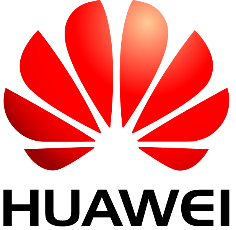


**华为HiAI DDK集成手册**



**文档版本 v2.0**

**发布日期 2018-02-24**

**华为技术有限公司**

|  |  |  |  |
| --- | --- | --- | --- |
|  | 版权所有 © 华为技术有限公司2018。 保留一切权利。  非经本公司书面许可，任何单位和个人不得擅自摘抄、复制本文档内容的部分或全部，并不得以任何形式传播。  商标声明  附件3-版权声明页图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  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。  HiAI申请方式  1.发送申请邮件到到邮箱：developer@huawei.com  2.邮件名称：HUAWEI HiAI+公司名称+产品名称  3.邮件正文：合作公司+联系人+联系方式+联系邮箱  4.我们将在收到邮件的5个工作内邮件给您反馈结果，请您注意查收。  **官网：**  http://developer.huawei.com/consumer/cn/devunion/ui/server/HiAI.html |  |  |

**目录**[**华为HiAI DDK集成手册** 1](#_Toc508113435)

[**1** **DDK简介** 1](#_Toc508113436)

[**2** **DDK包说明** 1](#_Toc508113437)

[**2.1 App source** 1](#_Toc508113438)

[**2.2 DDK** 2](#_Toc508113439)

[**2.3 Documents** 3](#_Toc508113440)

[**2.4 tools** 3](#_Toc508113441)

[**3** **集成概述** 3](#_Toc508113442)

[**4** **开发环境** 4](#_Toc508113443)

[**5** **模型转换** 4](#_Toc508113444)

[**5.1 Caffe模型转换** 4](#_Toc508113445)

[**5.2 Tensorflow模型转换** 5](#_Toc508113446)

[5.2.1 离线模型生成 5](#_Toc508113447)

[5.2.2 模型参数文件 6](#_Toc508113448)

[**6** **模型集成** 7](#_Toc508113449)

[**6.1 生成离线模型** 8](#_Toc508113450)

[6.1.1 Caffe模型转换sample 8](#_Toc508113451)

[**6.2 接口集成** 8](#_Toc508113452)

[6.2.1 创建模型管家 9](#_Toc508113453)

[6.2.2 模型加载 12](#_Toc508113454)

[6.2.3 运行模型 15](#_Toc508113455)

[6.2.4 卸载模型以及销毁模型管家 19](#_Toc508113456)

[**6.3 错误码的定义** 22](#_Toc508113457)

[**7** **Q&A** 22](#_Toc508113458)

**华为HiAI DDK集成手册**

# **DDK简介**

HiAI DDK(Device Development Kit)是海思发布的人工智能计算SDK，该SDK面向人工智能应用程序开发人员和机器学习算法人员，通过使用HiAI DDK，可以更好的提升移动端机器学习模型运行速度。

# **DDK包说明**

一个完整的DDK包含有如下图1的五个部分。

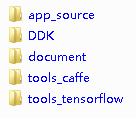


图1 DDK包内容

* app\_source为Android demo app的源码，
* DDK为HiAI开放的sdk，
* Document为开发参考文档，
* tools\_caffe和tools\_tensorflow是用于生成模型到指定格式的工具。

## **2.1 App source**

App source以Squeeze分类网络为例(模型下载地址： https://github.com/DeepScale/SqueezeNet/tree/master/SqueezeNet\_v1.1)，提供了从输入前处理到模型加载、模型前向计算，前向计算结果后处理，模型卸载，时间统计等一系列sample code示例，同时提供了同步，异步方式的sample code。

使用Android Studio(2.2及以上版本)( 请参考Google Android Developers网站：<https://developer.android.com/studio/index.html>)。

导入App source源码，运行，可以看到app支持从图库中选择图片或者拍照去做分类。App运行效果图如下图的图2。



图2: DDK App运行效果

## **2.2 DDK**

DDK包含了两部分:

* ai\_ddk\_demo:使用DDK接口集成的图片分类识别示例程序

ai\_ddk\_demo中classify\_jni.cpp提供的是同步JNI接口示例程序，classify\_async\_jni.cpp提供的是异步JNI接口示例程序。

* ai\_ddk\_lib依赖库和相关头文件。

libai\_client.so是DDK依赖的动态库;

HIAIModelManager.h 是DDK中的头文件,包含了一些动态库中的函数声明

## **2.3 Documents**

包含两个文档：

* DDK集成手册，即本文档，通过一个整体流程的案例介绍如何使用kirin DDK
* 接口说明文档，对DDK中提供的接口做了详细阐述。

## **2.4 tools**

caffe模型和tensorflow模型使用HiAI 加速，需先转换成指定的格式，针对caffe和tensorflow分别提供了转换工具。具体使用方式参见第5章的说明。

# **集成概述**

集成HiAI DDK，需要经过以下的步骤：

* 算子兼容性评估；

平台当前不支持自定义算子类型，算子支持及算子规格支持，请参考“HiAI DDK使用手册”章节2.4。

* 模型格式转换；

完成算子兼容性评估后，需要将caffe或者tensorflow模型转换为HiAI平台支持的模型格式。

* 模型集成；

模型使用流程包含创建模型管家，模型加载、模型计算、卸载模型和销毁模型管家五步。

# **开发环境**

* 模型转换工具运行在linux平台上。

Linux各镜像下载地址：<http://mirrors.ustc.edu.cn/>

* DDK编译推荐NDK r14b及其以上版本。

NDK下载地址：https://developer.android.com/ndk/downloads/index.html

* 应用开发JDK 8 + Android Studio。

Android Studio下载地址：<https://developer.android.com/studio/index.html>

Java JDK 8下载地址：http://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html

# **模型转换**

使用HiAI DDK加速，需要将caffe和tensorflow模型转换为指定的格式。本章节阐述如何转换模型。

## **5.1 Caffe模型转换**

Linux平台下使用cngen\_linux转换工具，该工具位于DDK包中的tools\_caffe文件夹下，使用方法：

./cngen\_linux –prototxt xxx.prototxt –model xxx.caffemodel，如下：

./cngen\_linux --prototxt test.prototxt --model test.caffemodel

选项：

--pipeline 0 表示是软流水还是非软流水,默认是1表示软流水  
--output\_dir dir 生成的离线模型的路径  
--output\_name model\_name 生成的离线模型的文件名

当看到如下的log时，则模型转换成功。

[covert\_model.cpp 79] ForwardPrefilled done!

model compress mode

ipuMaxMemory used: 327936768

Note：tools\_caffe文件夹下提供脚本方便进行转换，详情参见目录中readme。

## **5.2 Tensorflow模型转换**

Tensorflow离线模型转换既可以处理cpu模型，也可以处理ipu模型,

### 5.2.1 离线模型生成

DDK附件包中提供了linux下二进制转换工具pb\_to\_offline(tools\_tensorflow/pb\_to\_offline文件夹下)用于将cpu模型或者ipu模型转换成离线模型，其中pb\_to\_offline(依赖libipu.so)；

转换步骤如下：

export LD\_LIBRARY\_PATH=./so

./pb\_to\_offline --graph=model/graph\_def\_ipu.pb --param\_file=model/model.txt

首先将so文件夹下的libipu.so进行链接配置

pb\_to\_offline工具有两个参数

--graph参数：pb模型文件

--param\_file参数：模型的参数配置文件

成功生成离线模型时，会存在以下两个文件：

offlineModelName.cambricon

offlineModelName.cambricon.inputs\_outputs.aux

其中第一个文件为离线模型，第二个为中间文件。

Note：pb\_to\_offline文件夹下提供脚本方便进行转换，详情参见目录中readme。

### 5.2.2 模型参数文件

#### 模型参数文件包括两部分信息 ：

1. 模型名称：指定生成离线模型文件的名称

2. 模型 session\_run{}的节点信息

（1）输入节点个数及名称（即 Session::Run 的 input\_nodes 节点），以及输入节点的输入Tensor 的 Shape 信息。

（2）输出节点个数及名称（即 Session::Run 的 output\_nodes节点）

#### 参数文件具体格式及各个参数的含义 ：

1. model\_name:XXX.cambricon

指定生成离线模型文件的名称

2. session\_run{}

指定模型的session\_run{}，在大括号“{}”中包含input\_nodes、output\_nodes信息，即 Session：：Run 的起始节点和结束节点信息。

3. input\_nodes(n):

“xxxx”, n, h, w, c

指定 Session::Run 中输入节点信息，数字 n 表示输入节点的个数，

“xxxx”的名字表示输入节点的名字，

n、h、w、c 表示每个输入节点 Tensor 的 shape 的 N、C、H、W四个维度的 size。

4. output\_nodes(n):

“xxxx”

“yyyy”

“zzzz”

指定 Session::Run 中输出节点的名字，数字 n 表示输出节点的个数，

“xxxx”，“yyyy”，“zzzz”分别表示节点的名字。

例: 以下是inceptionV3的配置文件

model\_name: InceptionV3.cambricon

session\_run{

input\_nodes(1): //指定输入节点个数(1个)

"input",1,299,299,3 //指定输入节点的名称和shape

output\_nodes(1): //指定输出节点个数(1个)

"InceptionV3/Predictions/Softmax" //指定输出节点的名称

}

# **模型集成**

模型集成包括离线模型生成、使用加载模型接口加载离线模型，数据预处理，运行模型、取数据做后处理，卸载模型等。当前无论模型是caffe模型还是tensorflow模型，生成离线模型后，集成方式一致。

其中创建模型管家,加载模型、模型前向计算和卸载模型接口提供了同步和异步两种方式。

## **6.1 生成离线模型**

### 6.1.1 Caffe模型转换sample

以SqueezeNet网络为例，SqueezeNet模型的下载地址：

https://github.com/DeepScale/SqueezeNet/tree/master/SqueezeNet\_v1.1

将prototxt的输入层：

layer {

name: "data"

type: "Input"

top: "data"

input\_param { shape: { dim: 10 dim: 3 dim: 227 dim: 227 } }

}

改成：

input: "data"

input\_shape {

dim: 1

dim: 3

dim: 227

dim: 227

}

使用第五章的模型转换方法进行模型转换

## **6.2 接口集成**

DDK提供了同步和异步接口，应用开发者根据需求选择使用同步或者异步接口方式。本章节阐述同步，异步模式下单模型接口的使用，具体代码参考DDKDemo，其中DDK接口具体事宜参见DDK参考文档。

Demo中：

同步模式下：

应用层代码文件：SyncClassifyActivity.java；JNI层代码文件classify\_jni.cpp

异步模式下：

应用层代码文件：AsyncClassifyActivity.java；JNI层代码文件classify\_async\_jni.cpp

### 6.2.1 创建模型管家

#### 6.2.1.1 应用层创建模型管家：

* 同步模式:

通过调用JNI层loadModelSync函数，在加载模型前完成同步模型管家创建；

private class loadModelTask extends AsyncTask<Void, Void, Integer> {

@Override

protected Integer doInBackground(Void... voids) {

int ret = ModelManager.loadModelSync("hiai", mgr);

return ret;

}

* 异步模式:

通过调用JNI层registerListenerJNI函数，实现异步模式下模型管家创建。

protected void onCreate(Bundle savedInstanceState) {

super.onCreate(savedInstanceState);

getSupportActionBar().hide();

setContentView(R.layout.activity\_async\_classify);

mgr = getResources().getAssets();

int ret = ModelManager.registerListenerJNI(listener);

……

}

#### 6.2.1.2 JNI层创建模型管家

JNI层创建模型管家，通过在JNI函数中调用DDK接口HIAI\_ModelManager\_create来实现。

* 同步模式:

extern "C" JNIEXPORT jint JNICALL

Java\_com\_huawei\_hiaidemo\_ModelManager\_loadModelSync(JNIEnv \*env, jobject instance, jstring jmodelName, jobject assetManager){

……

modelManager = HIA\_ModelManager\_create(NULL);

……

* 异步模式:

extern "C"

JNIEXPORT jint JNICALL

Java\_com\_huawei\_hiaidemo\_ModelManager\_registerListenerJNI(JNIEnv \*env, jobject obj,

jobject callbacks) {

callbacksInstance = env->NewGlobalRef(callbacks);

jclass objClass = env->GetObjectClass(callbacks);

if (objClass) {

callbacksClass = reinterpret\_cast<jclass>(env->NewGlobalRef(objClass));

env->DeleteLocalRef(objClass);

}

listener.onLoadDone = onLoadDone;

listener.onRunDone = onRunDone;

listener.onUnloadDone = onUnloadDone;

listener.onTimeout = onTimeout;

listener.onError = onError;

listener.onServiceDied = onServiceDied;

modelManager = HIAI\_ModelManager\_create(&listener);

return 0;

}

其中，

NewGlobalRef 获取应用层传入的ModelManagerListener对象实例引用；

GetObjectClass 获取传入的ModelManagerListener对象类型；

DeleteLocalRef 用于应用层引用对象的释放。

异步模式中HIAI\_ModelManagerListener 对象中回调函数的注册：

以onLoadDone为例

void onLoadDone(void \*userdata, int taskId) {

LOGE("AYSNC JNI layer onLoadDone:", taskId);

JNIEnv \*env;

jvm->AttachCurrentThread(&env, NULL);

if (callbacksInstance != NULL) {

jmethodID onValueReceived = env->GetMethodID(callbacksClass, "onStartDone", "(I)V");

env->CallVoidMethod(callbacksInstance, onValueReceived, taskId);

}

}

其中

JNIEnv \*env;

jvm->AttachCurrentThread(&env, NULL);

用于在本地代码中获取当前线程中的JNIEnv指针，

GetMethodID 用于获取应用层传入的ModelManagerListener类中的函数接口

CallVoidMethod执行获取到的onStartDone函数调用

#### 6.2.1.3 DDK创建模型管家

DDK中创建模型管家函数原型：

HIAI\_ModelManager\* HIAI\_ModelManager\_create(HIAI\_ModelManagerListener\* listener);

创建模型管家时,通过参数来实现同步或者异步模型管家创建,参数为NULL 时，创建同步模型管家；传入HIAI\_ModelManagerListener 非空实例指针，创建异步模型管家。

### 6.2.2 模型加载

在使用模型前，需先加载模型， DDK支持单模型和多模型加载。同时支持模型从app源码目录assets中加载和从sdcard中加载。

#### 6.2.2.1 应用层加载模型：

* 同步模式:

通过调用JNI层loadModelSync函数，完成同步模式下模型加载；

private class loadModelTask extends AsyncTask<Void, Void, Integer> {

@Override

protected Integer doInBackground(Void... voids) {

int ret = ModelManager.loadModelSync("hiai", mgr);

return ret;

}

* 异步模式:

通过调用JNI层loadModelAsync函数，实现异步模式下模型加载。

protected void onCreate(Bundle savedInstanceState) {

super.onCreate(savedInstanceState);

getSupportActionBar().hide();

setContentView(R.layout.activity\_async\_classify);

mgr = getResources().getAssets();

int ret = ModelManager.registerListenerJNI(listener);

Log.e(TAG, "onCreate: " + ret);

ModelManager.loadModelAsync("hiai", mgr);

items = new ArrayList<>();

mgr = getResources().getAssets();

initView();

}

#### 6.2.2.2 JNI层加载模型

* 同步模式:

extern "C"

JNIEXPORT jint JNICALL

Java\_com\_huawei\_hiaidemo\_ModelManager\_loadModelSync(JNIEnv \*env, jobject instance, jstring jmodelName, jobject assetManager){

……

AAssetManager \*mgr = AAssetManager\_fromJava(env, assetManager);

LOGI("Attempting to load model...\n");

LOGE("model name is %s", modelname);

AAsset \*asset = AAssetManager\_open(mgr, modelname, AASSET\_MODE\_BUFFER);

if (nullptr == asset) {

LOGE("AAsset is null...\n");

}

const void \*data = AAsset\_getBuffer(asset);

if (nullptr == data) {

LOGE("model buffer is null...\n");

}

off\_t len = AAsset\_getLength(asset);

if (0 == len) {

LOGE("model buffer length is 0...\n");

}

HIAI\_ModelBuffer \*modelBuffer = HIAI\_ModelBuffer\_create\_from\_buffer(modelName,

(void \*) data, len, HIAI\_DevPerf::HIAI\_DEVPREF\_HIGH);

HIAI\_ModelBuffer \*modelBufferArray[] = {modelBuffer};

int ret = HIAI\_ModelManager\_loadFromModelBuffers(modelManager, modelBufferArray, 1);

LOGI("load model from assets ret = %d", ret);

env->ReleaseStringUTFChars(jmodelName, modelName);

AAsset\_close(asset);

return ret;

}

其中，在JNI中，通过以下代码获取到AssetManager：

AAssetManager \*mgr = AAssetManager\_fromJava(env, assetManager);

关于AssetManager的API，参考Android Developer网站:

<https://developer.android.com/ndk/reference/asset__manager_8h.html>

AAsset\* asset = AAssetManager\_open(mgr, "hiai.cambricon", AASSET\_MODE\_BUFFER);

AAssetManager\_open读取app源码下assets目录下hiai.cambricon文件，返回AAsset。

通过下面函数获取buffer地址以及buffer大小。

void \*data = (void \*)AAsset\_getBuffer(asset);

off\_t len = AAsset\_getLength(asset);

调用DDK接口函数HIAI\_ModelBuffer\_create\_from\_buffer 来创建HIAI\_ModelBuffer对象，并调用HIAI\_ModelManager\_loadFromModelBuffers 完成模型加载

* 异步模式:

extern "C"

JNIEXPORT void JNICALL

Java\_com\_huawei\_hiaidemo\_ModelManager\_loadModelAsync(JNIEnv \*env, jobject instance,

jstring jmodelName, jobject assetManager) {

……

HIAI\_ModelBuffer \*modelBuffer = HIAI\_ModelBuffer\_create\_from\_buffer(modelName,

(void \*) data, len, HIAI\_DevPerf::HIAI\_DEVPREF\_HIGH);

HIAI\_ModelBuffer \*modelBufferArray[] = {modelBuffer};

int ret = HIAI\_ModelManager\_loadFromModelBuffers(modelManager, modelBufferArray, 1);

LOGE("ASYNC JNI LAYER load model from assets ret = %d", ret);

env->ReleaseStringUTFChars(jmodelName, modelName);

AAsset\_close(asset);

}

加载过程和同步相同，只是在调用HIAI\_ModelManager\_loadFromModelBuffers接口函数时，注意传入的是异步模型管理引擎。

#### 6.2.2.3 DDK加载模型

DDK加载模型接口函数原型如下：

int HIAI\_ModelManager\_loadFromModelBuffers(HIAI\_ModelManager\* manager, HIAI\_ModelBuffer\* bufferArray[], int nBuffers);

其中：

manager：模型管理引擎对象接口（同步，异步）。

bufferArray[]：HIAI\_ModelBuffer，单模型和多模型均可。

nBuffers：加载模型的个数。

### 6.2.3 运行模型

#### 6.2.3.1 应用层运行模型：

* 同步模式:

通过调用JNI层runModelSync函数，完成同步模式下模型运行；

private class RunModelTask extends AsyncTask<Bitmap, Void, String[]> {

@Override

protected String[] doInBackground(Bitmap... bitmaps) {

float[] buffer = getPixel(bitmaps[0], RESIZED\_WIDTH, RESIZED\_HEIGHT);

initClassifiedImg = bitmaps[0];

predictedClass = ModelManager.runModelSync("hiai", buffer);

return predictedClass;

}

……

}

首先通过函数getPixel ，从图片中获取模型输入，然后通过调用JNI层runModelSync函数，完成同步模式下模型运行。

* 异步模式:

通过调用JNI层runModelAsync函数，完成异步模式下模型运行；

protected void onActivityResult(int requestCode, int resultCode, Intent data) {

super.onActivityResult(requestCode, resultCode, data);

if (resultCode == RESULT\_OK && data != null) switch (requestCode) {

case GALLERY\_REQUEST\_CODE:

try {

Bitmap bitmap;

ContentResolver resolver = getContentResolver();

Uri originalUri = data.getData();

bitmap = MediaStore.Images.Media.getBitmap(resolver, originalUri);

String[] proj = {MediaStore.Images.Media.DATA};

Cursor cursor = managedQuery(originalUri, proj, null, null, null);

cursor.moveToFirst();

Bitmap rgba = bitmap.copy(Bitmap.Config.ARGB\_8888, true);

final Bitmap initClassifiedImg = Bitmap.createScaledBitmap(rgba, RESIZED\_WIDTH, RESIZED\_HEIGHT, false);

final float[] pixels = getPixel(initClassifiedImg, RESIZED\_WIDTH, RESIZED\_HEIGHT);

show = initClassifiedImg;

ModelManager.runModelAsync("hiai", pixels);

} catch (IOException e) {

Log.e(TAG, e.toString());

}

break;

……

}

#### 6.2.3.2 JNI层运行模型

* 同步模式:

extern "C"

JNIEXPORT jobjectArray JNICALL

Java\_com\_huawei\_hiaidemo\_ModelManager\_runModelSync(JNIEnv \*env, jclass type, jstring jmodelName, jfloatArray jbuf) {

const char \*modelName = env->GetStringUTFChars(jmodelName, 0);

if (NULL == modelManager) {

LOGE("please load model first");

return NULL;

}

float \*dataBuff = NULL;

if (NULL != jbuf) {

dataBuff = env->GetFloatArrayElements(jbuf, NULL);

}

inputtensor = HIAI\_TensorBuffer\_create(1, 3, 227, 227);

HIAI\_TensorBuffer \*inputtensorbuffer[] = {inputtensor};

outputtensor = HIAI\_TensorBuffer\_create(1, 1000, 1, 1);

HIAI\_TensorBuffer \*outputtensorbuffer[] = {outputtensor};

float \*inputbuffer = (float \*) HIAI\_TensorBuffer\_getRawBuffer(inputtensor);

int length = HIAI\_TensorBuffer\_getBufferSize(inputtensor);

LOGE("SYNC JNI runModel modelname:%s", modelName);

memcpy(inputbuffer, dataBuff, length);

float time\_use;

struct timeval tpstart, tpend;

gettimeofday(&tpstart, NULL);

int ret = HIAI\_ModelManager\_runModel(

modelManager,

inputtensorbuffer,

1,

outputtensorbuffer,

1,

1000,

modelName);

LOGE("run model ret: %d", ret);

gettimeofday(&tpend, NULL);

……

首先通过函数env->GetFloatArrayElements(jbuf, NULL)获取应用层输入数据，

调用函数HIAI\_TensorBuffer\_create(1, 3, 227, 227)创建输入inputtensor，

接下来通过DDK接口函数

HIAI\_TensorBuffer\_getRawBuffer(inputtensor)获取输入inputtensor地址

HIAI\_TensorBuffer\_getBufferSize(inputtensor)获取输入inputtensor大小，

最后将应用层传入的dataBuff通过函数memcpy拷贝到inputtensor

拷贝完成后，再准备outputtensor，

输入输出准备好后，即可调用HIAI\_ModelManager\_runModel接口来运行模型，

模型运行完成后对运行后的数据进行些后处理。

* 异步模式:

extern "C"

JNIEXPORT void JNICALL

Java\_com\_huawei\_hiaidemo\_ModelManager\_runModelAsync(JNIEnv \*env, jobject instance,

jstring jmodelName, jfloatArray jbuf) {

……

int ret = HIAI\_ModelManager\_runModel(

modelManager,

inputtensorbuffer,

1,

outputtensorbuffer,

1,

1000,

modelName);

LOGE("ASYNC JNI layer runmodel ret: %d", ret);

……

}

异步模式下模型运行过程和同步相同，只是在调用HIAI\_ModelManager\_runModel接口函数时，注意传入的是异步模型管理引擎。

异步模式下运行模型后的后处理操作是通过回调函数onRunDone实现的。

#### 6.2.3.3 DDK运行模型

DDK运行模型接口函数原型如下：

int HIAI\_ModelManager\_runModel(

HIAI\_ModelManager\* manager,

HIAI\_TensorBuffer\* input[],

int nInput,

HIAI\_TensorBuffer\* output[],

int nOutput,

int ulTimeout,

const char\* modelName);

其中

manager：模型管理引擎对象接口。

input[]：模型输入，支持多输入。

nInput：模型输入的个数。

output[]：模型输出，支持多输出。

nOutput：模型输出的个数。

ulTimeout：超时时间。

modelName：模型名称。

### 6.2.4 卸载模型以及销毁模型管家

#### 6.2.4.1 应用层卸载模型和销毁模型管家：

* 同步模式:

通过调用JNI层unloadModelSync函数，完成同步模式下模型卸载；

protected void onDestroy() {

super.onDestroy();

int result = ModelManager.unloadModelSync();

if (AI\_OK == result) {

Toast.makeText(this, "unload model success.", Toast.LENGTH\_SHORT).show();

} else {

Toast.makeText(this, "unload model fail.", Toast.LENGTH\_SHORT).show();

}

}

* 异步模式:

通过调用JNI层unloadModelAsync函数，完成异步模式下模型卸载；

protected void onDestroy() {

super.onDestroy();

ModelManager.unloadModelAsync();

}

而异步模式下的模型管家卸载，是通过回调函数来完成的。

#### 6.2.4.2 JNI层卸载模型和销毁模型管家

* 同步模式:

extern "C"

JNIEXPORT jint JNICALL

Java\_com\_huawei\_hiaidemo\_ModelManager\_unloadModelSync(JNIEnv \*env, jobject instance) {

if (NULL == modelManager) {

LOGE("please load model first.");

return -1;

} else {

if (modelBuffer != NULL) {

HIAI\_ModelBuffer\_destroy(modelBuffer);

modelBuffer = NULL;

}

int ret = HIAI\_ModelManager\_unloadModel(modelManager);

LOGE("JNI unload model ret:%d", ret);

HIAI\_ModelManager\_destroy(modelManager);

modelManager = NULL;

return ret;

}

}

* 异步模式:

void onUnloadDone(void \*userdata, int taskStamp) {

LOGE("JNI layer onUnloadDone:", taskStamp);

JNIEnv \*env;

jvm->AttachCurrentThread(&env, NULL);

if (callbacksInstance != NULL) {

jmethodID onValueReceived = env->GetMethodID(callbacksClass, "onStopDone", "(I)V");

env->CallVoidMethod(callbacksInstance, onValueReceived, taskStamp);

}

HIAI\_ModelManager\_destroy(modelManager);

modelManager = NULL;

listener.onRunDone = NULL;

listener.onUnloadDone = NULL;

listener.onTimeout = NULL;

listener.onServiceDied = NULL;

listener.onError = NULL;

listener.onLoadDone = NULL;

}

extern "C"

JNIEXPORT void JNICALL

Java\_com\_huawei\_hiaidemo\_ModelManager\_unloadModelAsync(JNIEnv \*env, jobject instance) {

if (NULL == modelManager) {

LOGE("please load model first");

return;

} else {

if (modelBuffer != NULL) {

HIAI\_ModelBuffer\_destroy(modelBuffer);

modelBuffer = NULL;

}

int ret = HIAI\_ModelManager\_unloadModel(modelManager);

LOGE("ASYNC JNI layer unLoadModel ret:%d", ret);

}

}

其中，模型卸载通过函数unloadModelAsync完成卸载，而模型管家通过回调函数onUnloadDone完成销毁。

#### 6.2.4.3 DDK卸载模型和销毁模型管家：

DDK卸载模型接口函数原型如下：

int HIAI\_ModelManager\_unloadModel(HIAI\_ModelManager\* manager);

DK卸载模型销毁模型管家接口函数原型如下：

void HIAI\_ModelManager\_destroy(HIAI\_ModelManager\* manager);

manager：模型管理引擎对象接口。

## **6.3 错误码的定义**

模型管家的加载模型、运行模型、卸载模型当以上接口调用失败的时候，会返回一定的错误码。该错误码定义在DDK使用手册中有定义。

# **Q&A**

1. 如果模型有不支持的算子，怎么办？

比如alexnet的最后一层softmax NPU不支持，开发者需要在CPU上做处理。

1. 怎么测试前向计算时间？

在jni层runModel接口前后加时间打印。

1. 你们的DDK使用c++\_shared编译运行时，但是我的程序中使用了gnustl\_static编译运行时，怎么办？

基于我们提供的C++接口，根据自己的需求，做一层C接口的封装即可使用gnustl\_static编译运行时。

1. 我的app使用cmake进行编译，DDK支持cmake编译吗？

支持cmake编译，基于我们提供的C++接口，根据自己的需求，做一层C接口的封装即可。

1. 我已经完成了集成，但是运行效果(结果)不对，是什么原因？

重点从模型兼容性出发排查，参考“华为HiAI DDK使用手册”文件的“支持算子”章节，排查是不是模型中有不支持的算子规格。

1. 加载模型支持从sdcard或assets中加载吗？

模型加载支持从sdcard和assets中加载，具体的请参考demo程序。

1. 如何选择使用“同步”接口还是“异步”接口？

开发者根据自身需求选择。

1. 生成离线模型，什么是模型分段，如何判断模型分段？DDK对模型分段的支持如何？

在“华为HiAI DDK使用手册”文件的“支持算子”章节，提到了deconvolution layer的stride\*num\_output不能大于256，如果模型中有中间层是deconvolution layer，且这个deconv layer的stride\*num\_output大于了256，那么在使用cngen\_linux离线模型生成工具的时候，离线模型会分段，具体log信息如下：

----------------net.cpp | buff 1---------------

[net.cpp 1391] ipuOpenDeviceStream() stream 0

[net.cpp 1405] ipuCloseDeviceStream() stream 0

[NGPF:release] DDR Access Times: 76

[net.cpp 1539] ipuStreamExecuteOnce()

[net.cpp 1391] ipuOpenDeviceStream() stream 1

[net.cpp 1405] ipuCloseDeviceStream() stream 1

[NGPF:release] DDR Access Times: 7112

[net.cpp 1539] ipuStreamExecuteOnce()

ipuSwitchPipelineBuffer, org Index: 1

ipuSwitchPipelineBuffer, new Index: 0

[covert\_model.cpp 129] ForwardPrefilled done!

model compress mode

ipuMaxMemory used: 386669312

可以根据ipuOpenDeviceStream的个数确定离线模型的段数，这里是分成了两段(最终文件是一个文件)。DDK现在只支持运行第一段离线模型。