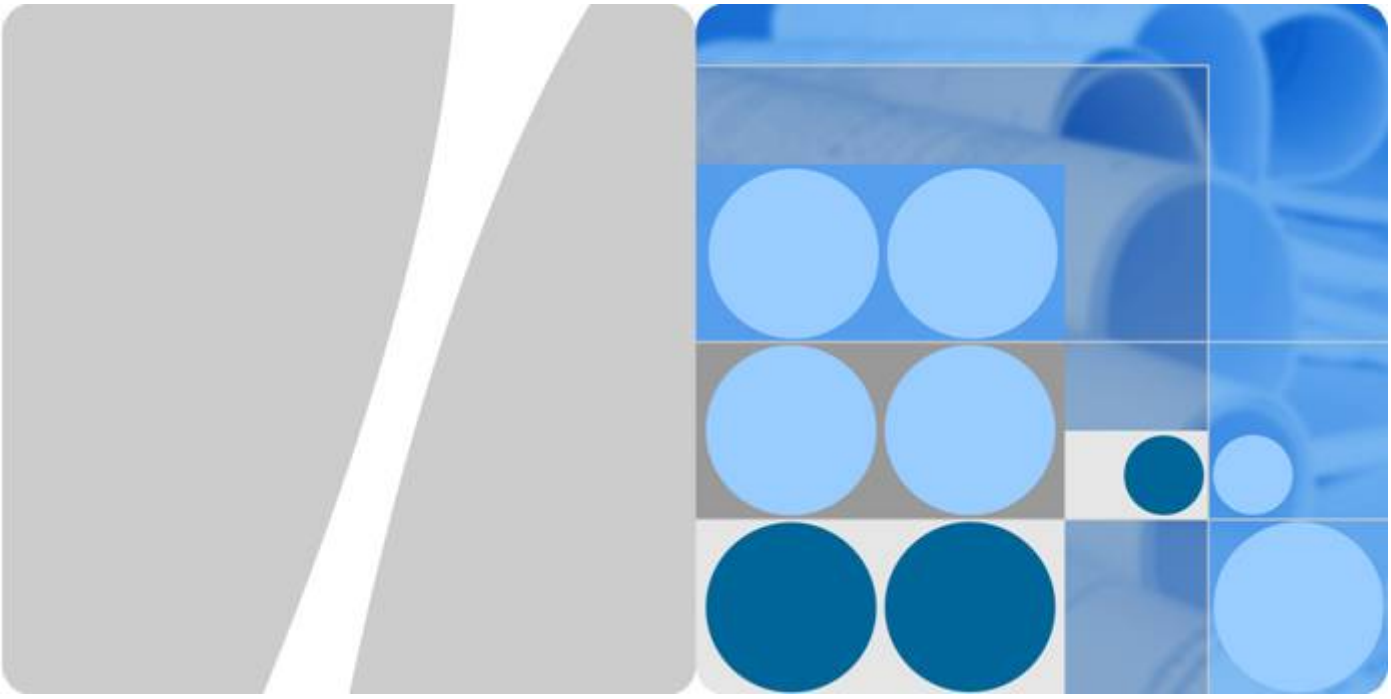


Ascend 310

V100R001

C++算子开发指导

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

Mind Studio中提供了深度学习框架Framework，可以将Caffe，Tensorflow等开源框架模型转换成Mind Studio支持的模型，并提供模型的加载、执行、卸载等管理能力。

在Mind Studio中进行模型转换时，如果模型中的算子在系统内置的算子库中未实现，则转换过程会报错。这时，未实现的算子就需要用户进行自定义，自定义的算子可加入到算子库中，使得模型转换过程可以正常进行。

在Mind Studio中我们提供了C++算子开发框架，可以开发自定义算子。C++算子开发框架是基于C/C++语言的自定义算子开发框架(目前只支持caffe)。

本文以Caffe网络模型Lenet-5内置算子Reduction的扩展自定义caffe_reduction_layer算子为例进行讲解，此算子目前Framework不支持，但可以通过编写算子插件的方式完成Caffe模型的转化，最终通过转化好的网络模型进行Engine编排，实现数字概率的累加验证功能。

C++算子与TE算子对比如表1-1所示。

表 1-1 C++算子与 TE 算子对比

参数	C++算子	TE算子
语言	C/C++	Python
运用场景	常用于非NN运算，非矩阵运算类算子	常用于NN运算，矩阵运算类算子
入门难度	入门难度低，适用于C/C++语言使用者	入门难度高，需要掌握如何用schedule原语完成切分buffer，才能获取比较好的性能。 通过使用TE封装的接口，可以降低难度。
易用性	C/C++语言编程，容易掌握	通过DSL的增强的接口，可以方便的组装出算子

2 C++算子开发和使用

2.1 数据准备

本示例以mylenet网络的reducation算子为例进行讲解，在操作本示例前需要准备以下数据：

表 2-1 准备数据

准备数据	如何获取
mylenet网络模型	请获取附件test_data.zip中的mylenet的模型文件“deploy_mylenet-1.prototxt”与权重文件“mylenet-1.caffemodel”
单算子测试数据生成脚本	请获取附件test_data.zip中的“gen_test_data.py”
ministdata数据图片	请获取附件test_data.zip中的“1.jpg”

2.2 操作流程

算子开发流程如图2-1所示。

图 2-1 自定义算子开发

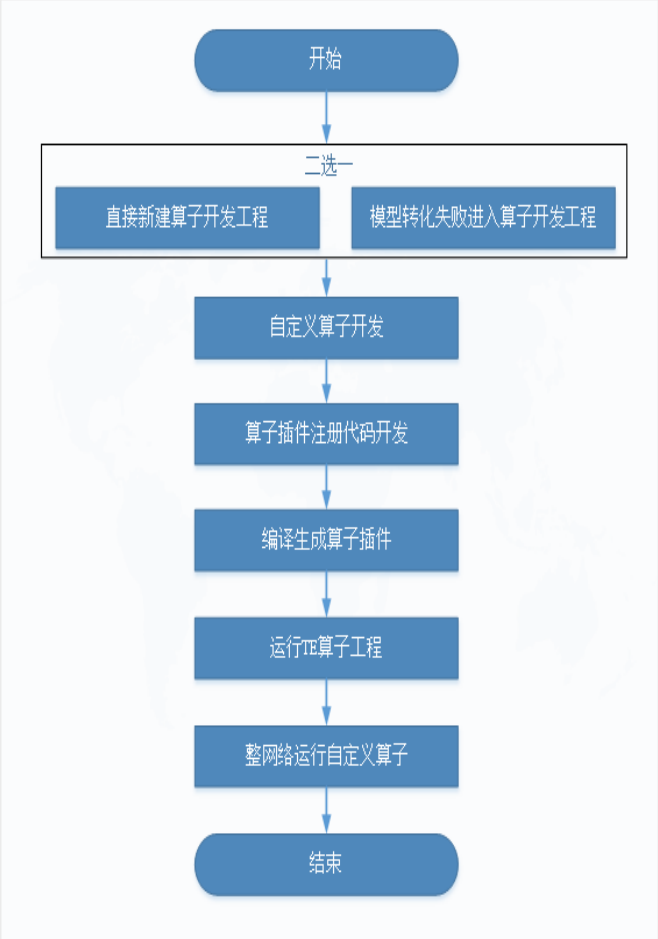


表 2-2 算子开发流程说明

阶段流程	说明	参考章节
创建C++算子工程	● 直接创建C++ Operator Project类型工程 ● 通过模型转化 > 转化失败进而进入算子创建工程	2.2.1 创建C++ Operator Project类型工程
算子开发	自定义算子开发	2.2.2.1 算子代码开发
插件开发	自定义算子插件开发	2.2.2.2 插件代码开发
编译工程	编译C++算子代码，生成算子插件。 完成C++算子开发，得到插件动态库和算子动态库。	2.2.3 编译C++ Operator Project类型工程

阶段流程	说明	参考章节
运行工程	运行单算子，通过Mind Studio界面上的单算子运行功能单独验证算子代码的正确性。	2.2.4 运行C++ Operator Project类型工程 调试C++ Operator Project类型工程
整网络运行	通过模型转化将算子动态库集成到模型中，进行流程编排及运行。	2.2.5 整网络运行自定义算子

2.2.1 创建 C++ Operator Project 类型工程

Mind Studio提供了两种进入算子创建工程界面的途径。

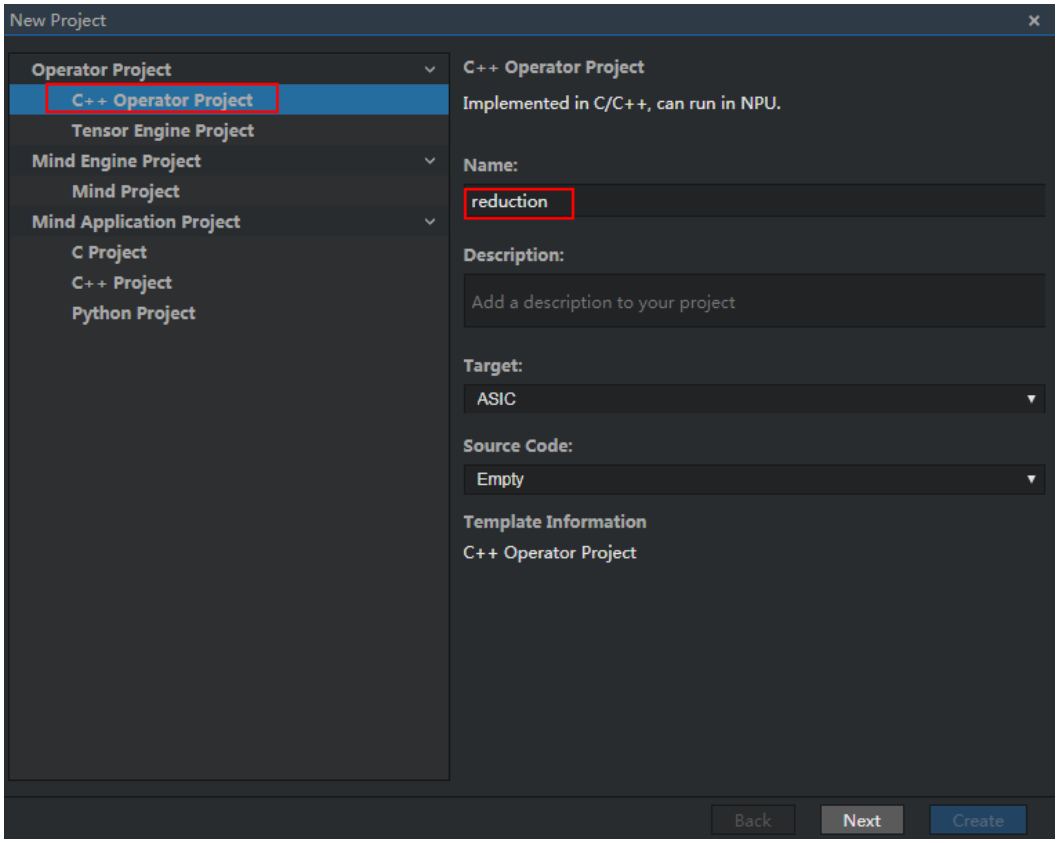
- 直接创建C++ Operator Project类型工程。
- 通过模型转化 > 转化失败进而进入算子创建工程界面。

2.2.1.1 直接新建算子开发工程

步骤1 在菜单栏上依次选择“File > New > New Project”，新建C++算子开发工程，如图2-2所示。

选择“C++ Operator Project”类型，填写“Name”，选择工程运行的“Target”，单击“Next”。

图 2-2 创建 C++ Operator Project 工程



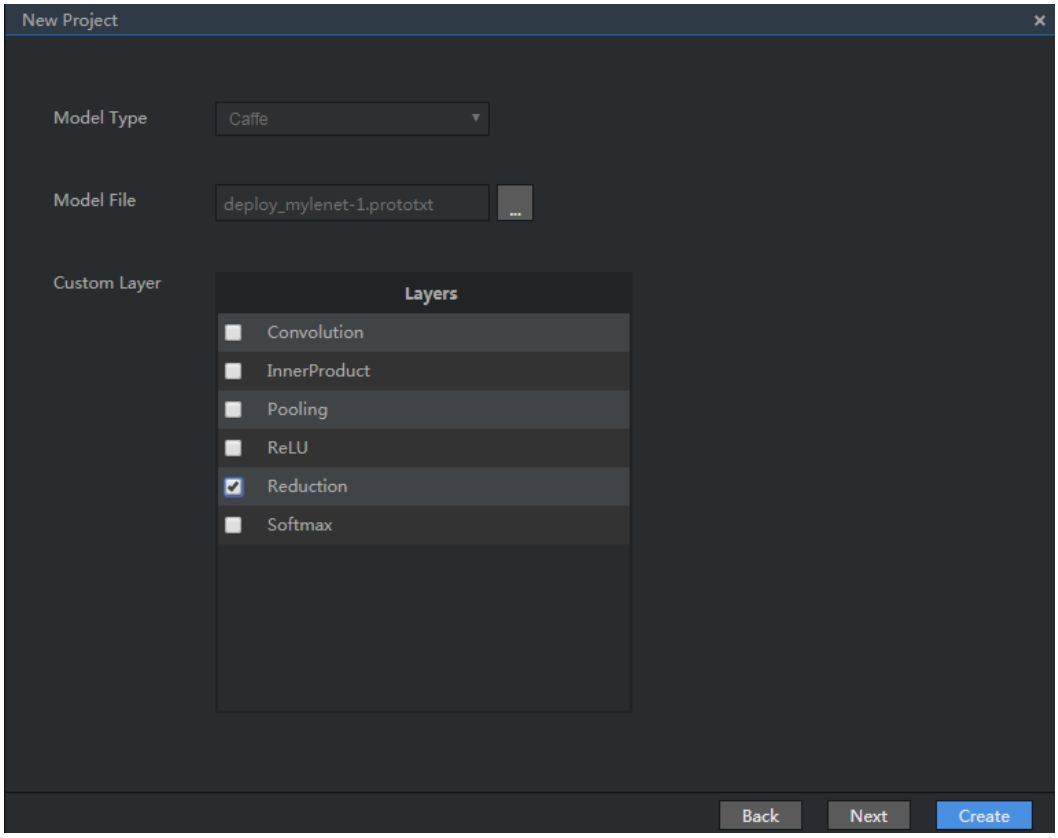
说明

目前暂不支持Atlas DK类型。

步骤2 单击“Model File”右侧...，选择Caffe模型的prototxt文件，在“Custom Layer”区域会列出模型中的每一层算子，选择“Reduction”算子，如图2-3所示。

如有想选择多个算子，则勾选相应算子。

图 2-3 选择模型文件及算子

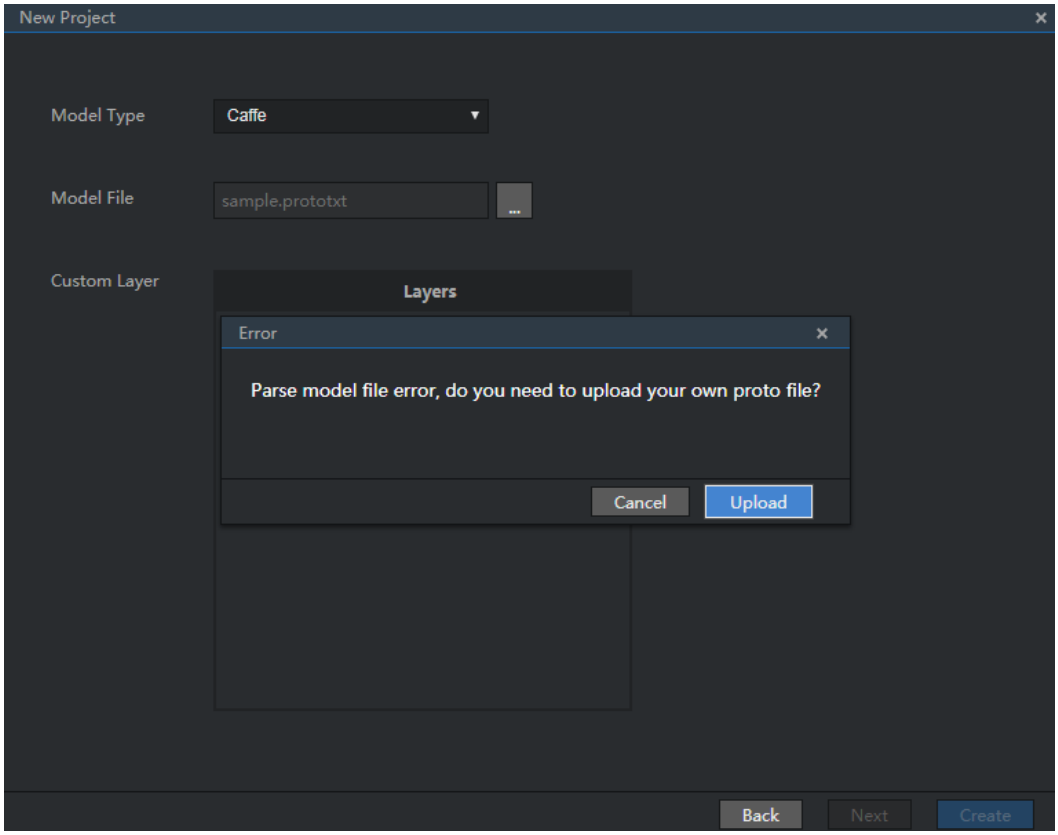


说明

模型文件请参考2.1 数据准备获取。

如果上传的prototxt需要自定义*caffe*.proto（系统内置的caffe.proto无法解析当前模型，存储路径“$HOME/tools/che/ddk/ddk/include/inc/custom/proto/caffe/caffe.proto”。），会导致第一次解析算子失败，如图2-4所示。

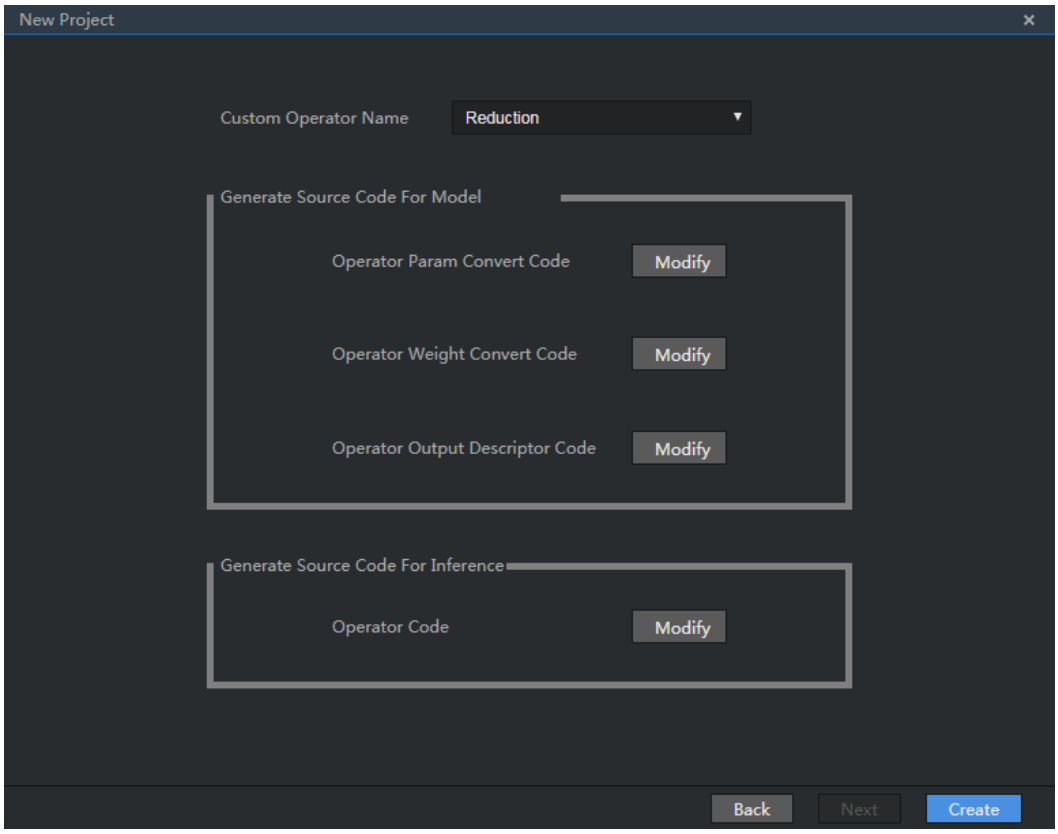
图 2-4 第一次解析算子失败



此时需要单击Upload，上传自定义的caffe.proto，系统会把所需字段合并到内置的caffe.proto中。

步骤3 若单击“Next”进入快速编辑界面，可对算子的几个模块代码进行编辑，如图2-5所示。

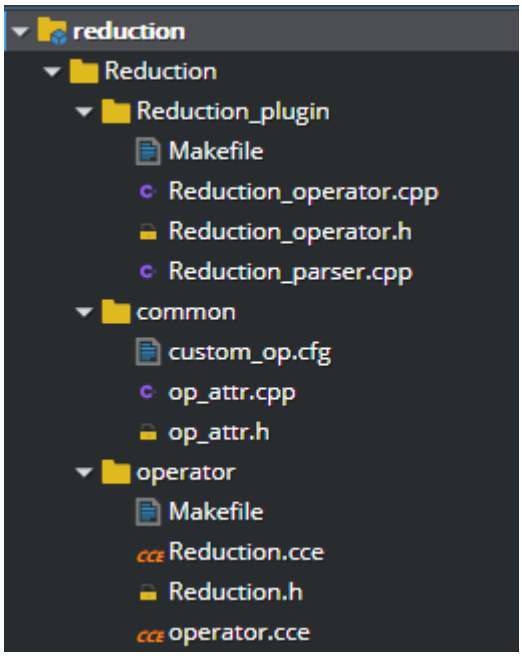
图 2-5 快速编辑界面



步骤4 完成后单击“Create”完成工程创建，也可以直接在图2-3中单击“Next”，完成算子工程的创建。

工程创建后的目录结构如图2-6所示。

图 2-6 工程创建后的目录结构



- Reduction_plugin：自定义算子插件代码目录
  - Reduction_operator.cpp：算子参数存取定义代码文件
  - Reduction_operator.h：算子参数存取定义头文件
  - Reduction_parser.cpp：算子参数解析代码文件（用户修改）
- common：单算子运行参数配置文件目录（仅单算子运行时需要修改）
  - custom_op.cfg：算子输入输出描述定义文件
  - op_attr.cpp：算子参数赋值代码文件
  - op_attr.h：算子参数结构体定义头文件
- operator：自定义算子代码目录
  - Reduction.cce：算子代码实现文件（用户实现）
  - Reduction.h：算子实现头文件
  - operator.cce：算子公用代码文件

----结束

2.2.1.2 模型转化失败进入算子开发工程

本章节使用Mind Studio的离线模型转换功能，将Caffe模型转化为NPU芯片支持的网络模型，转化过程中由于模型中有不支持的算子，导致转化失败，开发者可以通过转化失败界面进入自定义算子开发工程。

步骤1 创建需要使用模型文件进行AI工程开发的Mind工程，详情步骤请参见**2.2.5.2.1 创建工程**。

步骤2 在“Projects Explorer”窗口中双击“工程名 > XXX.mind”文件，在右侧tool栏中选择“Model > My Models”，单击“+”，添加网络模型。

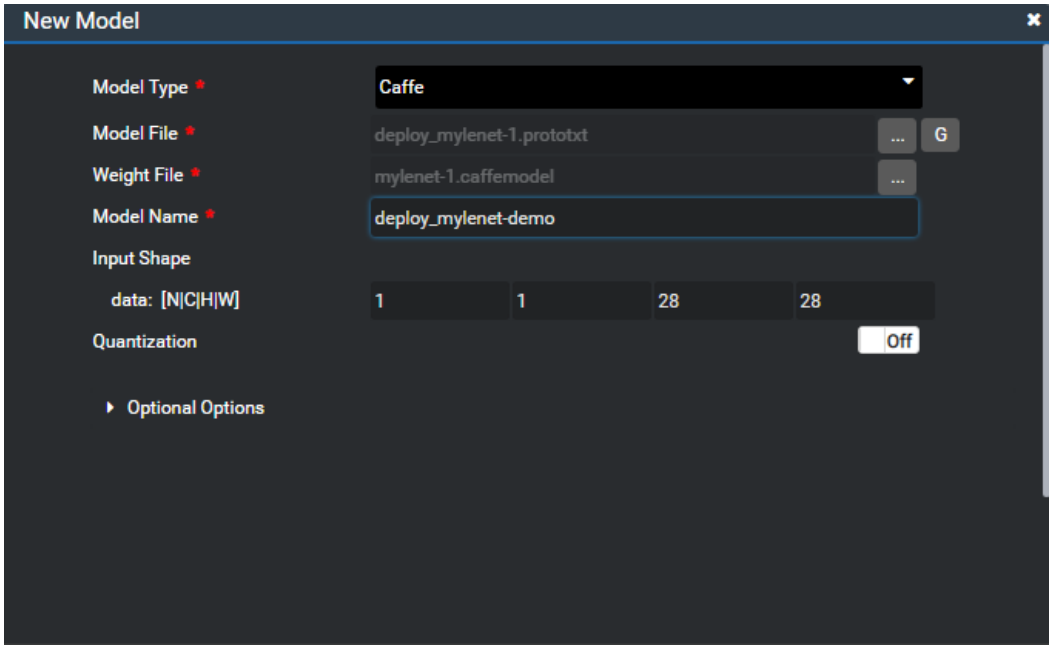
步骤3 在弹出的“NewModel”窗口，添加并配置需要添加的外部网络模型。

“Model type”选择“Caffe”。“Model file”和“Weight file”分别为模型文件和权重文件，如**图2-7**所示。

说明

模型文件与权重文件请参考**2.1 数据准备**获取。

图 2-7 选择模型文件

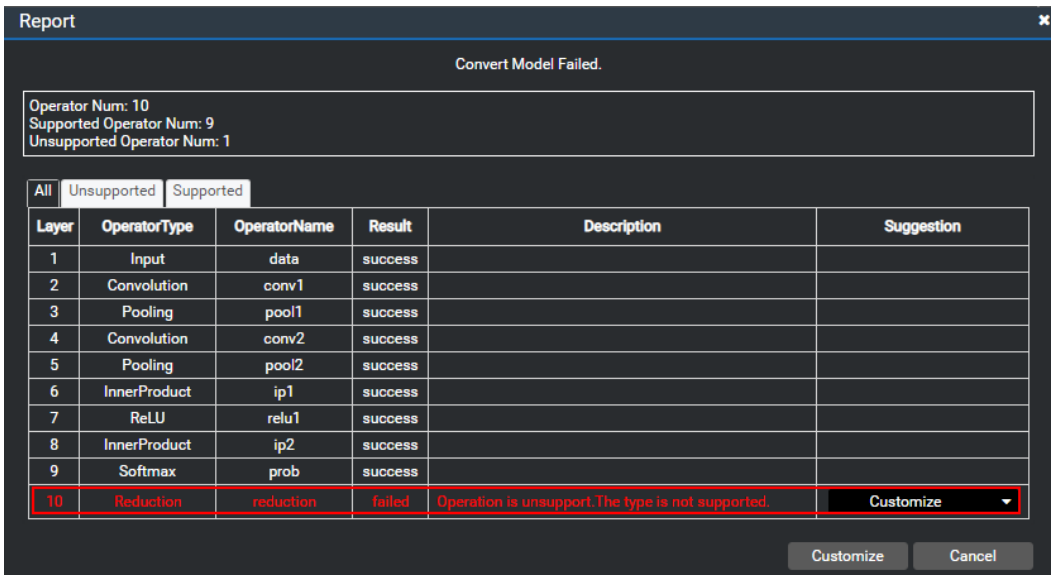


- “Model Name”自动填充模型文件的名称，用户可以在选择模型文件后自行修改为想要的名称。
- 工具会解析模型文件获取模型的默认“input shape”。

步骤4 单击“OK”，进行模型转化。

界面弹出如图2-8所示Report窗口，表示模型转化失败，模型中存在工具不支持的算子。

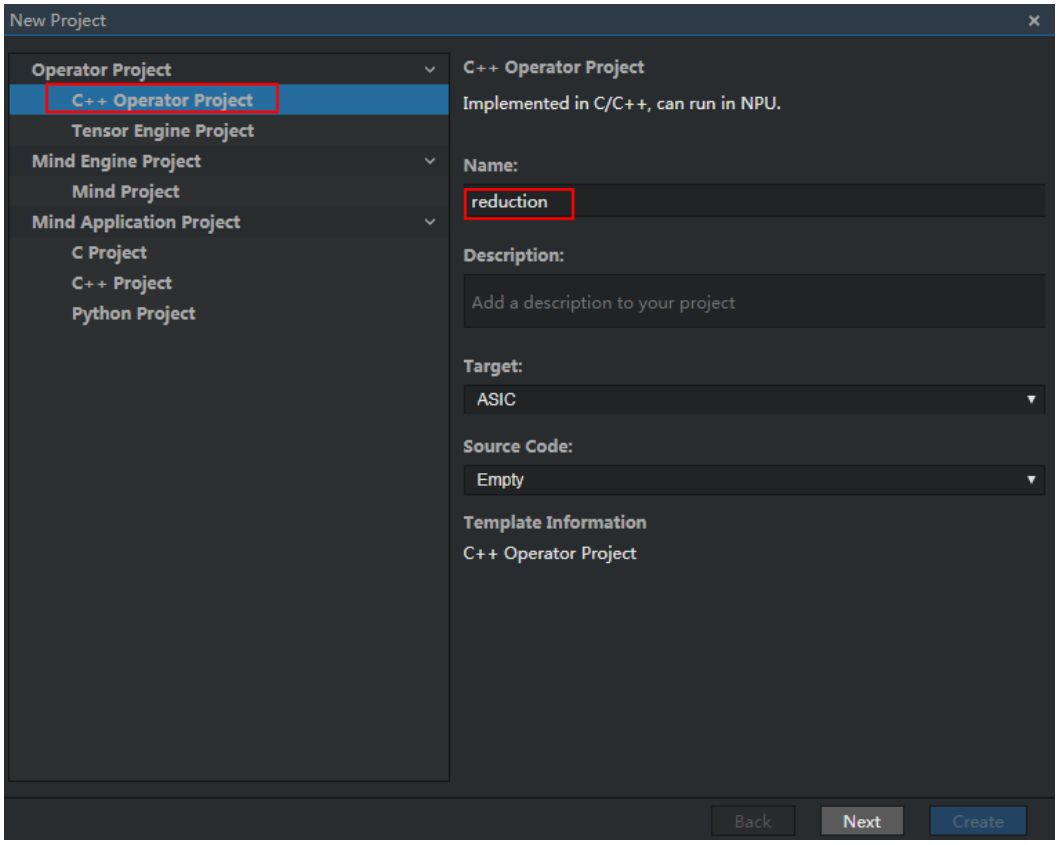
图 2-8 转化失败报告



步骤5 单击“Customize”，创建自定义算子工程。

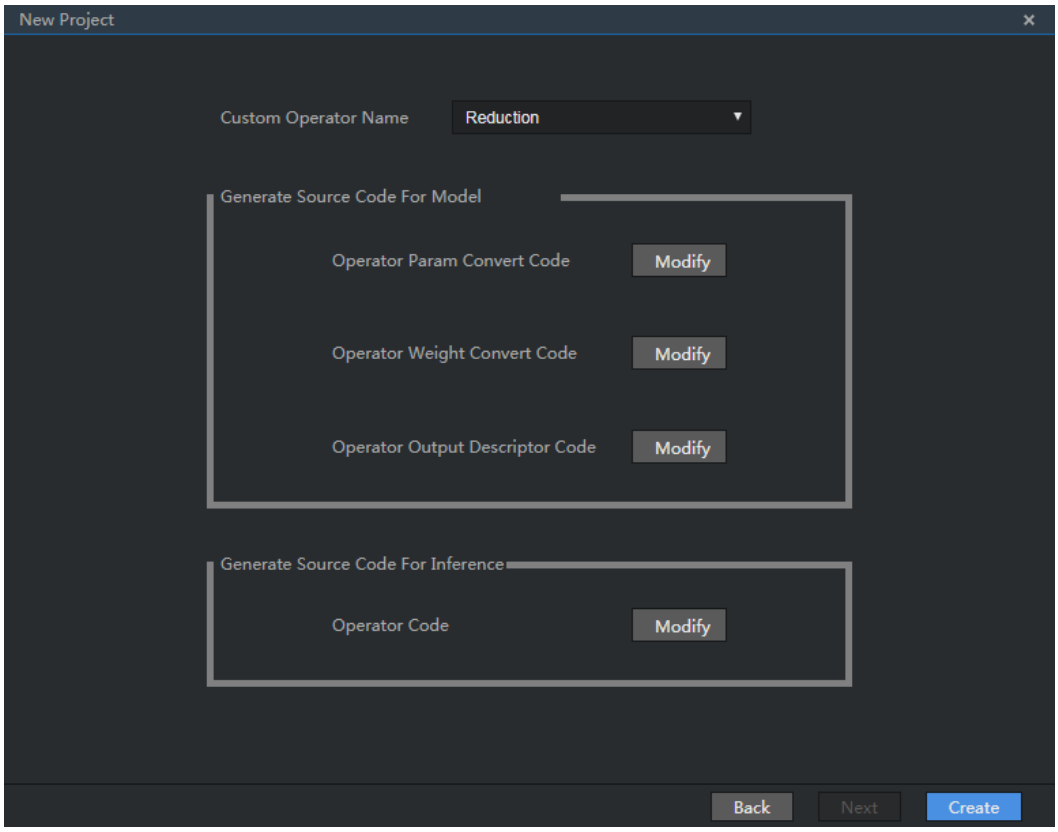
选择“C++ Operator Project”类型，填写“Name”，设置“Target”,如图2-9所示。

图 2-9 创建 C++ Operaotr Project 工程



步骤6 单击“Next”，进入代码编辑界面，如图2-10所示。

图 2-10 代码快速编辑界面

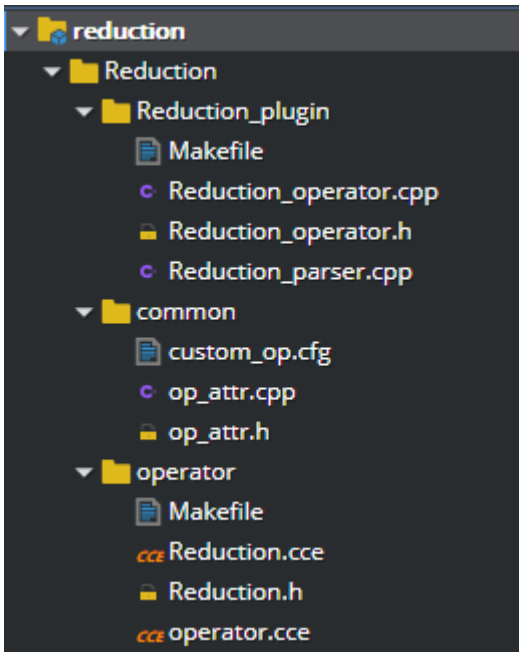


步骤7 单击“Create”，完成算子工程的创建。

单击“New Model”窗口右上角的关闭新增模型窗口。

工程创建后，在Projects区域可以看到创建的工程，如图2-11所示。

图 2-11 工程创建后的目录结构



- Reduction_plugin：自定义算子插件代码目录
  - Reduction_operator.cpp：算子参数存取定义代码文件
  - Reduction_operator.h：算子参数存取定义头文件
  - Reduction_parser.cpp：算子参数解析代码文件（用户修改）
- common：单算子运行参数配置文件目录（仅单算子运行时需要修改）
  - custom_op.cfg：算子输入输出描述定义文件
  - op_attr.cpp：算子参数赋值代码文件
  - op_attr.h：算子参数结构体定义头文件
- operator：自定义算子代码目录
  - Reduction.cce：算子代码实现文件（用户实现）
  - Reduction.h：算子实现头文件
  - operator.cce：算子公用代码文件

----结束

2.2.2 开发 C++ Operator Project 类型工程

2.2.2.1 算子代码开发

算子代码开发主要集中在operator目录下的“Reduction.cce”文件中。在reduction函数内补充算子逻辑代码。

附件reduction.zip为Reduction算子的实现代码。

reduction函数的输入参数详情见表2-3。

表 2-3 reduction 函数的输入参数

参数	类型	含义	备注
xDesc	opTensor_t *	输入数据描述	opTensor_t在ddk的include/include/inc/custom/common/aicpu_op.h内定义
xArray	void **	输入数据数组	-
yDesc	opTensor_t *	输出数据描述	-
yArray	void **	输出数据数组	-
opattr_handle	void *	算子参数结构体指针	为OpAttr类型，OpAttr结构体在op_attr.h文件内定义

以下为reduction算子的实现代码。

```
int64_t axis = ((OpAttr *) opattr_handle)->axis;
const void * x = inputArray[0];
```



```
void * y = outputArray[0];
if( x == NULL || y == NULL)
{
    return;
}
if(inputDesc->dim_cnt <=0 || inputDesc->dim_cnt >= CC_DEVICE_DIM_MAX)
{
    return;
}
for(int i=0;i<inputDesc->dim_cnt;i++)
{
    if(inputDesc->dim[i]<=0)
    {
        return;
    }
}
if(axis < 0)
{
    axis = axis + inputDesc->dim_cnt;
}
if( axis >= inputDesc->dim_cnt || axis < 0)
{
    return;
}
int dimAxisInner = 1;
int dimAxisOuter =1;
T sum =0;
T* pDst =(T*) y;
const T* pSrc = (const T*) x;
//calculate the element count before and after axis
for(int i=0; i < axis; i++)
{
    dimAxisInner = dimAxisInner * inputDesc->dim[i];
}
for(int i=axis+1; i<inputDesc->dim_cnt;i++ )
{
    dimAxisOuter = dimAxisOuter * inputDesc->dim[i];
}
for(int idxDimOuter=0; idxDimOuter< dimAxisOuter; idxDimOuter++)
{
    //axis Outer Dim begin pointer
    const T* pSrcOuter =(const T*) x + idxDimOuter* dimAxisInner * inputDesc->dim[axis];
    for(int idxAxisInner = 0; idxAxisInner < dimAxisInner; idxAxisInner++)
    {
        sum = 0;
        pSrc = pSrcOuter + idxAxisInner;
        // caculate the sum of dim[axis] elements. stride is the DimAxisInner.
        for( int j=0; j < inputDesc->dim[axis];j++)
        {
            //间隔步长DimAxisInner 取数相加。次数为axis所在维度size
            T value = *(pSrc + j * dimAxisInner );
            sum = sum+ value;
        }
        *pDst = sum;
        pDst++;
    }
}
```

说明

算子代码由定制编译器编译，开发时需要注意：

1. 不支持Class特性
2. 不支持STL模板特性
3. 不能定义全局变量
4. 不可以使用多线程

2.2.2.2 插件代码开发

插件代码开发主要集中在Reduction_plugin文件夹下的Reduction_parser.cpp文件中。修改AdjustParams和GetOutputDesc两个函数。

附件Reduction_parser.zip为Reduction算子插件的实现代码。

- AdjustParams

  该函数的主要功能是从op_def_里面取出算子的参数，组装成OpAttr结构体，结构体在后面的流程中会传给Reduction.cce中的算子函数的opattr_handle变量。结构体定义在common/op_attr.h中，创建工程时根据网络定义自动生成。默认生成的AdjustParams函数代码如图2-12所示。

  图 2-12 默认生成的 AdjustParams 函数代码

```
virtual Status AdjustParams() override
 {
     OpAttr op_attr;

    std::string operation = "";
    if (!GetOpAttr("operation", &operation , op_def_))
    {
        printf("GetOpAttr operation  failed!\n");
    }

    strncpy(&op_attr.operation[0], operation.c_str(), operation.size());
    op_attr.operation[operation.size()] = '\0';


    int64_t axis = -1;
    if (!GetOpAttr("axis", &axis , op_def_))
    {
        printf("GetOpAttr axis  failed!\n");
    }

    op_attr.axis  = axis ;


    float coeff = -1.0;
    if (!GetOpAttr("coeff", &coeff , op_def_))
    {
        printf("GetOpAttr coeff  failed!\n");
    }

    op_attr.coeff  = coeff ;

    AttrDef attr_def;
    attr_def.set_bt((char*)&op_attr, sizeof(OpAttr));

    std::string key = "opattr";
    auto attr = op_def_->mutable_attr();
    attr->insert(
        google::protobuf::MapPair<std::string, domi::AttrDef>(key, attr_def));

    return SUCCESS;
}
```

用户可以在这个函数里调整算子输入参数。

reduction算子需要对算子参数中的axis值进行修改，在函数开头添加如下代码，并删除函数内原有的axis定义。

```
vector<TensorDescriptor> v_output_desc;
v_output_desc.push_back(op_def_->input_desc(0));
int32_t axis = -1;
// ### 解析axis参数
if (!GetOpAttr("axis", &axis, op_def_))
```



```
    {
        printf("GetOpAttr axis failed!\n");
    }

    if (axis < 0) axis -= 2;
    if (axis < 0) axis += v_output_desc[0].dim_size();
    if (axis < 0 || axis >= v_output_desc[0].dim_size())
    {
        printf("invalid axis:%d, dim_size:%d\n", axis, v_output_desc[0].dim_size());
        return PARAM_INVALID;
    }
    AddOpAttr("axis", axis, op_def_);
```

修改后的代码如图2-13所示。

图 2-13 修改后的代码

```
virtual Status AdjustParams() override
{
    vector<TensorDescriptor> v_output_desc;
    v_output_desc.push_back(op_def_->input_desc(0));
    int32_t axis = -1;
    // ### 解析axis参数
    if (!GetOpAttr("axis", &axis, op_def_))
    {
        printf("GetOpAttr axis failed!\n");
    }

    if (axis < 0) axis -= 2;
    if (axis < 0) axis += v_output_desc[0].dim_size();
    if (axis < 0 || axis >= v_output_desc[0].dim_size())
    {
        printf("invalid axis:%d, dim_size:%d\n", axis, v_output_desc[0].dim_size());
        return PARAM_INVALID;
    }
    AddOpAttr("axis", axis, op_def_);

    OpAttr op_attr;

    std::string operation = "";
    if (!GetOpAttr("operation", &operation , op_def_))
    {
        printf("GetOpAttr operation  failed!\n");
    }

    strncpy(&op_attr.operation[0], operation.c_str(), operation.size());
    op_attr.operation[operation.size()] = '\0';


    //int64_t axis = -1;
    if (!GetOpAttr("axis", &axis , op_def_))
    {
        printf("GetOpAttr axis  failed!\n");
    }

    op_attr.axis  = axis ;


    float coeff = -1.0;
    if (!GetOpAttr("coeff", &coeff , op_def_))
    {
        printf("GetOpAttr coeff  failed!\n");
    }

    op_attr.coeff  = coeff ;
```

- GetOutputDesc

  该函数的主要作用是设置算子的输出描述信息，默认生成的代码如图2-14所示。

图 2-14 GetOutputDesc 默认生成的代码

```
virtual Status GetOutputDesc(vector<TensorDescriptor>& v_output_desc) override
{
    // TODO: add output desc to v_output_desc
    v_output_desc.push_back(op_def_->input_desc(0));

    return SUCCESS;
}
```

将输出描述结构体放入v_output_desc向量中，默认输出描述和输入相同。
用户可以在这个函数内修改输出数据描述信息。
输出描述的数据类型为TensorDescriptor，定义请参见**TensorDescriptor结构**。
该结构体在后面的流程中会传给算子函数的outputDesc变量。
reduction算子的GetOutputDesc函数的实现如下。

```
int32_t axis = -1;
// ### 解析axis参数
if (!GetOpAttr("axis", &axis, op_def_))
{
    printf("GetOpAttr axis failed!\n");
}
// 由于davinci模型会将所有shape补齐到4d，这里需要修复axis，指向原来2d时的位置
v_output_desc[0].set_dim(axis, 1);
```

修改后的代码如**图2-15**所示。

图 2-15 修改后的代码

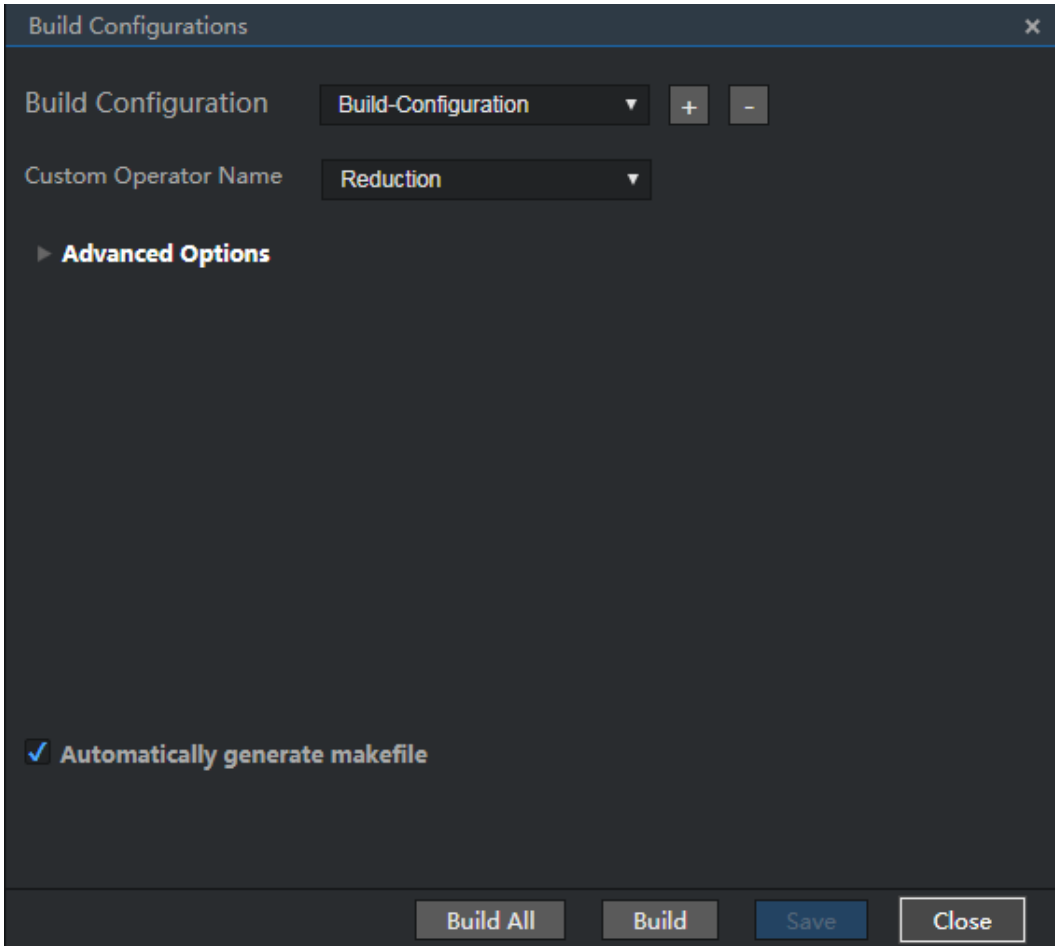
```
virtual Status GetOutputDesc(vector<TensorDescriptor>& v_output_desc) override
{
    // add output desc to v_output_desc
    v_output_desc.push_back(op_def_->input_desc(0));
    int32_t axis = -1;
    // ### 解析axis参数
    if (!GetOpAttr("axis", &axis, op_def_))
    {
        printf("GetOpAttr axis failed!\n");
    }
    // 由于davinci模型会将所有shape补齐到4d，这里需要修复axis，指向原来2d时的位置
    v_output_desc[0].set_dim(axis, 1);

    return SUCCESS;
}
```

2.2.3 编译 C++ Operator Project 类型工程

操作步骤

步骤1 选中C++ Operator工程，依次单击“Build > Edit Build Configuration”，如**图2-16**所示。

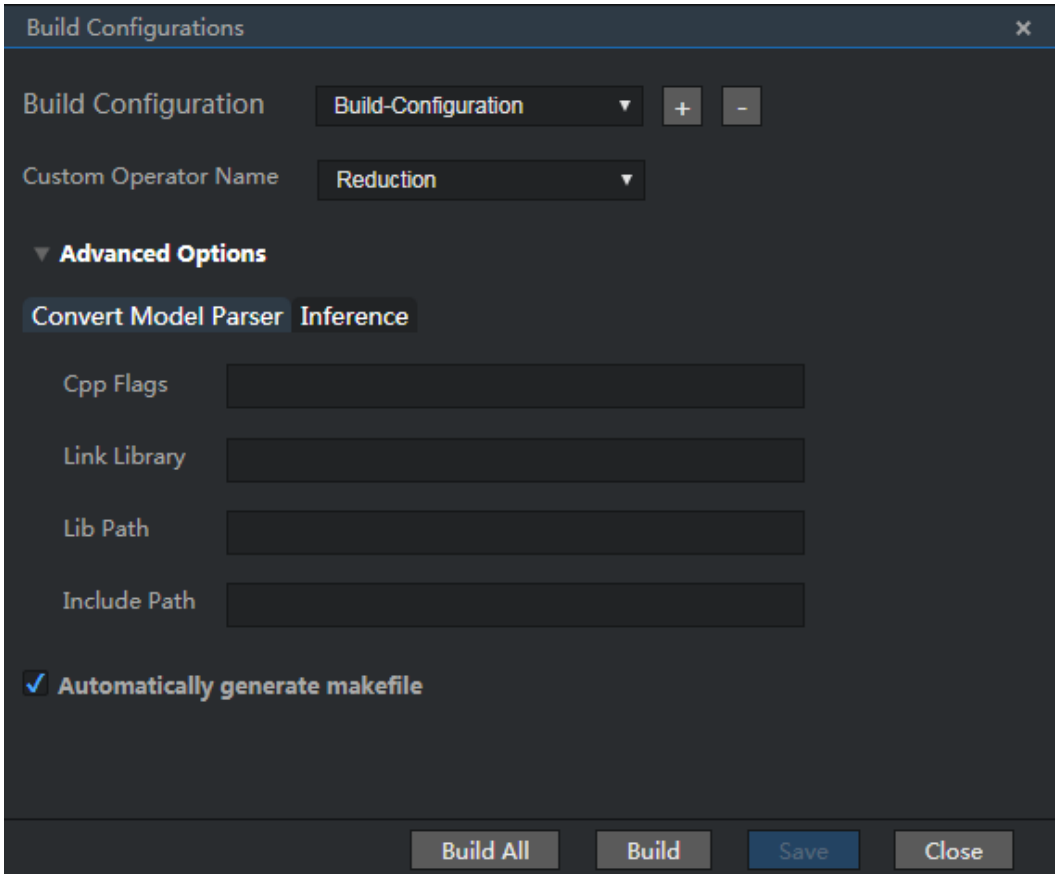
图 2-16 编译 C++ Operator Project 类型工程



步骤2 在“Custom Operator Name”栏选择需要编译的算子，单击“Build”进行编译。

C++算子编译功能支持配置额外的编译参数，如图2-17所示。

图 2-17 C++算子编译功能支持配置额外的编译参数



其中，Convert Model Parser页的配置对应算子插件的编译参数，Inference页的配置对应算子的编译参数。

配置选项的含义详见表2-4。

表 2-4 配置选项含义

配置项	含义
Cpp Flags	编译选项，用户配置。 对应CPPFLAGS。
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。
Automatically Generate Makefile	是否自动生成Makefile文件。

步骤3 单击界面下方C++ operator build 和C++ operator plugin build 标签页，查看编译结果，如图图2-18和图2-19所示。

图 2-18 C++算子编译结果

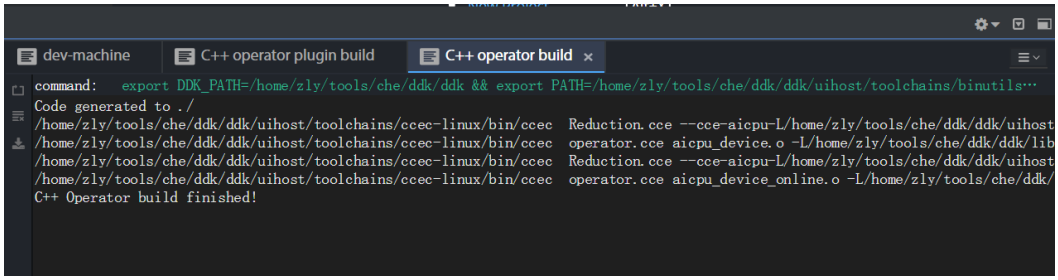
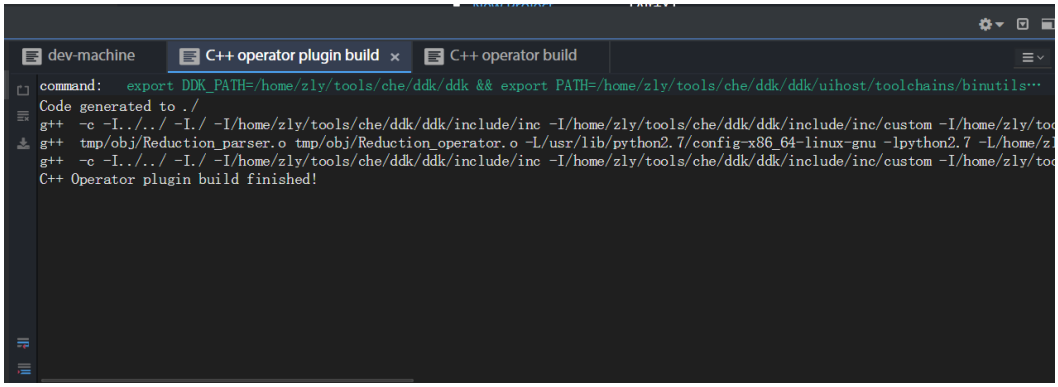


图 2-19 C++算子插件编译结果



----结束

2.2.4 运行 C++ Operator Project 类型工程

本节描述如何单独运行用户开发的算子，验证算子实现的正确性。

2.2.4.1 构造输入数据文件

构造输入数据，并以binary格式保存到文件，以下为reduction算子测试数据生成步骤。

步骤1 以Mind Studio安装用户登录Mind Studio所在服务器。

步骤2 将2.1 数据准备中的单算子测试数据生成脚本**gen_test_data.py**、mylenet网络模型文件“deploy_mylenet-1.prototxt”、权重文件“mylenet-1.caffemodel”、及ministdata数据图片“1.jpg”上传到Mind Studio所在Linux服务器的C++算子工程目录下，例如“/projects/reduction”。

步骤3 执行如下命令添加环境变量。

export LD_LIBRARY_PATH=~/tools/che/ddk/ddk/lib/x86_64-linux-gcc5.4/

步骤4 在“/projects/reduction”目录下执行**python gen_test_data.py**，生成样例数据。

reduction算子构造的输入数据如下：

其中.data为二进制格式，在后面流程中用到，.txt为文本格式，与二进制格式相对应。

数据文件说明如表2-5所示。

表 2-5 数据文件说明

数据文件	说明
reduction_input.data	reduction算子的二进制格式输入数据文件。
reduction_input.txt	将reduction算子的二进制格式文件以txt文件的方式显示出来，方便用户查看结果。
reduction_output.data	reduction算子二进制格式输出数据验证文件，用于验证算子运行后的输出结果是否正确。
reduction_output.txt	用户可见的reduction算子的输出数据文件，用处同上。

----结束

2.2.4.2 配置输入输出描述文件

输入输出描述文件为“custom_op.cfg”文件，位于算子文件夹的“common”目录下。

自定义C++算子参数配置文件使用protobuf文件格式编写，包含输入Tensor信息、输出Tensor信息。

文件格式是按照davinci的proto语法文件中OpDef定义格式书写。

```
name: "${opeartor_type}_op"                ---- 算子的name
type: "${operator_type}"      ---- 算子的type
input_desc {         ---- 算子的输入描述
    format: 3              --输入format:   格式，取值参见附录B CCE tensor format 类型，   一般填3
    data_type: 1           --输入data_type:   OP_DATA_FLOAT = 0  OP_DATA_HALF = 1
                                  OP_DATA_INT8 = 2   OP_DATA_INT32 = 3   OP_DATA_UINT8 = 4
    dim: 4                 --输入的维度值
    dim: 18
    dim: 19
    dim: 19
    real_dim_cnt: 4        --输入的维度数
}
output_desc {          ----算子的输出描述，结构同输入
    format: 3
    data_type: 1
    dim: 4
    dim: 18
    dim: 19
    dim: 19
    real_dim_cnt: 4
}
```

reduction算子需要修改自动生成的custom_op.cfg文件为。

```
name: "reduction"
type: "reduction_layer"
input_desc {
  format: 3
  data_type: 1
  dim: 1
  dim: 10
  real_dim_cnt: 2
}
output_desc {
  format: 3
  data_type: 1
  dim: 1
  real_dim_cnt: 1
}
```


2.2.4.3 设置算子参数值

算子参数值在common/op_attr.cpp文件内设置。默认会生成setOpParam函数，如图2-20所示。在函数内直接op_attr结构体的值，结构体定义在op_attr.h文件中。op_attr在单算子运行时会传递给算子函数的opattr_handle变量。

图 2-20 默认生成的 setOpParam 函数

```
#include "op_attr.h"
#include "string.h"
void  setOpParam(OpAttr * op_attr){

    /* set attr value here. */


    return ;

}
```

reduction算子的setOpParam实现如图2-21所示。

```
op_attr->axis = 1;
```

图 2-21 reduction 算子的 setOpParam 实现

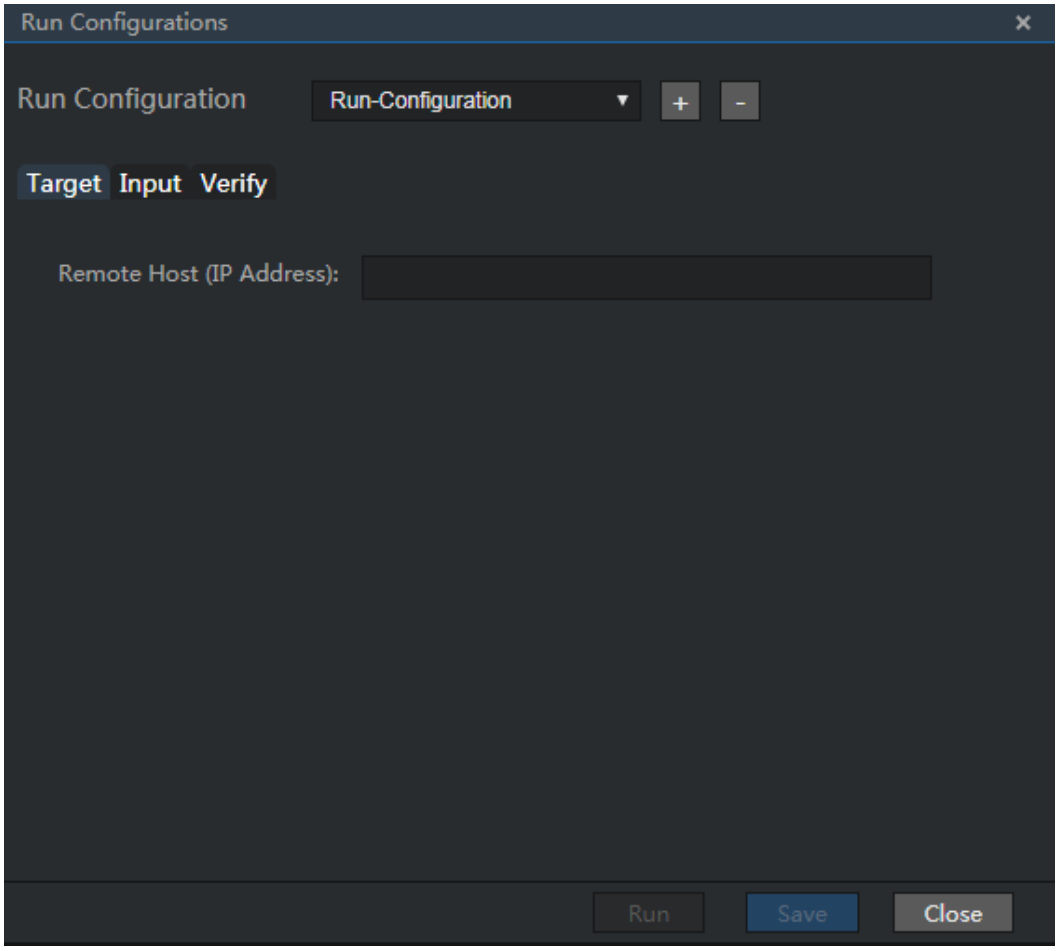
```
op_attr.cpp ×
1   #include "op_attr.h"
2   #include "string.h"
3   void  setOpParam(OpAttr * op_attr){
4
5       /* set attr value here. */
6       op_attr->axis = 1;
7
8       return ;
9
10  }
11
```

2.2.4.4 配置运行参数

操作步骤

步骤1 选中C++算子工程，依次单击“Run > Edit Run Configuration”，弹出运行窗口配置界面，如图2-22所示。

图 2-22 target 界面



步骤2 运行参数配置。

参数配置说明如表2-6所示。

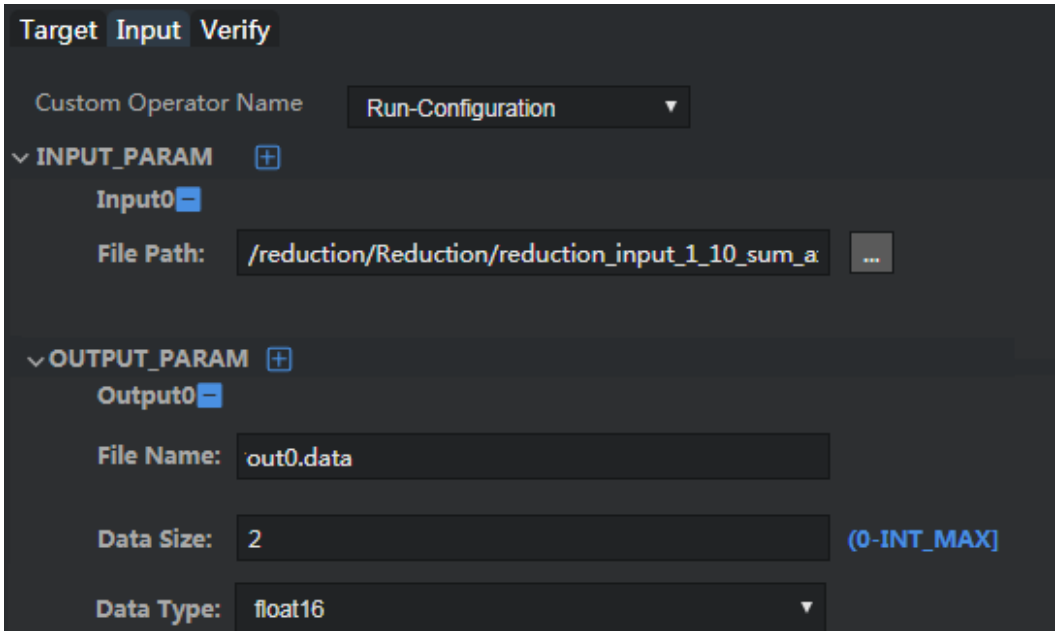
表 2-6 配置参数说明

配置参数	参数说明
Target配置	配置运行目标信息。

配置参数	参数说明
Input配置	描述算子的输入输出数据信息，配置示例如图2-23所示。 ● “Custom Operator Name” 单算子名的配置，工程新建后，名称会自动显示，本样例中算子名为Input。 ● INPUT_PARAM：输入数据的描述，可根据实际输入数据个数添加，最多允许20条输入数据。 Input0：第一个输入数据。 File Path：输入数据文件，选择2.2.4.1 构造输入数据文件构造的二进制输入数据文件。 ● OUTPUT_PARAM：输出数据的描述，可根据实际输出数据个数添加，最多允许10条输入数据。 Output0：第一个输出数据。 – File Name：输出数据文件，默认为输出到输出到output文件夹下的out0.data文件。 – Data Size：输出数据大小，单位字节。 – Data Type：输出数据类型，选择float16或者float32。
Verify配置	该配置页下的参数用于校验Run配置中输出结果和用户预期结果是否一致。此页签参数为可选参数，如果不配置，则运行时不校验输出数据正确性。 配置项意义如下，配置示例如图2-24所示： ● EXPECT_PARAM：用户预期的结果，用于和输出结果检验，数据个数需要与输出数据个数相同，最多允许10条输入数据。 Expect0：第一个验证数据。 File Path：验证数据文件。 ● Precision Deviation：精度偏差，表示单数据相对误差允许范围，数值范围为（0，1），精度偏差越小表示精度越高（默认为0.2）。 ● Statistical Discrepancy：统计偏差，整个数据集中精度偏差不满足门限的数据量占比，数值范围（0，1），统计偏差越小表示精度越高（默认为0.2）。

input界面：描述算子的输入输出数据信息，如图2-23所示。

图 2-23 input 界面

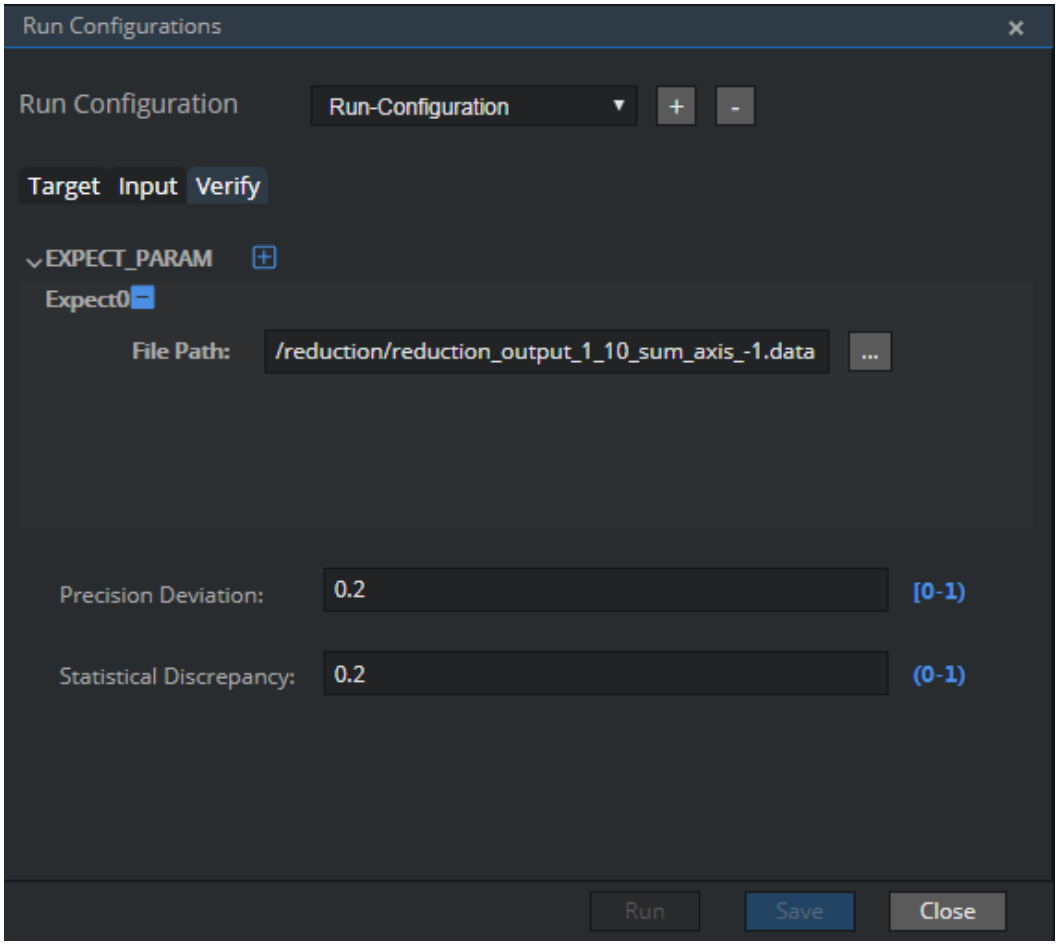


说明

reduction算子Input0 FIle Path选择**2.2.4.1 构造输入数据文件**中的输入文件，data size填写2（输出数据类型大小2字节*输出数据个数1）。

Verify界面：验证数据描述，非必填，如**图2-24**所示。

图 2-24 verify 界面



说明

Verify中的EXPECT_PARAM中的个数要与Run配置中的OUTPUT_PARAM中的个数相同，且Verify配置中的配置项不存在空值，Tensor Engine运行时才会进行校验操作。

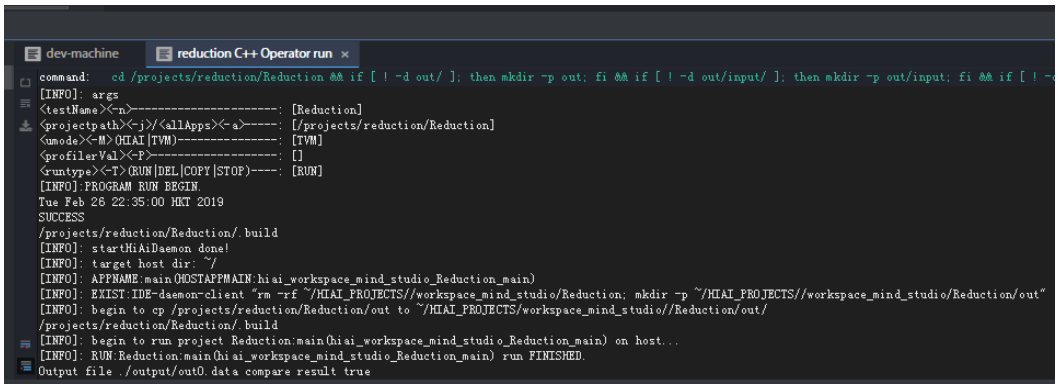
步骤3 完成配置后，单击“Save”保存。

----结束

2.2.4.5 运行单算子

完成运行配置后，单击Run运行，页面下方会出现“reduction C++ Operator run”窗口，如图2-25所示。

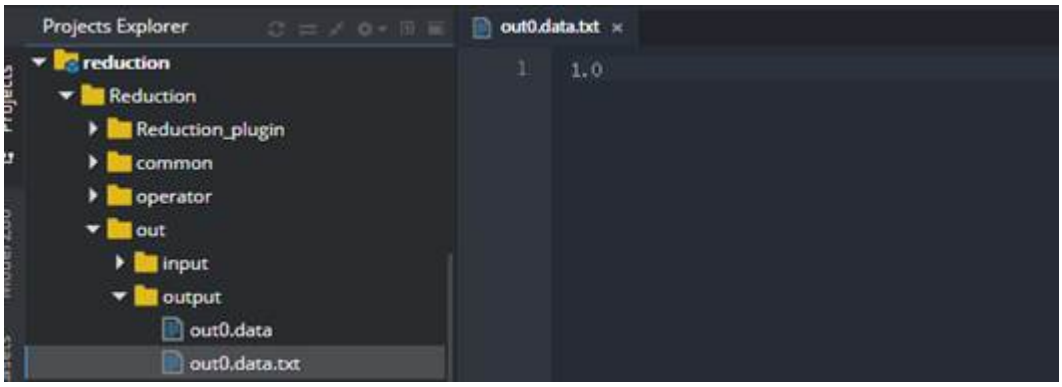
图 2-25 运行单算子



运行成功之后会打印run FINISHED 和比对结果。

在Ouput File Path参数指定的路径下会生成输出文件，打开.txt文件可以看到结果，如图2-26所示。

图 2-26 输出结果文件



说明

- 若用户想停止工程运行，可在菜单栏选择“Run > Stop > 工程名称”。
- 在运行时，会自动拷贝工程下模型文件、配置文件、链接库、二进制程序、输入输出数据等到host侧。为提高MindStudio 的运行性能，MindStudio不会自动删除host侧的数据，如用户不需要这些文件，可自行登录host侧，删除/home/HwHiAiUser/HIAI_DATANDMODELSET和/home/HwHiAiUser/HIAI_PROJECTS下的废弃文件夹。
- 同一Host机器上不支持并发运行单算子。

2.2.5 整网络运行自定义算子

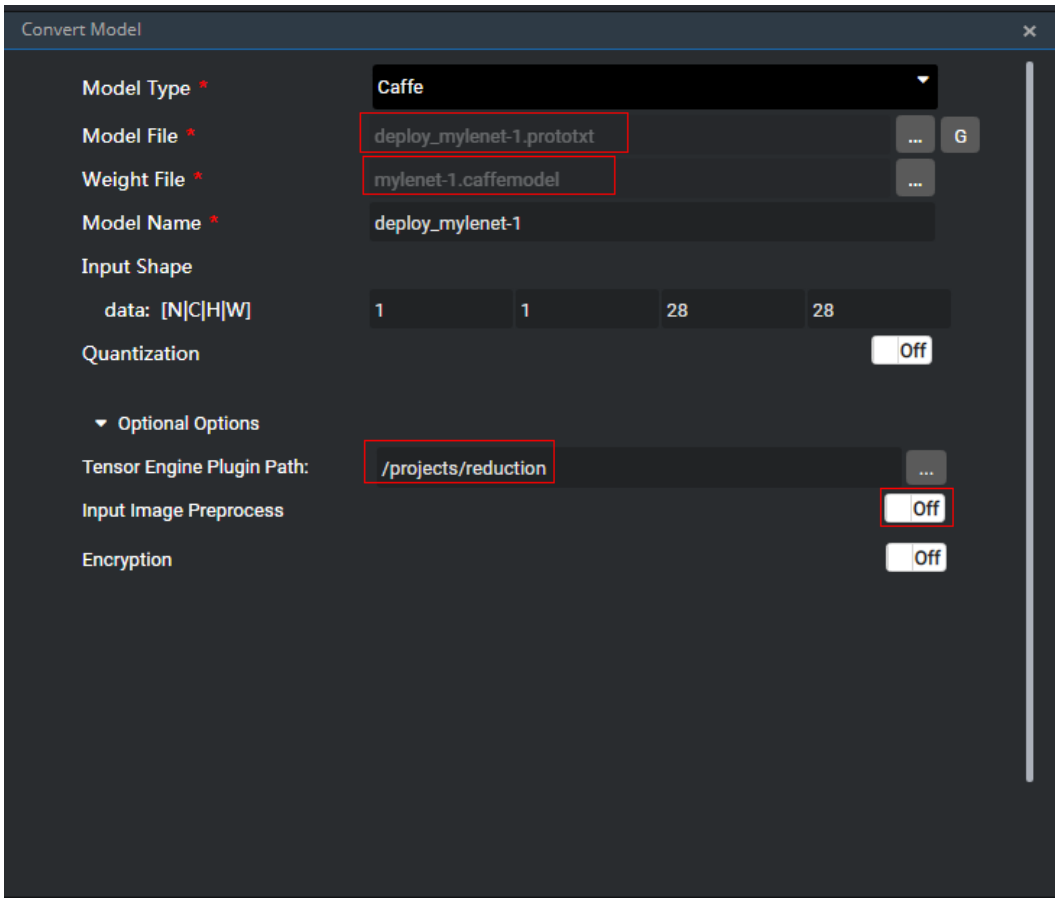
不同网络的运行步骤可能存在差异，本文以mylenet网络的运行步骤为例。

2.2.5.1 模型转换

步骤1 选中reduction工程，单击菜单栏“Tools > Convert Model”。

步骤2 在弹出的Convert Model窗口中，进行相应配置，如图2-27所示。

图 2-27 模型转换



- Model Type选择Caffe。
- Model File单击后面的...，选择需要转化的模型对应的portotxt文件，并上传。
- Weight File反击后面的...，选择需要转化的模型对应的caffemodel文件，并上传。
- 单击展开Optional Options，选择Plugin Path为reduction工程。
- 关闭Input Image Preprocess。

步骤3 所有配置完成后，单击“ok”，开始进行模型转换。

----结束

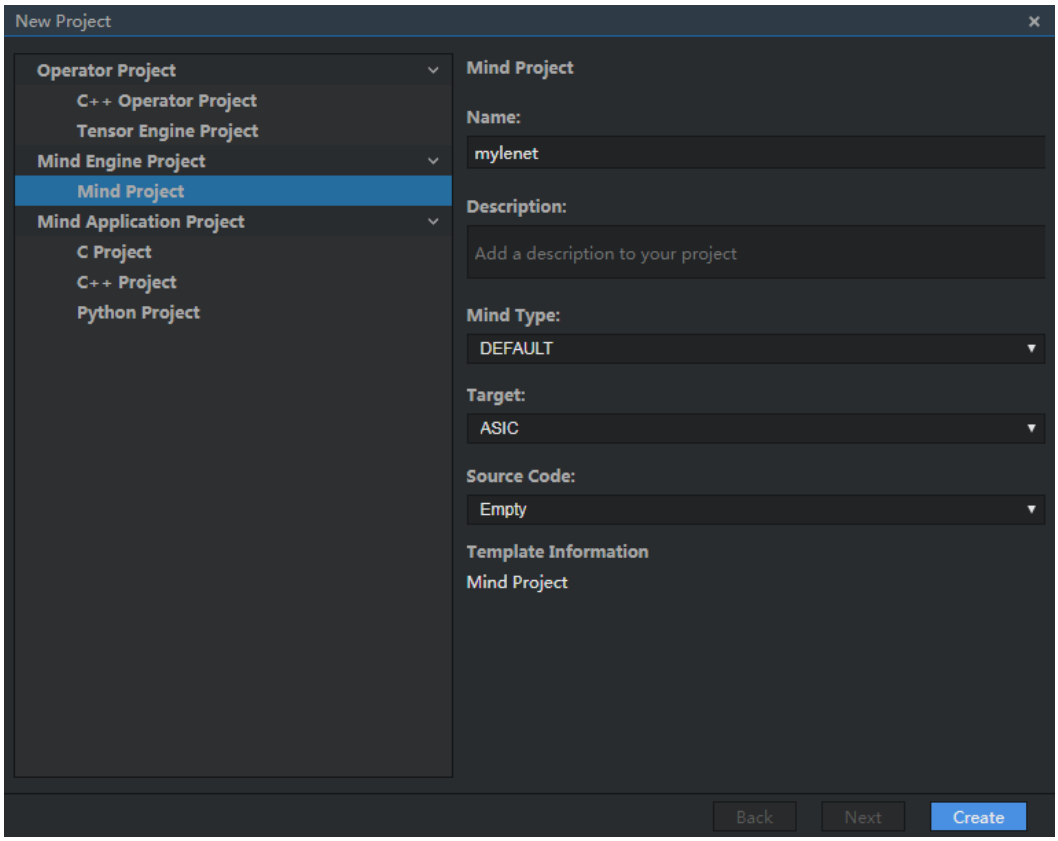
2.2.5.2 整网络运行

使用2.2.5.1 模型转换中转化好的模型，进行Engine编排，实现所有数字概率的累加验证功能。

2.2.5.2.1 创建工程

依次单击“File -> New -> New Project”，创建Mind Project工程，如图2-28所示。

图 2-28 创建 Mind Project 工程



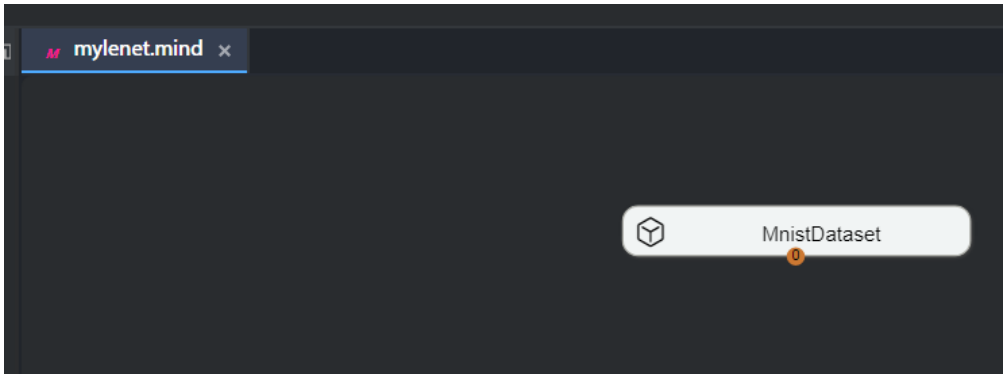
2.2.5.2.2 流程编排

操作步骤

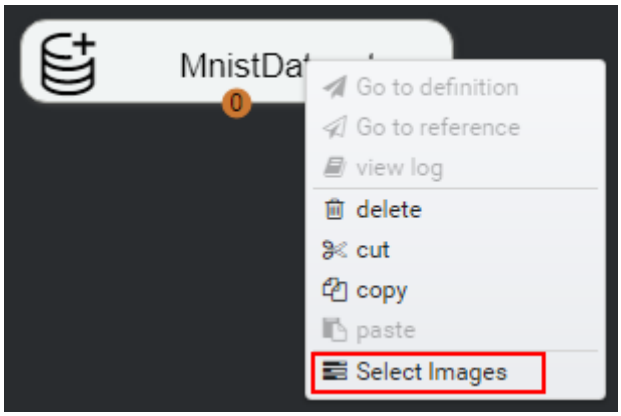
步骤1 创建Datasets节点，数据集节点基于内置的MnistDataset做相应的修改。

1. 双击打开mylenet.hiai文件，将右侧Datasets > BuiltIn Dataset > MnistDataset拖动至网络编排界面，如图2-29所示。

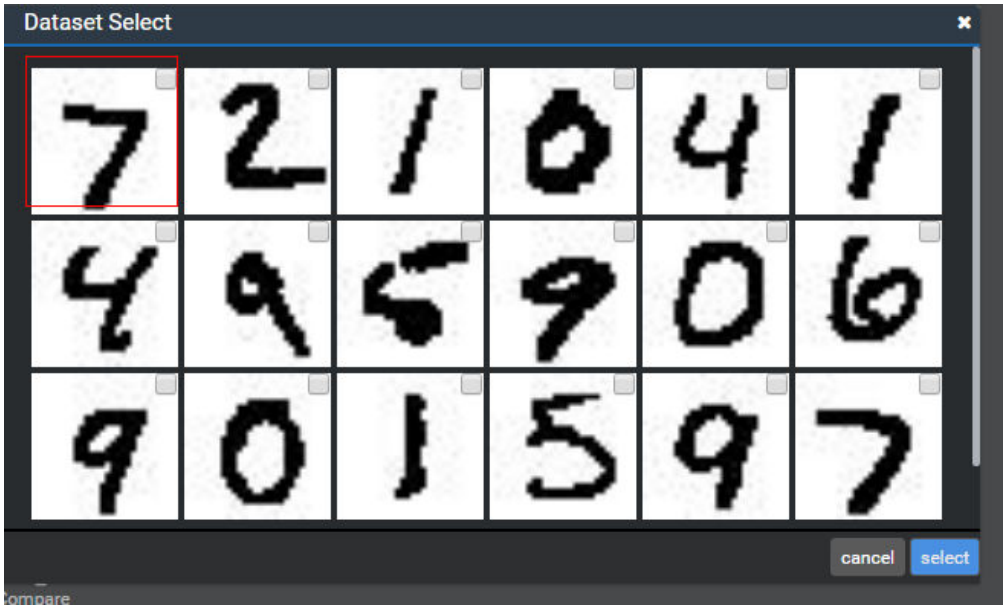
图 2-29 拖拽内置数据集



右键选择“Select Image”。



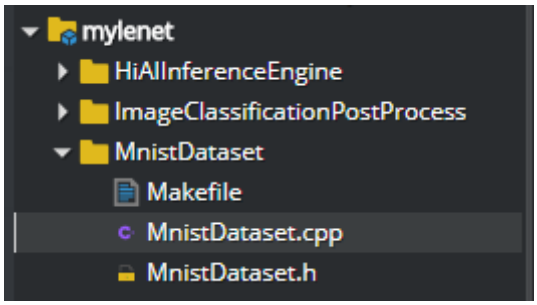
勾选其中一张图片后（此处选择第一张图片），单击“select”。



在左侧Demo工程下会出现跟“MnistDataset”数据集同名的文件夹，该文件中包括源码文件。

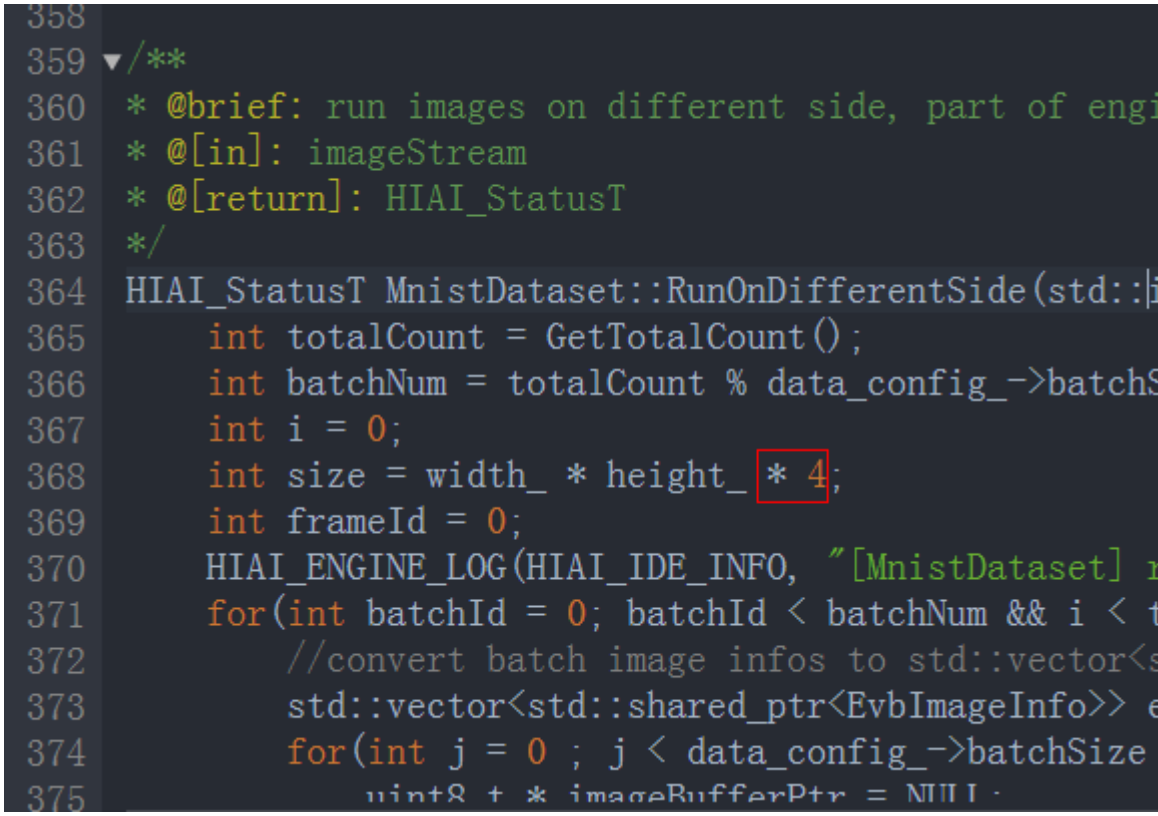
2. 内置的数据集示例为网络中处理一张灰度图，需要修改部分MinistDataset源码，打开MnistDataset.cpp源文件，如图2-30所示。

图 2-30 MnistDataset.cpp 源文件



3. 找到源文件的如下位置，修改相应参数，如图2-31所示。

图 2-31 修改源码



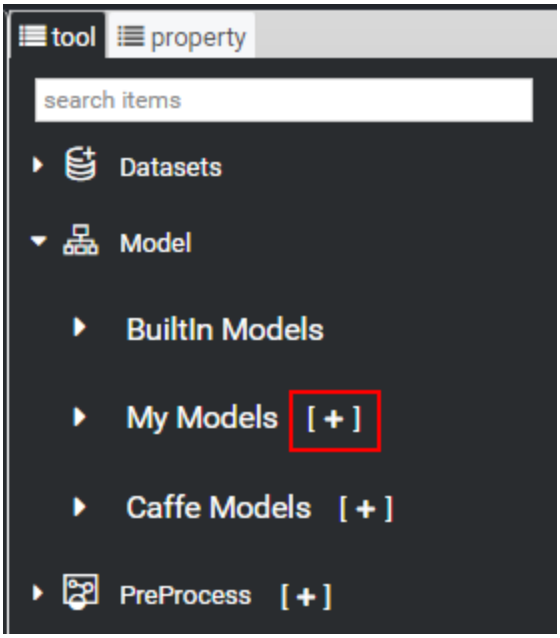
源码368行“size”修改为width_ * height_ *4。

步骤2 生成Model。

模型节点导入sample用例中的mylenet网络。

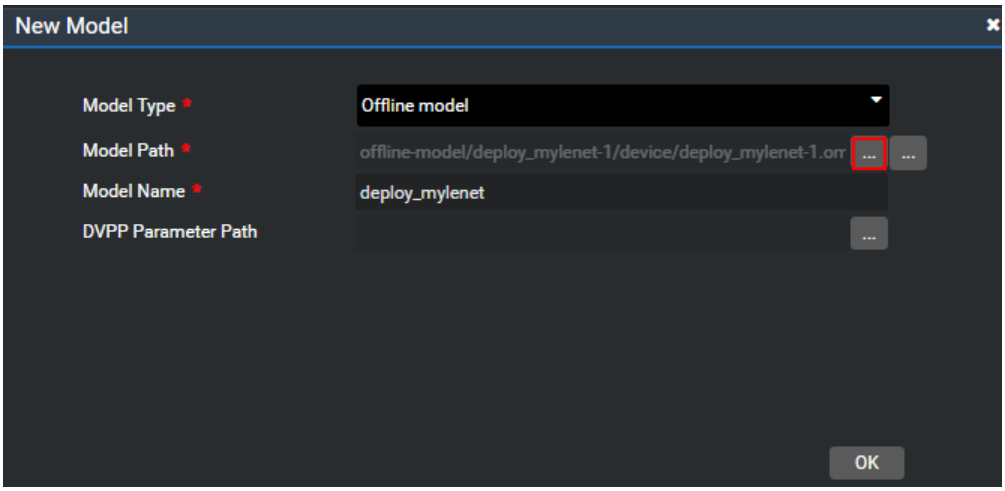
1. 选择自定义模型后面的[+]，新建自定义模型，如图2-32、图2-33所示。

图 2-32 选择模型



2. 在“Model Type”中选择“offlineModel”，单击“Device Model Path”左边的 ...，选择**2.2.5.1 模型转换**中的模型文件，将模型导入到Mind工程中，如**图2-33**所示。

图 2-33 选择转换好的模型

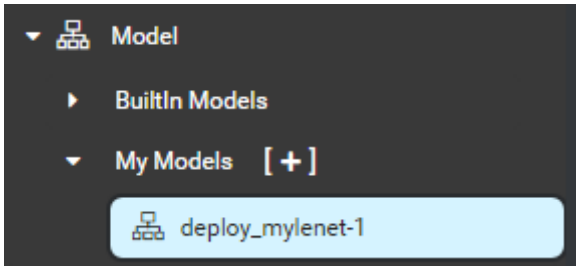


Model type选择OfflineModel。

Model File选择“my-model/deploy_mylenet-1/device/deploy_mylenet.om”文件。

3. 单击“ok”完成模型选择，如**图2-34**所示。

图 2-34 生成的自定义模型



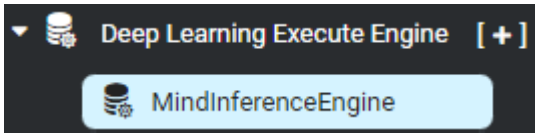
4. 将mylenet网络模型拖动到网络编排界面，如图2-35所示。

图 2-35 将自定义模型拖拽到编排界面

步骤3 PreProcess，该网络图片信息即为灰度图数据，数据预处理节点省略。

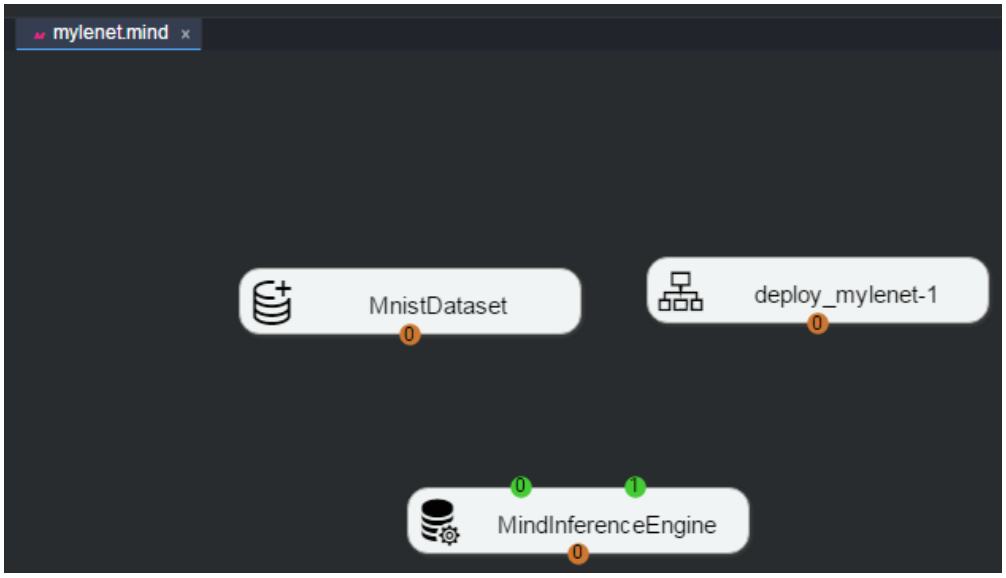
步骤4 Deep Learning Execute Engine，神经网络执行节点选择MindInferenceEngine，如图2-36所示。

图 2-36 神经网络执行节点



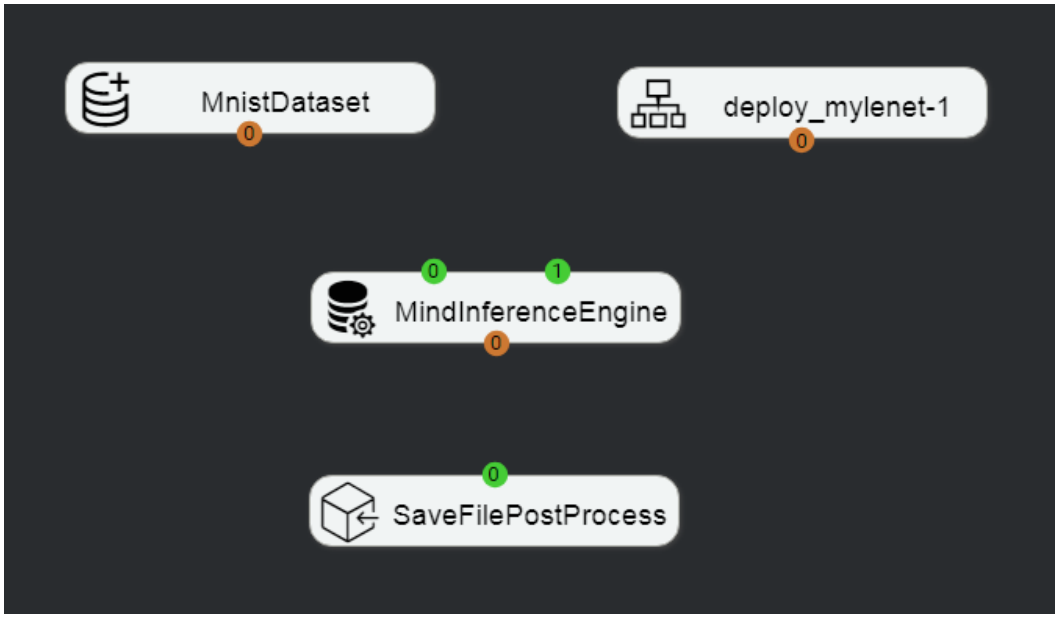
将Deep Learning Execute Engine节点拖动到网络编排界面，如图2-37所示。

图 2-37 拖拽至网络编排界面



步骤5 将“SaveFilePostProcess”节点拖动到网络编排界面，如图2-38所示。

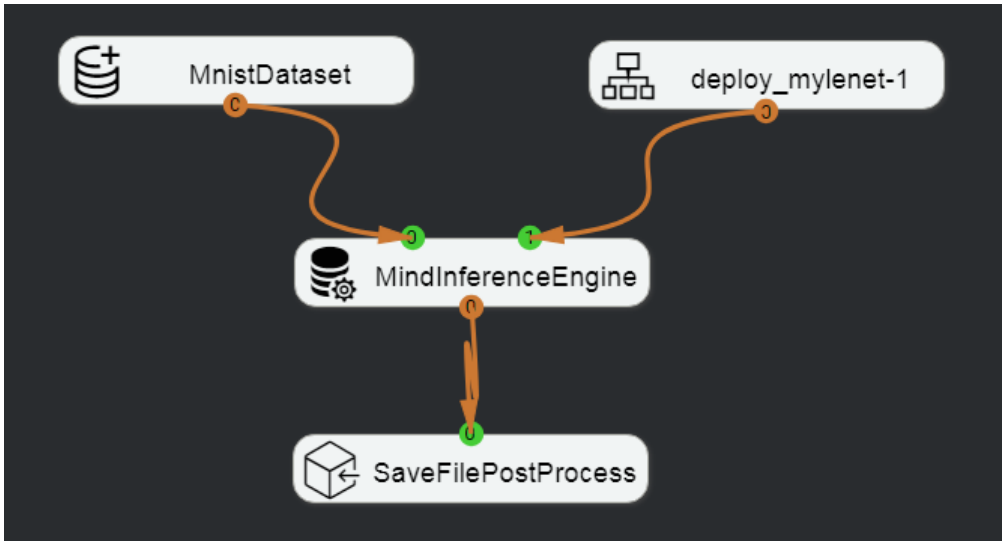
图 2-38 将 PostProcess 节点拖动到网络编排界面



步骤6 网络流程编排。

节点之间通过连线连接，如图2-39所示。

图 2-39 连接节点



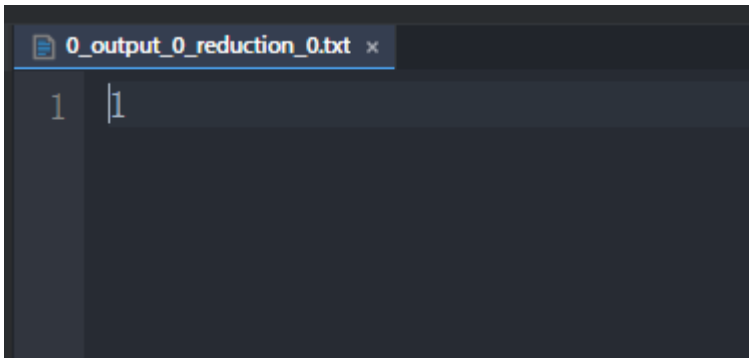
步骤7 依次执行Save、Generate、Run操作。

----结束

2.2.5.2.3 查看运行结果

通过mylenet工程下的“out/最新时间戳文件夹/SaveFilePostProcess_1/output_0”运行日志来查看运行结果，如图2-40所示。

图 2-40 查看运行结果



在lenet数字识别网络基础上，最后添加了由C++开发的reduction算子，对所有数字种类的概率进行累加并输出最终结果，最终结果概率累加值为1。

3 附录

3.1 CCE tensor format 类型

CCE tersor format类型如图3-1所示。

图 3-1 CCE tersor format 类型

```
typedef enum tagCcTensorFormat
{
    CC_TENSOR_NCHW = 0,          /**< NCHW */
    CC_TENSOR_NHWC,              /**< NHWC */
    CC_TENSOR_ND,                /**< Nd Tensor */
    CC_TENSOR_NC1HWC0,           /**< NC1HWC0 */
    CC_TENSOR_FRACTAL_Z,         /**< FRACTAL_Z */
    CC_TENSOR_NC1C0HWPAD,
    CC_TENSOR_NHWC1C0,
    CC_TENSOR_FSR_NCHW,
    CC_TENSOR_FRACTAL_DECONV,
    CC_TENSOR_C1HWNC0,
    CC_TENSOR_FRACTAL_DECONV_TRANSPOSE,
    CC_TENSOR_FRACTAL_DECONV_SP_STRIDE_TRANS,
    CC_TENSOR_RESERVED
} ccTensorFormat_t;
```

3.2 TensorDescriptor 结构

TensorDescriptor结构如图3-2所示。

图 3-2 TensorDescriptor 结构

```
// 多维数据描述
message TensorDescriptor {
    int32 format = 1;                       // 格式，取值参见CCE接口
    int32 data_type = 2;                    // 数据类型，取值参见CCE接口
    repeated int64 dim = 3;                 // 每个维度的大小
    uint32 size = 4;                        // 数据实际占用的空间大小
    bool reuse_input = 5;                   // 如果该Tensor是输出，是否直接复用输入的内存
    bool output_tensor = 7;                 // 该tensor是否是网络的输出Tensor
    DeviceType device_type = 8;
    bool input_tensor = 9;                  // 该tensor是否是网络的输入Tensor
    uint32 real_dim_cnt = 10;               // 实际维度大小
    uint32 reuse_input_index = 11;          // 复用内存的输入的序号
}
```


3.3 数据生成脚本实现

```
#coding=utf-8
'''
版权信息：华为技术有限公司， 版权所有（c） 2010-2019
功能：抽取caffe模型的指定层，及其对应的输出、输出、权重及偏置
创建：2019-1-29 16:50
'''

import caffe
import numpy as NP

def dumpData(inputData, name, fmt, data_type):
    if fmt == "binary" or fmt == "bin":
        fOutput = open(name, "wb")
        if(data_type == "float16"):
            for elem in NP.nditer(inputData, op_flags = ["readonly"]):
                fOutput.write(NP.float16(elem).tobytes())
        elif(data_type == "float32"):
            for elem in NP.nditer(inputData, op_flags = ["readonly"]):
                fOutput.write(NP.float32(elem).tobytes())
        elif(data_type == "int32"):
            for elem in NP.nditer(inputData, op_flags = ["readonly"]):
                fOutput.write(NP.int32(elem).tobytes())
        elif(data_type == "int8"):
            for elem in NP.nditer(inputData, op_flags = ["readonly"]):
                fOutput.write(NP.int8(elem).tobytes())
        elif(data_type == "uint8"):
            for elem in NP.nditer(inputData, op_flags = ["readonly"]):
                fOutput.write(NP.uint8(elem).tobytes())
    else:
        fOutput = open(name, "w")
        index = 0
        for elem in NP.nditer(inputData):
            fOutput.write("%f\t" % elem)
            index += 1
            if index % 16 == 0:
                fOutput.write("\n")


def extract_caffe_op(model, weights, extract_layer_name,
last_layer_name, img_path, data_input_shape, data_type):
    '''
    抽取caffe模型的指定层，及其对应的输出、输出、权重及偏置

    model: 模型文件
    weights: 权重文件
    extract_layer_name: 被抽取的层名
    last_layer_name:  被抽取层的上一层的层名
    img_path: 测试样本，用于前向推理
    data_input_shape: 网络数据层的输入数据shape
    data_type: 输出数据类型
    '''
    caffe.set_mode_cpu()
    net = caffe.Net(model, weights, caffe.TEST)
    print("blobs {}\nparams {}".format(net.blobs.keys(),
net.params.keys()))

    #保存extract layer的参数（如果有参数）
```

```
    if(net.params.has_key(extract_layer_name)):
        print("%s params size: %d"%(extract_layer_name,
len(net.params[extract_layer_name])))
        for idx in range(len(net.params[extract_layer_name])):
            param_file_name = "./extract_layer_param%d.bin"%(idx)
            print('============params %d================='%(idx))
            print(net.params[extract_layer_name][idx].data)
            print(net.params[extract_layer_name][idx].shape)
            with open(param_file_name, 'wb') as f:
                dumpData(net.params[extract_layer_name][idx].data,
extract_layer_name + '_param_' + idx + '.txt', "txt", data_type)
                dumpData(net.params[extract_layer_name][idx].data,
extract_layer_name + '_param_' + idx + '.bin', "bin", data_type)

    transform=caffe.io.Transformer({'data':net.blobs['data'].data.shape})

    transform.set_transpose('data',(2,0,1))
    transform.set_raw_scale('data',255)
    #transform.set_channel_swap('data',(2,1,0))

    #加载图像文件
    image=caffe.io.load_image(img_path, color = False)
    transformed_image=transform.preprocess('data',image)

    # 把警告过transform.preprocess处理过的图片加载到内存
    net.blobs['data'].data[...]=transformed_image
    #把加载到的图片缩放到固定的大小
    net.blobs['data'].reshape(*data_input_shape)
    net.forward()

    fea_in=net.blobs[last_layer_name].data   #提取上一层数据（特征）
    print('============extracted layer input==================')
    print(fea_in)
    print(fea_in.shape)

    fea_out=net.blobs[extract_layer_name].data   #提取当前层数据（特征）
    print('============extracted layer output==================')
    print(fea_out)
    print(fea_out.shape)

    dumpData(fea_in, extract_layer_name+ '_input.txt', "txt", data_type)
    dumpData(fea_in, extract_layer_name+ '_input.data', "bin", data_type)

    dumpData(fea_out, extract_layer_name+ '_output.txt', "txt",
data_type)
    dumpData(fea_out, extract_layer_name+ '_output.data', "bin",
data_type)

if __name__ == "__main__":
    model = './deploy_mylenet-1.prototxt'
    weights = './mylenet-1.caffemodel'
    extract_layer_name = 'reduction'
    last_layer_name = 'prob'
    img_path = './1.jpg'
    data_type = "float16"
    data_input_shape = (1,1,28,28)
    extract_caffe_op(model, weights, extract_layer_name,
last_layer_name, img_path, data_input_shape, data_type)
```

