



**Ascend 310**

**V100R001**

# **TE 自定义算子开发指导**

文档版本 01

发布日期 2019-03-12

版权所有 © 华为技术有限公司 2019。保留一切权利。

非经本公司书面许可，任何单位和个人不得擅自摘抄、复制本文档内容的部分或全部，并不得以任何形式传播。

## 商标声明



HUAWEI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](mailto:support@huawei.com)

客户服务电话：4008302118

## 目录

1 简介.....	1
2 场景说明.....	2
3 声明.....	3
4 自定义算子开发实例.....	4
4.1 操作流程.....	4
4.2 创建 TE 算子开发工程.....	5
4.2.1 直接新建算子开发工程.....	5
4.2.2 模型转化失败进入算子开发工程.....	10
4.3 自定义算子开发.....	13
4.4 算子插件开发.....	15
4.5 编译 TE 算子工程.....	19
4.6 运行 TE 算子工程.....	20
4.6.1 构造输入数据文件.....	20
4.6.2 配置运行参数.....	21
4.6.3 运行单算子.....	25
4.7 加载插件转换模型.....	28
5 自定义算子整网络执行.....	31
5.1 创建 Mind 工程.....	31
5.2 导入网络模型.....	33
5.3 编排流程.....	34
5.4 编译运行.....	37
6 算子开发参考.....	39
6.1 reduction 算子介绍.....	39
6.1.1 reduction 算子定义.....	39
6.1.2 reduction 算子代码解析.....	40
6.1.2.1 导入 tvn 模块.....	40
6.1.2.2 声明输入.....	40
6.1.2.3 计算过程.....	42
6.1.2.4 算子编译.....	42
6.1.3 算子插件注册代码解析.....	43
6.2 sign 算子介绍.....	45

6.2.1 sign 算子定义.....	45
6.2.2 算子实现代码解析.....	46
6.2.2.1 导入模块.....	46
6.2.2.2 声明输入.....	46
6.2.2.3 计算过程.....	48
6.2.2.4 算子编译.....	49
<b>7 算子插件开发函数参考.....</b>	<b>50</b>
7.1 相关函数列表.....	50
7.1.1 非 IR 方式.....	50
7.1.1.1 ParseParams 函数.....	50
7.1.1.2 ParseWeights 函数.....	51
7.1.1.3 ConvertWeight 函数.....	51
7.1.1.4 ConvertShape 函数.....	52
7.1.1.5 GetOutputDesc 函数.....	52
7.1.1.6 BuildTvmBinFile 函数.....	52
7.1.1.7 算子构造函数注册宏.....	53
7.1.1.8 调用示例.....	53
7.1.2 IR 方式.....	53
7.1.2.1 Attr 函数.....	53
7.1.2.2 ParseParams 函数.....	54
7.1.2.3 GetOutputDesc 函数.....	54
7.1.2.4 BuildTvmBinFile 函数.....	55
7.1.2.5 算子参数规格注册宏.....	55
7.1.2.6 算子解析函数注册宏.....	56
7.1.2.7 算子构造函数注册宏.....	56
7.1.2.8 调用示例.....	56
7.1.2.8.1 用户自定义类.....	56
7.1.2.8.2 调用 Attr 函数.....	57
7.1.2.8.3 自定义 ParseParams 函数.....	57
7.1.2.8.4 重写 GetOutputDesc 函数和 BuildTvmBinFile 函数.....	58
7.1.2.8.5 算子注册.....	60
7.2 框架内置算子列表.....	60

# 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支持的模型，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，则未实现的算子就需要用户进行自定义，Mind Studio提供将自定义算子加入到算子库的功能，使得模型转换过程可以正常进行。

在Mind Studio中提供了TE（Tensor Engine）算子开发框架，可以开发自定义算子。TE是基于TVM（Tensor Virtual Machine）的自定义算子开发框架，提供了基于Python语法的DSL语言供开发者开发编写自定义算子。

本文以定制的Lenet-5网络模型为例，描述自定义Reduction算子，并通过加载算子插件完成模型转化，最终执行流程编排实现数字概率累加验证功能的端到端流程。

## 2 场景说明

---

Tensor Engine自定义算子开发有如下两种应用场景。

- 开发人员直接创建Tensor Engine工程开发算子，然后将开发好的算子插件加载到模型中进行模型转化，将转化好的模型应用于AI程序开发中。
- 开发人员将训练好的网络模型导入Mind Studio中进行模型转化，因为模型中有系统不支持的算子会导致模型转化失败。此时开发人员可以通过模型转化失败界面进入自定义算子开发的界面进行算子开发，然后再将开发好的算子插件加载到模型中重新进行模型转化，将转化好的模型应用于AI程序开发中。

# 3 声明

---

- 请勿修改框架内置算子的实现（路径：`ddk/include/inc/custom`下的所有文件），否则将可能导致未知风险，例如：系统启动失败、模型无法正常转换等。
- 客户在开发算子插件时需要对自己的源代码负责，避免植入后门。

# 4 自定义算子开发实例

本章节描述reduction算子的开发及通过加载此算子插件进行模型转化的操作过程。

本示例使用定制的Lenet-5的网络模型，实现图片分类结果的reduction功能。

网络模型对应的模型文件在DDK包中，Mind Studio后台路径为Mind Studio安装用户下的：“\$HOME/tools/che/ddk/ddk/sample/customop/customop\_caffe\_demo/model”。请获取此网络模型到开发者本地PC机中。

## 说明

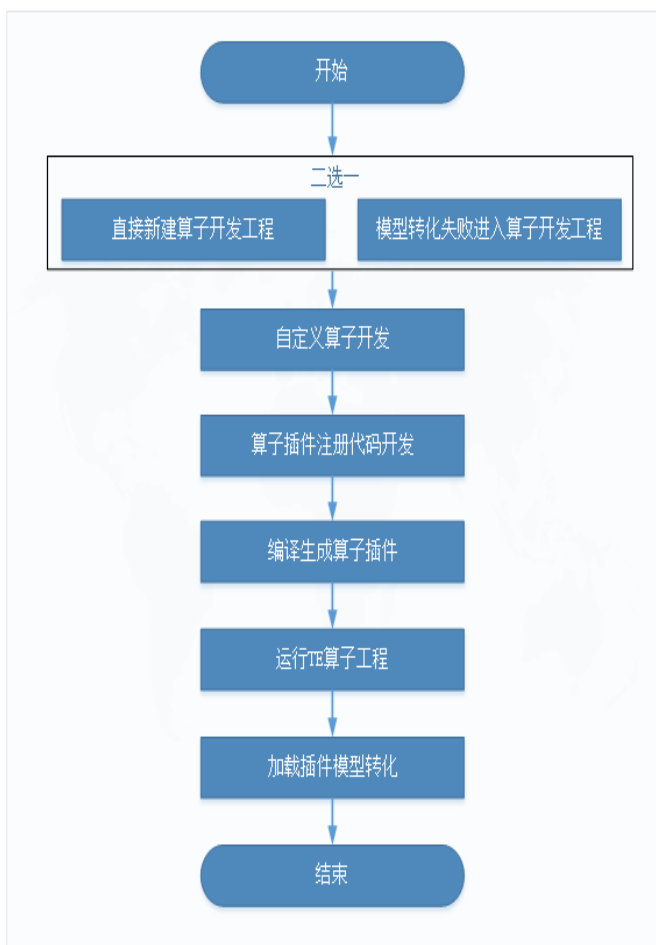
*\$HOME/tools*为DDK的默认安装路径。

## 4.1 操作流程

算子开发流程如[图4-1](#)所示。



图 4-1 自定义算子开发



主要流程如下：

1. 创建算子工程。
2. 自定义未实现的算子。
3. 开发算子插件，将算子注册到Framework中。
4. 用户完成开发后，需要重新编译插件，生成该自定义算子的\*.so插件文件。
5. 再次进行模型转换时，通过加载\*.so插件文件，才能识别自定义的算子，进而完成模型转换。

## 4.2 创建 TE 算子开发工程

创建算子开发工程，可以直接创建Tensor Engine类型工程，也可以通过模型转化 > 转化失败进而进入算子创建工程界面。

### 4.2.1 直接新建算子开发工程

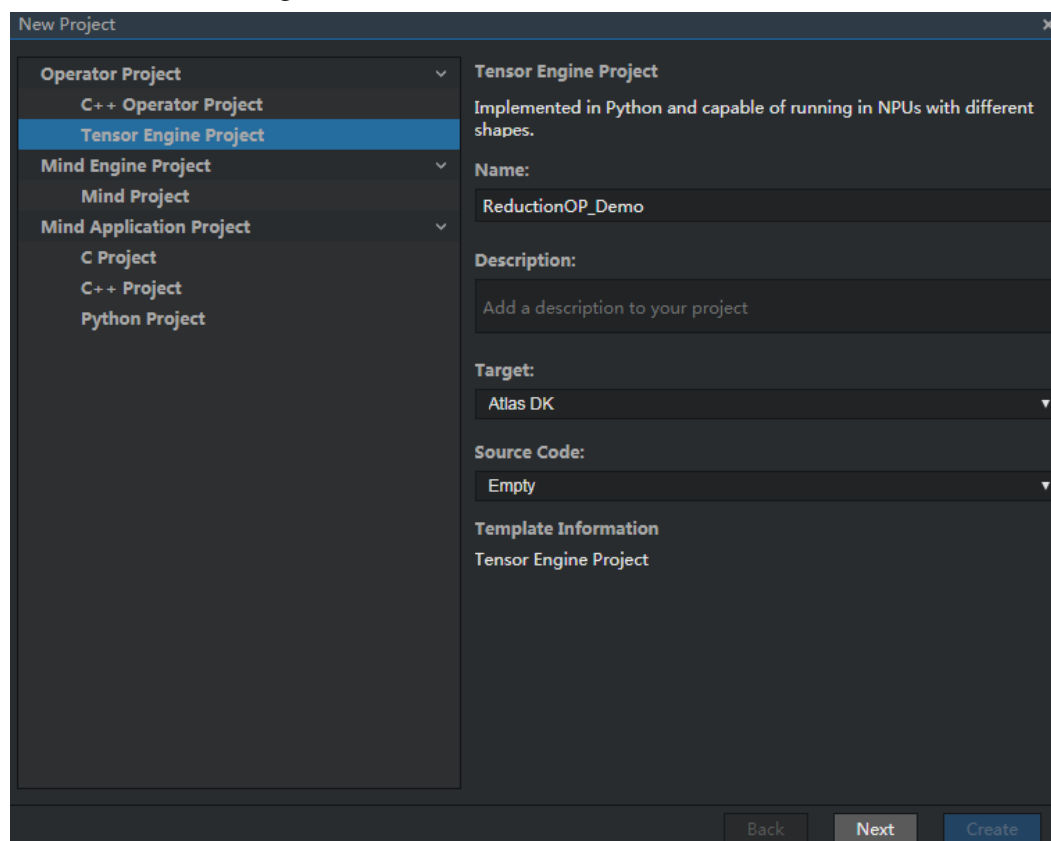
开发者知道所用模型中有不支持的算子，则可以参照本章节直接通过新建算子开发工程进行算子的开发。

**步骤1** 在菜单栏依次选择“File > New > New Project”，新建Tensor Engine工程。

选择“Tensor Engine Project”类型，填写“Name”，选择工程运行的“Target”，其中ASIC表示运行目标是EVB单板或者PCIe单板；Atlas DK表示运行目标是开发板。

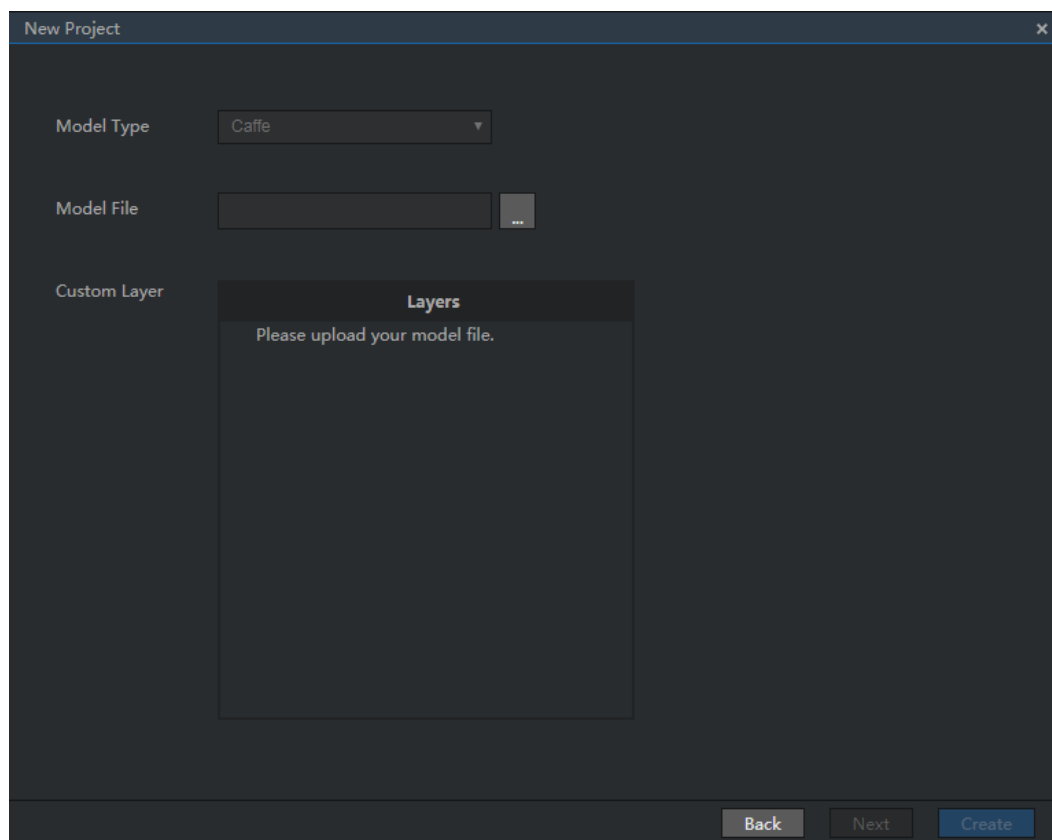
工程创建示例如图4-2所示。

图 4-2 创建 Tensor Engine 工程



**步骤2** 单击“Next”，弹出如图4-3所示界面。

图 4-3 选择模型文件




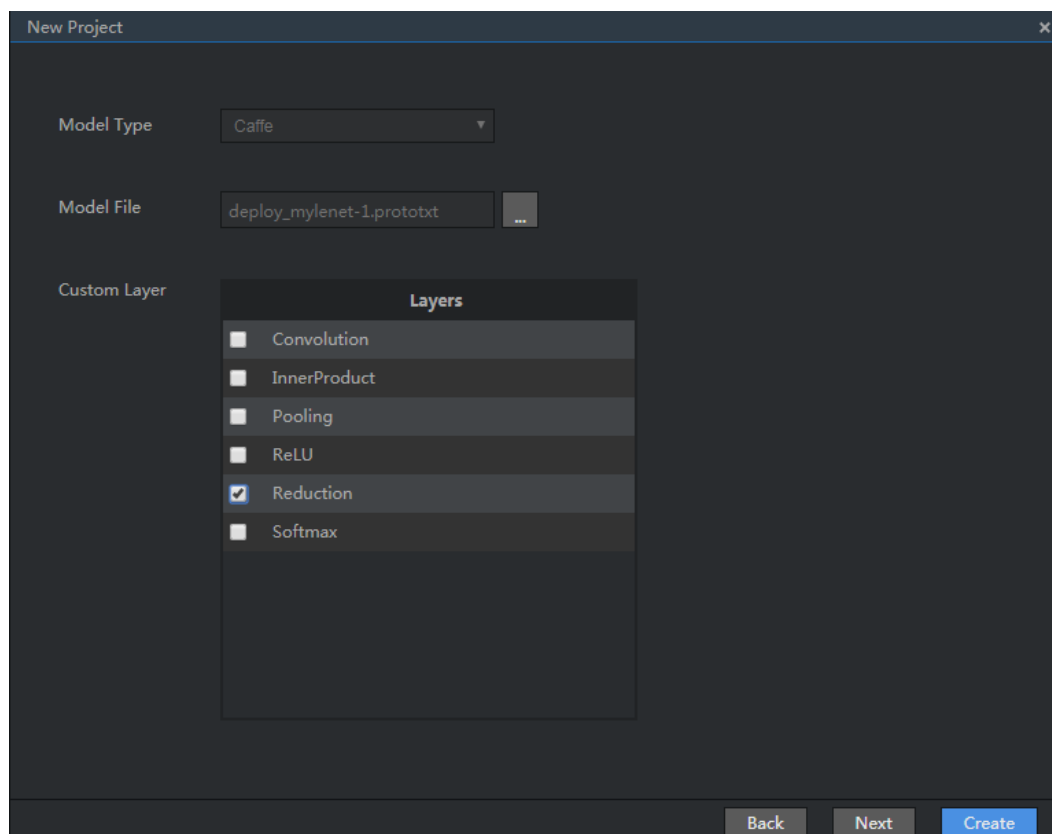
**步骤3** 单击“Model File”右侧，选择Caffe模型的prototxt文件，在“Custom Layer”区域会列出模型中的每一层算子，选择“Reduction”算子，如[图4-4](#)所示。如有用户想选择多个算子，则选择图中的多个层级。

图 4-4 选择模型文件及算子

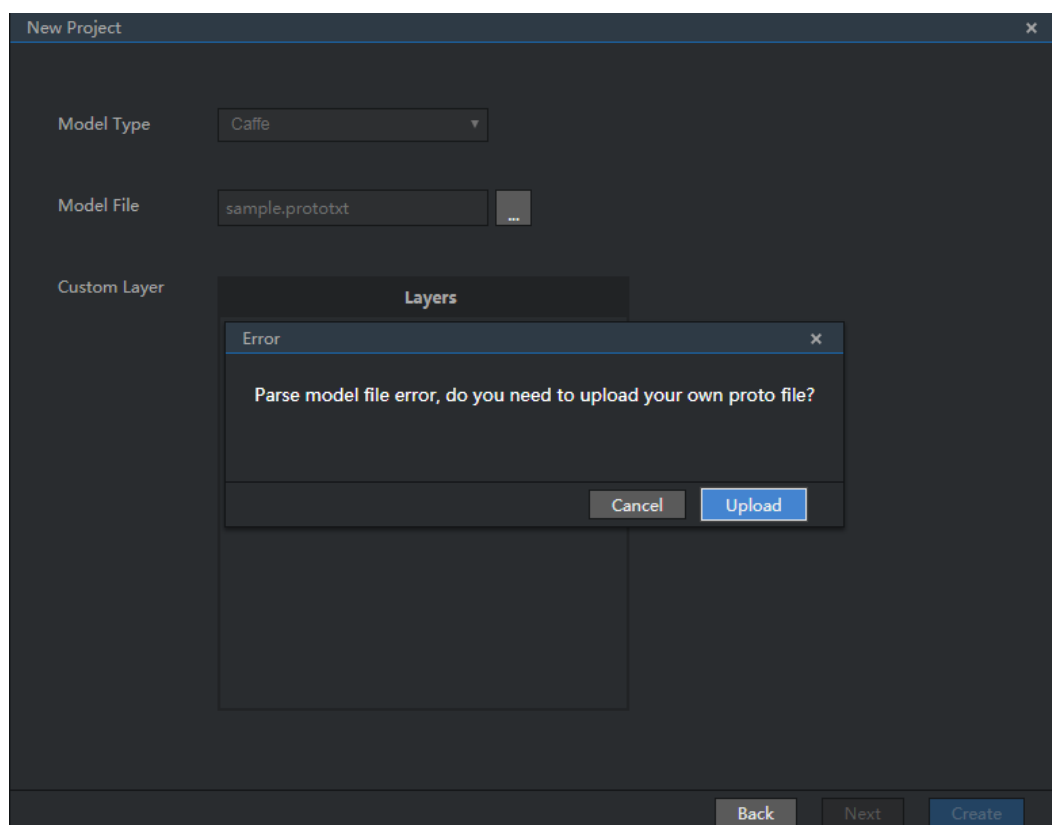


说明

本示例中使用的模型文件在DDK包中，路径为Mind Studio安装用户下的“\$HOME/tools/che/ddk/ddk/sample/customop/customop\_caffe\_demo/model”。

如果上传的prototxt中存在caffe.proto中未定义的算子（系统内置的caffe.proto无法解析当前模型，系统内置的caffe.proto的存储路径为：“\$HOME/tools/che/ddk/ddk/include/inc/custom/proto/caffe/caffe.proto”），会导致第一次解析算子失败，示例界面如下所示。

图 4-5 第一次解析算子失败

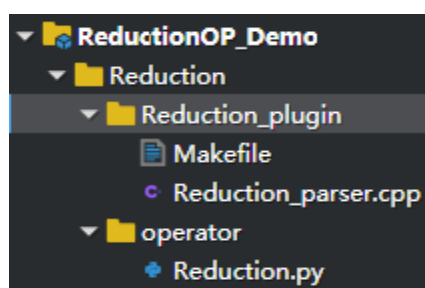


单击“Upload”，上传自定义的caffe.proto，系统会把所需字段合并到内置的caffe.proto中。

**步骤4** 单击“Create”，完成算子工程的创建。

工程创建后，在Projects区域可以看到创建的工程，如图4-6所示。

图 4-6 算子工程生成的文件



其中：

- operator目录下的python文件，对应自定义算子。
- Reduction\_plugin目录下包含一个cpp文件和一个Makefile文件，对应自定义算子插件文件和编译makefile文件。

我们要对“Reduction.py”文件和“Reduction\_parser.cpp”文件进行编辑，进行自定义算子和相应插件的开发。

----结束

## 4.2.2 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。

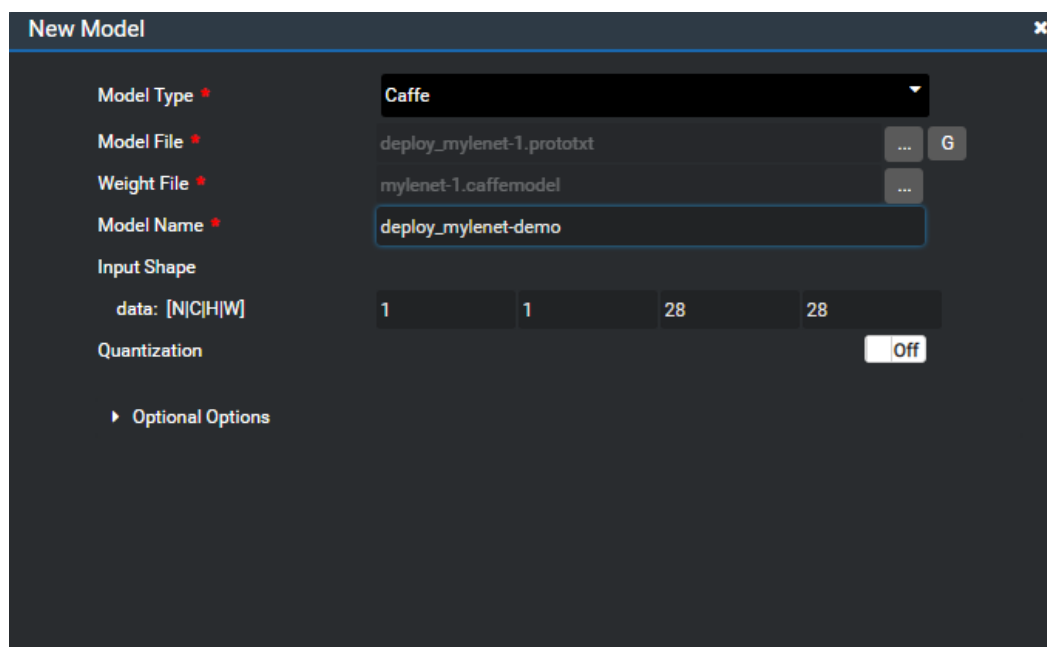
**步骤1** 创建需要使用模型文件进行AI工程开发的Mind工程，详情步骤请参见[5.1 创建Mind工程](#)。

**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“New Model”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4-7](#)所示。

图 4-7 选择模型文件

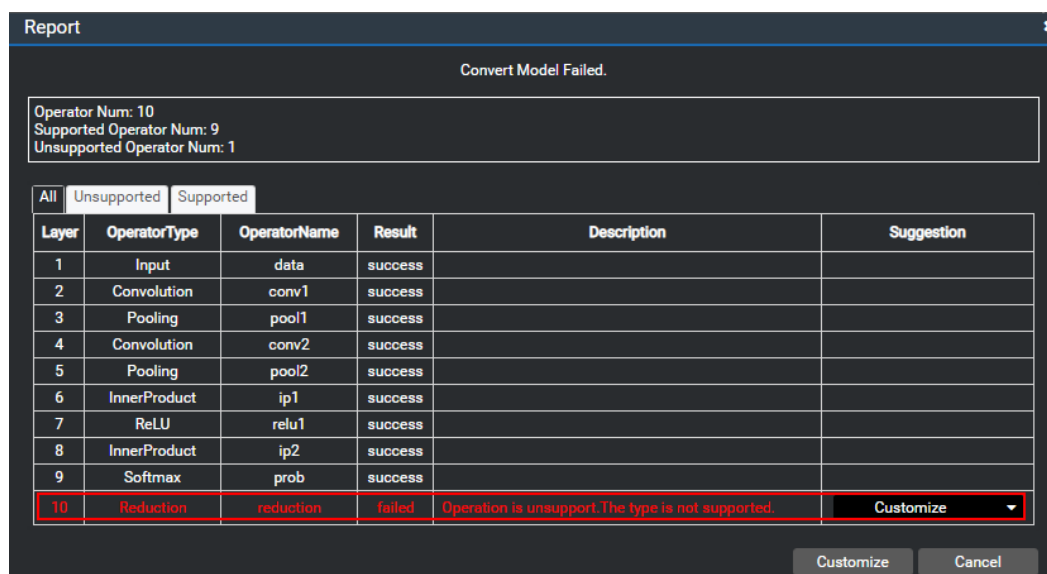


- “Model Name”自动填充模型文件的名称，用户可以在选择模型文件后自行修改为想要的名称。
- 工具会解析模型文件获取模型的默认“input shape”。

**步骤4** 单击“OK”，进行模型转化。

界面弹出如[图4-8](#)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。

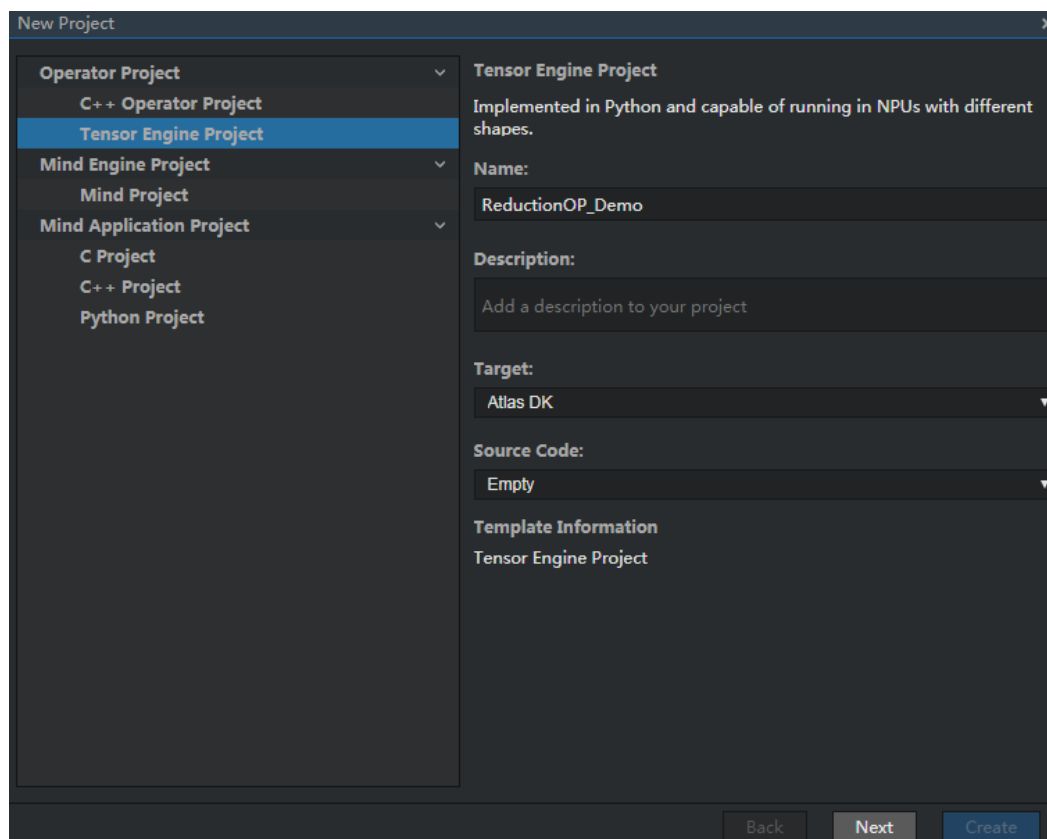
图 4-8 转化失败报告



**步骤5** 单击“Customize”，创建自定义算子工程。

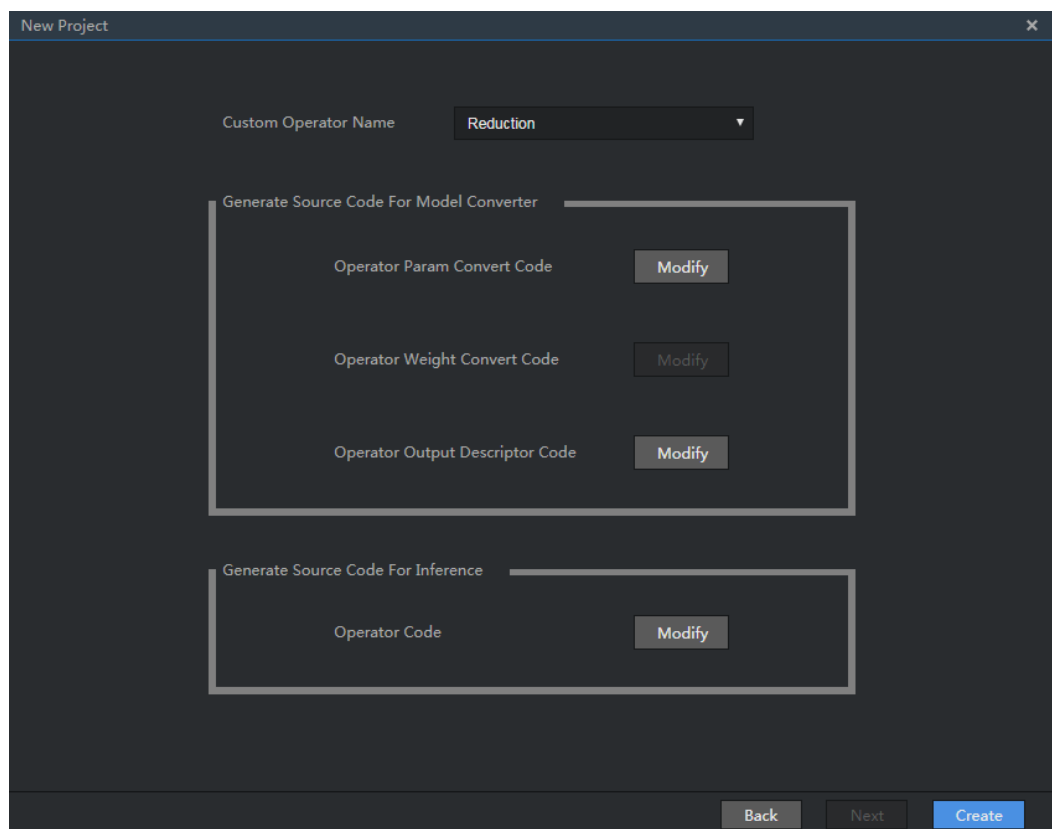
选择“Tensor Engine Project”类型，填写“Name”，设置“Target”，如图4-9所示。

图 4-9 创建 Tensor Engine 工程




**步骤6** 单击“Next”，进入代码编辑界面，如图4-10所示。

图 4-10 代码快速编辑界面

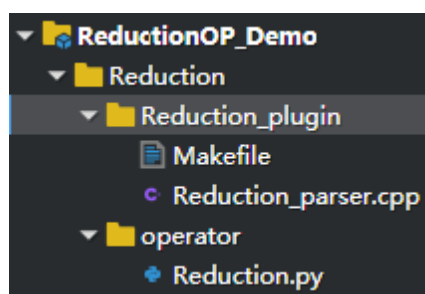


**步骤7** 单击“Create”，完成算子工程的创建。

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4-11所示。

图 4-11 算子工程生成的文件



其中：

- operator目录下的python文件，对应自定义算子。
- Reduction\_plugin目录下包含一个cpp文件和一个Makefile文件，对应自定义算子插件文件和编译makefile文件。

我们要对“Reduction.py”文件和“Reduction\_parser.cpp”文件进行编辑，进行自定义算子和相应的插件的开发。

----结束



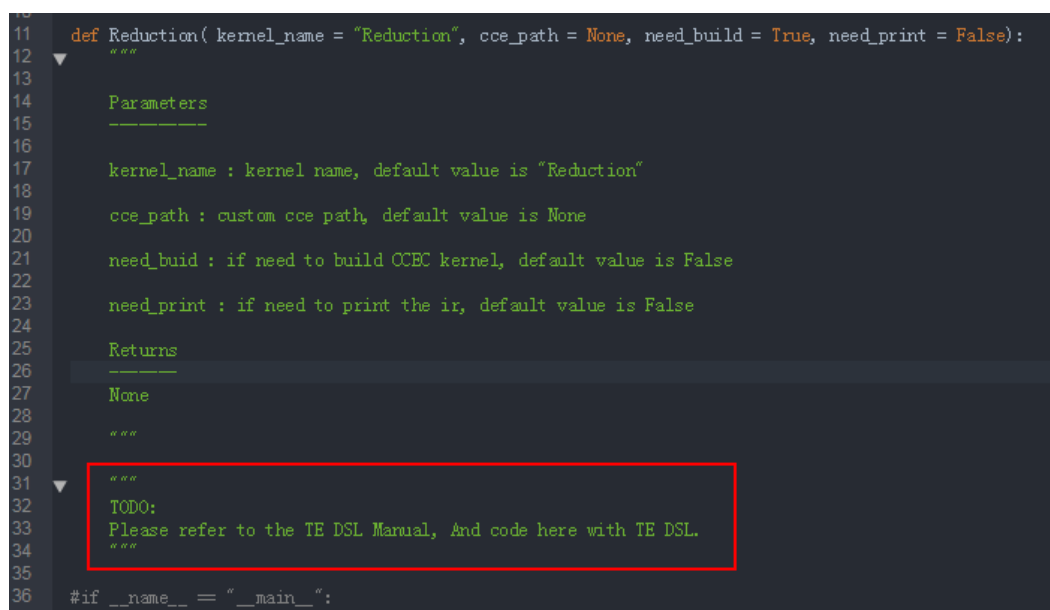
## 4.3 自定义算子开发

### 说明

支持自定义算子的输入输出数据类型为：“float16”和“float32”。

**步骤1** 双击打开算子工程中创建的Reduction.py文件，需要在本文中通过python代码实现相应的算子，如图4-12所示。

图 4-12 Python 文件模板



上图红框中表示需要使用TE的API接口，用DSL描述此算子。

**步骤2** 实现Reduction算子，同时需要根据数据维度、类型和算子参数更改实际入参。

如下代码是已经实现的Reduction算子的完整代码，可直接拷贝至Reduction.py文件中。Reduction算子介绍及算子实现解析请参见[6.1 reduction算子介绍](#)。

附件Reduction.zip为Reduction算子的实现代码。

### 说明

拷贝后请保持代码格式，否则编译可能会出错。

```

#coding=utf-8
"""
generated by Mind studio
"""
import te.lang.cce
from te import tvm
from topi import generic
from topi.cce import util

def Reduction(shape, dtype, axis, op, coeff, kernel_name="Reduction",
              need_build=True, need_print=False):
    """
    Reduce a tensor on a certain axis, and scale output with coeff
    Parameters
    -----

```

```
shape : shape of data
dtype : source data type, only support float16, float32
axis : the first axis to reduce, may be negative to index from the end (e.g., -1 for the last
axis).

    If axis == 0, the output Blob always has the empty shape (count 1), performing
reduction across the entire input.
    op : can only be one of "SUM, ASUM (sum of abs), SUMSQ (sum of sqr), MEAN"
coeff : scale for output
kernel_name : cce kernel name, default value is "cce_reductionLayer"
need_buid : if need to build CCEC kernel, default value is False
need_print : if need to print the ir, default value is False
Returns
-----
None
"""

#basic check
util.check_shape_rule(shape)
check_list = ["float16", "float32"]
if not (dtype.lower() in check_list):
    raise RuntimeError("Reduction only support %s while dtype is %s" % (",".join(check_list),
dtype))

ReductionOp = ("SUM", "ASUM", "SUMSQ", "MEAN")

# axis parameter check
if type(axis) != int:
    raise RuntimeError("type of axis value should be int")
if axis >= len(shape) or axis < -len(shape):
    raise RuntimeError(
        "input axis is out of range, axis value can be from %d to %d" % (-len(shape),
len(shape) - 1))
# op parameter check
if op not in ReductionOp:
    raise RuntimeError("op can only be one of SUM, ASUM, SUMSQ , MEAN")
# coeff parameter check
if type(coeff) != int and type(coeff) != float:
    raise RuntimeError("coeff must be a value")
# Preprocess
if axis < 0:
    axis = len(shape) + axis
shape = list(shape)
shapel = shape[:axis] + [reduce(lambda x, y: x * y, shape[axis:])]
inp_dtype = dtype.lower()
# define input
data = tvm.placeholder(shapel, name="data_input", dtype=inp_dtype)
# computational process
with tvm.target.cce():
    if op == "ASUM":
        data_tmp_input = te.lang.cce.vabs(data)
        cof = coeff
        tmp = te.lang.cce.vmults(data_tmp_input, cof)
    elif op == "SUMSQ":
        data_tmp_input = te.lang.cce.vmul(data, data)
        cof = coeff
        tmp = te.lang.cce.vmults(data_tmp_input, cof)
    elif op == "MEAN":
        size = shapel[-1]
        cof = float(coeff) * (size ** (-0.5))
        tmp = te.lang.cce.vmults(data, cof)
    elif op == "SUM":
        cof = coeff
        data_tmp_input = te.lang.cce.vmults(data, cof)
        tmp = data_tmp_input

    res_tmp = te.lang.cce.sum(tmp, axis=axis)
    res = te.lang.cce.cast_to(res_tmp, inp_dtype, f1628IntegerFlag = True)

    if op == "MEAN":
        size = shapel[-1]
```

```

    sqrt_size = size ** (-0.5)
    res = te.lang.cce.vmul(s(res_tmp, sqrt_size)

    sch = generic.auto_schedule(res)

    config = {"print_ir": need_print,
              "need_build": need_build,
              "name": kernel_name,
              "tensor_list": [data, res]}
    te.lang.cce.cce_build_code(sch, config)
if __name__ == "__main__":
    Reduction((2, 3, 4), "float16", 1, "SUM", coeff = 2, kernel_name = "Reduction")

```

----结束

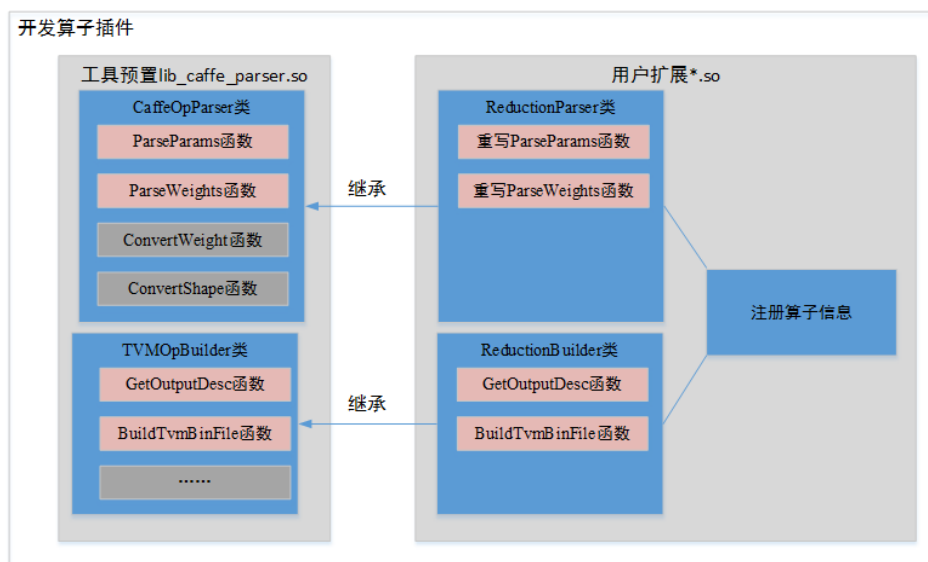
## 4.4 算子插件开发

### 原理介绍

在开发算子插件时，支持以下两种方式。

- 非IR（intermediate representation，中介码）方式
  - a. 自定义类（如：ReductionParser类），继承CaffeTVMOpParser基类，重写ParseParams、ParseWeights函数，实现caffe模型与自定义模型中算子参数、权值参数的映射。
  - b. 自定义类（如：ReductionBuilder类），继承TVMOpBuilder基类，重写GetOutputDesc函数（获取算子的输出描述）、BuildTvmBinFile函数（构建算子的二进制文件）。
  - c. 调用DOMI\_REGISTER\_OP完成算子注册。

图 4-13 算子插件开发原理图



- IR方式
  - a. 自定义类（如：CustomOperator类），继承Operator类，调用Attr函数（将属性值写入到离线模型）、重写ParseParams函数（完成caffe模型参数和权值的转换，将结果填到中自定义类中）。

- b. 自定义类（如：caffe\_reduction\_layerOpBuilder类），继承TVMOpBuilder基类，重写GetOutputDesc函数（获取算子的输出描述）、BuildTvmBinFile函数（构建算子的二进制文件）。
- c. 调用DOMI\_OP\_SCHEMA注册算子参数规格，调用DOMI\_REGISTER\_CAFFE\_PARSER注册算子解析函数，调用DOMI\_REGISTER\_OP完成算子注册。

## 算子插件注册代码开发

修改算子工程文件中的Reduction\_parser.cpp文件，实现此插件的代码开发（本示例使用非IR方式开发算子插件，IR方式请参见7.1.2 IR方式）。

修改好的代码示例如下所示，详细代码解析请参见6.1.3 算子插件注册代码解析。

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

### 说明

拷贝后请保持代码格式，否则编译可能会出错。

如下代码中的sys.path.append('/projects/ReductionOP\_Demo/Reduction/operator')中的ReductionOP\_Demo请修改为您的实际工程名称。

```
#include <Python.h>
#include "omg/register.h"
#include "proto/caffe/caffe.pb.h"
#include "omg/parser/caffe/caffe_tvm_op_parser.h"
#include "omg/model/op_builder/tvm_op_builder.h"
#include <string>
#include <iostream>
namespace domi
{
    // #### ReductionParser是解析器的类名，可以随意指定，只要最后注册时使用此类名即可
    class ReductionParser : public CaffeTVMOpParser
    {
    public:
        ReductionParser()
        {
        }

        ~ReductionParser() {}

        // ##将Caffe模型prototxt中的算子参数，映射到离线模型
        virtual Status ParseParams(const Message* op_src, OpDef* op_dest) override
        {
            const caffe::LayerParameter* layer =
                dynamic_cast<const caffe::LayerParameter*>(op_src);

            // #### 输入算子参数合法性校验
            if (nullptr == layer)
            {
                printf("Dynamic cast op_src to LayerParameter failed\n");
                return FAILED;
            }
            // #### 算子操作类型与其字符串对应的MAP
            std::map<::caffe::ReductionParameter_ReductionOp, std::string> operation_map = {
                { caffe::ReductionParameter_ReductionOp_SUM, "SUM" },
                { caffe::ReductionParameter_ReductionOp_ASUM, "ASUM" },
                { caffe::ReductionParameter_ReductionOp_SUMSQ, "SUMSQ" },
                { caffe::ReductionParameter_ReductionOp_MEAN, "MEAN" },
            };
            // #### 获取算子参数
            const ::caffe::ReductionParameter& param = layer->reduction_param();

            // #### 设置参数至op_dest中
            AddOpAttr("operation", operation_map[param.operation()], op_dest);
        }
    };
}
```

```
        AddOpAttr("axis", (int32_t)param.axis(), op_dest);
        AddOpAttr("coeff", param.coeff(), op_dest);
        return SUCCESS;
    }
    virtual Status ParseWeights(const Message* op_src, OpDef* op_dest) override
    {
        // TODO: parse weights
        return SUCCESS;
    }
};

// ##### ReductionBuilder是构建器的类名，可以随意指定，只要最后注册时使用此类名即可
class ReductionBuilder : public TVMOpBuilder
{
public:
    using TVMOpBuilder::TVMOpBuilder;

    ~ReductionBuilder()
    {
    }

    // ##### 获取输出tensor描述的处理函数
    virtual Status GetOutputDesc(vector<TensorDescriptor>& v_output_desc) override
    {
        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
        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;
        }
        v_output_desc[0].set_dim(axis, 1);

        return SUCCESS;
    }
    virtual Status BuildTvmBinFile(TVMBinInfo& tvm_bin_info) override //#####构算子二进制文件
    {
        // 1. call tvm python api to build bin files
        // 2. set path to TVMBinInfo tvm_bin_info
        PyObject *pModule = NULL;
        PyObject *pFunc = NULL;
        PyObject *pArg = NULL;
        PyObject *result = NULL;

        // ##### python初始化
        Py_Initialize();

        // ##### 导入topi.cce模块
        PyRun_SimpleString("import sys");
        PyRun_SimpleString("sys.path.append('/projects/ReductionOP_Demo/Reduction/operator')");
        pModule = PyImport_ImportModule("Reduction");
        if (!pModule)
        {
            PyErr_Print();
            printf("import topi.cce failed!\n");
            return FAILED;
        }
        std::string operation;
        int32_t axis = -1;
```

```
float coeff = 1;
// ##### 解析operation参数
if (!GetOpAttr("operation", &operation, op_def_))
{
    // ##### 增加异常处理、可维护信息
    printf("GetOpAttr operation failed!\n");
}
// ##### 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
if (axis < 0) axis -= 2;

// ##### 解析coeff参数
if (!GetOpAttr("coeff", &coeff, op_def_))
{
    printf("GetOpAttr coeff failed!\n");
}
// ##### 解析输入tensor描述
TensorDescriptor input_desc = op_def_>input_desc(0);

// ##### 解析输入shape值, 校验其大小是否为4
if (input_desc.dim_size() != 4)
{
    printf("The shape size is %d, which is not 4!", input_desc.dim_size());
    return FAILED;
}
// ##### 调用Python接口生成Tvm算子
pFunc = PyObject_GetAttrString(pModule, "Reduction");
pArg = Py_BuildValue(
    "(i,i,i,i), s, i, s, f, s",
    input_desc.dim(0), input_desc.dim(1), input_desc.dim(2), input_desc.dim(3), "float16",
    axis, operation.c_str(), coeff, "cce_reductionLayer_1_10_1_1_float16_3_SUMSQ_1_0");

PyEval_CallObject(pFunc, pArg);
// ##### 结束python调用
Py_Finalize();
// ##### 设置算子json文件路径
tvm_bin_info.json_file_path = "./kernel_meta/
cce_reductionLayer_1_10_1_1_float16_3_SUMSQ_1_0.json";
return SUCCESS;
}

virtual Status TransWeights(size_t& mem_offset) override
{
    // trans weights and calculate weights memory size
    return SUCCESS;
}
};

DOMI_REGISTER_OP("caffe_reduction_layer2") // caffe_reduction_layer2是算
子在davinci模型的类型名, 可以随意指定, 不可与已有类型名重复, 区分大小写
    .FrameworkType(CAFFE) // 算子归属, 枚举类型, CAFFE,
TENSORFLOW // Reduction表示该算子在caffe框
    .OriginOpType("Reduction") // 架中的类型名
    .ParserCreatorFn(PARSER_FN(ReductionParser)) // ReductionParser对应上面的类名
    .BuilderCreatorFn(BUILDER_FN(ReductionBuilder)) // ReductionBuilder对应上面的类名
    .ImplType(ImplType::TVM); // 实现类型, 当前只支持TVM
} // namespace domi
```

算子插件开发过程中涉及到的函数请参考[7 算子插件开发函数参考](#)。

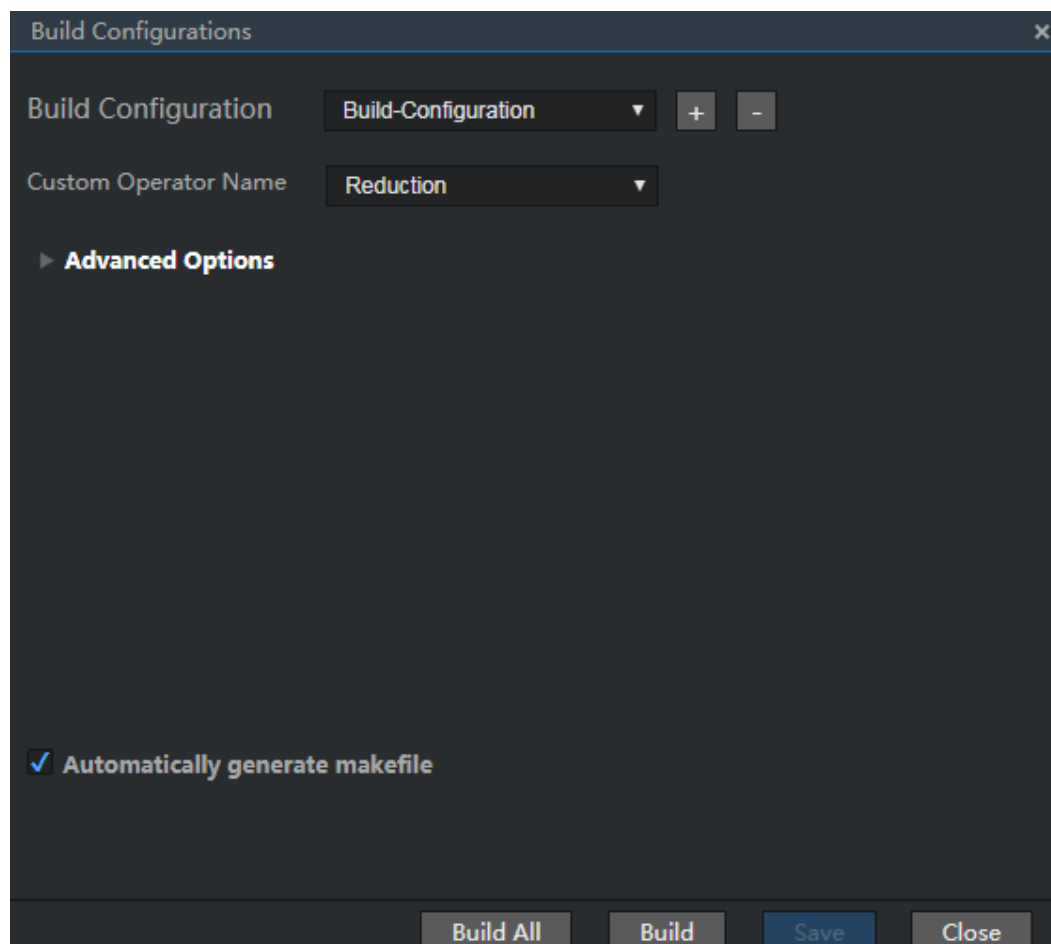
## 4.5 编译 TE 算子工程

编译自定义算子工程，生成自定义算子插件。

**步骤1** 选中创建的算子工程如`ReductionOP_Demo`，单击右键选择“Edit Build Configuration”。

弹出编译配置窗口，如图4-14所示。

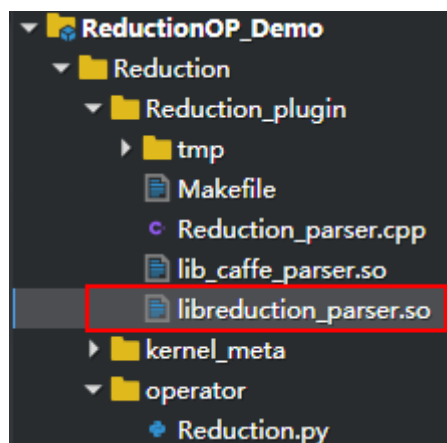
图 4-14 编译配置



**步骤2** 单击“Build”进行编译。

**步骤3** 编译运行日志提示编译完成后，在工程中会生成对应插件的.so文件，如图4-15所示。

图 4-15 编译后的结果



#### 说明

\*.so即为生成的自定义算子插件。

----结束

## 4.6 运行 TE 算子工程

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。

### 4.6.1 构造输入数据文件

运行单算子需要构造算子输入数据，并以二进制格式保存到算子工程下。

本章节使用DDK自带的DataGen.py生成Reduction算子的样例数据，操作步骤如下。

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

**步骤2** 将DDK中tvm样例工程中的DataGen.py脚本拷贝到算子工程目录下。

```
cp $HOME/tools/che/ddk/ddk/sample/tvm/DataGen.py /projects/ReductionOP_Demo
```

#### 说明

\$HOME/tools为DDK的默认安装路径。

**步骤3** 在算子工程目录下执行DataGen.py脚本，生成样例数据。

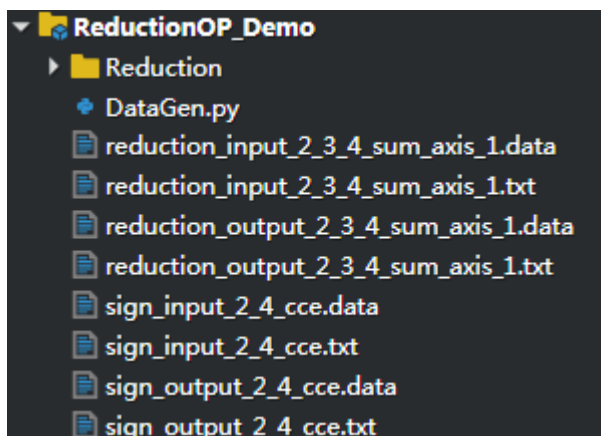
```
cd /projects/ReductionOP_Demo
```

```
python DataGen.py
```

在算子工程中会生成如图4-16所示数据文件。



图 4-16 样例数据生成



数据文件说明如表4-1所示。

表 4-1 数据文件说明

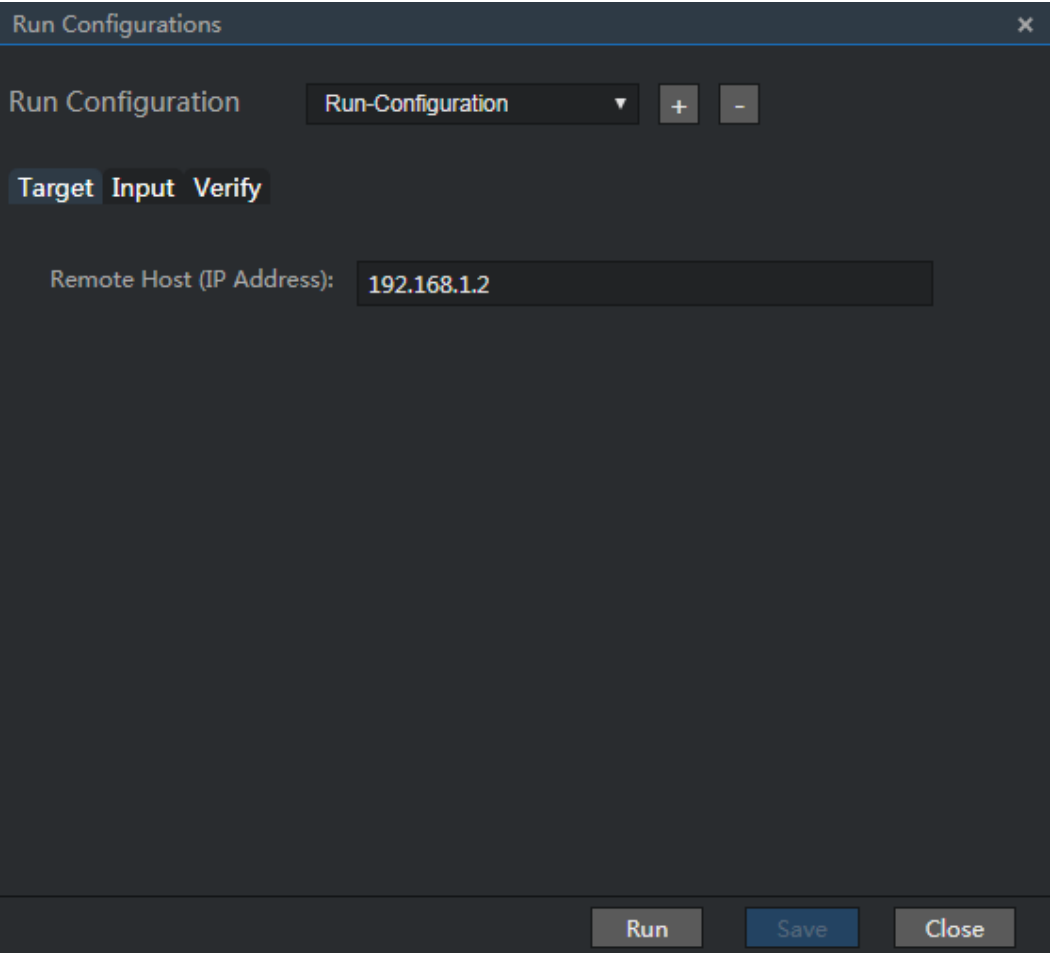
数据文件	说明
reduction_input_2_3_4_sum_axis_1.data	reduction算子的二进制格式输入数据文件。
reduction_input_2_3_4_sum_axis_1.txt	将reduction算子的二进制格式文件以txt文件的方式显示出来，方便用户查看结果。
reduction_output_2_3_4_sum_axis_1.data	reduction算子二进制格式输出数据验证文件，用于验证算子运行后的输出结果是否正确。
reduction_output_2_3_4_sum_axis_1.txt	用户可见的reduction算子的输出数据文件，用处同上。
sign_input_2_4_cce.data	sign算子的二进制格式输入数据文件。 本示例未使用此文件。
sign_input_2_4_cce.txt	将sign算子二进制格式文件以txt文件的方式显示出来，方便用户查看结果。 本示例未使用此文件。
sign_output_2_4_cce.data	sign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是否正确。 本示例未使用此文件。
sign_output_2_4_cce.txt	用户可见sign算子的输出数据文件，用处同上。 本示例未使用此文件。

----结束

## 4.6.2 配置运行参数

**步骤1** 选中TE算子工程，在菜单栏依次选择“Run > Edit Run Configuration...”，弹出运行窗口配置界面。如图4-17所示。

图 4-17 算子运行参数配置界面



**步骤2** 运行参数配置。

参数配置说明如[表4-2](#)所示。

表 4-2 TE 构建配置参数说明

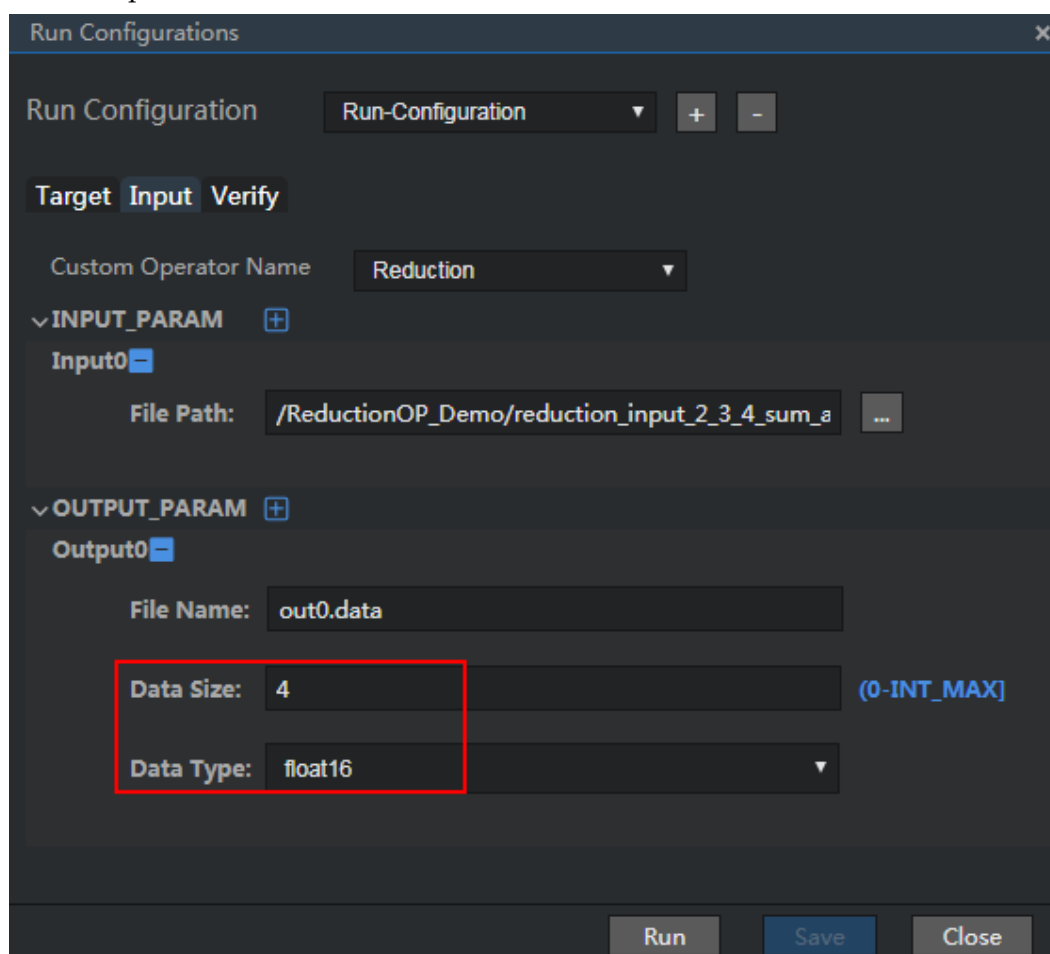
配置参数	参数说明
Target配置	配置运行目标信息。 <ul style="list-style-type: none"><li>● 对于ASIC类型工程，请填写host侧的IP地址。</li><li>● 对于Atlas DK类型工程，请填写开发板的IP地址。</li></ul>

配置参数	参数说明
Input配置	<p>描述算子的输入输出数据信息，配置示例如图4-18所示。</p> <ul style="list-style-type: none"> <li>● “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Reduction。</li> <li>● 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4.6.1 构造输入数据文件构造的二进制输入数据文件（不允许用户手工编辑文件路径）。</li> <li>● 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。 <ul style="list-style-type: none"> <li>- File Name: 输出数据文件，默认为输出到output文件夹下的out0.data文件。</li> <li>- Data Size: 输出数据大小，单位字节。</li> <li>- Data Type: 输出数据类型，选择float16或者float32。</li> </ul> </li> </ul>
Verify配置	<p>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。</p> <p>配置项意义如下，配置示例如图4-19所示。</p> <ul style="list-style-type: none"> <li>● 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: 选择验证数据文件（不允许用户手工编辑文件路径）。</li> <li>● 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）。</li> <li>● 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）。</li> </ul>

Input配置界面示例如图4-18所示。

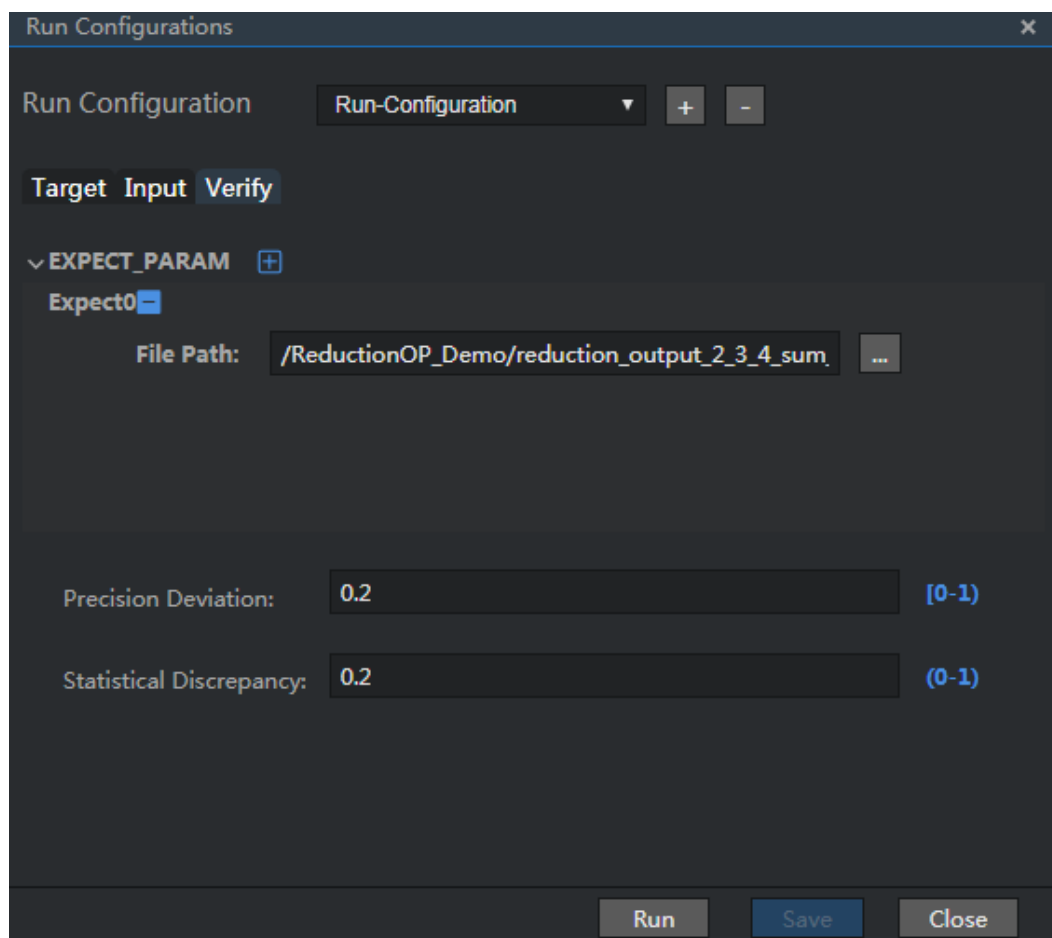
在Input\_PARAM中选择4.6.1 构造输入数据文件中构造的输入数据文件，在OUTPUT\_PARAM中填写输出文件名称，下拉选额OUTPUT\_PARAM的Data Type。

图 4-18 Input 配置界面示例



Verify配置界面示例如[图4-19](#)所示。

图 4-19 Verfiy 配置界面示例



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。

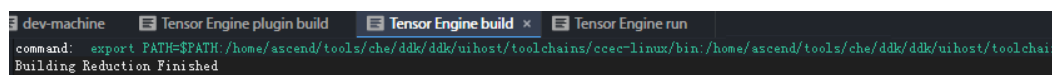
----结束

## 4.6.3 运行单算子

**步骤1** 选中对应的算子工程，在菜单栏选择“Run > Run > Run-Configuration”，运行算子工程。

开始构建、运行后，Mind Studio界面依次出现两个窗口，分别是“Tensor Engine build（编译过程）”和“ProjectName Tensor Engine run（运行过程）”，构建成功如[图4-20](#)所示。

图 4-20 Build 成功



进行文件校验后，运行成功终端输出如[图4-21](#)所示。

图 4-21 运行成功

```

dev-machine  Tensor Engine plugin build  Tensor Engine build  ReductionOP_Demo Tensor Engine run x
command: cd /projects/ReductionOP_Demo && bash .prepare_run.sh && export PATH=$PATH:/home/ascend/tools/lib && run.sh -w workspace
<runtype><-T>(RUN|DEL|COPY|STOP)----: [RUN]
[INFO]: PROGRAM RUN BEGIN.
Wed Feb 27 08:58:24 EST 2019
SUCCESS
/projects/ReductionOP_Demo
[INFO]: startHiAiDaemon done!
[INFO]: target host dir: ~/
[INFO]: APPNAME:main HOSTAPPMAIN:hi_ai_workspace_mind_studio_ReductionOP_Demo_main)
[INFO]: ~/HIAI_PROJECTS/workspace_mind_studio/ReductionOP_Demo is not exist
[INFO]: NOEXIST:IDE-daemon-client "mkdir -p ~/HIAI_PROJECTS/workspace_mind_studio/ReductionOP_Demo/out/"
[INFO]: begin to cp /projects/ReductionOP_Demo/Reduction/out to ~/HIAI_PROJECTS/workspace_mind_studio/ReductionOP_Demo/out/
/projects/ReductionOP_Demo
[INFO]: begin to run project ReductionOP_Demo:main(hi_ai_workspace_mind_studio_ReductionOP_Demo_main) on host...
[INFO]: RUN:ReductionOP_Demo:main(hi_ai_workspace_mind_studio_ReductionOP_Demo_main) run FINISHED.
Output file ./output/out0.data compare result true
    
```

#### 说明

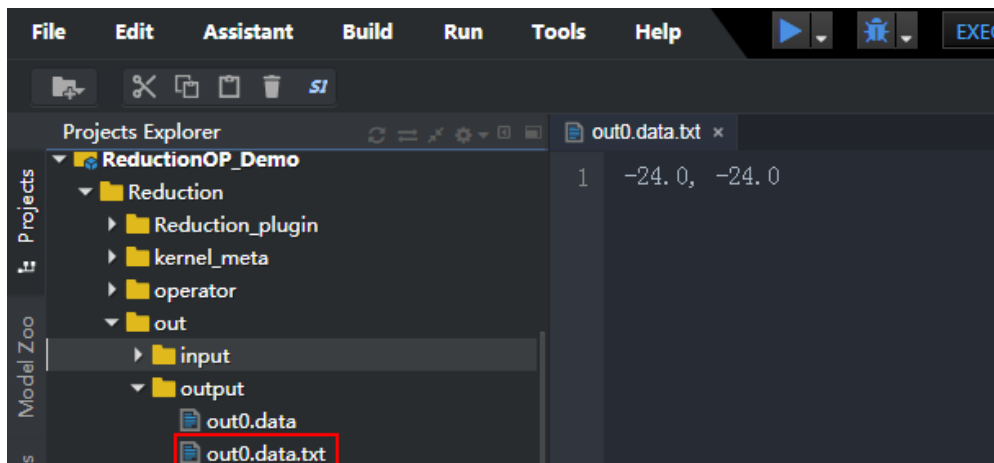
- 若用户想停止工程运行，选中对应的工程后，在菜单栏选择“Run > Stop > Run-Configuration”。
- 在运行时，会自动拷贝工程下配置文件、链接库、二进制程序、输入输出数据等到host侧的 home/HwHiAiUser/HIAI\_PROJECTS 文件夹下。为提高MindStudio 的运行性能，MindStudio 不会自动删除host侧的数据，如用户不需要这些文件，可自行登录host侧，删除/home/HwHiAiUser/HIAI\_PROJECTS下的废弃文件夹。
- 同一Host机器上不支持并发运行单算子。

#### 步骤2 输出验证。

运行成功后，在output文件下会生成输出文件与对比结果文件。

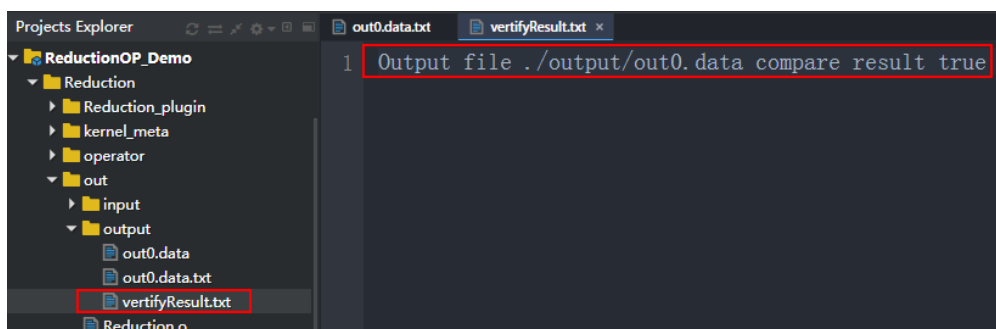
- 输出文件如图4-22所示，在out0.data.txt文件中可以看到结果，可以与4.6.1 构造输入数据文件中构造的输入数据文件reduction\_output\_2\_3\_4\_sum\_axis\_1.txt进行比对，验证实际输出与期望输出数据是否一致。

图 4-22 输出文件



- 对比结果文件如图4-23所示，compare result true表示实际输出与期望输出结果一致。

图 4-23 verifyResult 文件



----结束

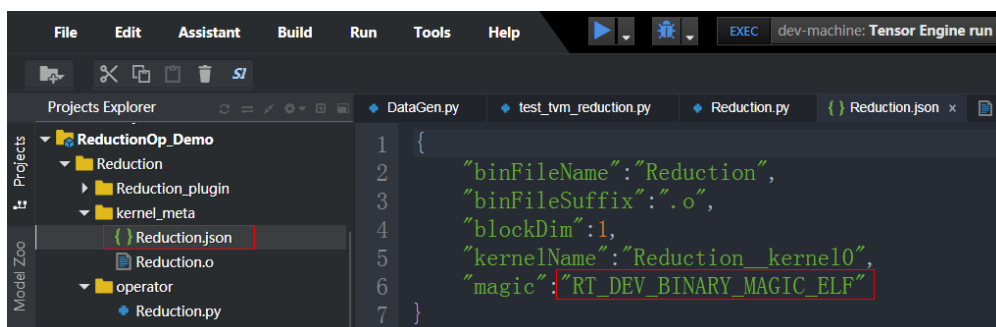
## 相关说明

Tensor Engine算子运行时，系统内部会自动根据算子类型判断算子是运行到AI Core中还是AI CPU中。可以通过json文件查看生成的算子是运行在AI CPU还是运行在AI CORE中。

json文件目录为“ReductionOp\_Demo/Reduction/kernel\_meta/Reduction.json”。

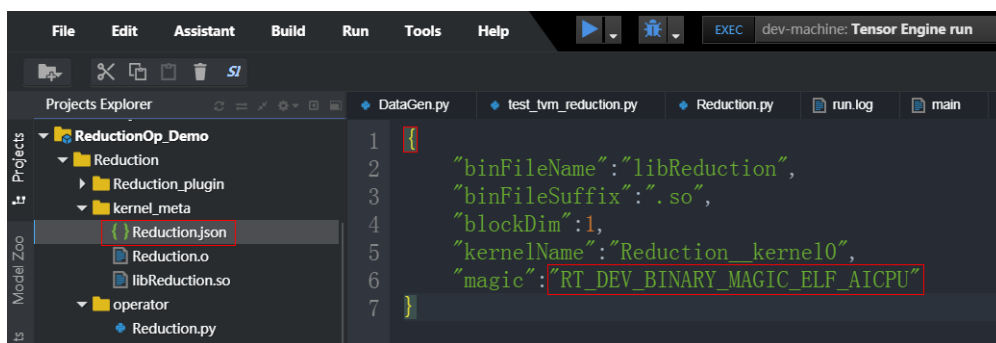
- “RT\_DEV\_BINARY\_MAGIC\_ELF”代表运行目标为AICORE，如图4-24所示。

图 4-24 运行目标为 AI Core 示例图



- “RT\_DEV\_BINARY\_MAGIC\_ELF\_AICPU”代表运行目标为AICPU，如图4-25所示。

图 4-25 运行目标为 AI CPU 示例图



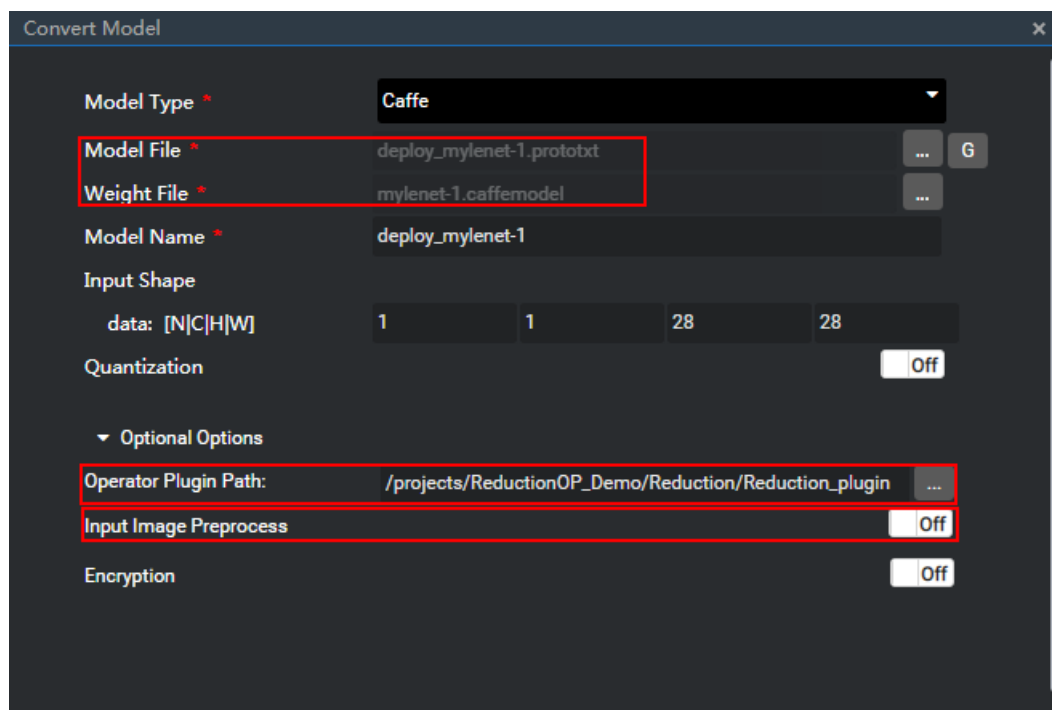
## 4.7 加载插件转换模型

自定义算子和自定义算子插件开发完成后，可以通过加载算子插件的形式对mylenet网络模型进行转化，供后续编排使用。

**步骤1** 选中算子开发工程名，例如`ReductionOP_Demo`，右键选择“Convert Model”。

**步骤2** 在弹出的Convert Model窗口中，进行相应配置，如图4-26所示。

图 4-26 模型转换配置







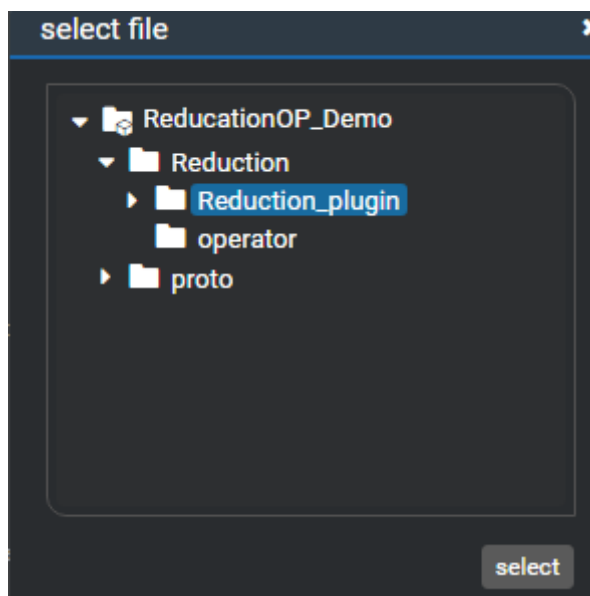
- Model Type选择Caffe。
- Model File单击后面的，选择需要转化的模型对应的portotxt文件，并上传。
- Weight File单击后面的，选择需要转化的模型对应的caffemodel文件，并上传。
- 单击Optional Options前面的，单击Tensor Engine Plugin Path后面的，在弹出页面中选择4.5 编译TE算子工程中编译生成的插件so所在的目录，如图4-27所示。



图 4-27 选择插件

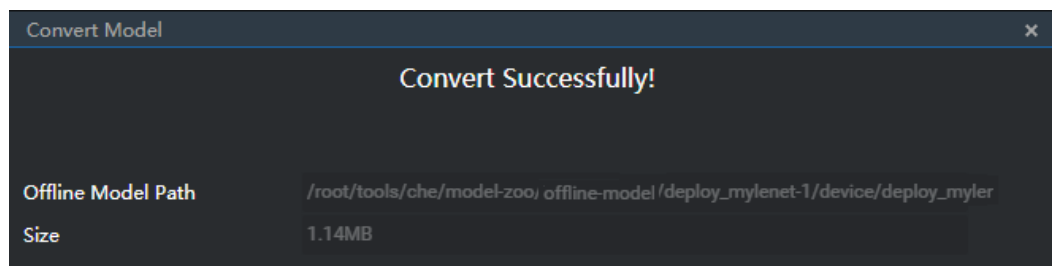


- 由于mylenet模型不加入预处理节点，故此模型转换程中需要将“Input Image Preprocess”按钮设为“off”。

**步骤3** 所有配置完成后，单击“ok”，开始进行模型转换。

模型转换成功后，会弹出如图3 模型转换成功所示提示信息。

图 4-28 模型转换成功

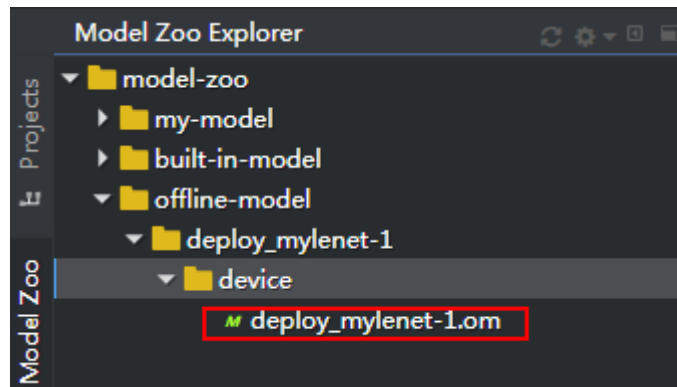


#### 说明

如果转换失败，会进行提示，并且会在工程目录生成日志文件“convertModel.log”，您可以在此文件中查看失败原因。

转化成功后，可以在“Model Zoo Explorer”窗口中查看生成的模型文件，如图4-29所示。

图 4-29 生成后模型



----结束

# 5 自定义算子整网络执行

---

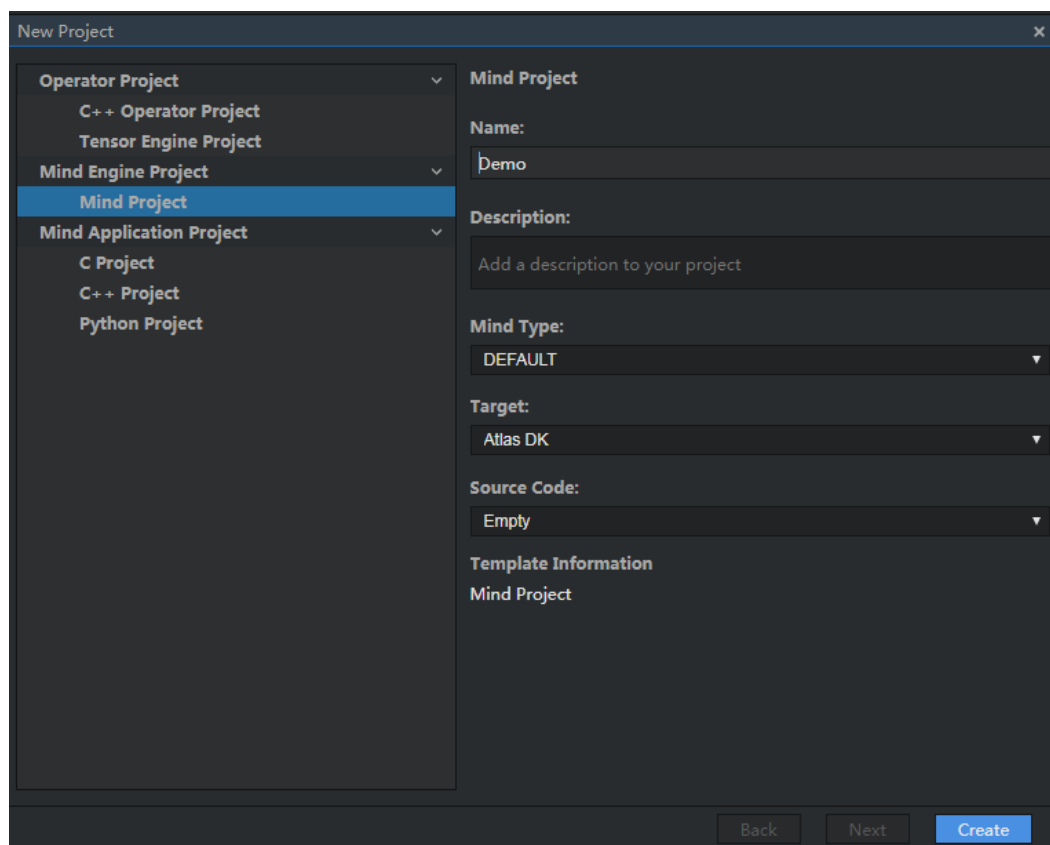
本章节通过使用[4.7 加载插件转换模型](#)生成的mylenet网络模型，通过Engine编排的方式，实现对每一张输入图片数字识别概率的累加验证功能。

## 5.1 创建 Mind 工程

如果使用模型转化失败进入算子开发工程的方式进行算子开发，即[4.2.2 模型转化失败进入算子开发工程](#)章节已经创建了Mind工程，则本章节跳过。

- 步骤1** 依次选择“File > New > New Project > ...”，弹出“New Project”窗口，创建一个新的工程。
- 步骤2** 在“New Project”窗口中选中“Mind Engine Project > Mind Project”工程，显示窗口配置界面，如[图5-1](#)所示。

图 5-1 创建 Mind Engine 工程



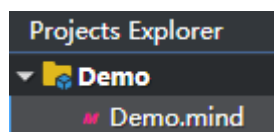
参考表5-1进行工程配置。

表 5-1 参数说明

参数	参数说明
Name	工程名称，自行配置。 名字需要为字符串，不包含空格，空格会自动填充为“-”。
Mind Type	选择“DEFAULT”，使用Mind Studio编排功能。
Target	<ul style="list-style-type: none"><li>“ASIC”表示运行目标是EVB单板或者PCIe单板。</li><li>“Atlas DK”表示运行目标是开发板。</li></ul>
Source Code From	选择“Empty”：自行开发工程，非外部导入。

**步骤3** 单击“Create”新建一个Mind工程，会自动生成跟工程名相同的mind文件，如图5-2所示，该文件不可复制、删除、重命名。

图 5-2 创建工程示例



----结束

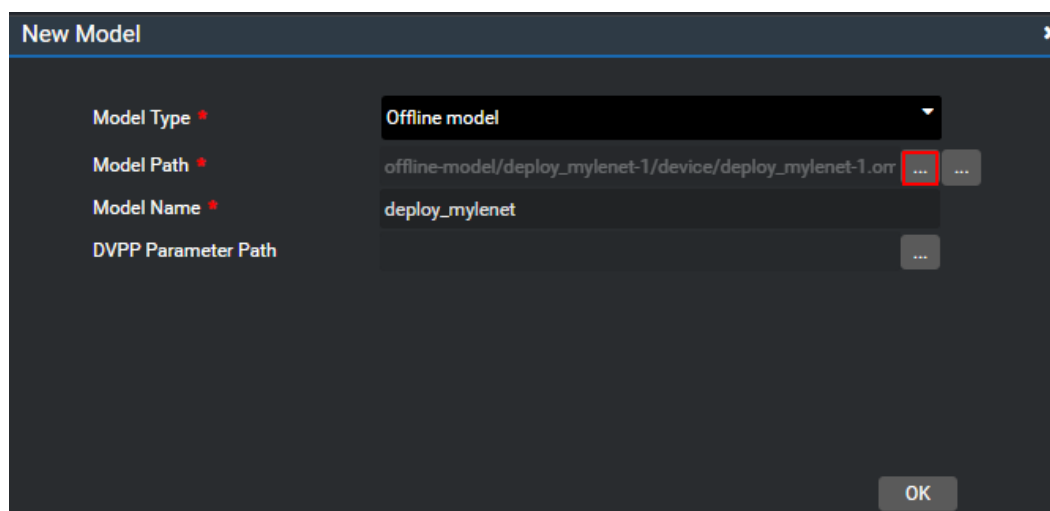
## 5.2 导入网络模型

**步骤1** 在“Projects Explorer”中双击*Demo.mind*文件。

**步骤2** 在“tool”栏中选择“Model > My Models”后的“+”，弹出“New Model”窗口。

**步骤3** 在“Model Type”中选择“offlineModel”，单击“Model Path”右侧左边的“...”（From Web Server），选择[4.7 加载插件转换模型](#)中的模型文件，将加载算子插件后转化的模型导入到Mind工程中，如[图5-3](#)所示。

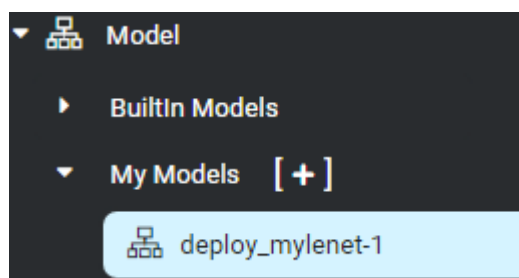
图 5-3 导入转化好的模型



导入成功后，会在“My Models”中看到新加的模型文件，可以将其用于流程编排拖拽。

如[图5-4](#)所示。

图 5-4 导入自定义模型



----结束

## 5.3 编排流程

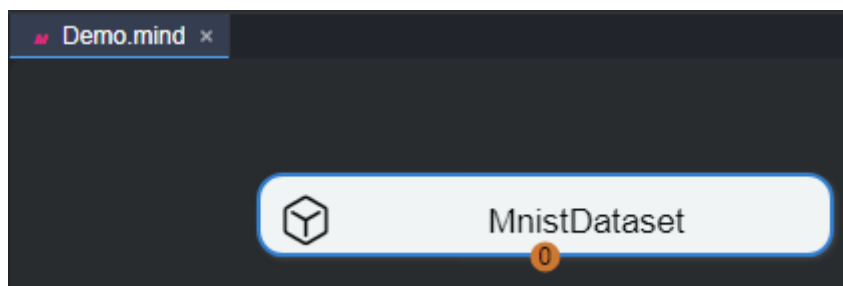
**步骤1** 双击5.1 创建Mind工程中创建的*Demo.mind*文件，打开Engine流程编排窗口。

**步骤2** 创建Datasets节点。

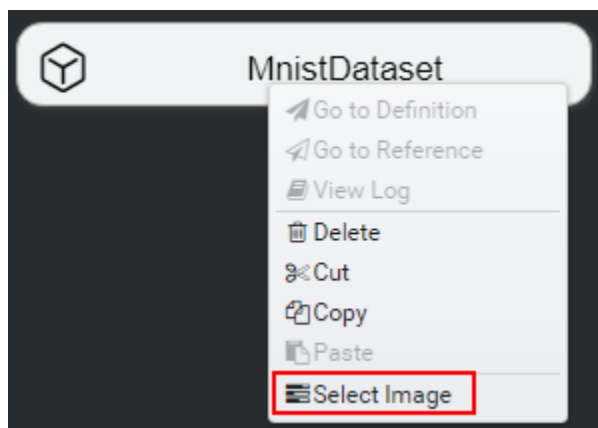
本示例基于工具内置的MnistDataset数据集做修改，操作步骤如下。

1. 将右侧内置数据集“Datasets > BuiltIn Datasets > MnistDataset”拖拽至网络编排界面，如图5-5所示。

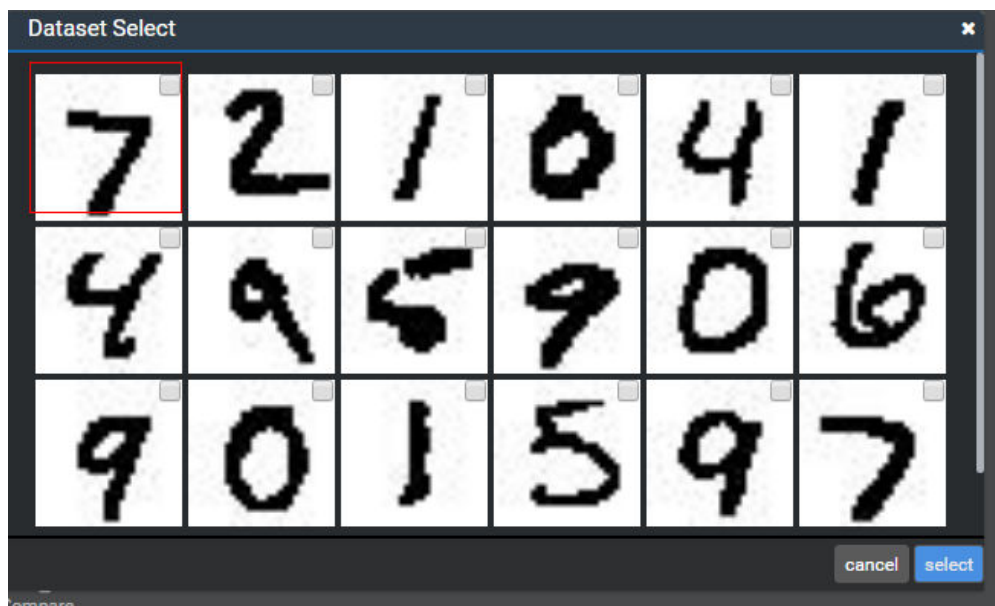
图 5-5 拖拽内置数据集



右键选择“Select Image”。



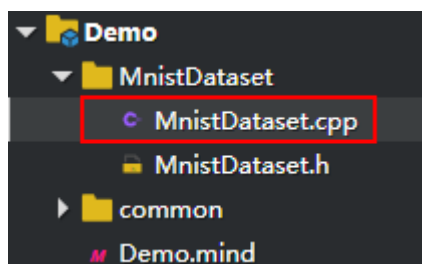
勾选其中一张图片后（此处选择第一张图片），单击“select”。



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. 本示例使用内置的数据集，在网络中处理一张灰度图，需要修改部分MnistDataset源码，双击打开MnistDataset.cpp源文件，如图5-6所示。

图 5-6 MnistDataset.cpp 源文件



3. 如果工程target类型是Atlas DK，修改如图5-7所示。

图 5-7 修改源码

```
HIAI_StatusT MnistDataset::RunOnSameSide(std::ifstream* imageStream){
    int totalCount = GetTotalCount();
    int batchNum = totalCount % data_config->batchSize == 0 ? totalCount / data_config->batchSize :
    int i = 0;
    int size = width * height * 4;
    int frameId = 0;
    HIAI_ENGINE_LOG(HIAI_IDE_INFO, "[MnistDataset] run on %s, run %d images, batch size %d",
    for(int batchId = 0; batchId < batchNum && i < totalCount; batchId++){
        //convert batch image infos to BatchImageParaWithScaleT
        std::shared_ptr<BatchImageParaWithScaleT> imageInfoBatch = std::make_shared<BatchImageParaWithScaleT>();
        for(int j = 0; j < data_config->batchSize && i < totalCount; j++){
            uint8_t * imageBufferPtr = NULL;
            try{
                imageBufferPtr = new uint8_t[size];
            }catch (const std::bad_alloc& e) {
                HIAI_ENGINE_LOG(HIAI_IDE_ERROR, "[MnistDataset] alloc data error!");
                return HIAI_ERROR;
            }
        }
    }
}
```

如果工程target类型是ASIC，修改如图5-8所示。

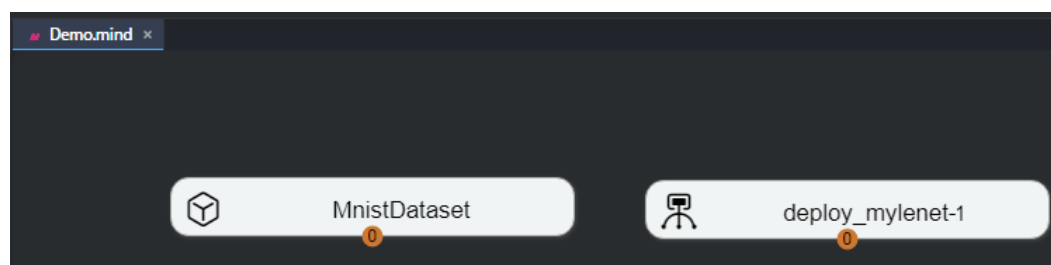
图 5-8 修改源码--ASIC

```
HIAI_StatusT MnistDataset::RunOnDifferentSide(std::ifstream* imageStream) {
    int totalCount = GetTotalCount();
    int batchNum = totalCount % data_config->batchSize == 0 ? totalCount / data_config->batchSize :
    int i = 0;
    int size = width * height * 4;
    int frameId = 0;
    HIAI_ENGINE_LOG(HIAI_IDE_INFO, "[MnistDataset] run on %s, run %d images, batch size is %d",
    for(int batchId = 0; batchId < batchNum && i < totalCount; batchId++){
        //convert batch image infos to std::vector<std::shared_ptr<EvbImageInfo>>
        std::vector<std::shared_ptr<EvbImageInfo>> evbImageInfoBatch;
        for(int j = 0 ; j < data_config->batchSize && i < totalCount; j++){
            uint8_t * imageBufferPtr = NULL;
            HIAI_StatusT get_ret = hiai::HIAIMemory::HIAI_DMalloc(size, (void*)&imageBufferPtr);
            if(HIAI_OK != get_ret || NULL == imageBufferPtr){
```

**步骤3** 创建Model节点。

将[5.2 导入网络模型](#)中添加的离线网络模型拖动到网络编排界面，如[图5-9](#)所示。

图 5-9 将自定义模型拖动到编排界面



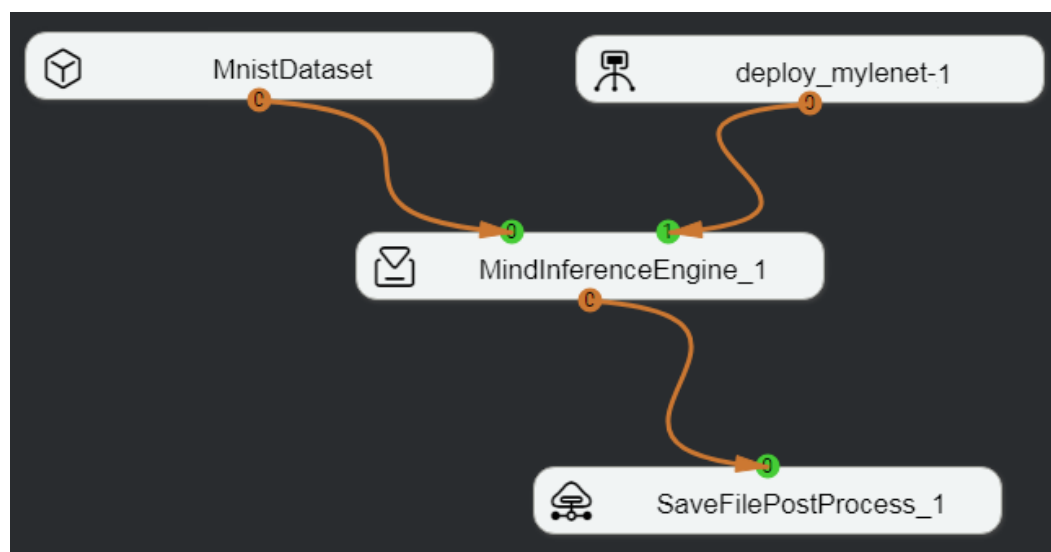
**步骤4** 创建Engine节点。将“Deep Learning Execute Engine > MindInferenceEngine”节点拖动到网络编排界面。

**步骤5** 将“PostProcess > SaveFilePostProcess”后处理节点拖动到画布。

**步骤6** 连接节点，完成网络流程编排。

节点之间通过连线连接，如[图5-10](#)所示。

图 5-10 连接节点





**步骤7** 单击画布下侧的“Save”，保存编排的流程。

----结束

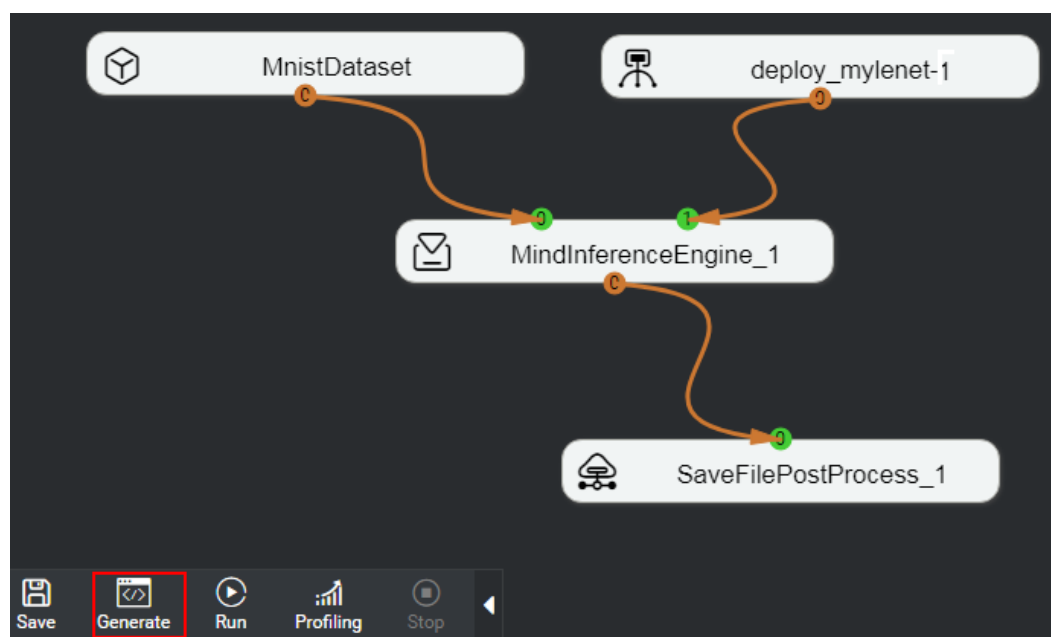
## 5.4 编译运行

Mind Engine Project支持一键式编译运行，根据工程的配置在运行时会执行不同的操作。

### 编译

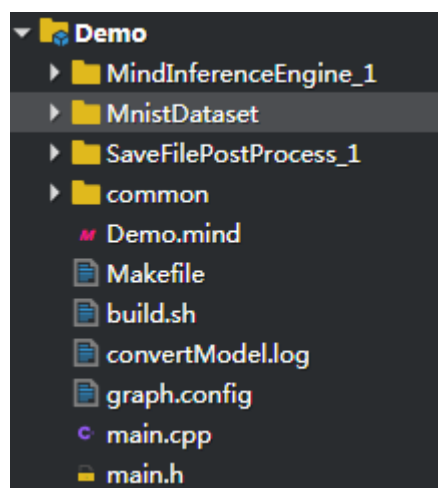
在完成网络结构的编辑之后，单击左下方的“Generate”来生成对应的源码与执行脚本，如[图5-11](#)所示。

图 5-11 工程编译



工程编译后，会自动生成对应的源码与可执行文件，如[图5-12](#)所示。

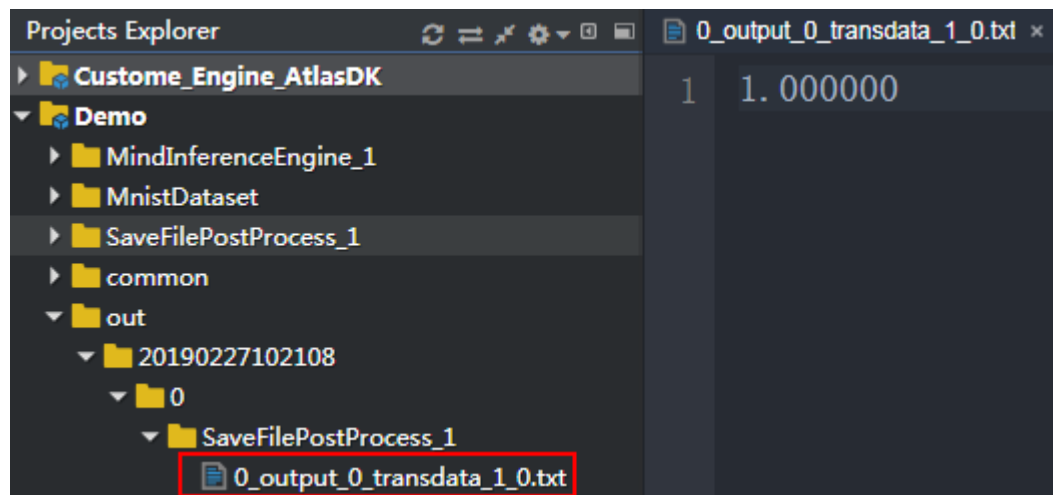
图 5-12 编译后生成文件



## 运行

单击“Run”，即可启动MindEngine编排流程，此时可以在“dev-machine”窗口中查看提示run FINISHED的运行输出。

在工程文件的out目录下查看最终结果，结果为1，其中20190227102108为以执行时间戳命名的目录。



### 说明

运行时，会自动拷贝工程下的模型文件、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Mind Studio的运行性能，Mind Studio不会自动删除Host侧的数据，如果用户不需要这些文件，可自行登录到Host侧，删除/home/HwHiAiUser/HIAI\_DATANDMODELSET和/home/HwHiAiUser/HIAI\_PROJECTS下的废弃文件夹。

# 6 算子开发参考

本章节对reduction算子及sign算子的定义及代码实现进行详细的讲解。

了解TE定义算子的更多详细信息请参见《Ascend 310 TensorEngine API》。

## 6.1 reduction 算子介绍

### 6.1.1 reduction 算子定义

这里实现的reduction算子是caffe中的redcution算子，对指定轴及其之后的轴做reduce操作，这个算子包含四种类型。

表 6-1 reduction 算子类型

算子类型	说明
SUM	对被reduce的所有轴求和。
ASUM	对被reduce的所有轴求绝对值后求和。
SUMSQ	对被reduce的所有轴求平方后再求和。
MEAN	对被reduce的所有轴求均值。

reduction算子需要指定一个轴，会对此轴及其之后的轴进行reduce操作。比如，输入的tensor的形状为[5,6,7,8]。

- 如果指定的轴是3，则输出tensor的形状为[5,6,7]。
- 如果指定的轴是2，则输出tensor的形状为[5,6]。
- 如果指定的轴是1，则输出tensor的形状为[5]。
- 如果指定的轴是0，则输出tensor的形状为[1]。

同时，此算子还支持对输出结果进行缩放，可以指定一个标量，对结果缩放。

reduction算子的输入和输出分别为。

- 输入

表 6-2 reduction 算子输入

输入参数	说明
x	一个Tensor，维度为N。
dtype	输入的数据类型，支持如下几种常见类型：float16，float32。
axis	指定reduce的轴，取值范围为：[-N, N-1]。
op	reduce操作的类型，支持：SUM,ASUM,SUMSQ,MEAN。
coeff	标量，对结果进行缩放。如果值为1，则不进行缩放。

- 输出

y: 与输入x数据类型相同的Tensor，其维度由输入tensor维度和指定轴决定。

reduction算子的实现和sign算子除了计算过程不同外，其他流程基本一致，所以下面只讲述如何描述计算过程。

## 6.1.2 reduction 算子代码解析

reduction算子实现的完整代码请参见[4 自定义算子开发实例](#)。

### 6.1.2.1 导入 tvn 模块

使用TE定义算子时，需要导入如下模块。

```
import te.lang.cce
from te import tvn
import topi
```

在创建好的python文件中写入上述代码，导入这两个模块。

### 6.1.2.2 声明输入

预处理

reduction算子是对“指定轴及其之后的轴”做reduce操作，所以我们可以把“指定轴及其之后的所有轴”看作是一根轴。这样，可以把输入Tensor做个预处理，把指定轴后的所有轴看作是一根轴，后续做reduce操作时，只需在最后一根轴上做reduce即可。

预处理代码如下。

```
if axis < 0:
    axis = len(shape) + axis
shape = list(shape)
shapel = shape[:axis] + [reduce(lambda x, y: x * y, shape[axis:])] # 把指定轴及其之后的轴合并为一根轴
```

定义一个输入的placeholder。

```
data_input = tvn.placeholder(shapel, name="data_input", dtype=inp_dtype)
```

在TE中，使用placeholder接口定义输入Tensor。placeholder就是一个占位符，它返回的是一个Tensor对象，表示一组输入数据，知道它的大小和数据类型。数据的具体内容在运行时才能知道。

```
shape = (1024, 1024)
dtype = "float16"
data = tvm.placeholder(shape, name="data", dtype=dtype)
```

其中参数解释如表6-3所示。

表 6-3 参数解释

参数	说明
shape	Tensor的尺寸，比如（10）或者（1024， 1024）或者（2， 3， 4）等。
name	定义输入的名字。
dtype	定义数据类型，比如“float16”、“float32”。

Tensor是TE中一个重要数据结构。它表示一个Tensor对象。

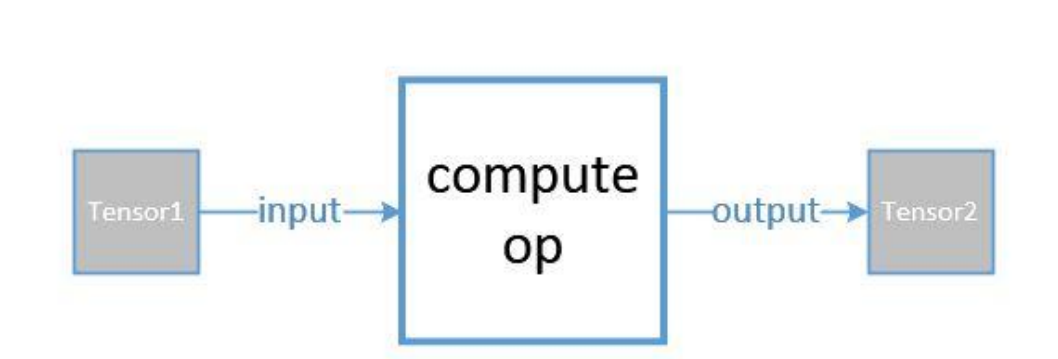
表 6-4 Tensor 对象参数解释

参数	说明
shape（尺寸）	比如（10）或者（1024， 1024）或者（2， 3， 4）等。
数据类型	“float16”、“float32”。
一个op	op描述了计算信息，可以是一个输入（placeholder）或者一个计算（compute）。

比如，用TE描述A+B=C，那么A、B、C都是一个Tensor，它们的shape和dtype是一样的。A和B是输入，它们的op就是一个placeholder，表示一个占位符。C是经过A和B相加得到的，C的op是一个compute，表示一个计算。一般，每个compute都是一个比较小的操作，比如一个Tensor加上一个常量、两个Tensor相乘等。

在TE中，一个计算过程，可以通过一组Tensor表示出来。

图 6-1 计算过程





说明

input也可以为两个或者多个。

### 6.1.2.3 计算过程

对不同操作做不同处理。

对ASUM操作，需要先对输入的数据做element-wise的绝对值操作。

```
if op == "ASUM":
    data_tmp_input = te.lang.cce.vabs(data_input)
```

如果是SUMSQ操作，则需要平方计算，通过把两个tensor相乘得到。

```
if op == "SUMSQ":
    data_tmp_input = te.lang.cce.vmul(data_input, data_input)
```

然后对缩放参数进行处理，就是把输入tensor乘上一个标量。

```
cof = coeff
tmp = te.lang.cce.vmul(data_input, cof)
```

然后做最后一个轴的求和操作，sum接口可以做此操作。

```
res_tmp = te.lang.cce.sum(tmp, axis = axis)
```

如果是MEAN操作，则需要做一次求均值操作。

```
if op == "MEAN":
    size = shape1[-1] # shape1[-1]是预处理后的最后一根轴，也就是被reduce掉的数的个数
    size_rec1 = 1.0 / size
    res_tmp = te.lang.cce.vmul(res_tmp, size_rec1)
```



说明

具体每个接口的详细说明，请参见《Ascend 310 TensorEngine API》。

### 6.1.2.4 算子编译

经过上面的处理，已经描述了算子的计算过程，然后把它交给TE，使用TE的auto\_schedule接口生成schedule对象。schedule可以理解为：描述的计算过程如何在硬件上高效执行。就是把相关的计算和硬件设备上的相关指令对应起来。

schedule对象中包含一个“中间表示”（IR），它用一种类似伪代码来描述计算过程，可以通过相关参数把它打印出来进行查看。关于schedule的详细描述，可参见[TVM官方文档](#)。

TE会根据schedule去构建（build）这个算子，最终生成可在硬件上执行的二进制文件。

代码如下。

```
# 定义schedule
with tvm.target.cce():
    sch = topi.generic.auto_schedule(res)

# 定义build参数
config = {"print_ir": True,
          "need_build": True,
          "name": "cce_reduction",
          "tensor_list": [data_input, res]}

# build算子,生成目标文件
te.lang.cce.cce_build_code(sch, config)
```



- with `tvm.target.cce()`表示指定硬件目标为CCE平台。
- `config`是一个map，配置build的相关参数：
  - `print_ir`: 是否打印IR，默认是True
  - `need_build`: 是否进行build，默认是True
  - `name`: 算子的名字，默认是`cce_op`，后面在调用算子时会使用这个名字。
  - `tensor_list`: 算子的输入和输出tensor列表，输入是placeholder接口返回的tensor对象，输出是经过计算后的tensor对象。

## 6.1.3 算子插件注册代码解析

### 算子参数解析

通过`IOParser::ParseParams`接口实现。

模型文件以及权值文件的读取与解析由离线模型转换工具OMG实现，插件仅解析自定义算子相关参数。

1. 校验算子的合法性，算子输入不能为空。

```
const caffe::LayerParameter* layer = dynamic_cast<const caffe::LayerParameter*>(op_src);  
// #### 输入算子参数合法性校验  
if (nullptr == layer)  
{  
    printf("Dynamic cast op_src to LayerParameter failed\n");  
    return FAILED;  
}
```

2. 解析算子的`operation`，`axis`，`coeff`参数，并保存。

```
std::map<::caffe::ReductionParameter_ReductionOp, std::string> operation_map = {  
    { caffe::ReductionParameter_ReductionOp_SUM, "SUM" },  
    { caffe::ReductionParameter_ReductionOp_ASUM, "ASUM" },  
    { caffe::ReductionParameter_ReductionOp_SUMSQ, "SUMSQ" },  
    { caffe::ReductionParameter_ReductionOp_MEAN, "MEAN" },  
};  
// get op params  
const ::caffe::ReductionParameter& param = layer->reduction_param();  
// set op params to op_dest  
AddOpAttr("operation", operation_map[param.operation()], op_dest);  
AddOpAttr("axis", (int32_t)param.axis(), op_dest);  
AddOpAttr("coeff", param.coeff(), op_dest);
```

### 输出 tensor 描述

通过`IOPBuilder::GetOutputDesc`接口实现。

通过输入tensor描述，推断输出tensor的描述。

由于输入、输出描述一样(shape, data type, data format)，可以直接将输入Tensor复制到输出，修改后的代码为。

```
virtual Status GetOutputDesc(vector<TensorDescriptor>& v_output_desc) override  
{  
    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
```

```
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;
}
v_output_desc[0].set_dim(axis, 1);
return SUCCESS;
}
```

## 构建算子二进制文件

通过IOpBuilder::BuildTvmBinFile接口实现。

该接口通过C++调用python代码（Reduction.py中代码），生成算子文件，然后由framework将此算子文件编译到模型中，修改后的代码为。

```
virtual Status BuildTvmBinFile(TVMBinInfo& tvm_bin_info) override
{
    // 1. call tvm python api to build bin files
    // 2. set path to TVMBinInfo tvm_bin_info
    PyObject *pModule = NULL;
    PyObject *pFunc = NULL;
    PyObject *pArg = NULL;
    PyObject *result = NULL;

    // ### python初始化
    Py_Initialize();

    // ### 导入topi.cce模块
    PyRun_SimpleString("import sys");
    // ### 导入当前自定义算子目录
    PyRun_SimpleString("sys.path.append('/projects/ReductionOP_Demo/Reduction/operator')");
    // ### 导入当前自定义算子python文件
    pModule = PyImport_ImportModule("Reduction");
    if (!pModule)
    {
        PyErr_Print();
        printf("import topi.cce failed!\n");
        return FAILED;
    }

    std::string operation;
    int32_t axis = -1;
    float coeff = 1;
    // ### 解析operation参数
    if (!GetOpAttr("operation", &operation, op_def_))
    {
        // ### 增加异常处理、可维护信息
        printf("GetOpAttr operation failed!\n");
    }

    // ### 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
    if (axis < 0) axis -= 2;

    // ### 解析coeff参数
    if (!GetOpAttr("coeff", &coeff, op_def_))
    {
        printf("GetOpAttr coeff failed!\n");
    }

    // ### 解析输入tensor描述
```



```
TensorDescriptor input_desc = op_def->input_desc(0);
// ### 解析输入shape值，校验其大小是否为4
if (input_desc.dim_size() != 4)
{
    printf("The shape size is %d, which is not 4!", input_desc.dim_size());
    return FAILED;
}
// 调用算子Python接口生成算子，其中"Reduction"为算子接口名。
pFunc = PyObject_GetAttrString(pModule, "Reduction");
std::string kernel_name = "reduction_float16_" +
    std::to_string(input_desc.dim(0)) + "_" +
    std::to_string(input_desc.dim(1)) + "_" +
    std::to_string(input_desc.dim(2)) + "_" +
    std::to_string(input_desc.dim(3));
pArg = Py_BuildValue(
    "(i,i,i,i), s, i, s, f, s",
    input_desc.dim(0), input_desc.dim(1), input_desc.dim(2), input_desc.dim(3), "float16",
    axis, operation.c_str(), coeff, kernel_name.c_str());
PyEval_CallObject(pFunc, pArg);
// ### 结束python调用
Py_Finalize();

// 设置算子文件路径及kernel name
tvm_bin_info.bin_file_path = "./kernel_meta/" + kernel_name + ".o";
tvm_bin_info.json_file_path = "./kernel_meta/" + kernel_name + ".json";
return SUCCESS;
}
```

## 算子注册

需要调用OMG接口注册自定义算子的信息，这样OMG在转换过程中才能识别自定义算子的类型。

注册实例如下。

```
// Register the op plugin
DOMI_REGISTER_OP("Reduction") // op name
    .FrameworkType(CAFFE) // enum type, CAFFE, TENSORFLOW
    .OriginOpType("Reduction") // name of the same op in CAFFE
    .ParserCreatorFn(PARSER_FN(ReductionOpParser)) // class name of op parser
    .BuilderCreatorFn(BUILDER_FN(ReductionOpBuilder)) // class name of op builder
    .ImpleType(ImpleType::TVM); // op type, TVM
}
```

## 6.2 sign 算子介绍

### 6.2.1 sign 算子定义

sign算子是一个**element-wise**算子，返回输入数据中每个元素的符号。对于输入数据中每个元素，其计算公式为。

$$y = \begin{cases} 1, & \text{if } x > 0 \\ 0, & \text{if } x = 0 \\ -1, & \text{if } x < 0 \end{cases}$$

sign算子的输入和输出分别为。

- 输入

- **x**: 一个**Tensor**。
  - **dtype**: 输入的数据类型，支持如下几种常见类型：float16、float32。
- 输出
- y**: 与输入x形状、数据类型相同的Tensor。

## 6.2.2 算子实现代码解析

下面给出sign算子的代码解析，完整代码如下。

```
import te.lang.cce
from te import tvn
import topi

def sign_cce(data_shape, dtype, kernel_name = "cce_sign", need_build = False, need_print = False):
    # 定义常量
    fp16_max = tvn.const(32768, dtype="float16")
    fp16_min = tvn.const(2 ** (-15), dtype = "float16")

    # 定义输入Tensor
    data = tvn.placeholder(data_shape, name="data", dtype=dtype)

    # 描述计算过程
    new_data = te.lang.cce.vmul(data, fp16_max)
    tmp2 = te.lang.cce.vabs(new_data)
    anuminate = te.lang.cce.vadds(tmp2, fp16_min)
    fp16_res = te.lang.cce.vmul(new_data, te.lang.cce.vrec(anuminate))
    res = te.lang.cce.round(fp16_res)

    # 编译算子
    # 定义schedule
    with tvn.target.cce():
        sch = topi.generic.auto_schedule(res)

    # 定义build参数
    config = {"print_ir" : need_print,
              "need_build" : need_build,
              "name" : kernel_name,
              "tensor_list": [data, res]
            }

    # build算子,生成目标文件
    te.lang.cce.cce_build_code(sch, config)

sign_cce((2, 4), "float16", need_build = True)
```

### 6.2.2.1 导入模块

使用TE定义算子时，需要导入如下模块。

```
import te.lang.cce
from te import tvn
import topi
```

在创建好的python文件中写入上述代码，导入这个模块。

### 6.2.2.2 声明输入

在TE中，使用placeholder接口定义输入Tensor。placeholder就是一个占位符，它返回的是一个Tensor对象，表示一组输入数据，知道它的大小和数据类型。数据的具体内容在运行时才能知道。

```
shape = (1024, 1024)
dtype = "float16"
data = tvn.placeholder(shape, name="data", dtype=dtype)
```

其中参数解释如表6-5所示。

表 6-5 placeholder 参数说明

参数	说明
shape	Tensor的尺寸，比如（10）或者（1024，1024）或者（2，3，4）等。
name	定义输入的名字。
dtype	定义数据类型，比如“float16”、“float32”。

Tensor是TE中一个重要数据结构。它表示一个Tensor对象。

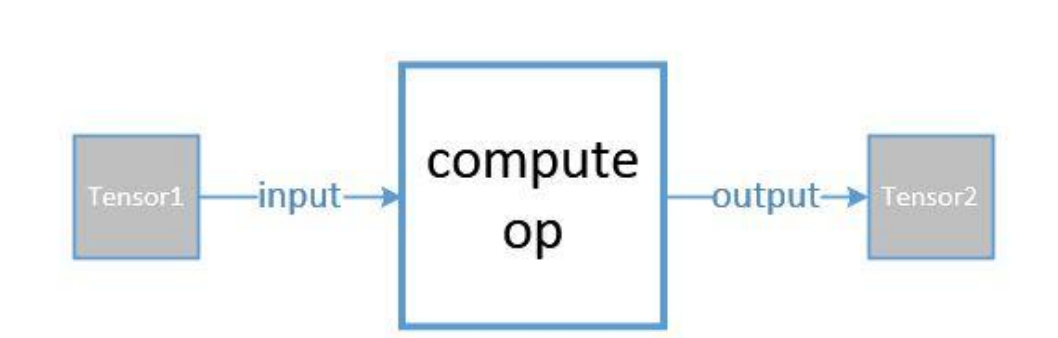
表 6-6 Tensor 对象说明

参数	说明
shape（尺寸）	比如（10）或者（1024，1024）或者（2，3，4）等。
数据类型	“float16”、“float32”。
一个op	op描述了计算信息，可以是一个输入（placeholder）或者一个计算（compute）。

比如，用TE描述A+B=C，那么A、B、C都是一个Tensor，它们的shape和dtype是一样的。A和B是输入，它们的op就是一个placeholder，表示一个占位符。C是经过A和B相加得到的，C的op是一个compute，表示一个计算。一般，每个compute都是一个比较小的操作，比如一个Tensor加上一个常量、两个Tensor相乘等。

在TE中，一个计算过程，可以通过一组Tensor表示出来。

图 6-2 计算过程



说明

input也可以为两个或者多个。

### 6.2.2.3 计算过程

一般求正负号使用的公式是 $x/|x|$ ，这个公式简单，但是无法支持 $x=0$ 的情况，所以我们用下面的公式进行计算。

$$y = \text{round}\left(\frac{x * 32768}{2^{-15} + |x * 32768|}\right)$$

其中round是四舍六入，遇0.5取最近偶数。

计算过程中有两个常量32768和 $2^{-15}$ ，先把他们定义出来。

# 定义常量

```
fp16_max=tvm.const(32768, dtype= "float16" )
```

```
fp16_min=tvm.const(2**(-15), dtype= "float16" )
```

 说明

tvm.const()返回的是一个常量对象。

根据上面的计算公式，可以把计算过程分为如下步骤。

**步骤1** 计算 $a = x * 32768$

**步骤2** 计算 $b = |a|$

**步骤3** 计算 $c = b + 2^{-15}$

**步骤4** 计算 $d = a / c$

**步骤5** 最后计算 $y = \text{round}(d)$

----结束

然后用TE API把这个计算过程描述出来。

**步骤1**  $a = x * 32768$

```
new_data = te.lang.cce.vmul(data, fp16_max)
```

**步骤2**  $b = |a|$

```
tmp2 = te.lang.cce.vabs(new_data)
```

**步骤3**  $c = b + 2^{-15}$

```
anuminate = te.lang.cce.vadds(tmp2, fp16_min)
```

**步骤4**  $d = a * (1/c)$

```
fp16_res = te.lang.cce.vmul(new_data, te.lang.cce.vrec(anuminate))
```

**步骤5**  $y = \text{round}(d)$

```
res = te.lang.cce.round(fp16_res)
```

----结束

代码中的每一行，得到的结果都是一个Tensor对象，它可以作为下一个计算的输入，计算得到另一个Tensor。

具体每个接口的详细说明，请参见《Ascend 310 TensorEngine API》。



说明

这里只是定义输入经过什么样的计算可以得到输出，并没有真正的计算发生。

### 6.2.2.4 算子编译

经过上面的处理，已经描述了算子的计算过程，然后把它交给TE，使用TE的 `auto_schedule` 接口生成 `schedule` 对象。`schedule` 可以理解为：描述的计算过程如何在硬件上高效执行。就是把相关的计算和硬件设备上的相关指令对应起来。

`schedule` 对象中包含一个“中间表示”（IR），它用一种类似伪代码来描述计算过程，可以通过相关参数把它打印出来进行查看。关于 `schedule` 的详细描述，可参见 [TVM 官方文档](#)。

TE 会根据 `schedule` 去构建（build）这个算子，最终生成可在硬件上执行的二进制文件。

代码如下。

```
# 定义schedule
with tvn.target.cce():
    sch = topl.generic.auto_schedule(res)

# 定义build参数
config = {"print_ir" : True,
         "need_build" : True,
         "name" : "cce_sign",
         "tensor_list": [data, res]}

# build算子,生成目标文件
te.lang.cce.cce_build_code(sch, config)
```



说明

- `with tvn.target.cce()` 表示指定硬件目标为 CCE 平台。
- `config` 是一个 map，配置 build 的相关参数。
  - `print_ir`: 是否打印 IR，默认是 True
  - `need_build`: 是否进行 build，默认是 True
  - `name`: 算子的名字，默认是 `cce_op`，后面在调用算子时会使用这个名字。
  - `tensor_list`: 算子的输入和输出 tensor 列表，输入是 `placeholder` 接口返回的 tensor 对象，输出是经过计算后的 tensor 对象。

# 7 算子插件开发函数参考

## 7.1 相关函数列表

本章介绍算子插件开发过程中涉及的函数，涉及到的宏定义及数据类型的介绍可以参考《Ascend 310 Framework API参考》。

### 7.1.1 非 IR 方式

#### 7.1.1.1 ParseParams 函数

##### 函数功能

用户需要自定义继承CaffeTVMOpParser的类，并重写函数ParseParams，实现将caffe模型prototxt中的算子参数映射到离线模型。

##### 函数原型

```
Status ParseParams(const Message* op_src, OpDef* op_dest);
```

##### 参数说明

参数	输入/输出	说明
op_src	输入	protobuf格式的数据结构（来源于caffe模型的prototxt文件），包含算子参数信息。
op_dest	输出	davinci离线模型的算子数据结构，保存算子信息。

### 7.1.1.2 ParseWeights 函数

#### 函数功能

用户需要自定义继承CaffeTVMOpParser的类，并重写CaffeTVMOpParser类的成员函数ParseWeights函数，实现将caffe模型caffemodel中的算子权值参数映射到离线模型。



说明

仅包含权值的算子需要重写此接口。

#### 函数原型

```
Status ParseWeights(const Message* op_src, OpDef* op_dest);
```

#### 参数说明

参数	输入/输出	说明
op_src	输入	protobuf格式的数据结构（来源于caffe模型的caffemodel文件），包含算子权值信息。
op_dest	输出	davinci离线模型的算子数据结构，保存算子信息。

### 7.1.1.3 ConvertWeight 函数

#### 函数功能

此函数不需要重写。

用户在重写ParseWeights函数时，需要调用ConvertWeight函数将caffe算子的权值数据，映射到davinci离线模型。

#### 函数原型

```
static Status ConvertWeight(const BlobProto& proto, WeightDef* weight);
```

#### 参数说明

参数	输入/输出	说明
proto	输入	ParseWeights里的op_src参数里的一个成员变量，保存算子权值。
weight	输出	davinci离线模型的算子权值数据结构，保存权值信息。

### 7.1.1.4 ConvertShape 函数

#### 函数功能

此函数不需要重写。

用户在重写ParseWeights函数时，需要调用ConvertWeight函数。

在ConvertWeight函数中会调用ConvertShape函数，将caffe模型的权值shape参数映射到davinci离线模型。

#### 函数原型

```
static void ConvertShape(const BlobProto& proto, ShapeDef* shape);
```

#### 参数说明

参数	输入/输出	说明
proto	输入	保存转换前的Shape信息。
shape	输出	保存转换后的Shape信息。

### 7.1.1.5 GetOutputDesc 函数

#### 函数功能

用户需要自定义继承TVMOpBuilder的类，并重写GetOutputDesc函数，用于获取算子的输出描述。

#### 函数原型

```
virtual Status GetOutputDesc(vector<TensorDescriptor>& v_output_desc);
```

#### 参数说明

参数	输入/输出	说明
v_output_desc	输出	存储算子的输出描述。

### 7.1.1.6 BuildTvmBinFile 函数

#### 函数功能

用户自定义继承TVMOpBuilder的类，重写BuildTvmBinFile函数，用于构建算子二进制文件。



## 函数原型

```
virtual Status BuildTvmBinFile(TVMBinInfo& tvm_bin_info);
```

## 参数说明

参数	输入/输出	说明
tvm_bin_info	输出	存储自定义算子二进制文件路径和ddk描述的数据。

### 7.1.1.7 算子构造函数注册宏

## 宏功能

调用DOMI\_REGISTER\_OP完成算子注册。

## 宏原型

```
DOMI_REGISTER_OP(name)
```

## 参数说明

参数	输入/输出	说明
name	输入	算子在davinci模型的类型名，可以随意指定，不可与已有类型名重复，区分大小写。 <b>说明</b> 已有类型请参见 <a href="#">7.2 框架内置算子列表</a> 。

### 7.1.1.8 调用示例

请参见[4.4 算子插件开发](#)。

## 7.1.2 IR 方式

### 7.1.2.1 Attr 函数

## 函数功能

用户自定义继承Operator的类，调用Attr函数，将属性值写入到离线模型对象中。该函数有三种函数原型，仅写入的格式不同，功能一致。

## 函数原型

```
Operator& Attr(const std::string& name, const TypeName& value);
```

```
Operator& Attr(const std::string& name, const std::vector< TypeName >& value);
Operator& Attr(const std::string& name, const domi::Tuple< TypeName >& value);
```

## 参数说明

参数	输入/输出	说明
name	输入	离线模型的属性名。
value	输入	从外部框架（如：caffe模型）获取的属性值，写入到离线模型对象中。

### 7.1.2.2 ParseParams 函数

#### 函数功能

用户自定义ParseParams函数，完成caffe模型参数和权值的转换，将结果填到自定义类中（如：CustomOperator类，该自定义类需要继承自Operator类）。

#### 函数原型

```
domi::Status ParseParams(const ::caffe::LayerParameter* layer, CustomOperator* op);
```

其中CustomOperator是用户自己扩展的Operator类。

## 参数说明

参数	输入/输出	说明
layer	输入	caffe模型的算子描述，由框架自动获取。
op	输出	用户自定义的Operator数据结构。

### 7.1.2.3 GetOutputDesc 函数

#### 函数功能

用户自定义继承TVMOpBuilder的类，并重写GetOutputDesc函数，用于获取算子的输出描述。

#### 函数原型

```
virtual Status GetOutputDesc(vector<TensorDescriptor>& v_output_desc);
```

## 参数说明

参数	输入/输出	说明
v_output_desc	输出	存储算子的输出描述。

### 7.1.2.4 BuildTvmBinFile 函数

#### 函数功能

用户自定义继承TVMOpBuilder的类，并重写BuildTvmBinFile函数，用于构建算子二进制文件。

#### 函数原型

```
virtual Status BuildTvmBinFile(TVMBinInfo& tvm_bin_info);
```

#### 参数说明

参数	输入/输出	说明
tvm_bin_info	输出	存储自定义算子二进制文件路径和ddk描述的数据。

### 7.1.2.5 算子参数规格注册宏

#### 宏功能

调用DOMI\_OP\_SCHEMA注册算子参数规格。

#### 宏原型

```
DOMI_OP_SCHEMA(name)
```

#### 参数说明

参数	输入/输出	说明
name	输入	算子在davinci模型的类型名，可以随意指定，不可与已有类型名重复，区分大小写。 <b>说明</b> 已有类型请参见 <a href="#">7.2 框架内置算子列表</a> 。

### 7.1.2.6 算子解析函数注册宏

#### 宏功能

调用DOMI\_REGISTER\_CAFFE\_PARSER注册算子解析函数。

#### 宏原型

DOMI\_REGISTER\_CAFFE\_PARSER(name, param\_clazz)

#### 参数说明

参数	输入/输出	说明
name	输入	算子在davinci模型的类型名，可以随意指定，不可与已有类型名重复，区分大小写。 <b>说明</b> 已有类型请参见 <a href="#">7.2 框架内置算子列表</a> 。
param_clazz	输入	用户自定义的Operator类名。

### 7.1.2.7 算子构造函数注册宏

#### 宏功能

调用DOMI\_REGISTER\_OP完成算子注册。

#### 宏原型

DOMI\_REGISTER\_OP(name)

#### 参数说明

参数	输入/输出	说明
name	输入	算子在davinci模型的类型名，可以随意指定，不可与已有类型名重复，区分大小写。 <b>说明</b> 已有类型请参见 <a href="#">7.2 框架内置算子列表</a> 。

### 7.1.2.8 调用示例

#### 7.1.2.8.1 用户自定义类

```
class CaffeReductionOperator : public Operator
{
public:
    CaffeReductionOperator();
    ~CaffeReductionOperator();
};
```

```
CaffeReductionOperator& Name(std::string name);

CaffeReductionOperator& Operation(std::string operation);

CaffeReductionOperator& Axis(int64_t axis);

CaffeReductionOperator& Coeff(float coeff);

std::string GetOperation() const;

int64_t GetAxis() const;

float GetCoeff() const;

};
```

#### 7.1.2.8.2 调用 Attr 函数

```
CaffeReductionOperator::CaffeReductionOperator()
: Operator("caffe_reduction_layer")
{
}

CaffeReductionOperator::~CaffeReductionOperator()
{
}

CaffeReductionOperator& CaffeReductionOperator::Name(std::string name)
{
    Operator::Name(name);
    return *this;
}

CaffeReductionOperator& CaffeReductionOperator::Operation(std::string operation)
{
    Attr("operation", operation);
    return *this;
}

CaffeReductionOperator& CaffeReductionOperator::Axis(int64_t axis)
{
    Attr("axis", axis);
    return *this;
}

CaffeReductionOperator& CaffeReductionOperator::Coeff(float coeff)
{
    Attr("coeff", coeff);
    return *this;
}

std::string CaffeReductionOperator::GetOperation() const
{
    return GetStringAttr("operation");
}

int64_t CaffeReductionOperator::GetAxis() const
{
    return GetIntAttr("axis");
}

float CaffeReductionOperator::GetCoeff() const
{
    return GetFloatAttr("coeff");
}
```

#### 7.1.2.8.3 自定义 ParseParams 函数

```
domi::Status ParseParams(const ::caffe::LayerParameter* layer, CaffeReductionOperator* op)
{
    // #### 算子操作类型与其字符串对应的MAP
    std::map<::caffe::ReductionParameter_ReductionOp, std::string> operation_map = {
        { caffe::ReductionParameter_ReductionOp_SUM, "SUM" },
        { caffe::ReductionParameter_ReductionOp_ASUM, "ASUM" },
        { caffe::ReductionParameter_ReductionOp_SUMSQ, "SUMSQ" },
    };
```

```
{ caffe::ReductionParameter_ReductionOp_MEAN, "MEAN" },
};
// ##### 获取算子参数
const ::caffe::ReductionParameter& param = layer->reduction_param();

// ##### 设置参数至op中

op->Operation(operation_map[param.operation()]);
op->Axis((int32_t)param.axis());
op->Coeff(param.coeff());
return SUCCESS;
}
```

#### 7.1.2.8.4 重写 GetOutputDesc 函数和 BuildTvmBinFile 函数

```
// ##### caffe_reduction_layerOpBuilder是构建器的类名，可以随意指定，只要最后注册时使用此类名即可
class caffe_reduction_layerOpBuilder : public TVMOpBuilder
{
public:
    using TVMOpBuilder::TVMOpBuilder;

    ~caffe_reduction_layerOpBuilder()
    {
    }

    // ##### 获取输出tensor描述的处理函数
    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));

        int32_t axis = -1;
        // ### 解析axis参数
        if (!GetOpAttr("axis", &axis, op_def_))
        {
            printf("GetOpAttr axis failed!\n");
        }

        // 由于om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
        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;
        }
        v_output_desc[0].set_dim(axis, 1);

        return SUCCESS;
    }

    virtual Status BuildTvmBinFile(TVMBinInfo& tvm_bin_info) override
    {
        // TODO:
        // 1. call tvm python api to build bin files
        // 2. set path to TVMBinInfo tvm_bin_info
        PyObject *pModule = NULL;
        PyObject *pFunc = NULL;
        PyObject *pArg = NULL;
        PyObject *result = NULL;

        // ### python初始化
        Py_Initialize();

        pModule = PyImport_ImportModule("topi.cce");
        if (!pModule) {
```

```
PyErr_Print();
printf("import topi.cce failed!\n");
return FAILED;
}

// set ddk version
pFunc= PyObject_GetAttrString(pModule, "te_set_version");
pArg= PyTuple_Pack(1,PyString_FromString(tvm_bin_info.ddk_version.c_str()));
result = PyObject_CallObject(pFunc, pArg);

if ( !result ) {
    PyErr_Print();
    printf("call te_set_version failed!\n");
    return FAILED;
}

string res_msg = PyString_AsString(result);

if ( res_msg != "success" ) {
    printf("call te_set_version failed,msg=%s", res_msg.c_str());
    return FAILED;
}

// ### 导入topi.cce模块
PyRun_SimpleString("import sys");
PyRun_SimpleString("sys.path.append('../python')");
pModule = PyImport_ImportModule("caffe_reduction_layer");
if (!pModule)
{
    PyErr_Print();
    printf("import topi.cce failed!\n");
    return FAILED;
}

std::string operation;
int32_t axis = -1;
float coeff = 1;
// ### 解析operation参数
if (!GetOpAttr("operation", &operation, op_def_))
{
    // ### 增加异常处理、可维护信息
    printf("GetOpAttr operation failed!\n");
}

// ### 解析axis参数
if (!GetOpAttr("axis", &axis, op_def_))
{
    printf("GetOpAttr axis failed!\n");
}

// 由于om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
if (axis < 0) axis -= 2;

// ### 解析coeff参数
if (!GetOpAttr("coeff", &coeff, op_def_))
{
    printf("GetOpAttr coeff failed!\n");
}

// ### 解析输入tensor描述
TensorDescriptor input_desc = op_def_>input_desc(0);

// ### 解析输入shape值, 校验其大小是否为4
if (input_desc.dim_size() != 4)
{
    printf("The shape size is %d, which is not 4!", input_desc.dim_size());
    return FAILED;
}

// 调用Python接口生成Tvm算子
```

```
pFunc = PyObject_GetAttrString(pModule, "caffe_reduction_layer_cce");
pArg = Py_BuildValue(
    "(i,i,i,i), s, i, s, f, s",
    input_desc.dim(0), input_desc.dim(1), input_desc.dim(2), input_desc.dim(3), "float16",
    axis, operation.c_str(), coeff, "cce_reductionLayer_1_10_1_1_float16_3_SUMSQ_1_0");

PyEval_CallObject(pFunc, pArg);

// ### 结束python调用
Py_Finalize();

// 设置算子文件路径及kernel name
tvm_bin_info.bin_file_path = "./kernel_meta/
cce_reductionLayer_1_10_1_1_float16_3_SUMSQ_1_0.o";
tvm_bin_info.json_file_path = "./kernel_meta/
cce_reductionLayer_1_10_1_1_float16_3_SUMSQ_1_0.json";
return SUCCESS;
}

virtual Status TransWeights(size_t& mem_offset) override
{
    // TODO:
    // trans weights and calculate weights memory size
    return SUCCESS;
}
```

### 7.1.2.8.5 算子注册

```
// 注册算子参数规格
DOMI_OP_SCHEMA(caffe_reduction_layer)
    .Input("x")
    .Output("y")
    .AttrRequired("operation", AttributeType::STRING)
    .AttrRequired("axis", AttributeType::INT)
    .AttrRequired("coeff", AttributeType::FLOAT);
// 注册算子解析函数
DOMI_REGISTER_CAFFE_PARSER("caffe_reduction_layer", CaffeReductionOperator)
    .SetParseParamsFn(ParseParams);
// 注册算子构造函数
DOMI_REGISTER_OP("caffe_reduction_layer") // ##### test_reduction是算子在om模型的类型名，可以随意
指定，不可与已有类型名重复，区分大小写
    .FrameworkType(CAFFE) // ##### 枚举类型，CAFFE, TENSORFLOW
    .OriginOpType("Reduction") // ##### Reduction111表示该算子在caffe框架中的类型名
    .BuilderCreatorFn(BUILDER_FN(caffe_reduction_layerOpBuilder)) // #####
caffe_reduction_layerOpBuilder对应上面的类名
    .ImplyType(ImplyType::TVM); // ##### 实现类型，当前只支持TVM
```

## 7.2 框架内置算子列表

- Input
- Convolution
- Correlation
- Correlation\_V2
- Deconvolution
- Pooling
- InnerProduct
- Scale
- BatchNorm



- Eltwise
- ReLU
- Sigmoid
- AbsVal
- TanH
- PReLU
- Softmax
- Reshape
- ConvolutionDepthwise
- Dropout
- Concat
- ROIPooling
- FSRDetectionOutput
- Detectpostprocess
- LRN
- Proposal
- Flatten
- PriorBox
- Normalize
- Permute
- NetOutput
- SSDDetectionOutput
- ChannelAxy
- PSROIPooling
- Power
- ROIAlign
- FreespaceExtract
- SpatialTransform
- ProposalSigmoid
- Slice
- Region
- Yolo
- YoloDetectionOutput
- Reorg
- Reverse
- Crop