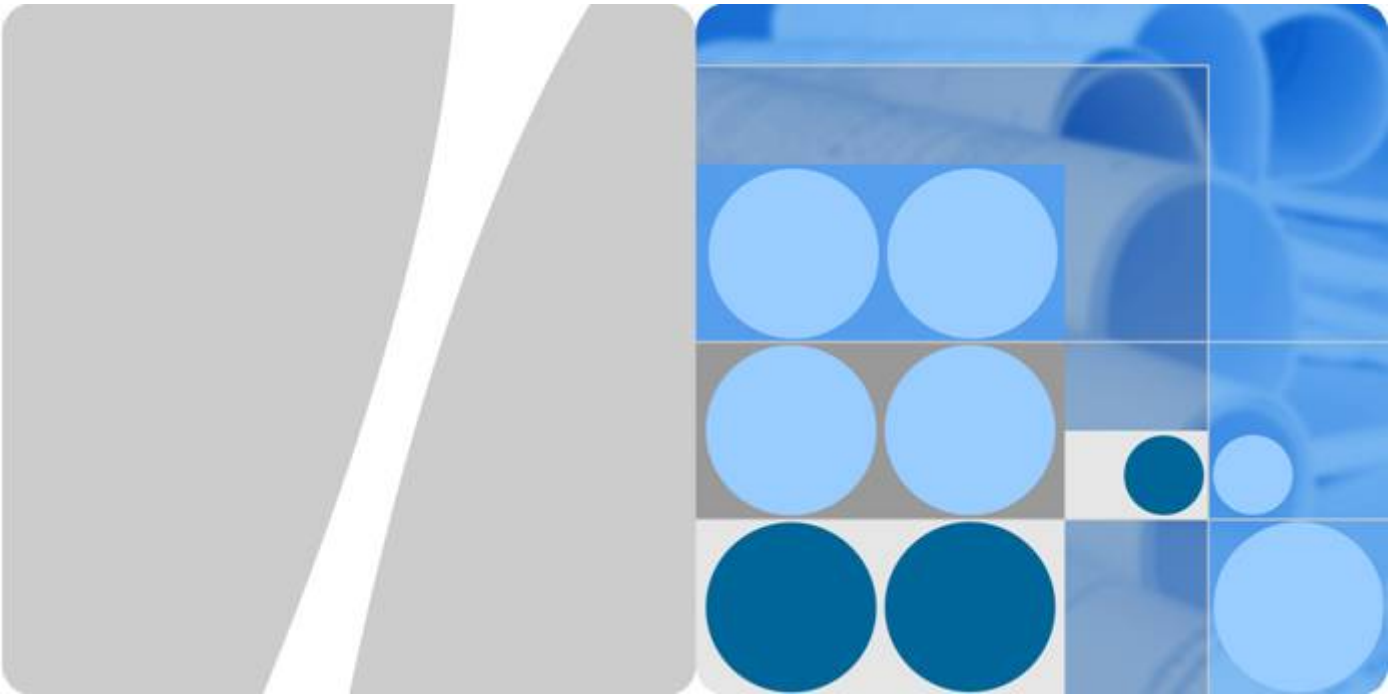

Ascend 310

V100R001

HiAI Engine API 参考

文档版本　01

发布日期　2019-03-12

华为技术有限公司

HUAWEI

商标声明


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

注意

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

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://www.huawei.com

客户服务邮箱： support@huawei.com

客户服务电话： 4008302118

目 录

1 简介

HiAI Engine软件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。HiAI Engine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。

HiAI Engine解决方案总体逻辑包含2个主要的部分，HiAI Engine Agent（运行在host侧）和HiAI Engine Manager（一般运行在Device侧，仿真环境下运行在host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。

- HiAI Engine Agent功能：
  a. 提供Ascend 310芯片功能接口。
  b. 完成与Host APP进行控制命令和处理数据的交互。
  c. 与Device间的IPC通信。
- HiAI Engine Manager功能：
  a. 根据配置文件完成业务流程的建立 。
  b. 根据命令完成业务流程的销毁及资源回收。
  c. 守护进程，负责IPC功能和Device侧DVPP Executor、Framework等软件的启动。

  说明

  - DVPP（Digital Vision Pre-Process，数字视觉预处理）：主要实现视频解码、视频编码、JPEG编解码、PNG解码、视觉预处理。
  - Framework是深度学习框架，可以将caffe、tensorflow等开源框架模型转换成Mind Studio支持的模型。

2 流程编排接口

2.1 流程串接接口

2.1.1 C 语言接口

2.1.1.1 HIAI_Init（与 C++共用）

初始化HiAI Engine的HDC通讯模块。如果在配置文件中为Graph配置所运行的device侧的deviceID，则以配置文件中为准，否则用此处设置的deviceID。

函数格式

HIAI_StatusT HIAI_Init(uint32_t deviceID)

说明

HIAI_StatusT结构体的定义请参见**6.1.2 HiAI Engine自定义数据类型**。

参数说明

参数	说明	取值范围
deviceID	Device 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

序号	错误码	错误码描述
2	HIAI_INIT_FAIL	hiai init fail

2.1.1.2 HIAI_CreateGraph

根据Graph配置文件（prototxt格式）创建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MINIRC上被调用。

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。	-
len	配置文件名的长度。	1~255

返回值

返回的部分错误码请参见“错误码示例”中的“错误码”列。

错误码示例

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

2.1.1.3 HIAI_DestroyGraph

销毁指定的Graph。

只允许在host侧或MINIRC上被调用。

函数格式

HIAI_StatusT HIAI_DestroyGraph (uint32_t graphID)

参数说明

参数	说明	取值范围
graphID	指定的Graph ID。	-

返回值

返回的部分错误码请参见“错误码示例”中的“错误码”列。

错误码示例

序号	错误码	错误码描述
1	HIAI_OK	running ok
2	HIAI_GRAPH_DESTROY_TIMEOUT	destroy timeout
3	HIAI_GRAPH_ENGINE_DESTROY_FAILED	engine destroy failed

2.1.2 C++接口

2.1.2.1 Graph::GetInstance

获取Graph实例，在创建Graph成功后可以正常使用该接口。

函数格式

static std::shared_ptr<Graph> Graph::GetInstance(uint32_t graphID)

参数说明

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

返回值

Graph的智能指针。

错误码示例

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

2.1.2.2 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。

只允许在host侧或MINIRC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

2.1.2.3 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。

只允许在host侧或MINIRC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。	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

2.1.2.4 Graph::DestroyGraph

销毁整个Graph。

只允许在host侧或MINIRC上被调用。

函数格式

static HIAI_StatusT Graph::DestroyGraph(uint32_t graphID)

参数说明

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

返回值

返回的部分错误码请参见“错误码示例”中的“错误码”列。

错误码示例

序号	错误码	错误码描述
1	HIAI_OK	running ok
2	HIAI_GRAPH_DESTROY_TIMEOUT	destroy graph timeout
3	HIAI_GRAPH_ENGINE_DESTROY_FAILED	destroy graph failed

2.2 数据存储接口

2.2.1 C 语言接口

2.2.1.1 HIAI_C_SendData

用户从外部向HiAI Engine发送数据（或消息）。

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**先注册。
- 该消息指针dataPtr控制权为调用者所有， HIAIEngine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。	-
dataPtr	指向具体的消息指针 （该消息指针控制权为调用者所有， HIAIEngine不管理该指针，由调用者负责释放）	-

返回值

返回的部分错误码请参见“错误码示例”中的“错误码”列。

错误码示例

序号	错误码	错误码描述
1	HIAI_OK	running ok
2	HIAI_GRAPH_ENGINE_NOT_EXIST	engine not exist
3	HIAI_GRAPH_NOT_EXIST	graph not exist
4	HIAI_GRAPH_NO_USEFUL_MEMORY	no useful memory
5	HIAI_GRAPH_SENDMSG_FAILED	hdc send msg failed
6	HIAI_GRAPH_INVALID_VALUE	graph invalid value

2.2.1.2 HIAI_SetRecvDataCallback

设置接收消息的回调函数。

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

2.2.2 C++接口

2.2.2.1 Graph::SendData

用户从外部向HiAI Engine发送void类型的数据到指定的端口。

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 timeOutMs = 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。	-
timeOutMs	发送数据时用户可以添加time_out作为时延，单位为毫秒(ms)。如果出现发送队列满或者内存池不足的情况，根据时延作为阻塞，默认为0，即不阻塞。	-

返回值

返回的部分错误码请参见“错误码示例”中的“错误码”列。

错误码示例

序号	错误码	错误码描述
1	HIAI_OK	running ok
2	HIAI_GRAPH_ENGINE_NOT_EXIST	engine not exist
3	HIAI_GRAPH_NOT_EXIST	graph not exist
4	HIAI_GRAPH_NO_USEFUL_MEMORY	no useful memory
5	HIAI_GRAPH_SENDMSG_FAILED	hdc send msg failed
6	HIAI_GRAPH_INVALID_VALUE	graph invalid value

2.2.2.2 HIAI_SendData

用户从外部向HiAI Engine发送void类型的数据到指定的端口。

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 timeoutMs=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。	-
timeoutMs	发送数据时用户可以添加time_out作为时延，单位为毫秒(ms)。如果出现发送队列满或者内存池不足的情况，根据时延作为阻塞，默认为0，即不阻塞。	-

返回值

返回的部分错误码请参见“错误码示例”中的“错误码”列。

错误码示例

序号	错误码	错误码描述
1	HIAI_OK	running ok
2	HIAI_GRAPH_ENGINE_NOT_EXIST	engine not exist
3	HIAI_GRAPH_NOT_EXIST	graph not exist
4	HIAI_GRAPH_NO_USEFUL_MEMORY	no useful memory
5	HIAI_GRAPH_SENDMSG_FAILED	hdc send msg failed
6	HIAI_GRAPH_INVALID_VALUE	graph invalid value

2.2.2.3 Graph::SetDataRecvFunctor

设置接收消息的回调函数。

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

2.3 接口使用示例

2.3.1 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 HIAI Engine
    HIAI_StatusT status = HIAI_Init(0);

    // Step2: Create and Start the Graph
    status = HIAI_CreateGraph(graph_config_proto_file, graph_config_len);
    if (status != HIAI_OK)
```



```
    {
        return status;
    }

    // Step3: 设置回调函数用于接收数据
    HIAI_PortID_t engine_id;
    engine_id.graph_id = g_graph_id;
    engine_id.engine_id = dest_engine_id;
    engine_id.port_id = 0;
    HIAI_SetDataRecvFunctor(engine_id, UserDefDataRecvCallBack);

    return HIAI_OK;
}


// User Application Main Example
int main()
{
    HIAI_StatusT ret = HIAI_OK;

    // 创建流程
    ret = HIAI_InitAndStartGraph();

    if (HIAI_OK != ret)
    {
        return -1;
    }

    // 读取数据，往流程灌输数据
    HIAI_PortID_t engine_id;
    engine_id.graph_id = g_graph_id;
    engine_id.engine_id = src_engine_id;
    engine_id.port_id = 0;

    int is_end_of_data = 0;
    while (!is_end_of_data)
    {
        UserDefDataTypeT* user_def_msg = (UserDefDataTypeT*)malloc(sizeof(UserDefDataTypeT));

        // 读取数据并填充 UserDefDataType，此数据类型注册见数据类型注册章节

        const char * message_name = "UserDefDataType";
        size_t msg_len = strlen(message_name);
        if (HIAI_OK != HIAI_C_SendData(engine_id, message_name, msg_len, user_def_msg))
        {
            // 队列满时返回发送数据失败，由业务逻辑处理是否重发或丢弃
            break;
        }

        free(user_def_msg);
        user_def_msg = nullptr;
    }

    // 处理结束时，删除整个Graph
    HIAI_DestroyGraph(g_graph_id);
    return 0;
}
```

2.3.2 C++接口使用示例

```
/**
* @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 HIAI Engine
    HIAI_StatusT status = HIAI_Init(0);

    // Step2: Create and Start the Graph
    status = hiai::Graph::CreateGraph(graph_config_proto_file);
    if (status != HIAI_OK)
```



```
    {
        HIAI_ENGINE_LOG(status, "Fail to start graph");
        return status;
    }

    // Step3: 设置回调函数用于接收数据
    std::shared_ptr<hiai::Graph> graph = hiai::Graph::GetInstance(g_graph_id);
    if (nullptr == graph)
    {
        HIAI_ENGINE_LOG("Fail to get the graph-%u", g_graph_id);
        return status;
    }

    // Specify the port id (default to zero)
    hiai::EnginePortID target_port_config;
    target_port_config.graph_id = g_graph_id;
    target_port_config.engine_id = dest_engine_id;
    target_port_config.port_id = 0;

    graph->SetDataRecvFunctor(target_port_config,
        std::shared_ptr<hiai::GraphDataRecvInterface>(
        new hiai::GraphDataRecvInterface()));

    return HIAI_OK;
}


// User Application Main Example
int main()
{
    HIAI_StatusT ret = HIAI_OK;

    // 创建流程
    ret = HIAI_InitAndStartGraph();

    if(HIAI_OK != ret)
    {
        HIAI_ENGINE_LOG("Fail to start graph");;
        return -1;
    }

    // 读取数据，往流程灌输数据
    std::shared_ptr<hiai::Graph> graph = hiai::Graph::GetInstance(g_graph_id);
    if (nullptr == graph)
    {
        HIAI_ENGINE_LOG("Fail to get the graph-%u", g_graph_id);
        return -1;
    }

    hiai::EnginePortID engine_id;
    engine_id.graph_id = g_graph_id;
    engine_id.engine_id = src_engine_id;
    engine_id.port_id = 0;

    bool is_end_of_data = false;
    while (!is_end_of_data)
    {
        std::shared_ptr<UserDefDataType> user_def_msg(new UserDefDataType);

        // 读取数据并填充 UserDefDataType，此数据类型注册见数据类型注册章节
        // 往Graph灌输数据
        if (HIAI_OK != graph->SendData(engine_id, "UserDefDataType",
            std::static_pointer_cast<void>(user_def_msg)))
        {
            // 队列满时返回发送数据失败，由业务逻辑处理是否重发或丢弃
            HIAI_ENGINE_LOG("Fail to send data to the graph-%u", g_graph_id);
            break;
        }
    }
```



```
    // 处理结束时，删除整个Graph
    hiai::Graph::DestroyGraph(g_graph_id);
    return 0;
}
```


3 Engine 实现接口

3.1 Engine 的构造函数与析构函数

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.。

说明

该Init接口是可选的，用户根据实际确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**5.4.1 错误码注册**。

错误码示例

本API的错误码由用户进行注册。

3.3 宏：HIAI_DEFINE_PROCESS

定义ENGINE的输入与输出端口数。

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)之前调用本API。

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。	-

3.4 宏：HIAI_IMPL_ENGINE_PROCESS

定义ENGINE具体实现的接口。

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API。

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。	-

参数	说明	取值范围
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。

3.5 Engine::SendData

将数据从本Engine发送到指定port_id。

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 portIdInput, 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
portIdInput	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	发送数据阻塞时延。	-

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_NULL_POINTER	null pointer
3	HIAI_GRAPH_ENGINE_NOT_EXIST	engine not exist
4	HIAI_GRAPH_SRC_PORT_NOT_EXIST	src port not exist

3.6 Engine::SetDataRecvFunctor

设置接收消息的回调函数。

函数格式

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

参数说明

参数	说明	取值范围
portIdInput	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

3.7 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 : 重载Engine Process处理逻辑
    * @[in]: 定义一个输入端口，一个输出端口
    */
    HIAI_DEFINE_PROCESS(MULTI_INPUT_OUTPUT_ENGINE_EXAMPLE_INPUT_SIZE,
MULTI_INPUT_OUTPUT_ENGINE_EXAMPLE_OUTPUT_SIZE)
private:

    // 私有实现一个成员变量，用来缓存输入队列
    hiai::MultiTypeQueue input_que_;
};

}

#endif //MULTI_INPUT_OUTPUT_ENGINE_EXAMPLE_H_
```

```
/**
* @file multi_input_output_engine_example.h
*
* Copyright(c)<2018>, <Huawei Technologies Co.,Ltd>
*
* @version 1.0
*
* @date 2018-4-25
*/

#include "multi_input_output_engine_example.h"
#include "use_def_data_type.h"
#include "use_def_errorcode.h"

namespace hiai {

/**
* @ingroup hiaiengine
* @brief HIAI_DEFINE_PROCESS : 实现多端口输入输出处理流程
* @[in]: 定义一个输入端口，一个输出端口，
*        并该Engine注册，其名为“HIAIMultiEngineExample”
*/
HIAI_IMPL_ENGINE_PROCESS("HIAIMultiEngineExample", HIAIMultiEngineExample,
MULTI_INPUT_OUTPUT_ENGINE_EXAMPLE_INPUT_SIZE)
```



```
{
    // This Engine has three input args and two output

    // 每个端口接收到的数据可能不是同一个时刻，因此某些输入端口可能是空的
    input_que_.PushData(0,arg0);
    input_que_.PushData(1,arg1);
    input_que_.PushData(2,arg2);

    std::shared_ptr<void> input_arg1;
    std::shared_ptr<void> input_arg2;
    std::shared_ptr<void> input_arg3;

    // 仅当三个端口都有输入数据时才继续后续处理,方式一
    if (!input_que_.PopAllData(input_arg1, input_arg2, input_arg3))
    {
        HIAI_ENGINE_LOG(this, HIAI_INVALID_INPUT_MSG, "fail to process");
        return HIAI_INVALID_INPUT_MSG;
    }

    // 仅当三个端口都有输入数据时才继续后续处理，方式二
/*
    if (!(input_que_.FrontData(0, input_arg1) && input_que_.FrontData(1, input_arg2) &&
input_que_.FrontData(2, input_arg3)))
    {
        HIAI_ENGINE_LOG(this, HIAI_INVALID_INPUT_MSG, "fail to process");
        return HIAI_INVALID_INPUT_MSG;
    } else {
         input_que_.PopData(0, input_arg1);
         input_que_.PopData(1, input_arg2);
         input_que_.PopData(2, input_arg3);
    }
*/
    //
    // 中间业务逻辑可以直接调用 DVPP API或AIModelManger API 处理。
    //

    // 分配内存用来保存结果
    std::shared_ptr<UseDefDataTypeT> output1 =
        std::make_shared<UseDefDataTypeT>();

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

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

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

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

    return HIAI_OK;
}

}
```


4 功能接口

4.1 离线模型管家（只提供 C++接口）

4.1.1 AIModelManager::Init

初始化接口，主要完成模型加载。

函数格式

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

参数说明

参数	说明	取值范围
config	配置信息。 关于AIConfig数据类型的定义，请参见**4.2.2 AIConfig**。	-
model_descs	模型描述信息列表，支持从内存、文件加载模型，内存加载模型优先级高于从文件加载模型。 关于AIModelDescription数据类型的定义，请参见**4.2.5 AIModelDescription**。	-

返回值

SUCCESS 初始化成功/FAILED 初始化失败。

调用示例

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

- 从文件加载模型：

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

- 从内存加载模型：

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

说明

model_size必须与模型实际大小保持一致。

4.1.2 AIModelManager::SetListener

设置模型管理回调函数。

说明

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

函数格式

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

参数说明

参数	说明	取值范围
listener	回调函数。 关于IAIListener数据类型的定义，请参见**4.2.8 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

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**4.2.6 AIContext**。	-
in_data	模型输入tensor列表。 关于IAITensor数据类型的定义，请参见**4.2.9 IAITensor**。	-
out_data	模型输出tensor列表。 关于IAITensor数据类型的定义，请参见**4.2.9 IAITensor**。	-
timeout	计算超时时间，预留参数，默认值为0，配置无效。	-

返回值

SUCCESS 初始化成功/FAILED 初始化失败。

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

参数说明

参数	说明	取值范围
in_data	输入tensor列表。 关于IAITensor数据类型的定义，请参见**4.2.9 IAITensor**。	-
out_data	输出tensor列表。 关于IAITensor数据类型的定义，请参见**4.2.9 IAITensor**。	-

返回值

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**4.2.9 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。	-
input_tensor	模型输入尺寸列表。 关于TensorDimension数据类型的定义，请参见**4.2.7 TensorDimension**。	-
output_tensor	模型输出尺寸列表。 关于TensorDimension数据类型的定义，请参见**4.2.7 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。

若输入的input_tensor或output_tensor的size不为0，则获取到的结果数据会追加vector的后面。

```
input_tensor
{
name = "data"                  #输入层的name
data_type = 0                  #预留数据类型，暂时不用
size = 20                      #内存大小，单位字节
format = 0                     #预留Tensor排布格式，暂时不用
dims = {1,3,224,224}
}
output_tensor
{
name =  "output_0_prob_0"      #输出Tensor的name，格式为：output_{数字}_{输出节点name}_{输出节点输出索引}
data_type = 0                  #预留数据类型，暂时不用
size = 20                      #内存大小，单位字节
format = 0                     #预留Tensor排布格式，暂时不用
dims = {1,1000,1,1}
}
```

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

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

返回值

无。

4.1.10 AITensorFactory::CreateTensor

创建模型tensor。

函数格式

std::shared_ptr<IAITensor> CreateTensor(const AITensorDescription &tensor_desc, void *buffer, int32_t size);

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4.2.4-AITensorDescription。	-
buffer	数据地址。 **说明** 建议输入数据（输入数据一般可直接使用框架传入的内存， 该内存是由框架通过HIAI_DMalloc申请得到）及输出数据都通过HIAI_DMalloc接口申请，这样就能够使能算法推理的零拷贝机制， 优化Process时间。	-
size	数据长度。 **说明** 单位为字节，size必须与数据实际大小一致。	-

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

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。

```
/**
* @file mngr_sample.h
*
* Copyright(c)<2018>, <Huawei Technologies Co.,Ltd>
*
* @version 1.0
*
* @date 2018-4-25
*/

#include <stdio.h>
#include <string>
#include <vector>
#include <iostream>
#include <fstream>
#include <assert.h>
#include "hiaiengine/ai_model_manager.h"

using namespace std;
using namespace hiai;

// 图片数据路径
static const std::string IMAGE_FILE_PATH = "/data/input_zebra.bin";
static const char* MODEL_PATH = "/data/ResNet.davincimodel";
static const char* MODEL_NAME = "resnet18";
/*
*** brief：读取输入数据
*/
```



```
char* ReadBinFile(const char *file_name, uint32_t *fileSize)
{
    std::filebuf *pbuf;
    std::ifstream filestr;
    size_t size;
    filestr.open(file_name, std::ios::binary);
    if (!filestr)
    {
        return nullptr;
    }

    pbuf = filestr.rdbuf();
    size = pbuf->pubseekoff(0, std::ios::end, std::ios::in);
    pbuf->pubseekpos(0, std::ios::in);
    char * buffer = (char*)malloc(size);
    if (nullptr == buffer)
    {
        return nullptr;
    }

    pbuf->sgetn(buffer, size);
    *fileSize = size;

    filestr.close();
    return buffer;
}


int main(int argc, char* argv[])
{
    vector<shared_ptr<IAITensor>> model_input;
    vector<shared_ptr<IAITensor>> model_output;

    AIModelManager model_mngr;
    AIModelDescription model_desc;
    AIConfig config;
    AIContext context;
    model_desc.set_path(MODEL_PATH);
    model_desc.set_name(MODEL_NAME);
    model_desc.set_type(0);
    vector<AIModelDescription> model_descs;
    model_descs.push_back(model_desc);
    // AIModelManager Init
    AIStatus ret = model_mngr.Init(config, model_descs);
    if (SUCCESS != ret)
    {
        printf("AIModelManager Init failed. ret = %d\n", ret);
        return -1;
    }
    // 输入tensor
    // 输入tensor会在读取图片数据后重新设置，这里的作用只是初始化
    AITensorDescription tensor_desc = AINeuralNetworkBuffer::GetDescription();
    shared_ptr<IAITensor> input_tensor = AITensorFactory::GetInstance()->CreateTensor(tensor_desc);
    if (nullptr == input_tensor)
    {
        printf("Create input_tensor failed.\n");
        return -1;
    }

    // 读取图片数据
    uint32_t image_data_size = 0;
    float* image_data = (float*)ReadBinFile(IMAGE_FILE_PATH.c_str(), &image_data_size);
    if (nullptr == image_data)
    {
        printf("ReadBinFile failed bin file path= %s \n", IMAGE_FILE_PATH.c_str());
        return -1;
    }

    // 将图片数据地址指针及长度，设置给input_simple_tensor
```



```
    shared_ptr<AISimpleTensor> input_simple_tensor =
static_pointer_cast<AISimpleTensor>(input_tensor);
    input_simple_tensor->SetBuffer((void*)image_data, image_data_size);
    model_input.push_back(input_tensor);

    // 创建输出tensor
    if(model_mngr.IsPreAllocateOutputMem())
    {
        ret = model_mngr.CreateOutputTensor(model_input, model_output);
        if (SUCCESS != ret)
        {
            printf("CreateOutputTensor failed.ret = %d\n", ret);
            delete image_data;
            return -1;
        }
    }else{
        // 用户创建tensor
        ret = model_mngr.GetModelIOTensorDim(MODEL_NAME, input_tensor_dims, output_tensor_dims);
        std::vector<TensorDimension> input_tensor_dims;
        std::vector<TensorDimension> output_tensor_dims;
        for(TensorDimension & dims : output_tensor_dims)
        {
            shared_ptr<IAITensor> output_tensor = AITensorFactory::GetInstance()-
>CreateTensor(tensor_desc);
            shared_ptr<AISimpleTensor> output_simple_tensor =
static_pointer_cast<AISimpleTensor>(output_tensor);
            output_simple_tensor->setBuffer((void*) new char[dims.size], dims.size);
            model_output.push_back(output_tensor)
        }
    }

    // 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;
}
```

示例 2 异步调用示例

请参见“~/tools/che/ddk/ddk/sample/customop/customop_app/main.cpp”。

4.2 数据类型

4.2.1 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
};
```

4.2.2 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
};
```

4.2.3 AITensorParaDescription

tensor 参数描述，详细实现参考ai_types.proto。

```
message AITensorParaDescription
{
    string name = 1;   // 参数名称
    string type = 2;   // 参数类型
    string value = 3;  // 参数值
    string desc = 4;   // 参数描述
    repeated AITensorParaDescription sub_paras = 5; // 子参数列表
};
```

4.2.4 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
};
```

4.2.5 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。

```
message 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; //保留字段
    string key = 7;  // 模型秘钥
    repeated string sub_key = 8; //保留字段
    enum Frequency   // 保留字段
    {
        UNSET   =0;
        LOW     =1;
        MEDIUM  =2;
        HIGH    =3;
    }
    Frequency frequency = 9;// 保留字段
    enum DeviceType  // 保留字段
    {
```



```
        NPU  = 0;
        IPU  = 1;
        MLU  = 2;
        CPU  = 3;
        NONE = 255;
    }
    DeviceType device_type = 10;// 保留字段
    enum Framework  // 保留字段
    {
        OFFLINE    =0;
        CAFFE      =1;
        TENSORFLOW =2;
    }
    Framework framework = 11;// 保留字段
    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
};
```

4.2.6 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。

```
class AIContext
{
public:
    /*
    * @brief 获取参数
    * @param [in] key  参数对应的key
    * @return sring key对应的值，如果不存在，则返回空字符串
    */
    const std::string GetPara(const std::string &key) const;

    /*
    * @brief 设置参数
    * @param [in] key    参数对应的key
    * @param [in] value  参数对应的value
    */
    void AddPara(const std::string &key, const std::string &value);

    /*
    * @brief 删除参数
    * @param [in] key 待删除参数对应的key
    */
    void DeletePara(const std::string &key);

    /*
    * @brief 获取所有参数
    * @param [out] keys 所有已设置的参数key
    */
    void GetAllKeys(std::vector<std::string> &keys);
private:
    std::map<std::string, std::string> paras_; /** 参数的名值对定义 */
};
```

4.2.7 TensorDimension

用于描述模型输入输出tensor信息，主要包含tensor的dim信息，数据类型，内存大小，tensor名字。

```
/*
* 描述Tensor尺寸
*/
struct TensorDimension
{
    std::vector<uint32_t> dims;
    uint32_t format;
    uint32_t data_type;
```



```
    uint32_t size;
    std::string  name;
};
```

4.2.8 IAIListener

异步调用Process时需要配置IAIListener，用于模型执行结束后，回调通知用于，具体实现如下：

```
/*
* 异步回调接口，由调用方实现
*/
class IAIListener
{
public:
    virtual ~IAIListener(){}
    /*
    * @brief 异步回调接口
    * @param [in] context 运行时上下文信息，包含nnnode运行时的一些可变参数配置
    * @param [in] result 执行结束时任务状态
    * @param [in] out_data 执行结束时的输出数据
    */
    virtual void OnProcessDone(const AIContext &context, int result,-
        const std::vector<std::shared_ptr<IAITensor>> &out_data) = 0;
    / *
      * @brief 服务死亡回调接口，当客户端到服务端挂死时，通知应用
      */
    virtual void OnServiceDied() {};
};
```

4.2.9 IAITensor

Porcess函数时使用IAITensor作为模型的输入输出。

```
/**
* 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;
};
```

4.3 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。

4.3.1 AIAlgAPIFactory

ALG的API注册工厂类。

```
class AIAlgAPIFactory
{
public:
    static AIAlgAPIFactory* GetInstance();

    /*
    * @brief 获取API
    * @param [in] name api名称
    * @return API原型指针
    */
    AI_ALG_API GetAPI(const std::string &name);

    /*
    * @brief 注册API
    * @param [in] desc api描述
    * @param [in] func api定义
    * @return SUCCESS 成功
    * @return 其他    失败
    */
    AIStatus RegisterAPI(const AIAPIDescription &desc, AI_ALG_API func);

    /*
    * @brief 获取所有API描述
    * @param [in] api_desc_list api描述列表
    */
    void GetAllAPIDescription(AIAPIDescriptionList &api_desc_list);

    /*
    * @brief 卸载注册API
    * @param [in] api_desc api描述
    */
    AIStatus UnRegisterApi(const AIAPIDescription &api_desc);

    /*
    * @brief 获取API描述
    * @param [in] name api名称
    * @param [in] api_desc api描述
    * @return SUCCESS 成功
    * @return 其他    失败
    */
    AIStatus GetAPIDescription(const std::string &name, AIAPIDescription &api_desc);

private:
    std::map<std::string, AI_ALG_API> func_map_;
    std::map<std::string, AIAPIDescription> desc_map_;
    std::mutex api_reg_lock_;
};
```

4.3.2 AIAlgAPIRegisterar

对AIAlgAPIFactory工厂类的注册封装。

```
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_;
    };
```

4.3.3 REGISTER_ALG_API_UNIQUE

API注册宏，在API实现中使用。

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

4.3.4 REGISTER_ALG_API

对REGISTER_ALG_API_UNIQUE的封装宏。

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

4.3.5 AIModelManager

模型管理类。

```
    class AIModelManager : public IAINNNode
    {
    public:
        AIModelManager();
        /*
        * @brief 是否可以预分配输出内存，该接口由业务NNNode实现，默认值为true.
        */
        virtual bool IsPreAllocateOutputMem() override;

       /*
        * @brief 获取AINNNodeDescription对象.
        */
        static AINNNodeDescription GetDescription();

        /*
        * @brief 获取已加载模型的输入输出尺寸
        * @param [in] model_name 模型名字
        * @param [out] input_tensor 模型输入尺寸
        * @param [out] output_tensor 模型输出尺寸
        * @return SUCCESS 成功
        * @return 其他 失败
        */
        AIStatus GetModelIOTensorDim(const std::string& model_name,
            std::vector<TensorDimension>& input_tensor, std::vector<TensorDimension>&
output_tensor);

        /*
        * @brief 设置线程推理请求ID
        * @param [in] request_id 模型名字
        * @return 无
        */
        static void SetRequestId(uint64_t request_id);

        ~AIModelManager();

    private:
        AIModelManagerImpl* impl_;
    };
```

4.3.6 getModelInfo

解析一个模型文件,获取模型信息。


```
    /**
     * @ingroup hiai
     * @brief 解析一个模型文件,获取模型信息
     * @param [in] model_path 模型文件路径
     * @param [in] key 模型解密秘钥
     * @param [out] model_desc 模型信息
     * @return Status 运行结果
     */

     hiai::AIStatus getModelInfo(const std::string & model_path,
         const std::string & key, AIModelDescription & model_desc);
```

4.3.7 IAINNNode

NN Node接口，业务提供方实现。

```
class IAINNNode
{
public:
    virtual  ~IAINNNode(){}
    /*
    * @brief 初始化接口，业务在该接口中实现模型加载或其他初始化动作
    * @param [in] model_desc 模型信息，如果不需要模型，则传入空的vector
    * @param [in] config 配置参数
    * @return SUCCESS 成功
    * @return 其他    失败
    */
    virtual AIStatus Init(const AIConfig &config,
        const std::vector<AIModelDescription> &model_descs = {}) = 0;

    /*
    * @brief 设置监听
    * @parm [in] 如果listener设置为nullptr，表示process接口为同步调用,否则为异步
    * @return SUCCESS 成功
    * @return 其他    失败
    */
    virtual AIStatus SetListener(std::shared_ptr<IAIListener> listener) = 0;

    /*
    * @brief 计算接口
    * @param [in] context 运行时上下文信息，包含nnnode运行时的一些可变参数配置
    * @param [in] in_data 输入数据
    * @param [out] out_data 输出数据
    * @param [in] timeout 超时时间，同步调用时无效
    * @return SUCCESS 成功
    * @return 其他    失败
    */
    virtual AIStatus Process(AIContext &context,
        const std::vector<std::shared_ptr<IAITensor>> &in_data,
        std::vector<std::shared_ptr<IAITensor>> &out_data, uint32_t timeout) = 0;

    /*
    * @brief 创建输出的Tensor列表
    * @param [in] in_data 输入tensor列表，计算输出时可能使用
    * @param [out] out_data 输出的tensor列表
    * @return SUCCESS 成功
    * @return 其他    失败
    */
    virtual AIStatus CreateOutputTensor(
        const std::vector<std::shared_ptr<IAITensor>> &in_data,
        std::vector<std::shared_ptr<IAITensor>> &out_data) { (void)in_data;
(void)out_data;return SUCCESS; }

    /*
    * @brief 是否可以预分配输出内存，该接口由业务NNNode实现，默认值为true.
    */
    virtual bool IsPreAllocateOutputMem() { return true; }

    /*
```



```
        * @brief 判断nnnode是否有效
        */
        virtual AIStatus IsValid() { return SUCCESS; }

        /*
        * @brief 查询node 支持的同步方式
        * @return BOTH    支持同步异步
        * @return ASYNC   仅支持异步
        * @return SYNC    仅支持同步
        */
        virtual AI_NODE_EXEC_MODE GetSupportedExecMode() { return AI_NODE_EXEC_MODE::BOTH; }

        /*
        * @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; }
    };
```

4.3.8 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）。

```
    class AINNNodeFactory
    {
    public:
        static AINNNodeFactory* GetInstance();

        /*
        * @brief 根据nnnode name创建NNNode
        * @param [in] name nnnode名称
        * @return std::shared_ptr<IAINNNode> name对应的NNNode对象指针，如果返回nullptr，表示找不到对应的nnnode
        */
        std::shared_ptr<IAINNNode> CreateNNNode(const std::string &name);

        /*
        * @brief 获取所有已注册nnnode的描述信息
        * @param [out] nnnode_desc 所有已注册的nnnode描述信息
        * @return SUCCESS 成功
        * @return 其他    失败
        */
        void GetAllNNNodeDescription(AINNNodeDescriptionList &nnnode_desc);

        /*
        * @brief 根据nnnode name获取nnnode的描述信息
        * @param [in] name nnnode名称
        * @param [out] engin_desc nnnode描述信息
        * @return SUCCESS 成功
        * @return 其他    失败
        */
        AIStatus GetNNNodeDescription(const std::string &name, AINNNodeDescription &engin_desc);

        /*
        * @brief 注册NNNode创建函数
        * @param [in] nnnode_desc nnnode描述信息
        * @param [in] create_func nnnode创建函数
        * @return SUCCESS 成功
        * @return 其他    失败
        */
        AIStatus RegisterNNNodeCreator(const AINNNodeDescription &nnnode_desc,
            AINNNODE_CREATE_FUN create_func);
        AIStatus RegisterNNNodeCreator(const string nnnode_str,
            AINNNODE_CREATE_FUN create_func);

        /*
        * @brief 注销NNNode
        * @param [in] name nnnode名称
```

```
        * @return SUCCESS 成功
        */
        AIStatus UnRegisterNNNode(const AINNNodeDescription &nnnode_desc);
        AIStatus UnRegisterNNNode(const string nnnode_str);

    private:
        std::map<std::string, AINNNODE_CREATE_FUN> create_func_map_;
        std::map<std::string, AINNNodeDescription> nnnode_desc_map_;
        std::mutex  map_lock_;
    };
```

4.3.9 AINNNodeRegisterar

NN Node注册类。

```
    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_;
    };
```

4.3.10 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)
```

4.3.11 AITensorGetBytes

获取Tensor字节大小。

```
extern uint32_t AITensorGetBytes(int32_t data_type);
```

4.3.12 AITensorFactory

Tensor工厂类，用来创建Tensor。

```
    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);
        /*
        * @brief 注册Tensor
        * @param [in] tensor_desc tensor描述
        * @param [in] create_func tensor创建函数
        */
        AIStatus RegisterTensor(const AITensorDescription &tensor_desc,
            CREATE_TENSOR_FUN create_func);
        AIStatus RegisterTensor(const string tensor_str, CREATE_TENSOR_FUN create_func)
        {
            // 加锁，防止多线程并发
            AITensorDescription tensor_desc;
            tensor_desc.set_type(tensor_str);
            return RegisterTensor(tensor_desc, create_func);
        }

       /*
        * @brief 卸载注册Tensor
        * @param [in] tensor_desc tensor描述
        */
        AIStatus UnRegisterTensor(const AITensorDescription &tensor_desc);
        AIStatus UnRegisterTensor(const string tensor_str)
        {
            AITensorDescription tensor_desc;
            tensor_desc.set_type(tensor_str);
            return UnRegisterTensor(tensor_desc);
        }

        /*
        * @brief 获取所有的Tensor列表
        * @param [out] tensor_desc_list 输出Tensor描述列表
        */
        void GetAllTensorDescription(AITensorDescriptionList &tensor_desc_list);

        /*
        * @brief 获取tensor描述
        */
        AIStatus GetTensorDescription(const std::string& tensor_type_name,
            AITensorDescription &tensor_desc);

    private:
        std::map<std::string, AITensorDescription> tensor_desc_map_;
        std::map<std::string, CREATE_TENSOR_FUN> create_func_map_;
        std::mutex tensor_reg_lock_;
    };
```

4.3.13 REGISTER_TENSOR_CREATER_UNIQUE

Tensor注册宏。

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

4.3.14 REGISTER_TENSOR_CREATER

对Tensor注册宏的封装宏。


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

4.3.15 AISimpleTensor

简单类型的数据Tensor。

```
class AISimpleTensor : public IAITensor
{
 public:
     AISimpleTensor();

     ~AISimpleTensor();

     /*
     * @brief 获取类型名称
     */
     virtual const char* const GetTypeName() override;

    /*获取数据地址*/
     void* GetBuffer();

     /*
     @brief 设置数据地址
     @param [in] data 数据指针
     @param [in] size 大小
     @param [in] isown 是否在tensor生命周期结束后，由tensor释放data地址内存
     */
     void SetBuffer(void *data, const int32_t size, bool isown=false);

     /*
     @brief 获取数据占用空间大小
     */
     uint32_t GetSize();

     /*
     @brief 分配数据空间
     */
     void* MallocDataBuffer(uint32_t size);

     /*
     * @brief  获取序列化后字节长度
     */
     virtual uint32_t ByteSizeLong() override;

     /*
     * @brief 通过AITensorDescription设置参数
     * @param [in] tensor_desc tensor描述
     * @return true: init成功
     * @return false: init失败，失败的可能原因比如tensor_desc与当前tensor不符
     */
     virtual bool Init(const AITensorDescription &tensor_desc) override;
     /*
     @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);

 private:
     void* data_;
     uint32_t size_;
     bool isowner_ ;

     static std::string type_name_;
 }
```


4.3.16 AINeuralNetworkBuffer

NN通用Buffer定义。

```
class AINeuralNetworkBuffer : public AISimpleTensor
{
public:
    AINeuralNetworkBuffer()
    {
        data_type_ = 0;
        number_    = 1;
        channel_   = 1;
        height_    = 1;
        width_     = 1;
        name_      = "";
    };

    /*
    * @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);
    /*
    * @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_;
};
```

4.3.17 AIImageBuffer

图像的通用Buffer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);
    /*
    * @brief 获取描述信息
    */
    static AITensorDescription GetDescription(
        const std::string &size = "0",const std::string &height="0",
        const std::string &width="0", const std::string format="JPEG");

    /*
    @brief 创建tensor
    */
    static std::shared_ptr<IAITensor> CreateTensor(const AITensorDescription& tensor_desc);

    /*
    @brief 获取图像格式
    */
    ImageFormat GetFormat();
    /*
    @brief 设置图像格式
    */
    void SetFormat(ImageFormat format);

    /*
    @brief 获取height
    */
    int32_t GetHeight();

    /*
    @brief 设置height
    */
    void SetHeight(int32_t height);

    /*
    @brief 获取width
    */
    int32_t GetWidth();

    /*
    @brief 设置width
    */
    void SetWidth(int32_t width);
private:
    ImageFormat format_;
    int32_t width_;
    int32_t height_;
};
```

4.3.18 HIAILog

HIAI的日志类。

```
  class HIAILog {
  public:
    /**
    * @ingroup hiaiengine
    * @brief 获取HIAILog指针
    */
    HIAI_LIB_INTERNAL static HIAILog* GetInstance();

    /**
    * @ingroup hiaiengine
    * @brief : 是否初始化
    * @return : 是否初始化 true 是
    */
    HIAI_LIB_INTERNAL bool IsInit() { return isInitFlag; }
```



```
    /**
    * @ingroup hiaiengine
    * @brief 获取log输出级别
    * @return :log输出级别
    */
    HIAI_LIB_INTERNAL uint32_t HIAIGetCurLogLevel();

    /**
    * @ingroup hiaiengine
    * @brief 判断该条log是否需要输出
    * @param [in]errorCode:消息的错误码
    * @return 是否能被输出
    */
    HIAI_LIB_INTERNAL bool HIAILogOutputEnable(const uint32_t errorCode);

    /**
    * @ingroup hiaiengine
    * @brief log对应的级别名称
    * @param [in]logLevel:宏定义的log级别
    */
    HIAI_LIB_INTERNAL std::string HIAILevelName(const uint32_t logLevel);

    /**
    * @ingroup hiaiengine
    * @brief 输出log
    * @param [in]moudleID:enum定义的组件id,
    * @param [in]logLevel:宏定义的log级别,
    * @param [in]strLog:输出的日志内容
    */
    HIAI_LIB_INTERNAL void HIAISaveLog(const int32_t moudleID,
        const uint32_t logLevel, const char* strLog);
  protected:
    /**
    * @ingroup hiaiengine
    * @brief HIAILog构造函数
    */
    HIAI_LIB_INTERNAL HIAILog();
    HIAI_LIB_INTERNAL ~HIAILog() {}
  private:
    /**
    * @ingroup hiaiengine
    * @brief HIAILog初始化函数
    */
    HIAI_LIB_INTERNAL void Init();
    HIAI_LIB_INTERNAL HIAILog(const HIAILog&) = delete;
    HIAI_LIB_INTERNAL HIAILog(HIAILog&&) = delete;
    HIAI_LIB_INTERNAL HIAILog& operator=(const HIAILog&) = delete;
    HIAI_LIB_INTERNAL HIAILog& operator=(HIAILog&&) = delete;
  private:
    HIAI_LIB_INTERNAL static std::mutex mutexHandle;
    uint32_t outputLogLevel;
    std::map<uint32_t, std::string> levelName;
    std::map<std::string, uint32_t> levelNum;
    static bool isInitFlag;
};
```

4.3.19 HIAI_ENGINE_LOG

日志分装宏，对HIAI_ENGINE_LOG_IMPL宏进行封装。

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


4.4 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

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。

函数格式

HIAI_StatusT HIAI_GetDeviceNum(uint32_t *devCount);

参数说明

参数	说明	取值范围
devCount	输出device个数的指针。	-

返回值

返回的部分错误码请参见“错误码示例”中的“错误码”列。

错误码

序号	错误码级别	错误码	错误码描述
1	HIAI_INFO	HIAI_OK	running ok
2	HIAI_ERROR	HIAI_GRAPH_GET_DEVNUM_ERROR	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。

函数格式

HIAI_StatusT HIAI_GetFirstDeviceId(uint32_t *firstDevID);

参数说明

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

返回值

返回的部分错误码请参见“错误码示例”中的“错误码”列。

错误码

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

调用示例

```
uint32_t first_dev_id;
HIAI_GetFirstDeviceId(&first_dev_id);
```

5.1.3 获取第一个 GraphID

获取GraphID。

函数格式

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

参数说明

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

返回值

C语言与C++通用接口：错误码。

错误码

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

调用示例

```
uint32_t first_graph_id;
HIAI_GetFirstGraphId(&first_graph_id);
```

5.1.4 获取下一个 DeviceID

获取下一个device的id。

函数格式

HIAI_StatusT HIAI_GetNextDeviceId(const uint32_t curDevID,uint32_t *nextDevID);

参数说明

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

返回值

返回的部分错误码请参见“错误码示例”中的“错误码”列。

错误码

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

调用示例

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

5.1.5 获取下一个 GraphID

获取下一个GraphID。

函数格式

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

参数说明

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

返回值

返回的部分错误码请参见“错误码示例”中的“错误码”列。

错误码

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

调用示例

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

5.1.6 获取 Engine 指针

根据engineID查找Engine对象指针。

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。

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。

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。

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。

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。

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 HiAI Engine 配置文件

解析配置文件。

函数格式

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

参数说明

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

返回值

返回的部分错误码请参见“错误码示例”中的“错误码”列。

错误码

序号	错误码级别	错误码	错误码描述
1	-	HIAI_GRAPH_GET_INSTANCE_NULL	-

5.1.13 获取 PCIe 的 Info

获取PCIe的Info。

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	-

参数	说明	取值范围
func	返回PCIe功能号	-

返回值

返回的部分错误码请参见“错误码示例”中的“错误码”列。

错误码

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

调用示例

```
uint32_t devId= 1;
uint32_t bus;
uint32_t dev;
uint32_t func;
HIAI_GetPCIeInfo(devId, &bus, &dev, &func);
```

5.1.14 获取版本号

获取API版本信息。

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();
```

5.2 数据类型序列化和反序列化

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。

说明

该接口需要在host和Device端同时注册。

函数格式

HIAI_REGISTER_DATA_TYPE(name, type)

参数说明

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

返回值

返回的部分错误码请参见“错误码示例”中的“错误码”列。

错误码示例

序号	错误码	错误码描述
1	HIAI_OK	running ok
2	HIAI_ENGINE_FUNCTOR_NULL	hiai engine function is null
3	HIAI_ENGINE_FUNCTOR_EXIST	hiai engine function is existed

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。

说明

该接口需要在host和Device端同时注册该类。

函数格式

HIAI_REGISTER_TEMPLATE_DATA_TYPE(name, type, basictype1, basictype2, …)

参数说明

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

返回值

返回的部分错误码请参见“错误码示例”中的“错误码”列。

错误码示例

序号	错误码	错误码描述
1	HIAI_OK	running ok
2	HIAI_ENGINE_FUNCTOR_NULL	hiai engine function is null
3	HIAI_ENGINE_FUNCTOR_EXIST	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。

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

使用场景：

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

说明

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

函数格式

HIAI_REGISTER_SERIALIZE_FUNC(name, type, hiaiSerializeFunc, hiaiDeSerializeFunc)

参数说明

参数	说明	取值范围
name	注册的消息名字。	-
type	自定义数据结构的类型。	-
hiaiSerializeFunc	序列化函数。	-
hiaiDeSerializeFunc	反序列化函数	-

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_ENGINE_FUNCTOR_NULL	hiai engine function is null
3	HIAI_ENGINE_FUNCTOR_EXIST	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。

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 data1;
    float data2;
}UseDefDataTypeT;

#ifdef __cplusplus
// 用户自定义的数据类型 （带模板类型）
template<typename T1, typename T2, typename T3>
class UseDefTemplateDataType
{
  public:
    std::vector<T1> data_vec_;
    std::map<T2, T3> data_map_;
};
#endif

#endif
```

```
/********************************************************************************
 *                     CopyRight (C) Hisilicon Co., Ltd.
 *
 *       Filename:  use_def_data_type_reg.cpp
 *    Description:  User defined Data Type Register
 *
 *        Version:  1.0
 *        Created:  2018-01-08 10:15:18
 *         Author:  h00384043
 *
 *       Revision:  initial draft;
 ********************************************************************************/

#include "use_def_data_type.h"
#include "hiaiengine/data_type_reg.h"

// 用户自定义的数据类型必须先注册到HIAIEngine，才能作为Engine之间的通信接口正常使用
template<class Archive>
void serialize(Archive& ar, UseDefDataTypeT& data)
{
    ar(data.data1, data.data2);
}

// 注册UseDefDataTypeT
HIAI_REGISTER_DATA_TYPE("UseDefDataTypeT", UseDefDataTypeT)


// 模板类型数据的注册
// 对于模板类型，对于所有需要使用的类型必须都注册
template<class Archive, typename T1, typename T2, typename T3>
void serialize(Archive& ar, UseDefTemplateDataType<T1, T2, T3>& data)
{
    ar(data.data_vec_, data.data_map_);
}
```

```
// 注册 UseDefTemplateDataType<int, float>
HIAI_REGISTER_TEMPLATE_DATA_TYPE("UseDefTemplateDataType_uint64_t_float_string",
UseDefTemplateDataType, uint64_t, float, std::string);

//... 注册其他类型

// 注册 UseDefTemplateDataType<int, int>
HIAI_REGISTER_TEMPLATE_DATA_TYPE("UseDefTemplateDataType_uint64_t_uint64_t_uint64_t",
UseDefTemplateDataType, uint64_t, uint64_t, uint64_t);
```

```
// 注册结构体的序列化和反序列化
typedef struct tagST_SC_IMAGE
{
    INT32  iWidth;                  // image width
    INT32  iHeight;                 // image height
    INT32  iWidthStep;              // Size of aligned image row in bytes
    INT32  iChannel;                // channels
    INT32  iDepth;                  // depth in bits
    UINT32 uiTimeStampH;
    UINT32 uiTimeStampL;
    UINT32 iSize;
    UINT64 uiID;
    EN_SC_IMAGE_TYPE eType;          // image type
    ST_SC_IMAGE_ROI *roi;            // Image ROI. If NULL, the whole image is selected
    UINT8*  pucImageData;            // Image data
    UINT64 uiPhyAddr;
}ST_SC_IMAGE;

std::shared_ptr<ST_SC_IMAGE> st_image = std::make_shared<ST_SC_IMAGE>();
st_image->iWidth = 1080;
st_image->iHeight = 1080;
st_image->iChannel = 1;
st_image->iWidthStep = 1080;
st_image->iSize = 1080*1080*3/2 ;
st_image->eType = SC_IMAGE_YUV420SP;
st_image->roi = (ST_SC_IMAGE_ROI*)malloc(sizeof(ST_SC_IMAGE_ROI));
st_image->pucImageData = nullptr;
uint8_t* align_buffer = nullptr;
HIAI_StatusT get_ret =  HIAIMemory::HIAI_DMalloc(st_image->iSize,(void*&)align_buffer,10000);
hiai::EnginePortID engine_id;
engine_id.graph_id = 1;
engine_id.engine_id = 1;
engine_id.port_id = 0;
HIAI_SendData(engine_id, "ST_SC_IMAGE", std::static_pointer_cast<void>(st_image),100000);

void GetStScImageSearPtr(void* inputPtr, std::string& ctrlStr, uint8_t*& dataPtr, uint32_t&
dataLen)

{
    // 如果结构体内有除了image之外的多个指针， 则进行拼接
    ST_SC_IMAGE* st_image = (ST_SC_IMAGE*)inputPtr;

    ctrlStr = std::string((char*)inputPtr, sizeof(ST_SC_IMAGE));
    if(nullptr != st_image->roi)
    {
        std::string image_roi_str = std::string((char*)st_image->roi, sizeof(ST_SC_IMAGE_ROI));
        ctrlStr += image_roi_str;
    }

    dataPtr = (UINT8*)st_image->pucImageData;
    dataLen= st_image->iSize;
}

std::shared_ptr<void> GetStScImageDearPtr(const char* ctrlPtr, const uint32_t& ctrlLen, const
uint8_t* dataPtr, const uint32_t& dataLen)
{
    ST_SC_IMAGE* st_sc_image = (ST_SC_IMAGE*)ctrlPtr;
    std::shared_ptr<hiai::BatchImagePara<uint8_t>> ImageInfo(new hiai::BatchImagePara<uint8_t>);
    hiai::ImageData<uint8_t> image_data;
```



```
    image_data.width = st_sc_image->iWidth;
    image_data.height = st_sc_image->iHeight;
    image_data.channel = st_sc_image->iChannel;
    image_data.width_step = st_sc_image->iWidthStep;
    if (st_sc_image->eType == SC_IMAGE_U8C3PLANAR)
    {
       image_data.size = st_sc_image->iWidth * st_sc_image->iHeight * 3;
       image_data.format = hiai::BGR888;
    }
    else if (SC_IMAGE_U8C3PACKAGE == st_sc_image->eType)
    {
       image_data.size = st_sc_image->iWidth * st_sc_image->iHeight * 3;
       image_data.format = hiai::BGR888;
    }
    else if (st_sc_image->eType == SC_IMAGE_YUV420SP)
    {
       image_data.size = st_sc_image->iSize;//st_sc_image->iWidth * st_sc_image->iHeight * 3 / 2;
       image_data.format = hiai::YUV420SP;
    }
    image_data.data.reset(dataPtr, hiai::Graph::ReleaseDataBuffer);
    ImageInfo->v_img.push_back(image_data);
    ImageInfo->b_info.frame_ID.push_back(0);
    ImageInfo->b_info.batch_size = ImageInfo->b_info.frame_ID.size();

    return std::static_pointer_cast<void>(ImageInfo);
}

HIAI_REGISTER_SERIALIZE_FUNC("ST_SC_IMAGE", ST_SC_IMAGE, GetStScImageSearPtr, GetStScImageDearPtr);
```

5.3 内存管理

5.3.1 HIAIMemory::HIAI_DMalloc（c++专用接口）

通过HiAI接口申请内存块，配合高效数据传输使用。

使用场景：

通过HiAI Engine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。
- 如申请内存后，使用SendData发送数据，框架会将申请的内存纳入内存池进行统一管理，程序运行结束后释放。
- 如果在调用HIAI_DMalloc时参数选择申请大页内存，在SendData结束后不会将该块内存纳入内存池，需要手动调用HIAI_DFree释放该内存。
- 使用HiAI接口分配内存，可申请的内存大小限制为256K～128M。

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 = HIAI_MEMORY_ATTR_NONE)

参数说明

参数	说明	取值范围
dataSize	内存块大小。	256K - 128M
dataBuffer	内存指针。	-
timeOut	当内存申请失败时，提供时延进行阻塞等待有空余内存，默认值为MALLOC_DEFAULT_TIME_OUT（表示100000毫秒）。	-

flag	若flag为HIAI_MEMORY_ATTR_NONE（默认值），则分配普通内存。若flag为HIAI_MEMORY_HUGE_PAGE则从大页中分配，但此内存空间有限，仅建议给DVPP输出内存使用。	typedef enum {
		HIAI_MEMORY_ATTR_NONE = 0,
		// 大页内存
		HIAI_MEMORY_HUGE_PAGE = (0x1 << 0),
		IAI_MEMORY_ATTR_MAX
		} HIAI_MEMORY_ATTR;

返回值

返回的部分错误码请参见“错误码示例”中的“错误码”列。

错误码示例

序号	错误码	错误码描述
1	HIAI_OK	running ok
2	HIAI_GRAPH_NOT_EXIST	graph not exist
3	HIAI_GRAPH_MEMORY_POOL_NOT_EXISTED	memory pool is not existed
4	HIAI_GRAPH_MALLOC_LARGER	failed to malloc buffer due to the size larger than 128M
5	HIAI_MEMORY_POOL_UPDATE_FAILED	failed to update memory pool
6	HIAI_GRAPH_SENDMSG_FAILED	hdc send msg failed
7	HIAI_GRAPH_MEMORY_POOL_INITED	hdc send msg failed

序号	错误码	错误码描述
8	HIAI_GRAPH_NO_MEMORY	no memory

5.3.2 HIAIMemory::HIAI_DFree（c++专用接口）

释放通过HiAI接口申请的内存。

使用场景：

当用户通过**5.3.1 HIAIMemory::HIAI_DMalloc（c++专用接口）**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， 如果已经调用SendData接口，则不需要调用该接口。

说明

- 该接口只用于释放通过HiAI接口申请的内存。
- 如果通过HiAI接口申请内存后， 配合使用SendData发送了该内存块， 无需调用该接口释放。

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_FOND_MEMORY	can't find the memory

5.3.3 HIAI_DMalloc（C 语言与 C++通用接口）

通过HiAI接口申请内存块，配合高效数据传输使用。

使用场景：

通过HiAI Engine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。
- 如申请内存后， 使用SendData发送数据，框架会自动释放该内存， 无需使用**5.3.4 HIAI_DFree（C语言与C++通用接口）**释放。
- 使用HiAI接口分配内存, 可申请的内存大小限制为256K～128M。

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。	256K - 128M
timeOut	当内存申请失败时，提供时延进行阻塞等待有空余内存。	-

flag	若flag为HIAI_MEMORY_ATTR_NONE，则分配普通内存。
	若flag为HIAI_MEMORY_HUGE_PAGE则从大页中分配，但此内存空间有限，仅建议给DVPP输出内存使用。
		typedef enum {
		HIAI_MEMORY_ATTR_NONE = 0,
		// 大页内存
		HIAI_MEMORY_HUGE_PAGE = (0x1 << 0),
		IAI_MEMORY_ATTR_MAX
		} HIAI_MEMORY_ATTR;

返回值

使用HIAI_DMalloc接口申请到的内存地址。

5.3.4 HIAI_DFree（C 语言与 C++通用接口）

释放通过HiAI接口申请的内存。

使用场景：

当用户通过**5.3.3 HIAI_DMalloc（C语言与C++通用接口）**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， 如果已经调用SendData接口，则不需要调用该接口。

说明

- 该接口只用于释放通过HiAI接口申请的内存。
- 如果通过HiAI接口申请内存后， 配合使用SendData发送了该内存块， 无需调用该接口释放。

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_FOND_MEMORY	can't find the memory

5.4 日志

5.4.1 错误码注册

5.4.1.1 宏 HIAI_DEF_ERROR_CODE

注册错误码。

本宏封装用到了以下函数：

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


函数格式

HIAI_DEF_ERROR_CODE(moduleId, logLevel, codeName, codeDesc)

参数说明

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

返回值

无。

5.4.1.2 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
enum
{
    HIAI_INVALID_INPUT_MSG_CODE = 0
};
HIAI_DEF_ERROR_CODE(USE_DEFINE_ERROR, HIAI_ERROR, HIAI_INVALID_INPUT_MSG, \
    "invalid input message pointer");

#endif
```

5.4.2 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。

5.4.2.1 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* _func_,const char* _file_,int _line_,

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

参数说明

参数	说明	取值范围
graph	Graph对象指针	-
errorCode	错误码	-
format	log描述	-

调用示例

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

5.4.2.2 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* _func_,const char* _file_,int _line_,

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	-

调用示例

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


5.4.2.3 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* _func_,const char* _file_,int _line_,
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	-

调用示例

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

5.4.2.4 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* _func_,const char* _file_,int _line_,
const char* format, ...);
```

参数说明

参数	说明	取值范围
format	log描述	-

调用示例

```
HIAI_ENGINE_LOG（"RUNNING OK"）;
```

5.4.2.5 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* _func_,const char* _file_,int _line_,

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	-

调用示例

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

5.4.2.6 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* _func_,const char* _file_,int _line_,

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

参数说明

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

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"）;
```


5.4.2.7 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* _func_,const char* _file_,int _line_,
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	-

调用示例

```
HIAI_ENGINE_LOG（MODID_OTHER, HIAI_OK, ″RUNNING OK″）;
```

5.4.2.8 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* _func_,const char* _file_,int _line_,
const int32_t module, const char* format, ...);
```

参数说明

参数	说明	取值范围
module	模块名枚举ID	-
format	log描述	-

调用示例

```
HIAI_ENGINE_LOG（MODID_OTHER, ″RUNNING OK″）;
```


5.5 队列管理 MultiTypeQueue 接口

支持多种类型的消息队列保存。

5.5.1 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。

5.5.2 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）。

5.5.3 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）。

5.5.4 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）。

5.5.5 PopAllData

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

函数格式

template<typename T1>

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

template<typename T1, typename T2>

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

.....

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

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

std::shared_ptr<T2>& arg2, std::shared_ptr<T3>& arg3,

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

std::shared_ptr<T6>& arg6, std::shared_ptr<T7>& arg7,

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

std::shared_ptr<T16>& arg16)

参数说明

参数	说明	取值范围
arg1	队列函数指针	-
...	队列函数指针	-
arg16	队列函数指针	-

返回值

仅当所有队列头部都有数据时才读取成功并返回true，否则返回false。

5.5.6 接口使用示例

见**3.7 调用示例**。

5.6 事件注册接口

5.6.1 GraphImpl::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）；

函数格式

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

参数说明

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

返回值

成功返回HIAI_OK, 否则则为失败；

5.7 其他

5.7.1 Graph::UpdateEngineConfig

通过HiAI接口更新指定Engine的参数。

使用场景：

通过该接口，更新流程运行过程中指定的运行参数。

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

函数格式

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

参数说明

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

返回值

返回的部分错误码请参见“错误码示例”中的“错误码”列。

错误码示例

序号	错误码	错误码描述
1	HIAI_OK	running ok
2	HIAI_GRAPH_NOT_EXIST	graph not exist
3	HIAI_GRAPH_MEMORY_POOL_NOT_EXISTED	memory pool is not existed
4	HIAI_GRAPH_MALLOC_LARGER	failed to malloc buffer due to the size larger than 128M
5	HIAI_MEMORY_POOL_UPDATE_FAILED	failed to update memory pool
6	HIAI_GRAPH_SENDMSG_FAILED	hdc send msg failed
7	HIAI_GRAPH_MEMORY_POOL_INITED	hdc send msg failed
8	HIAI_GRAPH_NO_MEMORY	no memory

5.7.2 DataRecvInterface::RecvData

设置回调的接口类函数。

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

5.7.3 Graph::SetPublicKeyForSignatureEncryption

该接口用于设置公钥，该公钥用于对Device侧HiAI Engine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。

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HiAI Engine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-----
```

```
MIIBCAKCAQEAwEWUHPTRBuYAC8JcBfXcJuWLPB/a35U7SPXGZpEec66gHpjnCQSX
X6uvDTG+k+JZ5+Ytl2wAIOHqAm6eyFEmQ9w5HTb3OZgernYs7gPWdkR2pWslpOY+
LQWmvMSb5CStc+m41Tz7oLlfhRLyW9C8KA+WGbCDaIBwR2OOrPFqORHyRG9i5mzq
1A6IfsaJLgexeOUGoDBAoTpU4bh6lQvE9FroRt/DjeTRzXQOi6pwOnaV714i69B1
yMHfwvoNucy4jeHxI1hWQajMUQBzGFGPKPPuUl648UcvXbkrF3rExn5/N12a+3G2
/eM+igulhfUTEs9HQw+W7HaxjFxFyWywbQIBAw==
-----END RSA PUBLIC KEY-----"
auto ret = Graph::SetPublicKeyForSignatureEncryption(publicKey);
```


6 附录

6.1 HiAI Engine 数据类型

6.1.1 Protobuffer 数据类型

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

HiAI Engine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
    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HiAI Engine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; //HiAI Engine 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 EngineConfig
{
    enum RunSide
    {
        DEVICE=0;   //运行在DEVICE侧
        HOST=1;      //运行在host侧
    }

    uint32 id =1;   //Engine ID (节点)
    string engine_name =2; //Engine 节点名字
    repeated string so_name=3; //需要拷贝的所有动态库so文件名列表，会先按照用户配置的顺序先进行加载，加载失败再按照其他的加载顺序进行尝试加载
    RunSide side=4; //部署在host侧还是Device侧
    uint32 thread_num = 5;//线程数量
    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
    uint32 holdModelFileFlag = 13; //是否保留本engine的模型文件，0不保留，非0保留
}
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
}
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
    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;  //HiAI Engine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;
}
```

6.1.2 HiAI Engine 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：HIAIEngine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;
    };
```

6.2 HiAI Engine 已经注册的数据结构

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

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


    // name:"vector_uint8_t"                        type:vector<uint8_t>
    // name:"vector_float"                          type:vector<float>


    // name:"ImageData_uint8_t"                     type:ImageData<uint8_t>
    // name:"ImageData_float"                       type:ImageData<float>


    // name:"ImagePara_uint8_t"                     type:ImagePara<uint8_t>
    // name:"ImagePara_float"                       type:ImagePara<float>


    // name:"BatchImagePara_uint8_t"                type:BatchImagePara<uint8_t>
    // name:"BatchImagePara_float"                  type:BatchImagePara<float>


    // name:"Line_uint8_t"                          type:Line<uint8_t>
    // name:"Line_float"                            type:Line<float>


    // name:"Rectangle_uint8_t"                     type:Rectangle<uint8_t>
    // name:"Rectangle_float"                       type:Rectangle<float>


    // name:"Polygon_uint8_t"                       type:Polygon<uint8_t>
    // name:"Polygon_float"                         type:Polygon<float>


    // name:"MaskMatrix_uint8_t"                    type:MaskMatrix<uint8_t>
    // name:"MaskMatrix_float"                      type:MaskMatrix<float>


    // name:"ObjectLocation_uint8_t_uint8_t"        type:ObjectLocation<uint8_t, uint8_t>
    // name:"ObjectLocation_float_float"            type:ObjectLocation<float, uint8_t>


    // name:"DetectedObjectPara_uint8_t_uint8_t"    type:DetectedObjectPara<uint8_t, uint8_t>
    // name:"DetectedObjectPara_float_float"        type:DetectedObjectPara<float, float>
```



```
    // name:"BatchDetectedObjectPara_uint8_t_uint8_t"
type:BatchDetectedObjectPara<uint8_t,uint8_t>
    // name:"BatchDetectedObjectPara_float_float"
type:BatchDetectedObjectPara<float,float>
    // name:"BatchDetectedObjectPara_Rectangle_Point2D_int32_t"
type:BatchDetectedObjectPara<Rectangle<Point2D>, int32_t>
    // name:"BatchDetectedObjectPara_Rectangle_Point2D_float"
type:BatchDetectedObjectPara<Rectangle<Point2D>, float>


    // name:"RegionImage_uint8_t"                  type:RegionImage<uint8_t>
    // name:"RegionImage_float"                    type:RegionImage<float>


    // name:"RegionImagePara_uint8_t"              type:RegionImagePara<uint8_t>
    // name:"RegionImagePara_float"                type:RegionImagePara<float>


    // name:"BatchRegionImagePara_uint8_t"         type:BatchRegionImagePara<uint8_t>
    // name:"BatchRegionImagePara_float"           type:BatchRegionImagePara<float>


    // name:"Attribute_uint8_t"                    type:Attribute<uint8_t>
    // name:"Attribute_float"                      type:Attribute<float>


    // name:"AttributeTopN_uint8_t"                type:AttributeTopN<uint8_t>
    // name:"AttributeTopN_float"                  type:AttributeTopN<float>


    // name:"AttributeVec_uint8_t"                 type:AttributeVec<uint8_t>
    // name:"AttributeVec_float"                   type:AttributeVec<float>


    // name:"AttributeResultPara_uint8_t"          type: AttributeVec<uint8_t>
    // name:"AttributeResultPara_float"            type:AttributeVec<float>


    // name:"BatchAttributeResultPara_uint8_t"     type:AttributeVec<uint8_t>
    // name:"BatchAttributeResultPara_float"       type:AttributeVec<float>


    // name:"Feature_uint8_t"                      type:AttributeVec<uint8_t>
    // name:"Feature_float"                        type:AttributeVec<float>


    // name:"FeatureList_uint8_t"                  type:FeatureList<uint8_t>
    // name:"FeatureList_float"                    type:FeatureList<float>


    // name:"FeatureVec_uint8_t"                   type:FeatureVec<uint8_t>
    // name:"FeatureVec_float"                     type:FeatureVec<float>


    // name:"FeatureResultPara_uint8_t"            type:FeatureResultPara<uint8_t>
    // name:"FeatureResultPara_float"              type:FeatureResultPara<float>


    // name:"BatchFeatureResultPara_uint8_t"       type:BatchFeatureResultPara<uint8_t>
    // name:"BatchFeatureResultPara_float"         type:BatchFeatureResultPara<float>


    // name:"FeatureRecord_uint8_t"                type:FeatureRecord<uint8_t>
    // name:"FeatureRecord_float"                  type:FeatureRecord<float>


    // name:"RetrievalSet_uint8_t_uint8_t"         type:RetrievalSet<uint8_t, uint8_t>
```



```
// name:"RetrievalSet_float_float"              type:RetrievalSet<float, float>


// name:"EvaluationResult_uint8_t"              type:EvaluationResult<uint8_t>
// name:"EvaluationResult_float"                type:EvaluationResult<float>


// name:"EvaluationResultVec_uint8_t"           type:EvaluationResultVec<uint8_t>
// name:"EvaluationResultVec_float"             type:EvaluationResultVec<float>


// name:"EvaluationResultPara_uint8_t"          type:EvaluationResultPara<uint8_t>
// name:"EvaluationResultPara_float"            type:EvaluationResultPara<float>


// name:"BatchEvaluationResultPara_uint8_t"     type:BatchEvaluationResultPara<uint8_t>
// name:"BatchEvaluationResultPara_float"       type:BatchEvaluationResultPara<float>


// name:"Classification_uint8_t"                type:Classification<uint8_t>
// name:"Classification_float"                  type:Classification<float>


// name:"ClassificationTopN_uint8_t"            type:ClassificationTopN<uint8_t>
// name:"ClassificationTopN_float"              type:ClassificationTopN<float>


// name:"ClassificationVec_uint8_t"             type:ClassificationVec<uint8_t>
// name:"ClassificationVec_float"               type:ClassificationVec<float>


// name:"ClassificationResultPara_uint8_t"      type:ClassificationResultPara<uint8_t>
// name:"ClassificationResultPara_float"        type:ClassificationResultPara<float>


// name:"BatchClassificationResultPara_uint8_t" type:BatchClassificationResultPara<uint8_t>
// name:"BatchClassificationResultPara_float"   type:BatchClassificationResultPara<float>


////////////////////////////////////////////////////
//////////////     batch信息     ////////////////////
////////////////////////////////////////////////////
struct BatchInfo {
    bool is_first = false;                    // 是否为第一个batch
    bool is_last = false;                     // 是否为最后一个batch
    uint32_t batch_size = 0;                // 当前batch的实际大小
    uint32_t max_batch_size = 0;            // batch预设大小（最大容量
    uint32_t batch_ID = 0;                  // 当前batch ID号
    uint32_t channel_ID = 0;                // 处理当前batch的通道ID号
    uint32_t processor_stream_ID = 0;       // 处理器计算流ID号


    std::vector<uint32_t> frame_ID;         // batch中图像帧ID号
    std::vector<uint32_t> source_ID;        // batch中图像源ID号


    std::vector<uint64_t> timestamp;        // batch中图像的时间戳
};


template<class Archive>
void serialize(Archive& ar, BatchInfo& data) {
    ar(data.is_first, data.is_last, data.batch_size,
        data.max_batch_size, data.batch_ID, data.channel_ID,
        data.processor_stream_ID, data.frame_ID, data.source_ID,
        data.timestamp);
}
```



```
////////////////////////////////////////////////////
//////////////     frame信息     ///////////////////
////////////////////////////////////////////////////
struct FrameInfo {
    bool is_first = false;                    // 是否为第一个frame
    bool is_last = false;                     // 是否为最后一个frame
    uint32_t channel_ID = 0;                // 处理当前frame的通道ID号
    uint32_t processor_stream_ID = 0;      // 处理器计算流ID号
    uint32_t frame_ID = 0;                  // 图像帧ID号
    uint32_t source_ID = 0;                 // 图像源ID号
    uint64_t timestamp = 0;                 // 图像的时间戳
};


template<class Archive>
void serialize(Archive& ar, FrameInfo& data) {
    ar(data.is_first, data.is_last, data.channel_ID,
        data.processor_stream_ID, data.frame_ID, data.source_ID,
        data.timestamp);
}


////////////////////////////////////////////////////
/////////////////  1. image tensor ////////////////////
////////////////////////////////////////////////////


// 图像格式
enum IMAGEFORMAT {
    RGB565,           //  Red 15:11, Green 10:5, Blue 4:0
    BGR565,           //  Blue 15:11, Green 10:5, Red 4:0
    RGB888,           //  Red 24:16, Green 15:8, Blue 7:0
    BGR888,           //  Blue 24:16, Green 15:8, Red 7:0
    BGRA8888,         //  Blue 31:24, Green 23:16, Red 15:8, Alpha 7:0
    ARGB8888,         //  Alpha 31:24, Red 23:16, Green 15:8, Blue 7:0
    RGBX8888,
    XRGB8888,
    YUV420Planar,     //  I420
    YVU420Planar,     //  YV12
    YUV420SP,         //  NV12
    YVU420SP,         //  NV21
    YUV420Packed,     //  YUV420 Interleaved
    YVU420Packed,     //  YVU420 Interleaved
    YUV422Planar,     //  Three arrays Y,U,V.
    YVU422Planar,
    YUYVPacked,       //  422 packed per payload in planar slices
    YVYUPacked,       //  422 packed
    UYVYPacket,       //  422 packed
    VYUYPacket,       //  422 packed
    YUV422SP,         //  422 packed
    YVU422SP,
    YUV444Interleaved,  //  Each pixel contains equal parts YUV
    Y8,
    Y16,
    RAW
};


enum OBJECTTYPE {
    OT_VEHICLE,
    OT_HUMAN,
    OT_NON_MOTOR,
    OT_FACE,
    OT_FACE_BODY,
    OT_PLATE
};


template<class T>   // T: uint8_t  float
```

```
struct ImageData {
    IMAGEFORMAT format;   // 图像格式


    uint32_t width  = 0;       // 图像宽
    uint32_t height = 0;      // 图像高
    uint32_t channel = 0;     // 图像通道数
    uint32_t depth = 0;       // 位深
    uint32_t height_step = 0;  // 对齐高度
    uint32_t width_step = 0;  // 对齐宽度
    uint32_t size = 0;        // 数据大小（Byte
    std::shared_ptr<T> data;   // 数据指针
};


template<class Archive, class T>
void serialize(Archive& ar, ImageData<T>& data) {
    ar(data.format, data.width, data.height, data.channel,
        data.depth, data.height_step, data.width_step, data.size);
    if (data.size > 0 && data.data.get() == nullptr) {
        data.data.reset(new(std::nothrow) T[data.size]);
    }
    ar(cereal::binary_data(data.data.get(), data.size * sizeof(T)));
}


// 参数
template<class T>
struct ImagePara {
    FrameInfo f_info;          // frame信息
    ImageData<T> img;          // 图像
};


template<class Archive, class T>
void serialize(Archive& ar, ImagePara<T>& data) {
    ar(data.f_info, data.img);
}


// 参数
template<class T>
struct BatchImagePara {
    BatchInfo b_info;                  // batch信息
    std::vector<ImageData<T> > v_img;   // batch中的图像
};


template<class Archive, class T>
void serialize(Archive& ar, BatchImagePara<T>& data) {
    ar(data.b_info, data.v_img);
}




// 角度
typedef struct {
    float x;
    float y;
    float z;
}Angle;


template<class Archive>
void serialize(Archive& ar, Angle& data) {
    ar(data.x, data.y, data.z);
}
```



```
    // 2D点
    struct Point2D {
        int32_t x;
        int32_t y;
    };


    template<class Archive>
    void serialize(Archive& ar, Point2D& data) {
        ar(data.x, data.y);
    }


    ////////////////////////////////////////////////////
    //////////////     ROI信息     ////////////////////
    ////////////////////////////////////////////////////


    struct RoiPolygon {
        uint32_t         uiPtNum;       //多边形顶点数
        std::vector<Point2D> astPts;    //多个检索结果（TopN）
    };


    template<class Archive>
    void serialize(Archive &ar, RoiPolygon &data) {
        ar(data.uiPtNum, data.astPts);
    }


    struct ArgsRoiPolygon {
        RoiPolygon stRoiPolygon; // 配置项Roi
        uint32_t iMinFace; // 最小人脸26*26
        uint32_t iMaxFace; // 最大人脸300*300
        uint32_t imgWidth;  // 视频帧图像宽
        uint32_t imgHeight; // 视频帧图像高
    };


    template<class Archive>
    void serialize(Archive &ar, ArgsRoiPolygon &data) {
        ar(data.stRoiPolygon, data.iMinFace, data.iMaxFace, data.imgWidth, data.imgHeight,
data.iMinFace);
    }
    // 3D点
    struct Point3D {
        int32_t x;
        int32_t y;
        int32_t z;
    };


    template<class Archive>
    void serialize(Archive& ar, Point3D& data) {
        ar(data.x, data.y, data.z);
    }


    // 2D或3D空间上的线
    template<class T>   // T: Point2D  Point3D
    struct Line {
        T start;
        T end;
    };


    template<class Archive, class T>
    void serialize(Archive& ar, Line<T>& data) {
```



```
    ar(data.start, data.end);
}


// 2D或3D空间上的矩形平面
template<class T>  // T: Point2D  Point3D
struct Rectangle {
    T anchor_lt;
    T anchor_rb;
};


template<class Archive, class T>
void serialize(Archive& ar, Rectangle<T>& data) {
    ar(data.anchor_lt, data.anchor_rb);
}


// 2D或3D空间上的多边形平面
template<class T>  // T: Point2D  Point3D
struct Polygon {
    std::vector<T> anchors;
};


template<class Archive, class T>
void serialize(Archive& ar, Polygon<T>& data) {
    ar(data.anchors);
}


// 立方体
struct ROICube {
    Point3D anchor;
    uint32_t length;
    uint32_t width;
    uint32_t height;
};


template<class Archive>
void serialize(Archive& ar, ROICube& data) {
    ar(data.anchor, data.length, data.width, data.height);
}


// 掩膜矩阵
template<class T>    // T: int32_t float etc.
struct MaskMatrix {
    uint32_t cols;
    uint32_t rows;
    std::shared_ptr<T> data;
};


template<class Archive, class T>
void serialize(Archive& ar, MaskMatrix<T>& data) {
    ar(data.cols, data.rows);
    if (data.cols*data.rows > 0 && data.data.get() == nullptr) {
        data.data.reset(new(std::nothrow) T[data.cols*data.rows]);
    }
    ar(cereal::binary_data(data.data.get(),
        data.cols * data.rows * sizeof(T)));
}


// RLE编码
struct RLECode {
    uint32_t len;
```



```
    uint32_t cols;
    uint32_t rows;
    std::shared_ptr<uint32_t> data;
};


template<class Archive>
void serialize(Archive& ar, RLECode& data) {
    ar(data.len, data.cols, data.rows);
    if (data.len > 0 && data.data.get() == nullptr) {
        data.data.reset(new(std::nothrow) uint32_t[data.len]);
    }
    ar(cereal::binary_data(data.data.get(),
        data.len * sizeof(uint32_t)));
}


template<class T1, class T2>    // T1: Point2D Rectangle RLECode etc.
struct ObjectLocation {          // T2: int8_t float etc.
    std::vector<uint32_t> v_obj_id;
    std::vector<T1> range;           // 目标范围描述
    std::vector<Angle> angle;        // 角度信息
    std::vector<T2> confidence;      // 置信度
    std::vector<uint32_t> label;     // 目标类型标签
};


template<class Archive, class T1, class T2>
void serialize(Archive& ar, ObjectLocation<T1, T2>& data) {
    ar(data.v_obj_id, data.range, data.angle,
       data.confidence, data.label);
}


// 参数
template<class T1, class T2>
struct DetectedObjectPara {
    FrameInfo f_info;                   // frame信息
    ObjectLocation<T1, T2> location;     // 多个目标位置
};


template<class Archive, class T1, class T2>
void serialize(Archive& ar, DetectedObjectPara<T1, T2>& data) {
    ar(data.f_info, data.location);
}


// 参数
template<class T1, class T2>
struct BatchDetectedObjectPara {
    //  batch信息
    BatchInfo b_info;
    //  对应每帧图像的多个目标位置
    std::vector<ObjectLocation<T1, T2> > v_location;
};


template<class Archive, class T1, class T2>
void serialize(Archive& ar, BatchDetectedObjectPara<T1, T2>& data) {
    ar(data.b_info, data.v_location);
}







//////////////////////////////////////////////////////
```



```
/////////////// 3. region image tensor ////////////////
////////////////////////////////////////////////////////


template<class T>
struct RegionImage {
    std::vector<ImageData<T> > region;
    std::vector<uint32_t> v_obj_id;
};


template<class Archive, class T>
void serialize(Archive& ar, RegionImage<T>& data) {
    ar(data.v_obj_id, data.region);
}


// 参数
template<class T>
struct RegionImagePara {
    FrameInfo f_info;             // frame信息
    RegionImage<T> region;        // 每帧对应的小图
};


template<class Archive, class T>
void serialize(Archive& ar, RegionImagePara<T>& data) {
    ar(data.f_info, data.region);
}


// 参数
template<class T>
struct BatchRegionImagePara {
    // batch信息
    BatchInfo b_info;
    // 每帧图像若干region image，用一个指向Image数组的指针来引用
    std::vector<RegionImage<T>> v_region;
};


template<class Archive, class T>
void serialize(Archive& ar, BatchRegionImagePara<T>& data) {
    ar(data.b_info, data.v_region);
}




// 递归ID树节点
struct IDTreeNode {
    std::vector<uint32_t>  nodeID;     // 节点上的ID值
    std::vector<bool> is_leaf;         // 是否为叶子节点
    std::vector<IDTreeNode> node;     // 子节点结构指针
};


template <class Archive>
void serialize(Archive& ar, IDTreeNode& data) {
    ar(data.nodeID, data.is_leaf, data.node);
}


// 参数
struct IDTreePara {
    FrameInfo f_info;    // frame信息
    IDTreeNode tree;     // 图像中的ID Tree
};
```



```
template <class Archive>
void serialize(Archive& ar, IDTreePara& data) {
    ar(data.f_info, data.tree);
}


// 参数
struct BatchIDTreePara {
    BatchInfo b_info;                   // batch信息
    std::vector<IDTreeNode> v_tree;     // 每帧图像中的ID Tree
};


template <class Archive>
void serialize(Archive& ar, BatchIDTreePara& data) {
    ar(data.b_info, data.v_tree);
}




template<class T>
struct Attribute {
    std::string attr_value;    // ÊôÐÔÈ¡Öµ
    T score;               // ÖÃÐÅ¶È
};


template <class Archive, class T>
void serialize(Archive& ar, Attribute<T>& data) {
    ar(data.attr_value, data.score);
}


template<class T>
struct AttributeTopN {
    uint32_t obj_ID;
    std::map<std::string, std::vector<Attribute<T>> > attr_map;
};


template <class Archive, class T>
void serialize(Archive& ar, AttributeTopN<T>& data) {
    ar(data.obj_ID, data.attr_map);
}


template<class T>
struct AttributeVec {
    std::vector<AttributeTopN<T>> obj_attr;     // ¶à,öÄ¿±êµÄÊôÐÔ
};


template <class Archive, class T>
void serialize(Archive& ar, AttributeVec<T>& data) {
    ar(data.obj_attr);
}


// ²ÎÊý
template<class T>
struct AttributeResultPara {
    FrameInfo f_info;         // frameÐÅÏ¢
    AttributeVec<T> attr;     // Í¼ÏñµÄ¶à,öÄ¿±êµÄÊôÐÔ
};


template <class Archive, class T>
```

```
void serialize(Archive& ar, AttributeResultPara<T>& data) {
    ar(data.f_info, data.attr);
}



// ²ÎÊý
template<class T>
struct BatchAttributeResultPara {
    BatchInfo b_info;                          // batchÐÅÏ¢
    std::vector<AttributeVec<T>> v_attr;
};


template <class Archive, class T>
void serialize(Archive& ar, BatchAttributeResultPara<T>& data) {
    ar(data.b_info, data.v_attr);
}


template<class T>
struct Feature {
    int32_t len;                // 特征向量长度
    std::shared_ptr<T> feature;      // 特征向量指针
};


template<class Archive, class T>
void serialize(Archive& ar, Feature<T>& data) {
    ar(data.len);
    if (data.len > 0 && data.feature.get() == nullptr) {
        data.feature.reset(new(std::nothrow) T[data.len]);
    }
    ar(cereal::binary_data(data.feature.get(), data.len * sizeof(T)));
}



template<class T>
struct FeatureList {
    uint32_t obj_ID;                // 目标ID
    std::vector<Feature<T>> feature_list;    // 目标的多个特征
};


template<class Archive, class T>
void serialize(Archive& ar, FeatureList<T>& data) {
    ar(data.obj_ID, data.feature_list);
}


template<class T>
struct FeatureVec {
    std::vector<FeatureList<T>> obj_feature;   // 多个目标的属性
};


template<class Archive, class T>
void serialize(Archive& ar, FeatureVec<T>& data) {
    ar(data.obj_feature);
}


// 参数
template<class T>
struct FeatureResultPara {
    FrameInfo f_info;         // frame信息
```



```
    FeatureVec<T> feature;     // 图像的多个目标的特征
};


template<class Archive, class T>
void serialize(Archive& ar, FeatureResultPara<T>& data) {
    ar(data.f_info, data.feature);
}


// 参数
template<class T>
struct BatchFeatureResultPara {
    // batch信息
    BatchInfo b_info;
    // 对应每帧图像的多个目标的特征
    std::vector<FeatureVec<T>> v_feature;
};


template<class Archive, class T>
void serialize(Archive& ar, BatchFeatureResultPara<T>& data) {
    ar(data.b_info, data.v_feature);
}




template<class T>
struct FeatureRecord {
    uint32_t ID;            // 特征图ID
    uint32_t len;           // 特征向量长度
    std::shared_ptr<T> feature;  // 特征向量指针
};


template<class Archive, class T>
void serialize(Archive& ar, FeatureRecord<T>& data) {
    ar(data.ID, data.len);
    if (data.len > 0 && data.feature.get() == nullptr) {
        data.feature.reset(new(std::nothrow) T[data.len]);
    }
    ar(cereal::binary_data(data.feature.get(), data.len * sizeof(T)));
}


template<class T1, class T2>
struct RetrievalSet {
    uint32_t TopN;       // TopN结果设置
    T1 threshold;        // 相似度阈值
    std::vector<FeatureRecord<T2>> record;    // 特征集合中所有的记录
};


template<class Archive, class T1, class T2>
void serialize(Archive& ar, RetrievalSet<T1, T2>& data) {
    ar(data.TopN, data.threshold, data.record);
}


// 参数
template<class T1, class T2>
struct RetrievalSetPara {
    RetrievalSet<T1, T2> set;   // 参数和特征集合
};


template<class Archive, class T1, class T2>
void serialize(Archive& ar, RetrievalSetPara<T1, T2>& data) {
```



```
    ar(data.set);
}


struct RetrievalResult {
    uint32_t ID;        // 特征图ID
    float similarity;   // 相似度
};


template<class Archive>
void serialize(Archive& ar, RetrievalResult& data) {
    ar(data.ID, data.similarity);
}


struct RetrievalResultTopN {
    std::vector<RetrievalResult> result_list;    // 多个检索结果（TopN）
};


template<class Archive>
void serialize(Archive& ar, RetrievalResultTopN& data) {
    ar(data.result_list);
}


// 参数
struct RetrievalResultPara {
    FrameInfo f_info;               // frame信息
    RetrievalResultTopN result;     // 检索结果
};


template<class Archive>
void serialize(Archive& ar, RetrievalResultPara& data) {
    ar(data.f_info, data.result);
}


// 参数
struct BatchRetrievalResultPara {
    // batch信息
    BatchInfo b_info;
    // 对应每帧图像的检索结果
    std::vector<RetrievalResultTopN> v_result;
};


template<class Archive>
void serialize(Archive& ar, BatchRetrievalResultPara& data) {
    ar(data.b_info, data.v_result);
}


template<class T>
struct EvaluationResult {
    uint32_t obj_ID;           // 目标ID
    std::string description;    // 描述
    T score;                   // 评价分数
};


template <class Archive, class T>
void serialize(Archive& ar, EvaluationResult<T>& data) {
    ar(data.obj_ID, data.description, data.score);
}
```



```
template<class T>
struct EvaluationResultVec {
    std::vector<EvaluationResult<T>> result;    // 多个目标的评价结果
};


template <class Archive, class T>
void serialize(Archive& ar, EvaluationResultVec<T>& data) {
    ar(data.result);
}


// 参数
template<class T>
struct EvaluationResultPara {
    FrameInfo f_info;                 // frame信息
    EvaluationResultVec<T> result;    // 评价结果
};


template <class Archive, class T>
void serialize(Archive& ar, EvaluationResultPara<T>& data) {
    ar(data.f_info, data.result);
}


// 参数
template<class T>
struct BatchEvaluationResultPara {
    // batch信息
    BatchInfo b_info;
    // 对应每帧图像的评价结果
    std::vector<EvaluationResultVec<T>> v_result;
};


template <class Archive, class T>
void serialize(Archive& ar, BatchEvaluationResultPara<T>& data) {
    ar(data.b_info, data.v_result);
}




template<class T>
struct Classification {
    std::string class_value;    // 类别取值
    T score;                    // 置信度
};


template<class Archive, class T>
void serialize(Archive& ar, Classification<T>& data) {
    ar(data.class_value, data.score);
}


template<class T>
struct ClassificationTopN {
    uint32_t obj_ID;                                // 目标ID
    std::vector<Classification<T>> class_result;   // 分类结果（TopN）
};


template<class Archive, class T>
void serialize(Archive& ar, ClassificationTopN<T>& data) {
    ar(data.obj_ID, data.class_result);
}
```



```
template<class T>
struct ClassificationVec {
    std::vector<ClassificationTopN<T>> class_list;   // 多个目标的分类
};


template<class Archive, class T>
void serialize(Archive& ar, ClassificationVec<T>& data) {
    ar(data.class_list);
}


// 参数
template<class T>
struct ClassificationResultPara {
    FrameInfo f_info;                           // frame信息
    ClassificationVec<T> classification;        // 多个目标的分类
};


template<class Archive, class T>
void serialize(Archive& ar, ClassificationResultPara<T>& data) {
    ar(data.f_info, data.classification);
}


// 参数
template<class T>
struct BatchClassificationResultPara {
    // batch信息
    BatchInfo b_info;
    // 对应每帧图像的多个目标的分类
    std::vector<ClassificationVec<T>> v_class;
};




template<class Archive, class T>
void serialize(Archive& ar, BatchClassificationResultPara<T>& data) {
    ar(data.b_info, data.v_class);
}


struct RawDataBuffer {
    uint32_t len_of_byte;  // size length
    std::shared_ptr<uint8_t> data;  // 一块buffer，可转为用户自定义类型
};


template<class Archive>
void serialize(Archive& ar, RawDataBuffer& data) {
    ar(data.len_of_byte);
    if (data.len_of_byte > 0 && data.data.get() == nullptr) {
        data.data.reset(new(std::nothrow) uint8_t[data.len_of_byte]);
    }
    ar(cereal::binary_data(data.data.get(), data.len_of_byte *
        sizeof(uint8_t)));
}


// common raw databuffer struct
struct BatchRawDataBuffer {
    BatchInfo b_info;  // batch info
    std::vector<RawDataBuffer> v_info;
};
```



```
    template<class Archive>
    void serialize(Archive& ar, BatchRawDataBuffer& data) {
        ar(data.b_info, data.v_info);
    }




    template<typename T> const char* TypeName(void);


#define REGISTER_TYPE_DEFINITION(type) \
    template<> inline const char* TypeName<type>(void) { return #type; }


    REGISTER_TYPE_DEFINITION(int8_t);
    REGISTER_TYPE_DEFINITION(uint8_t);
    REGISTER_TYPE_DEFINITION(int16_t);
    REGISTER_TYPE_DEFINITION(uint16_t);
    REGISTER_TYPE_DEFINITION(int32_t);
    REGISTER_TYPE_DEFINITION(uint32_t);
    REGISTER_TYPE_DEFINITION(int64_t);
    REGISTER_TYPE_DEFINITION(uint64_t);
    REGISTER_TYPE_DEFINITION(float);
    REGISTER_TYPE_DEFINITION(double);
    REGISTER_TYPE_DEFINITION(Point2D);
    REGISTER_TYPE_DEFINITION(Point3D);
    REGISTER_TYPE_DEFINITION(Line<Point2D>);
    REGISTER_TYPE_DEFINITION(Line<Point3D>);
    REGISTER_TYPE_DEFINITION(Rectangle<Point2D>);
    REGISTER_TYPE_DEFINITION(Rectangle<Point3D>);
    REGISTER_TYPE_DEFINITION(Polygon<Point2D>);
    REGISTER_TYPE_DEFINITION(Polygon<Point3D>);
    REGISTER_TYPE_DEFINITION(ROICube);
    REGISTER_TYPE_DEFINITION(MaskMatrix<int8_t>);
    REGISTER_TYPE_DEFINITION(MaskMatrix<int32_t>);
    REGISTER_TYPE_DEFINITION(MaskMatrix<float>);
    REGISTER_TYPE_DEFINITION(RLECode);
    REGISTER_TYPE_DEFINITION(RoiPolygon);
    REGISTER_TYPE_DEFINITION(ArgsRoiPolygon);
}
```

6.3 其他对外接口说明

本章节描述了HiAI Engine提供给其它组件调用的接口，不建议客户直接进行调用。

给其他组件提供的PROFILING统计接口。

```
1.1 获取实例函数
static ProfileStatistic* ProfileStatistic::GetInstance();
1.2 注册数据类型函数
void ProfileStatistic::RegisterType(const uint32_t& typeID, const std::string& profileTypeDes);
1.3 记录开始时间
void ProfileStatistic::RecordStartTime(const uint32_t& profileType);
1.4 记录结束时间
void ProfileStatistic::RecordEndTime(const uint32_t& profileType,const uint64_t& reqID = 0);
1.5 返回Graph的统计数据
std::string ProfileStatistic::ProfileStatisticString(const uint32_t& graphId);
1.6 返回Engine的统计数据
std::string ProfileStatistic::ProfileSingleEngine(const std::vector<StaticInfo>& profileInfoVec);
1.7 记录Graph的开始时间
void ProfileStatistic::RecordGraphStartTime(const uint32_t& graphId);
1.8 记录Graph的结束时间
void ProfileStatistic::RecordGraphEndTime(const uint32_t& graphId);
1.9 添加Engine的ID
void ProfileStatistic::AddEngineID(const uint32_t& threadID, const std::string& engineName, const
uint32_t& graphId);
1.10 判断是否需要写入统计数据
```



```
bool ProfileStatistic::NeedWrite();
1.11 设置性能数据记录开关
static void ProfileStatistic::SetProfileSwitch(bool switchForProfile);
1.12 判断统计功能是否可用
static bool& ProfileStatistic::IsProfileEnable();
1.13 性能统计停止等待
void ProfileStatistic::StopWait();
1.14 获取当前时间
uint64_t ProfileStatistic::ProfileGetCurrentTime();
1.15 获取锁
static std::mutex& ProfileStatistic::GetMutex();
1.16 获取更新锁
static std::mutex& ProfileStatistic::GetUpdateMutex();

1.17 记录统计的开始时间
PROFILESTATIC_STARTTIME(profileType)
1.18 记录统计的结束时间
PROFILESTATIC_ENDTIME(profileType)
1.19 记录Engine统计的结束时间
PROFILESTATIC_ENGINE_ENDTIME(profileType, reqID)
1.20 增加需要统计的Engine
PROFILESTATIC_ADDENGINE(threadID, engineName, graphId)
1.21 增加需要统计的数据类型
PROFILESTATIC_REGTYPE(profileID, profileType, profileDes)
1.22 记录Graph的开始时间
PROFILESTATIC_GRAPH_STARTTIME(graphId)
1.23 记录Graph的结束时间
PROFILESTATIC_GRAPH_ENDTIME(graphId)
1.23 自动注册统计数据类型接口
ProfileTypeRegister::ProfileTypeRegister(const uint32_t& profileType, const std::string&
profileDes)
```

6.4 示例

6.4.1 编排配置示例

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

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”。
- 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”。

```
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
  }
}
```

6.4.2 性能优化传输示例

```
(1)使用性能优化方案传输数据，必须使用对发送数据的接口进行手动序列化和反序列化：
// 注：序列化函数在发送端使用，反序列化在接收端使用，所以这个注册函数最好在接收端和发送端都注册一
遍；

   结构体
```



```
typedef struct
{
    uint32_t frameId;
    uint8_t  bufferId;
    uint8_t* image_data;
    uint8_t  image_size;
}TEST_STR;

// 序列化TEST_STR结构体，该函数只需要调动注册函数进行注册，参数说明：
// 输入：inputPtr，TEST_STR结构体指针
// 输出：ctrlStr，控制信息地址
         imageData，数据信息指针
         imageLen，数据信息大小
void GetTestStrSearPtr(void* inputPtr, std::string& ctrlStr, uint8_t*& imageData, uint32_t&
imageLen)
{
    // 获取结构体buffer
    TEST_STR* test_str = (TEST_STR*)inputPtr;
    ctrlStr = std::string((char*)inputPtr, sizeof(TEST_STR));

    // 获取数据信息，直接将结构体的数据指针赋值传递
    imageData = (uint8_t*)test_str->image_data;
    imageLen = test_str->image_size;
}

// 反序列化结构体，回传回来的将是结构体buffer和数据块buffer
// 输入：ctrlPtr，控制信息地址
         ctrlLen，控制信息大小
         imageData，数据信息指针
         imageLen，数据信息大小
// 输出：std::shared_ptr<void>，指向结构体的智能指针
std::shared_ptr<void> GetTestStrDearPtr(const char* ctrlPtr, const uint32_t& ctrlLen,const
uint8_t* imageData, const uint32_t& imageLen)
{
    // 获取结构体
    TEST_STR* test_str = (TEST_STR*)ctrlPtr;

    // 获取传输过来的大内存数据
    // 注：大内存数据最好赋值给智能指针，并注册删除器，如果不是使用智能指针或者没有注册删除器，则需
要在释放该内存时手动调用hiai::Graph::ReleaseDataBuffer(void* ptr);
    std::shared_ptr<TEST_STR<uint8_t>> shared_data = std::make_shared<TEST_STR<uint8_t>>();
    shared_data->frameId = test_str->frame_ID;
    shared_data->bufferId= test_str->bufferId;
    shared_data->image_size = imageLen;
    shared_data->image_data.reset(imageData, hiai::Graph::ReleaseDataBuffer);

    // 返回智能指针给到Device端Engine使用
    return std::static_pointer_cast<void>(shared_data);
}
// 注册序列化反序列化函数
HIAI_REGISTER_SERIALIZE_FUNC("TEST_STR", TEST_STR, GetTestStrSearPtr, GetTestStrDearPtr);
```

(2) 在发送数据时，需要使用注册的数据类型，另外配合使用HIAI_DMalloc分配数据内存，可以使性能更优
    注：在从host侧向Device侧搬运数据时，使用HIAI_DMalloc方式将很大的提供传输效率，建议优先使用
HIAI_DMalloc，该内存接口目前支持256K - 128M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);
```

