**人脸检测实验**



华为技术有限公司

目录

[人脸检测实验 2](#_Toc25223513)

[1实验介绍 2](#_Toc25223514)

[2实验目的 2](#_Toc25223515)

[3预备知识 2](#_Toc25223516)

[4实验环境 2](#_Toc25223517)

[5实验原理及流程 4](#_Toc25223518)

[5.1实验原理 4](#_Toc25223519)

[5.2实验流程 5](#_Toc25223520)

[6实验任务及步骤 6](#_Toc25223521)

[任务一 模板工程创建及引擎配置文件编写 6](#_Toc25223522)

[任务二 应用代码编写 13](#_Toc25223523)

[任务三 编译文件编写 26](#_Toc25223524)

[任务四 模型转换 28](#_Toc25223525)

[任务五 工程编译及运行 32](#_Toc25223526)

# 人脸检测实验

## 1实验介绍

本实验主要介绍人脸检测项目代码开发并部署在Atlas 200 DK开发板上执行的方法。通过Atlas 200 DK开发板来实现人脸检测推理实验，使用Atlas 200 DK外接的摄像头获取的视频数据作为输入，实时检测视频画面中的人脸，并将检测后的结果展示出来。用户可以通过人脸检测项目对Atlas 200 DK开发板在AI方面的应用有全面的认识。

## 2实验目的

了解熟悉人脸检测应用代码的编写(C++语言)。

掌握将应用部署在Atlas 200 DK开发板上的操作。

## 3预备知识

具备一定的深度学习理论知识，对业界主流的深度学习框架（Caffe、TensorFlow等）有一定了解。

具备C++开发能力，具备一定的Shell脚本、Python脚本开发能力。

了解Cmake的语法知识和google-protobuf的数据格式

了解Linux操作系统的基本使用。

## 4实验环境

实验环境需要从硬件和软件两个方面进行准备：

（1） 硬件配件准备环境：

使用Atlas 200 DK前，需自行购买相关配件，包含制作Atlas 200 DK启动系统的micro SD卡、读卡器，与Ubuntu虚拟机相连接的Type-C数据线及摄像头等配件，详细的配件信息如表4.1所示：

表4-1硬件配件清单

|  |  |  |
| --- | --- | --- |
| 配件名称 | 描述 | 推荐型号 |
| SD卡 | 用于制作Atlas 200 DK开发者板启动系统。 | 推荐使用经过测试的SD卡：  三星UHS-I U3 CLASS 10 64G  金士顿UHS-I U1 CLASS 10 64G |
| 读卡器 | 使用读卡器制作SD卡的场景。 | 支持USB3.0协议 |
| Type-C连接线 | 用于将开发板与Mind Studio所在服务器通过USB方式连接。 | 支持USB3.0的Type-C连接线 |
| 摄像头 | 用于与Atlas 200 DK连接获取视频。 | 兼容树莓派摄像头，如果使用树莓派摄像头，需要额外购买黄色排线 |
| 摄像头支架  （可选） | 用于固定摄像头。 | 树莓派透明摄像头支架 |

（2）软件部署环境：

已在Ubuntu机器上搭建好Mind Studio1.3版本的环境（参考文档：https://ascend.huawei.com/doc/mindstudio/2.1.0(beta)/zh/zh-cn\_topic\_0188160926.html）。

已配置部署好Atlas 200DK环境（参考文档：<https://ascend.huawei.com/doc/Atlas%20200%20DK/1.31.0.0(beta)/zh/zh-cn_topic_0199239495.html>）。

## 5运行人脸检测工程

## 6实验原理及流程

### 6.1实验原理

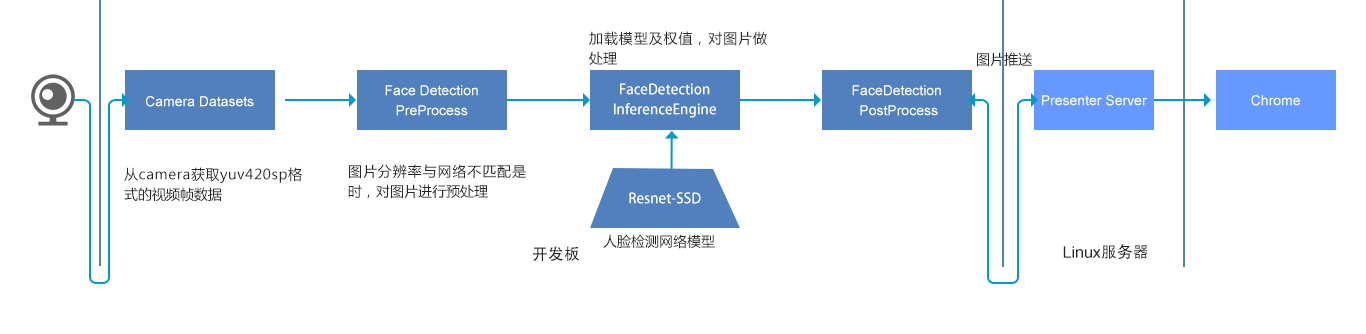


图5.1 人脸检测实验原理图

由原理图可知，本实验的数据采集、图片数据预处理、推理计算及检测结果后处理环节均在Atlas 200 DK上完成，在Linux主机PC端调用PresenterServer服务器，将人脸检测的结果显示在PC的网页上，其中，在主机PC端实现检测结果的展示可以有多种处理方式，读者可根据实际情况，自定义人脸检测结果的显示方式，例如直接将检测数据结果打印在终端里显示，或者将检测数据结果保存在文件中。

在本实验中，主要聚焦在Atlas 200 DK开发板上的应用案例移植环节，因此读者需要重点关注摄像头采集图片数据处理、图片数据预处理及推理图片、检测结果后处理环节的操作。

完整的实验流程涉及到的模块介绍如下：

1. CameraDatasets模块与Camera驱动进行交互，从摄像头获取YUV420SP格式的视频数据。Atlas 200 DK提供了一套帮助开发者轻松获取摄像头图像的API接口媒体库，详细的接口使用方法可参考Media API ( https://ascend.huawei.com/doc/Atlas%20200%20DK/1.31.0.0(beta)/zh/zh-cn\_topic\_0190813674.html ) 。
2. FaceDetectionPreProcess是图像预处理模块，当输入图片的分辨率与网络模型要求的分辨率不匹配时，对图片进行resize预处理。AscendDK提供了EZDVPP接口，对DVPP接口进行了封装，帮助开发者更方便地对图片进行预处理，详细的接口使用方法可参考EZDVPP (https://ascend.huawei.com/doc/Atlas200DK/1.31.0.0(beta)/zh/zh-cn\_topic\_0195891320.html)。
3. FaceDetectionlnferenceEngine会加载已训练好的人脸检测网络模型及其权值，对图片做推理，并将图片转化为JPEG格式，推理接口详细使用方法可参考Matrix API (https://ascend.huawei.com/doc/Atlas%20200%20DK/1.31.0.0(beta)/zh/zh-cn\_topic\_0195891333.html)。
4. FaceDetectionPostProcess将接收到的JPEG图片及推理结果通过调用Presenter Agent的API发送到主机端UI host上部署的Presenter Server服务进程，Presenter Agent API的详细使用方法可参考Presenter Agent (https://ascend.huawei.com/doc/atlas200dk/1.3.0.0/zh/zh-cn\_topic\_0165841897.html）。
5. Presenter Server根据接收到的推理结果，在JPEG图片上进行人脸位置及置信度的标记，并将图像信息推送给主机端Web Ul，用户可通过Chrome浏览器访问Presenter Server,实时查看视频中的人脸检测信息。

### 6.2实验流程

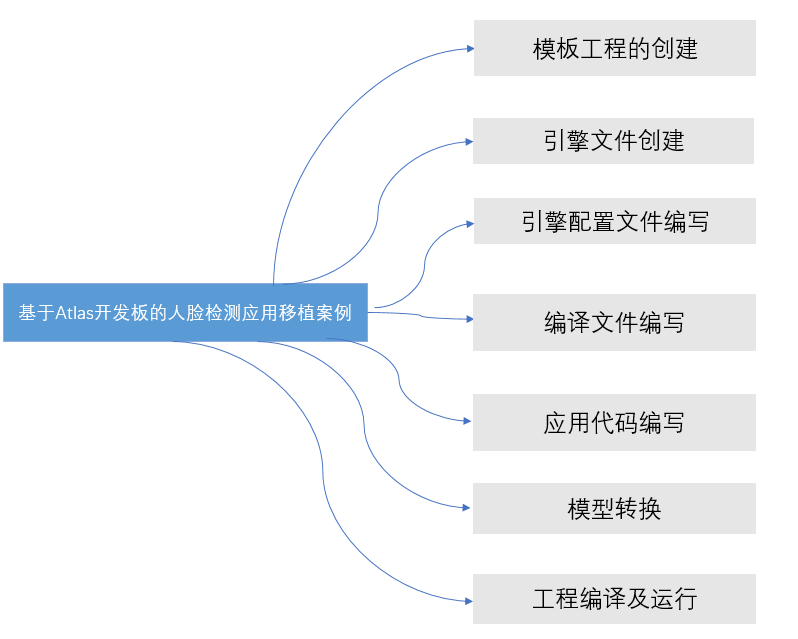


图 5.2 人脸检测应用案例移植流程图

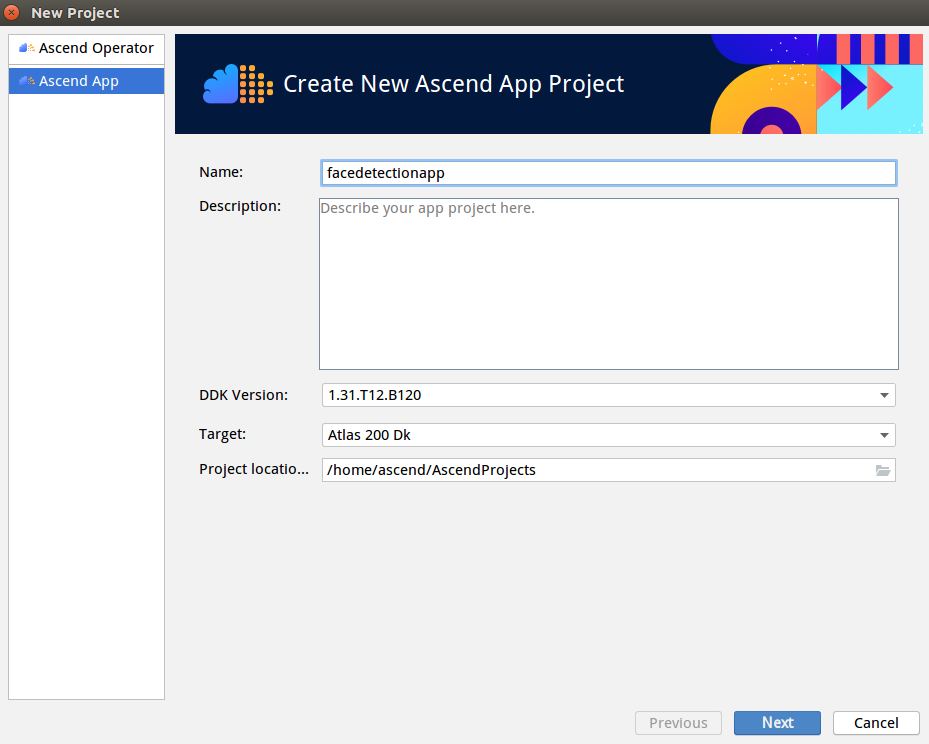
在本实验中，默认已完成硬件环境和软件环境的准备工作，在此基础上进行人脸检测应用项目的实验操作，由上图可知，本实验需要分别在Ubuntu主机PC端完成基于C++的人脸检测应用代码的编写和编译工作，以及人脸检测模型转换，最后在Atlas 200 DK开发板上进行项目部署执行工作。

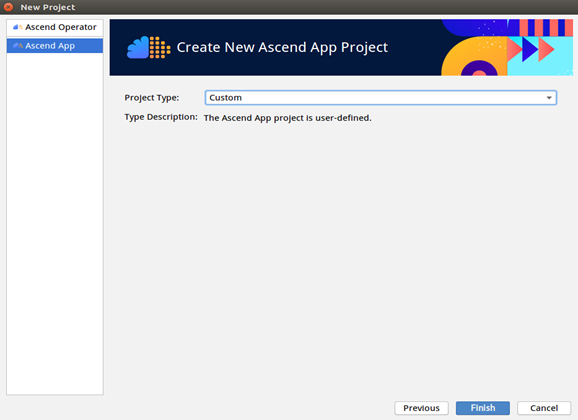
本案例移植的源代码编写及编译以码云源码为例进行说明，实验任务及步骤将围绕图5.2所示五个方面分别展开介绍。

## 7实验任务及步骤

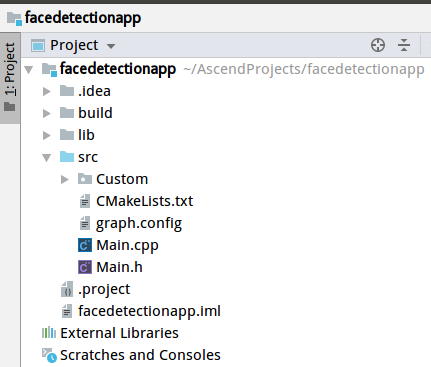
### 任务一 模板工程创建

打开MindStudio，创建Atlas200场景的自定义的工程（File->New->New Project），建议工程名选用“facedetectionapp”，工程名没有特殊要求，如下图所示:





生成自定义工程的主程序main.cpp、graph.config流程配置文件，以及编译需要的CMakeList.txt文件，可以看到如下图所示：



自定义工程源码目录

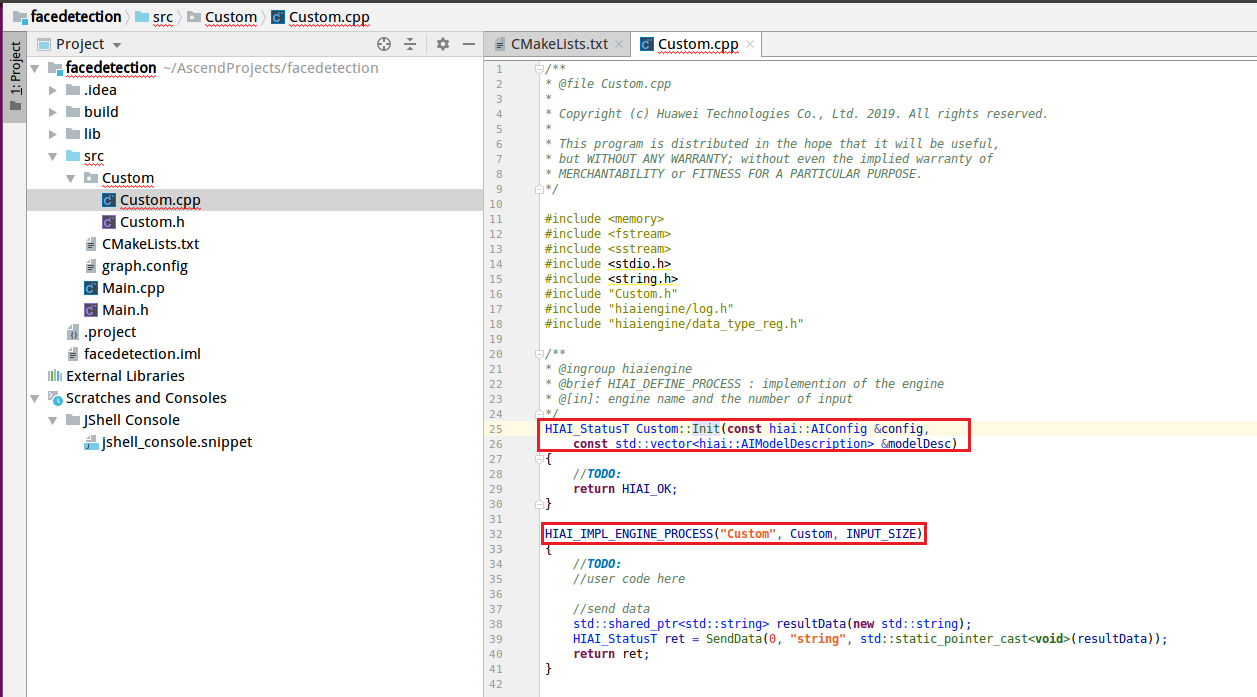
相应的工程文件说明：

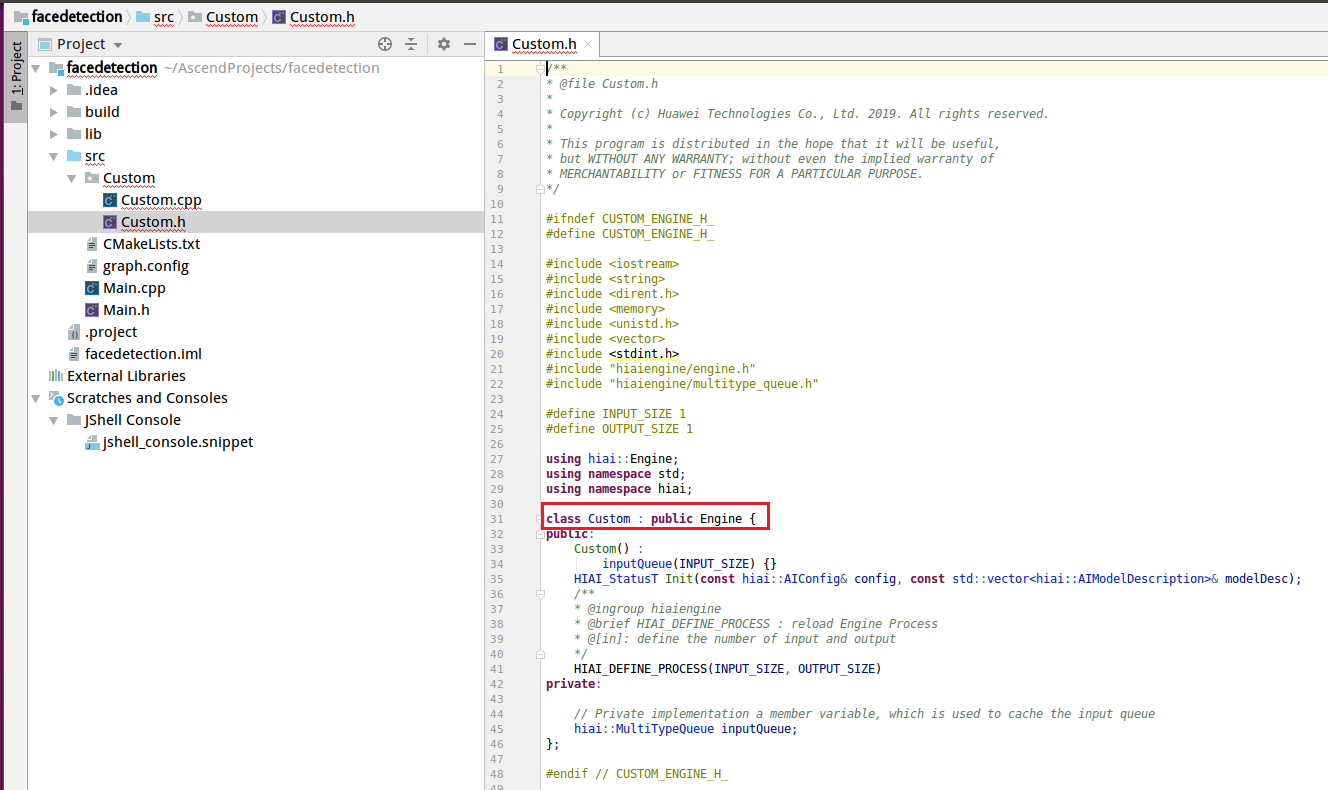
生成的自定义工程可以编译执行，在src/目录下生成一个custom引擎如下图所示：

Custom.c custom.h就是对应的Engine文件。创建会自动生成2个函数，init()和HIAI\_IMPL\_ENGINE\_PROCESS()函数，系统运行会自动依次调用这2个函数，用户无需关注engine如何启动。

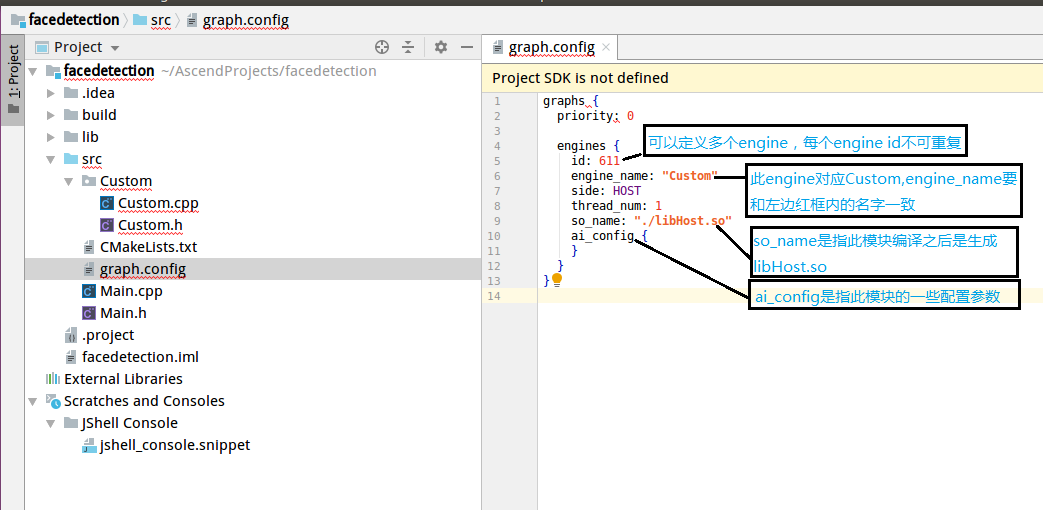
计算引擎（Engine）说明：计算引擎（Engine）为基于Martix框架的业务软件的基本功能单元，允许用户自定义Engine的实现（输入图片数据、对图片进行分类处理、输出对图片数据的分类预测结果等）。

每一个Engine定义了初始化函数Init()和Process()， 在创建Graph时，会自动运行Init()，从而实现Engine的参数初始化（包含内存分配和模型加载），Process()实现数据的传输和业务逻辑。Init和HIAI\_IMPL\_ENGINE\_PROCESS这两个函数名和对应的参数保持不变，每个功能模块的初始化在Init中实现，HIAI\_IMPL\_ENGINE\_PROCESS这个函数实现模块的具体功能，包括和其他模块的数据通信，例如可以通过SendData函数将数据传送到下一个engine，也可以通过arg0参数接收其他engine传递过来的数据。目前只有一个engine，所以如何传递数据放在下面会具体说明。





Graph.config为引擎(engine)的配置文件，包含引擎的id,引擎的名字以及引擎运行于host侧或是device侧（host侧指开发板上的Linux系统，device侧指开发板的推理模块），如下图所示。

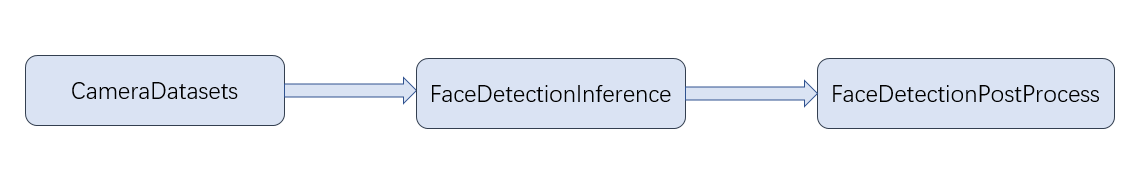


这个创建的custom工程就已经是一个完整的工程，可以编译运行。

### 任务二 引擎文件创建

我们以完成人脸检测应用为例，实现此功能需要设计三个模块分为：Camera模块(摄像头采集图片数据)，推理模块(图片数据预处理及模型推理)，检测结果后处理模块(接收到的JPEG图片及推理结果发送给Presenter Server进程，Presenter Server将接收到的推理结果推送到Web Ul上去展示)。

模块处理逻辑是camera采集摄像头数据，将采集到的图像信息通过engine间传输数据的方式传给推理模块，推理模块接收到camera传送过来的数据，做推理，将推理的结果通过engine间传输数据方式传给后处理模块，后处理模块接收到推理模块传送过来的信息推送到present server，通过web ui展示。

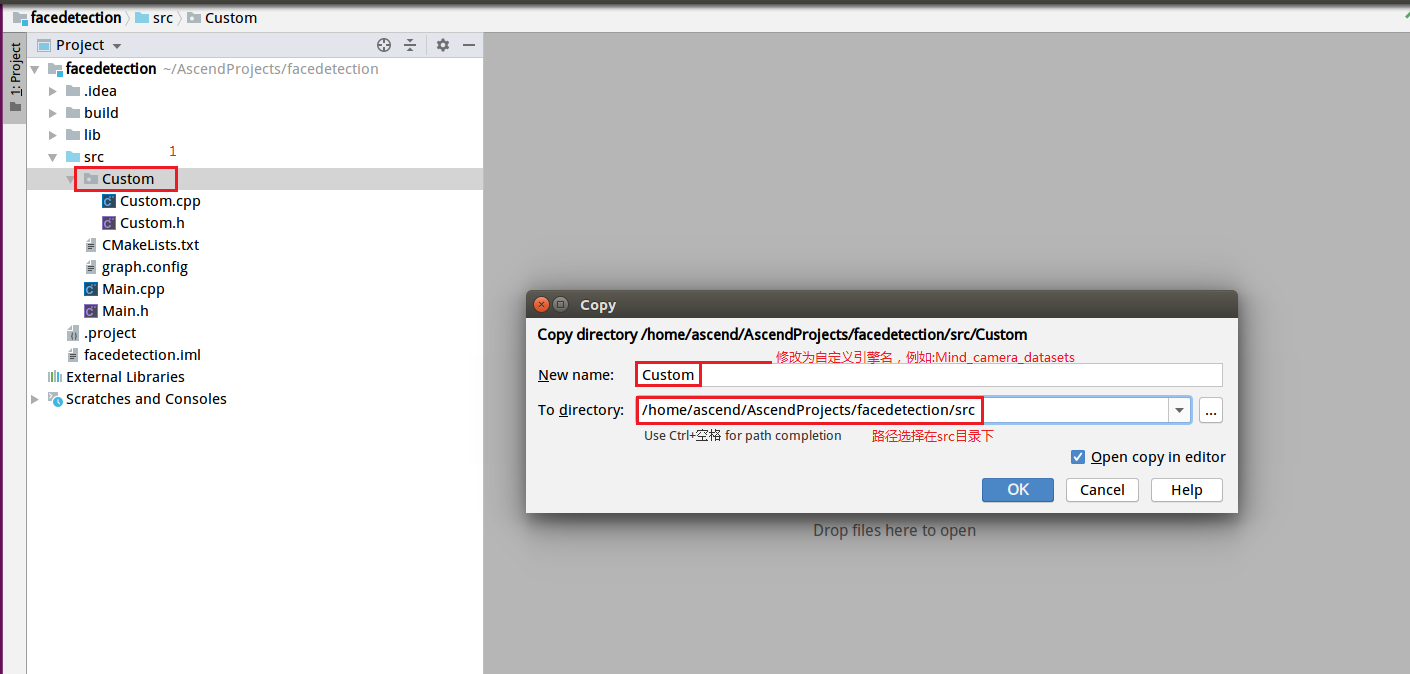


.三个功能模块对应三个引擎，我们以custom引擎为例生成上述三个引擎文件。

1. 复制三个Custom文件夹在src目录下，并将复制的三个Custom文件夹依次更名为Mind\_camera\_datasets（自定义的引擎名）, face\_detection\_inference，face\_detection\_post\_process。

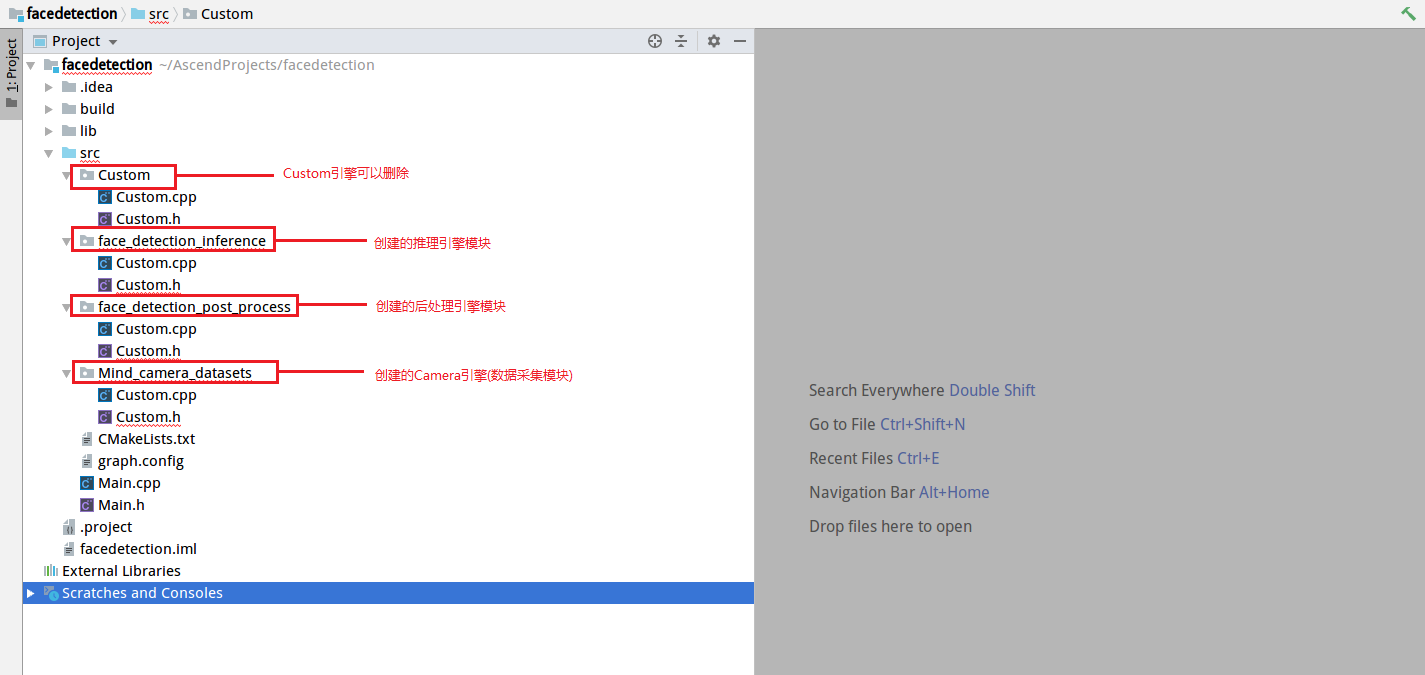
例如生成Mind\_camera\_datasets引擎，鼠标选中Custom文件夹后，右键选择Copy，Paste(或者使用快捷键Ctrl+c,Ctrl+v),生成引擎模板文件，之后参考如下图所示，修改文件夹名为Mind\_camera\_datasets，即我们自定义的引擎名。

生成自定义引擎示意图

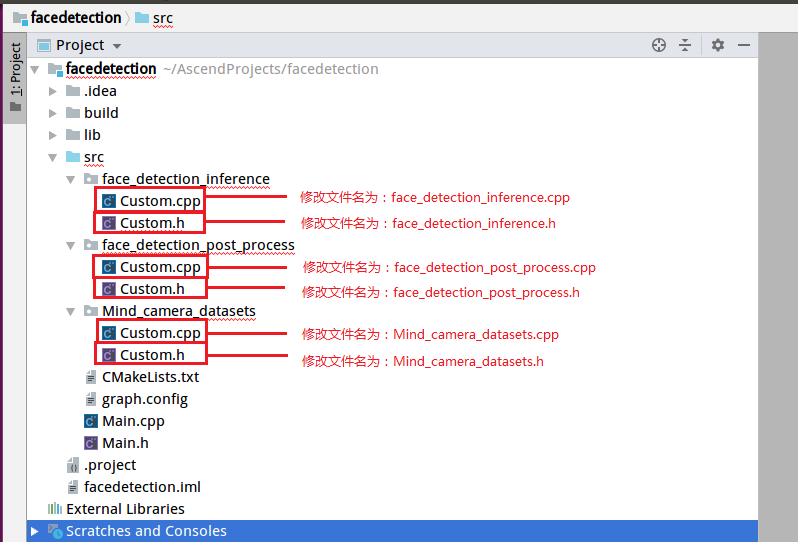


同理，可以照此方法创建另外两个engine模板文件。

创建模块文件成功之后，如下图所示：

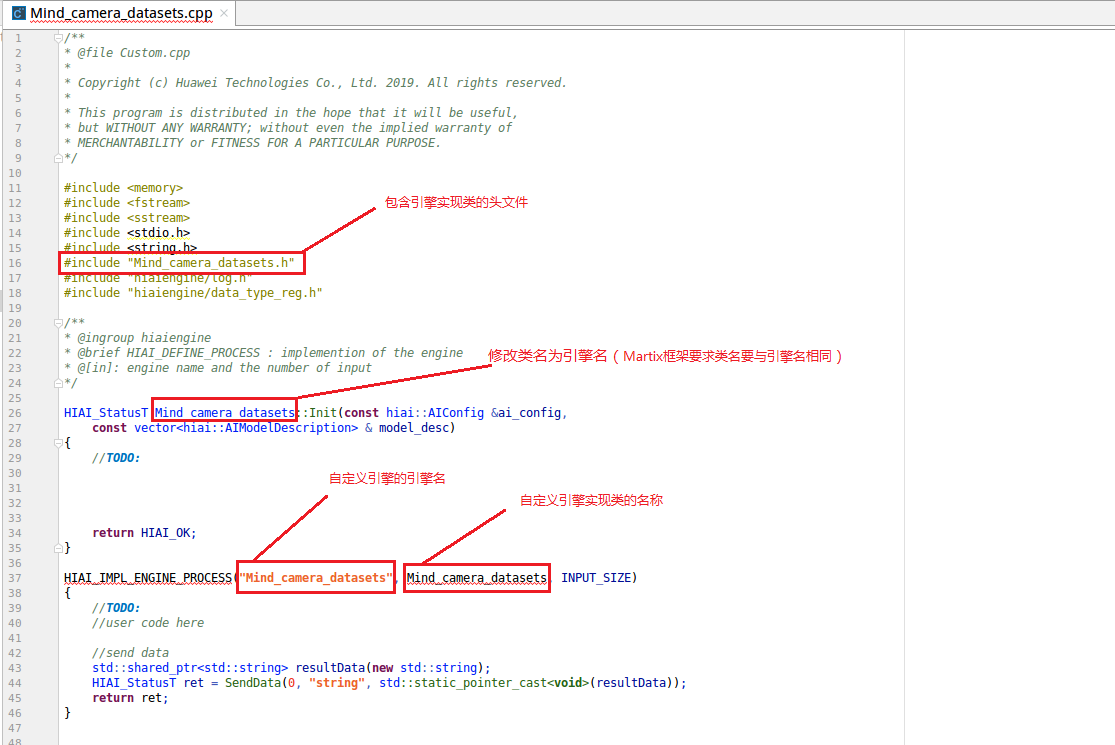


1. 修改每个引擎目录下的源文件和头文件的文件名，如下图所示：

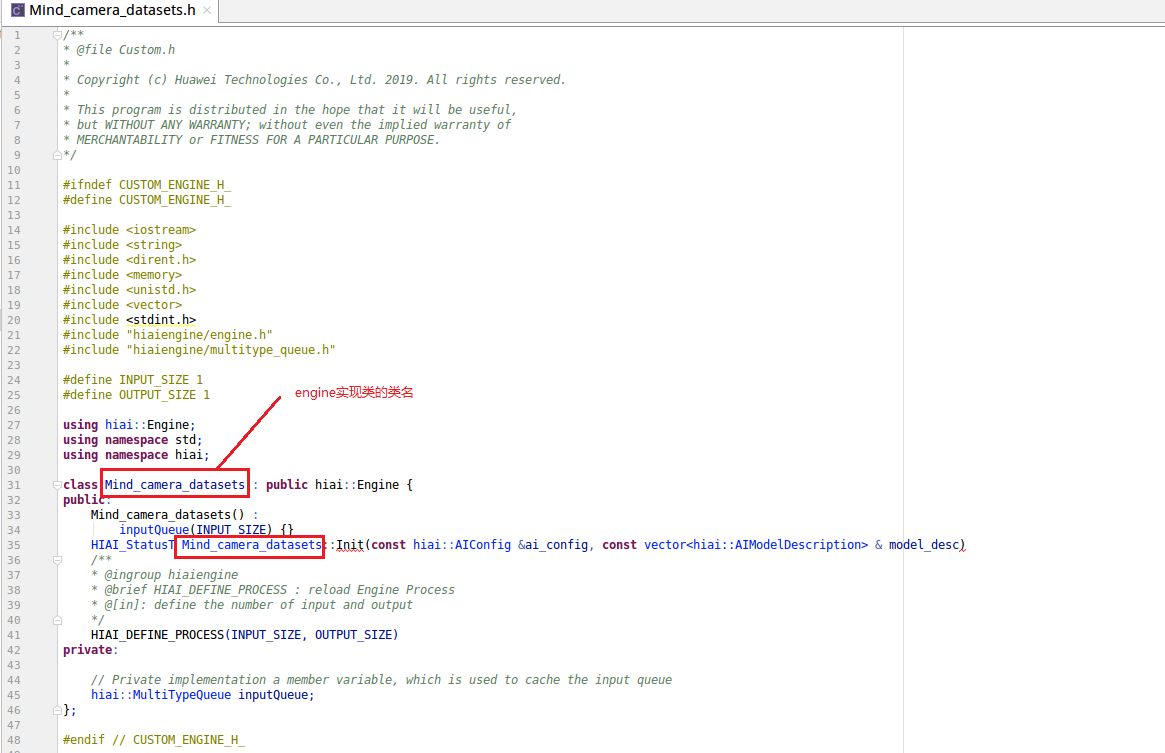


1. 修改每个引擎的源码和头文件，将类名和engine文件名对应，以Mind\_camera\_datasets引擎为例说明，如下图所示：

Mind\_camera\_datasets.cpp文件修改如下图，头文件也对应修改。



Mind\_camera\_datasets.h模板文件



同理其他2个engine修改类名和对应的头文件。

### 任务三 引擎配置文件编写

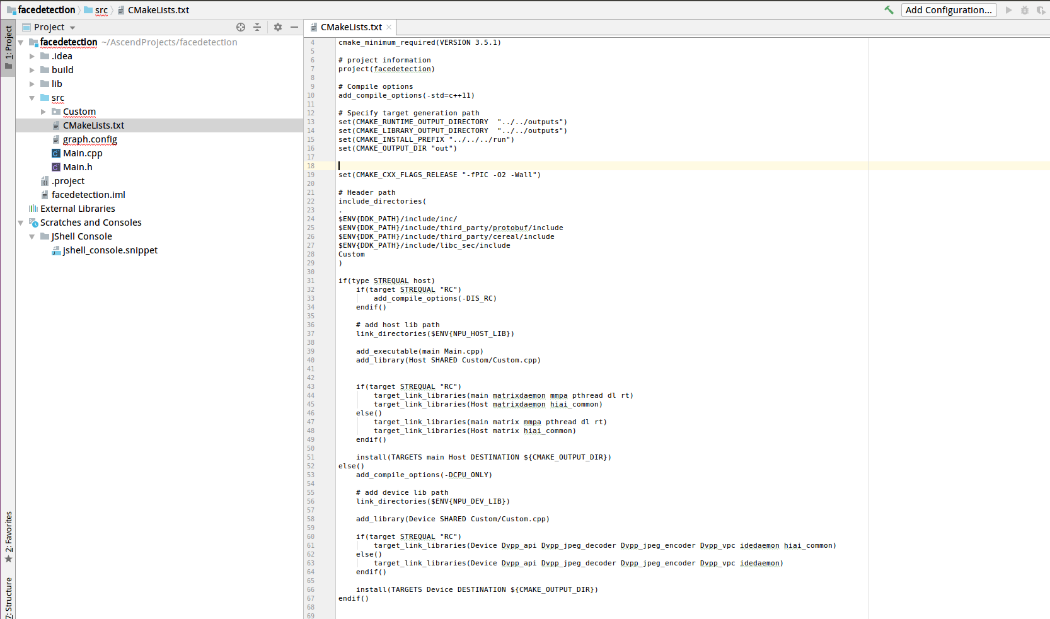
此时3个engine模板框架已经搭建好了，增加了3个engine，对应的graph\_template.config也需要修改，graph\_template.config修改后会通过脚本自动同步到graph.config中，修改如下图所示。



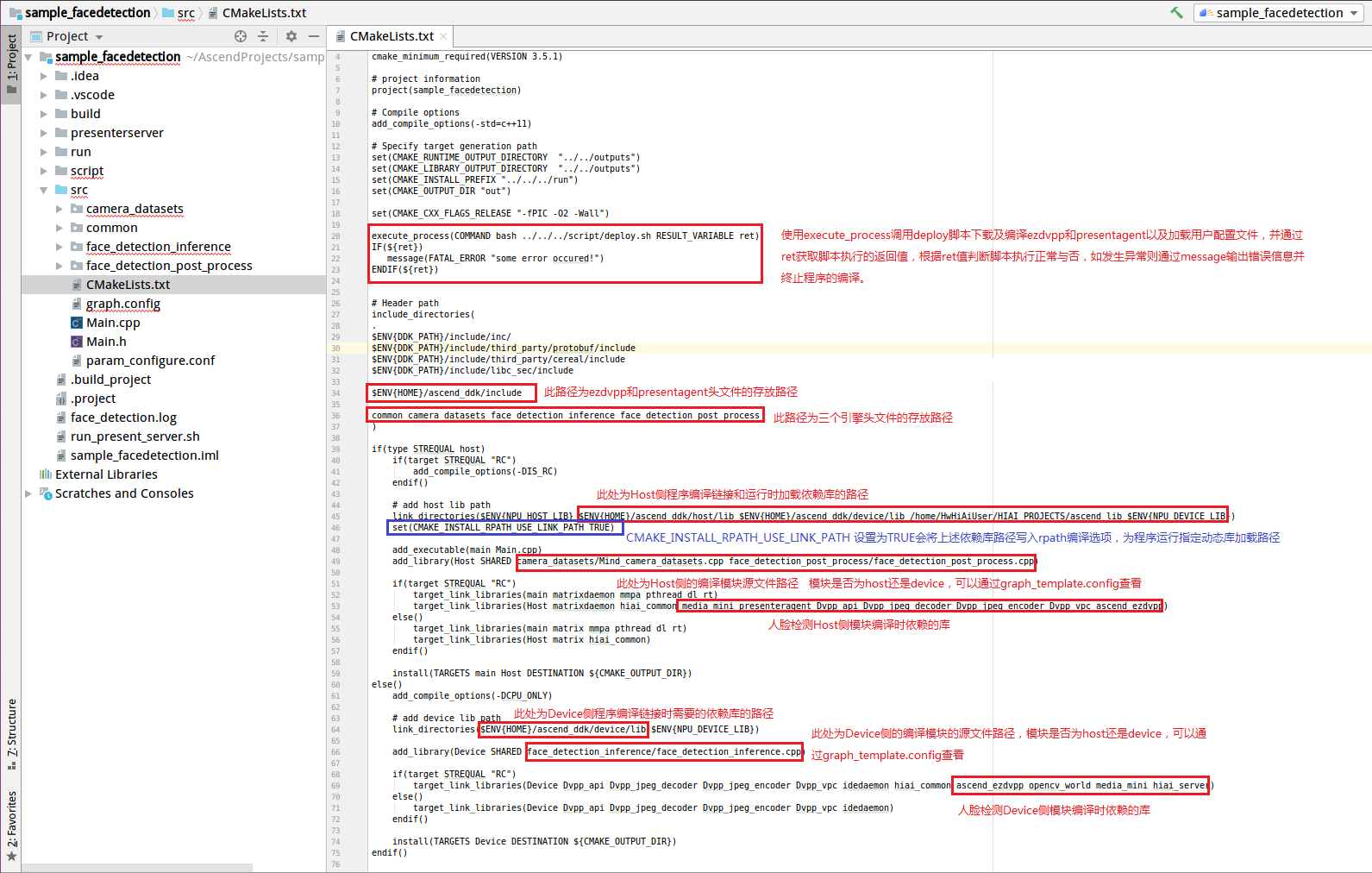
此时engine已经添加完毕，人脸检测还需要presentserver模块在web ui端显示，其中推理模块还依赖EZDVPP，后处理模块依赖Presenter Agent，所以要将presentserver、EZDVPP、Presenter Agent和之前添加的三个engine（face\_detection\_inference、face\_detection\_post\_process、Mind\_camera\_datasets）添加到编译文件。

### 任务四 编译文件编写

自动生成的cmakelist.txt如下图所示：



Sample\_facedetection的CmakeList.txt如下图所示，增加的部分用红色标记说明，可对比参考码云源码的CmakeList.txt



### 任务五 应用代码编写

源码可以从https://gitee.com/Atlas200DK/sample-facedetection/tree/ly/仓中的代码下载至安装Mind Studio所在的Ubuntu服务器的任意目录

跟自定义工程相比，增加了

* script目录，
* presenterserver目录，
* src/目录下增加了三个引擎文件camera\_datasets，face\_detection\_inference，face\_detection\_post\_process和用户配置文件：param\_configure.conf。
* run\_present\_server.sh : 配置及开启presenter\_server展示界面的脚本

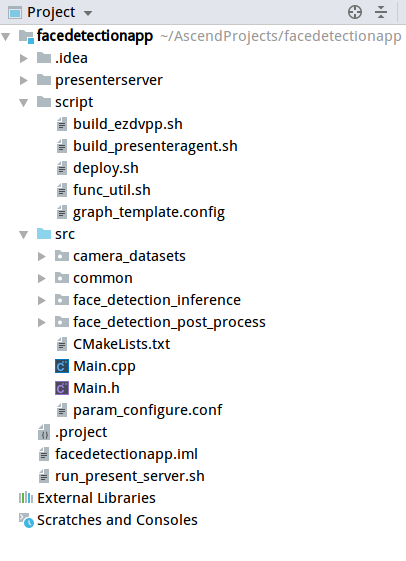
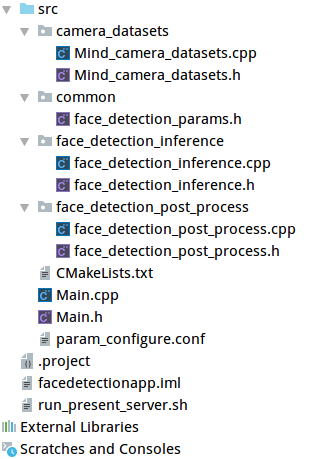
具体如图6.1所示的文件目录结构，各个目录说明如下：

* presenterserver：存放Presenter Server源码，根据接收到的推理结果，在JPEG图片上进行人脸位置及置信度的标记，并将图像信息推送给Web UI。
* param\_configure.conf：此文件中定义了开发板的ip地址，摄像头的通道以及presenterserver展示的名字。
* script目录：包含人脸检测代码要依赖两个公共库，ezdvpp和presentagent。

EZDVPP对DVPP接口进行了封装，用于简化直接调用DVPP接口的复杂度，主要功能包括：Presentagent将摄像头采集到的图像格式转化（包含将YUV转换为JPG，JPEG转换为YUV），图像的crop以及图像的resize。

Presenter Agent将接受到的JPEG图片及推理结果发送到UI HOST上部署的Presenter Server服务进程。此目录下还存放有部署脚本，引擎配置文件以及模型文件：部署脚本文件：deploy.sh；脚本功能函数存放文件：func\_util.sh；下载，编译以及部署ezdvpp的脚本文件：build\_ezdvpp.sh；下载，编译及部署presenteragent的脚本文件：build\_presenteragent.sh；引擎配置文件：graph\_template.config；人脸检测模型文件。

1. deploy.sh部署脚本功能说明：
2. 从./src目录下的param\_configure.conf文件中读取用户输入的参数
3. 将./script目录下的引擎配置文件：graph\_template.config复制到./src目录下，更名为graph.config （graph\_template.config是graph.config的模板文件，最终程序编译时要使用graph.config）
4. 将从a)中获取到的参数经过处理,到./src/graph.config中替换ai\_config中的value值。
5. 下载，编译以及部署库文件到开发板指定路径中：1.调用build\_ezdvpp.sh实现下载，编译以及部署ezdvpp；2.调用build\_presenteragent.sh实现下载，编译及部署presenteragent

（a）项目工程目录 （b）项目源文件目录

图6.1 facedetection项目工程文件目录结构图

main.cpp编写

main.cpp文件主要功能如图6.2所示，具体描述及代码实现详见下文：

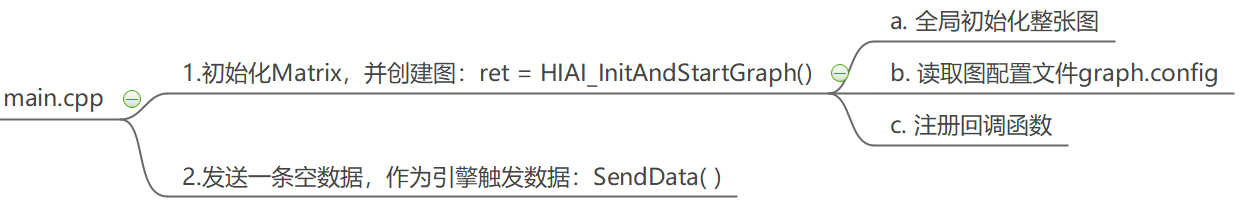


图6.2 main.cpp函数文件功能

初始化Matrix、创建图，main函数主体如下所示：

int main(int argc, char\* argv[])  
{  
 HIAI\_StatusT ret = HIAI\_OK;  
 char \* dirc = strdup(argv[0]);  
 if (dirc)  
 {  
 char \* dname = ::dirname(dirc);  
 chdir(dname);  
 HIAI\_ENGINE\_LOG("chdir to %s", dname);  
 free(dirc);  
 }  
 // 1.创建图  
 ret = HIAI\_InitAndStartGraph();  
  
 // 2.向下一个引擎发送数据：从main方法中，调用SendData方法给数据集引擎发送消息；数据集引擎收到后开始设置摄像头属性并产生图像数据

// 根据图的ID（GRAPH\_ID），拿到当前工程的图对象 graph  
 std::shared\_ptr<hiai::Graph> graph = hiai::Graph::GetInstance(GRAPH\_ID);  
   
 // 将数据送到SourceEngine的0号端口  
 hiai::EnginePortID engine\_id;  
 engine\_id.graph\_id = GRAPH\_ID;  
 engine\_id.engine\_id = 958; // 数据集引擎Mind\_camera\_datasets的编号  
 engine\_id.port\_id = 0; //数据集引擎Mind\_camera\_datasets的输入端口：0  
 std::shared\_ptr<std::string> src\_data(new std::string);  
 graph->SendData(engine\_id, "string", std::static\_pointer\_cast<void>(src\_data));

//主线程等待子线程发出退出指令。引擎可以注册SetDataRecvFunctor回调，控制flag的值。当flag被赋值为小于0的值时，主线程结束等待，销毁graph，整个APP退出。  
 for (;;)  
 {  
 if(flag <= 0)  
 {  
 break;  
 }else  
 {  
 usleep(100000);  
 }  
 }  
 hiai::Graph::DestroyGraph(GRAPH\_ID);  
 return 0;  
}

其中，HIAI\_InitAndStartGraph()函数展开内容如下所示：

HIAI\_StatusT HIAI\_InitAndStartGraph()  
{  
 // Step1: 在运行引擎之前，需要全局初始化整张图  
 HIAI\_StatusT status = HIAI\_Init(0);  
  
 // Step2: 读取图配置文件graph.config并创建图  
 status = hiai::Graph::CreateGraph("./graph.config");  
  
 // Step3 注册回调函数：向三个engine中的最后一个engine（本实验为后处理引擎）注册一个回调方法  
 std::shared\_ptr<hiai::Graph> graph = hiai::Graph::GetInstance(GRAPH\_ID);  
 int leaf\_array[1] = {601}; //最后一个引擎的id，即后处理引擎  
 //这里只注册一个回调方法，因此for循环只执行一次，后续可以根据实际需求，选择注册多个回调函数方法  
 for(int i = 0;i < 1;i++){  
 hiai::EnginePortID target\_port\_config;  
 target\_port\_config.graph\_id = GRAPH\_ID;  
 target\_port\_config.engine\_id = leaf\_array[i];   
 target\_port\_config.port\_id = 0;//最后一个引擎的输出端口：0号端口  
 graph->SetDataRecvFunctor(target\_port\_config,  
 std::shared\_ptr<CustomDataRecvInterface>(  
 new CustomDataRecvInterface("")));  
 }  
 return HIAI\_OK;  
}

其中，回调函数方法如下所示：

HIAI\_StatusT CustomDataRecvInterface::RecvData  
 (const std::shared\_ptr<void>& message)  
 //message指最后一个引擎（本实验中为后处理引擎）中输出的信息  
{  
 std::shared\_ptr<std::string> data =  
 std::static\_pointer\_cast<std::string>(message);

//flag是一个全局变量，主线程和所有的引擎线程都可以访问，为了防止多线程竞争，需要在访问前加锁，访问结束后解锁。  
 mt.lock();  
 flag--;   
 mt.unlock();  
 return HIAI\_OK;  
}

至此，main.cpp的主要代码实现及说明已介绍完毕，接下来将分别介绍本实验的3个引擎实现的功能代码。

Mind\_camera\_datasets.cpp编写

在本实验项目中，不需要用到main函数传入的数据，只需要从main函数文件中获得一个触发数据激活Mind\_camera\_dataset引擎即可，Camera会自动采集获取外部数据。

在本实验项目中，3个引擎文件.cpp中，都会包含一个初始化函数和一个数据处理函数，Mind\_camera\_datasets.cpp文件结构框架如图6.3所示：

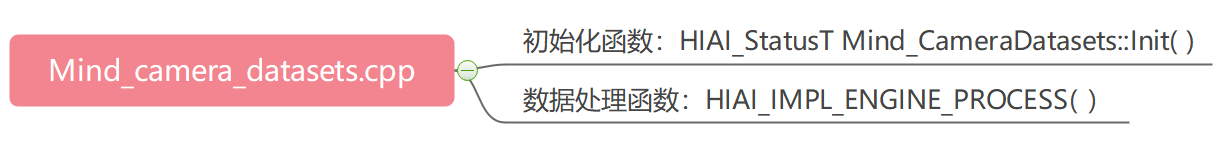


图6.3 Mind\_camera\_datasets.cpp文件结构框架

其中，数据处理函数的主要功能函数结构如图6.4所示：

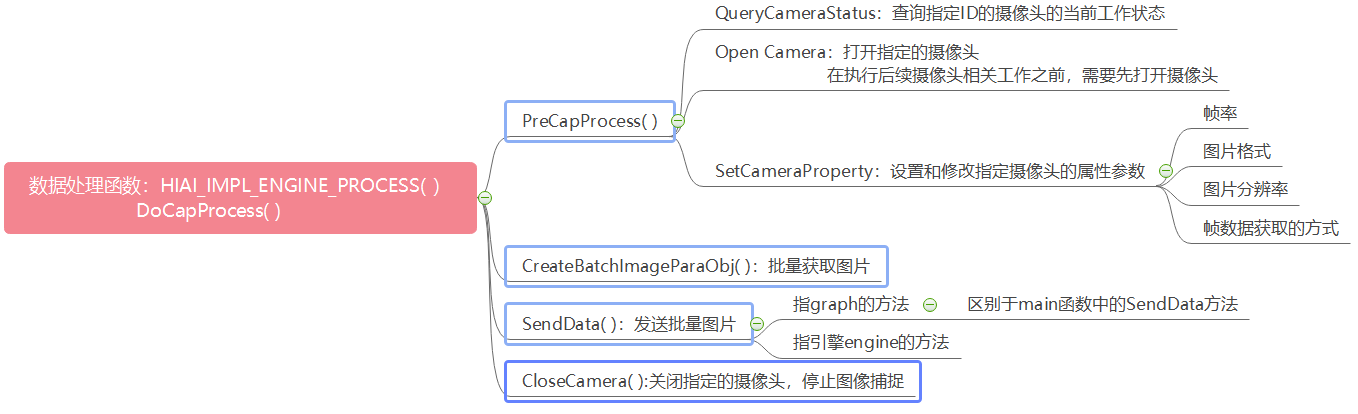


图6.4 Camera数据集引擎数据处理函数的主要功能函数结构

数据处理函数HIAI\_IMPL\_ENGINE\_PROCESS( )具体代码实现如下所示：

HIAI\_IMPL\_ENGINE\_PROCESS("Mind\_camera\_datasets", Mind\_CameraDatasets, INPUT\_SIZE)  
{  
 HIAI\_ENGINE\_LOG("[CameraDatasets] start process!");  
 DoCapProcess();//这里不需要使用main方法传进来的数据，只需要main方法做一个传输数据的动作，数据集引擎接收到这个传输数据的动作后，开始设置摄像头属性并获取图像数据  
 HIAI\_ENGINE\_LOG("[CameraDatasets] end process!");  
 return HIAI\_OK;  
}

其中，DoCapProcess( )函数主要代码内容及相关注释如下所示：

bool Mind\_CameraDatasets::DoCapProcess() {  
 CameraOperationCode ret\_code = PreCapProcess();  
  
 SetExitFlag (CAMERADATASETS\_RUN);  
  
 HIAI\_StatusT hiai\_ret = HIAI\_OK;  
 int read\_ret = 0;  
 int read\_size = 0;  
 bool read\_flag = false;  
 while (GetExitFlag() == CAMERADATASETS\_RUN) {

//CreateBatchImageParaObj创建BatchImageParaWithScaleT结构体,此结构体为接收一帧图像数据开辟缓冲区  
 shared\_ptr < BatchImageParaWithScaleT > pobj = CreateBatchImageParaObj();  
 NewImageParaT\* pimg\_data = &pobj->v\_img[0];  
 uint8\_t\* pdata = pimg\_data->img.data.get();  
 read\_size = (int) pimg\_data->img.size;  
  
 // do read frame from camera  
 read\_ret = ReadFrameFromCamera(config\_->channel\_id, (void\*) pdata,  
 &read\_size);  
 // indicates failure when readRet is 1  
 read\_flag = ((read\_ret == 1) && (read\_size == (int) pimg\_data->img.size));  
 }  
  
 // "BatchImageParaWithScaleT" 表示 向下一个引擎中发送数据的类型为该类型  
 // SendData方法中的第一个参数0，表示在本引擎中，即Camera引擎中的0号端口输出数据  
 hiai\_ret = SendData(0, "BatchImageParaWithScaleT",  
 static\_pointer\_cast<void>(pobj));  
 }  
 }  
  
 // close camera：关闭指定的摄像头，停止图像捕捉  
 CloseCamera(config\_->channel\_id);  
 return true;  
}

其中，数据处理函数的主要调用函数为DoCapProcess( )函数，如图5.21所示，在DoCapProcess( )函数中，还涉及到PreCapProcess( )函数、CreateBatchImageParaObj( )函数、SendData( )函数以及CloseCamera( )函数，这些函数的功能已在图5.21中做了说明，这里不做赘述，主要列出相关函数功能的代码实现如下：

**PreCapProcess( )函数**

Mind\_CameraDatasets::CameraOperationCode Mind\_CameraDatasets::PreCapProcess() {  
 MediaLibInit();//这里调用第三方库，详见产品文档"API参考"  
 // QueryCameraStatus：查询指定ID的摄像头的当前工作状态  
 // 详见API参考 > Media API参考 > 媒体库接口 > QueryCameraStatus  
 CameraStatus status = QueryCameraStatus(config\_->channel\_id);  
 if (status != CAMERA\_STATUS\_CLOSED) {  
 HIAI\_ENGINE\_LOG("[CameraDatasets] PreCapProcess.QueryCameraStatus "  
 "{status:%d} failed.",status);  
 return kCameraNotClosed;  
 }  
  
 // Open Camera：打开指定的摄像头，在执行后续摄像头相关工作之前，需要先打开摄像头  
 // 详见文档 Atlas 200 DK > API参考 > Media API参考 > 媒体库接口 > OpenCamera  
 int ret = OpenCamera(config\_->channel\_id);  
 // 如果返回0，表示打开摄像头失败  
 if (ret == 0) {  
 HIAI\_ENGINE\_LOG("[CameraDatasets] PreCapProcess OpenCamera {%d} "  
 "failed.",config\_->channel\_id);  
 return kCameraOpenFailed;  
 }  
  
 // SetCameraProperty：设置和修改指定摄像头的属性参数，这里设置帧率  
 ret = SetCameraProperty(config\_->channel\_id, CAMERA\_PROP\_FPS,  
 &(config\_->fps));  
 // 返回0表示 设置摄像头属性失败  
 if (ret == 0) {  
 HIAI\_ENGINE\_LOG("[CameraDatasets] PreCapProcess set fps {fps:%d} "  
 "failed.",config\_->fps);  
 return kCameraSetPropeptyFailed;  
 }  
  
 // 设置采集数据的图片格式  
 ret = SetCameraProperty(config\_->channel\_id, CAMERA\_PROP\_IMAGE\_FORMAT,  
 &(config\_->image\_format));  
 // 返回0表示 设置摄像头属性失败  
 if (ret == 0) {  
 HIAI\_ENGINE\_LOG("[CameraDatasets] PreCapProcess set image\_fromat "  
 "{format:%d} failed.",config\_->image\_format);  
 return kCameraSetPropeptyFailed;  
 }  
  
 // 设置图片分辨率大小  
 CameraResolution resolution;  
 resolution.width = config\_->resolution\_width;  
 resolution.height = config\_->resolution\_height;  
 ret = SetCameraProperty(config\_->channel\_id, CAMERA\_PROP\_RESOLUTION,  
 &resolution);  
 // 返回0表示 设置摄像头属性失败  
 if (ret == 0) {  
 HIAI\_ENGINE\_LOG("[CameraDatasets] PreCapProcess set resolution "  
 "{width:%d, height:%d } failed.",  
 config\_->resolution\_width, config\_->resolution\_height);  
 return kCameraSetPropeptyFailed;  
 }  
  
 // 设置帧数据获取的方式  
 CameraCapMode mode = CAMERA\_CAP\_ACTIVE;  
 ret = SetCameraProperty(config\_->channel\_id, CAMERA\_PROP\_CAP\_MODE, &mode);  
 // 返回0表示 设置摄像头属性失败  
 if (ret == 0) {  
 HIAI\_ENGINE\_LOG("[CameraDatasets] PreCapProcess set cap mode {mode:%d}"  
 " failed.",mode);  
 return kCameraSetPropeptyFailed;  
 }  
  
 return kCameraOk;  
}

**CreateBatchImageParaObj( )函数：**

shared\_ptr<BatchImageParaWithScaleT>  
// CreateBatchImageParaObj创建BatchImageParaWithScaleT结构体

Mind\_CameraDatasets::CreateBatchImageParaObj() {  
 shared\_ptr < BatchImageParaWithScaleT > pobj = make\_shared<  
 BatchImageParaWithScaleT>();  
//初始化结构体成员，保存Camera设置参数  
 pobj->b\_info.is\_first = (frame\_id\_ == kInitFrameId);  
 pobj->b\_info.is\_last = false;  
 // handle one batch every time  
 pobj->b\_info.batch\_size = 1;  
 pobj->b\_info.max\_batch\_size = kMaxBatchSize;  
 pobj->b\_info.batch\_ID = 0;  
 pobj->b\_info.channel\_ID = config\_->channel\_id;  
 pobj->b\_info.processor\_stream\_ID = 0;  
 pobj->b\_info.frame\_ID.push\_back(frame\_id\_++);  
 pobj->b\_info.timestamp.push\_back(time(nullptr));  
 //根据设定的图像格式和分辨率事先计算好缓冲区大小，以确保缓冲区大小足够容纳输出的图像数据  
 NewImageParaT img\_data;  
 // channel begin from zero  
 img\_data.img.channel = 0;  
 img\_data.img.format = YUV420SP;  
 img\_data.img.width = config\_->resolution\_width;  
 img\_data.img.height = config\_->resolution\_height;  
 // YUV size in memory is width\*height\*3/2  
 img\_data.img.size = config\_->resolution\_width \* config\_->resolution\_height \* 3  
 / 2;  
 //为接收一帧图像数据开辟缓冲区  
 shared\_ptr <uint8\_t> data(new uint8\_t[img\_data.img.size],  
 default\_delete<uint8\_t[]>());  
 img\_data.img.data = data;  
  
 pobj->v\_img.push\_back(img\_data);  
  
 return pobj;  
}

**SendData( )函数**：

// "BatchImageParaWithScaleT" 表示 向下一个引擎中发送数据的类型为该类型  
// SendData方法中的第一个参数0，表示在本引擎中，即Camera引擎中的0号端口输出数据  
hiai\_ret = SendData(0, "BatchImageParaWithScaleT",  
 static\_pointer\_cast<void>(pobj));

**CloseCamera( )函数：**

// close camera：关闭指定的摄像头，停止图像捕捉  
CloseCamera(config\_->channel\_id);

face\_detection\_inference.cpp编写

face\_detection\_inference.cpp文件结构框架如图6.5所示，依然是包含了2个主要的函数：初始化函数和数据处理函数，其中数据处理函数主要功能结构如图6.6所示，下文将对各个函数功能的代码实现做详细呈现。



图6.5 face\_detection\_inference.cpp文件结构框架

在推理引擎中，有需要对图片数据进行预处理操作，本实验项目将数据预处理环节与推理过程编都统一排在了推理引擎节点，如图6.6所示，在数据处理函数功能实现框架中，执行推理过程之前，需要对图片进行相关的预处理操作。

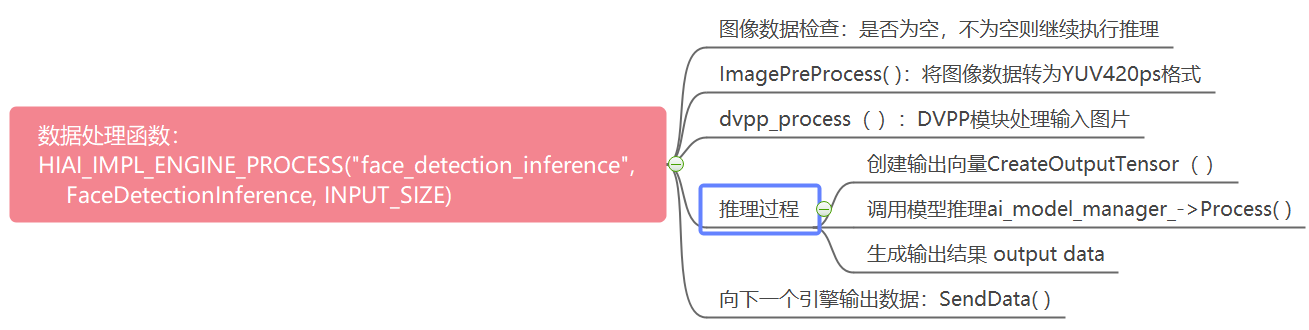


图6.6 推理引擎数据处理函数的主要功能结构

其中，推理过程的代码实现及相关注释说明如下所示：

// 1. 创建输出向量  
hiai::AIContext ai\_context;  
std::vector<std::shared\_ptr<hiai::IAITensor>> output\_data\_vector;  
hiai::AIStatus ret = ai\_model\_manager\_->CreateOutputTensor(  
 input\_data\_vec, output\_data\_vector);// input\_data\_vec 为输入模型的输入向量  
 SendData(kSendDataPort, "EngineTransT",  
 std::static\_pointer\_cast<void>(trans\_data));  
 return HIAI\_ERROR;  
}  
  
// 2. process 调用模型推理  
HIAI\_ENGINE\_LOG("aiModelManager->Process start!");  
ret = ai\_model\_manager\_->Process(ai\_context, input\_data\_vec,  
 output\_data\_vector,  
 AI\_MODEL\_PROCESS\_TIMEOUT);  
 SendData(kSendDataPort, "EngineTransT",  
 std::static\_pointer\_cast<void>(trans\_data));  
 return HIAI\_ERROR;  
}  
HIAI\_ENGINE\_LOG("aiModelManager->Process end!");  
  
// 生成输出结果 output data  
trans\_data->status = true;

face\_detection\_post\_process.cpp文件

本实验项目的后处理引擎，是在接收到推理引擎发送的数据后，整理数据格式并调用PresenterServer的接口，进行检测结果在主机侧的回显。如同上述2）、3）所述文件结构，face\_detection\_post\_process.cpp文件结构框架也有2个，如图6.7所示，其中，初始化函数主要是为了获取图配置信息，建立PresenterServer链接，便于检测结果的回显。

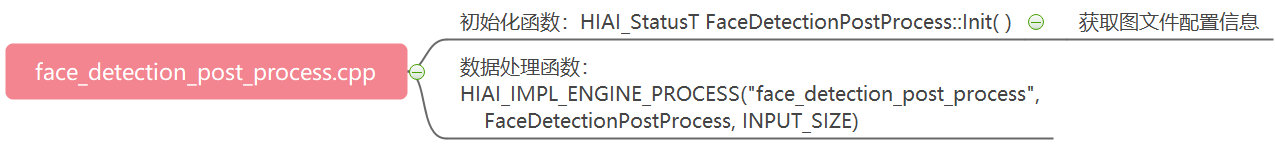


图6.7 face\_detection\_post\_process.cpp文件结构框架



图6.8 后处理引擎数据处理函数的主要功能结构

后处理引擎数据处理函数HIAI\_IMPL\_ENGINE\_PROCESS( )代码实现如下所示：

HIAI\_IMPL\_ENGINE\_PROCESS("face\_detection\_post\_process",  
 FaceDetectionPostProcess, INPUT\_SIZE) {  
 // 检查传入数据是否为空，不为空则继续向下处理  
 if (arg0 == nullptr) {  
 HIAI\_ENGINE\_LOG(HIAI\_ENGINE\_RUN\_ARGS\_NOT\_RIGHT,  
 "Failed to process invalid message.");  
 return HIAI\_ERROR;  
 }  
  
 // 检查原始图片数据是否为空，并将数据类型转换为“EngineTransT”类型  
 std::shared\_ptr<EngineTransT> inference\_res = std::static\_pointer\_cast<  
 EngineTransT>(arg0);  
   
 // 推理失败，则返回处理原始图片  
 if (!inference\_res->status) {  
 HIAI\_ENGINE\_LOG(HIAI\_OK, inference\_res->msg.c\_str());  
 HIAI\_ENGINE\_LOG(HIAI\_OK, "will handle original image.");  
 return HandleOriginalImage(inference\_res);  
 }  
  
 // 推理成功则继续处理推理结果  
 return HandleResults(inference\_res);  
}

这里，主要介绍推理成功后，处理推理结果的函数代码实现，如下所示：

HIAI\_StatusT FaceDetectionPostProcess::HandleResults(  
 const std::shared\_ptr<EngineTransT> &inference\_res) {  
 HIAI\_StatusT status = HIAI\_OK;  
 std::vector<NewImageParaT> img\_vec = inference\_res->imgs;  
 std::vector<OutputT> output\_data\_vec = inference\_res->output\_datas;//从inference\_res结果中取出推理的输出数据output\_datas  
 // 处理每一张图  
 for (uint32\_t ind = 0; ind < inference\_res->b\_info.batch\_size; ind++) {  
 // result  
 int32\_t out\_index = ind \* kDealResultIndex;  
 OutputT out = output\_data\_vec[out\_index];  
 std::shared\_ptr<hiai::AISimpleTensor> result\_tensor = std::make\_shared<  
 hiai::AISimpleTensor>();  
 result\_tensor->SetBuffer(out.data.get(), out.size);  
 int32\_t size = result\_tensor->GetSize() / sizeof(float);  
 float result[size];  
 errno\_t mem\_ret = memcpy\_s(result, sizeof(result),  
 result\_tensor->GetBuffer(),  
 result\_tensor->GetSize());  
 // 如果内存拷贝图片失败，则跳过对这张图片的处理  
 if (mem\_ret != EOK) {  
 HIAI\_ENGINE\_LOG(HIAI\_ENGINE\_RUN\_ARGS\_NOT\_RIGHT,  
 "handle results: memcpy\_s() error=%d", mem\_ret);  
 continue;  
 }  
  
 uint32\_t width = img\_vec[ind].img.width;  
 uint32\_t height = img\_vec[ind].img.height;  
 uint32\_t img\_size = img\_vec[ind].img.size;  
  
 // 循环处理推理结果  
 std::vector<DetectionResult> detection\_results;  
 float \*ptr = result;  
 for (int32\_t k = 0; k < size - kEachResultSize; k += kEachResultSize) {  
 ptr = result + k;  
 // attribute  
 float attr = ptr[kAttributeIndex];  
 // confidence  
 float score = ptr[kScoreIndex];  
  
 //定义一个检测结果  
 DetectionResult one\_result;  
 // 左上角坐标  
 Point point\_lt, point\_rb;  
 point\_lt.x = ptr[kAnchorLeftTopAxisIndexX] \* width;  
 point\_lt.y = ptr[kAnchorLeftTopAxisIndexY] \* height;  
 // 右下角坐标  
 point\_rb.x = ptr[kAnchorRightBottomAxisIndexX] \* width;  
 point\_rb.y = ptr[kAnchorRightBottomAxisIndexY] \* height;  
  
 one\_result.lt = point\_lt;  
 one\_result.rb = point\_rb;  
  
 HIAI\_ENGINE\_LOG(HIAI\_DEBUG\_INFO,  
 "score=%f, lt.x=%d, lt.y=%d, rb.x=%d, rb.y=%d", score,  
 point\_lt.x, point\_lt.y, point\_rb.x, point\_rb.y);  
 int32\_t score\_percent = score \* kScorePercent;  
 one\_result.result\_text.append(kFaceLabelTextPrefix);  
 one\_result.result\_text.append(to\_string(score\_percent));  
 one\_result.result\_text.append(kFaceLabelTextSuffix);  
  
 // push back  
 detection\_results.emplace\_back(one\_result);  
 }  
  
 int32\_t ret;  
 // 把结果中的数据发送到PresenterServer  
 ret = SendImage(height, width, img\_size, img\_vec[ind].img.data.get(), detection\_results);  
 }

处理推理成功的检测结果时，这里的SendImage( )不是向下一个引擎发送数据，而是将检测结果中的关键数据向PresenterServer发送，在主机侧接收SendImage( )的数据后，利用坐标、置信度、属性等数据进行检测结果的可视化，本实验项目是利用PresenterServer实现检测结果的显示，读者也可根据实际情况，自定义人脸检测结果的显示方式，例如直接将检测数据结果打印在终端里显示，或者将检测数据结果保存在文件中。SendImage( )函数具体实现代码如下所示：

// 函数接口参数：图片的高、宽、尺寸、数据

int32\_t FaceDetectionPostProcess::SendImage(uint32\_t height, uint32\_t width,  
 uint32\_t size, u\_int8\_t \*data, std::vector<DetectionResult>& detection\_results) {  
 int32\_t status = kFdFunSuccess;  
 // 相关参数  
 ImageFrame image\_frame\_para;  
 image\_frame\_para.format = ImageFormat::kJpeg;  
 image\_frame\_para.width = width;  
 image\_frame\_para.height = height;  
 image\_frame\_para.size = size;  
 image\_frame\_para.data = data;  
 image\_frame\_para.detection\_results = detection\_results;  
  
 PresenterErrorCode p\_ret = PresentImage(presenter\_channel\_.get(),  
 image\_frame\_para);  
 return status;  
}

至此，在主机侧的项目代码编写实现的思路已介绍完毕，接下来需要编写CMakeLists.txt编译文件以及将本实验项目用到的模型进行转换，以便后续部署在Atlas开发板上，进行人脸检测推理。

### 任务六 模型转换

该任务将Caffe模型转换为Ascend 310芯片支持的Davinci模型文件，需要通过Mind Studio转换模型功能，实现人脸检测模型的转换。

在本项目中，原始网络模型是基于Caffe的Resnet10-SSD300模型，因此，需要将人脸检测的Caffe模型转换为Davinci模型文件，通常可以在ModelArts平台上完成所需深度学习模型的生成，再将训练好的模型导入Mind Studio中进行模型转换，获得Ascend 310芯片支持的Davinci模型文件，操作步骤如下：

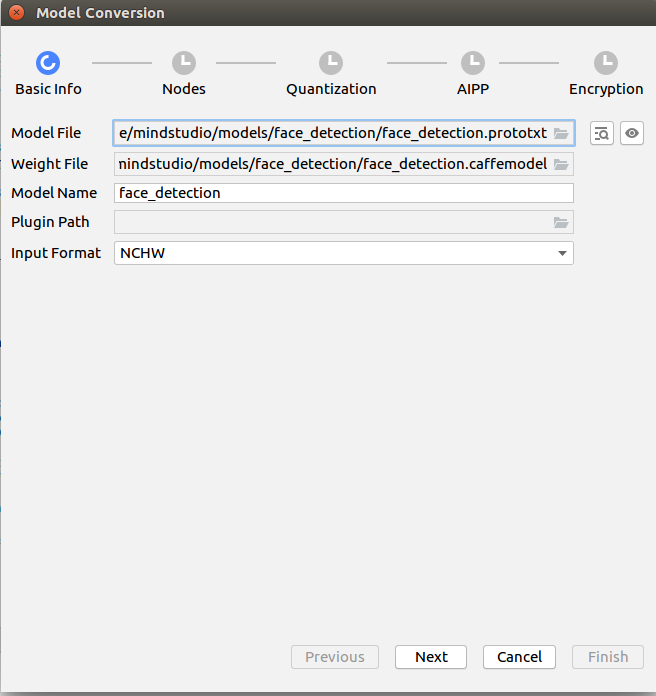
1. 获取人脸检测模型的网络和权重文件：

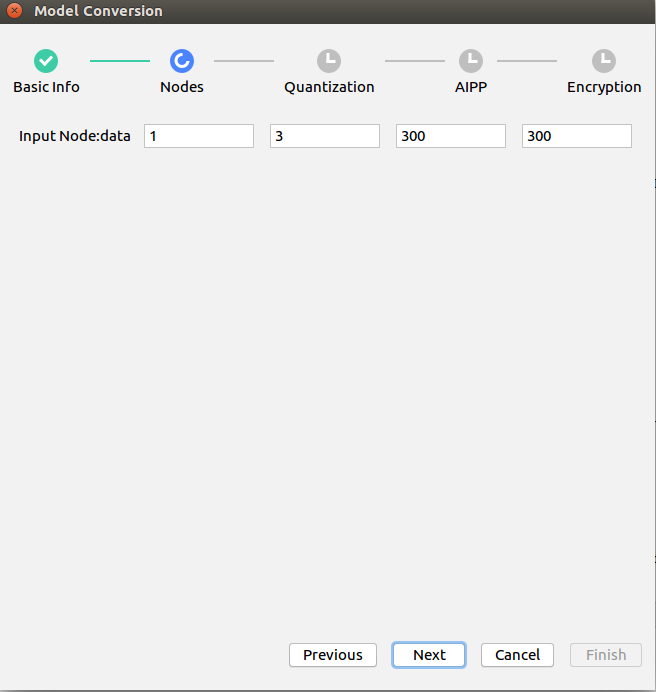
prototxt网络文件下载：从https://gitee.com/HuaweiAscend/models/blob/master/computer\_vision/object\_detect/face\_detection/face\_detection.prototxt处获取已验证的Prototxt下载地址文件。

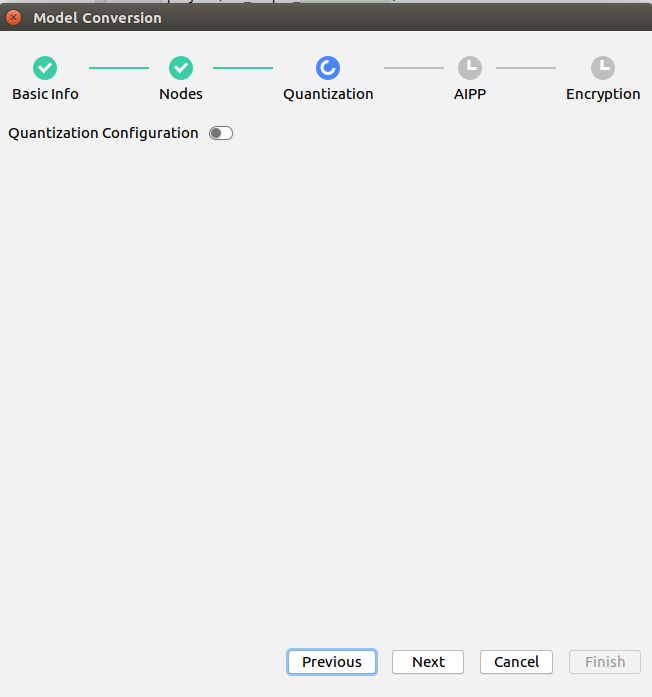
权重文件下载：从https://obs-model-ascend.obs.cn-east-2.myhuaweicloud.com/face\_detection/face\_detection.caffemodel 下载权重文件。

1. 在Mind Studio操作界面的顶部菜单栏中选择“Tools > Model Convert”，进入模型转换界面。在弹出的Model Conversion操作界面中，Model File与Weight File分别选择上述步骤中下载的模型文件和权重文件。

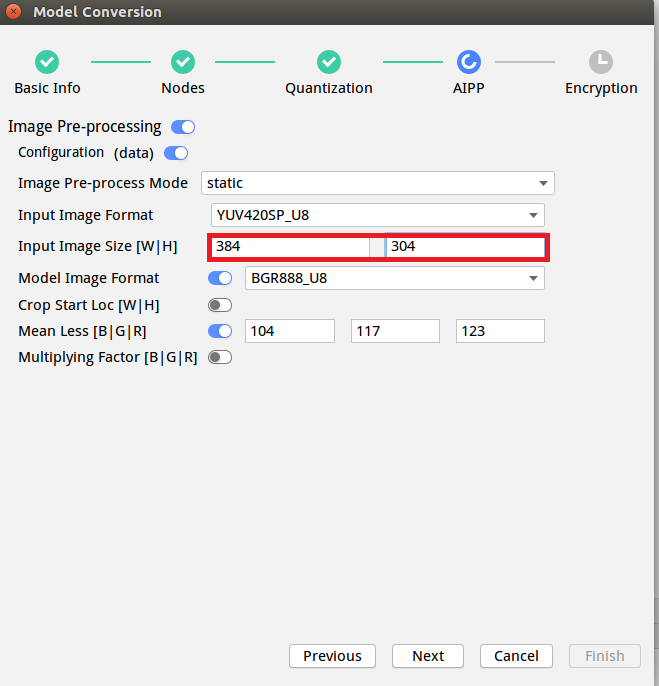
Model Name填写对应的模型名称：face\_detection。







**Input Image Size[W][H]**分别修改为384,304 此处需要做128\*16对齐,如下图所示。



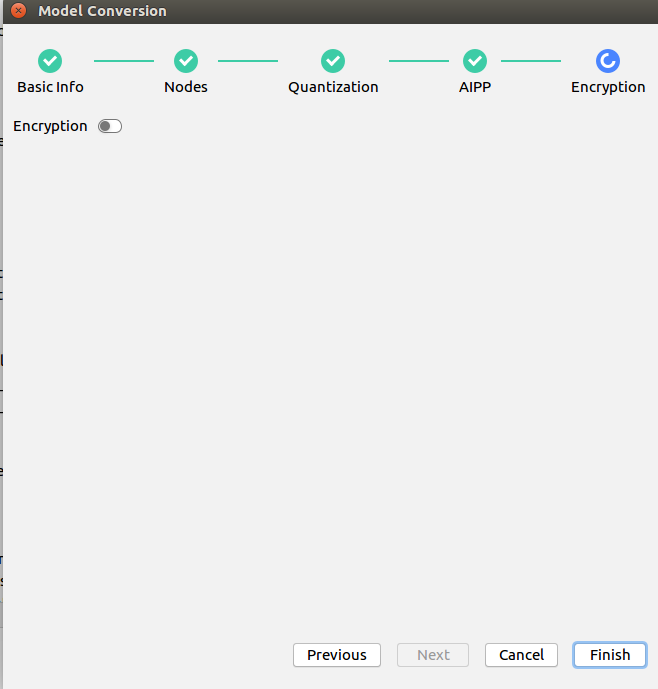


图6.11 离线模型转换示意图

1. 转换时会出现如下报错信息，此时在DetectionOutput层的Suggestion中选择SSDDetectionOutput，并点击Retry，如图6.12所示。

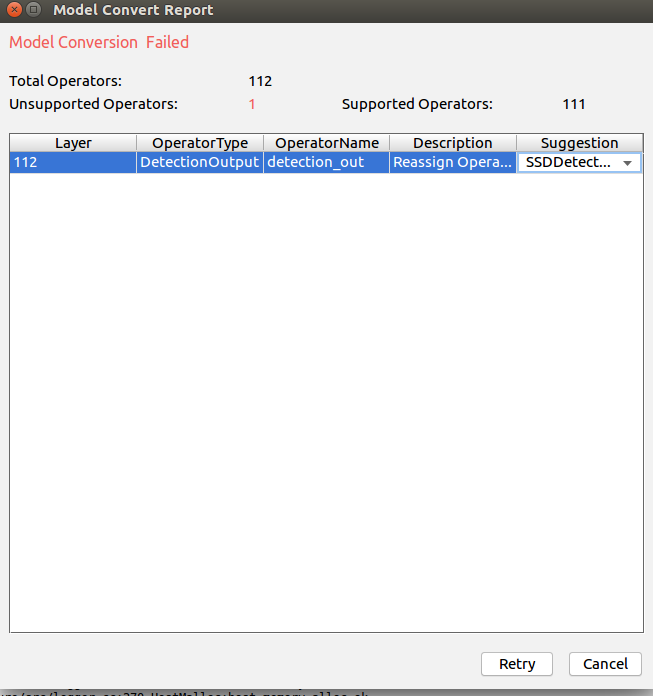
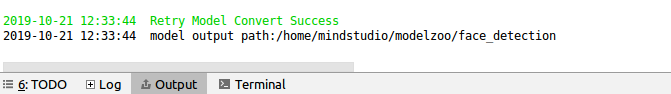


图6.12 离线模型转换报错示意图

1. 转换模型成功后，存放的地址如下图所示路径：$HOME/modelzoo/face\_detection

将转换好的模型文件(.om文件) 上传到“facedetectionapp/script/”目录下。



### 任务七 工程编译及运行

设置环境变量

在安装了Mind Studio所在的Ubuntu主机侧，设置环境变量DDK\_HOME，在终端中输入如下指令：

**vim ~/.bashrc**

执行该命令后，在弹出的新窗口中，按下键盘字母“i”，通过键盘上的向下箭头按键移动光标至文本的最后一行，并在最后一行添加DDK\_HOME，LD\_LIBRARY\_PATH及NPU\_DEVICE\_LIB的环境变量，需要添加的内容如下：

**export DDK\_HOME=$HOME/.mindstudio/huawei/ddk/1.31.T12.B120/ddk**

export NPU\_DEVICE\_LIB=$DDK\_HOME/../RC/host-aarch64\_Ubuntu16.04.3/lib

**export LD\_LIBRARY\_PATH=$DDK\_HOME/lib/x86\_64-linux-gcc5.4**



* 如果此环境变量已经添加，则此步骤可跳过。

完成文本编辑后，键盘按下“**esc**”按键，再继续键盘输入 **:wq!** 保存退出，即完成了环境变量配置。

执行如下命令使环境变量生效。

**source ~/.bashrc**

在配置文件中填写相关参数。在工程中找到src目录，找到param\_configure.conf文件。该配置文件内容如下：

remote\_host=

data\_source=

presenter\_view\_app\_name=

需要手动添加参数配置：

 remote\_host：Atlas 200 DK开发者板的IP地址。

data\_source: 摄像头所属Channel，取值为Channel-1或者Channel-2，查询摄像头所属Channel的方法请参考Atlas 200 DK使用指南中的“如何查看摄像头所属Channel”。

presenter\_view\_app\_name: 用户自定义的在PresenterServer界面展示的View Name，此View Name需要在Presenter Server展示界面唯一。



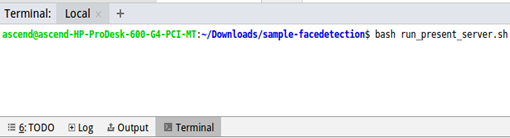
* 三个参数必须全部填写，否则无法通过build。
* 注意所填参数不用使用“”。
* 注意presenter\_view\_app\_name必须是大小写字母、数字、“/”的组合，位数至少为1位

开始build，下在工具栏中找到build 单击Build-Configuration。会在工程目录下生成build和run文件夹。

启动Presenter Server

其中Presenter Server用于接收Application发送过来的数据并通过浏览器进行结果展示。

切换到命令行模式，进入到工程目录下，例如：”facedetectionapp/”。



执行如下命令在后台启动Face Detection应用的Presenter Server主程序。

bash run\_present\_server.sh

当提示“Please choose one to show the presenter in browser(default: 127.0.0.1):”时，请输入在浏览器中访问Presenter Server服务所使用的IP地址（一般为访问Mind Studio的IP地址。）

如图6.13所示，请在“Current environment valid ip list”中选择通过浏览器访问Presenter Server服务使用的IP地址。

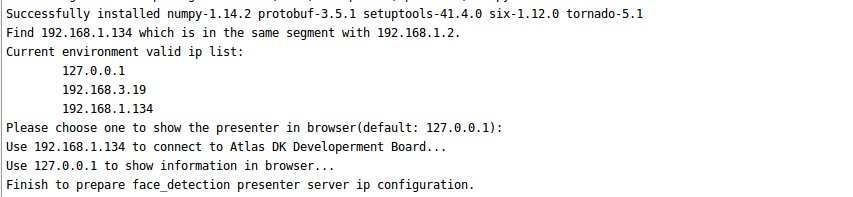


图6.13示意图

如图6.14所示，表示presenter\_server的服务启动成功。

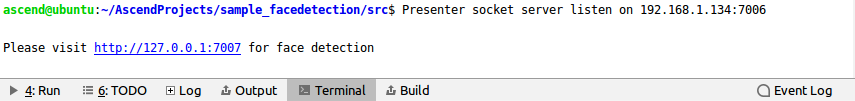
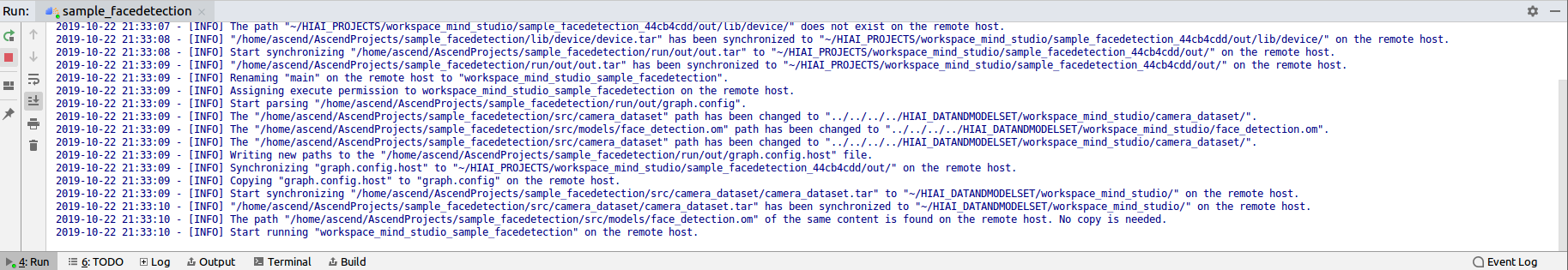


图6.14presenter\_server进程启动

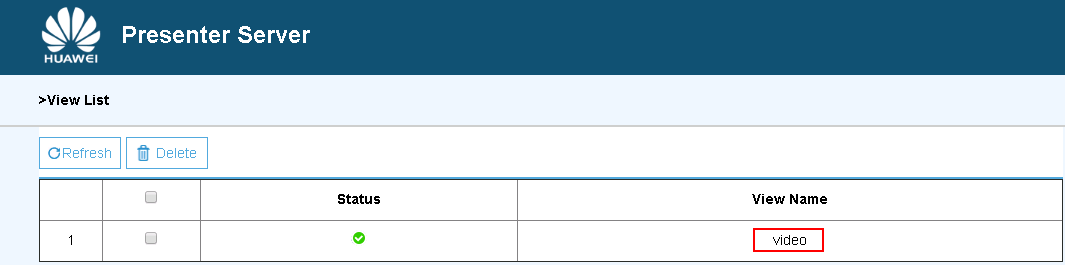
使用上图提示的URL登录Presenter Server，仅支持Chrome浏览器。IP地址为上图中输入的IP地址，端口号默为7007，如下图所示，表示Presenter Server启动成功。



在工具栏找到Run按钮，单击，如下图所示，可执行程序已经在开发板运行。



使用启动Presenter Server服务时提示的URL登录 Presenter Server 网站，等待Presenter Agent传输数据给服务端，单击“Refresh”刷新，当有数据时相应的Channel 的Status变成绿色，如下图所示：





* Face Detection的Presenter Server最多支持10路Channel同时显示，每个 *presenter\_view\_app\_name* 对应一路Channel。
* 由于硬件的限制，每一路支持的最大帧率是20fps，受限于网络带宽的影响，帧率会自动适配为较低的帧率进行展示。

单击右侧对应的View Name链接，比如上图的“video”，查看结果，对于检测到的人脸，会给出置信度的标注。

----结束