Ascend 310
V100R001

HiAI Engine 开发指导

文档版本	01
发布日期	2019-03-12

华为技术有限公司

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

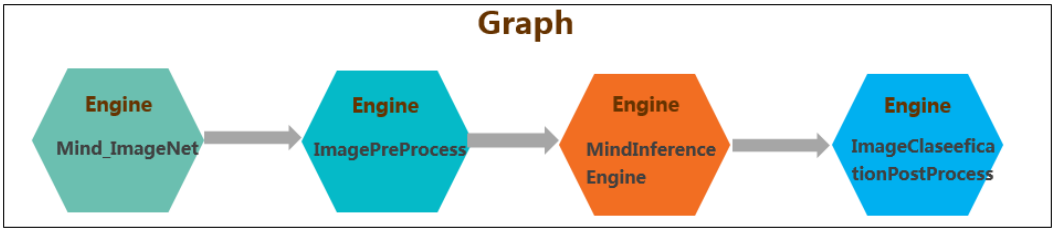
1 概述

1.1 软件概述

HiAI Engine软件运行于操作系统OS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具有多节点调度和多进程管理功能，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中的Engine是一个具体的业务节点，一个业务节点表示一次处理过程（例如某节点可以对图片进行分类处理，输入图片数据，输出对图片数据的分类预测结果），多个Engine组成一个Graph，Graph负责对Engine的管理，如图1-1所示。

图 1-1 Graph 与 Engine 关系示例



每个Engine节点间的连接在Graph配置文件中配置，节点间数据的实际流向根据具体业务在节点中实现，通过向业务的开始节点灌入数据启动整个Engine计算流程，业务节点的属性配置是运行该节点所需的参数。

1.2 Engine 流程执行

流程的执行依赖NPU平台提供的HiAI Engine执行引擎。HiAI Engine是一个通用业务流程执行引擎，主要包含：

- **HiAI Engine Agent**：运行在**host侧**（host侧指与Device相连接的X86服务器、WindowsPC、3559以及AMD服务器等，会利用device提供的NN计算能力，完成业务），其功能如下。

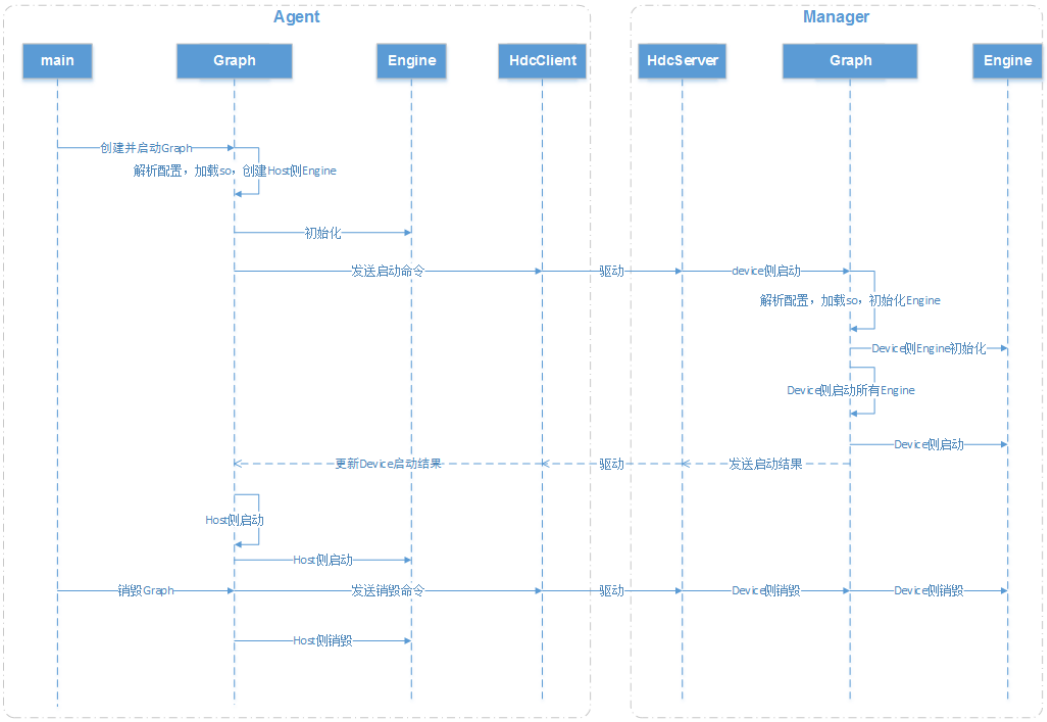
- 提供NPU芯片功能接口。
- 完成与Host APP进行控制命令和处理数据的交互。
- 与NPU间的IPC通信。

- **HiAI Engine Manager**：运行在**Device侧**（Device侧指安装了NPU芯片的硬件设备，利用PCIe接口与host侧连接，为host提供NN计算能力。仿真环境下，HiAI Engine Manager运行在host侧），其功能如下。
  - 根据配置文件完成业务流程的建立。
  - 根据命令完成业务流程的销毁及资源回收。
  - 守护进程，负责IPC功能和Device侧DVPP Executor、Framework、CCE、Runtime等软件的启动。

流程执行时，HiAI Engine会将任务通过HiAI Engine Agent调度到Device侧执行，执行完毕后的结果会在Mind Studio显示。

HiAI Engine Agent与HiAI Engine Manager交互流程请参见**图1-2**。

图 1-2 HiAI Engine 流程图



2 HiAI Engine 开发

用户使用HiAI Engine编程框架有以下两种开发方式。

- Default工程：Mind Studio提供图形化拖拽界面，对AI应用进行编排。此种方式下，框架代码由Mind Studio进行图形化拖拽的时候自动生成，相应的Makefile文件也同时生成，此种方式代码结构相对固定，即Engine流程编排。
- Custom工程：用户自定义工程，此种方式无图形化编排界面，框架代码、Makefile及编译脚本由用户自定义开发，此种方式下，用户代码结构比较灵活，由用户自定义开发。

2.1 Default 工程开发

Mind Studio的Default类型的Mind Engine工程，提供AI引擎可视化拖拽式编程及算法代码自动生成技术，极大的降低了开发者的门槛。业务开发人员通过拖拽图形化业务节点、连接业务节点、编辑业务节点属性的方式编排和运行业务流程，可以实现业务流程编排**“0”**编码。

Default工程的详细开发流程及样例讲解请参见《Mind Studio快速入门》中“构建首个机器学习应用程序”章节。

2.2 自定义工程开发

本节描述通过创建Mind Engine的自定义工程进行机器学习应用开发的端到端流程。目前支持C++语言的自定义开发。

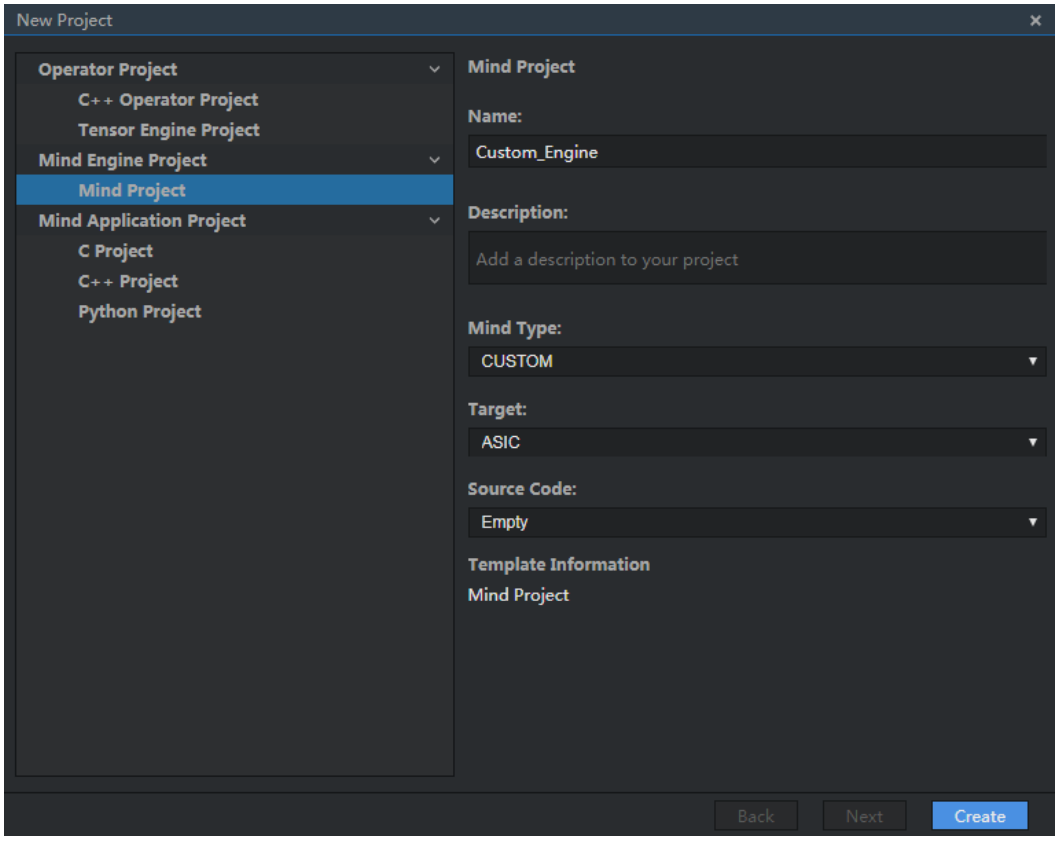
2.2.1 新建自定义工程

步骤1 在Mind Studio登录界面输入用户名、密码进行登录。

步骤2 在Mind Studio操作界面选择“File > New > New Project ”，进入工程创建界面。

选择“Mind Engine Project > Mind Project”，配置相关参数，如**图2-1**所示。

图 2-1 创建自定义 Mind Project 工程



参考说明如表2-1所示。

表 2-1 参数说明

参数	参数说明
Name	工程名称，自行配置。 格式要求为：字符串，不包含空格和中文，空格会自动填充为“-”。
Mind Type	自定义工程请选择“CUSTOM”。 选项说明如下： ● DEFAULT：创建的工程会生成画布，用户可以使用拖拽的方式进行工程编排。 ● CUSTOM：自定义工程，不会生成画布。
Target	工程运行环境，本示例以ASIC和Atlas DK为例分别进行说明。 ● ASIC：连接EVB、PCIe单板，本示例对应工程名为：Custom_Engine ● Atlas DK：连接开发者板，本示例对应工程名为：Custom_Engine_AtlasDK

参数	参数说明
Source Code	此处选择Empty。 代码来源： ● Empty：选择该项，表示工程非外部导入。 ● Local(Web Client)：选择该项，下方出现文件上传输入框。 ● Local(Web Server)：选择该项，则下方出现输入框填写Mind Studio服务器端代码路径。

步骤3 单击“Create”，完成工程创建。

----结束

2.2.2 导入样例工程代码

本示例使用Mind Studio DDK中自带的样例代码进行开发讲解。

步骤1 将DDK Sample中的hiaiengine样例代码拷贝至新建的自定义工程*Custom_Engine*中。

以Mind Studio安装用户进入Mind Studio后台服务器，执行以下命令，拷贝hiaiengine样例工程代码到自定义工程。

```
cp -rf $HOME/tools/che/ddk/ddk/sample/hiaiengine/* /projects/Custom_Engine/
```

说明

- *$HOME/tools*为DDK的默认安装路径。
- Custom_Engine为ASIC场景新建的自定义Engine工程名，若为开发板场景，需更换为Custom_Engine_AtlasDK。
- 用户也可以在新建工程的时候，直接通过“Source Code From”选择“Local(Web Server)”导入hiaiengine样例工程，详细操作请参见《Ascend 310 Mind Studio基本操作》中的“DDK样例工程导入”。

步骤2 hiaiengine样例工程代码导入后，自定义工程代码结构如图2-2所示。

图 2-2 自定义 Engine 工程代码结构

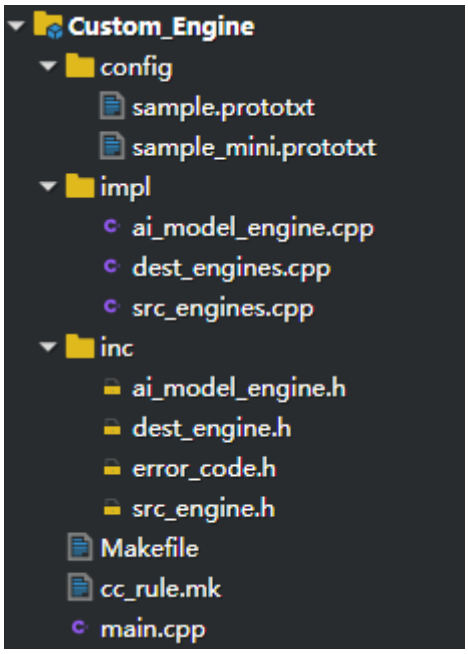


表 2-2 代码结构说明

一级目录	二级目录	说明
config	sample.prototxt	ASIC场景的串接配置文件
	sample_mini.prototxt	Atlas DK场景的串接配置文件
impl	ai_mode_engine.cpp	模型Engine的实现
	dest_engines.cpp	Dest Engine的实现
	src_engines.cpp	Src Engine的实现
inc	ai_model_engine.h	头文件
	dest_engine.h	Dest Engine的头文件
	src_engine.h	Src Engine的头文件
	error_code.h	定义错误类型
Makefile	-	编译脚本
cc_rule.mk	-	Makefile编译脚本所加载的文件
main.cpp	-	主程序

----结束

2.2.3 工程实现

描述自定义Engine工程端到端的实现流程。

2.2.3.1 ASIC 场景

2.2.3.1.1 配置库信息及环境变量

步骤1 以Mind Studio安装用户进入Mind Studio后台/projects/目录，将include目录、lib目录、bin目录映射到该目录下面，命令为：

 # cd /projects

 # ln -s ~/tools/che/ddk/ddk/include/

 # ln -s ~/tools/che/ddk/ddk/uihost/

说明

命令中的*~/tools*为DDK的默认安装路径。

步骤2 设置环境变量。

vim ~/.bashrc

执行如下命令在最后一行添加“/projects/uihost/lib/”的环境变量。

export LD_LIBRARY_PATH="/projects/uihost/lib/"

输入**:wq!**保存退出。

执行如下命令使环境变量生效。

source ~/.bashrc

----结束

2.2.3.1.2 离线模型转化

将外部网络模型转化为华为NPU支持的Davinci模型，有以下两种方式。

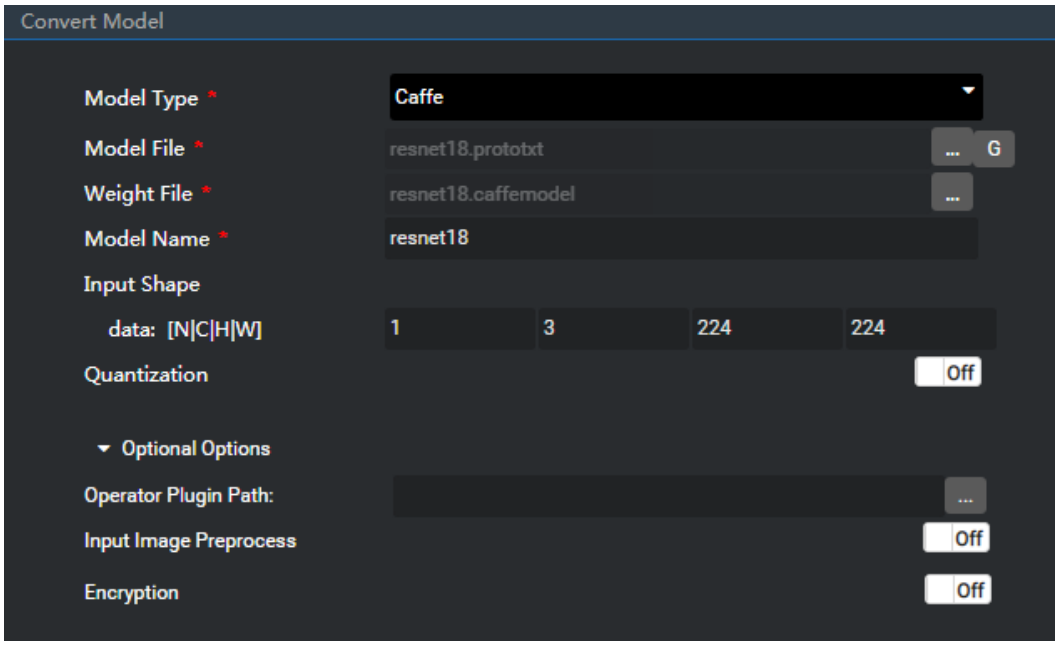
- 通过Mind Studio操作界面提供的离线模型转化功能进行模型转化。
- 通过后台命令进行离线模型转化。

下面分别详细介绍两种方式的操作方法。

通过 Mind Studio 界面操作转化

步骤1 选中自定义Engine工程，右击选择“Convert Model”，或者在菜单栏依次选择“Tools > Convert Model”，进入模型转化界面，如图2-3所示。

图 2-3 离线模型转化



说明

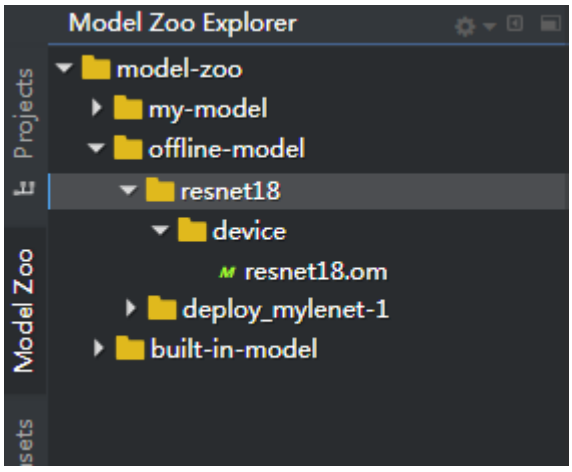
1. 使用自定义工程进行模型转化，需要将Input Image Preprocess图像预处理开关关掉。关于该参数的说明以及Quantization量化参数的说明请参见《Ascend 310 Mind Studio基本操作》中“模型管理>参考”章节。
2. 本文使用resnet18网络模型为例进行讲解，用户自行获取此模型文件。

在“Model File”与“Weight File”中分别选择用户需要进行转化的模型的模型文件及权重文件，单击“ok”进行离线模型转化。

离线模型转化的其他参数的详细解释请参见《Ascend 310 Mind Studio基本操作》中“模型管理”章节。

步骤2 模型转化成功后，在“model-zoo > offline-model”中可以看到转化成功的离线模型，如图2-4所示。

图 2-4 转化好的离线模型



在Model Zoo窗口中选中转化好的离线模型，例如device > resnet18.om（本示例target为ASIC或AtlasDK的工程），右键选择“Copy Path”，在Mind Studio所在后台服务器中粘贴，可以获得当前模型文件所在的路径。

步骤3 将离线模型文件拷贝至当前自定义工程。

```
cp $HOME/tools/che/model-zoo/offline-model/resnet18/device/resnet18.om /
projects/Custom_Engine/
```

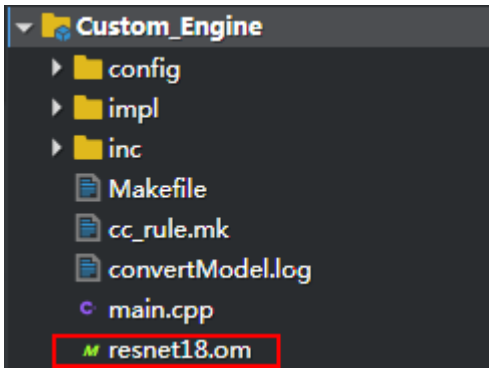
说明

“$HOME/tools/che/model-zoo/offline-model/resnet18/device/resnet18.om”为转换好的模型文件路径，其中$HOME/tools为DDK的默认安装路径。

“Custom_Engine”为自定义工程名。

如图2-5所示。

图 2-5 添加转换好的离线模型文件



----结束

通过后台命令转化

步骤1 将用户需要进行转化的模型的模型文件及权重文件放到工程目录下，例如将resnet18.caffemoel和resnet18.prototxt放到Mind Studio服务器的*/projects/Custom_Engine/*目录下。

步骤2 以Mind Studio安装用户在*Custom_Engine*目录下执行以下命令将外部网络转化为Davinci模型。

```
# /projects/uihost/bin/omg --model=resnet18.prototxt --
weight=resnet18.caffemodel --framework=0  --output=resnet18
```

其中，

- --model项的值为resnet18.prototxt文件的相对路径。
- --weight项的值为resnet18.caffemodel文件的相对路径。
- --framework为原始框架类型。
  - 0：caffe
  - 1：caffe2
  - 2：mxnet
  - 3：tensorflow。
- --output项的值为输出模型文件的名称，用户可自定义。

执行完毕后，会生成后缀名为.om的模型文件，例如resnet18.om文件。

说明

使用自定义工程进行模型转化，需要将AIPP参数关闭，如果您想使用或了解OMG命令中的AIPP以及Quantization参数的使用方法，请参见《Ascend 310 模型加解密使用指导》中的“加密自定义模型 > 通过命令行方式 > 使用omg命令加密”章节。

----结束

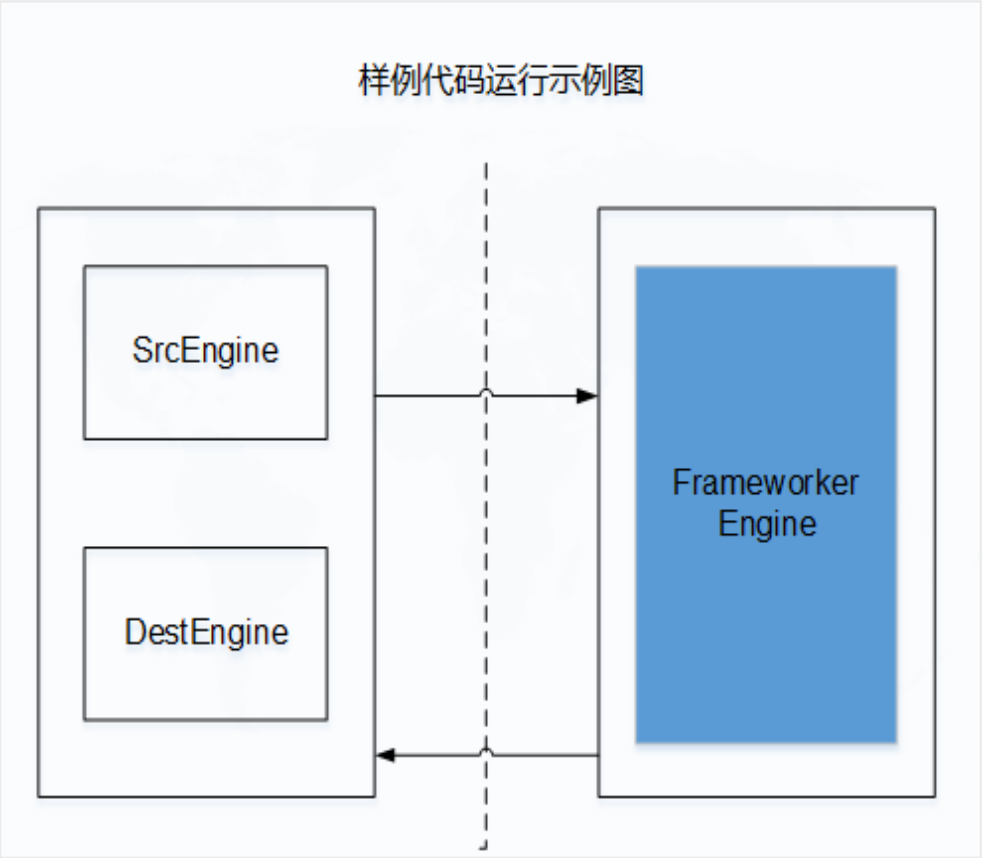
2.2.3.1.3 Engine 实现

本示例通过实现“SrcEngine”、“DestEgnine”、“FrameworkerEngine”这三个Engine来实现resnet-18网络的数据串联。

- SrcEngine：读取发送BGR二进制文件。
- FrameworkerEngine：调用resnet-18网络模型进行计算，并输出结果给DestEngine。
- DestEngine：保存生成的结果文件。

代码运行示例如图2-6所示。

图 2-6 代码运行示例



说明

本节介绍代码实现说明，运行此样例工程不需要进行实现代码的修改。

Engine实现类的Process函数可以将具体需要实现的功能封装在其中。

数据流通过以下方式在Engine间进行传输。

```
std::shared_ptr<Typename> input_arg = std::static_pointer_cast<Typename>(arg0);
//可以通过此函数获取Graph::Send传进来的数据流。
```



```
hiai::Engine::SendData(0, "Typename", std::static_pointer_cast<void>(input_arg));
//可以通过此函数将数据流发送给需要传的Engine的port。
```

详细的接口介绍请参见《HiAI Engine API参考》。

SrcEngine/DestEngine

- SrcEngine 实现样例代码：*Custom_Engine*/impl/src_engines.cpp。

```
HIAI_IMPL_ENGINE_PROCESS("SrcEngine", SrcEngine, SOURCE_ENGINE_INPUT_SIZE)
{
    // 进行读文件操作
    std::shared_ptr<std::string> input_arg =
        std::static_pointer_cast<std::string>(arg0);

    if (nullptr == input_arg)
    {
        HIAI_ENGINE_LOG(this, HIAI_INVALID_INPUT_MSG, "fail to process invalid message");
        return HIAI_INVALID_INPUT_MSG;
    }

    // 直接透传，将输入输出到端口0
    hiai::Engine::SendData(0, "string", std::static_pointer_cast<void>(input_arg));

    return HIAI_OK;
}
```

- DestEngine 实现样例代码：*Custom_Engine*/impl/dest_engines.cpp。

```
HIAI_IMPL_ENGINE_PROCESS("DestEngine", DestEngine, DEST_ENGINE_INPUT_SIZE)
{
    // 进行保存文件操作
    // It is assumed that the first parameter
    // should be shared_ptr<std::string>
    std::shared_ptr<std::string> input_arg =
        std::static_pointer_cast<std::string>(arg0);

    if (nullptr == input_arg)
    {
        HIAI_ENGINE_LOG(this, HIAI_INVALID_INPUT_MSG, "fail to process invalid message");
        return HIAI_INVALID_INPUT_MSG;
    }

    // 直接透传，将输入输出到端口0
    hiai::Engine::SendData(0, "string", std::static_pointer_cast<void>(input_arg));

    return HIAI_OK;
}
```

FrameworkerEngine

实现样例代码：*Custom_Engine*/impl/ai_model_engine.cpp。

```
HIAI_IMPL_ENGINE_PROCESS("FrameworkerEngine", FrameworkerEngine, FRAMEWORK_ENGINE_INPUT_SIZE)
{
    hiai::AIStatus ret = hiai::SUCCESS;
    HIAI_StatusT hiai_ret = HIAI_OK;
    // Framework 处理
    std::shared_ptr<std::string> input_arg =
        std::static_pointer_cast<std::string>(arg0);
    if (nullptr == input_arg)
    {
        return HIAI_INVALID_INPUT_MSG;
    }
    std::cout<<"FrameworkerEngine Process"<<std::endl;

    // ai_model_manager_进行处理
    std::vector<std::shared_ptr<hiai::IAITensor>> input_data_vec;

    uint32_t len = 150528;
```



```
    std::cout << "HIAIAippOp::Go to process" << std::endl;

    std::shared_ptr<hiai::AINeuralNetworkBuffer> neural_buffer =
std::shared_ptr<hiai::AINeuralNetworkBuffer>(new hiai::AINeuralNetworkBuffer());//
std::static_pointer_cast<hiai::AINeuralNetworkBuffer>(input_data);
    neural_buffer->SetBuffer((void*)(input_arg->c_str()), (uint32_t)(len * sizeof(float)));
    std::shared_ptr<hiai::IAITensor> input_data =
std::static_pointer_cast<hiai::IAITensor>(neural_buffer);
    input_data_vec.push_back(input_data);

    // 调用Process
    hiai::AIContext ai_context;
    std::vector<std::shared_ptr<hiai::IAITensor>> output_data_vec;
    ret = ai_model_manager_->CreateOutputTensor(input_data_vec, output_data_vec);
    if (hiai::SUCCESS != ret)
    {
        HIAI_ENGINE_LOG(this, HIAI_AI_MODEL_MANAGER_PROCESS_FAIL, "[DEBUG] fail to process
ai_model");
        return HIAI_AI_MODEL_MANAGER_PROCESS_FAIL;
    }

    ret = ai_model_manager_->Process(ai_context, input_data_vec, output_data_vec, 0);

    if (hiai::SUCCESS != ret)
    {
        HIAI_ENGINE_LOG(this, HIAI_AI_MODEL_MANAGER_PROCESS_FAIL, "[DEBUG] fail to process
ai_model");
        return HIAI_AI_MODEL_MANAGER_PROCESS_FAIL;
    }
    std::cout<<"[DEBUG] output_data_vec size is "<< output_data_vec.size()<<std::endl;
    for (uint32_t index = 0; index < output_data_vec.size(); index++)
    {
        // 将数据发送到尾节点输出
        std::shared_ptr<hiai::AINeuralNetworkBuffer> output_data =
std::static_pointer_cast<hiai::AINeuralNetworkBuffer>(output_data_vec[index]);
        std::shared_ptr<std::string> output_string_ptr = std::shared_ptr<std::string>(new
std::string((char*)output_data->GetBuffer()));

        // 配置EnginePort
        hiai_ret = SendData(0, "string", std::static_pointer_cast<void>(output_string_ptr));
        if (HIAI_OK != hiai_ret)
        {
            HIAI_ENGINE_LOG(this, HIAI_SEND_DATA_FAIL, "fail to send data");
        }
    }
    return HIAI_OK;
}
```

2.2.3.1.4 Engine 串接配置

Engine的相关参数以及Engine的port间连接在Graph中配置，本示例中串接配置文件为config/sample.prototxt，其中FrameworkerEngine中model_path的vaule中的模型名称请更改为**2.2.3.1.2 离线模型转化**中生成的模型文件名称，配置示例如**2.2.3.1.4 Engine串接配置**所示。如果为AtlasDK场景，串接配置文件为config/sample_mini.prototxt。

图 2-7 流程串接文件配置示例

```
graphs {
  graph_id: 100
  priority: 1
  engines {
    id: 1000
    engine_name: "SrcEngine"
    so_name: "./libSrcEngine.so"
    side: HOST
    thread_num: 1
  }
  engines {
    id: 1001
    engine_name: "FrameworkerEngine"
    so_name: "./libFrameworkerEngine.so"
    side: DEVICE
    thread_num: 1
    ai_config{
        items{
            name: "model_path"
            value: "./matrix/resnet18.om"
        }
    }
  }
  engines {
    id: 1002
    engine_name: "DestEngine"
    so_name: "./libDestEngine.so"
    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
  }
}
```

本示例表示本graphs的ID是100，其中engines表示每一个engine的属性配置，connects表示engine的流程串接关系。engines中的side表示此Engine的运行target。

说明

Engine支持跨Graph串接，在connects的target_port_id里配置接收端的Graph ID即可。一般在有多个D芯片，为了使各个模型运行在不同的D芯片上（尽量让相同的模型在一个D芯片上进行推理，这样可以减少不必要的内存消耗）的复杂场景下，可以使用跨Graph串接。

2.2.3.1.5 Engine 串接实现

实现样例代码：*Custom_Engine*/main.cpp。

- 初始化。
```
HIAI_Init();
```
- 通过CreateGraph接口创建Graph。
```
hiai::Graph::CreateGraph(graph_config_proto_file);
```
- 设置回调函数，继承于DataRecvInterface的回调函数可以直接从Engine实现类的Port口获取数据。
```
graph->SetDataRecvFunctor(target_port_config,
    std::shared_ptr<DdkDataRecvInterface>(
    new DdkDataRecvInterface(test_dest_filename)));
```
- 通过Graph::SendData向Engine灌入数据，并启动Engine。
```
graph->SendData(engine_id, "string", std::static_pointer_cast<void>(src_data));
```

2.2.3.2 AtlasDK 场景

开发板场景无需操作**2.2.3.1.1 配置库信息及环境变量**，其余请参见**2.2.3.1.2 离线模型转化~2.2.3.1.5 Engine串接实现**。

AtlasDK场景本示例使用的工程名为：Custom_Engine_AtlasDK

2.2.4 工程编译

2.2.4.1 ASIC 场景

自定义工程编译有以下两种操作方式。

- 无需编写Makefile，通过Mind Studio操作界面提供的编译界面进行编译。
- 编写Makefile，通过后台命令实现工程编译。

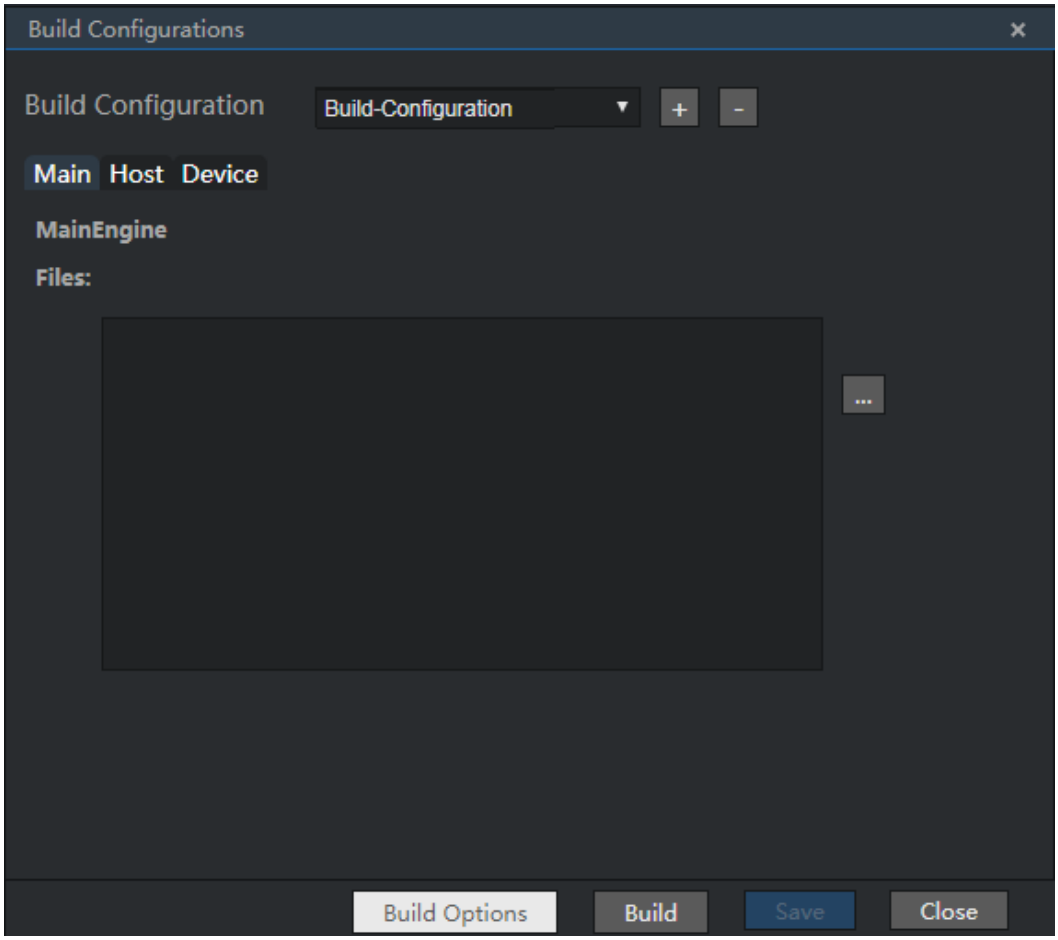
下面分别详细介绍两种方式的操作方法。

通过 Mind Studio 界面进行工程编译

此种编译方式，用户无需编写Makefile文件，详细操作步骤如下。

步骤1 选中要进行配置工程结构的工程，单击依次选择“Build > Edit Build Configuration”，弹出自定义工程的配置页面，如**图2-8**所示。

图 2-8 自定义工程配置界面



步骤2 配置自定义工程。

自定义配置工程结构分为**Main**、**Host**、**Device**三部分。

每个.so文件需要根据sample.prototxt文件中的每个Engine的side值在相应页签配置。

Main、Host与Device的参数描述如表2-3所示。

表 2-3 配置说明

配置项	说明
Main	主函数侧的实现文件，选择执行文件的cpp文件，即main.cpp文件。 保存配置后生成CMakeLists.txt文件（路径：工程根目录下build文件夹）。 编译成功生成main文件（路径：out文件夹）。

配置项	说明
Host	添加需要运行在Host侧的Engine，并填写Engine的名称，该名称不能与工程名同名。(可单击HOST_PARAM后面的添加Engine，也可以单击相应Engine右侧的，删除Engine。) 各个hostEngine保存配置生成与Engine Name同名的文件夹，包含CMakeLists.txt文件（路径：工程根目录下build文件夹）。 编译成功生成与Engine Name同名的.so文件（路径：out文件夹）。
Device	添加运行在Device侧的Engine，并填写Engine的名称，该名称不能与工程名同名。(可单击Device_PARAM后面的添加Engine，也可以单击相应Engine右侧的，删除Engine。) 各个DeviceEngine保存配置生成与Engine Name同名的文件夹，包含CMakeLists.txt文件（路径：工程根目录下build文件夹）。 编译成功生成与Engine Name同名的.so文件（路径：out文件夹）。

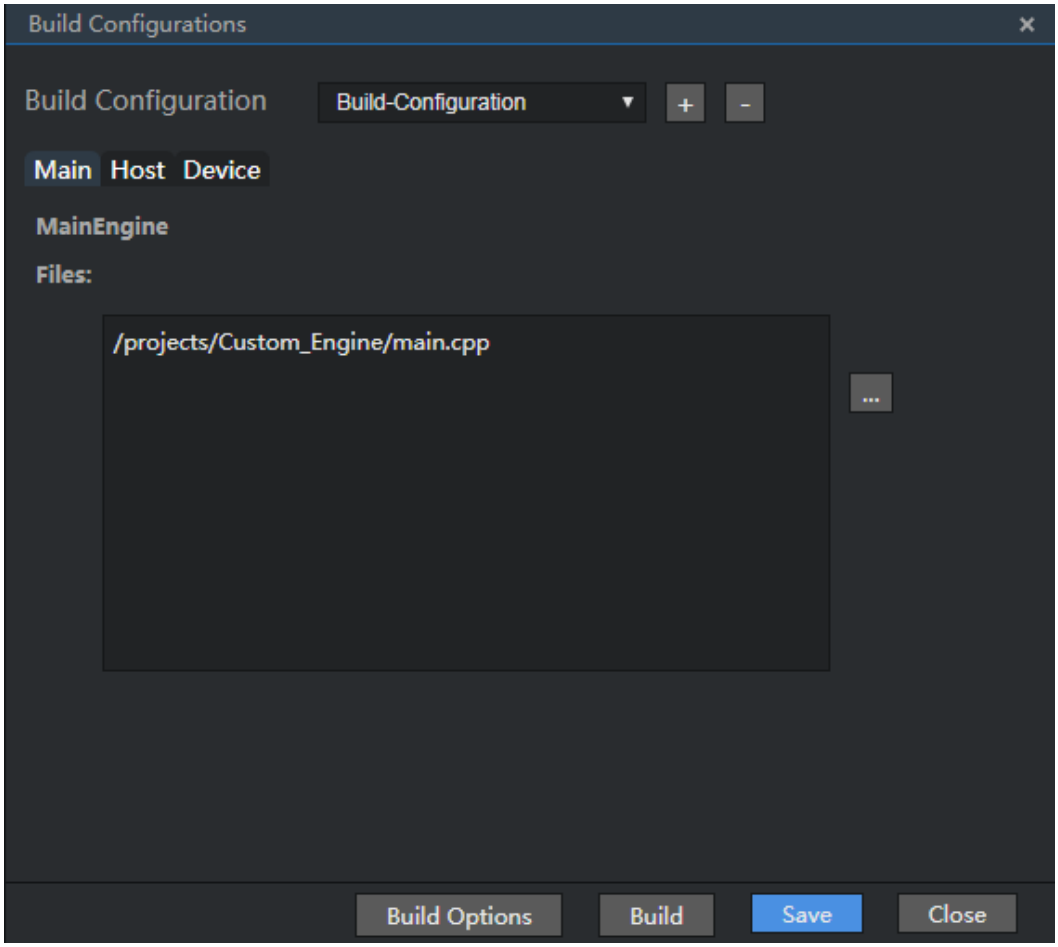
说明

.so文件需要签名，防止文件被篡改，签名方法：

1. 由openssl工具中的RSA_generate_key接口生成公钥pub.pem+私钥pri.pem，密钥长度建议2048bit以上。
2. 通过私钥，利用SHA256算法对.so文件进行签名，生成.so.signature签名文件，与原.so文件放在同一目录。
3. 公钥需要由SetPublicKeyForSignatureEncryption接口传给HiAI Engine。接口信息请参见《HiAIEngine API参考》。

三个页签的配置依据是根据Engine的so文件需要放置在哪一侧运行。由于样例是ASIC工程，样例中SrcEngine和DestEngine在host侧运行；FrameworkerEngine在Device侧运行，编译配置示例下图所示。

图 2-9 Main 页签配置示例

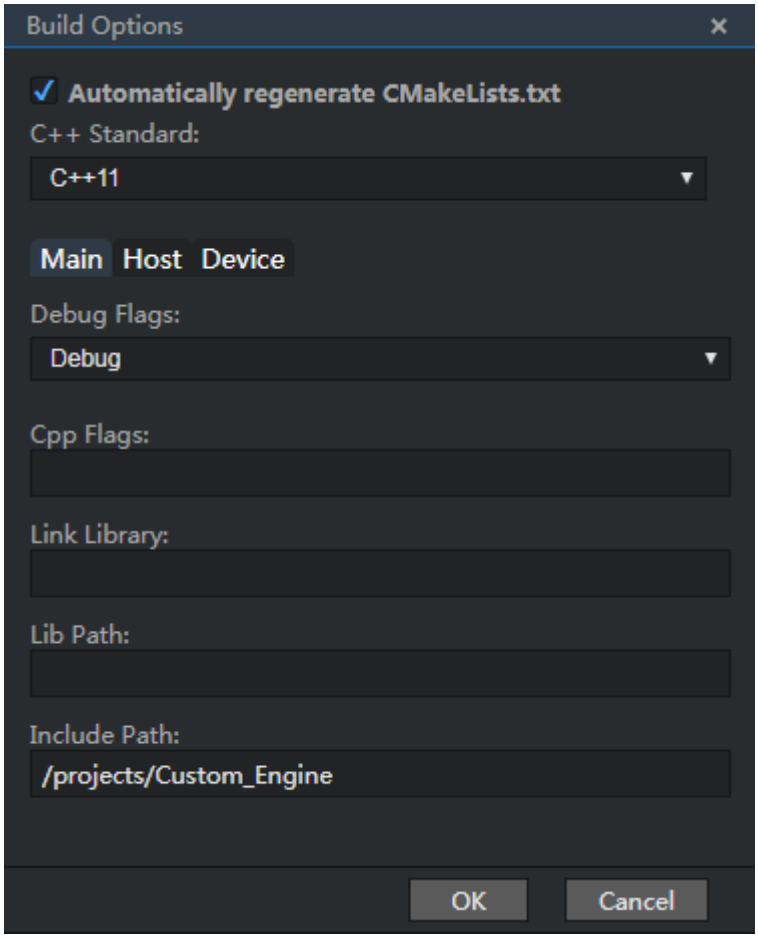


单击图2-9中的“Build Options”，弹出如图2-10所示界面。

说明

- 若不单击“Build Options”或不配置“Build Options”配置窗口中的配置项，编译时将使用默认配置值进行编译。
- “Build Options”中，Main和Device页签都需要配置Include Path，此处都配置为：/projects/Custom_Engine。

图 2-10 Build Options 配置界面



Build Options配置页面中各个页签的配置参数如表2-4所示。

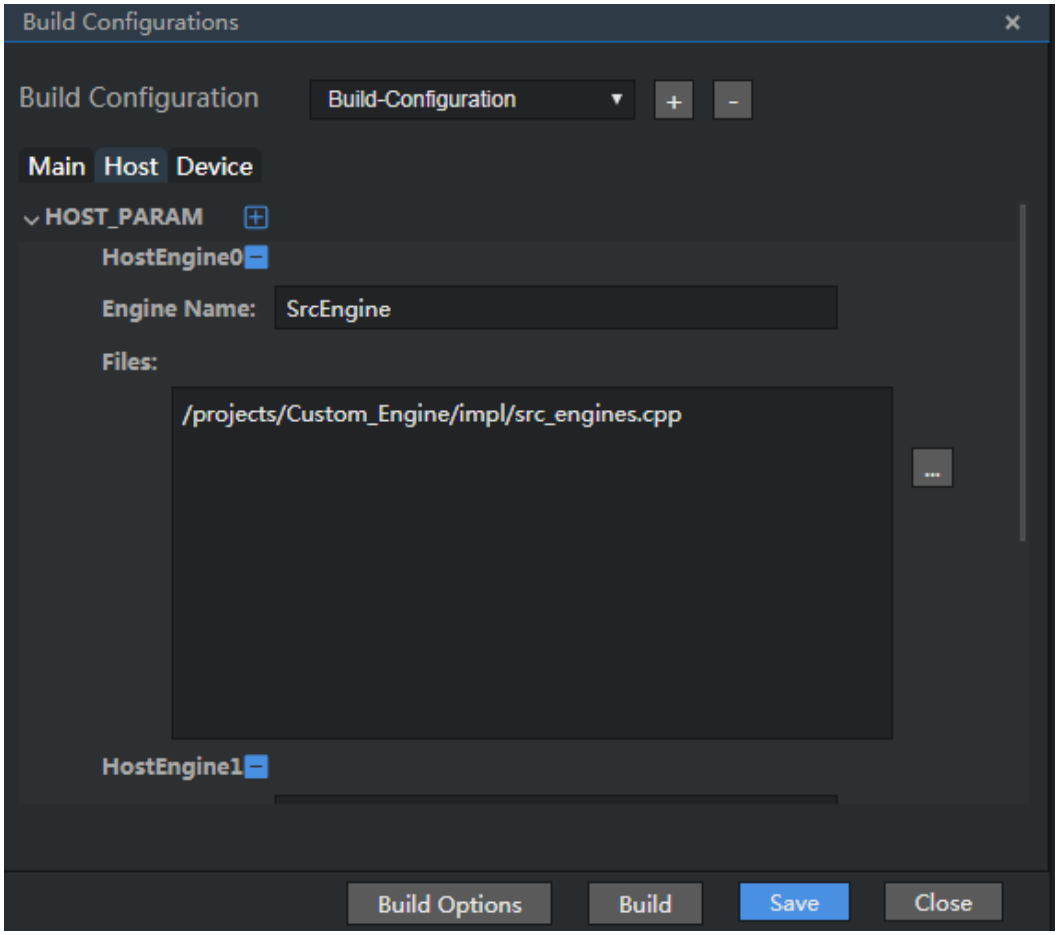
表 2-4 Build Options 配置参数说明

Build Options配置项	说明
Automatically regenerate CMakeLists.txt	默认为勾选状态。 ● 若为勾选状态，每次保存配置时会自动覆盖CMakeLists.txt文件。（如果用户对Makefile文件有修改，也会覆盖。） ● 若为未勾选状态，保存配置时不自动覆盖CMakeLists.txt文件。
C++ Standard	C++标准，分为如下两种。 ● C++ 11 ● C++ 98 默认为C++ 11。

Build Options配置项	说明
Debug Flags	编译模式，分为如下两种。 ● Debug ● Release 默认Debug模式。
Cpp Flags	编译选项，用户配置。
Link Library	引用的链接库，用户可配置。 链接的动态库，对应编译命令中-l参数的值。
Lib Path	链接库路径，目前使用缺省路径，用户可增加配置。 动态库查找路径，对应编译命令中-L参数的值。
Include Path	头文件路径，目前使用缺省路径，用户可增加配置。 头文件查找路径，对应编译命令中-I参数的值。

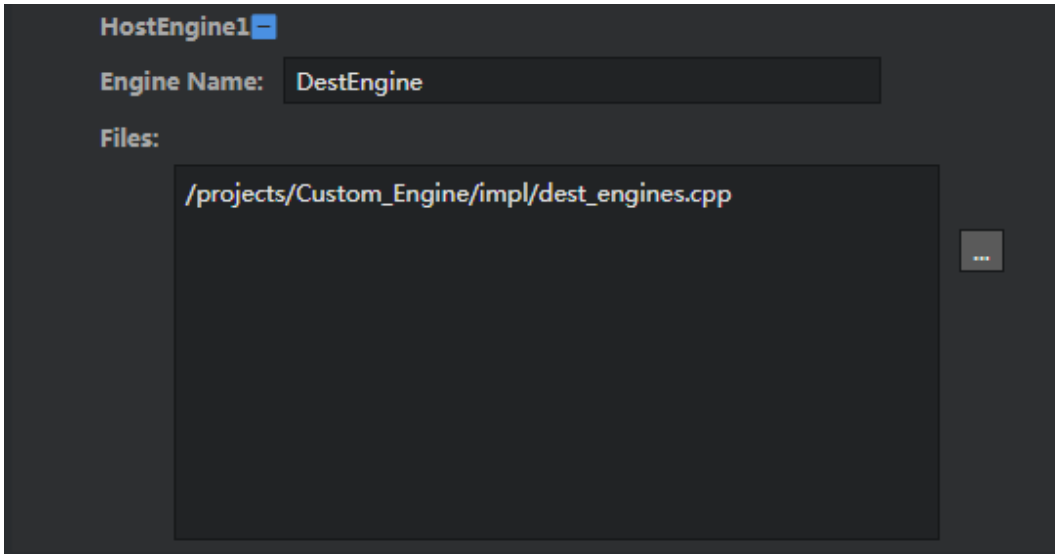
Host页签配置示例如图2-11所示。

图 2-11 Host 页签配置__SrcEngine



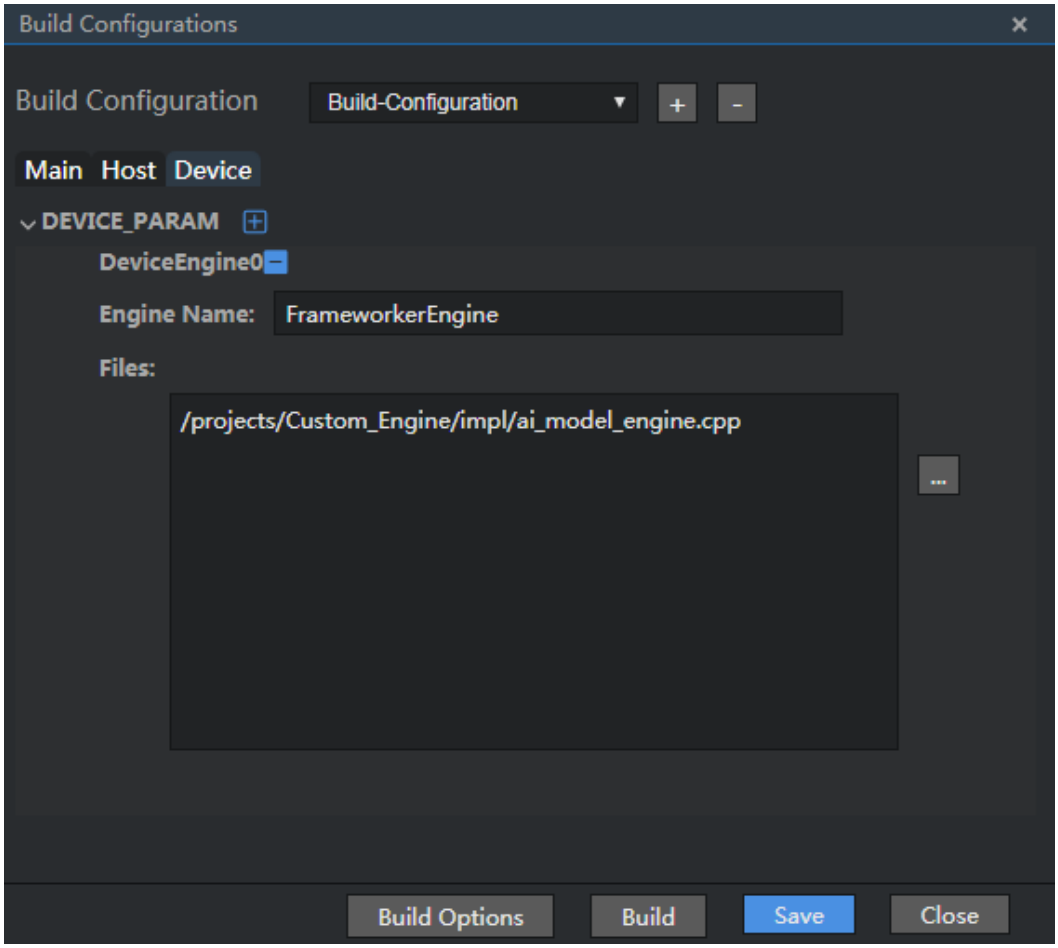
单击HOST_PARAM参数后面的  增加DestEngine配置，如图2-12所示。

图 2-12 host 页签配置_DestEngine



Device页签配置示例如图2-13所示。

图 2-13 Device 页签配置

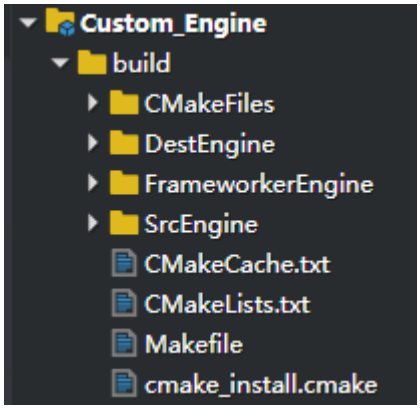


步骤3 配置完成后，单击“Save”，保存工程结构的配置，单击“Build”，或者在菜单栏选择“Build > Build > 工程名称”进行编译。

保存工程结构配置分两种情况。

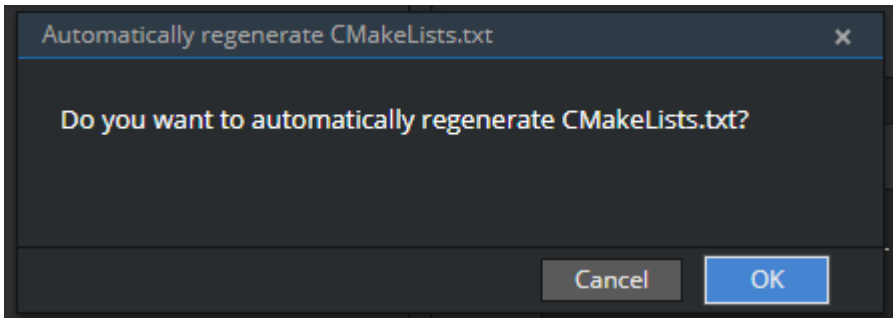
- 若“Build Options”配置窗口中的“Automatically regenerate CMakeLists.txt”配置为勾选状态，界面下方的“build”窗口内会显示CMakeLists.txt文件生成信息，工程根目录下自动生成保存CMakeLists.txt文件的build文件夹，如图2-14所示。

  图 2-14 工程目录示例

- 若“Build Options”配置窗口中的“Automatically regenerate CMakeLists.txt”配置为未勾选状态。
  - 当工程根目录下不存在build文件夹或工程根目录下的build文件夹不包含CMakeLists.txt文件，则会自动生成包含CMakeLists.txt文件的build文件夹，如图2-14所示。
  - 当工程根目录下存在包含CMakeLists.txt文件的build文件夹，将会弹出Automatically regenerate CMakeLists.txt窗口，如图2-15所示。

    图 2-15 弹出窗口

    若单击“OK”，则覆盖原有CMakeLists.txt文件。

    若单击“Cancel”，则使用当前工程目录下存在的CMakeLists.txt文件进行编译，不再重新生成CMakeLists.txt文件。

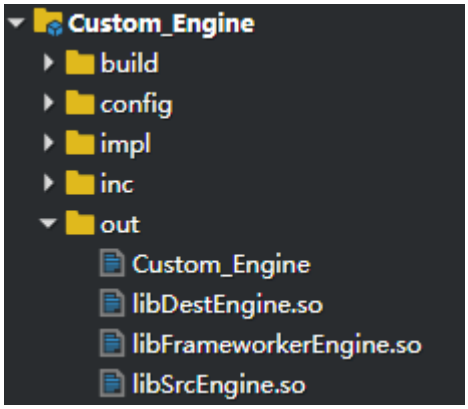
工程编译按成后，build窗口编译结果如图2-16所示。

图 2-16 编译输出窗口

```
CMakeLists.txt generated to /projects/Custom_Engine/build/
CMakeLists.txt generated to /projects/Custom_Engine/build/SrcEngine
CMakeLists.txt generated to /projects/Custom_Engine/build/DestEngine
CMakeLists.txt generated to /projects/Custom_Engine/build/FrameworkerEngine
CMakeLists.txt generated
-- Configuring done
-- Generating done
-- Build files have been written to: /projects/Custom_Engine/build
Scanning dependencies of target Custom_Engine
[ 12%] Linking CXX executable /projects/Custom_Engine/out/Custom_Engine
[ 25%] Built target Custom_Engine
[ 50%] Built target SrcEngine
[ 75%] Built target DestEngine
[100%] Built target FrameworkerEngine
```

编译成功后，会在工程根目录下生成out文件夹，如图2-17所示。

图 2-17 编译后 out 文件



out文件夹中的“Custom_Engine”文件为Main Engine生成的可执行文件。

说明

1. 工程中未被Main Engine、Host Engines、Device Engines选中路径的文件夹以及文件在编译时会被忽略。
2. 如果对未配置过的工程进行一键式编译，若工程根目录下存在包含CMakeLists.txt文件的build文件夹，该工程会使用build文件夹进行编译，否则会使用选中的配置对该工程进行编译。
3. 重命名后的工程，需修改之前的Main Engine/Host Engines/Device Engines中Files配置项的路径值。若运行之前的配置，生成器会根据原来的Files内的路径值生成原工程名的无效工程。

----结束

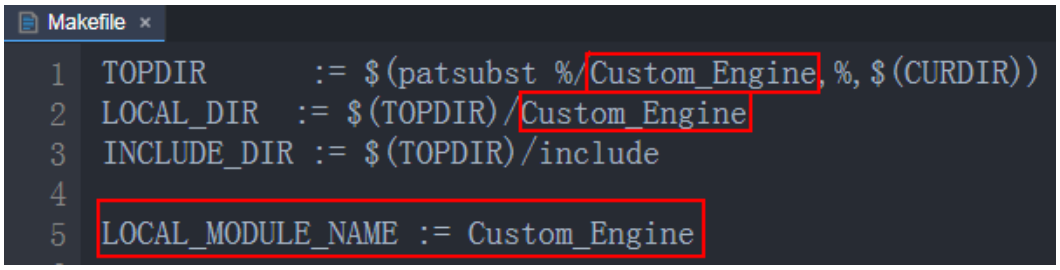
通过后台命令进行工程编译

步骤1 修改Makefile文件。

样例工程根目录下已提供Makefile文件，用户可根据此示例文件进行修改，Makefile文件路径，*Custom_Engine*/Makefile。

- 将“TOPDIR”与“LOCAL_DIR”中的“sample/hiaiengine”字段修改成工程名（工程名为创建工程时自定义，例如*Custom_Engine*），如图2-18所示。

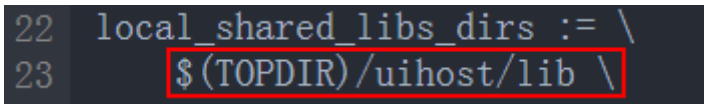
图 2-18 修改 Makefile 文件工程名



“LOCAL_MODULE_NAME”为生成的可执行文件的名称，用户可以自定义，例如可修改为*Custom_Engine*。

- 将local_shared_libs_dirs的值修改为实际链接的库/uihost/lib（**2.2.3.1.1 配置库信息及环境变量**中所配置的库），如**图2-19**所示。

图 2-19 修改 Makefile 文件中 lib 库路径



Makefile文件解析如下所示。

```
TOPDIR      := $(patsubst %/Custom_Engine,%,$(CURDIR))
LOCAL_DIR  := $(TOPDIR)/Custom_Engine
INCLUDE_DIR := $(TOPDIR)/include
LOCAL_MODULE_NAME := Custom_Engine
CC_FLAGS := -std=c++11
local_src_files := \  #编译相关的源文件
    $(LOCAL_DIR)/main.cpp \
    $(LOCAL_DIR)/impl/impl_engines.cpp \
    $(LOCAL_DIR)/impl/ai_model_engine.cpp \
local_inc_dirs := \   #编译相关的头文件
    $(LOCAL_DIR) \
    $(INCLUDE_DIR)/inc \
    $(INCLUDE_DIR)/third_party/protobuf/include \
    $(INCLUDE_DIR)/third_party/cereal/include \
    $(INCLUDE_DIR)/libc_sec/include
local_shared_libs_dirs := \ #需要链接的lib路径，对应界面编译菜单中的Lib Path
    $(TOPDIR)/uihost/lib \
local_shared_libs := \  #需要链接的动态库，对应界面编译菜单中的Link Library
    hiai_server_cmodel \
    hiai_common \
    matrix_cmodel \
    pthread \
    protobuf \
include ./cc_rule.mk  #包含其他makefile
```

cc_rule.mk文件解析如下所示。

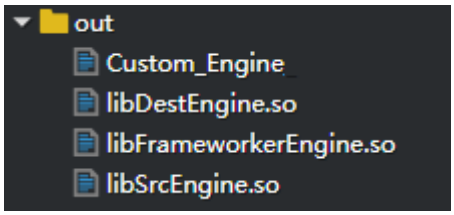
```
Q := @
exist = $(shell if [ -f $(../../toolchains/aarch64-linux-gcc6.3/bin/aarch64-linux-gnu-g+
+) ]; then echo "exist"; else echo "notexist"; fi;)
ifeq ($(exist), exist)
#exit 0
CPP := g++
else
exit 0
endif
FULL_SRC_FILES        := $(local_src_files)
FULL_INC_DIRS         := $(foreach inc_dir, $(local_inc_dirs), -I$(inc_dir))
SHARED_LIBRARIES      := $(foreach shared_lib, $(local_shared_libs), -l$(shared_lib))
SHARED_LIBRARIES_DIRS := $(foreach shared_lib_dir, $(local_shared_libs_dirs), -L$
```



```
(shared_lib_dir) -Wl,-rpath-link,$(shared_lib_dir))
LOCAL_OBJ_PATH        := $(LOCAL_DIR)/out  #定义编译输出的路径
LOCAL_LIBRARY         := $(LOCAL_OBJ_PATH)/$(LOCAL_MODULE_NAME)
FULL_C_SRCS           := $(filter %.c,$(FULL_SRC_FILES))
FULL_C_OBJS           := $(patsubst $(LOCAL_DIR)/%.c,$(LOCAL_OBJ_PATH)/%.o, $(FULL_C_SRCS))
FULL_CPP_SRCS         := $(filter %.cpp,$(FULL_SRC_FILES))
FULL_CPP_OBJS         := $(patsubst $(LOCAL_DIR)/%.cpp,$(LOCAL_OBJ_PATH)/%.o, $
(FULL_CPP_SRCS))
all: do_pre_build do_build
do_pre_build:
 $(Q)echo - do [$@]
 $(Q)mkdir -p $(LOCAL_OBJ_PATH)
do_build: $(LOCAL_LIBRARY) | do_pre_build
 $(Q)echo - do [$@]
# $(Q)rm -rf $(LOCAL_OBJ_PATH)
$(LOCAL_LIBRARY): $(FULL_C_OBJS) $(FULL_CPP_OBJS) | do_pre_build
 $(Q)echo [LD] $@
 $(Q)$(CPP) $(CC_FLAGS) -o $(LOCAL_LIBRARY) $(FULL_C_OBJS) $(FULL_CPP_OBJS) -Wl,--whole-
archive  -Wl,--no-whole-archive -Wl,--start-group  -Wl,--end-group $(SHARED_LIBRARIES_DIRS) $
(SHARED_LIBRARIES)
$(FULL_C_OBJS): $(LOCAL_OBJ_PATH)/%.o : $(LOCAL_DIR)/%.c  | do_pre_build
 $(Q)echo [CC] $@
 $(Q)mkdir -p $(dir $@)
 $(Q)$(CPP) $(CC_FLAGS)  $(FULL_INC_DIRS) -c $< -o $@
$(FULL_CPP_OBJS): $(LOCAL_OBJ_PATH)/%.o : $(LOCAL_DIR)/%.cpp  | do_pre_build
 $(Q)echo [CC] $@
 $(Q)mkdir -p $(dir $@)
 $(Q)$(CPP) $(CC_FLAGS) $(FULL_INC_DIRS) -c $< -o $@
```

步骤2 进入“/projects/Custom_Engine”目录，进行Make，执行命令**make -j**，编译成功后，会在“Custom_Engine/out”目录下生成此工程的可执行程序，例如图2-20所示。

图 2-20 out 文件



----结束

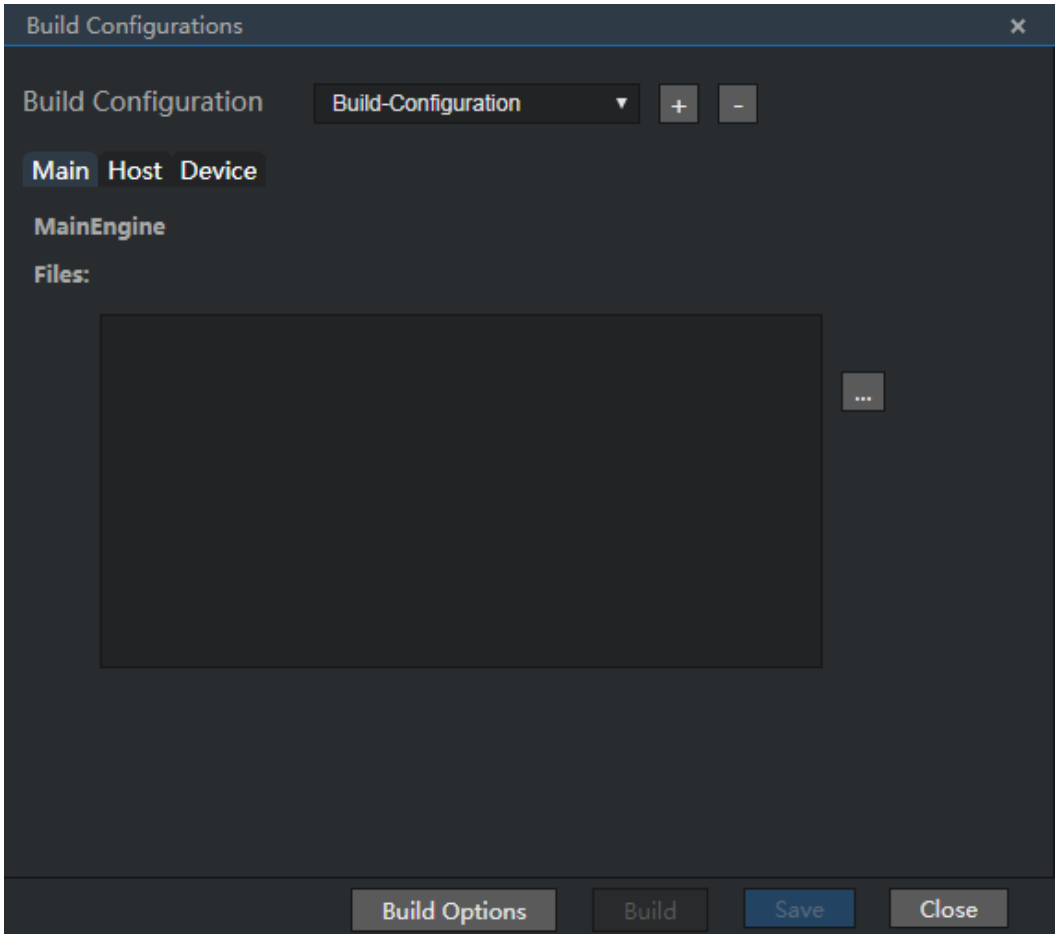
2.2.4.2 AtlasDK 场景

通过 Mind Studio 界面进行工程编译

此种编译方式，用户无需编写Makefile文件，详细操作步骤如下。

步骤1 选中要进行配置工程结构的工程，单击依次选择“Build > Edit Build Configuration”，弹出自定义工程的配置页面，如图2-21所示。

图 2-21 自定义工程配置界面



步骤2 配置自定义工程。

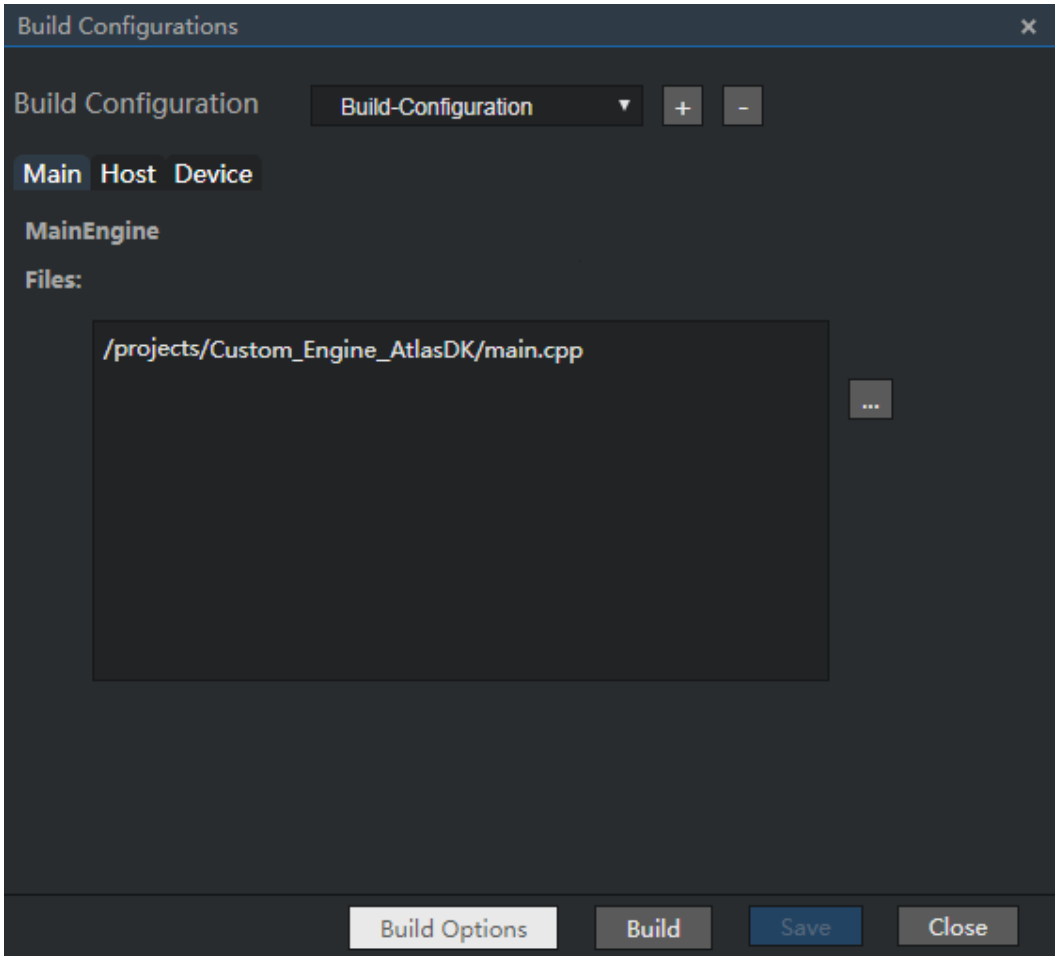
自定义配置工程结构分为**Main**、**Host**、**Device**三部分。

每个.so文件需要根据sample.prototxt文件中的每个Engine的side值在相应页签配置。

Main、Host与Device的参数描述请参见**通过Mind Studio界面进行工程编译**中的**步骤2**。

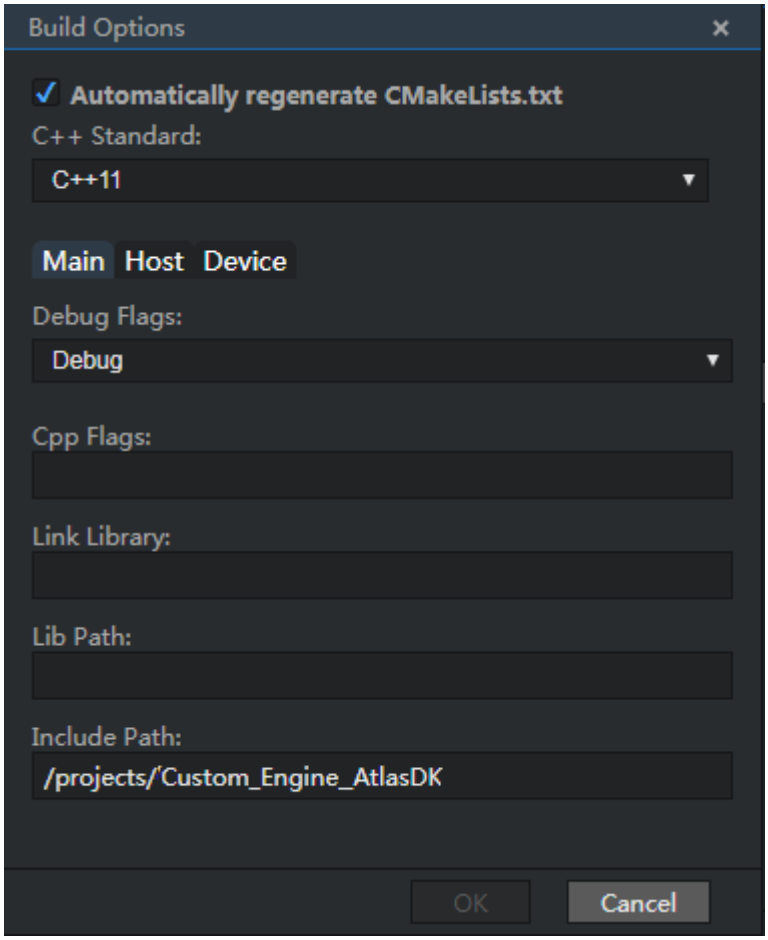
本示例工程只需要配置Main和Device两个页签，Main页签配置如**图2-22**所示。

图 2-22 Main 页签配置示例



单击图2-22中的“Build Options”，弹出如图2-23所示界面。

图 2-23 Build Options 配置页面 1

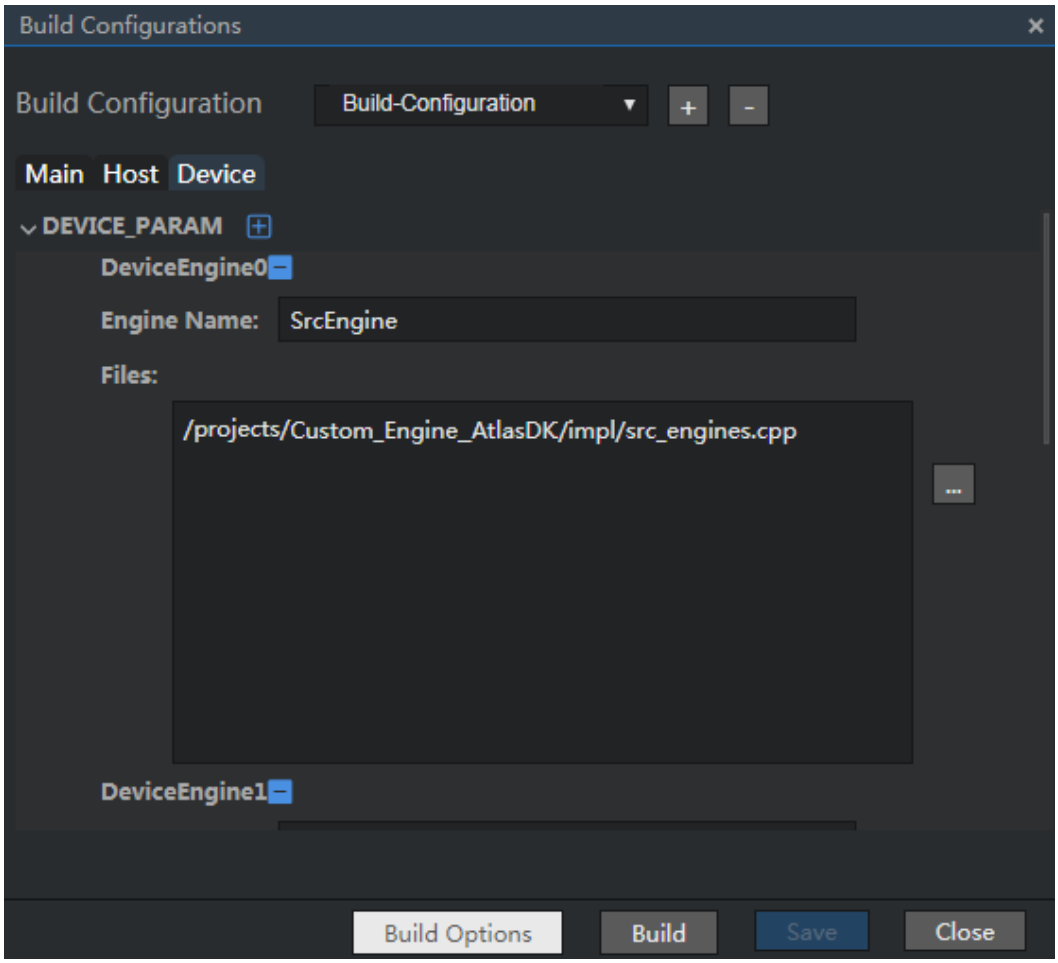


说明

- 若不单击“Build Options”或不配置“Build Options”配置窗口中的配置项，编译时将使用默认配置值进行编译。
- “Build Options”中，Main和Device页签都需要配置Include Path，此处都配置为：/projects/Custom_Engine_AtlasDK。

Device页签配置示例如图2-24所示。

图 2-24 Device 页签配置



单击“DEVICE_PARAM”参数后面的  增加engine信息，分别配置FrameworkerEngine以及DestEngine信息，如图2-25、图2-26所示：

图 2-25 FrameworkerEngine 配置信息

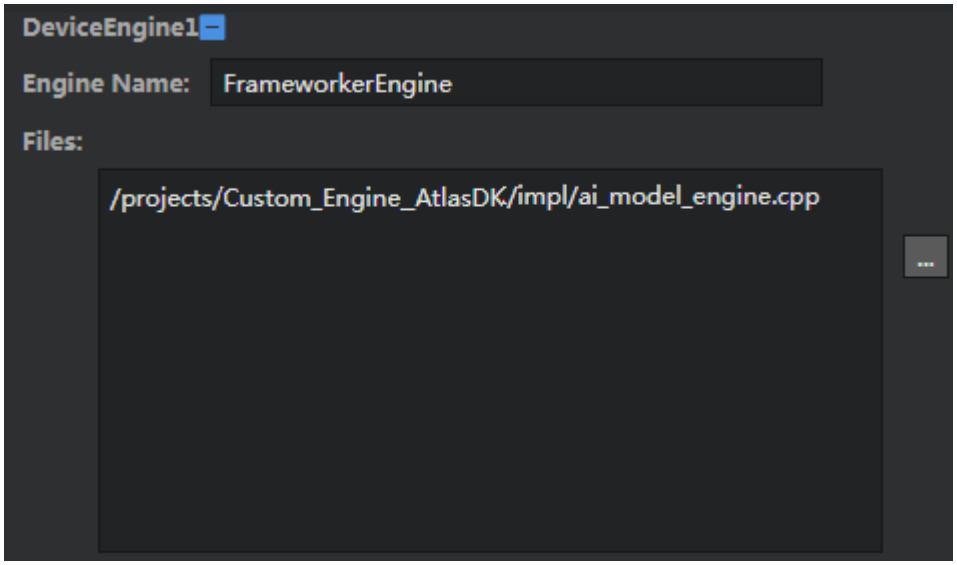
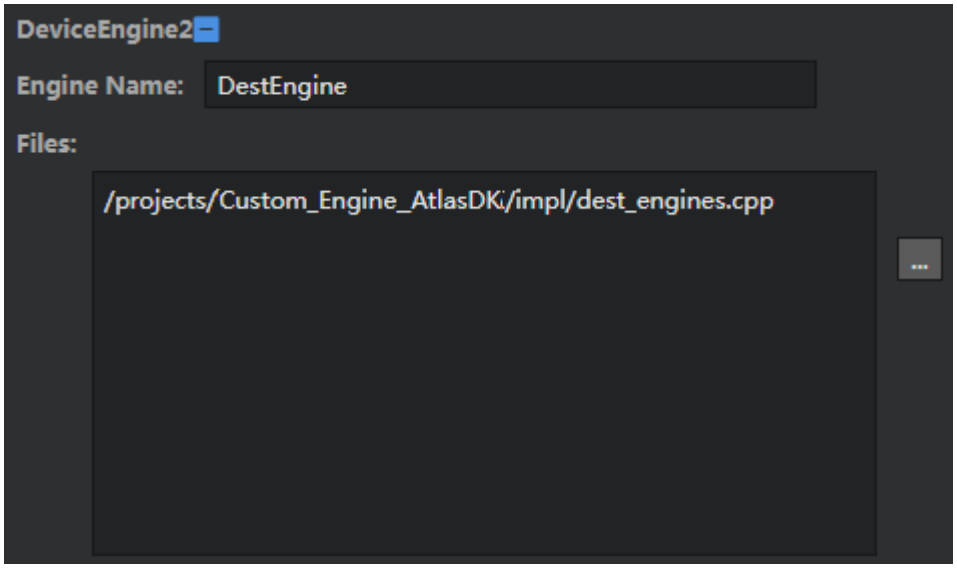


图 2-26 DestEngine 配置信息

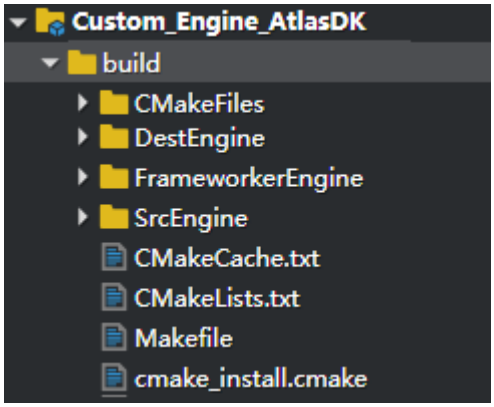


步骤3 配置完成后，单击“Save”，保存工程结构的配置，单击“Build”，或者在菜单栏选择“Build > Build > 工程名称”进行编译。

保存工程结构配置分两种情况。

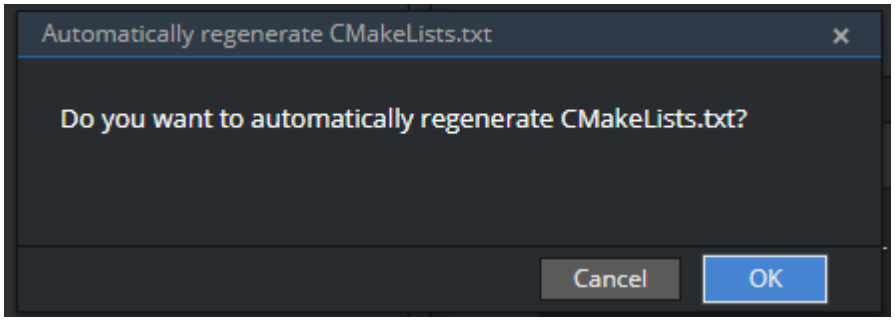
- 若“Build Options”配置窗口中的“Automatically regenerate CMakeLists.txt”配置为勾选状态，界面下方的“build”窗口内会显示CMakeLists.txt文件生成信息，工程根目录下自动生成保存CMakeLists.txt文件的build文件夹，如图2-27所示。

  图 2-27 工程目录示例

- 若“Build Options”配置窗口中的“Automatically regenerate CMakeLists.txt”配置为未勾选状态。
  - 当工程根目录下不存在build文件夹或工程根目录下的build文件夹不包含CMakeLists.txt文件，则会自动生成包含CMakeLists.txt文件的build文件夹，如图2-27所示。
  - 当工程根目录下存在包含CMakeLists.txt文件的build文件夹，将会弹出Automatically regenerate CMakeLists.txt窗口，如图2-28所示。

图 2-28 弹出窗口

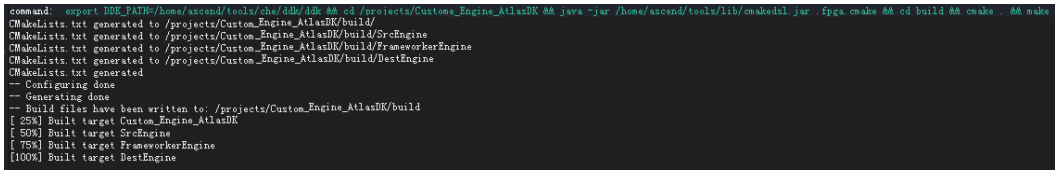


若单击“OK”，则覆盖原有CMakeLists.txt文件。

若单击“Cancel”，则使用当前工程目录下存在的CMakeLists.txt文件进行编译，不再重新生成CMakeLists.txt文件。

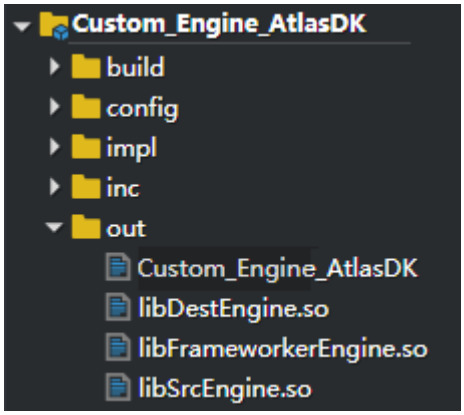
工程编译按成后，build窗口编译结果如图2-29所示。

图 2-29 编译输出窗口



编译成功后，会在工程根目录下生成out文件夹，如图2-30所示。

图 2-30 编译后 out 文件



out文件夹中的“Custom_Engine_AtlasDK”为Main Engine生成的可执行文件。

说明

1. 工程中未被Main Engine、Host Engines、Device Engines选中路径的文件夹以及文件在编译时会被忽略。
2. 如果对未配置过的工程进行一键式编译，若工程根目录下存在包含CMakeLists.txt文件的build文件夹，该工程会使用build文件夹进行编译，否则会使用选中的配置对该工程进行编译。
3. 重命名后的工程，需修改之前的Main Engine/Host Engines/Device Engines中Files配置项的路径值。若运行之前的配置，生成器会根据原来的Files内的路径值生成原工程名的无效工程。

----结束

2.2.5 工程执行

2.2.5.1 ASIC 场景

步骤1 构造输入数据。

main.cpp函数中所需输入数据如图2-31所示。

图 2-31 main 函数输入数据

```
19 static const std::string test_src_file = "./matrix/test.yuv";
20 static const std::string test_dest_filename = "./matrix/matrix_dvpp_framework_test_result";
21 static const std::string graph_config_proto_file = "./matrix/config/sample.prototxt";
```

ai_model_engine.cpp所需测试输入数据如图2-32所示。

图 2-32 网络模型 engine 输入数据

```
Makefile   main.cpp   ai_model_engine.cpp ×
 97     uint32_t len = 150528;
 98     uint32_t size = 0;
 99     std::string input_file = "./matrix/input_zebra.bin";
100     float* bufData = (float*)ReadBinFile(input_file.c_str(), &size);
101     std::cout << "HIAIAippOp::Go to process" << std::endl;
```

说明

输入数据路径为编译生成的可执行文件的相对路径。

1. 以Mind Studio安装用户在执行程序目录下创建matrix目录，执行以下命令。

   #cd /projects/Custom_Engine/out/

   # mkdir matrix

2. 拷贝测试文件test.yuv文件到matrix目录。

   说明

   test.yuv为测试文件，本用例中只作为透传，任意内容都可以。

   例如，可以直接在matrix目录下执行**touch test.yuv**命令创建一空测试文件。

3. 将config文件夹拷贝到matrix目录下。

   #cp -r /projects/Custom_Engine/config /projects/Custom_Engine/out/matrix/

4. 拷贝测试数据bin文件到matrix目录。

   bin文件路径为“$HOME/tools/che/ddk > ddk > sample > classify_net_asic > test_data > resnet-18 > data > src_test.bin”，手动复制到matrix目录，本用例中需重命名为input_zebra.bin。

   #cp $HOME/tools/che/ddk/ddk/sample/classify_net_asic/test_data/resnet-18/data/src_test.bin /projects/Custom_Engine/out/matrix/

   说明

   *$HOME/tools*为DDK的默认安装路径。

   #cd /projects/Custom_Engine/out/matrix/

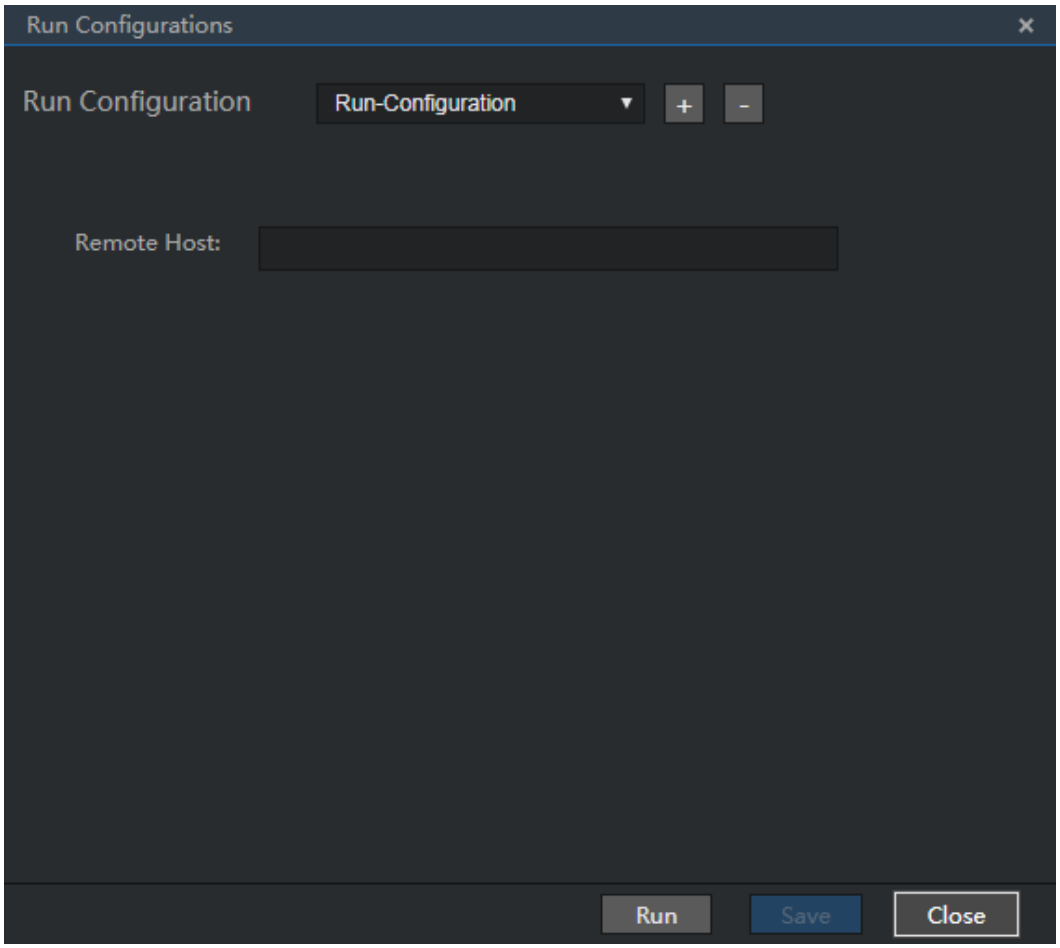
   #mv src_test.bin input_zebra.bin

5. 拷贝网络模型到matrix目录下，并修改graph配置文件中的网络模型名称。
   执行以下命令拷贝2.2.3.1.2 离线模型转化章节生成的模型文件resnet18.om。
   #cp /projects/Custom_Engine/resnet18.om /projects/Custom_Engine/out/matrix/

步骤2 工程执行。

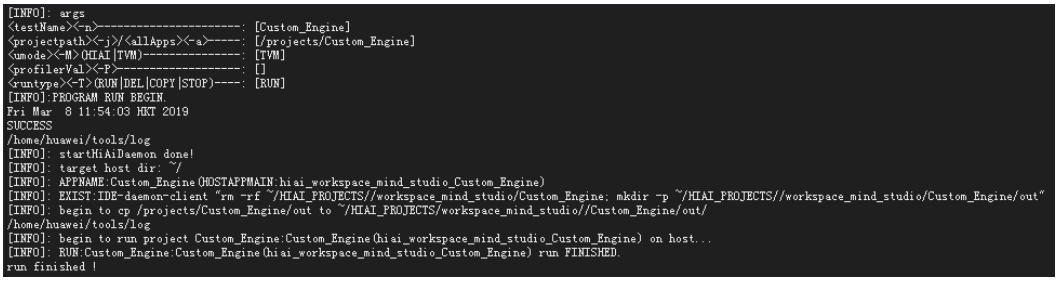
- 通过Mind Studio界面执行。
  选择工程，在菜单栏依次选择“Run > Edit Run Configuration”，弹出运行配置界面，如图2-33所示，输入Remote Host，然后单击“Run”运行工程。

图 2-33 运行配置界面



运行成功后，输出如图2-34所示信息，并会在host侧服务器“/home/HwHiAiUser/HIAI_PROJECTS/workspace_mind_studio/Custom_Engine/out/matrix”目录下生成“matrix_dvpp_framework_test_result”检测结果文件。

图 2-34 运行输出



- 通过后台命令执行可执行文件。

  登录host侧服务器，本例中Custom_Engine工程所在host侧服务器地址为：“/home/HwHiAiUser/HIAI_PROJECTS/workspace_mind_studio/Custom_Engine”

  在out目录下执行可执行程序**./hiai_workspace_mind_studio_Custom_Engine**

  运行成功后，会在out/matrix目录下生成matrix_dvpp_framework_test_result检测的结果文件。

  说明

  – 以上服务器地址以及运行的可执行程序名都是示例，请以实际为准。

  – 如果执行过程中提示如下错误。

  ```
  ./hiaiengine: error while loading shared libraries: libhiai_server_cmodel.so: cannot open shared object file: No such file or directory
  ```

  请执行如下命令将libhiai_server_cmodel.so所在目录加入环境变量。

  export LD_LIBRARY_PATH="/projects/uihost/lib/"

----结束

2.2.5.2 AtlasDK 场景

1. 以HwHiAiUser用户ssh登录到开发板。

   ssh HwHiAiUser@*192.168.1.2*

   su HwHiAiUser

   说明

   192.168.1.2为开发板的IP地址，请根据实际情况修改。

2. 在HwHiAiUser用户家目录下创建matrix目录，此处以在“/home/HwHiAiUser/”*“temp”*目录创建matrix为例，执行以下命令：

   # **cd /home/*HwHiAiUser***

   # **mkdir *temp***

   # **cd temp**

   # **mkdir matrix**

3. 拷贝测试文件test.yuv文件到开发板matrix目录。

   说明

   test.yuv为测试文件，本用例中只作为透传，任意内容都可以。

   例如，可以直接在matrix目录下执行**touch test.yuv**命令创建一空测试文件。

4. 将Custom_Engine_AtlasDK工程下的config文件拷贝到开发板的matrix目录下，并将config文件中sample_mini.prototxt重命名为sample.prototxt。

   退出开发板到Mind Studio所在服务器，执行如下命令：

#scp -r /projects/Custom_Engine_AtlasDK/config/sample_mini.prototxt HwHiAiUser@192.168.1.2:/home/HwHiAiUser/*temp*/matrix/config/sample.prototxt

5. 拷贝编译后的文件到开发板的*temp*目录。

   在Mind Studio所在服务器，执行如下命令：

   #scp -r /projects/Custom_Engine_AtlasDK/out HwHiAiUser@192.168.1.2:/home/HwHiAiUser/*temp*

6. 拷贝测试数据bin文件到开发板matrix目录。

   在Mind Studio所在服务器，执行如下命令：

   #scp $HOME/tools/che/ddk/ddk/sample/classify_net_asic/test_data/resnet-18/data/src_test.bin HwHiAiUser@192.168.1.2:/home/HwHiAiUser/*temp*/matrix/input_zebra.bin

7. 参见步骤4~6拷贝Custom_Engine_AtlasDK工程下的网络模型到开发板matrix目录下。

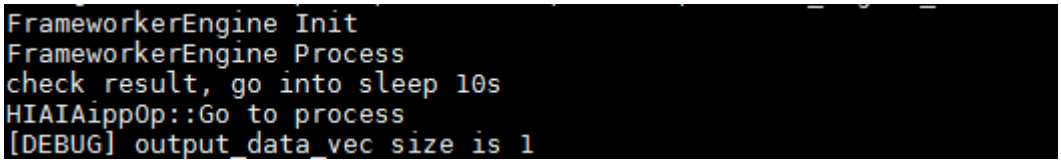
   网络模型为2.2.3.1.2 离线模型转化章节生成的模型文件resnet18.om。

8. 在开发板的"/home/HwHiAiUser/*temp*"目录执行如下可执行程序：

   ./Custom_Engine_AtlasDK

   如果弹出如图2-35所示信息则说明工程执行成功。

图 2-35 运行成功界面



工程执行完毕后，在相应的"/home/HwHiAiUser/*temp*/matrix"目录下生成**matrix_dvpp_framework_test_result**检测结果文件。

3 参考

3.1 数据类型注册

本用例中也用到自定义数据类型（EngineTrans）注册，在此进行额外的说明，也可通过示例代码查看自定义数据类型注册方法。在跨进程或跨设备进行通讯时需要对数据序列化与反序列化，但可以进行序列化与反序列化的前提条件是先对该数据类型在HiAI Engine进行注册。若干常用的数据类型、模板数据类型在HiAI Engine中已经进行了注册，详见~/tools/che/ddk/ddk/include/inc/hiaiengine/data_type.h。以下为对自定义数据类型以及自定义模板数据类型的注册方法。

3.1.1 自定义模板数据类型注册

步骤1 定义自定义模板数据类型。

```
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_);
}
```

步骤2 注册自定义模板数据类型。

- 注册UseDefTemplateDataType<uint64_t, float, std::string>

```
HIAI_REGISTER_TEMPLATE_DATA_TYPE("UseDefTemplateDataType_uint64_t_float_string",
UseDefTemplateDataType, uint64_t, float, std::string);
```

- 注册UseDefTemplateDataType<uint64_t, uint64_t, uint64_t>

```
HIAI_REGISTER_TEMPLATE_DATA_TYPE("UseDefTemplateDataType_uint64_t_uint64_t_uint64_t",
UseDefTemplateDataType, uint64_t, uint64_t, uint64_t);
```

----结束

3.1.2 自定义数据类型注册

步骤1 定义自定义数据类型*UseDefDataTypeT*。

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



```
    ar(data.data1, data.data2);
}
```

步骤2 注册自定义数据类型*UseDefDataTypeT*。

```
HIAI_REGISTER_DATA_TYPE("UseDefDataTypeT", UseDefDataTypeT);
```

----结束

3.2 多端口消息接收

本用例中未涉及多端口输入输出的处理流程，在此进行额外的说明，也可通过示例代码查看自定义数据类型注册方法。在实际使用HiAI Engine的情况中存在一个Engine的输出端口结果发送到多个Engine的输入端口，或者一个Engine的输入端口接受多个Engine的输出端口发送来的信息。对Engine的多个端口的处理都由用户根据自己的实际情况进程处理。

下面举例说明多端口输入的Engine实例通过私有成员变量input_que_进行多端口的输入消息管理。

步骤1 PushData

```
input_que_.PushData(0,arg0);
input_que_.PushData(1,arg1);
input_que_.PushData(2,arg2);
```

步骤2 PopData

```
std::shared_ptr<std::string> input_arg1;
std::shared_ptr<UseDefDataTypeT> input_arg2;
std::shared_ptr<UseDefTemplateDataType<uint64_t, uint64_t, uint64_t>> input_arg3;
```

步骤3 后续处理。

仅当三个端口都有输入数据时才执行此步骤操作。

```
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;
}
```

----结束

A FAQ

A.1 用户新定义的数据类型怎么在框架中使用

如果该数据类型不作为Engine的输入输出参数，则可直接使用；否则，请参见**3.1 数据类型注册**进行注册。

