否则返回false (例如队列为空等)。

5.6.5 PopAllData

PopAllData读取所有队列头部数据。仅当所有队列头部都有数据时才读取成功,并删除 队列头部数据。否则,返回失败但并不删除任何数据。

函数格式

template<typename T1>

bool MultiTypeQueue::PopAllData(std::shared_ptr<T1>& arg1);

template<typename T1, typename T2>

bool MultiTypeQueue::PopAllData(std::shared_ptr<T1>& arg1, std::shared_ptr<T2>& arg2);

•••••

template<typename T1, typename T2, typename T3, typename T4, typename T5, typename T6, typename T7, typename T8, typename T9, typename T10, typename T11, typename T12, typename T13, typename T14, typename T15, typename T16>

bool MultiTypeQueue::PopAllData(std::shared_ptr<T1>& arg1,

std::shared ptr<T2>& arg2, std::shared ptr<T3>& arg3,

std::shared_ptr<T4>& arg4, std::shared_ptr<T5>& arg5,

std::shared ptr<T6>& arg6, std::shared ptr<T7>& arg7,

std::shared_ptr<T8>& arg8, std::shared_ptr<T9>& arg9,

std::shared ptr<T16>& arg16)

参数	说明	取值范围
arg1	队列函数指针	-
	队列函数指针	-

参数	说明	取值范围
arg16	队列函数指针	-

返回值

仅当所有队列头部都有数据时才读取成功并返回true,否则返回false。

5.6.6 接口使用示例

见3.7 调用示例。

5.7 事件注册接口(C++语言)

5.7.1 Graph::RegisterEventHandle

用户调用RegisterEventHandle接口订阅感兴趣的事件,当前支持Host-Device断开连接事件,用户订阅该事件,当连接断开时,用户可接收到该事件,并在注册的回调函数中处理断开逻辑(如停掉主程序的等待)。该接口在graph.h中定义。

函数格式

HIAI_StatusT Graph::RegisterEventHandle(const HIAIEvent& event, const std::function<HIAI_StatusT(void)>&callBack)

参数说明

参数	说明	取值范围
event	订阅的事件	typedef enum { HIAI_DEVICE_DISCONNECT_E VENT // 断开消息 }HIAIEvent;
callBack	事件回调函数	用户订阅的回调函数

返回值

成功返回HIAI_OK, 否则为失败。

5.8 其他(C++语言)

5.8.1 DataRecvInterface::RecvData

用户需要定义一个DataRecvInterface类的子类,初始化一个子类的对象,作为数据接收回调函数,传入2.4.1 Graph::SetDataRecvFunctor或3.5

Engine::SetDataRecvFunctor。同时,用户需要在子类中重载实现RecvData接口。该接口在data recv interface.h中定义。

函数格式

virtual HIAI_StatusT DataRecvInterface::RecvData(const std::shared_ptr<void>& message)

参数说明

参数	说明	取值范围
message	回调消息	-

返回值

返回的部分错误码请参见"错误码示例"中的"错误码"列。

错误码示例

序号	错误码	错误码描述
1	HIAI_OK	running ok

5.8.2 Graph::SetPublicKeyForSignatureEncryption

该接口用于设置公钥,该公钥用于对Device侧Matrix要加载的*.so文件进行数字签名验证。需要在创建Graph之前调用该接口。该接口在graph.h中定义。

数字签名采用的算法为SHA256withRSA,本组件只提供数字签名的验证功能,加签的功能需要由用户自己基于该算法进行对需要的SO进行签名,签名文件在host的存放路径需要与SO在统一目录,并且签名文件的名称为SO名称+ ".signature",比如SO的名称为"libhosttodevice.so",签名文件的名称应为"libhosttodevice.so.signature"。

⚠注意

如果不调用Graph::SetPublicKeyForSignatureEncryption接口或者调用Graph::SetPublicKeyForSignatureEncryption接口的返回值不是HIAI_OK,则Matrix无法设置公钥,Device侧不会启用签名校验机制,因此无法通过数字签名识别出可能被篡改的*.so文件。

函数格式

 $static\ HIAI_StatusT\ Graph:: SetPublicKeyForSignatureEncryption (const\ std:: string\&publicKey)$

参数说明

参数	说明	取值范围
publicKey	数字签名验证需要的公钥	-

返回值

返回的部分错误码请参见"错误码示例"中的"错误码"列。

错误码示例

序号	错误码	错误码描述
1	HIAI_OK	running ok

std::string publicKey =

"-----BEGIN RSA PUBLIC KEY-----\n"

"MIIBCAKCAQEAt+kdmu8CdViw2xHlh/JWX01/0AitDgYGd+9RadULGZmjOtt/UQLv\n"

"EYPZaXC8E0a9e97kqfg/ZHinu04XG5RhXv2J0kwkhuBAeCCllefvrQy50BRqdLYq\n"

"EZe8wU00adSMDsLbPL52b+Nd09/zX+MUzvsBzitWJbiH96s4xCiEERX87/uQfr6F\n"

"1HLtrNtooeF2Vmx0m3n0yzh4kcSouLgb/00+v0I+fnR+CTNG95IvzDN0YLWD6ZkL\n"

"c7JyIYFZAOCNx9S1Phqbrfj0V5XSG3g3CW0TopUDfHyhAgZt5vACMpeDDx+89tg2\n"

"RfT4M9DH/qKzkL10URvsMShRMD6/PwzsPwIBAw==\n"

"----END RSA PUBLIC KEY-----";

auto ret = Graph::SetPublicKeyForSignatureEncryption(publicKey);

6 低功耗接口(C++语言)

- 6.1 函数接口
- 6.2 数据类型
- 6.3 接口使用示例

6.1 函数接口

6.1.1 PowerState::SubscribePowerStateEvent

用户注册低功耗回调函数。该接口在power state.h中定义。

函数格式

HIAI_StatusT SubscribePowerStateEvent(PowerStateNotifyCallbackT notifyCallback)

参数说明

参数	说明	取值范围
notifyCallback	用户希望注册的回调函 数。	-

返回值

返回的部分错误码请参见"错误码示例"中的"错误码"列。

错误码示例

序号	错误码	错误码描述
1	HIAI_OK	running ok

序号	错误码	错误码描述
2	HIAI_POWER_STATE_CALLBA CK_ERROR	App callback function is nullptr

6.1.2 PowerState::UnsubscribePowerStateEvent

用户注销先前注册的低功耗回调函数。该接口在power_state.h中定义。

函数格式

HIAI StatusT UnsubscribePowerStateEvent(PowerStateNotifyCallbackT notifyCallback)

参数说明

参数	说明	取值范围
notifyCallback	用户希望注销的回调函数	-

返回值

返回的部分错误码请参见"错误码示例"中的"错误码"列。

错误码示例

序号	错误码	错误码描述
1	HIAI_OK	running ok
2	HIAI_POWER_STATE_CALLBA CK_ERROR	App callback function is nullptr

6.2 数据类型

6.2.1 枚举: PowerState::DEVICE_POWER_STATE

```
enum DEVICE_POWER_STATE {
    DEVICE_POWER_STATE_SO = 0,
    DEVICE_POWER_STATE_SUSPEND,
    DEVICE_POWER_STATE_DO,
    DEVICE_POWER_STATE_D3,

DEVICE_POWER_STATE_D3,

DEVICE_POWER_STATE_ENABLE,
    DEVICE_POWER_STATE_DISABLE,
    DEVICE_POWER_STATE_DISABLE,
    DEVICE_POWER_STATE_MAX

};
```

6.2.2 PowerState::PowerStateNotifyCallbackT

typedef HIAI_StatusT (*PowerStateNotifyCallbackT)(uint32_t, DEVICE_POWER_STATE);

6.3 接口使用示例

```
用户写回调函数
HIAI_StatusT AppPwrChangeProc(uint32_t deviceId, hiai::PowerState::DEVICE_POWER_STATE npuState) {
    // 此处用户可以进行若干处理
    // ……
    return HIAI_OK;
}
注册回调
hiai::PowerState::GetInstance()->SubscribePowerStateEvent(AppPwrChangeProc);
注销回调
hiai::PowerState::GetInstance()->UnsubscribePowerStateEvent(AppPwrChangeProc);
```

- 7.1 Matrix数据类型
- 7.2 Matrix已经注册的数据结构
- 7.3 示例

7.1 Matrix 数据类型

7.1.1 Protobuffer 数据类型

关于Protobuffer的使用,请参见https://developers.google.com/protocol-buffers/docs/reference/proto3-spec。

Matrix用到的数据类型Protobuffer格式:

```
ai_types.proto
```

```
syntax = "proto3"; IAITensor
package hiai;
// Tensor参数描述
message AITensorParaDescription
   string name = 1; // 参数名称
   string type = 2; // 参数类型
string value = 3; // 参数值
string desc = 4; // 参数描述
   repeated AITensorParaDescription sub_paras = 5; // 子参数列表
// Tensor描述
message AITensorDescription
   string name = 1; // Tensor名称
   string type = 2; // Tensor类型
   repeated string compatible_type = 3; // 指定可以兼容的所有父类类型
   repeated AITensorParaDescription paras = 4; // 参数列表
// Tensor描述列表
message AITensorDescriptionList
   repeated AITensorDescription tensor descs = 1; // tensor列表
```

```
// 通用配置项
// 如果存在sub items,则当前节点为父节点,value值无效
message AIConfigItem
   string name = 1; // 配置项名称 string value = 2; // 配置项值,此处配置模型文件所在的路径时,包含文件名(名称中只允许有数字、
字母、下划线和点号),可以将参数值配置为单个模型文件的路径(如:./test_data/model/resnet18.om);也
可以将模型文件打包成tar包后,将参数值配置为tar包所在的路径(如:./test_data/model/resnet18.tar)。
若存在多个AIConfigItem,在配置tar包路径时,不允许在同一个目录下同时存在名称相同但格式不同的文件,例
如./test_data/model/test、./test_data/model/test.tar。创建Graph接口包含Graph::CreateGraph或
HIAI CreateGraph.
   repeated AIConfigItem sub items = 3; // 配置子项
// nnnode/api运行时配置参数
message AIConfig
   repeated AIConfigItem items = 1; // 配置项列表
// 模型描述
message AIModelDescription
   string name = 1; // 模型名称
   int32 type = 2; // 模型类型, 当前仅支持DAVINCI_OFFLINE_MODEL类型, 值为0
   string version = 3; // 模型版本int32 size = 4; // 模型大小
   string path = 5; // 模型路径
   string key = 8; // 模型秘钥
   bytes data = 100; // 模型数据
   repeated AITensorDescription inputs = 6; // 输入Tensor描述
   repeated AITensorDescription outputs = 7; // 输出Tensor描述
// NNNode描述
message AINNNodeDescription
   string name = 1; // NNNode名称
   string desc = 2; // NNNode描述
   bool isPreAllocateOutputMem = 3; // 是否预分配输出内存
   AIConfig config = 4; // 配置参数
   repeated AIModelDescription model_list = 5; // NNNode需要的模型列表
   repeated AITensorDescription inputs = 6; // 输入Tensor描述 repeated AITensorDescription outputs = 7; // 输出Tensor描述
   bool need_verify = 8; // 串联时是否需要校验Tensor匹配
   repeated string ignored_check_aitensor = 9; // 指定在串联时不与Inputs的Tensor做匹配校验的Tensor
列表
};
// NNNode描述列表
message AINNNodeDescriptionList
   repeated AINNNodeDescription nnnode descs = 1; // NNNode列表
// API描述
message AIAPIDescription
   string name = 1; // API名称
   string desc = 2; // API描述
   bool isPreAllocateOutputMem = 3; // 是否预分配输出内存
   AIConfig config = 4; // 配置参数
   repeated AITensorDescription inputs = 5; // 输入Tensor描述
   repeated AITensorDescription outputs = 6; // 输出Tensor描述
   bool need_verify = 7; // 串联时是否需要校验Tensor匹配
   repeated string ignored_check_aitensor = 8; // 指定在串联时不与Inputs的Tensor做匹配校验的Tensor
列表
```

```
};
// API描述列表
message AIAPIDescriptionList
   repeated AIAPIDescription api_descs = 1; // API列表
// AI Operation 描述
message AIOPDescription
   enum OP_Type
       AI\_API = 0;
       AI_NNNODE = 1;
   OP_Type type = 1;
   AINNNodeDescription nnnode_desc = 2;
   AIAPIDescription api desc = 3;
// AI Operation 描述列表
message AIOPDescriptionList
   repeated AIOPDescription op_descs = 1; // AI Operation列表
// IDE传给Matrix的接口
message NodeDesc
  string name=1; // IAINNNode或ALG_API 名字
  AIConfig config=2; //IAINNNode或ALG_API 需要的初始化参数
  repeated AIModelDescription mode_desc=3; //IAINNNode 需要的初始化参数
message EngineDesc
   enum RunSide
       DEVICE=0;
       HOST=1;
   enum EngineType
       NORMAL=0;
       SOURCE=1;
       DEST=2;
   uint32 id =1; //Engine ID (节点)
   EngineType type=2; //
   string name =3; //Engine 节点名字
   repeated string so_name=4; //需要拷贝的所有动态库so文件名列表,会先按照用户配置的顺序先进行加
载,加载失败再按照其他的加载顺序进行尝试加载
   RunSide side=5; //部署在host侧还是Device侧
   int32 priority=6; //节点优先级
   uint32 instance_cnt=7; //实例个数(相当于线程个数)
   repeated uint32 next_node_id=8; //下一个节点列表
   bool user_input_cb=9; //IDE 可以忽略
bool user_output_cb=10; //IDE 可以忽略
   repeated NodeDesc oper=11; //Matrix Node 列表
message GraphInitDesc
   int32 priority=1; //Graph 整个进程的优先级
   //Runside side = 2; //当前假定Graph 配置在host侧,不可配置
```

```
message GeneralFileBuffer
{
   bytes raw_data = 1;
   string file_name = 2;
}
```

graph_config.proto

```
syntax = "proto3";
import "ai types.proto";
package hiai;
message DumpDef
   bytes model_name = 1;
                        //模型名称
   bytes is_dump_all = 2;
                     //是否打开Dump开关,用于Dump模型中每层算子的信息
   repeated bytes layer = 3; //需要Dump模型中哪些层的算子信息
message OAMConfigDef
   repeated DumpDef dump_list = 1;
   bytes dump_path = 2; //dump文件存放的路径
message EngineConfig
   enum RunSide
      DEVICE=0;
               //运行在DEVICE侧
               //运行在host侧
      HOST=1:
   uint32 id =1; //Engine ID (节点)
   string engine_name =2; //Engine 节点名字
   repeated string so_name=3; //需要拷贝的所有动态库so文件名列表,会先按照用户配置的顺序先进行加
载,加载失败再按照其他的加载顺序进行尝试加载
   RunSide side=4; //部署在host侧还是Device侧
   uint32 thread_num = 5;//线程数量。多路解码时,则该参数值推荐设置为1,如果thread_num值大于1,则
线程之间的解码无法保证顺序。
   uint32 thread_priority = 6;//线程优先级
   uint32 queue_size = 7;//队列大小
   AIConfig ai_config = 8; //Aiconfig配置文件
   repeated AIModelDescription ai_model = 9; //AIModelDescription
   repeated string internal so name=10; //不需要拷贝的所有动态库so文件名列表
   uint32 wait_inputdata_max_time = 12; //当前已经收到数据后等待下一个数据的最大超时时间,单位为毫
   uint32 holdModelFileFlag = 13; //是否保留本engine的模型文件,0不保留,非0保留
   OAMConfigDef oam_config = 14; // OAM Config配置
   bool is_repeat_timeout_flag = 15; //engine未收到数据是否重复做超时处理, 0不重复, 1重复。例如多
路数据分析时,数据量较大的情况下,建议将该参数值设置为1,做重复超时处理。与wait_inputdata_max_time
参数配合使用,如果is_repeat_timeout_flag设置为0,则通过wait_inputdata_max_time参数设置单次超时时
长;如果is_repeat_timeout_flag设置为1,则通过wait_inputdata_max_time参数设置周期超时时长。
message ConnectConfig
  uint32 src_engine_id=1; //发送端的EngineID
uint32 src_port_id = 2; //发送端的PortID
   uint32 target_graph_id=3; //接收端的GraphID
   uint32 target_engine_id=4; //接收端的PortID
   uint32 target_port_id=5; //接收端的PortID
   bool receive_memory_without_dvpp=6; //参数默认值为0,表示Deivce上运行的目标Engine的接收内存需要
满足4G地址空间限制;若将参数值配置为1,表示Device上运行的目标Engine的接收内存不用满足4G地址空间限
制。用户可在Graph配置文件中将所有在Device上运行的目标Engine的receive memory without dvpp参数(在
connects属性下)配置为1,也可以将所有connects属性下的receive_memory_without_dvpp参数都配置为1(因为
在Host上运行的目标Engine即使配置了该参数也无影响),这样Matrix接收内出池就不使用4G地址空间限制,从
而提高系统内存使用率。当前DVPP中,VPC、JPEGE、JPEGD、PNGD功能的输入内存需要满足4G地址空间限制,
VDEC、VENC功能的输入内存可以满足或不满足4G地址空间限制。
message GraphConfig
```

```
uint32 graph_id = 1; //GraphID
   int32 priority = 2; //优先级
   string device_id = 3;//设备id配置,例如"0"
    repeated EngineConfig engines = 3; //所有的engine。多路解码时推荐配置多个Engine,一个Engine对应
  个线程,如果一个Engine对应多个线程,解码时无法保证顺序。
   repeated ConnectConfig connects = 4; //连接方式
message GraphConfigList
   repeated GraphConfig graphs = 1; //单个Graph的配置参数
message ProfileConfig
   string matrix_profiling = 1; //Matrix的性能统计开关, "on"为开
   string ome_profiling = 2; //OME的性能统计开关, "on"为开 string cce_profiling =3; //CCE的性能统计开关, "on"为开
   string runtime_profiling = 4; //RUNTIME的性能统计开关, "on"为开string PROFILER_TARGET = 5; //透传Profiling的参数
   string PROFILER JOBCTX = 6;
   string src_path = 7;
   string dest_path = 8;
   string runtime_config = 9;
    string RTS_PATH = 10;
   string profiler_jobctx_path = 11;
    string profiler_target_path = 12;
message GraphUpdateConfig
   GraphConfig updataGraphConfig = 1;
   repeated uint32 del_engine_ids = 2;
   repeated ConnectConfig del_connects = 3;
```

7.1.2 Matrix 自定义数据类型

- HIAI StatusT: 用于定义返回码的数据类型为uint32 t。
- EnginePortID: Matrix中用于定义端口的结构体。

```
struct EnginePortID {
    uint32_t graph_id;
    uint32_t engine_id;
    uint32_t port_id;
};
```

7.2 Matrix 已经注册的数据结构

Matrix中传输的数据结构需要先进行注册,以下数据结构Matrix已经进行了注册,可直接使用。

```
namespace hiai {
                                             ==>>>>>> message type
    // message name <<<<<==
    // name:"BatchInfo"
                                                       type:BatchInfo
    // name:"FrameInfo"
                                                       type:FrameInfo
    // name:"IMAGEFORMAT"
                                                       type: IMAGEFORMAT
    // name:"Angle"
                                                       type:Angle
    // name:"Point2D"
                                                       type:Point2D
    // name:"Point3D"
                                                       type:Point3D
    // name:"ROICube"
                                                       type:ROICube
    // name:"RLECode"
// name:"IDTreeNode"
                                                       type: RLECode
                                                       type: IDTreeNode
    // name:"IDTreePara"
                                                       type:IDTreePara
    // name:"BatchIDTreePara"
                                                       type:BatchIDTreePara
    // name:"RetrievalResult"
                                                       type:RetrievalResult
```

```
// name:"RetrievalResultTopN"
                                                      type:RetrievalResultTopN
    // name:"RetrievalResultPara"
                                                      type:RetrievalResultPara
    // name:"RawDataBuffer"
                                                      type:RawDataBuffer
    // name:"BatchRawDataBuffer"
// name:"string"
                                                      type:BatchRawDataBuffer
                                                      type:string
    // name:"vector_uint8_t"
// name:"vector_float"
                                                      type:vector<uint8_t>
                                                      type:vector<float>
    // name:"ImageData_uint8_t"
                                                      type:ImageData<uint8_t>
    // name:"ImageData_float"
                                                      type:ImageData<float>
    // name:"ImagePara_uint8_t"
                                                      type:ImagePara<uint8_t>
    // name:"ImagePara_float"
                                                      type:ImagePara<float>
    // name: "BatchImagePara uint8 t"
                                                      type:BatchImagePara<uint8 t>
    // name: "BatchImagePara_float"
                                                      type:BatchImagePara<float>
    // name:"Line_uint8_t"
                                                      type:Line<uint8_t>
    // name:"Line_float"
                                                      type:Line(float)
                                                      type:Rectangle<uint8_t>
    // name:"Rectangle_uint8_t"
    // name:"Rectangle_float'
                                                      type:Rectangle<float>
    // name:"Polygon_uint8_t"
                                                      type:Polygon(uint8_t)
    // name:"Polygon_float"
                                                      type:Polygon(float)
    // name:"MaskMatrix_uint8_t"
                                                      type:MaskMatrix<uint8_t>
    // name:"MaskMatrix_float"
                                                      type:MaskMatrix<float>
    // name: "ObjectLocation_uint8_t_uint8_t"
                                                      type:ObjectLocation(uint8_t, uint8_t)
    // name: "ObjectLocation_float_float"
                                                      type:ObjectLocation(float, uint8_t)
    // name: "DetectedObjectPara uint8 t uint8 t"
                                                      type:DetectedObjectPara(uint8_t, uint8_t)
    // name: "DetectedObjectPara_float_float"
                                                      type:DetectedObjectPara(float, float)
    // name:"BatchDetectedObjectPara_uint8_t_uint8_t"
type:BatchDetectedObjectPara<uint8_t, uint8_t>
    // name:"BatchDetectedObjectPara_float_float"
type:BatchDetectedObjectPara<float, float>
    // name:"BatchDetectedObjectPara_Rectangle_Point2D_int32_t"
type:BatchDetectedObjectPara<Rectangle<Point2D>, int32_t>
    // name:"BatchDetectedObjectPara_Rectangle_Point2D_float"
type:BatchDetectedObjectPara<Rectangle<Point2D>, float>
    // name:"RegionImage_uint8_t"
                                                      type:RegionImage(uint8_t)
    // name:"RegionImage_float"
                                                      type:RegionImage<float>
    // name:"RegionImagePara_uint8_t"
                                                      type:RegionImagePara<uint8_t>
    // name:"RegionImagePara_float"
                                                      type:RegionImagePara<float>
    // name: "BatchRegionImagePara_uint8_t"
                                                      type:BatchRegionImagePara<uint8_t>
    // name: "BatchRegionImagePara_float"
                                                      type:BatchRegionImagePara<float>
```

```
// name:"Attribute_uint8_t"
                                                 type:Attribute(uint8_t)
// name:"Attribute_float"
                                                 type:Attribute<float>
// name:"AttributeTopN_uint8_t"
                                                 type:AttributeTopN<uint8_t>
// name:"AttributeTopN_float"
                                                 type:AttributeTopN<float>
// name:"AttributeVec_uint8_t"
                                                 type:AttributeVec<uint8 t>
// name: "AttributeVec_float"
                                                 type:AttributeVec<float>
// name: "AttributeResultPara_uint8_t"
                                                 type: AttributeVec<uint8 t>
// name:"AttributeResultPara_float"
                                                 type:AttributeVec<float>
// name: "BatchAttributeResultPara_uint8_t"
                                                 type:AttributeVec<uint8_t>
// name: "BatchAttributeResultPara_float"
                                                 type:AttributeVec<float>
// name:"Feature_uint8_t"
                                                 type:AttributeVec<uint8_t>
// name: "Feature_float"
                                                 type:AttributeVec<float>
// name:"FeatureList_uint8_t"
                                                 type:FeatureList<uint8 t>
// name: "FeatureList_float"
                                                 type:FeatureList<float>
// name: "FeatureVec_uint8_t"
                                                 type:FeatureVec<uint8_t>
// name: "FeatureVec float"
                                                 type:FeatureVec<float>
// name:"FeatureResultPara_uint8_t"
                                                 type:FeatureResultPara<uint8_t>
// name: "FeatureResultPara_float"
                                                 type:FeatureResultPara(float)
// name: "BatchFeatureResultPara_uint8_t"
                                                 type:BatchFeatureResultPara(uint8 t)
// name: "BatchFeatureResultPara_float"
                                                 type:BatchFeatureResultPara<float>
// name: "FeatureRecord_uint8_t"
                                                 type:FeatureRecord(uint8_t)
// name: "FeatureRecord_float"
                                                 type:FeatureRecord<float>
// name:"RetrievalSet_uint8_t_uint8_t"
                                                 type:RetrievalSet<uint8_t, uint8_t>
// name: "RetrievalSet float float"
                                                 type:RetrievalSet<float, float>
// name:"EvaluationResult_uint8_t"
                                                 type:EvaluationResult<uint8_t>
// name: "EvaluationResult_float"
                                                 type:EvaluationResult<float>
// name: "EvaluationResultVec_uint8_t"
                                                 type:EvaluationResultVec<uint8_t>
                                                 type:EvaluationResultVec<float>
// name: "EvaluationResultVec_float"
// name: "EvaluationResultPara_uint8_t"
                                                 type:EvaluationResultPara<uint8_t>
// name: "EvaluationResultPara_float"
                                                 type:EvaluationResultPara(float)
// name:"BatchEvaluationResultPara_uint8_t"
                                                 type:BatchEvaluationResultPara<uint8 t>
// name: "BatchEvaluationResultPara_float"
                                                 type:BatchEvaluationResultPara(float)
// name:"Classification_uint8_t"
                                                 type:Classification(uint8_t)
// name:"Classification_float"
                                                 type:Classification(float)
// name:"ClassificationTopN_uint8_t"
                                                 type:ClassificationTopN(uint8_t)
```

```
// name: "ClassificationTopN_float"
                                            type:ClassificationTopN<float>
// name: "ClassificationVec uint8 t"
                                            type:ClassificationVec<uint8 t>
// name: "ClassificationVec float"
                                            type:ClassificationVec<float>
// name: "ClassificationResultPara_uint8_t"
                                            type:ClassificationResultPara(uint8_t)
// name:"ClassificationResultPara_float"
                                            type:ClassificationResultPara(float)
// name: "BatchClassificationResultPara_uint8_t" type:BatchClassificationResultPara<uint8_t> // name: "BatchClassificationResultPara_float" type:BatchClassificationResultPara<float>
batch信息
                             struct BatchInfo {
   bool is first = false;
                                      // 是否为第一个batch
   bool is_lirst = lalse,
bool is_last = false;
uint32_t batch_size = 0;
uint32_t max_batch_size = 0;
                                    // 是否为最后一个batch
// 当前batch的实际大小
// batch预设大小(最大容量)
   uint32_t batch_ID = 0;
uint32_t channel ID = 0;
                                    // 当前batch ID号
                                   // 处理当前batch的通道ID号
// 处理器计算流ID号
   uint32 t channel ID = 0;
   uint32_t processor_stream_ID = 0;
   std::vector(uint32 t) frame ID;
                                       // batch中图像帧ID号
   std::vector<uint32 t> source ID;
                                       // batch中图像源ID号
                                       // batch中图像的时间戳
   std::vector<uint64_t> timestamp;
};
template < class Archive>
void serialize(Archive& ar, BatchInfo& data) {
   ar(data.is_first, data.is_last, data.batch_size,
       data.max_batch_size, data.batch_ID, data.channel_ID,
       data.processor_stream_ID, data.frame_ID, data.source_ID,
       data.timestamp);
struct FrameInfo {
   bool is_first = false;
                                      // 是否为第一个frame
   bool is_last = false; // 是否为最后一个frame
uint32_t channel_ID = 0; // 处理当前frame的通道ID号
   uint32_t processor_stream_ID = 0;
                                     // 处理器计算流ID号
   uint32_t frame_ID = 0;
uint32_t source_ID = 0;
                                     // 图像帧ID号
                                     // 图像源ID号
   uint64_t timestamp = 0;
                                     // 图像的时间戳
};
template < class Archive>
void serialize(Archive& ar, FrameInfo& data) {
   ar(data.is_first, data.is_last, data.channel_ID,
       data.processor_stream_ID, data.frame_ID, data.source_ID,
       data.timestamp);
```

```
// 图像格式
enum IMAGEFORMAT {
   RGB565,
                    // Red 15:11, Green 10:5, Blue 4:0
   BGR565,
                    // Blue 15:11, Green 10:5, Red 4:0
    RGB888,
                        Red 24:16, Green 15:8, Blue 7:0
                    // Blue 24:16, Green 15:8, Red 7:0
   BGR888.
   BGRA8888,
                    // Blue 31:24, Green 23:16, Red 15:8, Alpha 7:0
   ARGB8888,
                    // Alpha 31:24, Red 23:16, Green 15:8, Blue 7:0
   RGBX8888,
   XRGB8888,
    YUV420Planar,
                    // I420
    YVU420Planar,
                    // YV12
   YIIV420SP.
                    // NV12
   YVU420SP,
                    // NV21
                    // YUV420 Interleaved
// YVU420 Interleaved
   YUV420Packed,
   YVU420Packed,
   YUV422Planar,
                    // Three arrays Y, U, V.
   YVU422Planar,
    YUYVPacked,
                    // 422 packed per payload in planar slices
   YVYUPacked,
                    // 422 packed
   UYVYPacket,
                    // 422 packed
    VYUYPacket,
                    // 422 packed
                    // 422 packed
   YUV422SP.
   YVU422SP,
   YUV444Interleaved, // Each pixel contains equal parts YUV
   Y8,
   Y16.
   RAW
};
enum OBJECTTYPE {
   OT VEHICLE,
   OT HUMAN,
   OT_NON_MOTOR,
   OT_FACE,
   OT_FACE_BODY,
   OT_PLATE
};
template<class T> // T: uint8_t float
struct ImageData {
    IMAGEFORMAT format; // 图像格式
                              // 图像宽
   uint32_t width = 0;
   uint32 t height = 0;
                             // 图像高
   uint32\_t channel = 0;
                             // 图像通道数
   uint32_t depth = 0;
                             // 位深
   uint32_t height_step = 0; // 对齐高度
   uint32_t width_step = 0; // 对齐宽度
                             // 数据大小 (Byte)
   uint32_t size = 0;
    std::shared_ptr<T> data; // 数据指针
template < class Archive, class T>
void serialize(Archive& ar, ImageData<T>& data) {
   ar (data. format, data. width, data. height, data. channel,
        data.depth, data.height_step, data.width_step, data.size);
    if (data.size > 0 && data.data.get() == nullptr)
       data.data.reset(new(std::nothrow) T[data.size]);
   ar(cereal::binary_data(data.data.get(), data.size * sizeof(T)));
```

```
// 参数
template<class T>
struct ImagePara {
   FrameInfo f_info;
                         // frame信息
                          // 图像
   ImageData<T> img;
};
template \small{\langle class~Archive,~class~T \rangle}
void serialize(Archive& ar, ImagePara<T>& data) {
   ar(data.f_info, data.img);
// 参数
template<class T>
struct BatchImagePara {
   BatchInfo b info;
                                // batch信息
   std::vector<ImageData<T> > v_img; // batch中的图像
};
template\langle class \ Archive, \ class \ T \rangle
void serialize(Archive& ar, BatchImagePara<T>& data) {
   ar(data.b_info, data.v_img);
// 角度
typedef struct {
   float x;
   float y;
   float z;
}Angle;
template <class Archive>
void serialize(Archive& ar, Angle& data) {
   ar(data.x, data.y, data.z);
// 2D点
struct Point2D {
   int32_t x;
   int32_t y;
template <class Archive>
void serialize(Archive& ar, Point2D& data) {
   ar(data.x, data.y);
struct RoiPolygon {
                              //多边形顶点数
   uint32_t
                uiPtNum;
   std::vector<Point2D> astPts;
                              //多个检索结果(TopN)
};
```

```
template<class Archive>
    void serialize(Archive &ar, RoiPolygon &data) {
        ar(data.uiPtNum, data.astPts);
    struct ArgsRoiPolygon {
        RoiPolygon stRoiPolygon; // 配置项Roi
        uint32_t iMinFace; // 最小人脸26*26
        uint32_t iMaxFace; // 最大人脸300*300
uint32_t imgWidth; // 视频帧图像宽
        uint32_t imgHeight; // 视频帧图像高
   };
    template <class Archive>
    void serialize(Archive &ar, ArgsRoiPolygon &data) {
        ar(data.\,stRoiPolygon,\,\,data.\,iMinFace,\,\,data.\,iMaxFace,\,\,data.\,imgWidth,\,\,data.\,imgHeight,
data.iMinFace);
    // 3D点
    struct Point3D {
        int32\_t x;
        int32_t y;
        int32_t z;
   };
    template < class Archive >
    void serialize(Archive& ar, Point3D& data) {
        ar(data.x, data.y, data.z);
    // 2D或3D空间上的线
    template<class T> // T: Point2D Point3D
    struct Line {
        T start;
        T end;
   };
    template <class Archive, class T>
    void serialize(Archive& ar, Line<T>& data) {
        ar (data. start, data. end);
    // 2D或3D空间上的矩形平面
    template \verb|<| class T> | // T: Point2D | Point3D|
    struct Rectangle {
        T anchor_lt;
        T anchor_rb;
   };
    template <class Archive, class T>
    void serialize(Archive& ar, Rectangle<T>& data) {
        ar(data.anchor_lt, data.anchor_rb);
    // 2D或3D空间上的多边形平面
    template \verb|<| class T> | // T: Point2D | Point3D|
    struct Polygon {
        std::vector<T> anchors;
```

```
template<class Archive, class T>
void serialize(Archive& ar, Polygon<T>& data) {
    ar (data. anchors);
// 立方体
struct ROICube {
    Point3D anchor;
    uint32_t length;
    uint32_t width;
   uint32_t height;
};
template <class Archive>
void serialize(Archive& ar, ROICube& data) {
    ar(data.anchor, data.length, data.width, data.height);
// 掩膜矩阵
template(class T)
                   // T: int32_t float etc.
struct MaskMatrix {
   uint32_t cols;
    uint32_t rows;
    std::shared_ptr<T> data;
};
template <class Archive, class T>
void serialize(Archive& ar, MaskMatrix<T>& data) {
    ar (data. cols, data. rows);
    if (data.cols*data.rows > 0 && data.data.get() == nullptr) {
       data.data.reset(new(std::nothrow) T[data.cols*data.rows]);
    ar(cereal::binary_data(data.data.get(),
       data.cols * data.rows * sizeof(T)));
// RLE编码
struct RLECode {
    uint32_t len;
    uint32 t cols;
   uint32_t rows;
    std::shared_ptr<uint32_t> data;
};
template <class Archive>
void serialize (Archive& ar, RLECode& data) {
    ar(data.len, data.cols, data.rows);
    if (data.len > 0 && data.data.get() == nullptr) {
       data.data.reset(new(std::nothrow) uint32_t[data.len]);
    ar(cereal::binary_data(data.data.get(),
       data.len * sizeof(uint32_t)));
}
template < class T1, class T2>
                               // T1: Point2D Rectangle RLECode etc.
                                // T2: int8_t float etc.
struct ObjectLocation {
    std::vector<uint32_t> v_obj_id;
                                    // 目标范围描述
    std::vector<T1> range;
    std::vector<Angle> angle;
                                    // 角度信息
                                    // 置信度
    std::vector<T2> confidence;
    std::vector<uint32_t> label; // 目标类型标签
```

```
};
template<class Archive, class T1, class T2>
void serialize(Archive& ar, ObjectLocation<T1, T2>& data) {
   ar(data.v_obj_id, data.range, data.angle,
      data.confidence, data.label);
// 参数
template <class T1, class T2>
struct DetectedObjectPara {
                                      // frame信息
   FrameInfo f_info;
   ObjectLocation<T1, T2> location;
                                      // 多个目标位置
};
template \langle class Archive, class T1, class T2\rangle
void serialize(Archive& ar, DetectedObjectPara<T1, T2>& data) {
   ar(data.f_info, data.location);
// 参数
template<class T1, class T2>  
struct BatchDetectedObjectPara {
    // batch信息
   BatchInfo b_info;
   // 对应每帧图像的多个目标位置
   std::vector<ObjectLocation<T1, T2> > v_location;
};
template \langle class Archive, class T1, class T2\rangle
void serialize(Archive& ar, BatchDetectedObjectPara<T1, T2>& data) {
   ar(data.b_info, data.v_location);
template <class T>
struct RegionImage {
   std::vector<ImageData<T> > region;
    std::vector<uint32_t> v_obj_id;
};
template \small{\langle class~Archive,~class~T \rangle}
void serialize(Archive& ar, RegionImage<T>& data) {
   ar(data.v_obj_id, data.region);
// 参数
template \small{\langle class \ T \rangle}
struct RegionImagePara {
                                // frame信息
   FrameInfo f_info;
    RegionImage<T> region;
                                // 每帧对应的小图
};
```

```
template<class Archive, class T>
void serialize(Archive& ar, RegionImagePara<T>& data) {
    ar(data.f_info, data.region);
// 参数
template <class T>
struct BatchRegionImagePara {
    // batch信息
   BatchInfo b_info;
   // 每帧图像若干region image,用一个指向Image数组的指针来引用
    std::vector<RegionImage<T>>> v_region;
};
template \small{\langle class~Archive,~class~T \rangle}
void\ serialize (Archive\&\ ar,\ BatchRegionImagePara < T>\&\ data)\ \{
   ar(data.b_info, data.v_region);
// 递归ID树节点
struct IDTreeNode {
                                   // 节点上的ID值
// 是否为叶子节点
   std::vector<uint32_t> nodeID;
    std::vector<bool> is_leaf;
                                     // 子节点结构指针
    std::vector<IDTreeNode> node;
};
template <class Archive>
void serialize(Archive& ar, IDTreeNode& data) {
    ar(data.nodeID, data.is_leaf, data.node);
// 参数
struct IDTreePara {
   FrameInfo f_info;
                        // frame信息
                        // 图像中的ID Tree
    IDTreeNode tree;
};
template <class Archive>
void serialize(Archive& ar, IDTreePara& data) {
   ar(data.f_info, data.tree);
// 参数
struct BatchIDTreePara {
                                       // batch信息
   BatchInfo b_info;
                                       // 每帧图像中的ID Tree
    std::vector<IDTreeNode> v_tree;
};
template <class Archive>
void serialize(Archive& ar, BatchIDTreePara& data) {
    ar(data.b_info, data.v_tree);
template < class T>
struct Attribute {
```

```
std::string attr_value;
    T score;
template <class Archive, class T>
void serialize(Archive& ar, Attribute<T>& data) {
    ar(data.attr_value, data.score);
template<class T>
struct AttributeTopN {
    uint32_t obj_ID;
    std::map<std::string, std::vector<Attribute<T>> > attr_map;
template \langle class \ Archive, \ class \ T \rangle
void serialize(Archive& ar, AttributeTopN<T>& data) {
    ar(data.obj_ID, data.attr_map);
template < class T>
struct AttributeVec {
    std::vector<AttributeTopN<T>> obj_attr;
template <class Archive, class T>
void serialize(Archive& ar, AttributeVec<T>& data) {
    ar(data.obj_attr);
// ² ÎÊý
template \small{\langle class \ T \rangle}
struct AttributeResultPara {
    FrameInfo f_info;
    AttributeVec<T> attr;
};
template \langle class \ Archive, \ class \ T \rangle
void serialize(Archive& ar, AttributeResultPara<T>& data) {
    ar(data.f_info, data.attr);
// ^2ÎÊý
template <class T>
struct BatchAttributeResultPara {
    BatchInfo b_info;
    std::vector \verb|\langle Attribute Vec < T >> v_attr;|
template \langle class \ Archive, \ class \ T \rangle
void serialize(Archive& ar, BatchAttributeResultPara<T>& data) {
    ar(data.b_info, data.v_attr);
template < class T>
struct Feature {
                                   // 特征向量长度
    int32_t len;
```

```
std::shared_ptr<T> feature;
                                   // 特征向量指针
};
template <class Archive, class T>
void serialize(Archive& ar, Feature<T>& data) {
    ar(data.len);
    if (data.len > 0 && data.feature.get() == nullptr)
        data.feature.reset(new(std::nothrow) T[data.len]);
    ar(cereal::binary_data(data.feature.get(), data.len * sizeof(T)));
template<class T>
struct FeatureList {
                                    // 目标ID
    uint32_t obj_ID;
    std::vector<Feature<T>>> feature list; // 目标的多个特征
};
template \small{\langle class~Archive,~class~T \rangle}
void serialize(Archive& ar, FeatureList<T>& data) {
    ar(data.obj_ID, data.feature_list);
template <class T>
struct FeatureVec {
    std::vector<FeatureList<T>> obj_feature; // 多个目标的属性
template<class Archive, class T>
void serialize(Archive& ar, FeatureVec<T>& data) {
    ar(data.obj_feature);
// 参数
template<class T>
struct FeatureResultPara {
                             // frame信息
    FrameInfo f_info;
    FeatureVec<T> feature;
                             // 图像的多个目标的特征
};
template<class Archive, class T>
void serialize(Archive& ar, FeatureResultPara<T>& data) {
    ar(data.f_info, data.feature);
// 参数
template<class T>
struct BatchFeatureResultPara {
    // batch信息
    BatchInfo b_info;
    // 对应每帧图像的多个目标的特征
    std::vector<FeatureVec<T>>> v_feature;
};
template \small{\langle class~Archive,~class~T \rangle}
void serialize(Archive& ar, BatchFeatureResultPara<T>& data) {
    ar(data.b_info, data.v_feature);
```

```
template<class T>
struct FeatureRecord {
                           // 特征图ID
   uint32_t ID;
   uint32_t len;
                          // 特征向量长度
   std::shared_ptr<T> feature; // 特征向量指针
template <class Archive, class T>
void serialize(Archive& ar, FeatureRecord<T>& data) {
   ar(data.ID, data.len);
   if (data.len > 0 && data.feature.get() == nullptr) {
       data.feature.reset(new(std::nothrow) T[data.len]);
   ar(cereal::binary_data(data.feature.get(), data.len * sizeof(T)));
template < class T1, class T2>
struct RetrievalSet {
   uint32 t TopN;
                        // TopN结果设置
                        // 相似度阈值
   T1 threshold;
   std::vector<FeatureRecord<T2>> record;
                                           // 特征集合中所有的记录
};
template <class Archive, class T1, class T2>
void serialize(Archive& ar, RetrievalSet<T1, T2>& data) {
   ar(data.TopN, data.threshold, data.record);
// 参数
template <class T1, class T2>
struct RetrievalSetPara {
   RetrievalSet<T1, T2> set; // 参数和特征集合
template < class Archive, class T1, class T2>
void serialize(Archive& ar, RetrievalSetPara<T1, T2>& data) {
   ar(data.set);
struct RetrievalResult {
   uint32 t ID; // 特征图ID
   float similarity; // 相似度
template <class Archive>
void serialize(Archive& ar, RetrievalResult& data) {
   ar(data.ID, data.similarity);
struct RetrievalResultTopN {
   std::vector<RetrievalResult> result_list; // 多个检索结果(TopN)
template <class Archive>
void serialize(Archive& ar, RetrievalResultTopN& data) {
   ar(data.result_list);
```

```
// 参数
struct RetrievalResultPara {
                                     // frame信息
    FrameInfo f_info;
    RetrievalResultTopN result;
                                     // 检索结果
};
template <class Archive>
void serialize(Archive& ar, RetrievalResultPara& data) {
    ar(data.f_info, data.result);
// 参数
struct BatchRetrievalResultPara {
    // batch信息
    BatchInfo b info;
    // 对应每帧图像的检索结果
    std::vector<RetrievalResultTopN> v_result;
};
template <class Archive>
void serialize(Archive& ar, BatchRetrievalResultPara& data) {
    ar(data.b_info, data.v_result);
template<class T>
struct EvaluationResult {
                                // 目标ID
    uint32_t obj_ID;
    std::string description;
                                // 描述
    T score;
                                // 评价分数
};
template \langle class \ Archive, \ class \ T \rangle
void serialize(Archive& ar, EvaluationResult<T>& data) {
    ar(data.obj_ID, data.description, data.score);
template < class T>
struct EvaluationResultVec {
    std::vector<EvaluationResult<T>> result;
                                                 // 多个目标的评价结果
template \langle class \ Archive, \ class \ T \rangle
void serialize(Archive& ar, EvaluationResultVec<T>& data) {
    ar(data.result);
// 参数
template \small{\langle class \ T \rangle}
struct EvaluationResultPara {
    FrameInfo f info:
                                       // frame信息
    EvaluationResultVec<T> result;
                                       // 评价结果
};
template \langle class \ Archive, \ class \ T \rangle
void serialize(Archive& ar, EvaluationResultPara<T>& data) {
    ar(data.f_info, data.result);
```

```
// 参数
template <class T>
struct BatchEvaluationResultPara {
    // batch信息
    BatchInfo b_info;
    // 对应每帧图像的评价结果
    std::vector<EvaluationResultVec<T>>> v_result;
template <class Archive, class T>
void serialize(Archive& ar, BatchEvaluationResultPara<T>& data) {
    ar(data.b_info, data.v_result);
template \small{\langle class \ T \rangle}
struct Classification {
                                // 类别取值
    std::string class_value;
    T score;
                                // 置信度
};
template <class Archive, class T>
void serialize(Archive& ar, Classification<T>& data) {
    ar(data.class_value, data.score);
template<class T>
struct ClassificationTopN {
    uint32_t obj_ID;
                                                    // 目标ID
    std::vector〈Classification〈T〉〉 class_result; // 分类结果 (TopN)
};
template\langle class \ Archive, \ class \ T \rangle
void serialize(Archive& ar, ClassificationTopN<T>& data) {
    ar(data.obj_ID, data.class_result);
template<class T>
struct ClassificationVec {
    std::vector〈ClassificationTopN〈T〉〉 class_list; // 多个目标的分类
template<class Archive, class T>
void serialize(Archive& ar, ClassificationVec\langle T \rangle& data) {
    ar(data.class_list);
// 参数
template<class T>
struct ClassificationResultPara {
    FrameInfo f_info;
                                                // frame信息
    ClassificationVec<T> classification;
                                                // 多个目标的分类
};
template<class Archive, class T>
void serialize(Archive& ar, ClassificationResultPara<T>& data) {
    ar(data.f_info, data.classification);
```

```
// 参数
    template<class T>
   struct BatchClassificationResultPara {
        // batch信息
       BatchInfo b_info;
        // 对应每帧图像的多个目标的分类
       std::vector<ClassificationVec<T>>> v_class;
   };
    template \small{\langle class~Archive,~class~T \rangle}
   void serialize(Archive& ar, BatchClassificationResultPara<T>& data) {
       ar(data.b_info, data.v_class);
   struct RawDataBuffer {
       uint32_t len_of_byte; // size length
        std::shared_ptr<uint8_t> data; // 一块buffer, 可转为用户自定义类型
   };
   template <class Archive>
    void serialize(Archive& ar, RawDataBuffer& data) {
       ar(data.len of byte);
        if (data.len_of_byte > 0 && data.data.get() == nullptr) {
            data.data.reset(new(std::nothrow) uint8_t[data.len_of_byte]);
       ar(cereal::binary_data(data.data.get(), data.len_of_byte *
            sizeof(uint8_t)));
   // common raw databuffer struct
   struct BatchRawDataBuffer {
       BatchInfo b_info; // batch info
       std::vector<RawDataBuffer> v_info;
   };
    template < class Archive>
   void serialize(Archive& ar, BatchRawDataBuffer& data) {
       ar(data.b_info, data.v_info);
   template<typename T> const char* TypeName(void);
#define REGISTER_TYPE_DEFINITION(type) \
   template<> inline const char* TypeName<type>(void) { return #type; }
   REGISTER TYPE DEFINITION(int8 t);
    REGISTER_TYPE_DEFINITION(uint8_t);
   REGISTER TYPE DEFINITION(int16 t);
   REGISTER_TYPE_DEFINITION(uint16_t);
   REGISTER_TYPE_DEFINITION(int32_t);
   REGISTER_TYPE_DEFINITION(uint32_t);
    REGISTER_TYPE_DEFINITION(int64_t);
   REGISTER_TYPE_DEFINITION(uint64_t);
   REGISTER_TYPE_DEFINITION(float);
```

```
REGISTER_TYPE_DEFINITION(double);
REGISTER_TYPE_DEFINITION(Point2D);
REGISTER_TYPE_DEFINITION(Point3D);
REGISTER TYPE DEFINITION(Line<Point2D>);
REGISTER TYPE DEFINITION(Line<Point3D>);
REGISTER_TYPE_DEFINITION(Rectangle<Point2D>);
REGISTER_TYPE_DEFINITION(Rectangle<Point3D>);
REGISTER_TYPE_DEFINITION(Polygon<Point2D>);
REGISTER_TYPE_DEFINITION(Polygon<Point3D>);
REGISTER_TYPE_DEFINITION(ROICube);
REGISTER_TYPE_DEFINITION(MaskMatrix<int8_t>);
REGISTER_TYPE_DEFINITION(MaskMatrix<int32_t>);
REGISTER TYPE DEFINITION(MaskMatrix<float>);
REGISTER_TYPE_DEFINITION(RLECode);
REGISTER_TYPE_DEFINITION(RoiPolygon);
{\tt REGISTER\_TYPE\_DEFINITION\,(ArgsRoiPolygon)\,;}
```

7.3 示例

7.3.1 编排配置示例

Graph创建配置文件(graph.prototxt)为proto格式,其示例如下(该示例同时创建两个Graph,每个Graph创建多个Engine并配置映射关系)。

注意

如果用户想要传送文件,必须满足下列条件,否则系统会默认传递的是字符串或数字:

- Linux环境下,文件要包含相对路径或者绝对路径,例如: "/home/1.txt"或者"../test/2.txt"。注意如果字符串中使用了"\"或"/"符号可能会被误判为文件,请避免使用。
- Windows环境下,文件要包含相对路径或者绝对路径,例如: "c:\1.txt",或者"..\test\2.txt"。注意如果字符串中使用了\"或"/"符号可能会被误判为文件,请避免使用

```
graphs {
 graph_id: 100
 device_id: "0"
 priority: 1
  engines {
    id: 1000
    engine name: "SrcEngine"
    side: HOST
    thread_num: 1
 engines {
    id: 1001
    engine_name: "HelloWorldEngine"
    so name: "./libhelloworld.so"
    side: DEVICE
    thread num: 1
 engines {
    id: 1002
    engine_name: "DestEngine"
    side: HOST
    thread_num: 1
```

```
connects {
   src_engine_id: 1000
   src_port_id: 0
   target_engine_id: 1001
   target_port_id: 0
 connects {
   src_engine_id: 1001
   src_port_id: 0
   target_engine_id: 1002
   target_port_id: 0
graphs {
 graph_id: 200
 device_id: "1"
 priority: 1
 engines {
   id: 1000
   engine name: "SrcEngine"
   side: HOST
   thread_num: 1
 engines {
   id: 1001
   engine_name: "HelloWorldEngine"
   internal_so_name: "/lib64/libhelloworld.so"
   side: DEVICE
   thread_num: 1
 engines {
   id: 1002
   engine_name: "DestEngine"
   side: HOST
   thread_num: 1
 connects {
   src_engine_id: 1000
   src_port_id: 0
   target_engine_id: 1001
   target_port_id: 0
 connects {
   src_engine_id: 1001
   src_port_id: 0
   target engine id: 1002
   target_port_id: 0
```

7.3.2 性能优化传输示例

```
(1)使用性能优化方案传输数据,必须对要传输的数据进行手动序列化和反序列化:
// 注: 序列化函数在发送端使用,反序列化在接收端使用,所以这个注册函数最好在接收端和发送端都注册一遍;
结构体
typedef struct
{
    uint32_t frameId;
    uint8_t bufferId;
    uint8_t* image_data;
    uint8_t image_size;
} TEST_STR;

// 序列化TEST_STR结构体,该函数只需要调动注册函数进行注册,参数说明:
// 输入: inputPtr, TEST_STR结构体指针
// 输出: ctrlStr, 控制信息地址
    imageData, 数据信息指针
```

```
imageLen,数据信息大小
void GetTestStrSearPtr(void* inputPtr, std::string& ctrlStr, uint8_t*& imageData, uint32_t&
imageLen)
   // 获取结构体buffer
   TEST_STR* test_str = (TEST_STR*)inputPtr;
   ctrlStr = std::string((char*)inputPtr, sizeof(TEST_STR));
   // 获取数据信息,直接将结构体的数据指针赋值传递
   imageData = (uint8_t*)test_str->image_data;
   imageLen = test_str->image_size;
// 反序列化结构体,回传回来的将是结构体buffer和数据块buffer
// 输入: ctrlPtr, 控制信息地址
       ctrlLen, 控制信息大小
       imageData, 数据信息指针
       imageLen, 数据信息大小
// 输出: std::shared_ptr<void>, 指向结构体的智能指针
std::shared ptr<void> GetTestStrDearPtr(const char* ctrlPtr, const uint32 t& ctrlLen, const
uint8_t* imageData, const uint32_t& imageLen)
   // 获取结构体
   TEST_STR* test_str = (TEST_STR*)ctrlPtr;
   // 获取传输过来的大内存数据
   // 注: 大内存数据最好赋值给智能指针,并注册删除器,如果不是使用智能指针或者没有注册删除器,则需
要在释放该内存时手动调用hiai::Graph::ReleaseDataBuffer(void* ptr);
   std::shared_ptr<TEST_STR<uint8_t>> shared_data = std::make_shared<TEST_STR<uint8_t>>();
   shared data->frameId = test str->frame ID;
   shared_data->bufferId= test_str->bufferId;
   shared_data->image_size = imageLen;
   \verb|shared_data-\rangle image_data.reset(imageData, \verb|hiai::Graph::ReleaseDataBuffer)|;|
   // 返回智能指针给到Device端Engine使用
   return std::static_pointer_cast<void>(shared_data);
// 注册序列化反序列化函数
HIAI_REGISTER_SERIALIZE_FUNC("TEST_STR", TEST_STR, GetTestStrSearPtr, GetTestStrDearPtr);
(2) 在发送数据时,需要使用注册的数据类型,另外配合使用HIAI_DMalloc分配数据内存,可以使性能更优
   注:在从host侧向Device侧搬运数据时,使用HIAI_DMalloc方式将很大的提供传输效率,建议优先使用
HIAI_DMalloc,该内存接口目前支持0 - (256M Bytes - 96 Bytes)的数据大小,如果数据超出该范围,则需要
使用malloc接口进行分配;
   // 使用Dmalloc接口申请数据内存,10000为时延,为10000毫秒,表示如果内存不足,等待10000毫秒;
   HIAI StatusT get ret = HIAIMemory::HIAI DMalloc(width*align height*3/2, (void*&)align buffer,
10000);
   // 发送数据,调用该接口后无需调用HIAI Dfree接口,10000为时延
   graph->SendData(engine_id_0, "TEST_STR", std::static_pointer_cast<void>(align_buffer), 10000);
```