CANN
6.3.RC2

TensorFlow Parser Scope 融合规则开发指南

文档版本 01

发布日期 2023-07-26

华为技术有限公司

版权所有 © 华为技术有限公司 2023。 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


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暗示的声明或保证。

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。

目 录

 版权所有 © 华为技术有限公司

 版权所有 © 华为技术有限公司

 版权所有 © 华为技术有限公司

 版权所有 © 华为技术有限公司

 版权所有 © 华为技术有限公司

 版权所有 © 华为技术有限公司

 版权所有 © 华为技术有限公司

1 概述

1.1 什么是 Scope 融合

为了实现高性能的计算，往往需要对子图中的小算子进行融合，使得融合后的大算子（多对一场景）或小算子组合（多对多场景）可以充分利用硬件加速资源。

在TensorFlow框架中，通过TensorFlow的作用域函数tf.name_scope()，可以将不同的对象及操作放在由tf.name_scope()指定的作用域中，便于在tensorboard中展示清晰的逻辑关系图：

```
import tensorflow as tf;
tf.reset_default_graph()

#定义一块名为xx_name_scope的区域，并在其中工作
with tf.name_scope('xx_name_scope'):
    a = tf.constant(1,name='my_a')
    b = tf.Variable(2,name='my_b')
    c = tf.add(a,b,name='my_add')
print("a.name = "+a.name)
print("b.name = "+b.name)
print("c.name = "+c.name)

# 保存graph用于tensorboard绘图
with tf.Session() as sess:
  sess.run(tf.global_variables_initializer())
  writer = tf.summary.FileWriter("./test",sess.graph)
  print(sess.run(c))
writer.close()
```

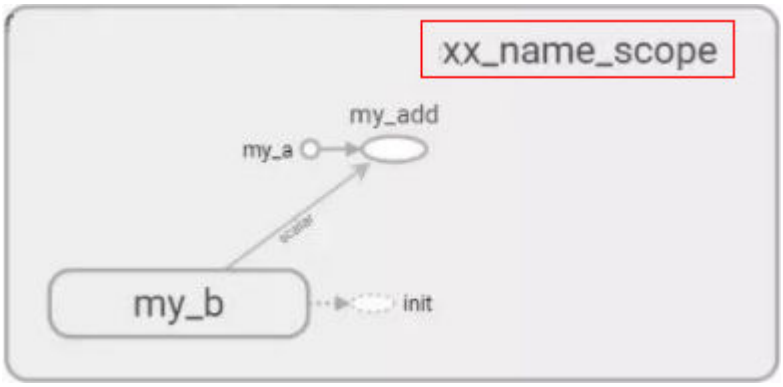
输出结果：

```
a.name = xx_name_scope/my_a:0
b.name = xx_name_scope/my_b:0
c.name = xx_name_scope/my_add:0
```

从输出结果可以看出，在tf.name_scope()下的所有对象和操作，其name属性前都加了xx_name_scope，用以表示这些内容全在其范围下。

 版权所有 © 华为技术有限公司

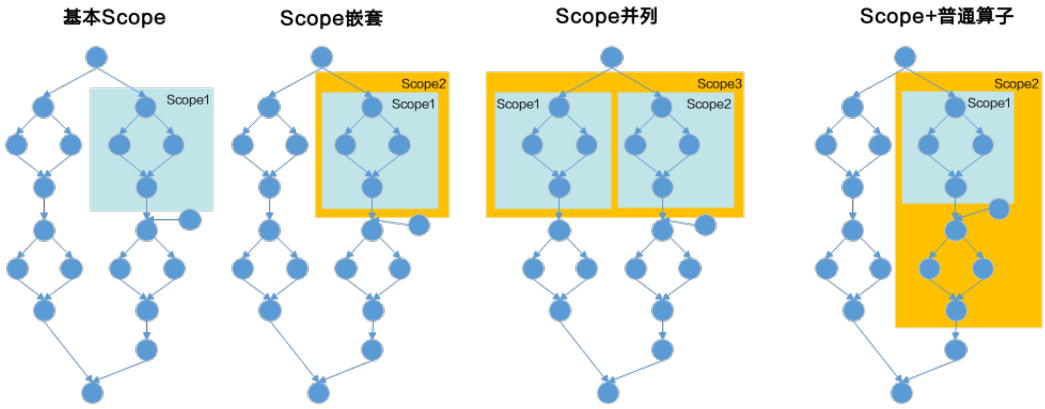
从tensorboard中也可以看到Scope信息：



Scope融合是一种基于Scope来进行融合的能力，把Scope内的多个小算子替换为一个大算子或多个算子组合，以实现效率的提升。例如将tf.layernorm生成的layernorm/batchnorm和layernorm/moments这两个Scope，融合为LayerNorm算子。

当前除了支持基本的Scope内所有算子融合外，还支持更加灵活的融合配置， 例如嵌套式识别、并列式识别、Scope+普通算子识别，如图1-1所示。

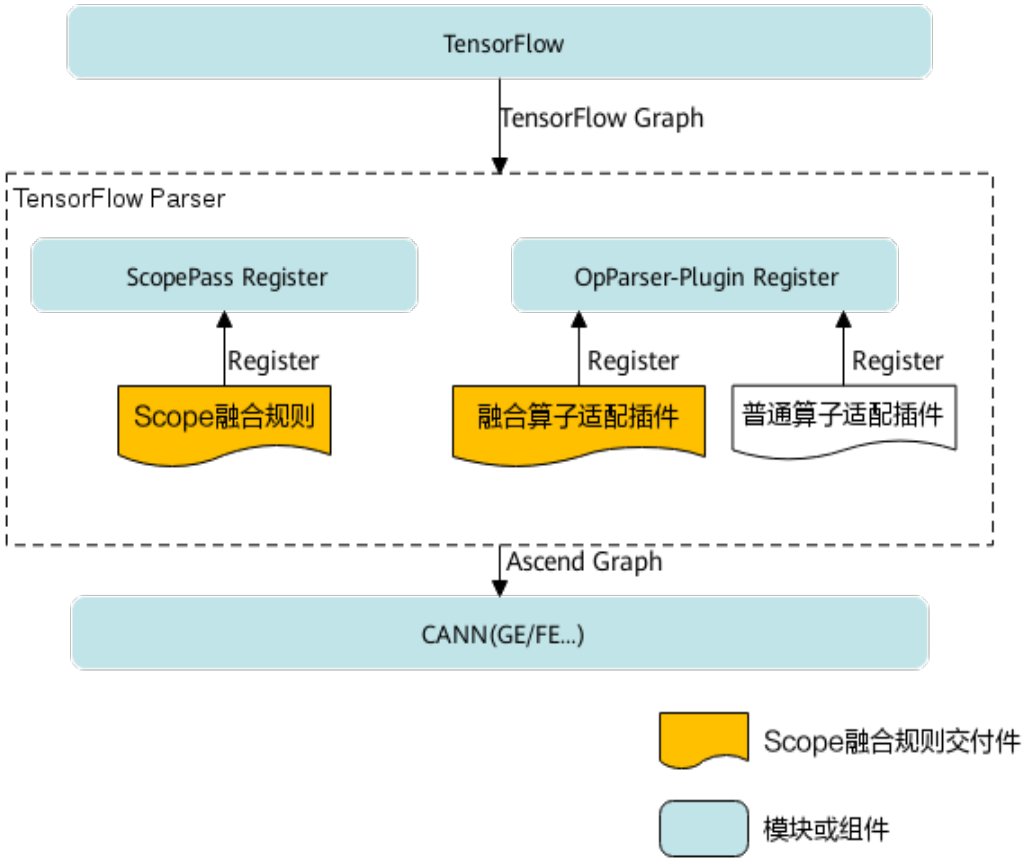
图 1-1 Scope 融合支持的场景



 版权所有 © 华为技术有限公司

1.2 Scope 融合实现方案

Scope 融合实现方案



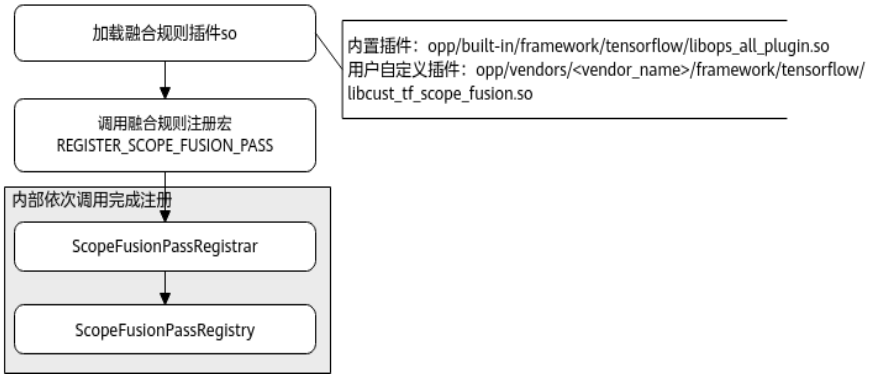
- ScopePass Register：用于融合规则注册。
- OpParser-Plugin Register：用于融合算子Parser注册，将Scope内的TensorFlow算子映射成适配昇腾AI处理器的融合算子。普通算子映射也是由该模块完成。

Scope 融合规则注册

在ATC模型转换或者在TensorFlow框架内执行图时，系统会将“Ascend-cann-toolkit安装目录/ascend-toolkit/latest/opp/built-in/framework/”下的融合规则插件so（包括所有内置融合规则和自定义融合规则）自动加载到ScopePass Register。在后续Parser执行阶段可根据该融合规则的使能状态，决定是否创建并执行。

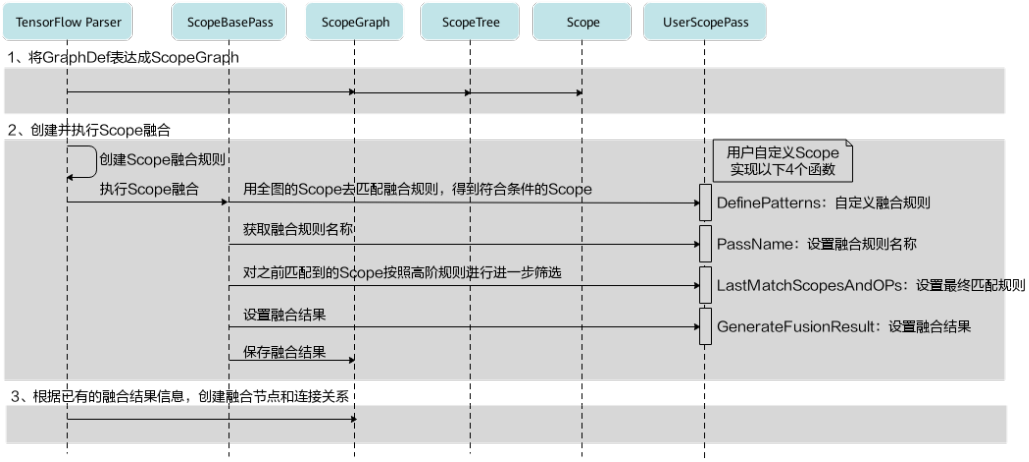
 版权所有 © 华为技术有限公司

图 1-2 Scope 融合规则注册机制



Scope 融合规则创建和执行

图 1-3 Scope 融合规则执行流程



1. TensorFlow Parser在模型解析过程中， 调用ScopePassManager提供的能力， 将GraphDef里的Node信息、Scope信息用Scope融合的数据类型表示出来，从上到下的层级为ScopeGraph、ScopeTree、Scope，生成ScopeGraph。
2. TensorFlow Parser创建并执行Scope融合，最后将最终匹配结果保存到ScopeGraph。主要包括以下几步：
   a. TensorFlow Parser根据已注册的融合规则的使能状态，创建Scope融合规则。
   b. 获取已创建的Scope融合规则，用全图的Scope，按照融合规则注册的先后顺序，逐一去匹配这些规则，如果匹配到则设置对应Scope的类型。

      说明

      如果用户自定义融合规则的名称和内置融合规则名称一样，则按照用户自定义融合规则匹配。

   c. 对上一步匹配到的Scope按Scope的连接关系等进行进一步筛选。例如上一步得到的Scope并不一定是最终目标融合的Scope，而需要筛选出有并列关系的Scope，或者筛选出有嵌套关系的Scope等。
   d. 对最终匹配到的Scope设置融合结果， 包含融合节点的名字、类型、输入、输出、描述等。

 版权所有 © 华为技术有限公司

e. 将融合结果保存到第1步生成的ScopeGraph中。

3. TensorFlow Parser后续流程会根据融合结果信息，执行添加节点、连接节点关系、构造IR Graph等操作。

关键数据结构

ScopeGraph、ScopeTree等数据结构提供了Scope融合所需的各种能力和数据保存功能，接口定义请参见“Ascend-cann-toolkit安装目录/ascend-toolkit/latest/”目录下的“/include/register/scope/scope_fusion_pass_register.h”，关键类的作用简介请参考**表1-1**。

表 1-1 关键类的作用简介

类名	作用简介	参考文档
Scope	Scope类型和属性定义。	**6.1.2 Scope类**
ScopeTree	存储所有Scope信息的一个树结构，可以查询Scope相关的子Scope和Node信息。	**6.1.4 ScopeTree类**
ScopeGraph	包含ScopeTree，同时定义了Scope关系匹配计算的成员函数，以及Scope融合识别的最终结果。	**6.1.5 ScopeGraph类**
ScopePattern	Scope匹配规则，目前主要包括以下三种： • NodeOpTypeFeature：基于scope中某一类型算子的个数或者个数的倍数匹配。 • NodeAttrFeature：基于scope中某一类型算子的某一属性的值匹配。 • ScopeFeature：基于scope自身或者其子scope的特征匹配。	**6.1.11 ScopePattern类** **6.1.8 NodeOpTypeFeature类** **6.1.9 NodeAttrFeature类** **6.1.10 ScopeFeature类**
ScopeBaseFeature	以上三种匹配规则的基类，定义三种匹配规则的基本操作。	**6.1.7 ScopeBaseFeature类**
ScopeBasePass	用户自定义融合规则的基类，提供接口定义、通用执行流程、与通用规则匹配流程的实现。	**6.1.13 ScopeBasePass类**
ScopesResult	用于保存经过进一步匹配和筛选后最终留下来的Scope。	**6.1.12 ScopesResult类**
FusionScopesResult	保存融合结果， 包括设置融合算子名称、类型、输入、输出、描述、内部算子组合信息（多对多场景）等。	**6.1.3 FusionScopesResult类**
ScopeUtil	提供通用工具类函数。	**6.1.14 ScopeUtil类**

 版权所有 © 华为技术有限公司

类名	作用简介	参考文档
ScopeAttrValue	在**NodeAttrFeature**规则定义中使用的数据结构，用于定义属性相关的规则。	**6.1.6 ScopeAttrValue类**

1.3 Scope 融合规则分类

系统提供内置的Scope融合规则，详细请参考《**TensorFlow Parser Scope融合规则参考**》，同时开放接口供用户自定义，本文主要介绍如何自定义Scope融合规则。

无论是内置还是自定义融合规则，都分为如下两类：

- 通用融合规则（General）：各网络通用的Scope融合规则；默认生效，不支持用户指定失效。
- 定制化融合规则（Non-General）：特定网络适用的Scope融合规则；默认不生效，在ATC模型转换或者在TensorFlow框架内执行图时，用户可以使能相应的融合规则，使之生效。

1.4 约束与限制

- Scope融合仅支持TensorFlow模型，不支持其他框架模型；通过算子原型构图时，不支持Scope融合；通过Tensorflow Parser解析方式构图时(具体操作可参考《**Ascend Graph开发指南**》中的"原始模型转换为Graph"章节)，支持Scope融合。
- 用户在设计Scope融合规则时，应保持唯一性，避免图中同一个Scope匹配到多条融合规则。

 版权所有 © 华为技术有限公司

2 环境准备

参见《**CANN 软件安装指南**》进行开发环境搭建，并确保开发套件包Ascend-cann-toolkit安装完成。

- 软件包提供了Scope融合规则开发接口，接口定义文件所在路径：“Ascend-cann-toolkit安装目录/ascend-toolkit/latest/compiler/include/register/scope/scope_fusion_pass_register.h”。
- 软件包提供了融合后的算子原型，算子原型定义文件所在路径：“Ascend-cann-toolkit安装目录/ascend-toolkit/latest/opp/built-in/op_proto/inc/”。
- 最终编译生成的Scope融合规则插件so存放路径：“Ascend-cann-toolkit安装目录/ascend-toolkit/latest/opp/built-in/framework/custom/”。

 版权所有 © 华为技术有限公司

3 融合规则开发

3.1 开发流程及关键接口

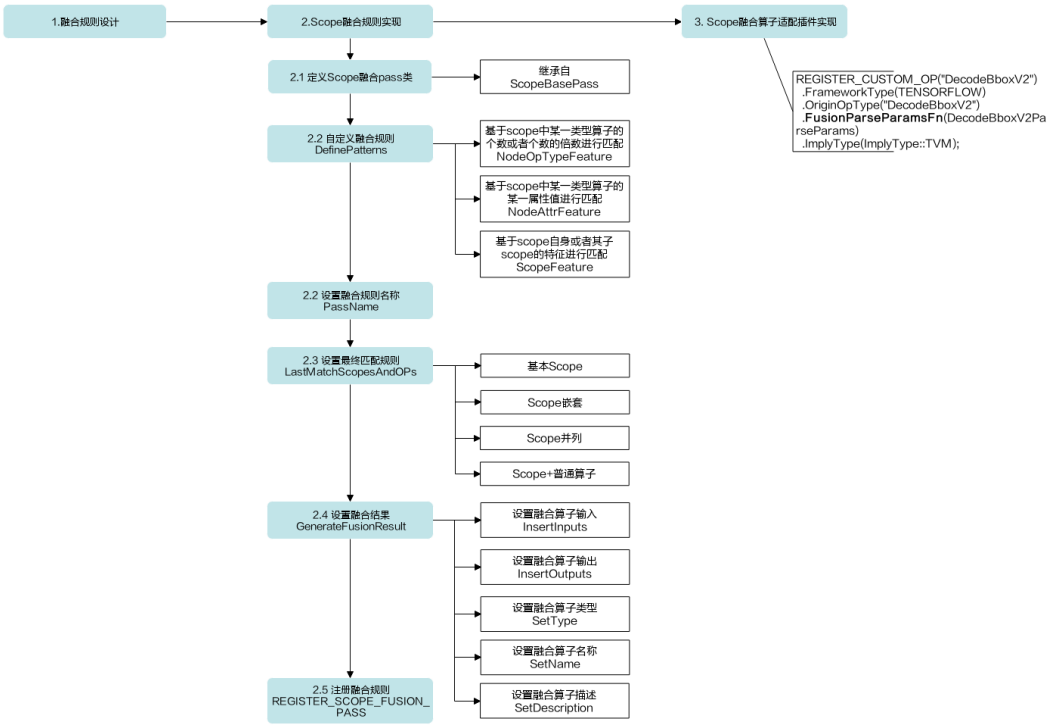
Scope融合按照融合场景区分为多对一融合场景和多对多融合场景。本节介绍两种场景的开发流程及关键接口。

Scope 多对一融合场景

Scope多对一融合场景，即把Scope内的多个小算子融合为一个大算子。

 版权所有 © 华为技术有限公司

图 3-1 多对一场景实现流程

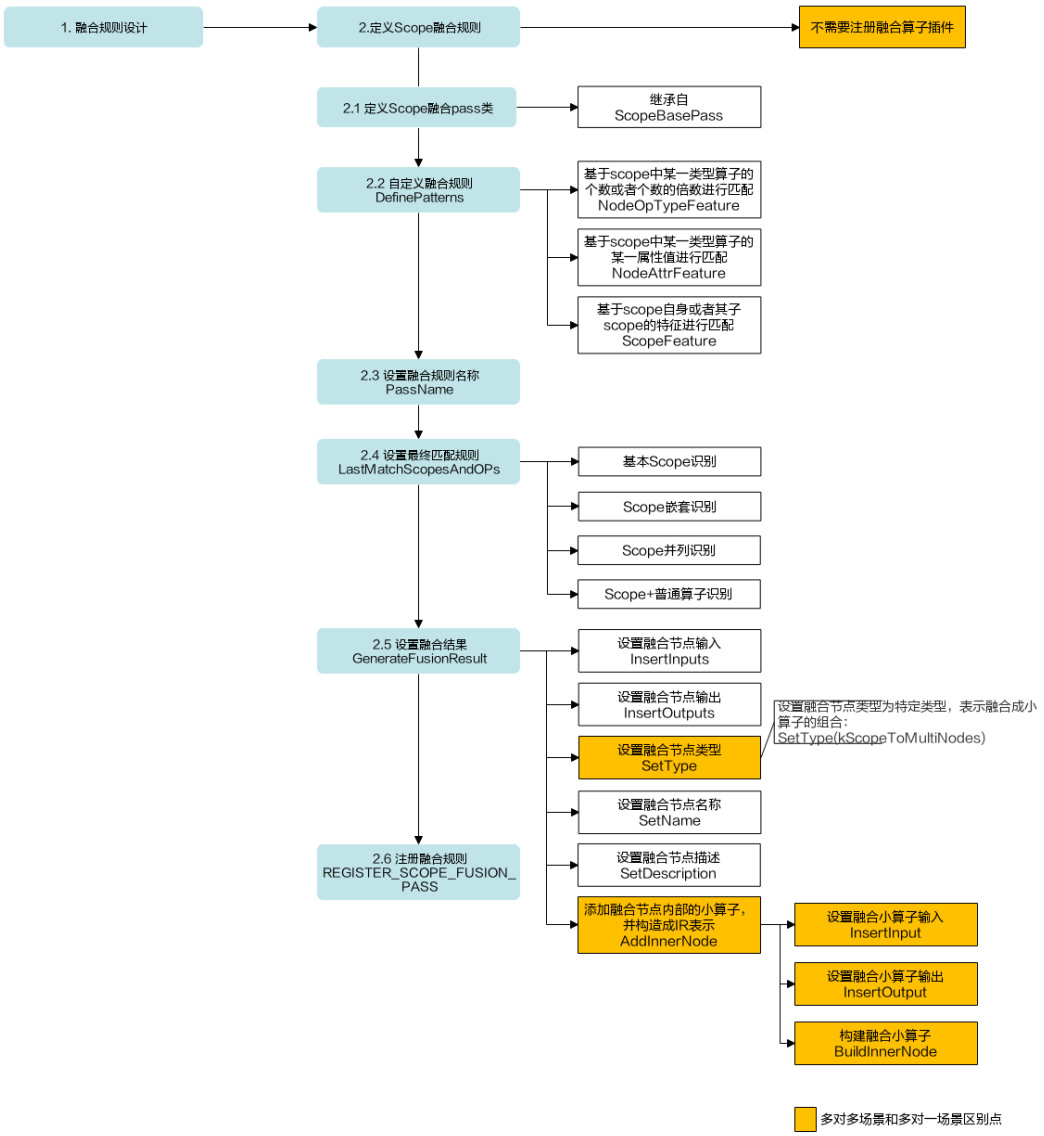


Scope 多对多融合场景

Scope多对多融合场景，即把Scope内的多个小算子融合为一系列小算子组合。

 版权所有 © 华为技术有限公司

图 3-2 多对多场景实现流程



多对多融合场景和多对一融合场景大部分相同，主要区别为：

1. 在融合结果设置函数**6.1.13.5 GenerateFusionResult**中，需要设置内部小算子组合的连接关系。
2. 内部小算子后面将不再走插件解析， 因此小算子需用户自行构造成IR表示，并设置好所需属性等参数。

下面章节，主要以Scope融合对多一场景为例，介绍如何进行融合规则设计和融合规范开发。

3.2 融合规则设计

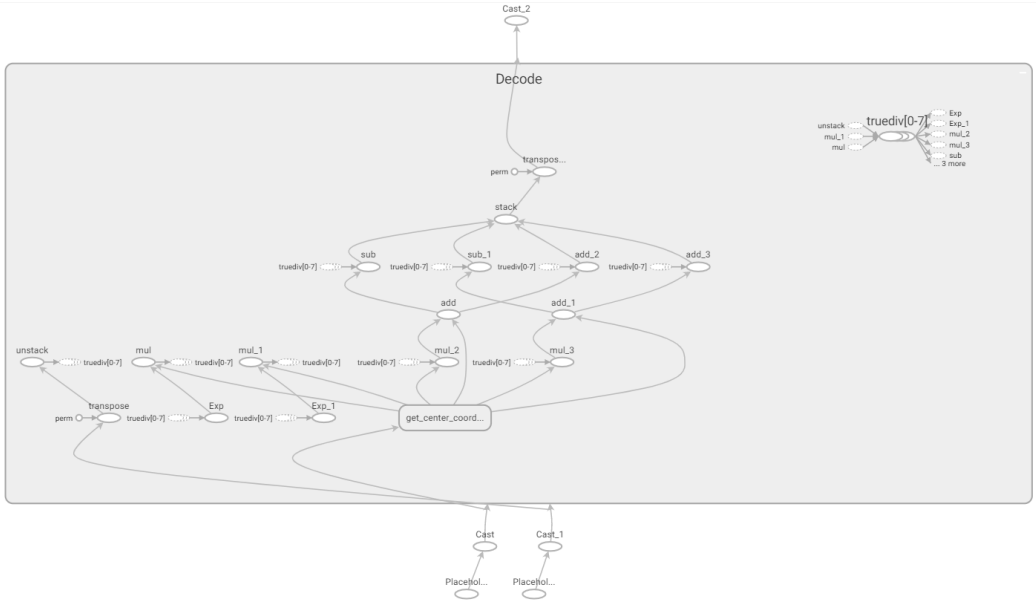
明确融合方案

下面以多对一融合规则DecodeBboxV2ScopeFusionPass为例介绍如何设计和开发Scope融合规则。

 版权所有 © 华为技术有限公司

该融合规则的目标是把Decode scope下的所有小算子融合为一个大算子DecodeBboxV2，Scope内包括2个Exp算子/4个Mul算子/4个Sub算子/2的倍数个RealDiv算子/2个Unpack算子/1个Pack算子/3个Transpose算子/不能包括Softmax算子，如图3-3所示。

图 3-3 融合示意图



明确融合算子原型

DecodeBboxV2融合算子原型为：

算子原型API可以参考《**Ascend Graph开发指南**》的"参考-> Ascend Graph接口参考 -> 原型定义接口"。

```
REG_OP(DecodeBboxV2)
    .INPUT(boxes, TensorType({DT_FLOAT16,DT_FLOAT}))
    .INPUT(anchors, TensorType({DT_FLOAT16,DT_FLOAT}))
    .OUTPUT(y, TensorType({DT_FLOAT16,DT_FLOAT}))
    .ATTR(scales, ListFloat, {1.0, 1.0, 1.0, 1.0})
    .ATTR(decode_clip, Float, 0.0)
    .ATTR(reversed_box, Bool, false)
    .OP_END_FACTORY_REG(DecodeBboxV2)
```

明确融合对应关系

如图3-3所示，

- transpose的第0个输入作为DecodeBboxV2的第0个输入boxes；
- get_center_coordinates_and_sizes/transpose的第0个输入作为DecodeBboxV2的第1个输入anchors；
- transpose_1的输出作为DecodeBboxV2的输出y。

 版权所有 © 华为技术有限公司

3.3 创建代码目录

用户可以直接单击**Link**，下载配套版本的sample包，从“samples/cplusplus/level1_single_api/4_op_dev/1_custom_op”目录中获取样例。并对其中的样例进行修改，追加自己的自定义融合规则。工程目录结构如下：

```
├─── framework                  // 算子插件实现文件目录
│    ├─── CMakeLists.txt
│    ├─── tf_plugin
│        └─── decode_bbox_v2_scope_fussion_plugin.cpp  // Scope融合算子适配插件实现文件，仅多对一场景需要
│        └─── CMakeLists.txt
│    ├─── tf_scope_fusion_pass
│        └─── decode_bbox_v2_scope_fusion_pass.h                  // Scope融合规则头文件
│        └─── decode_bbox_v2_scope_fusion_pass.cpp                // Scope融合规则实现文件
│        └─── CMakeLists.txt
```

3.4 Scope 融合规则实现

3.4.1 定义融合规则类

在融合规则头文件（例如decode_bbox_v2_scope_fusion_pass.h）中，定义Scope融合规则类， 继承自**6.1.13 ScopeBasePass类**。

```
#ifndef FRAMEWORK_TF_SCOPE_FUSION_PASS_DECODE_BBOX_V2_PASS_H_    // 条件编译
#define FRAMEWORK_TF_SCOPE_FUSION_PASS_DECODE_BBOX_V2_PASS_H_    // 宏定义

#include <string>
#include <vector>
#include "register/scope/scope_fusion_pass_register.h"

namespace ge {
    class DecodeBboxV2ScopeFusionPass : public ScopeBasePass {
    protected:
        std::vector<ScopeFusionPatterns> DefinePatterns() override;
        std::string PassName() override;
        Status LastMatchScopesAndOPs(std::shared_ptr<ScopeGraph> &scope_graph,
std::vector<ScopesResult> &results) override;
        void GenerateFusionResult(const std::vector<Scope *> &scopes, FusionScopesResult *fusion_rlt)
override;
    private:
        void GenScopePatterns(ScopeFusionPatterns &patterns);
    };
} // namespace ge
#endif  // FRAMEWORK_TF_SCOPE_FUSION_PASS_DECODE_BBOX_V2_PASS_H_ 结束条件编译
```

3.4.2 定义融合规则

在融合规则实现文件（例如decode_bbox_v2_scope_fusion_pass.cpp）中，定义Scope的融合规则，便于系统基于此规则在全图的Scope中匹配，找到符合相应规则的Scope。

主要过程为：

1. 包含头文件。
   ```
   #include "decode_bbox_v2_scope_fusion_pass.h"
   ```
2. 定义融合规则中使用到的常量。
   ```
   namespace ge {
       namespace {
   ```

 版权所有 © 华为技术有限公司

```
    const char *const kScopeType = "DecodeBboxV2FusionOp";      // 定义融合结果类型
    const char *const kScopeTypeDecodeBboxV2 = "DecodeBboxV2";  // 定义目标Scope类型
    const char *const kOpType = "DecodeBboxV2";                 // 用于日志打印
  } // namespace
...
} // namespace ge
```

融合规则实现文件中定义的其他所有函数均在namespace ge命名空间内。

3. 通过**6.1.13.2 DefinePatterns**自定义融合规则，用于在全图Scope中匹配到符合相应规则的Scope，目前支持以下三种模式：
   - 基于scope中某一类型算子的个数或者个数的倍数匹配，关键接口为**NodeOpTypeFeature**。

     例如要求Scope内含有2个Exp算子：

     ```
     decode_bbox_v2_pattern->AddNodeOpTypeFeature(NodeOpTypeFeature("Exp", 2, 0));      // Exp num is 2
     ```

   - 基于scope中某一类型算子的某一属性的值匹配，关键接口为**NodeAttrFeature**。

     例如要求Split算子的num_split属性值必须为4：

     ```
     basic_lstm_cell_sigmoid->AddNodeAttrFeature(NodeAttrFeature("Split", "num_split", ge::DT_INT32, 4));
     ```

   - 基于scope自身或者其子scope的特征匹配，关键接口为**ScopeFeature**。示例：

     根据Scope的名称匹配，例如要求Scope的LastName为while。

     ```
     p_lstm_relu_while->AddScopeFeature(ScopeFeature("", 0, "while"));
     ```

     根据其子Scope的类型和个数匹配，例如要求Scope内包含1个子scope，子scope类型为kLstmCellReluType。

     ```
     p_lstm_relu_while->AddScopeFeature(ScopeFeature(kLstmCellReluType, 1, ""));
     ```

     根据其子Scope的名称匹配，例如要求子scope的LastName中包含字符串“while”。

     ```
     p_lstm_relu_while->AddScopeFeature(ScopeFeature("", 0, "", "while"));
     ```

     将以上特征结合起来设定匹配规则，例如要求Scope的LastName为while，且包含1个子scope，子scope类型为kLstmCellReluType。

     ```
     p_lstm_relu_while->AddScopeFeature(ScopeFeature(kLstmCellReluType, 1, "while"));
     ```

下面示例中，通过**NodeOpTypeFeature**定义融合规则，要求Scope内包括2个Exp算子/4个Mul算子/4个Sub算子/2的倍数个RealDiv算子/2个Unpack算子/1个Pack算子/3个Transpose算子/不能包括Softmax算子。

```
// DecodeBboxV2ScopeFusionPass子类实现自定义融合规则
  std::vector <ScopeFusionPatterns> DecodeBboxV2ScopeFusionPass::DefinePatterns() {
    std::vector <ScopeFusionPatterns> patterns_list;
    ScopeFusionPatterns pattern;
    GenScopePatterns(pattern);
    patterns_list.push_back(pattern);
    return patterns_list;
  }
// 根据Scope关键特征，编写融合规则
  void DecodeBboxV2ScopeFusionPass::GenScopePatterns(ScopeFusionPatterns &patterns) {
    std::vector < ScopePattern * > batch;
    ScopePattern *decode_bbox_v2_pattern = new(std::nothrow) ScopePattern();
    if (decode_bbox_v2_pattern == nullptr) {
      OP_LOGE(kOpType, "Alloc an object failed.");
      return;
    }
    decode_bbox_v2_pattern->SetSubType(kScopeTypeDecodeBboxV2);                    // 设置Scope类型
    decode_bbox_v2_pattern->AddNodeOpTypeFeature(NodeOpTypeFeature("Exp", 2, 0));      // Exp num is 2
    decode_bbox_v2_pattern->AddNodeOpTypeFeature(NodeOpTypeFeature("Mul", 4, 0));      //
```

 版权所有 © 华为技术有限公司

```
Mul num is 4
        decode_bbox_v2_pattern->AddNodeOpTypeFeature(NodeOpTypeFeature("Sub", 4, 0));        //
Sub num is 4
        decode_bbox_v2_pattern->AddNodeOpTypeFeature(NodeOpTypeFeature("RealDiv", 0, 2));    //
RealDiv num is 2*n
        decode_bbox_v2_pattern->AddNodeOpTypeFeature(NodeOpTypeFeature("Unpack", 2, 0));     //
Unpack num is 2
        decode_bbox_v2_pattern->AddNodeOpTypeFeature(NodeOpTypeFeature("Pack", 1, 0));       //
Pack num is 1
        decode_bbox_v2_pattern->AddNodeOpTypeFeature(NodeOpTypeFeature("Transpose", 3, 0));  //
Transpose num is 3
        decode_bbox_v2_pattern->AddNodeOpTypeFeature(NodeOpTypeFeature("Softmax", -1, 0));   //
doesn't have Softmax

        OP_LOGI(kOpType, "Add GenScopePatterns DecodeBboxV2.");
        batch.push_back(decode_bbox_v2_pattern);
        patterns.push_back(batch);
    }
```

对于比较复杂的Scope，可以结合使用上面三种匹配方式，来定义Scope融合规则。

3.4.3 设置融合规则名称

通过**6.1.13.3 PassName**设置融合规则名称，本例为DecodeBboxV2ScopeFusionPass。

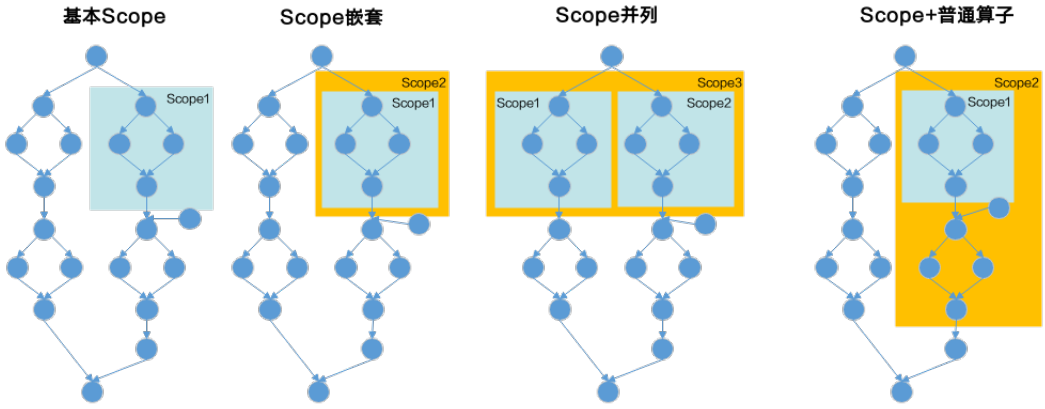
```
    std::string DecodeBboxV2ScopeFusionPass::PassName() { return
std::string("DecodeBboxV2ScopeFusionPass"); }
```

3.4.4 设置最终匹配规则

介绍

符合**3.4.2 定义融合规则**要求的Scope并不一定是最终的融合目标，比如需要进行并列Scope、嵌套Scope等的筛选和判断，如**图3-4**所示，此时通过**6.1.13.4 LastMatchScopesAndOPs**设置最终匹配规则，对匹配到的Scope进行进一步筛选，将符合的Scope保存到**ScopesResult**中。

图 3-4 Scope 融合支持的场景



 版权所有 © 华为技术有限公司

基本 Scope 匹配规则

本例中，期望融合的目标Scope的类型为kScopeTypeDecodeBboxV2，不需要设置更高阶的规则，直接找到scope保存到results返回即可，返回结果的类型为**6.1.12 ScopesResult类**。

```
  Status DecodeBboxV2ScopeFusionPass::LastMatchScopesAndOPs(shared_ptr <ScopeGraph>
&scope_graph,
                                                std::vector <ScopesResult> &results) {
    OP_LOGI(kOpType, "LastMatchScopesAndOPs start.");
    if (scope_graph == nullptr) {
      OP_LOGE(kOpType, "Input params is nullptr.");
      return FAILED;
    }
    const ScopeTree *scope_tree = scope_graph->GetScopeTree();
    if (scope_tree == nullptr) {
      OP_LOGE(kOpType, "Scope tree is nullptr.");
      return FAILED;
    }
    const std::vector<Scope *> &scopes = scope_tree->GetAllScopes();

    for (auto &scope : scopes) {
      // Class ScopeTree guarantees scope is not empty.
      AscendString op_subtype;
      Status ret = scope->SubType(op_subtype);
      if (ret != SUCCESS) {
        return FAILED;
      }
      AscendString op_name;
      ret = scope->Name(op_name);
      if (ret != SUCCESS) {
        return FAILED;
      }
      if (op_subtype == kScopeTypeDecodeBboxV2) {
        OP_LOGI(kOpType, "DecodeBbox LastMatchScopesAndOPs match scope %s.",
op_name.GetString());
        ScopesResult result;
        std::vector < Scope * > result_scopes;
        result_scopes.push_back(scope);
        result.SetScopes(result_scopes);
        results.push_back(result);
      }
    }
    return (!(results.empty())) ? SUCCESS : FAILED;
  }
```

Scope 并列匹配规则

用户还可以定义更复杂一些的Scope并列匹配规则，例如下面示例中，首先找到类型为kScopeTypeBatchnorm和kScopeTypeMoments的Scope，然后判断如果两个Scope在网络中的同一层，则进行融合。

```
/**
 * @brief LastMatch for multiple scopes
 */
Status ScopeLayerNormPass::LastMatchScopesAndOPs(std::shared_ptr<ScopeGraph>& scope_graph,
                                                 std::vector<ScopesResult>& results) {
  if (scope_graph == nullptr) {
    OP_LOGE(kOpType, "Input params is nullptr.");
    return domi::PARAM_INVALID;
  }
  const ScopeTree* scope_tree = scope_graph->GetScopeTree();
  if (scope_tree == nullptr) {
    OP_LOGE(kOpType, "Scope tree is nullptr.");
    return domi::PARAM_INVALID;
  }
  const std::vector<Scope*>& scopes = scope_tree->GetAllScopes();
```

 版权所有 © 华为技术有限公司

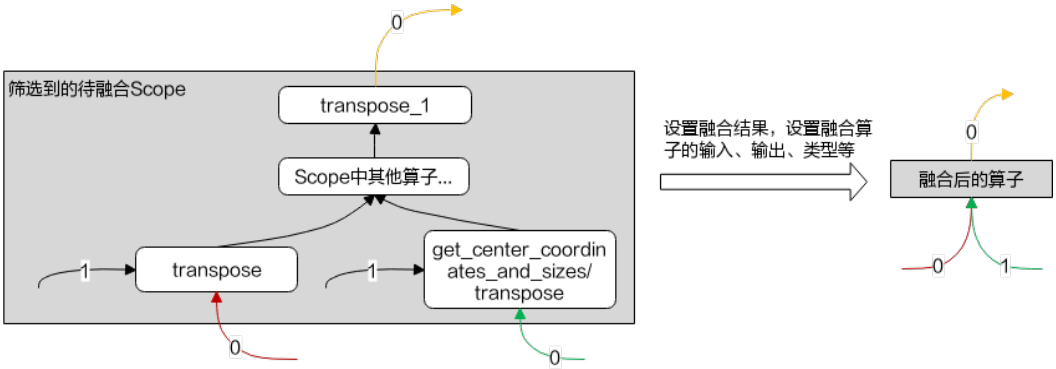
```
 std::vector<Scope*> fusion_scopes_bn;
 std::vector<Scope*> fusion_scopes_m;
 for (auto& scope : scopes) {
   // Class ScopeTree guarantees scope is not empty.
   AscendString op_subtype;
   Status ret = scope->SubType(op_subtype);
   if (ret != SUCCESS) {
       return FAILED;
   }
   if (op_subtype == kScopeTypeBatchnorm) {
     fusion_scopes_bn.push_back(scope);
   } else if (op_subtype == kScopeTypeMoments) {
     fusion_scopes_m.push_back(scope);
   }
 }

 if (fusion_scopes_bn.size() == fusion_scopes_m.size()) {
   // the two scope batchnorm and moments in the same layernorm
   for (size_t i = 0; i < fusion_scopes_bn.size(); i++) {
     auto scope_bn = fusion_scopes_bn[i];
     for (size_t j = 0; j < fusion_scopes_m.size(); j++) {
       auto scope_m = fusion_scopes_m[j];
       AscendString scope_bn_name;
       Status ret = scope_bn->Name(scope_bn_name);
       if (ret != SUCCESS) {
           return FAILED;
       }
       AscendString  scope_m_name;
       ret = scope_m->Name(scope_m_name);
       if (ret != SUCCESS) {
           return FAILED;
       }
       std::string scope_m_name_str;
       std::string scope_bn_name_str;
       if (scope_m_name.GetString() != nullptr) {
           scope_m_name_str = scope_m_name.GetString();
       }
       if (scope_bn_name.GetString() != nullptr) {
           scope_bn_name_str = scope_bn_name.GetString();
       }
       int pos_bn = scope_bn_name_str .find("batchnorm");
       int pos_m = scope_m_name_str .find("moments");
       int is_biggan_bn = scope_bn_name_str .find("resblock");
       int is_biggan_m = scope_m_name_str .find("resblock");
       if (is_biggan_bn != -1 || is_biggan_m != -1) {
         return FAILED;
       }
       if (pos_bn != -1 && pos_m != -1 && scope_bn_name_str.substr(0, pos_bn) ==
scope_m_name_str.substr(0, pos_m)) {
         // scope result
         ScopesResult result;
         std::vector<Scope*> result_scopes;
         result_scopes.push_back(scope_bn);
         result_scopes.push_back(scope_m);
         result.SetScopes(result_scopes);
         results.push_back(result);
         OP_LOGI(kOpType, "scope:%s, and scope:%s is connect.", scope_bn_name.GetString(),
scope_m_name.GetString());
         break;
       }
     }
   }
 }
 return (!(results.empty())) ? SUCCESS : FAILED;
}
```

 版权所有 © 华为技术有限公司

3.4.5 设置融合结果

通过6.1.13.5 GenerateFusionResult设置融合结果， 包含融合算子的名字、类型、输入、 输出、描述。最终融合结果保存在fusion_rlt返回，返回结果的类型为6.1.3 FusionScopesResult类。

图 3-5 融合结果示意图



1. 通过6.1.3.8 InsertInputs设置融合算子输入，例如：

```
fusion_rlt->InsertInputs("transpose", {0, kFusionDisableIndex});
```

   - 第一个参数表示融合算子的输入，即scope内部小算子name（除去scope名称的部分）。
   - 第二个参数表示输入index的映射，是一个vector类型。vector的index表示scope内部小算子的输入index，具体值表示融合算子的输入index，如果融合算子没有使用这个index，则使用占位符kFusionDisableIndex表示。

   表 3-1 示例说明

序号	代码示例
示例1	fusion_rlt->InsertInputs("transpose", {0, kFusionDisableIndex}); 表示将transpose的第0个输入作为融合算子的第0个输入，transpose的第1个输入不使用，使用占位符kFusionDisableIndex表示。
示例2	fusion_rlt->InsertInputs("transpose", {1, kFusionDisableIndex}); 表示将transpose的第0个输入作为融合算子的第1个输入，transpose的第1个输入不使用，使用占位符kFusionDisableIndex表示。
示例3	fusion_rlt->InsertInputs("transpose", {kFusionDisableIndex, 0}); 表示将transpose的第1个输入作为融合算子的第0个输入，transpose的第0个输入不使用，使用占位符kFusionDisableIndex表示。

2. 通过6.1.3.9 InsertOutputs设置融合算子输出，使用注意点和设置融合算子输入类似。例如：

```
// 设置融合算子输出， 将transpose_1的第0个输出作为融合算子的输出
fusion_rlt->InsertOutputs("transpose_1", {0});
```

3. 通过6.1.3.4 SetType设置融合算子的结果类型，例如：

```
// 设置融合算子的类型
fusion_rlt->SetType(kScopeType);
```

 版权所有 © 华为技术有限公司

需要注意的是，此处传入的类型需要和融合算子插件注册的OriginOpType保持一致：

```
REGISTER_CUSTOM_OP("DecodeBboxV2")
    .FrameworkType(TENSORFLOW)          // 原始框架为Tensorflow
    .OriginOpType("DecodeBboxV2FusionOp")   // 算子在原始框架中的类型，和GenerateFusionResult的SetType的内容保持一致
    .FusionParseParamsFn(DecodeBboxV2ParseParams)  // 用来注册解析融合算子属性的函数
    .ImplyType(ImplyType::TVM);         // 指定算子的实现方式，ImplyType::TVM表示该算子是TBE算子
```

如果识别出scope不满足条件，则不融合，可以设置type为**kScopeInvalidType**然后返回。

```
if (scopes.size() != 1) {
  fusion_rlt->SetType(kScopeInvalidType);
  return;
}
```

4. 通过**6.1.3.3 SetName**设置融合算子名称。需要注意的是，为保持融合算子名称全局唯一，建议尽量不要自行命名，可以根据scope的名称设置，例如：

```
// 设置融合算子的名称
AscendString scope_name;
Status ret = scopes[0]->Name(scope_name);
std::string scope_name_str;
if (scope_name.GetString() != nullptr) {
    scope_name_str = scope_name.GetString();
}
fusion_rlt->SetName(scope_name_str .substr(0, scope_name.length() - 1).c_str());
```

5. 通过**6.1.3.5 SetDescription**设置融合算子描述，例如：

```
// 设置融合算子的描述
fusion_rlt->SetDescription("");
```

完整代码示例为：

```
void CustomScopeDecodeBboxV2Pass::GenerateFusionResult(const std::vector<Scope *> &scopes,
FusionScopesResult *fusion_rlt) {
  if (fusion_rlt == nullptr) {
    return;
  }
  if (scopes.size() != 1) {
    fusion_rlt->SetType(kScopeInvalidType);  // 如果识别出scope不满足条件，可以设置type为kScopeInvalidType然后返回
    return;
  }

  // 设置融合算子输入，将transpose的第0个输入作为融合算子的第0个输入， transpose的第一个输入不使用
  fusion_rlt->InsertInputs("transpose", {0, kFusionDisableIndex});
  // 设置融合算子输入，将get_center_coordinates_and_sizes/transpose的第0个输入作为融合算子的第1个输入，get_center_coordinates_and_sizes/transpose的第一个输入不使用
  fusion_rlt->InsertInputs("get_center_coordinates_and_sizes/transpose", {1, kFusionDisableIndex});
  // 设置融合算子输出， 将transpose_1的第0个输出作为融合算子的输出
  fusion_rlt->InsertOutputs("transpose_1", {0});

  // 设置融合算子类型
  fusion_rlt->SetType(kScopeType);
  // 设置融合算子名称
  AscendString scope_name;
  Status ret = scopes[0]->Name(scope_name);
  if (ret != SUCCESS) {
      return;
  }
  std::string scope_name_str;
  if (scope_name.GetString() != nullptr) {
      scope_name_str = scope_name.GetString();
  }
  fusion_rlt->SetName(scope_name_str .substr(0, scope_name.length() - 1).c_str());
  // 设置融合算子描述
  fusion_rlt->SetDescription("");
  OP_LOGI(kOpType, "Set fusion result successfully.");
  return;
}
```

 版权所有 © 华为技术有限公司

3.4.6 注册融合规则

通过6.1.17 REGISTER_SCOPE_FUSION_PASS向GE注册Scope融合规则，例如：

```
REGISTER_SCOPE_FUSION_PASS("DecodeBboxV2ScopeFusionPass", DecodeBboxV2ScopeFusionPass, false);
```

- 第一个参数为融合规则名称。
- 第二个参数为融合规则类。
- 第三个参数为is_general。
  - 值为true表示通用规则，默认生效；
  - 值为false表示定制化规则，默认不生效，需要用户自行在图编译时通过参数enable_scope_fusion_passes配置生效。

3.5 Scope 融合算子适配插件实现

本节介绍如何实现融合算子适配插件，将基于原始框架的小算子映射成适配昇腾AI处理器的融合算子，并将算子信息注册到GE中。

在融合算子适配插件实现文件（例如decode_bbox_v2_scope_fussion_plugin.cpp）中，完成相应功能实现。

Scope融合算子Parser注册流程复用普通算子的注册逻辑，通过6.2.2.3 REGISTER_CUSTOM_OP宏，按照指定的算子名称完成算子的注册。

```
REGISTER_CUSTOM_OP("DecodeBboxV2")
    .FrameworkType(TENSORFLOW)              // 原始框架为Tensorflow
    .OriginOpType("DecodeBboxV2FusionOp")   // 算子在原始框架中的类型，和GenerateFusionResult的SetType的内容保持一致
    .FusionParseParamsFn(DecodeBboxV2ParseParams)  // 用来注册解析融合算子属性的函数
    .ImplyType(ImplyType::TVM);             // 指定算子的实现方式，ImplyType::TVM表示该算子是TBE算子
```

说明

更多介绍请参考《TBE&AI CPU算子开发指南》，本节仅介绍Scope融合算子Parser注册和普通算子注册的差异点。

Scope融合算子Parser注册和普通算子的差异点在于，注册Parser函数的接口由6.2.2.7 ParseParamsByOperatorFn变为6.2.2.9 FusionParseParamsFn（Overload）。原因是两个Parser函数入参不同：

- 普通算子注册函数6.2.2.7 ParseParamsByOperatorFn的回调函数原型：

  ```
  using ParseParamByOpFunc = std::function<domi::Status(const ge::Operator &, ge::Operator &)>;
  ```

  输入为框架定义的原始算子Operator类对象。更多介绍请参考《TBE&AI CPU算子开发指南》。
- Scope融合算子注册函数6.2.2.9 FusionParseParamsFn（Overload）的回调函数原型：

  ```
  using FusionParseParamByOpFunc = std::function<domi::Status(const std::vector<ge::Operator> &, ge::Operator &)>;
  ```

  输入为scope内部所有算子的NodeDef；输出为融合算子数据结构，保存融合算子信息。

用户自定义并实现回调函数，完成原始模型中小算子属性到融合算子的属性映射，将结果填到Operator类中。

```
Status FusionParseParamByOpFunc(const std::vector<ge::Operator> &op_src,  ge::Operator &op_dest)>;
```

以下是FusionParseParamsFn的实现代码示例，目标是从原始模型中的小算子找到融合算子scales属性信息。

 版权所有 © 华为技术有限公司

```
Status ParseFloatFromConstNode(const ge::Operator *node, float &value) {
  if (node == nullptr) {
    return FAILED;
  }
  ge::Tensor tensor;
  auto ret = node->GetAttr("value", tensor);
  if (ret != ge::GRAPH_SUCCESS) {
    AscendString op_name;
    ret = node->GetName(op_name);
    if (ret != ge::GRAPH_SUCCESS) {
      return FAILED;
    }
    OP_LOGE(op_name.GetString(), "Failed to get value from %s", op_name.GetString());
    return FAILED;
  }
  uint8_t *data_addr = tensor.GetData();
  value = *(reinterpret_cast<float *>(data_addr));
  return SUCCESS;
}

// 用户自定义实现回调函数
Status DecodeBboxV2ParseParams(const std::vector<ge::Operator> &inside_nodes, ge::Operator &op_dest) {
  std::map<std::string, std::string> scales_const_name_map;
  std::map<string, const ge::Operator *> node_map;
  for (const auto &node : inside_nodes) {
    ge::AscendString op_type;
    ge::graphStatus ret = node.GetOpType(op_type);
    if (ret != ge::GRAPH_SUCCESS) {
      return FAILED;
    }
    ge::AscendString op_name;
    ret = node.GetName(op_name);
    string str_op_name;
    if (op_name.GetString() != nullptr) {
      str_op_name = op_name.GetString();
    }
    if (op_type == kBoxesDiv) {
      if (node.GetInputsSize() < kRealDivInputSize) {
        OP_LOGE(op_name.GetString(), "Input size of %s is invalid, which is %zu.", kBoxesDiv,
node.GetInputsSize());
        return FAILED;
      }
      ge::AscendString input_unpack_name0;
      ret = node.GetInputDesc(0).GetName(input_unpack_name0);
      string str_input_unpack_name0;
      if (input_unpack_name0.GetString() != nullptr) {
        str_input_unpack_name0 = input_unpack_name0.GetString();
      }
      ge::AscendString input_unpack_name1;
      ret = node.GetInputDesc(1).GetName(input_unpack_name1);
      string str_input_unpack_name1;
      if (input_unpack_name1.GetString() != nullptr) {
        str_input_unpack_name1 = input_unpack_name1.GetString();
      }
      if (str_input_unpack_name0.find(kBoxesUnpack) != string::npos) {
        scales_const_name_map.insert({str_op_name, str_input_unpack_name1 });
      }
    }
    node_map[str_op_name] = &node;
  }

  std::vector<float> scales_list = {1.0, 1.0, 1.0, 1.0};
  if (scales_const_name_map.size() != kScaleSize) {
    ge::AscendString op_name;
    ge::graphStatus ret = op_dest.GetName(op_name);
    if (ret != ge::GRAPH_SUCCESS) {
      return FAILED;
    }
    OP_LOGI(op_name.GetString(), "Boxes doesn't need scale.");
```

 版权所有 © 华为技术有限公司

```
  } else {
    size_t i = 0;
    for (const auto &name_pair : scales_const_name_map) {
      float scale_value = 1.0;
      auto ret = ParseFloatFromConstNode(node_map[name_pair.second], scale_value);
      if (ret != SUCCESS) {
        return ret;
      }
      scales_list[i++] = scale_value;
    }
  }
  op_dest.SetAttr("scales", scales_list);
  return SUCCESS;
}
```

3.6 多对多场景开发注意点

简介

Scope多对多融合场景，即把Scope内的多个小算子融合为一系列小算子组合。

实现流程

实现流程请参考**Scope多对多融合场景**，不同点在于：

1. 在融合结果设置函数GenerateFusionResult中需要设置内部小算子组合的连接关系。
2. 内部小算子后面将不再走插件解析， 因此小算子需用户自行构造成IR表示，并设置好所需属性等参数。

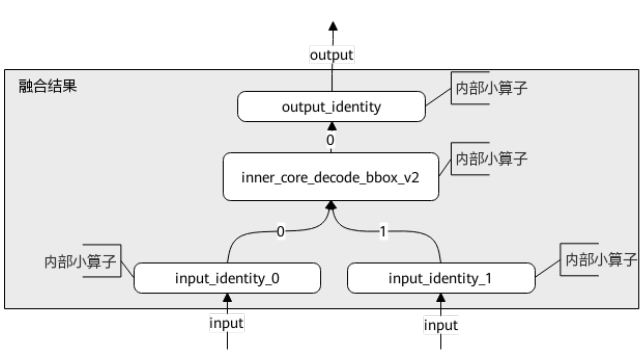
设置融合结果

这里为了逻辑的正确，示例的代码是把DecodeBboxV2算子输入输出前均加上Identity算子，以此来达到添加多个算子的目的，实际操作过程中可根据具体的算子功能需求添加小算子组合。

主要过程包括：

1. 设置融合结果的名字、描述，以及与外部的输入、输出，和多对一场景相同；
2. 设置融合算子类型为特定类型**6.1.3.10.16 kScopeToMultiNodes**，表明融合成多个算子组合；
   ```
   fusion_rlt->SetType(kScopeToMultiNodes); // 设置特定类型，表明融合成多个算子组合
   ```
3. 通过**6.1.3.12 AddInnerNode**设置对应的小算子，并设置名字、类型、属性、输入、输出等相关信息。

图 3-6 添加融合小算子示意图



 版权所有 © 华为技术有限公司

例如：

```
auto in_identity_1 = fusion_rlt->AddInnerNode("input_identity_1", "Identity");
CHECK_INNER_NODE_CONDITION(in_identity_1 != nullptr, fusion_rlt);
ret = in_identity_1->InsertInput(kInputFromFusionScope, 1) // 输入来自融合结果边界的第1个输入
  .InsertOutput("inner_core_decode_bbox_v2", 1)              // 输出给内部算子
  .BuildInnerNode();
CHECK_INNER_NODE_CONDITION(ret == ge::GRAPH_SUCCESS, fusion_rlt);
```

- **6.1.3.10.18 kInputFromFusionScope**：表示目标小算子的输入来自scope边界，另外**6.1.3.10.19 kOutputToFusionScope**表示目标小算子的输出送到scope边界。
- **6.1.3.10.3 InsertInput**：表示设置目标小算子的输入。
- **6.1.3.10.4 InsertOutput**：表示设置目标小算子的输出。
- **6.1.3.10.5 BuildInnerNode**：表示构建内部小算子。

完整代码示例为：

```
void DecodeBboxV2MultiScopeFusionPass::GenerateFusionResult(const std::vector<Scope *> &scopes,
FusionScopesResult *fusion_rlt) {
  if (fusion_rlt == nullptr) {
    return;
  }
  if (scopes.size() != 1) {
    fusion_rlt->SetType(kScopeInvalidType);
    return;
  }

  // 设置融合结果输入，来自transpose的第0个输入作为融合结果的第0个输入，transpose的第一个输入不使用
  fusion_rlt->InsertInputs("transpose", {0, kFusionDisableIndex});
  // 设置融合结果输入，将get_center_coordinates_and_sizes/transpose的第0个输入作为融合结果的第1个输入，get_center_coordinates_and_sizes/transpose的第一个输入不使用
  fusion_rlt->InsertInputs("get_center_coordinates_and_sizes/transpose", {1, kFusionDisableIndex});
  // 设置融合结果输出， 将transpose_1的第0个输出作为融合结果的输出
  fusion_rlt->InsertOutputs("transpose_1", {0});

  fusion_rlt->SetType(kScopeToMultiNodes);  // 设置特定类型，表明融合成多个小算子组合
  AscendString scope_name;
  Status ret = scopes[0]->Name(scope_name);
  if (ret != SUCCESS) {
    return ;
  }
  std::string str_scope_name;
  if (scope_name != nullptr) {
    str_scope_name = scope_name.GetString();
  }
  fusion_rlt->SetName(str_scope_name.substr(0, str_scope_name.length() - 1).c_str());
  fusion_rlt->SetDescription("");

  // 添加融合小算子
  auto in_identity_0 = fusion_rlt->AddInnerNode("input_identity_0", "Identity");
  CHECK_INNER_NODE_CONDITION(in_identity_0 != nullptr, fusion_rlt);
  Status ret = in_identity_0->InsertInput(kInputFromFusionScope, 0)  // 输入来自融合结果边界的第0个输入
    .InsertOutput("inner_core_decode_bbox_v2", 0)                    // 输出给内部小算子
    .BuildInnerNode();
  CHECK_INNER_NODE_CONDITION(ret == ge::GRAPH_SUCCESS, fusion_rlt);
  std::string str_attr = "input_0_identity_attr";
  in_identity_0->MutableOperator()->SetAttr("key", str_attr);

  auto in_identity_1 = fusion_rlt->AddInnerNode("input_identity_1", "Identity");
  CHECK_INNER_NODE_CONDITION(in_identity_1 != nullptr, fusion_rlt);
  ret = in_identity_1->InsertInput(kInputFromFusionScope, 1) // 输入来自融合结果边界的第1个输入
    .InsertOutput("inner_core_decode_bbox_v2", 1)            // 输出给内部小算子
    .BuildInnerNode();
  CHECK_INNER_NODE_CONDITION(ret == ge::GRAPH_SUCCESS, fusion_rlt);

  auto core_decode_bbox = fusion_rlt->AddInnerNode("inner_core_decode_bbox_v2", kScopeType);
```

 版权所有 © 华为技术有限公司

```
  CHECK_INNER_NODE_CONDITION(core_decode_bbox != nullptr, fusion_rlt);
  ret = core_decode_bbox->InsertInput("input_identity_0", 0)
    .InsertInput("input_identity_1", 0)
    .InsertOutput("output_identity", 0)
    .BuildInnerNode();
  CHECK_INNER_NODE_CONDITION(ret == ge::GRAPH_SUCCESS, fusion_rlt);
  // 根据需要设置融合后小算子参数
  auto parser_ret = DecodeBboxV2ParseParams(fusion_rlt->Nodes(), core_decode_bbox-
>MutableOperator());
  CHECK_INNER_NODE_CONDITION(parser_ret == SUCCESS, fusion_rlt);

  auto out_identity = fusion_rlt->AddInnerNode("output_identity", "Identity");
  CHECK_INNER_NODE_CONDITION(out_identity != nullptr, fusion_rlt);
  ret = out_identity->InsertInput("inner_core_decode_bbox_v2", 0) // 输入来自内部小算子的第0个输出
    .InsertOutput(kOutputToFusionScope, 0)                        // 输出给融合结果边界的第0个输出
    .BuildInnerNode();
  CHECK_INNER_NODE_CONDITION(ret == ge::GRAPH_SUCCESS, fusion_rlt);

  ret = fusion_rlt->CheckInnerNodesInfo();
  CHECK_INNER_NODE_CONDITION(ret == ge::GRAPH_SUCCESS, fusion_rlt);

  OP_LOGI(kOpType, "Set fusion multi-to-multi result successfully.");
  return;
}
```

内部小算子所需的信息处理

融合结果内部的小算子可能需要设置某些属性等处理，由于小算子不再走插件去解析，因此需要在构造小算子时添加。示例中的目标小算子DecodeBboxV2需要根据原图scope中小算子获取scale信息。

```
namespace {
Status ParseFloatFromConstNode(const ge::OperatorPtr node, float &value) {
  if (node == nullptr) {
    return FAILED;
  }
  ge::Tensor tensor;
  auto ret = node->GetAttr("value", tensor);
  if (ret != ge::GRAPH_SUCCESS) {
    AscendString op_name;
    graphStatus ret = node->GetName(op_name);
    if (ret != ge::GRAPH_SUCCESS) {
      return FAILED;
    }
    OP_LOGE(kOpType, "Failed to get value from %s", op_name.GetString());
    return FAILED;
  }
  uint8_t *data_addr = tensor.GetData();
  value = *(reinterpret_cast<float *>(data_addr));
  return SUCCESS;
}

Status DecodeBboxV2ParseParams(const std::vector<ge::OperatorPtr> &inside_nodes, ge::Operator *op_dest)
{
  if (op_dest == nullptr) {
    OP_LOGE(kOpType, "Dest operator is nullptr.");
    return FAILED;
  }
  std::map<std::string, std::string> scales_const_name_map;
  std::map<string, ge::OperatorPtr> node_map;
  for (const auto &node : inside_nodes) {
    if (node == nullptr) {
      OP_LOGE(kOpType, "Inner operator is nullptr.");
      return FAILED;
    }
    ge::AscendString op_type;
    ge::graphStatus ret = node.GetOpType(op_type);
    if (ret != ge::GRAPH_SUCCESS) {
      return FAILED;
```

 版权所有 © 华为技术有限公司

```
  }
  ge::AscendString op_name;
  ret = node.GetName(op_name);
  string str_op_name;
  if (op_name.GetString() != nullptr) {
    str_op_name = op_name.GetString();
  }
  if (op_type == kBoxesDiv) {
    if (node->GetInputsSize() < kRealDivInputSize) {
      OP_LOGE(kOpType, "Input size of %s is invalid, which is %zu.", kBoxesDiv, node->GetInputsSize());
      return FAILED;
    }
    ge::AscendString input_unpack_name0;
    ret = node.GetInputDesc(0).GetName(input_unpack_name0);
    string str_input_unpack_name0;
    if (input_unpack_name0.GetString() != nullptr) {
      str_input_unpack_name0 = input_unpack_name0.GetString();
    }
    ge::AscendString input_unpack_name1;
    ret = node.GetInputDesc(1).GetName(input_unpack_name1);
    string str_input_unpack_name1;
    if (input_unpack_name1.GetString() != nullptr) {
      str_input_unpack_name1 = input_unpack_name1.GetString();
    }
    if (str_input_unpack_name0.find(kBoxesUnpack) != string::npos) {
      scales_const_name_map.insert({str_op_name, str_input_unpack_name1 });
    }
  }
  node_map[str_op_name] = &node;
}

std::vector<float> scales_list = {1.0, 1.0, 1.0, 1.0};
if (scales_const_name_map.size() != kScaleSize) {
  OP_LOGI(op_dest.GetName().c_str(), "Boxes doesn't need scale.");
} else {
  size_t i = 0;
  for (const auto &name_pair : scales_const_name_map) {
    float scale_value = 1.0;
    auto ret = ParseFloatFromConstNode(node_map[name_pair.second], scale_value);
    if (ret != SUCCESS) {
      return ret;
    }
    scales_list[i++] = scale_value;
  }
}
op_dest->SetAttr("scales", scales_list);
return SUCCESS;
}
}  // namespace
```

 版权所有 © 华为技术有限公司

4 融合规则生效

在3.4.6 注册融合规则时，定义了融合规则的生效方式：

- 当定义为通用规则时，默认生效，不支持用户指定失效。
- 当定义为定制化规则时，默认不生效，此时，可以参考如下生效该融合规则。

表 4-1 定制化融合规则生效方式

场景	生效方式
离线推理场景下，使用离线模型转换工具编译TensorFlow原始模型	通过模型转换命令行参数**enable_scope_fusion_passes**指定需要生效的融合规则，多个用“,”分隔： **--enable_scope_fusion_passes = DecodeBboxV2ScopeFusionPass**
离线推理场景下，解析TensorFlow原始模型	通过aclgrphParseTensorFlow接口解析TensorFlow原始模型时，通过**ENABLE_SCOPE_FUSION_PASSES**参数指定需要生效的融合规则，多个用“,”分隔： {ge::AscendString(ge::ir_option::**ENABLE_SCOPE_FUSION_PASSES**), ge::AscendString("**DecodeBboxV2ScopeFusionPass**")},
训练或在线推理场景下，在TensorFlow框架内执行	通过TensorFlow框架运行配置参数**enable_scope_fusion_passes**指定需要生效的融合规则，多个用“,”分隔： import tensorflow as tf from npu_bridge.estimator import npu_ops from tensorflow.core.protobuf.rewriter_config_pb2 import RewriterConfig config = tf.ConfigProto() custom_op = config.graph_options.rewrite_options.custom_optimizers.add() custom_op.name = "NpuOptimizer" custom_op.parameter_map["use_off_line"].b = True **custom_op.parameter_map["enable_scope_fusion_passes"].s = tf.compat.as_bytes("DecodeBboxV2ScopeFusionPass")** config.graph_options.rewrite_options.remapping = RewriterConfig.OFF with tf.Session(config=config) as sess: sess.run()

 版权所有 © 华为技术有限公司

5 样例参考

单击**Gitee**或**Github**，进入Ascend samples开源仓，参见README中的“版本说明”下载配套版本的sample包，从“samples/cplusplus/level1_single_api/4_op_dev/1_custom_op/framework/tf_scope_fusion_pass”目录中获取相关样例。融合规则样例的介绍、编译部署、验证方法请参考“samples/cplusplus/level1_single_api/4_op_dev/1_custom_op”下的README。

 版权所有 © 华为技术有限公司

6 接口参考

6.1 Scope 融合规则开发接口

6.1.1 简介

主要介绍Scope融合规则开发接口。

您可以在Ascend-cann-toolkit开发套件包安装目录下的“ascend-toolkit/latest/include/register/scope/scope_fusion_pass_register.h”查看接口定义。

6.1.2 Scope 类

6.1.2.1 Scope 构造函数和析构函数

函数功能

Scope构造函数和析构函数。

函数原型

Scope();

~Scope();

参数说明

参数名	输入/输出	描述
name	输入	Scope名称，按照指定的名称构造Scope

 版权所有 © 华为技术有限公司

参数名	输入/输出	描述
sub_type	输入	Scope子类型名称

返回值

Scope构造函数返回Scope类型的对象。

异常处理

无。

约束说明

无。

6.1.2.2 Init

函数功能

初始化Scope对象。

函数原型

须知

数据类型为string的接口后续版本会废弃，建议使用数据类型为非string的接口。

Status Init(const std::string &name, const std::string &sub_type = "", Scope *father_scope = nullptr);

Status Init(const char *name, const char *sub_type, Scope *father_scope = nullptr);

参数说明

参数名	输入/输出	描述
name	输入	Scope名称
sub_type	输入	Scope子类型名称
father_scope	输入	父Scope指针

返回值

无。

 版权所有 © 华为技术有限公司

约束说明

无。

6.1.2.3 Name

函数功能

获取Scope的名称。

函数原型

须知

数据类型为string的接口后续版本会废弃，建议使用数据类型为非string的接口。

const std::string &Name() const;

Status Name(AscendString &name) const;

参数说明

无。

返回值

参数名	描述
-	Scope名称

约束说明

无。

6.1.2.4 SubType

函数功能

获取Scope子类型名称。

函数原型

须知

数据类型为string的接口后续版本会废弃，建议使用数据类型为非string的接口。

const std::string &SubType() const;

 版权所有 © 华为技术有限公司

Status SubType(AscendString &sub_type) const;

参数说明

无。

返回值

参数名	描述
-	Scope子类型名称

约束说明

无。

6.1.2.5 AllNodesMap

函数功能

获取和Scope相关联的nodes map。

其中包括scope本身关联的node，子scope关联的node，层层递归下去，取得所有相关联的nodes。

函数原型

须知

数据类型为string的接口后续版本会废弃，建议使用数据类型为非string的接口。

const std::unordered_map<std::string, ge::OperatorPtr> &AllNodesMap() const;

Status AllNodesMap(std::unordered_map<AscendString, ge::OperatorPtr> &node_map) const;

参数说明

无。

返回值

参数名	描述
-	Scope的nodes map

 版权所有 © 华为技术有限公司

约束说明

无。

6.1.2.6 GetSubScope

函数功能

获取Scope的子Scope信息。

函数原型

须知

数据类型为string的接口后续版本会废弃，建议使用数据类型为非string的接口。

Scope *GetSubScope(const std::string &scope_name) const;

Scope *GetSubScope(const char *scope_name) const;

参数说明

参数名	输入/输出	描述
scope_name	输入	Scope的名称

返回值

参数名	描述
-	子Scope指针

约束说明

无。

6.1.2.7 LastName

函数功能

获取Scope名称字符串中，最后一个“/”分隔符后的字符串。

 版权所有 © 华为技术有限公司

函数原型

须知

数据类型为string的接口后续版本会废弃，建议使用数据类型为非string的接口。

const std::string LastName() const;

Status LastName(AscendString &name) const;

参数说明

无。

返回值

参数名	描述
-	Scope名称字符串中，最后一个“/”分隔符后的字符串。

约束说明

无。

6.1.2.8 GetAllSubScopes

函数功能

获取Scope关联的子Scope，子Scope关联的子Scope，层层递归下去，取得所有相关联的Scope集合。

函数原型

const std::vector<Scope *> &GetAllSubScopes() const;

参数说明

无。

返回值

参数名	描述
-	所有相关联的Scope集合

 版权所有 © 华为技术有限公司

约束说明

无。

6.1.2.9 GetFatherScope

函数功能

获取Scope的父Scope信息。

函数原型

const Scope *GetFatherScope() const;

参数说明

无。

返回值

参数名	描述
-	父Scope的指针

约束说明

无。

6.1.3 FusionScopesResult 类

6.1.3.1 FusionScopesResult 构造函数和析构函数

函数功能

FusionScopesResult构造函数和析构函数。

函数原型

FusionScopesResult();

~FusionScopesResult();

参数说明

无。

返回值

FusionScopesResult构造函数返回FusionScopesResult类型的对象。

 版权所有 © 华为技术有限公司

异常处理

无。

约束说明

无。

6.1.3.2 Init

函数功能

初始化FusionScopesResult对象。

函数原型

Status Init();

参数说明

无。

返回值

无。

约束说明

无。

6.1.3.3 SetName

函数功能

设置Scope result的名称。

函数原型

须知

数据类型为string的接口后续版本会废弃，建议使用数据类型为非string的接口。

void SetName(const std::string &name);

void SetName(const char *name);

 版权所有 © 华为技术有限公司

参数说明

参数名	输入/输出	描述
name	输入	Scope result 的名称

返回值

无。

约束说明

无。

6.1.3.4 SetType

函数功能

设置Scope result的类型。

函数原型

须知

数据类型为string的接口后续版本会废弃，建议使用数据类型为非string的接口。

void SetType(const std::string &type);

void SetType(const char *type);

参数说明

参数名	输入/输出	描述
type	输入	Scope result的类型名称

返回值

无。

约束说明

无。

 版权所有 © 华为技术有限公司

6.1.3.5 SetDescription

函数功能

设置Scope result的描述信息。

函数原型

须知

数据类型为string的接口后续版本会废弃，建议使用数据类型为非string的接口。

void SetDescription(const std::string &description);

void SetDescription(const char *description);

参数说明

参数名	输入/输出	描述
description	输入	Scope result 的描述

返回值

无。

约束说明

无。

6.1.3.6 Name

函数功能

获取Scope result的名称。

函数原型

须知

数据类型为string的接口后续版本会废弃，建议使用数据类型为非string的接口。

const std::string &Name() const;

const Status Name(AscendString &name) const;

 版权所有 © 华为技术有限公司

参数说明

无。

返回值

参数名	描述
-	Scope reslut 的名称

约束说明

无。

6.1.3.7 Nodes

函数功能

获取Scope result中的nodes信息。

函数原型

const std::vector<ge::OperatorPtr> &Nodes() const;

参数说明

无。

返回值

参数名	描述
-	Scope reslut 中包含的nodes

约束说明

无。

6.1.3.8 InsertInputs

函数功能

在FusionScopesResult中插入融合算子的输入信息。

 版权所有 © 华为技术有限公司

函数原型

须知

数据类型为string的接口后续版本会废弃，建议使用数据类型为非string的接口。

void InsertInputs(const std::string &inner_op_name, const std::vector<int32_t> &index_map);

void InsertInputs(const char *inner_op_name, const std::vector<int32_t> &index_map);

参数说明

参数名	输入/输出	描述
inner_op_name	输入	融合算子名称
index_map	输入	融合算子index信息，是一个vector类型。vector的index表示scope内部小算子的输入index，具体值表示融合算子的输入index，如果融合算子没有使用这个index，则使用占位符kFusionDisableIndex表示。

返回值

无。

约束说明

无。

6.1.3.9 InsertOutputs

函数功能

在FusionScopesResult中插入融合算子的输出信息。

函数原型

须知

数据类型为string的接口后续版本会废弃，建议使用数据类型为非string的接口。

void InsertOutputs(const std::string &inner_op_name, const std::vector<int32_t> &index_map);

void InsertOutputs(const char *inner_op_name, const std::vector<int32_t> &index_map);

 版权所有 © 华为技术有限公司

参数说明

参数名	输入/输出	描述
inner_op_name	输入	融合算子名称
index_map	输入	融合算子index信息

返回值

无。

约束说明

无。

6.1.3.10 InnerNodeInfo 类

InnerNodeInfo类用于提供Scope融合多对多功能，即将Scope融合为多个算子

6.1.3.10.1 SetName

函数功能

设置InnerNodeInfo的名称。

函数原型

须知

数据类型为string的接口后续版本会废弃，建议使用数据类型为非string的接口。

InnerNodeInfo &SetName(const std::string &name);

InnerNodeInfo &SetName(const char *name);

参数说明

参数名	输入/输出	描述
name	输入	InnerNodeInfo的名称

返回值

无。

约束说明

无。

 版权所有 © 华为技术有限公司

6.1.3.10.2 SetType

函数功能

设置InnerNodeInfo的类型。

函数原型

须知

数据类型为string的接口后续版本会废弃，建议使用数据类型为非string的接口。

InnerNodeInfo &SetType(const std::string &type);

InnerNodeInfo &SetType(const char *type);

参数说明

参数名	输入/输出	描述
type	输入	InnerNodeInfo的类型名称

返回值

无。

约束说明

无。

6.1.3.10.3 InsertInput

函数功能

在InnerNodeInfo中插入输入算子相关信息

函数原型

须知

数据类型为string的接口后续版本会废弃，建议使用数据类型为非string的接口。

InnerNodeInfo &InsertInput(const std::string &input_node, int32_t peer_out_idx);

InnerNodeInfo &InsertInput(const char *input_node, int32_t peer_out_idx);

 版权所有 © 华为技术有限公司

参数说明

参数名	输入/输出	描述
input_node	输入	输入节点
peer_out_idx	输入	对应输出的index

返回值

无。

约束说明

无。

6.1.3.10.4 InsertOutput

函数功能

在InnerNodeInfo中插入输出算子相关信息

函数原型

须知

数据类型为string的接口后续版本会废弃，建议使用数据类型为非string的接口。

InnerNodeInfo &InsertOutput(const std::string &output_node, int32_t peer_in_idx);

InnerNodeInfo &InsertOutput(const char *output_node, int32_t peer_in_idx);

参数说明

参数名	输入/输出	描述
output_node	输入	输出节点
peer_in_idx	输入	对应输入的index

返回值

无。

约束说明

无。

 版权所有 © 华为技术有限公司

6.1.3.10.5 BuildInnerNode

函数功能

构建InnerNode

函数原型

ge::graphStatus BuildInnerNode();

参数说明

无。

返回值

无。

约束说明

无。

6.1.3.10.6 SetInputFormat

函数功能

设置InnerNode输入格式

函数原型

须知

数据类型为string的接口后续版本会废弃，建议使用数据类型为非string的接口。

ge::graphStatus SetInputFormat(const std::string &input_name, const std::string &format);

ge::graphStatus SetInputFormat(const char *input_name, const char *format);

参数说明

参数名	输入/输出	描述
input_name	输入	输入算子名
format	输入	算子格式信息

返回值

无。

 版权所有 © 华为技术有限公司

约束说明

无。

6.1.3.10.7 SetOutputFormat

函数功能

设置InnerNode输出算子格式

函数原型

须知

数据类型为string的接口后续版本会废弃，建议使用数据类型为非string的接口。

ge::graphStatus SetOutputFormat(const std::string &output_name, const std::string &format);

ge::graphStatus SetOutputFormat(const char *output_name, const char *format);

参数说明

参数名	输入/输出	描述
output_name	输入	输出算子名称
format	输入	输出算子格式信息

返回值

无。

约束说明

无。

6.1.3.10.8 SetDynamicInputFormat

函数功能

设置DynamicInput格式

函数原型

须知

数据类型为string的接口后续版本会废弃，建议使用数据类型为非string的接口。

 版权所有 © 华为技术有限公司

ge::graphStatus SetDynamicInputFormat(const std::string &input_name, uint32_t index, const std::string &format);

ge::graphStatus SetDynamicInputFormat(const char *input_name, uint32_t index, const char *format);

参数说明

参数名	输入/输出	描述
input_name	输入	InnerNode输入算子名称
index	输入	InnerNode输入算子index信息
format	输入	InnerNode输入算子格式信息

返回值

无。

约束说明

无。

6.1.3.10.9 SetDynamicOutputFormat

函数功能

设置DynamicOutput格式

函数原型

须知

数据类型为string的接口后续版本会废弃，建议使用数据类型为非string的接口。

ge::graphStatus SetDynamicOutputFormat(const std::string &output_name, uint32_t index, const std::string &format);

ge::graphStatus SetDynamicOutputFormat(const char *output_name, uint32_t index, const char *format);

参数说明

参数名	输入/输出	描述
output_name	输入	InnerNode输出算子名称
index	输入	InnerNode输出算子index信息
format	输入	InnerNode输出算子格式信息

 版权所有 © 华为技术有限公司

返回值

无。

约束说明

无。

6.1.3.10.10 MutableOperator

函数功能

定义Operator信息

函数原型

ge::Operator *MutableOperator()

参数说明

无。

返回值

无。

约束说明

无。

6.1.3.10.11 GetName

函数功能

获取InnerNodeInfo名

函数原型

须知

数据类型为string的接口后续版本会废弃，建议使用数据类型为非string的接口。

string GetName() const;

ge::graphStatus GetName(AscendString &name) const;

参数说明

无。

 版权所有 © 华为技术有限公司

返回值

无。

约束说明

无。

6.1.3.10.12 GetType

函数功能

获取InnerNodeInfo类型

函数原型

须知

数据类型为string的接口后续版本会废弃，建议使用数据类型为非string的接口。

string GetType() const;

ge::graphStatus GetType(AscendString &type) const;

参数说明

无。

返回值

无。

约束说明

无。

6.1.3.10.13 GetInputs

函数功能

获取InnerNodeInfo输入信息

函数原型

须知

数据类型为string的接口后续版本会废弃，建议使用数据类型为非string的接口。

std::vector<std::pair<std::string, int32_t>> GetInputs() const;

 版权所有 © 华为技术有限公司

ge::graphStatus GetInputs(std::vector<std::pair<AscendString, int32_t>> &inputs) const;

参数说明

无。

返回值

输入Vector信息。

约束说明

无。

6.1.3.10.14 GetOutputs

函数功能

获取InnerNodeInfo输出信息

函数原型

须知

数据类型为string的接口后续版本会废弃，建议使用数据类型为非string的接口。

std::vector<std::pair<std::string, int32_t>> GetOutputs() const;

ge::graphStatus GetOutputs(std::vector<std::pair<AscendString, int32_t>> &outputs) const;

参数说明

无。

返回值

输出Vector信息。

约束说明

无。

6.1.3.10.15 InnerNodeInfo 构造函数和析构函数

函数功能

InnerNodeInfo构造函数和析构函数

 版权所有 © 华为技术有限公司

函数原型

须知

数据类型为string的接口后续版本会废弃，建议使用数据类型为非string的接口。

explicit InnerNodeInfo(const char *fusion_node_name);

InnerNodeInfo(const char *fusion_node_name, const char *name, const char *type);

explicit InnerNodeInfo(const std::string &fusion_node_name);

InnerNodeInfo(const std::string &fusion_node_name, const std::string &name, const std::string &type);

InnerNodeInfo(InnerNodeInfo &&other) noexcept;

InnerNodeInfo &operator=(InnerNodeInfo &&other) noexcept;

InnerNodeInfo(const InnerNodeInfo &) = delete;

InnerNodeInfo &operator=(const InnerNodeInfo &) = delete;

~InnerNodeInfo();

参数说明

无。

返回值

InnerNodeInfo构造函数返回InnerNodeInfo类型的对象。

约束说明

无。

6.1.3.10.16 kScopeToMultiNodes

常量定义

特殊定义的类型，融合结果中配置了这个类型表示希望将scope映射成多个D算子。

常量定义原型

const char *const kScopeToMultiNodes = "ScopeToMultiNodes";

约束说明

无。

 版权所有 © 华为技术有限公司

6.1.3.10.17 kScopeInvalidType

常量定义

特殊定义的类型，融合结果中配置了这个类型表示有错误发生。

常量定义原型

const char *const kScopeInvalidType = "ScopeInvalidType";

约束说明

无。

6.1.3.10.18 kInputFromFusionScope

常量定义

表示小算子的输入来自scope边界

常量定义原型

const char *const kInputFromFusionScope = "InputFromFusionScope";

约束说明

无。

6.1.3.10.19 kOutputToFusionScope

常量定义

表示小算子输入到scope边界

常量定义原型

const char *const kOutputToFusionScope = "OutputToFusionScope";

约束说明

无。

6.1.3.11 CHECK_INNER_NODE_CONDITION 宏

宏功能

检查FusionScopeResult是否满足条件。

 版权所有 © 华为技术有限公司

6.1.3.12 AddInnerNode

函数功能

增加InnerNode节点。

函数原型

须知

数据类型为string的接口后续版本会废弃，建议使用数据类型为非string的接口。

InnerNodeInfo *AddInnerNode(const string &name, const string &type);

InnerNodeInfo *AddInnerNode(const char *name, const char *type);

参数说明

参数名	输入/输出	描述
name	输入	InnerNode的名称
type	输入	InnerNode的类型

返回值

无。

约束说明

无。

6.1.3.13 MutableRecentInnerNode

函数功能

最近添加的InnerNode节点。

函数原型

InnerNodeInfo *MutableRecentInnerNode();

参数说明

无。

返回值

无。

 版权所有 © 华为技术有限公司

约束说明

无。

6.1.3.14 MutableInnerNode

函数功能

获取指定index的InnerNode，获取之后可以对该InnerNode进行修改。

函数原型

InnerNodeInfo *MutableInnerNode(uint32_t index);

参数说明

参数名	输入/输出	描述
index	输入	InnerNode节点的index

返回值

无。

约束说明

无。

6.1.3.15 CheckInnerNodesInfo

函数功能

检查InnerNodeInfo输入输出。

函数原型

ge::graphStatus CheckInnerNodesInfo();

参数说明

无。

返回值

无。

约束说明

无。

 版权所有 © 华为技术有限公司

6.1.4 ScopeTree 类

6.1.4.1 ScopeTree 构造函数和析构函数

函数功能

ScopeTree构造函数和析构函数。

函数原型

ScopeTree();

~ScopeTree();

参数说明

无。

返回值

ScopeTree构造函数返回ScopeTree类型的对象。

异常处理

无。

约束说明

无。

6.1.4.2 Init

函数功能

初始化ScopeTree对象

函数原型

Status Init();

参数说明

无。

返回值

参数名	描述
-	SUCCESS：初始化成功 其他：初始化失败

 版权所有 © 华为技术有限公司

约束说明

无。

6.1.4.3 GetAllScopes

函数功能

获取ScopeTree中包含的所有Scope信息。

函数原型

const std::vector<Scope *> &GetAllScopes() const;

参数说明

无。

返回值

参数名	描述
-	ScopeTree中包含的所有Scope

约束说明

无。

6.1.5 ScopeGraph 类

6.1.5.1 ScopeGraph 构造函数和析构函数

函数功能

ScopeGraph构造函数和析构函数。

函数原型

ScopeGraph();

~ScopeGraph();

参数说明

无。

 版权所有 © 华为技术有限公司

返回值

ScopeGraph构造函数返回ScopeGraph类型的对象。

异常处理

无。

约束说明

无。

6.1.5.2 Init

函数功能

初始化ScopeGraph对象

函数原型

Status Init();

参数说明

无。

返回值

参数名	描述
-	SUCCESS：初始化成功 其他：初始化失败

约束说明

无。

6.1.5.3 GetScopeTree

函数功能

获取ScopeTree对象实例。

函数原型

const ScopeTree *GetScopeTree() const;

参数说明

无。

 版权所有 © 华为技术有限公司

返回值

参数名	描述
-	ScopeTree对象实例

约束说明

无。

6.1.5.4 GetNodesMap

函数功能

获取ScopeGraph中包含的nodes信息

函数原型

须知

数据类型为string的接口后续版本会废弃，建议使用数据类型为非string的接口。

const std::unordered_map<std::string, ge::OperatorPtr> &GetNodesMap() const;

Status GetNodesMap(std::unordered_map<AscendString, ge::OperatorPtr> &nodes_map) const;

参数说明

无。

返回值

参数名	描述
-	ScopeGraph中nodes信息

约束说明

无。

6.1.6 ScopeAttrValue 类

 版权所有 © 华为技术有限公司

6.1.6.1 ScopeAttrValue 构造函数和析构函数

函数功能

ScopeAttrValue构造函数和析构函数。

函数原型

ScopeAttrValue();

ScopeAttrValue(ScopeAttrValue const &attr_value);

ScopeAttrValue &operator=(ScopeAttrValue const &attr_value);

~ScopeAttrValue();

参数说明

无。

返回值

ScopeAttrValue构造函数返回ScopeAttrValue类型的对象。

异常处理

无。

约束说明

无。

6.1.6.2 SetIntValue

函数功能

设置ScopeAttrValue的整型值。

函数原型

void SetIntValue(int64_t value);

参数说明

参数名	输入/输出	描述
value	输入	整型value值

返回值

无。

 版权所有 © 华为技术有限公司

约束说明

无。

6.1.6.3 SetFloatValue

函数功能

设置ScopeAttrValue的浮点类型值。

函数原型

void SetFloatValue(float value);

参数说明

参数名	输入/输出	描述
value	输入	浮点value值

返回值

无。

约束说明

无。

6.1.6.4 SetStringValue

函数功能

设置ScopeAttrValue的string类型值。

函数原型

须知

数据类型为string的接口后续版本会废弃，建议使用数据类型为非string的接口。

void SetStringValue(std::string value);

void SetStringValue(const char *value);

 版权所有 © 华为技术有限公司

参数说明

参数名	输入/输出	描述
value	输入	string value值

返回值

无。

约束说明

无。

6.1.6.5 SetBoolValue

函数功能

设置ScopeAttrValue的布尔类型值。

函数原型

void SetBoolValue(bool value);

参数说明

参数名	输入/输出	描述
value	输入	布尔value值

返回值

无。

约束说明

无。

6.1.7 ScopeBaseFeature 类

基类。

6.1.7.1 ScopeBaseFeature 析构函数

函数功能

ScopeBaseFeature析构函数。

 版权所有 © 华为技术有限公司

函数原型

virtual ~ScopeBaseFeature();

参数说明

无。

返回值

无。

异常处理

无。

约束说明

无。

6.1.7.2 Match

函数功能

Scope融合匹配接口。

函数原型

virtual bool Match(const Scope *scope) = 0;

参数说明

参数名	输入/输出	描述
scope	输入	用于匹配的Scope对象指针

返回值

参数名	描述
-	true：匹配成功 false：匹配失败

约束说明

用户不直接调用。

6.1.8 NodeOpTypeFeature 类

 版权所有 © 华为技术有限公司

6.1.8.1 NodeOpTypeFeature 构造函数和析构函数

继承ScopeBaseFeature。

函数功能

NodeOpTypeFeature构造函数和析构函数。

函数原型

须知

数据类型为string的接口后续版本会废弃，建议使用数据类型为非string的接口。

NodeOpTypeFeature(std::string nodeType, int num, int step = 0);

NodeOpTypeFeature(const char *node_type, int num, int step = 0);

NodeOpTypeFeature(NodeOpTypeFeature const &feature);

NodeOpTypeFeature &operator=(NodeOpTypeFeature const &feature);

~NodeOpTypeFeature();

参数说明

参数名	输入/输出	描述
nodeType	输入	算子类型
num	输入	算子数量，算子个数为0时，num为-1。
step	输入	步长，图中算子的倍数。

返回值

NodeOpTypeFeature构造函数返回NodeOpTypeFeature类型的对象。

异常处理

无。

约束说明

无。

6.1.8.2 Match

函数功能

Scope融合匹配接口。

 版权所有 © 华为技术有限公司

函数原型

bool Match(const Scope *scope) override;

参数说明

参数名	输入/输出	描述
scope	输入	用于匹配的Scope对象指针

返回值

参数名	描述
-	true：匹配成功 false：匹配失败

约束说明

用户不直接调用。

6.1.9 NodeAttrFeature 类

继承ScopeBaseFeature

6.1.9.1 NodeAttrFeature 构造函数和析构函数

函数功能

NodeAttrFeature构造函数和析构函数。

函数原型

须知

数据类型为string的接口后续版本会废弃，建议使用数据类型为非string的接口。

NodeAttrFeature(std::string nodeType, std::string attr_name, ge::DataType datatype, ScopeAttrValue &attr_value);

NodeOpTypeFeature(const char *node_type, int num, int step = 0);

NodeAttrFeature(NodeAttrFeature const &feature);

NodeAttrFeature &operator=(NodeAttrFeature const &feature);

~NodeAttrFeature();

 版权所有 © 华为技术有限公司

参数说明

参数名	输入/输出	描述
nodeType	输入	算子类型
attr_name	输入	属性名称
datatype	输入	属性值的类型，当前支持DT_FLOAT/DT_INT32/DT_STRING/DT_BOOL。
attr_value	输入	属性值

返回值

NodeAttrFeature构造函数返回NodeAttrFeature类型的对象。

异常处理

无。

约束说明

无。

6.1.9.2 Match

函数功能

Scope融合匹配接口。

函数原型

bool Match(const Scope *scope) override;

参数说明

参数名	输入/输出	描述
scope	输入	用于匹配的Scope对象指针

返回值

参数名	描述
-	true：匹配成功 false：匹配失败

 版权所有 © 华为技术有限公司

约束说明

用户不直接调用。

6.1.10 ScopeFeature 类

继承ScopeBaseFeature。

6.1.10.1 ScopeFeature 构造函数和析构函数

函数功能

ScopeFeature构造函数和析构函数。

函数原型

须知

数据类型为string的接口后续版本会废弃，建议使用数据类型为非string的接口。

ScopeFeature(std::string sub_type, int32_t num, std::string suffix = "", std::string sub_scope_mask = "", int step = 0);

ScopeFeature(const char *sub_type, int32_t num, const char *suffix, const char *sub_scope_mask, int step = 0);

ScopeFeature(ScopeFeature const &feature);

ScopeFeature &operator=(ScopeFeature const &feature);

~ScopeFeature();

参数说明

参数名	输入/输出	描述
sub_type	输入	sub_scope的type类型。
num	输入	scope中所有sub_scope的type类型与sub_type相同的个数。
suffix	输入	scope的后缀名（LastName并去掉index）。 假设scope的名称为：fastrcnn_predictions/strided_slice_1，则sub_scope_mask配置为strided_slice时可以匹配成功。
sub_scope_mask	输入	sub_scope后缀名（LastName并去掉index）的掩码。 假设sub_scope的名称为：fastrcnn_predictions/strided_slice_1，则sub_scope_mask配置为slice时可以匹配成功。
step	输入	步长，用户无需设置，保持默认0。

 版权所有 © 华为技术有限公司

返回值

ScopeFeature构造函数返回ScopeFeature类型的对象。

异常处理

无。

约束说明

无。

6.1.10.2 Match

函数功能

Scope融合匹配接口。

函数原型

bool Match(const Scope *scope) override;

参数说明

参数名	输入/输出	描述
scope	输入	用于匹配的Scope对象指针

返回值

参数名	描述
-	true：匹配成功 false：匹配失败

约束说明

用户不直接调用。

6.1.11 ScopePattern 类

 版权所有 © 华为技术有限公司

6.1.11.1 ScopePattern 构造函数和析构函数

函数功能

ScopePattern构造函数和析构函数。

函数原型

ScopePattern();

~ScopePattern();

参数说明

无。

返回值

ScopePattern构造函数返回ScopePattern类型的对象。

异常处理

无。

约束说明

无。

6.1.11.2 SetSubType

函数功能

Scope匹配到融合规则之后，设置该Scope的类型。

函数原型

须知

数据类型为string的接口后续版本会废弃，建议使用数据类型为非string的接口。

ScopePattern &SetSubType(const std::string &sub_type);

ScopePattern &SetSubType(const char *sub_type);

参数说明

参数名	输入/输出	描述
sub_type	输入	子类型名称

 版权所有 © 华为技术有限公司

返回值

参数名	描述
-	本实例对象的引用

约束说明

无。

6.1.11.3 AddNodeOpTypeFeature

函数功能

添加NodeOpTypeFeature对象到ScopePattern。

函数原型

ScopePattern &AddNodeOpTypeFeature(NodeOpTypeFeature feature);

参数说明

参数名	输入/输出	描述
feature	输入	NodeOpTypeFeature对象

返回值

参数名	描述
-	本实例对象的引用

约束说明

无。

6.1.11.4 AddNodeAttrFeature

函数功能

添加NodeAttrFeature对象到ScopePattern。

函数原型

ScopePattern &AddNodeAttrFeature(NodeAttrFeature feature);

 版权所有 © 华为技术有限公司

参数说明

参数名	输入/输出	描述
feature	输入	NodeAttrFeature对象

返回值

参数名	描述
-	本实例对象的引用

约束说明

无。

6.1.11.5 AddScopeFeature

函数功能

添加ScopeFeature对象到ScopePattern。

函数原型

ScopePattern &AddScopeFeature(ScopeFeature feature);

参数说明

参数名	输入/输出	描述
feature	输入	ScopeFeature对象

返回值

参数名	描述
-	本实例对象的引用

约束说明

无。

6.1.12 ScopesResult 类

 版权所有 © 华为技术有限公司

6.1.12.1 ScopesResult 构造函数和析构函数

函数功能

ScopesResult构造函数和析构函数。

函数原型

ScopesResult();

ScopesResult(ScopesResult const &result);

ScopesResult &operator=(ScopesResult const &result);

~ScopesResult();

参数说明

无。

返回值

ScopesResult构造函数返回ScopesResult类型的对象。

异常处理

无。

约束说明

无。

6.1.12.2 SetScopes

函数功能

设置ScopeResult中包含的scopes。

函数原型

void SetScopes(std::vector<Scope *> &scopes);

参数说明

参数名	输入/输出	描述
scopes	输入	scope vector

返回值

无。

 版权所有 © 华为技术有限公司

约束说明

无。

6.1.12.3 SetNodes

函数功能

设置ScopeResult中包含的nodes。

函数原型

void SetNodes(std::vector<ge::OperatorPtr> &nodes);

参数说明

参数名	输入/输出	描述
nodes	输入	nodes vector

返回值

无。

约束说明

无。

6.1.13 ScopeBasePass 类

6.1.13.1 ScopeBasePass 构造函数和析构函数

函数功能

ScopeBasePass构造函数和析构函数。

函数原型

ScopeBasePass();

virtual ~ScopeBasePass();

参数说明

无。

返回值

无。

 版权所有 © 华为技术有限公司

异常处理

无。

约束说明

抽象类，不能被实例化，但是构造函数会在子类实例化之前被调用。

6.1.13.2 DefinePatterns

函数功能

方法定义，子类实现各自的融合策略并构建Patterns。

函数原型

virtual std::vector<ScopeFusionPatterns> DefinePatterns() = 0;

参数说明

无。

返回值

参数名	描述
-	ScopePattern集合

约束说明

无。

6.1.13.3 PassName

函数功能

方法定义，定义scope融合规则的名称。

函数原型

virtual std::string PassName() = 0;

参数说明

无。

 版权所有 © 华为技术有限公司

返回值

参数名	描述
-	scope融合规则的名称

约束说明

无。

6.1.13.4 LastMatchScopesAndOPs

函数功能

方法定义，子类实现scope匹配及运算符融合方法。

函数原型

virtual Status LastMatchScopesAndOPs(std::shared_ptr<ScopeGraph> &scope_graph, std::vector<ScopesResult> &results) = 0;

参数说明

参数名	输入/输出	描述
scope_graph	输入	指向ScopeGraph对象的shared_ptr指针
results	输出	融合结果集

返回值

参数名	描述
-	SUCCESS：成功 其他：失败

约束说明

无。

6.1.13.5 GenerateFusionResult

函数功能

方法定义，子类实现各自的融合结果，并设置最终融合算子的输入和输出。

 版权所有 © 华为技术有限公司

函数原型

virtual void GenerateFusionResult(const std::vector<Scope *> &scopes, FusionScopesResult *fusion_rlt) = 0;

参数说明

参数名	输入/输出	描述
scopes	输入	ScopeResult中包含的scope集合
fusion_rlt	输入	融合结果

返回值

无。

约束说明

无。

6.1.14 ScopeUtil 类

工具类，提供静态方法调用。

6.1.14.1 StringReplaceAll

函数功能

将字符串str中包含old_value的字符串用new_value替换。

函数原型

须知

数据类型为string的接口后续版本会废弃，建议使用数据类型为非string的接口。

static std::string StringReplaceAll(std::string str, const std::string &old_value, const std::string &new_value);

static AscendString StringReplaceAll(const char *str, const char *old_value, const char *new_value);

参数说明

参数名	输入/输出	描述
str	输入	待处理的源字符串

 版权所有 © 华为技术有限公司

参数名	输入/输出	描述
old_value	输入	待替换的字符串
new_value	输入	用来替换old_value的字符串

返回值

参数名	描述
-	替换后的新字符串

约束说明

无。

6.1.14.2 FreeScopePatterns

函数功能

释放ScopePattern集的资源。

函数原型

static void FreeScopePatterns(ScopeFusionPatterns &patterns);

参数说明

参数名	输入/输出	描述
patterns	输入	待释放的ScopePattern集

返回值

无。

约束说明

无。

6.1.14.3 FreeOneBatchPattern

函数功能

释放一个batch包含的ScopePattern资源。

 版权所有 © 华为技术有限公司

函数原型

static void FreeOneBatchPattern(std::vector<ScopePattern *> &one_batch_pattern);

参数说明

参数名	输入/输出	描述
patterns	输入	一个batch包含的ScopePattern集

返回值

无。

约束说明

无。

6.1.15 ScopeFusionPassRegistry 类

6.1.15.1 ScopeFusionPassRegistry 析构函数

函数功能

ScopeFusionPassRegistry析构函数。

函数原型

~ScopeFusionPassRegistry();

参数说明

无。

返回值

无。

异常处理

无。

约束说明

- 单例类，构造函数不可见。
- 用户不直接调用。

 版权所有 © 华为技术有限公司

6.1.15.2 GetInstance

函数功能

获得ScopeFusionPassRegistry单例对象。

函数原型

static ScopeFusionPassRegistry &GetInstance();

参数说明

无。

返回值

参数名	描述
-	ScopeFusionPassRegistry单例对象的引用

约束说明

用户不直接调用。

6.1.15.3 RegisterScopeFusionPass

函数功能

注册融合算子。

函数原型

须知

数据类型为string的接口后续版本会废弃，建议使用数据类型为非string的接口。

void RegisterScopeFusionPass(const std::string &pass_name, CreateFn create_fn, bool is_general);

void RegisterScopeFusionPass(const char *pass_name, CreateFn create_fn, bool is_general);

参数说明

参数名	输入/输出	描述
pass_name	输入	融合算子名称
create_fn	输入	返回融合算子对象的函数指针

 版权所有 © 华为技术有限公司

参数名	输入/输出	描述
is_general	输入	true：通用规则，默认生效 false：定制化规则，默认不生效

返回值

无。

约束说明

REGISTER_SCOPE_FUSION_PASS内部调用，用户不直接调用。

6.1.16 ScopeFusionPassRegistrar 类

6.1.16.1 ScopeFusionPassRegistrar 构造函数和析构函数

函数功能

ScopeFusionPassRegistrar构造函数和析构函数。

函数原型

ScopeFusionPassRegistrar(const char *pass_name, ScopeBasePass *(*create_fn)(), bool is_general);

~ScopeFusionPassRegistrar();

参数说明

参数名	输入/输出	描述
pass_name	输入	融合算子名称
create_fn	输入	返回融合算子对象的函数指针
is_general	输入	true：通用规则，默认生效 false：定制化规则，默认不生效

返回值

ScopeFusionPassRegistrar构造函数返回ScopeFusionPassRegistrar类型的对象。

异常处理

无。

 版权所有 © 华为技术有限公司

约束说明

REGISTER_SCOPE_FUSION_PASS内部调用，用户不直接调用。

6.1.17 REGISTER_SCOPE_FUSION_PASS

宏功能

注册Scope融合算子。

宏原型

REGISTER_SCOPE_FUSION_PASS(pass_name, scope_pass, is_general)

参数说明

参数	输入/输出	说明
pass_name	输入	融合算子名称
scope_pass	输入	返回融合算子对象的函数指针
is_general	输入	true：通用规则，默认生效 false：定制化规则，默认不生效

约束说明

is_generral 参数只允许传 true 或者 false，传其他值可能会导致规则不能按预期生效。

其他关联宏

以下宏为**REGISTER_SCOPE_FUSION_PASS**关联的宏，用户无需关注。

REGISTER_SCOPE_FUSION_PASS_UNIQ_HELPER

REGISTER_SCOPE_FUSION_PASS_UNIQ

6.2 算子插件适配接口

6.2.1 简介

开发人员完成自定义算子的实现代码后，需要进行适配插件的开发将基于第三方框架的算子映射成适配昇腾AI处理器的算子，可调用REGISTER_CUSTOM_OP宏实现算子转换。在调用REGISTER_CUSTOM_OP宏时，以REGISTER_CUSTOM_OP开始，以“.”链接FrameworkType、OriginOpType、ParseParamsFn等接口。

例如：

```
REGISTER_CUSTOM_OP("OpType")
  .FrameworkType(TENSORFLOW)
  .OriginOpType("OriginOpType")
```

 版权所有 © 华为技术有限公司

```
  .ParseParamsByOperatorFn(ParseParamFunc)
  .ImplyType(ImplyType::TVM);
```

您可以在CANN软件安装后文件存储路径下的“include/register/register.h”文件中查看接口定义。

6.2.2 OpRegistrationData 类

6.2.2.1 总体说明

开发人员完成自定义算子的实现代码后，需要进行适配插件的开发将基于第三方框架的算子映射成适配昇腾AI处理器的算子，可调用REGISTER_CUSTOM_OP宏实现算子转换。在调用REGISTER_CUSTOM_OP宏时，以REGISTER_CUSTOM_OP开始，以“.”链接FrameworkType、OriginOpType、ParseParamsFn等接口。

例如：

```
REGISTER_CUSTOM_OP("OpType")
  .FrameworkType(TENSORFLOW)
  .OriginOpType("OriginOpType")
  .ParseParamsByOperatorFn(ParseParamFunc)
  .ImplyType(ImplyType::TVM);
```

6.2.2.2 OpRegistrationData 构造函数和析构函数

函数功能

OpRegistrationData构造函数和析构函数。

函数原型

须知

数据类型为string的接口后续版本会废弃，建议使用数据类型为非string的接口。

OpRegistrationData(const std::string& om_optype);

OpRegistrationData(const char *om_optype);

~OpRegistrationData()

参数说明

参数名	输入/输出	描述
om_optype	输入	指定适配昇腾AI处理器的模型支持的算子类型。

返回值

OpRegistrationData构造函数返回OpRegistrationData类型的对象。

 版权所有 © 华为技术有限公司

异常处理

无。

约束说明

无。

6.2.2.3 REGISTER_CUSTOM_OP 宏

宏功能

按指定名称注册算子。

宏原型

REGISTER_CUSTOM_OP(name)

参数说明

参数	输入/输出	说明
name	输入	算子类型名称

6.2.2.4 FrameworkType

函数功能

设置原始模型的框架类型。

函数原型

OpRegistrationData &FrameworkType(const domi::FrameworkType& fmk_type)

 版权所有 © 华为技术有限公司

参数说明

参数	输入/输出	说明

fmk_type	输入	框架类型
		• CAFFE
		• TENSORFLOW
		• ONNX
		FrameworkType 枚举值的定义如下：
		enum FrameworkType {
		CAFFE = 0,
		MINDSPORE = 1,
		TENSORFLOW = 3,
		ANDROID_NN,
		ONNX,
		FRAMEWORK_RESERVED,
		};

6.2.2.5 OriginOpType

函数功能

设置原始模型的算子类型或算子类型列表。

函数原型

须知

数据类型为string的接口后续版本会废弃，建议使用数据类型为非string的接口。

OpRegistrationData &OriginOpType(const std::vector<ge::AscendString> &ori_op_type_list);

OpRegistrationData &OriginOpType(const char *ori_op_type);

OpRegistrationData &OriginOpType(const std::initializer_list<std::string>& ori_optype_list);

OpRegistrationData &OriginOpType(const std::string& ori_optype)

参数说明

参数	输入/输出	说明
ori_optype_list	输入	原始模型算子类型列表

 版权所有 © 华为技术有限公司

参数	输入/输出	说明
ori_optype	输入	原始模型算子类型

6.2.2.6 ParseParamsFn

函数功能

注册解析算子属性的函数。

函数原型

OpRegistrationData &ParseParamsFn(const ParseParamFunc& parseParamFn)

参数说明

参数	输入/输出	说明
parseParamFn	输入	解析算子属性的函数ParseParamFunc，请参见**回调函数ParseParamFunc**。 针对TensorFlow框架，若原始TensorFlow框架算子属性与适配昇腾AI处理器的模型中算子属性一一对应（属性个数与顺序一致），可直接使用**AutoMappingFn**函数自动实现映射。

约束说明

对于自定义算子插件，ParseParamsFn后续版本将会废弃，请使用**6.2.2.7 ParseParamsByOperatorFn**接口进行算子属性的解析。

若用户已使用ParseParamsFn接口进行了算子插件的开发，请执行如下操作进行新接口适配：

1. 请重新使用**6.2.2.7 ParseParamsByOperatorFn**接口进行算子插件的开发。
2. 请基于新版本自定义算子样例工程的编译脚本重新进行自定义算子工程的编译。

回调函数 ParseParamFunc

用户自定义并实现FusionParseParamFunc类函数，完成原始模型中算子属性到适配昇腾AI处理器的模型中算子属性映射，将结果填到Operator类中。

Status ParseParamFunc(const Message* op_origin, ge::Operator& op_dest)

表 6-1 参数说明

参数	输入/输出	说明
op_origin	输入	protobuf格式的数据结构（来源于原始模型的prototxt文件），包含算子属性信息。

 版权所有 © 华为技术有限公司

参数	输入/输出	说明
op_dest	输出	适配昇腾AI处理器的模型的算子数据结构，保存算子信息。 关于Operator类，请参见6.3.3 Operator类。

6.2.2.7 ParseParamsByOperatorFn

函数功能

注册解析用户自定义算子属性的函数

函数原型

OpRegistrationData &ParseParamsByOperatorFn(const ParseParamByOpFunc &parse_param_by_op_fn)

参数说明

参数	输入/输出	说明
parse_param_by_op_fn	输入	解析用户自定义算子属性的函数，请参见回调函数 ParseParamByOpFunc。

回调函数 ParseParamByOpFunc

用户自定义并实现ParseParamByOpFunc类函数，完成原始模型中算子属性到适配昇腾AI处理器的模型中属性的映射，将结果填到Operator类中。

Status ParseParamByOpFunc(const ge::Operator & op_origin, ge::Operator& op_dest)

表 6-2 参数说明

参数	输入/输出	说明
op_origin	输入	框架定义的Operator类对象，包含解析出的原始模型中自定义算子属性信息，关于Operator类，请参见6.3.3 Operator类。
op_dest	输出	适配昇腾AI处理器的模型中的算子数据结构，保存算子信息。 关于Operator类，请参见6.3.3 Operator类。

约束说明

无。

 版权所有 © 华为技术有限公司

6.2.2.8 FusionParseParamsFn

函数功能

注册解析融合算子属性的函数。

函数原型

OpRegistrationData &FusionParseParamsFn(const FusionParseParamFunc &fusionParseParamFn)

参数说明

参数	输入/输出	说明
fusionParseParamFn	输入	解析融合算子属性的函数，请参见**回调函数FusionParseParamFunc**。

约束说明

对于融合算子插件，FusionParseParamsFn接口后续版本将会废弃，请使用**6.2.2.9 FusionParseParamsFn（Overload）**接口进行融合算子属性的解析。

回调函数 FusionParseParamFunc

用户自定义并实现FusionParseParamFunc类函数，完成原始模型中属性到适配昇腾AI处理器的模型中属性的映射，将结果填到Operator类中。

Status FusionParseParamFunc(const vector<const google::protobuf::Message *> &v_op_origin, ge::Operator &op_dest)

表 6-3 参数说明

参数	输入/输出	说明
v_op_origin	输入	一组scope内的protobuf格式的数据结构（来源于原始模型的prototxt文件），包含算子属性信息。
op_dest	输出	融合算子数据结构，保存融合算子信息。 关于Operator类，请参见**6.3.3 Operator类**。

6.2.2.9 FusionParseParamsFn（Overload）

函数功能

注册解析融合算子属性的函数，为**6.2.2.8 FusionParseParamsFn**的重载函数。

 版权所有 © 华为技术有限公司

函数原型

OpRegistrationData &FusionParseParamsFn(const FusionParseParamByOpFunc &fusion_parse_param_fn)

参数说明

参数	输入/输出	说明
fusion_parse_param_fn	输入	解析融合算子属性的函数，请参见**回调函数 FusionParseParamByOpFunc**。

回调函数 FusionParseParamByOpFunc

用户自定义并实现FusionParseParamByOpFunc类函数，完成原始模型中属性到适配昇腾AI处理器的模型中的属性映射，将结果填到Operator类中。

Status FusionParseParamByOpFunc(const std::vector<ge::Operator> &op_src, ge::Operator &op_dest)>;

表 6-4 参数说明

参数	输入/输出	说明
op_src	输入	一组scope内存储原始模型中算子属性的融合算子数据结构， 关于Operator类，请参见**6.3.3 Operator类**。
op_dest	输出	融合算子数据结构，保存融合算子信息。 关于Operator类，请参见**6.3.3 Operator类**。

调用示例

```
REGISTER_CUSTOM_OP(XXXXXX)
.FrameworkType(TENSORFLOW)
.FusionParseParamsFn(FusionParseParamsFn)
.OriginOpType(XXXXX)
.ImplyType(XXXXX);
}
```

6.2.2.10 ParseSubgraphPostFn

函数功能

根据算子类型，注册算子的子图中输入输出节点跟算子的输入输出的对应关系函数实现。

 版权所有 © 华为技术有限公司

函数原型

须知

数据类型为string的接口后续版本会废弃，建议使用数据类型为非string的接口。

OpRegistrationData &OpRegistrationData::ParseSubgraphPostFn(const ParseSubgraphFunc &subgraph_post_fn)

OpRegistrationData &ParseSubgraphPostFn(const ParseSubgraphFuncV2 &subgraph_post_fn);

参数说明

参数	输入/输出	说明
subgraph_post_fn	输入	子图中输入输出节点跟算子的输入输出的对应关系函数对象。 请详见回调函数**ParseSubgraphFuncV2**

约束说明

无。

回调函数 ParseSubgraphFuncV2

用户自定义并实现ParseSubgraphFuncV2函数，完成解析子图中输入输出节点跟算子的输入输出的对应关系功能，回调函数原型定义如下：

Status ParseSubgraphFuncV2(const ge::AscendString &subgraph_name, const ge::Graph &graph)

表 6-5 参数说明

参数	输入/输出	说明
subgraph_name	输入	子图名字。
graph	输出	构造的子图。

6.2.2.11 ParseOpToGraphFn

函数功能

注册实现算子一对多子图映射的函数，即将算子映射为多个算子。

 版权所有 © 华为技术有限公司

函数原型

OpRegistrationData &ParseOpToGraphFn(const ParseOpToGraphFunc &parse_op_to_graph_fn)

参数说明

参数	输入/输出	说明
parse_op_to_graph_fn	输入	实现算子一对多映射，进行子图构造的函数。 请参见**回调函数 ParseOpToGraphFunc** 。

约束说明

实现一对多子图映射时，插件注册时首先需要将原始框架中的算子映射成昇腾AI处理器中的PartitionedCall算子，并在**ParseParamsByOperatorFn**函数中使用“SetAttr”接口设置original_type。

实现样例请参见**调用示例**。

回调函数 ParseOpToGraphFunc

用户自定义并实现ParseOpToGraphFunc函数，通过IR模型构建方式完成一对多子图的构造，构图详细介绍请参考《**Ascend Graph开发指南**》。

回调函数原型定义如下：

Status ParseOpToGraphFunc(const ge::Operator &op, ge::Graph &graph)

表 6-6 参数说明

参数	输入/输出	说明
op	输入	PartitionedCall算子数据结构，Operator类对象。
graph	输出	构造的子图。

子图输入输出关系构建方式如下：

- 输入：通过添加Data节点标识，Data节点的index属性表示原节点的第index个输入边。
- 输出：通过Graph::SetOutputs()接口设置，该接口的入参为**std::vector<std::pair<Operator, std::vector<size_t>>>**，输出边按照设置的输出顺序相连。

调用示例

以将Add算子转换成Addn+Abs为例。

 版权所有 © 华为技术有限公司

实现Add算子到PartitionedCall算子的映射函数示例如下所示：

```
Status ParseParams(const ge::Operator &op_src, ge::Operator& op_dest)
{
    ...
    op_dest.SetAttr("original_type", "ai.onnx::11::Add");
}
```

一对多子图构造函数实现示例如下所示：

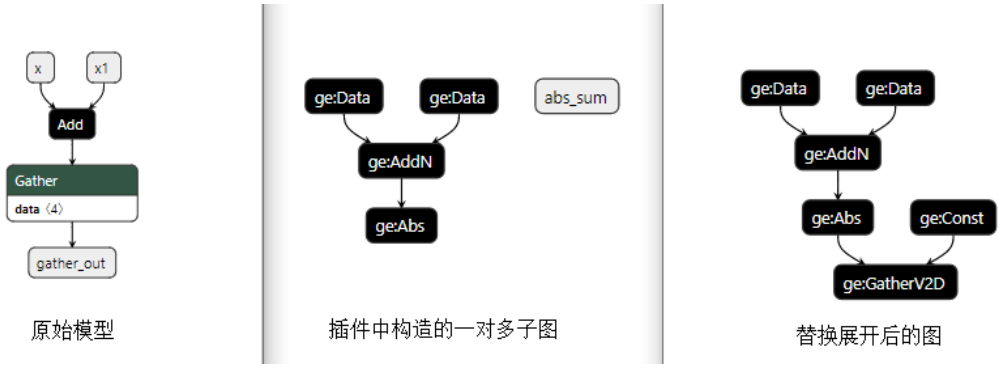
```
static Status ParseOpToGraph(const Operator &op, Graph &graph) {
  auto data_0 = op::Data().set_attr_index(0);
  auto data_1 = op::Data().set_attr_index(1);
  auto addn = op::AddN("addn_sum").create_dynamic_input_x(2)
      .set_dynamic_input_x(0, data_0)
      .set_dynamic_input_x(1, data_1)
      .set_attr_N(2);
  auto abs = op::Abs("abs_sum").set_input_x(addn);
  std::vector<Operator> inputs{data_0, data_1};
  std::vector<std::pair<Operator, std::vector<size_t>>> output_indexs;
  output_indexs.emplace_back(abs, vector<std::size_t>{0});
  graph.SetInputs(inputs).SetOutputs(output_indexs);
  return domi::SUCCESS;
}
```

进行注册：

```
REGISTER_CUSTOM_OP("PartitionedCall")
.FrameworkType(xx)
.OriginOpType(xx)
.ParseParamsByOperatorFn(ParseParams)
.ParseOpToGraphFn(ParseOpToGraph)
.ImplyType(ImplyType::TVM);
}
```

图6-1为将Add算子进行一对多子图映射后的示例。

图 6-1 一对多转换示意图



6.2.2.12 ImplyType

函数功能

设置算子执行方式。

函数原型

OpRegistrationData& ImplyType(const domi::ImplyType& imply_type)

参数说明

参数	输入/输出	说明

imply_type	输入	算子执行方式。
		enum class ImplyType : unsigned int
		{
		BUILDIN = 0,// 内置算子，由OME正常执行
		TVM, // 编译成tvm bin文件执行
		CUSTOM, // 由用户自定义计算逻辑，通过CPU执行
		AI_CPU, // AI CPU 自定义算子类型
		INVALID = 0xFFFFFFFF,
		};

6.2.2.13 DelInputWithCond

函数功能

根据算子属性，删除算子指定输入边。

函数原型

须知

数据类型为string的接口后续版本会废弃，建议使用数据类型为非string的接口。

OpRegistrationData& DelInputWithCond(int inputIdx, const std::string& attrName, bool attrValue)

OpRegistrationData &DelInputWithCond(int input_idx, const char *attr_name, bool attr_value);

参数说明

参数	输入/输出	说明
inputIdx	输入	需要删除的输入边编号。
attrName	输入	属性名字。
attrValue	输入	属性的值。

约束说明

无。

 版权所有 © 华为技术有限公司

6.2.2.14 DelInputWithOriginalType

函数功能

根据算子类型，删除算子指定输入边。

函数原型

须知

数据类型为string的接口后续版本会废弃，建议使用数据类型为非string的接口。

OpRegistrationData &DelInputWithOriginalType(int input_idx, const std::string &ori_type)

OpRegistrationData &DelInputWithOriginalType(int input_idx, const char *ori_type);

参数说明

参数	输入/输出	说明
input_idx	输入	需要删除的输入边编号。
ori_type	输入	删除节点的原始算子类型。

约束说明

无。

6.2.2.15 GetImplyType

函数功能

获取算子执行方式。

函数原型

ImplyType GetImplyType () const

参数说明

无。

约束说明

无。

 版权所有 © 华为技术有限公司

6.2.2.16 GetOmOptype

函数功能

获取模型的算子类型。

函数原型

须知

数据类型为string的接口后续版本会废弃，建议使用数据类型为非string的接口。

string GetOmOptype () const

Status GetOmOptype(ge::AscendString &om_op_type) const;

参数说明

无。

约束说明

无。

6.2.2.17 GetOriginOpTypeSet

函数功能

获取原始模型的算子类型集合。

函数原型

须知

数据类型为string的接口后续版本会废弃，建议使用数据类型为非string的接口。

set<std::string> GetOriginOpTypeSet () const

Status GetOriginOpTypeSet(std::set<ge::AscendString> &ori_op_type) const;

参数说明

无。

约束说明

无。

 版权所有 © 华为技术有限公司

6.2.2.18 GetFrameworkType

函数功能

获取原始模型的框架类型。

函数原型

FrameworkType GetFrameworkType() const

参数说明

无。

约束说明

无。

6.2.2.19 GetParseParamFn

函数功能

获取解析算子属性的函数。

函数原型

ParseParamFunc GetParseParamFn() const

参数说明

无。

约束说明

GetParseParamFn接口后续版本将会废弃，请使用**6.2.2.7 ParseParamsByOperatorFn**接口获取算子属性。

6.2.2.20 GetParseParamByOperatorFn

函数功能

获取解析算子属性的函数。

函数原型

ParseParamByOpFunc GetParseParamByOperatorFn() const

参数说明

无。

 版权所有 © 华为技术有限公司

约束说明

无。

6.2.2.21 GetFusionParseParamFn

函数功能

获取解析融合算子属性的函数。

函数原型

FusionParseParamFunc GetFusionParseParamFn() const

参数说明

无。

约束说明

GetFusionParseParamFn接口后续版本将会废弃，请使用**6.2.2.22 GetFusionParseParamByOpFn**接口获取融合算子的属性。

6.2.2.22 GetFusionParseParamByOpFn

函数功能

获取解析融合算子属性的函数。

函数原型

FusionParseParamByOpFunc GetFusionParseParamByOpFn() const;

参数说明

无。

约束说明

无。

调用示例

无

6.2.2.23 GetParseSubgraphPostFn

函数功能

根据算子类型，获取算子注册的子图中输入输出节点跟算子的输入输出的对应关系实现的函数对象。

 版权所有 © 华为技术有限公司

函数原型

须知

GetParseSubgraphPostFn()函数后续版本将废弃，建议使用GetParseSubgraphPostFn(ParseSubgraphFuncV2 &func)函数。

- **ParseSubgraphFunc GetParseSubgraphPostFn() const**

  该函数会返回ParseSubgraphFunc类型的函数对象，ParseSubgraphFunc函数的声明如下：

  ```
  using ParseSubgraphFunc = std::function<Status(const std::string &subgraph_name, const ge::Graph &graph)>
  ```

- **Status GetParseSubgraphPostFn(ParseSubgraphFuncV2 &func) const**

  该函数会返回ParseSubgraphFuncV2类型的函数对象，ParseSubgraphFuncV2函数的声明如下：

  ```
  using ParseSubgraphFuncV2 = std::function<Status(const ge::AscendString &subgraph_name, const ge::Graph &graph)>
  ```

参数说明

- GetParseSubgraphPostFn()函数

  无。

- GetParseSubgraphPostFn(ParseSubgraphFuncV2 &func)函数

参数	输入/输出	说明
ParseSubgraphFuncV2	输出	实现算子注册的子图中输入输出节点跟算子的输入输出对应关系的函数对象。

约束说明

无。

6.2.2.24 GetParseOpToGraphFn

函数功能

获取将算子映射为一对多子图的实现函数。

函数原型

ParseOpToGraphFunc GetParseOpToGraphFn() const

参数说明

无。

 版权所有 © 华为技术有限公司

约束说明

无。

6.2.2.25 AutoMappingFn

函数功能

自动映射回调函数。

函数原型

Status AutoMappingFn(const google::protobuf::Message* op_src, ge::Operator& op)

参数说明

参数	输入/输出	说明
op_src	输入	转换前原始模型中的算子，包含原始模型中算子的属性。
op	输入	适配昇腾AI处理器的算子。 关于Operator类，请参见6.3.3 Operator类。

约束说明

若原始TensorFlow算子与适配昇腾AI处理器的算子属性无法一一映射，AutoMappingFn函数无法应用于回调函数ParseParamsByOperatorFn中，此种场景下，请在回调函数ParseParamsByOperatorFn中使用6.2.2.26 AutoMappingByOpFn接口进行可以映射成功的属性的自动解析，使用示例请参见调用示例。

6.2.2.26 AutoMappingByOpFn

函数功能

自动映射回调函数。

函数原型

Status AutoMappingByOpFn(const ge::Operator &op_src, ge::Operator &op);

参数说明

参数	输入/输出	说明
op_src	输入	转换前原始模型中的算子，包含原始模型中算子的属性。

 版权所有 © 华为技术有限公司

参数	输入/输出	说明
op	输入	适配昇腾AI处理器的算子。

关于Operator类，请参见**6.3.3 Operator类**

调用示例

原始TensorFlow算子与适配昇腾AI处理器的算子属性一一映射的场景：

```
REGISTER_CUSTOM_OP("SoftplusGrad")
.FrameworkType(TENSORFLOW)
.OriginOpType("SoftplusGrad")
.ParseParamsByOperatorFn(AutoMappingByOpFn)
.ImplyType(ImplyType::TVM);
}
```

原始TensorFlow算子与适配昇腾AI处理器的算子属性无法一一映射的场景：

```
Status ParseResizeArea(const ge::Operator &op_src, ge::Operator& op)
 {
   AutoMappingByOpFn(op_src, op);

   ge::TensorDesc input_tensor = op.GetInputDesc("images");
   input_tensor.SetOriginFormat(ge::FORMAT_NHWC);
   input_tensor.SetFormat(ge::FORMAT_NHWC);
   auto ret = op.UpdateInputDesc("images", input_tensor);
   if(ret != ge::GRAPH_SUCCESS){
       return FAILED;
   }
   ge::TensorDesc output_tensor = op.GetOutputDesc("y");
   output_tensor.SetOriginFormat(ge::FORMAT_NHWC);
   output_tensor.SetFormat(ge::FORMAT_NHWC);
   auto ret_output = op.UpdateOutputDesc("y", output_tensor);
   if(ret_output != ge::GRAPH_SUCCESS){
       return FAILED;
   }
   return SUCCESS;
 }
// register ResizeArea op to GE
REGISTER_CUSTOM_OP("ResizeArea")
  .FrameworkType(TENSORFLOW)
  .OriginOpType("ResizeArea")
  .ParseParamsByOperatorFn(ParseResizeArea)
  .ImplyType(ImplyType::AI_CPU);
}  // namespace domi
```

6.2.2.27 AutoMappingFnDynamic

函数功能

动态输入/输出算子的自动映射回调函数。

函数原型

Status AutoMappingFnDynamic(const google::protobuf::Message *op_src, ge::Operator &op, std::map<std::string, std::pair<std::string, std::string>> dynamic_name_attr_value, int in_pos = -1, int out_pos = -1)

 版权所有 © 华为技术有限公司

参数说明

参数	输入/输出	说明
op_src	输入	转换前原始模型中的算子，包含原始模型中算子的属性。
op	输入	适配昇腾AI处理器的算子。
dynamic_name_attr_value	输入	描述动态输入输出实际个数，key表示动态端口是输入还是输出，key的取值： • in：代表算子的输入。 • out：代表算子的输出。
in_pos	输入	动态输入的端口id。
out_pos	输入	动态输出的端口id。

约束说明

若原始TensorFlow算子与适配昇腾AI处理器的算子属性无法一一映射，AutoMappingFnDynamic函数无法应用于回调函数**ParseParamsByOperatorFn**中，此种场景下，请在回调函数**ParseParamsByOperatorFn**中使用**6.2.2.28 AutoMappingByOpFnDynamic**接口进行可以映射成功的属性的自动解析，使用示例请参见**调用示例**。

代码示例

动态输入的代码示例：

```
// register MapStage op to GE
Status MapStageMapping(const google::protobuf::Message* op_src, ge::Operator& op) {
  map<string, pair<string, string>> value;
  value["in"] = pair<string, string>("values", "fake_dtypes");
  AutoMappingFnDynamic(op_src, op, value);
  return SUCCESS;
}

REGISTER_CUSTOM_OP("MapStage")
    .FrameworkType(TENSORFLOW)
    .OriginOpType("MapStage")
    .ParseParamsFn(MapStageMapping)
    .ImplyType(ImplyType::AI_CPU);
```

动态输出的代码示例：

```
Status AutoMappingFnSplit(const google::protobuf::Message* op_src, ge::Operator& op) {
  map<string, pair<string, string>> value;
  value["out"] = pair<string, string>("y", "num_split");
  AutoMappingFnDynamic(op_src, op, value);
  return SUCCESS;
}

REGISTER_CUSTOM_OP("Split")
    .FrameworkType(TENSORFLOW)
    .OriginOpType("Split")
    .ParseParamsFn(AutoMappingFnSplit)
    .ImplyType(ImplyType::TVM);
}  // namespace domi
```

 版权所有 © 华为技术有限公司

6.2.2.28 AutoMappingByOpFnDynamic

函数功能

动态输入/输出算子的自动映射回调函数。

函数原型

Status AutoMappingByOpFnDynamic(const ge::Operator &op_src,
ge::Operator &op,

const vector<DynamicInputOutputInfo> &dynamic_name_attr_value);

参数说明

参数	输入/输出	说明
op_src	输入	转换前原始模型中的算子，包含原始模型中算子的属性。 关于Operator类，请参见**6.3.3 Operator类**
op	输入	适配昇腾AI处理器的算子。 关于Operator类，请参见**6.3.3 Operator类**
dynamic_name_attr_value	输入	描述动态输入输出实际个数，**DynamicInputOutputInfo**数据结构请参见**DynamicInputOutputInfo数据结构说明**。

DynamicInputOutputInfo 数据结构说明

```
const int64_t kMaxNameLength = 1048576; // 1M
enum DynamicType {
kInvalid = 0,
kInput = 1,
kOutput = 2
};

struct DynamicInputOutputInfo {
DynamicType type;
const char *port_name;
int64_t port_name_len;
const char *attr_name;
int64_t attr_name_len;
DynamicInputOutputInfo() :
type(kInvalid), port_name(nullptr), port_name_len(0),
attr_name(nullptr), attr_name_len(0) {}
DynamicInputOutputInfo(DynamicType type, const char *port_name, int64_t port_name_len,
const char *attr_name, int64_t attr_name_len) :
type(type),
port_name(port_name),
port_name_len(port_name_len),
attr_name(attr_name),
attr_name_len(attr_name_len) {}
};
```

 版权所有 © 华为技术有限公司

参数	说明
type	指定是动态输入或输出。 0：无效值 1：输入 2：输出
port_name	端口名字，输入或者输出的Name。
port_name_len	端口名字长度，最大长度为**kMaxNameLength**
attr_name	属性名字。
attr_name_len	属性名字长度，最大长度为**kMaxNameLength**

调用示例

```
Status QueueDequeueUpToMapping(const  ge::Operator& op_src, ge::Operator& op) {
vector<DynamicInputOutputInfo> dynamic_name_attr_value;
string port_name = "components";
string attr_name = "component_types";
DynamicInputOutputInfo name_attr(kOutput, port_name.c_str(), port_name.size(), attr_name.c_str(),
attr_name.size());
dynamic_name_attr_value.push_back(name_attr);
AutoMappingByOpFnDynamic(op_src, op, dynamic_name_attr_value);
return SUCCESS;
}

REGISTER_CUSTOM_OP("QueueDequeueUpTo")
.FrameworkType(TENSORFLOW)
.OriginOpType("QueueDequeueUpToV2")
.ParseParamsByOperatorFn(QueueDequeueUpToMapping)
.ImplyType(ImplyType::AI_CPU);
```

6.2.2.29 AutoMappingSubgraphIndex

函数功能

设置子图的输入输出和主图对应父节点输入输出的对应关系。

函数原型

Status AutoMappingSubgraphIndex(const ge::Graph &graph,

const std::function<int(int data_index)> &input,

const std::function<int(int netoutput_index)> &output)

Status AutoMappingSubgraphIndex(const ge::Graph &graph,

const std::function<Status(int data_index, int &parent_input_index)> &input,

const std::function<Status(int netoutput_index, int &parent_output_index)> &output)

 版权所有 © 华为技术有限公司

参数说明

参数	输入/输出	说明
graph	输入	子图对象
input	输入	输入对应关系函数
output	输入	输出对应关系函数

约束说明

无。

6.2.2.30 InputReorderVector

函数功能

支持在算子插件中调整算子的输入参数顺序，此接口为内部使用接口，外部开发者无需关注。

函数原型

OpRegistrationData &InputReorderVector(const vector<int> &input_order)

参数说明

参数	输入/输出	说明
input_order	输入	算子输入的调整列表， 下标表示原输入索引，下标对应的值表示调整后新的输入索引。例如：第三方框架的算子A对应的昇腾AI处理器算子为AD，原输入0为in0， 原输入1为in1，原输入2为in2，插件调用接口传入input_order = {1，0，2}，那么解析后算子AD的输入0为in1， 输入1为in0，输入2为in2。

返回值

OpRegistrationData类的引用。

异常处理

无。

约束说明

无。

 版权所有 © 华为技术有限公司

6.2.3 OpReceiver 类

6.2.3.1 OpReceiver 构造函数和析构函数

函数功能

OpReceiver构造函数，接收自定义算子的注册信息。

函数原型

OpReceiver(OpRegistrationData& reg_data);

~OpReceiver()

参数说明

参数	输入/输出	说明
reg_data	输入	需要注册的算子信息。

返回值

OpReceiver构造函数返回OpReceiver类型的对象。

异常处理

无。

约束说明

无。

6.2.4 DECLARE_ERRORNO

错误码及描述注册宏，该宏对外提供如下四个错误码供用户使用：

- SUCCESS：成功。
- FAILED：失败。
- PARAM_INVALID：参数不合法。
- SCOPE_NOT_CHANGED：Scope融合规则未匹配到，忽略当前pass。

声明如下所示：

```
DECLARE_ERRORNO(0, 0, SUCCESS, 0);
DECLARE_ERRORNO(0xFF, 0xFF, FAILED, 0xFFFFFFFF);
DECLARE_ERRORNO_COMMON(PARAM_INVALID, 1);  // 50331649
DECLARE_ERRORNO(SYSID_FWK, 1, SCOPE_NOT_CHANGED, 201);
```

您可以在CANN软件安装后文件存储路径安装目录“include/register/register_error_codes.h”下查看错误码定义。

 版权所有 © 华为技术有限公司

6.3 Operator 接口

6.3.1 简介

本文档主要描述Operator相关接口，您可以在CANN软件安装后文件存储路径下的“include/graph”路径下查看对应接口的头文件。

接口分类	头文件
AscendString类	ascend_string.h
Operator类	operator.h
Tensor类	tensor.h
TensorDesc类	tensor.h
Shape类	tensor.h
AttrValue类	attr_value.h
Memblock类	ge_allocator.h
Allocator类	ge_allocator.h
数据类型和枚举值	types.h

6.3.2 AscendString 类

6.3.2.1 AscendString 构造函数和析构函数

函数功能

AscendString构造函数和析构函数。

函数原型

AscendString() = default;

~AscendString() = default;

explicit AscendString(const char* name);

参数说明

参数名	输入/输出	描述
name	输入	字符串名称。

 版权所有 © 华为技术有限公司

返回值

AscendString构造函数返回AscendString类型的对象。

异常处理

无。

约束说明

无。

6.3.2.2 GetString

函数功能

获取字符串地址。

函数原型

const char* GetString() const

约束说明

无

参数说明

无

返回值

参数名	类型	描述
-	char	字符串地址。

6.3.2.3 关系符重载

对于AscendString对象大小比较的使用场景（例如map数据结构的key进行排序），通过重载以下关系符实现。

```
bool operator<(const AscendString& d) const;
bool operator>(const AscendString& d) const;
bool operator<=(const AscendString& d) const;
bool operator>=(const AscendString& d) const;
bool operator==(const AscendString& d) const;
bool operator!=(const AscendString& d) const;
```

6.3.3 Operator 类

 版权所有 © 华为技术有限公司

6.3.3.1 Operator 构造函数和析构函数

函数功能

Operator构造函数和析构函数。

函数原型

须知

数据类型为string的接口后续版本会废弃，建议使用数据类型为非string的接口。

Operator();

explicit Operator(const string& type)

explicit Operator(const char *type);

explicit Operator(const string& name, const string& type)

Operator(const ge::AscendString &name, const ge::AscendString &type);

Operator(const char *name, const char *type);

virtual ~Operator()

参数说明

参数名	输入/输出	描述
type	输入	算子类型。
name	输入	算子名称。

返回值

Operator构造函数返回Operator类型的对象。

异常处理

无。

约束说明

无。

6.3.3.2 AddControlInput

函数功能

添加算子的控制边，控制边目前只是控制算子的执行顺序。

 版权所有 © 华为技术有限公司

函数原型

Operator& AddControlInput(const Operator& src_oprt)

参数说明

参数名	输入/输出	描述
src_oprt	输入	控制边对应的源算子。

返回值

参数名	类型	描述
-	Operator&	算子对象本身。

异常处理

无。

约束说明

无。

6.3.3.3 BreakConnect

函数功能

删除当前算子与前一个算子之间的所有连接关系，删除当前算子与下一个算子之间的所有连接关系。

函数原型

void BreakConnect() const

参数说明

无。

返回值

无。

异常处理

无。

 版权所有 © 华为技术有限公司

约束说明

无。

6.3.3.4 IsEmpty

函数功能

判断operator对象是否为空，空表示不可用。

函数原型

bool IsEmpty() const

参数说明

无。

返回值

参数名	类型	描述
-	const	• True：非空。 • False：为空。

异常处理

无。

约束说明

无。

6.3.3.5 InferShapeAndType

函数功能

推导Operator输出的shape和DataType。

关于DataType数据类型的定义，请参见**6.3.10.2 DataType**。

函数原型

graphStatus InferShapeAndType()

参数说明

无。

 版权所有 © 华为技术有限公司

返回值

参数名	类型	描述
-	graphStatus	推导成功，返回GRAPH_SUCCESS，否则，返回GRAPH_FAILED。

异常处理

无。

约束说明

无。

6.3.3.6 GetAttr

函数功能

根据属性名称获取对应的属性值。

函数原型

须知

数据类型为string的接口后续版本会废弃，建议使用数据类型为非string的接口。

graphStatus GetAttr(const string& name, int64_t& attr_value) const;

graphStatus GetAttr(const char *name, int64_t &attr_value) const;

graphStatus GetAttr(const string& name, int32_t& attr_value) const;

graphStatus GetAttr(const char *name, int32_t &attr_value) const;

graphStatus GetAttr(const string& name, uint32_t& attr_value) const;

graphStatus GetAttr(const char *name, uint32_t &attr_value) const;

graphStatus GetAttr(const string& name, std::vector<int64_t>& attr_value) const;

graphStatus GetAttr(const char *name, std::vector<int64_t> &attr_value) const;

graphStatus GetAttr(const string& name, std::vector<int32_t>& attr_value) const;

graphStatus GetAttr(const char *name, std::vector<int32_t> &attr_value) const;

 版权所有 © 华为技术有限公司

graphStatus GetAttr(const string& name, std::vector<uint32_t>& attr_value) const;

graphStatus GetAttr(const char *name, std::vector<uint32_t> &attr_value) const;

graphStatus GetAttr(const string& name, float& attr_value) const;

graphStatus GetAttr(const char *name, float &attr_value) const;

graphStatus GetAttr(const string& name, std::vector<float>& attr_value) const;

graphStatus GetAttr(const char *name, std::vector<float> &attr_value) const;

graphStatus GetAttr(const string& name, AttrValue& attr_value) const;

graphStatus GetAttr(const char *name, AttrValue &attr_value) const;

graphStatus GetAttr(const string& name, string& attr_value) const;

graphStatus GetAttr(const char *name, AscendString &attr_value) const;

graphStatus GetAttr(const string& name, std::vector<string>& attr_value) const;

graphStatus GetAttr(const char *name, std::vector<AscendString> &attr_values) const;

graphStatus GetAttr(const string& name, bool& attr_value) const;

graphStatus GetAttr(const char *name, bool &attr_value) const;

graphStatus GetAttr(const string& name, std::vector<bool>& attr_value) const;

graphStatus GetAttr(const char *name, std::vector<bool> &attr_value) const;

graphStatus GetAttr(const string& name, Tensor& attr_value) const;

graphStatus GetAttr(const char *name, Tensor &attr_value) const;

graphStatus GetAttr(const string& name, std::vector<Tensor>& attr_value) const;

graphStatus GetAttr(const char *name, std::vector<Tensor>& attr_value) const;

graphStatus GetAttr(const string& name, OpBytes& attr_value) const;

graphStatus GetAttr(const char *name, OpBytes &attr_value) const;

graphStatus GetAttr(const string& name, std::vector<std::vector<int64_t>>& attr_value) const;

graphStatus GetAttr(const char *name, std::vector<std::vector<int64_t>> &attr_value) const;

graphStatus GetAttr(const string& name, std::vector<ge::DataType>& attr_value) const;

graphStatus GetAttr(const char *name, std::vector<ge::DataType> &attr_value) const;

 版权所有 © 华为技术有限公司

graphStatus GetAttr(const string& name, ge::DataType& attr_value) const;

graphStatus GetAttr(const char *name, ge::DataType &attr_value) const;

graphStatus GetAttr(const string& name, std::vector<ge::NamedAttrs>& attr_value) const;

graphStatus GetAttr(const char *name, std::vector<ge::NamedAttrs> &attr_value) const;

graphStatus GetAttr(const string& name, ge::NamedAttrs& attr_value) const;

graphStatus GetAttr(const char *name, ge::NamedAttrs &attr_value) const;

参数说明

参数名	输入/输出	描述
name	输入	属性名称。
attr_value	输出	返回的int64_t表示的整型类型属性值。
attr_value	输出	返回的int32_t表示的整型类型属性值。
attr_value	输出	返回的uint32_t表示的整型类型属性值。
attr_value	输出	返回的vector<int64_t>表示的整型列表类型属性值。
attr_value	输出	返回的vector<int32_t>表示的整型列表类型属性值。
attr_value	输出	返回的vector<uint32_t>表示的整型列表类型属性值。
attr_value	输出	返回的浮点类型的属性值。
attr_value	输出	返回的浮点列表类型的属性值。
attr_value	输出	返回的AttrValue类型的属性值。
attr_value	输出	返回的布尔类型的属性值。
attr_value	输出	返回的布尔列表类型的属性值。
attr_value	输出	返回的字符串类型的属性值。
attr_value	输出	返回的字符串列表类型的属性值。
attr_value	输出	返回的Tensor类型的属性值。
attr_value	输出	返回的Tensor列表类型的属性值。
attr_value	输出	返回的Bytes，即字节数组类型的属性值，OpBytes即vector<uint8_t>。
attr_value	输出	返回的量化数据的属性值。

 版权所有 © 华为技术有限公司

参数名	输入/输出	描述
attr_value	输出	返回的vector<vector<int64_t>>表示的整型二维列表类型属性值。
attr_value	输出	返回的vector<ge::DataType>表示的DataType列表类型属性值。
attr_value	输出	返回的DataType类型的属性值。
attr_value	输出	返回的vector<ge::NamedAttrs>表示的NamedAttrs列表类型属性值。
attr_value	输出	返回的NamedAttrs类型的属性值。

返回值

参数名	类型	描述
-	graphStatus	找到对应name，返回GRAPH_SUCCESS，否则返回GRAPH_FAILED。

异常处理

无。

约束说明

无。

6.3.3.7 GetAllAttrNamesAndTypes

函数功能

获取该算子所有的属性名称和属性类型。

函数原型

须知

数据类型为string的接口后续版本会废弃，建议使用数据类型为非string的接口。

const std::map<std::string, std::string> GetAllAttrNamesAndTypes() const

graphStatus GetAllAttrNamesAndTypes(std::map<ge::AscendString, ge::AscendString> &attr_name_types) const

 版权所有 © 华为技术有限公司

参数说明

参数名	输入/输出	描述
attr_name_types	输出	所有的属性名称和属性类型。

返回值

参数名	类型	描述（参数说明、取值范围等）
-	graphStatus	GRAPH_FAILED：失败。 GRAPH_SUCCESS：成功。

异常处理

无。

约束说明

无。

6.3.3.8 GetDynamicInputNum

函数功能

获取算子的动态Input的实际个数。

函数原型

须知

数据类型为string的接口后续版本会废弃，建议使用数据类型为非string的接口。

int GetDynamicInputNum(const string& name) const

int GetDynamicInputNum(const char *name) const

参数说明

参数名	输入/输出	描述
name	输入	算子的动态Input名。

 版权所有 © 华为技术有限公司

返回值

参数名	类型	描述
-	int	实际动态Input的个数。 当name非法，或者算子无动态Input时，返回-1。

约束说明

无。

6.3.3.9 GetDynamicInputDesc

函数功能

根据name和index的组合获取算子动态Input的TensorDesc。

函数原型

须知

数据类型为string的接口后续版本会废弃，建议使用数据类型为非string的接口。

TensorDesc GetDynamicInputDesc(const string& name,uint32_t index) const

TensorDesc GetDynamicInputDesc(const char *name, uint32_t index) const

参数说明

参数名	输入/输出	描述
name	输入	算子动态Input的名称。
index	输入	算子动态Input编号，编号从1开始。

 版权所有 © 华为技术有限公司

返回值

参数名	类型	描述
-	TensorDesc	获取TensorDesc成功，则返回算子动态Input的TensorDesc；获取失败，则返回TensorDesc默认构造的对象，其中，主要设置DataType为DT_FLOAT（表示float类型），Format为FORMAT_NCHW（表示NCHW）。

异常处理

无。

约束说明

无。

6.3.3.10 GetDynamicOutputNum

函数功能

获取算子的动态Output的实际个数。

函数原型

须知

数据类型为string的接口后续版本会废弃，建议使用数据类型为非string的接口。

int GetDynamicOutputNum(const string& name) const

int GetDynamicOutputNum(const char *name) const

参数说明

参数名	输入/输出	描述
name	输入	算子的动态Output名。

 版权所有 © 华为技术有限公司

返回值

参数名	类型	描述
-	int	实际动态Output的个数。 当name非法，或者算子无动态Output时，返回0。

约束说明

无。

6.3.3.11 GetDynamicOutputDesc

函数功能

根据name和index的组合获取算子动态Output的TensorDesc。

函数原型

须知

数据类型为string的接口后续版本会废弃，建议使用数据类型为非string的接口。

TensorDesc GetDynamicOutputDesc(const char *name, uint32_t index) const

TensorDesc GetDynamicOutputDesc(const string& name, uint32_t index) const

参数说明

参数名	输入/输出	描述
name	输入	算子动态Output的名称。
index	输入	算子动态Output编号，编号从1开始。

 版权所有 © 华为技术有限公司

返回值

参数名	类型	描述
-	TensorDesc	获取TensorDesc成功，则返回算子动态Output的TensorDesc；获取失败，则返回TensorDesc默认构造的对象，其中，主要设置DataType为DT_FLOAT（表示float类型），Format为FORMAT_NCHW（表示NCHW）。

异常处理

无。

约束说明

无。

6.3.3.12 GetDynamicSubgraph

函数功能

根据子图名称和子图索引获取算子对应的动态输入子图。

函数原型

须知

数据类型为string的接口后续版本会废弃，建议使用数据类型为非string的接口。

Graph GetDynamicSubgraph(const string &name, uint32_t index) const

Graph GetDynamicSubgraph(const char *name, uint32_t index) const

参数说明

参数名	输入/输出	描述
name	输入	子图名。
index	输入	同名子图的索引。

返回值

Graph对象。

 版权所有 © 华为技术有限公司

异常处理

无。

约束说明

无。

6.3.3.13 GetDynamicSubgraphBuilder

函数功能

根据子图名称和子图索引获取算子对应的动态输入子图的构建函数对象。

函数原型

须知

数据类型为string的接口后续版本会废弃，建议使用数据类型为非string的接口。

SubgraphBuilder GetDynamicSubgraphBuilder(const string &name, uint32_t index) const

SubgraphBuilder GetDynamicSubgraphBuilder(const char *name, uint32_t index) const

参数说明

参数	输入/输出	描述
name	输入	子图名。
index	输出	同名子图的索引。

返回值

SubgraphBuilder对象。

异常处理

无。

约束说明

无。

 版权所有 © 华为技术有限公司

6.3.3.14 GetInferenceContext

函数功能

获取当前算子传递infershape推导所需要的关联信息，比如前面算子的shape和DataType信息。

函数原型

InferenceContextPtr GetInferenceContext() const

参数说明

无。

返回值

参数名	类型	描述
-	InferenceContextPtr	返回当前operator的推理上下文。 InferenceContextPtr是指向InferenceContext类的指针的别名： `using InferenceContextPtr = std::shared_ptr<InferenceContext>;`

异常处理

无。

约束说明

无。

6.3.3.15 GetInputConstData

函数功能

如果指定算子Input对应的节点为Const节点，可调用该接口获取Const节点的数据。

函数原型

须知

数据类型为string的接口后续版本会废弃，建议使用数据类型为非string的接口。

graphStatus GetInputConstData(const string& dst_name, Tensor& data) const

graphStatus GetInputConstData(const char *dst_name, Tensor &data) const

 版权所有 © 华为技术有限公司

参数说明

参数名	输入/输出	描述
dst_name	输入	输入名称。
data	输出	返回Const节点的数据Tensor。

返回值

参数名	类型	描述
-	graphStatus	如果指定算子Input对应的节点为Const节点且获取数据成功，返回GRAPH_SUCCESS，否则，返回GRAPH_FAILED。

异常处理

无。

约束说明

无。

6.3.3.16 GetInputsSize

函数功能

获取当前算子Input个数。

函数原型

size_t GetInputsSize() const

参数说明

无。

返回值

参数名	类型	描述
-	size_t	返回当前算子Input个数。

异常处理

无。

 版权所有 © 华为技术有限公司

约束说明

无。

6.3.3.17 GetInputDesc

函数功能

根据算子Input名称或Input索引获取算子Input的TensorDesc。

函数原型

须知

数据类型为string的接口后续版本会废弃，建议使用数据类型为非string的接口。

TensorDesc GetInputDesc(const string& name) const;

TensorDesc GetInputDescByName(const char *name) const;

TensorDesc GetInputDesc(uint32_t index) const

参数说明

参数名	输入/输出	描述
name	输入	算子Input名称。 当无此算子Input名称时，则返回TensorDesc默认构造的对象，其中，主要设置6.3.10.2 DataType为DT_FLOAT（表示float类型），6.3.10.1 Format为FORMAT_NCHW（表示NCHW）。
index	输入	算子Input索引。 当无此算子Input索引时，则返回TensorDesc默认构造的对象，其中，主要设置6.3.10.2 DataType为DT_FLOAT（表示float类型），6.3.10.1 FormatFORMAT_NCHW（表示NCHW）。

返回值

参数名	类型	描述
-	TensorDesc	算子Input的TensorDesc。

 版权所有 © 华为技术有限公司

异常处理

无。

约束说明

无。

6.3.3.18 GetName

函数功能

获取算子名称。

函数原型

须知

数据类型为string的接口后续版本会废弃，建议使用数据类型为非string的接口。

string GetName() const

graphStatus GetName(ge::AscendString &name) const

参数说明

参数名	输入/输出	描述
name	输出	算子名称。

返回值

参数名	类型	描述
-	graphStatus	GRAPH_FAILED：失败。 GRAPH_SUCCESS：成功。

异常处理

无。

约束说明

无。

 版权所有 © 华为技术有限公司

6.3.3.19 GetSubgraph

函数功能

根据子图名称获取算子对应的子图。

函数原型

须知

数据类型为string的接口后续版本会废弃，建议使用数据类型为非string的接口。

Graph GetSubgraph(const string &name) const

Graph GetSubgraph(const char *name) const

参数说明

参数名	输入/输出	描述
name	输入	子图名称。

返回值

Graph对象。

异常处理

无。

约束说明

无。

6.3.3.20 GetSubgraphBuilder

函数功能

根据子图名称获取算子对应的子图构建的函数对象。

函数原型

须知

数据类型为string的接口后续版本会废弃，建议使用数据类型为非string的接口。

SubgraphBuilder GetSubgraphBuilder(const string &name) const

 版权所有 © 华为技术有限公司

SubgraphBuilder GetSubgraphBuilder(const char *name) const

参数说明

参数名	输入/输出	描述
name	输入	子图名称。

返回值

SubgraphBuilder对象。

异常处理

无。

约束说明

无。

6.3.3.21 GetSubgraphNamesCount

函数功能

获取一个算子的子图个数。

函数原型

size_t Operator::GetSubgraphNamesCount() const

参数说明

无。

返回值

参数名	类型	描述
-	size_t	返回当前算子子图个数。

异常处理

无。

约束说明

无。

 版权所有 © 华为技术有限公司

6.3.3.22 GetSubgraphNames

函数功能

获取一个算子的子图名称列表。

函数原型

须知

数据类型为string的接口后续版本会废弃，建议使用数据类型为非string的接口。

std::vector<std::string> GetSubgraphNames() const

graphStatus GetSubgraphNames(std::vector<ge::AscendString> &names) const

参数说明

参数名	输入/输出	描述
names	输出	获取一个算子的子图名称列表。

返回值

参数名	类型	描述
-	graphStatus	GRAPH_FAILED: 失败。 GRAPH_SUCCESS: 成功。

异常处理

无。

约束说明

无。

6.3.3.23 GetOpType

函数功能

获取算子类型。

 版权所有 © 华为技术有限公司

函数原型

须知

数据类型为string的接口后续版本会废弃，建议使用数据类型为非string的接口。

string GetOpType() const

graphStatus GetOpType(ge::AscendString &type) const

参数说明

参数名	输入/输出	描述
type	输出	算子类型。

返回值

参数名	类型	描述
-	graphStatus	GRAPH_FAILED：失败。 GRAPH_SUCCESS：成功。

异常处理

无。

约束说明

无。

6.3.3.24 GetOutputDesc

函数功能

根据算子Output名称或Output索引获取算子Output的TensorDesc。

函数原型

须知

数据类型为string的接口后续版本会废弃，建议使用数据类型为非string的接口。

TensorDesc GetOutputDesc(const string& name) const

TensorDesc GetOutputDescByName(const char *name) const

 版权所有 © 华为技术有限公司

TensorDesc GetOutputDesc(uint32_t index) const

参数说明

参数名	输入/输出	描述
name	输入	算子Output名称。 当无此算子Output名称时，返回TensorDesc默认构造的对象，其中，主要设置6.3.10.2 DataType为DT_FLOAT（表示float类型），6.3.10.1 Format为FORMAT_NCHW（表示NCHW）。
index	输入	算子Output索引。 当无此算子Output索引时，则返回TensorDesc默认构造的对象，其中，主要设置6.3.10.2 DataType为DT_FLOAT（表示float类型），6.3.10.1 Format为FORMAT_NCHW（表示NCHW）。

返回值

参数名	类型	描述
-	TensorDesc	算子Output的TensorDesc。

异常处理

无。

约束说明

无。

6.3.3.25 GetOutputsSize

函数功能

获取算子所有Output的个数。

函数原型

size_t GetOutputsSize() const

参数说明

无。

 版权所有 © 华为技术有限公司

返回值

参数名	类型	描述
-	size_t	返回当前算子的Output个数。

约束说明

无。

6.3.3.26 SetAttr

函数功能

设置算子属性的属性值。

算子可以包括多个属性，初次设置值后，算子属性值的类型固定，算子属性值的类型包括：

- 整型：接受int64_t、uint32_t、int32_t类型的整型值

  使用SetAttr(const string& name, int64_t attrValue)设置属性值，以GetAttr(const string& name, int32_t& attrValue) 、GetAttr(const string& name, uint32_t& attrValue) 取值时，用户需保证整型数据没有截断，同理针对int32_t和uint32_t混用时需要保证不被截断。
- 整型列表：接受std::vector<int64_t>、std::vector<int32_t>、std::vector<uint32_t>、std::initializer_list<int64_t>&&表示的整型列表数据
- 浮点数：float
- 浮点数列表：std::vector<float>
- 字符串：string
- 字符串列表：std::vector<string>
- 布尔：bool
- 布尔列表：std::vector<bool>
- Tensor：Tensor
- Tensor列表：std::vector<Tensor>
- Bytes：字节数组，SetAttr接受通过OpBytes（即vector<uint8_t>），和（const uint8_t* data, size_t size）表示的字节数组
- 量化数据：UsrQuantizeFactorParams
- AttrValue类型
- 整型二维列表类型：std::vector<std::vector<int64_t>
- DataType列表类型：vector<ge::DataType>
- DataType类型：DataType
- NamedAttrs类型： NamedAttrs
- NamedAttrs列表类型： vector<<NamedAttrs>>

 版权所有 © 华为技术有限公司

函数原型

须知

数据类型为string的接口后续版本会废弃，建议使用数据类型为非string的接口。

Operator &SetAttr(const string& name, int64_t attr_value);

Operator &SetAttr(const char *name, int64_t attr_value);

Operator &SetAttr(const string& name, int32_t attr_value);

Operator &SetAttr(const char *name, int32_t attr_value);

Operator &SetAttr(const string& name, uint32_t attr_value);

Operator &SetAttr(const char *name, uint32_t attr_value);

Operator &SetAttr(const string& name, const std::vector<int64_t>& attr_value);

Operator &SetAttr(const char *name, const std::vector<int64_t> &attr_value);

Operator &SetAttr(const string& name, const std::vector<int32_t>& attr_value);

Operator &SetAttr(const char *name, const std::vector<int32_t> &attr_value);

Operator &SetAttr(const string& name, const std::vector<uint32_t>& attr_value);

Operator &SetAttr(const char *name, const std::vector<uint32_t> &attr_value);

Operator &SetAttr(const string& name, std::initializer_list<int64_t>&& attr_value);

Operator &SetAttr(const char *name, std::initializer_list<int64_t> &&attr_value);

Operator &SetAttr(const string& name, float attr_value);

Operator &SetAttr(const char *name, float attr_value);

Operator &SetAttr(const string& name, const std::vector<float>& attr_value);

Operator &SetAttr(const char *name, const std::vector<float> &attr_value);

Operator &SetAttr(const string& name, AttrValue&& attr_value);

Operator &SetAttr(const char *name, AttrValue &&attr_value);

Operator &SetAttr(const string& name, const string& attr_value);

Operator &SetAttr(const char *name, const char *attr_value);

Operator &SetAttr(const char *name, const AscendString &attr_value);

Operator &SetAttr(const string& name, const std::vector<string>& attr_value);

 版权所有 © 华为技术有限公司

Operator &SetAttr(const char *name, const std::vector<AscendString> &attr_values);

Operator &SetAttr(const string& name, bool attr_value);

Operator &SetAttr(const char *name, bool attr_value);

Operator &SetAttr(const string& name, const std::vector<bool>& attr_value);

Operator &SetAttr(const char *name, const std::vector<bool> &attr_value);

Operator &SetAttr(const string& name, const Tensor& attr_value);

Operator &SetAttr(const char *name, const Tensor &attr_value);

Operator &SetAttr(const string& name, const std::vector<Tensor>& attr_value);

Operator &SetAttr(const char *name, const std::vector<Tensor> &attr_value);

Operator &SetAttr(const string& name, const OpBytes& attr_value);

Operator &SetAttr(const char *name, const OpBytes &attr_value);

Operator &SetAttr(const string& name, const std::vector<std::vector<int64_t>>& attr_value);

Operator &SetAttr(const char *name, const std::vector<std::vector<int64_t>> &attr_value);

Operator &SetAttr(const string& name, const std::vector<ge::DataType>& attr_value);

Operator &SetAttr(const char *name, const std::vector<ge::DataType> &attr_value);

Operator &SetAttr(const string& name, const ge::DataType& attr_value);

Operator &SetAttr(const char *name, const ge::DataType &attr_value);

Operator &SetAttr(const string& name, const ge::NamedAttrs &attr_value);

Operator &SetAttr(const char *name, const ge::NamedAttrs &attr_value);

Operator &SetAttr(const string& name, const std::vector<ge::NamedAttrs> &attr_value);

Operator &SetAttr(const char *name, const std::vector<ge::NamedAttrs> &attr_value);

参数说明

参数名	输入/输出	描述
name	输入	属性名称。
attr_value	输入	需设置的int64_t表示的整型类型属性值。
attr_value	输入	需设置的int32_t表示的整型类型属性值。
attr_value	输入	需设置的uint32_t表示的整型类型属性值。

 版权所有 © 华为技术有限公司

参数名↵	输入/输出↵	描述↵
attr_value	输入	需设置的vector<int64_t>表示的整型列表类型属性值。
attr_value	输入	需设置的vector<int32_t>表示的整型列表类型属性值。
attr_value	输入	需设置的vector<uint32_t>表示的整型列表类型属性值。
attr_value	输入	需设置的std::initializer_list<int64_t>&&表示的整型列表类型属性值。
attr_value	输入	需设置的浮点类型的属性值。
attr_value	输入	需设置的浮点列表类型的属性值。
attr_value	输入	需设置的布尔类型的属性值。
attr_value	输入	需设置的布尔列表类型的属性值。
attr_value	输入	需设置的AttrValue类型的属性值。
attr_value	输入	需设置的字符串类型的属性值。
attr_value	输入	需设置的字符串列表类型的属性值。
attr_value	输入	需设置的Tensor类型的属性值。
attr_value	输入	需设置的Tensor列表类型的属性值。
attr_value	输入	需设置的Bytes，即字节数组类型的属性值，OpBytes即vector<uint8_t>。
data	输入	需设置的Bytes，即字节数组类型的属性值，指定了字节流的首地址。
size	输入	需设置的Bytes，即字节数组类型的属性值，指定了字节流的长度。
attr_value	输入	需设置的量化数据的属性值。
attr_value	输入	需设置的vector<vector<int64_t>>表示的整型二维列表类型属性值。
attr_value	输入	需设置的vector<ge::DataType>表示的DataType列表类型属性值。
attr_value	输入	需设置的DataType类型的属性值。
attr_value	输入	需设置的NamedAttrs类型的属性值。
attr_value	输入	需设置的vector<ge::NamedAttrs>表示的NamedAttrs列表类型的属性值。

 版权所有 © 华为技术有限公司

返回值

参数名	类型	描述
-	Operator&	对象本身。

异常处理

无。

约束说明

无。

6.3.3.27 SetInput

函数功能

设置算子Input，即由哪个算子的输出连到本算子。

有如下几种SetInput方法：

如果指定srcOprt第0个Output为当前算子Input，使用第一个函数原型设置当前算子Input，不需要指定srcOprt的Output名称。

如果指定srcOprt的其它Output为当前算子Input，使用第二个函数原型设置当前算子Input，需要指定srcOprt的Output名称。

如果指定srcOprt的其它Output为当前算子Input，使用第三个函数原型设置当前算子Input，需要指定srcOprt的第index个Output。

函数原型

须知

数据类型为string的接口后续版本会废弃，建议使用数据类型为非string的接口。

Operator& SetInput(const string& dst_name, const Operator& src_oprt);

Operator &SetInput(const char *dst_name, const Operator &src_oprt);

Operator& SetInput(const string& dst_name, const Operator& src_oprt, const string &name)

Operator &SetInput(const char *dst_name, const Operator &src_oprt, const char *name);

Operator& SetInput(const string& dst_name, const Operator& src_oprt, uint32_t index)

Operator &SetInput(const char *dst_name, const Operator &src_oprt, uint32_t index);

 版权所有 © 华为技术有限公司

参数说明

参数名	输入/输出	描述
dst_name	输入	当前算子Input名称。
src_oprt	输入	Input名称为dstName的输入算子对象。
name	输入	srcOprt的Output名称。
index	输入	srcOprt的第index个Output。

返回值

参数名	类型	描述
-	Operator&	当前调度者本身。

异常处理

无。

约束说明

无。

6.3.3.28 SetInferenceContext

函数功能

向当前算子传递infershape推导所需要的关联信息，比如前面算子的shape和DataType信息。

函数原型

void SetInferenceContext(const InferenceContextPtr &inference_context)

参数说明

参数名	输入/输出	描述
inference_context	输入	当前operator的推理上下文。 InferenceContextPtr是指向InferenceContext类的指针的别名： `using InferenceContextPtr = std::shared_ptr<InferenceContext>;`

 版权所有 © 华为技术有限公司

返回值

无。

异常处理

无。

约束说明

无。

6.3.3.29 SetInputAttr

函数功能

设置算子输入Tensor属性的属性值。

算子可以包括多个属性，初次设置值后，算子属性值的类型固定，算子属性值的类型包括：

- 整型：接受int64_t、uint32_t、int32_t类型的整型值
  以int64_t为例，使用:
  SetInputAttr(const char_t *dst_name, const char_t *name, int64_t attr_value);
  SetInputAttr(const int32_t index, const char_t *name, int64_t attr_value);
  设置属性值，以:
  GetInputAttr(const int32_t index, const char_t *name, int64_t &attr_value) const;
  GetInputAttr(const char_t *dst_name, const char_t *name, int64_t &attr_value) const
  取值时，用户需保证整型数据没有截断，同理针对int32_t和uint32_t混用时需要保证不被截断。
- 整型列表：接受std::vector<int64_t>、std::vector<int32_t>、std::vector<uint32_t>、std::initializer_list<int64_t>&&表示的整型列表数据
- 浮点数：float
- 浮点数列表：std::vector<float>
- 字符串：string
- 布尔：bool
- 布尔列表：std::vector<bool>

函数原型

Operator &SetInputAttr(const int32_t index, const char_t *name, const char_t *attr_value);

Operator &SetInputAttr(const char_t *dst_name, const char_t *name, const char_t *attr_value);

Operator &SetInputAttr(const int32_t index, const char_t *name, const AscendString &attr_value);

 版权所有 © 华为技术有限公司

Operator &SetInputAttr(const char_t *dst_name, const char_t *name, const AscendString &attr_value);

Operator &SetInputAttr(const int32_t index, const char_t *name, int64_t attr_value);

Operator &SetInputAttr(const char_t *dst_name, const char_t *name, int64_t attr_value);

Operator &SetInputAttr(const int32_t index, const char_t *name, int32_t attr_value);

Operator &SetInputAttr(const char_t *dst_name, const char_t *name, int32_t attr_value);

Operator &SetInputAttr(const int32_t index, const char_t *name, uint32_t attr_value);

Operator &SetInputAttr(const char_t *dst_name, const char_t *name, uint32_t attr_value);

Operator &SetInputAttr(const int32_t index, const char_t *name, bool attr_value);

Operator &SetInputAttr(const char_t *dst_name, const char_t *name, bool attr_value);

Operator &SetInputAttr(const int32_t index, const char_t *name, float32_t attr_value);

Operator &SetInputAttr(const char_t *dst_name, const char_t *name, float32_t attr_value);

Operator &SetInputAttr(const int32_t index, const char_t *name, const std::vector<AscendString> &attr_value);

Operator &SetInputAttr(const char_t *dst_name, const char_t *name, const std::vector<AscendString> &attr_value);

Operator &SetInputAttr(const int32_t index, const char_t *name, const std::vector<int64_t> &attr_value);

Operator &SetInputAttr(const char_t *dst_name, const char_t *name, const std::vector<int64_t> &attr_value);

Operator &SetInputAttr(const int32_t index, const char_t *name, const std::vector<int32_t> &attr_value);

Operator &SetInputAttr(const char_t *dst_name, const char_t *name, const std::vector<int32_t> &attr_value);

Operator &SetInputAttr(const int32_t index, const char_t *name, const std::vector<uint32_t> &attr_value);

Operator &SetInputAttr(const char_t *dst_name, const char_t *name, const std::vector<uint32_t> &attr_value);

Operator &SetInputAttr(const int32_t index, const char_t *name, const std::vector<bool> &attr_value);

 版权所有 © 华为技术有限公司

Operator &SetInputAttr(const char_t *dst_name, const char_t *name, const std::vector<bool> &attr_value);

Operator &SetInputAttr(const int32_t index, const char_t *name, const std::vector<float32_t> &attr_value);

Operator &SetInputAttr(const char_t *dst_name, const char_t *name, const std::vector<float32_t> &attr_value);

参数说明

参数名	输入/输出	描述
name	输入	属性名称。
index	输入	输入索引
dst_name	输入	输入边名称
attr_value	输入	需设置的int64_t表示的整型类型属性值。
attr_value	输入	需设置的int32_t表示的整型类型属性值。
attr_value	输入	需设置的uint32_t表示的整型类型属性值。
attr_value	输入	需设置的vector<int64_t>表示的整型列表类型属性值。
attr_value	输入	需设置的vector<int32_t>表示的整型列表类型属性值。
attr_value	输入	需设置的vector<uint32_t>表示的整型列表类型属性值。
attr_value	输入	需设置的浮点类型的属性值。
attr_value	输入	需设置的浮点列表类型的属性值。
attr_value	输入	需设置的布尔类型的属性值。
attr_value	输入	需设置的布尔列表类型的属性值。
attr_value	输入	需设置的字符串类型的属性值。
attr_value	输入	需设置的字符串列表类型的属性值。

返回值

参数名	类型	描述
-	Operator&	对象本身。

异常处理

无。

 版权所有 © 华为技术有限公司

约束说明

无。

6.3.3.30 SetOutputAttr

函数功能

设置算子输出Tensor属性的属性值。

算子可以包括多个属性，初次设置值后，算子属性值的类型固定，算子属性值的类型包括：

- 整型：接受int64_t、uint32_t、int32_t类型的整型值
  以int64_t为例，使用:
  SetInputAttr(const char_t *dst_name, const char_t *name, int64_t attr_value);
  SetInputAttr(const int32_t index, const char_t *name, int64_t attr_value);
  设置属性值，以:
  GetInputAttr(const int32_t index, const char_t *name, int64_t &attr_value) const;
  GetInputAttr(const char_t *dst_name, const char_t *name, int64_t &attr_value) const
  取值时，用户需保证整型数据没有截断，同理针对int32_t和uint32_t混用时需要保证不被截断。
- 整型列表：接受std::vector<int64_t>、std::vector<int32_t>、std::vector<uint32_t>、std::initializer_list<int64_t>&&表示的整型列表数据
- 浮点数：float
- 浮点数列表：std::vector<float>
- 字符串：string
- 布尔：bool
- 布尔列表：std::vector<bool>

函数原型

Operator &SetOutputAttr(const int32_t index, const char_t *name, const char_t *attr_value);

Operator &SetOutputAttr(const char_t *dst_name, const char_t *name, const char_t *attr_value);

Operator &SetOutputAttr(const int32_t index, const char_t *name, const AscendString &attr_value);

Operator &SetOutputAttr(const char_t *dst_name, const char_t *name, const AscendString &attr_value);

Operator &SetOutputAttr(const int32_t index, const char_t *name, int64_t attr_value);

Operator &SetOutputAttr(const char_t *dst_name, const char_t *name, int64_t attr_value);

 版权所有 © 华为技术有限公司

Operator &SetOutputAttr(const int32_t index, const char_t *name, int32_t attr_value);

Operator &SetOutputAttr(const char_t *dst_name, const char_t *name, int32_t attr_value);

Operator &SetOutputAttr(const int32_t index, const char_t *name, uint32_t attr_value);

Operator &SetOutputAttr(const char_t *dst_name, const char_t *name, uint32_t attr_value);

Operator &SetOutputAttr(const int32_t index, const char_t *name, bool attr_value);

Operator &SetOutputAttr(const char_t *dst_name, const char_t *name, bool attr_value);

Operator &SetOutputAttr(const int32_t index, const char_t *name, float32_t attr_value);

Operator &SetOutputAttr(const char_t *dst_name, const char_t *name, float32_t attr_value);

Operator &SetOutputAttr(const int32_t index, const char_t *name, const std::vector<AscendString> &attr_value);

Operator &SetOutputAttr(const char_t *dst_name, const char_t *name, const std::vector<AscendString> &attr_value);

Operator &SetOutputAttr(const int32_t index, const char_t *name, const std::vector<int64_t> &attr_value);

Operator &SetOutputAttr(const char_t *dst_name, const char_t *name, const std::vector<int64_t> &attr_value);

Operator &SetOutputAttr(const int32_t index, const char_t *name, const std::vector<int32_t> &attr_value);

Operator &SetOutputAttr(const char_t *dst_name, const char_t *name, const std::vector<int32_t> &attr_value);

Operator &SetOutputAttr(const int32_t index, const char_t *name, const std::vector<uint32_t> &attr_value);

Operator &SetOutputAttr(const char_t *dst_name, const char_t *name, const std::vector<uint32_t> &attr_value);

Operator &SetOutputAttr(const int32_t index, const char_t *name, const std::vector<bool> &attr_value);

Operator &SetOutputAttr(const char_t *dst_name, const char_t *name, const std::vector<bool> &attr_value);

Operator &SetOutputAttr(const int32_t index, const char_t *name, const std::vector<float32_t> &attr_value);

Operator &SetOutputAttr(const char_t *dst_name, const char_t *name, const std::vector<float32_t> &attr_value);

 版权所有 © 华为技术有限公司

参数说明

参数名	输入/输出	描述
name	输入	属性名称。
index	输入	输出索引
dst_name	输入	输出边名称
attr_value	输入	需设置的int64_t表示的整型类型属性值。
attr_value	输入	需设置的int32_t表示的整型类型属性值。
attr_value	输入	需设置的uint32_t表示的整型类型属性值。
attr_value	输入	需设置的vector<int64_t>表示的整型列表类型属性值。
attr_value	输入	需设置的vector<int32_t>表示的整型列表类型属性值。
attr_value	输入	需设置的vector<uint32_t>表示的整型列表类型属性值。
attr_value	输入	需设置的浮点类型的属性值。
attr_value	输入	需设置的浮点列表类型的属性值。
attr_value	输入	需设置的布尔类型的属性值。
attr_value	输入	需设置的布尔列表类型的属性值。
attr_value	输入	需设置的字符串类型的属性值。
attr_value	输入	需设置的字符串列表类型的属性值。

返回值

参数名	类型	描述
-	Operator&	对象本身。

异常处理

无。

约束说明

无。

 版权所有 © 华为技术有限公司

6.3.3.31 GetInputAttr

函数功能

根据属性名称获取算子输入Tensor对应的属性值。

函数原型

graphStatus GetInputAttr(const int32_t index, const char_t *name, AscendString &attr_value) const;

graphStatus GetInputAttr(const char_t *dst_name, const char_t *name, AscendString &attr_value) const;

graphStatus GetInputAttr(const int32_t index, const char_t *name, int64_t &attr_value) const;

graphStatus GetInputAttr(const char_t *dst_name, const char_t *name, int64_t &attr_value) const;

graphStatus GetInputAttr(const int32_t index, const char_t *name, int32_t &attr_value) const;

graphStatus GetInputAttr(const char_t *dst_name, const char_t *name, int32_t &attr_value) const;

graphStatus GetInputAttr(const int32_t index, const char_t *name, uint32_t &attr_value) const;

graphStatus GetInputAttr(const char_t *dst_name, const char_t *name, uint32_t &attr_value) const;

graphStatus GetInputAttr(const int32_t index, const char_t *name, bool &attr_value) const;

graphStatus GetInputAttr(const char_t *dst_name, const char_t *name, bool &attr_value) const;

graphStatus GetInputAttr(const int32_t index, const char_t *name, float32_t &attr_value) const;

graphStatus GetInputAttr(const char_t *dst_name, const char_t *name, float32_t &attr_value) const;

graphStatus GetInputAttr(const int32_t index, const char_t *name, std::vector<AscendString> &attr_value) const;

graphStatus GetInputAttr(const char_t *dst_name, const char_t *name, std::vector<AscendString> &attr_value) const;

graphStatus GetInputAttr(const int32_t index, const char_t *name, std::vector<int64_t> &attr_value) const;

graphStatus GetInputAttr(const char_t *dst_name, const char_t *name, std::vector<int64_t> &attr_value) const;

graphStatus GetInputAttr(const int32_t index, const char_t *name, std::vector<int32_t> &attr_value) const;

 版权所有 © 华为技术有限公司

graphStatus GetInputAttr(const char_t *dst_name, const char_t *name, std::vector<int32_t> &attr_value) const;

graphStatus GetInputAttr(const int32_t index, const char_t *name, std::vector<uint32_t> &attr_value) const;

graphStatus GetInputAttr(const char_t *dst_name, const char_t *name, std::vector<uint32_t> &attr_value) const;

graphStatus GetInputAttr(const int32_t index, const char_t *name, std::vector<bool> &attr_value) const;

graphStatus GetInputAttr(const char_t *dst_name, const char_t *name, std::vector<bool> &attr_value) const;

graphStatus GetInputAttr(const int32_t index, const char_t *name, std::vector<float32_t> &attr_value) const;

graphStatus GetInputAttr(const char_t *dst_name, const char_t *name, std::vector<float32_t> &attr_value) const;

参数说明

参数名	输入/输出	描述
name	输入	属性名称。
index	输入	输入索引。
dst_name	输入	输入边名称。
attr_value	输出	获取到的int64_t表示的整型类型属性值。
attr_value	输出	获取到的int32_t表示的整型类型属性值。
attr_value	输出	获取到的uint32_t表示的整型类型属性值。
attr_value	输出	获取到的vector<int64_t>表示的整型列表类型属性值。
attr_value	输出	获取到的vector<int32_t>表示的整型列表类型属性值。
attr_value	输出	获取到的vector<uint32_t>表示的整型列表类型属性值。
attr_value	输出	获取到的浮点类型的属性值。
attr_value	输出	获取到的浮点列表类型的属性值。
attr_value	输出	获取到的布尔类型的属性值。
attr_value	输出	获取到的布尔列表类型的属性值。
attr_value	输出	获取到的字符串类型的属性值。
attr_value	输出	获取到的字符串列表类型的属性值。

 版权所有 © 华为技术有限公司

返回值

参数名	类型	描述
-	graphStatus	找到对应属性，返回GRAPH_SUCCESS，否则返回GRAPH_FAILED。

异常处理

无。

约束说明

无。

6.3.3.32 GetOutputAttr

函数功能

根据属性名称获取算子输出Tensor对应的属性值。

函数原型

graphStatus GetOutputAttr(const int32_t index, const char_t *name, AscendString &attr_value) const;

graphStatus GetOutputAttr(const char_t *dst_name, const char_t *name, AscendString &attr_value) const;

graphStatus GetOutputAttr(const int32_t index, const char_t *name, int64_t &attr_value) const;

graphStatus GetOutputAttr(const char_t *dst_name, const char_t *name, int64_t &attr_value) const;

graphStatus GetOutputAttr(const int32_t index, const char_t *name, int32_t &attr_value) const;

graphStatus GetOutputAttr(const char_t *dst_name, const char_t *name, int32_t &attr_value) const;

graphStatus GetOutputAttr(const int32_t index, const char_t *name, uint32_t &attr_value) const;

graphStatus GetOutputAttr(const char_t *dst_name, const char_t *name, uint32_t &attr_value) const;

graphStatus GetOutputAttr(const int32_t index, const char_t *name, bool &attr_value) const;

graphStatus GetOutputAttr(const char_t *dst_name, const char_t *name, bool &attr_value) const;

 版权所有 © 华为技术有限公司

graphStatus GetOutputAttr(const int32_t index, const char_t *name, float32_t &attr_value) const;

graphStatus GetOutputAttr(const char_t *dst_name, const char_t *name, float32_t &attr_value) const;

graphStatus GetOutputAttr(const int32_t index, const char_t *name, std::vector<AscendString> &attr_value) const;

graphStatus GetOutputAttr(const char_t *dst_name, const char_t *name, std::vector<AscendString> &attr_value) const;

graphStatus GetOutputAttr(const int32_t index, const char_t *name, std::vector<int64_t> &attr_value) const;

graphStatus GetOutputAttr(const char_t *dst_name, const char_t *name, std::vector<int64_t> &attr_value) const;

graphStatus GetOutputAttr(const int32_t index, const char_t *name, std::vector<int32_t> &attr_value) const;

graphStatus GetOutputAttr(const char_t *dst_name, const char_t *name, std::vector<int32_t> &attr_value) const;

graphStatus GetOutputAttr(const int32_t index, const char_t *name, std::vector<uint32_t> &attr_value) const;

graphStatus GetOutputAttr(const char_t *dst_name, const char_t *name, std::vector<uint32_t> &attr_value) const;

graphStatus GetOutputAttr(const int32_t index, const char_t *name, std::vector<bool> &attr_value) const;

graphStatus GetOutputAttr(const char_t *dst_name, const char_t *name, std::vector<bool> &attr_value) const;

graphStatus GetOutputAttr(const int32_t index, const char_t *name, std::vector<float32_t> &attr_value) const;

graphStatus GetOutputAttr(const char_t *dst_name, const char_t *name, std::vector<float32_t> &attr_value) const;

参数说明

参数名	输入/输出	描述
name	输入	属性名称。
index	输入	输出索引。
dst_name	输入	输出边名称。
attr_value	输出	获取到的int64_t表示的整型类型属性值。
attr_value	输出	获取到的int32_t表示的整型类型属性值。
attr_value	输出	获取到的uint32_t表示的整型类型属性值。

 版权所有 © 华为技术有限公司

参数名	输入/输出	描述
attr_value	输出	获取到的vector<int64_t>表示的整型列表类型属性值。
attr_value	输出	获取到的vector<int32_t>表示的整型列表类型属性值。
attr_value	输出	获取到的vector<uint32_t>表示的整型列表类型属性值。
attr_value	输出	获取到的浮点类型的属性值。
attr_value	输出	获取到的浮点列表类型的属性值。
attr_value	输出	获取到的布尔类型的属性值。
attr_value	输出	获取到的布尔列表类型的属性值。
attr_value	输出	获取到的字符串类型的属性值。
attr_value	输出	获取到的字符串列表类型的属性值。

返回值

参数名	类型	描述
-	graphStatus	找到对应属性，返回GRAPH_SUCCESS，否则返回GRAPH_FAILED。

异常处理

无。

约束说明

无。

6.3.3.33 TryGetInputDesc

函数功能

根据算子Input名称获取算子Input的TensorDesc。

函数原型

须知

数据类型为string的接口后续版本会废弃，建议使用数据类型为非string的接口。

 版权所有 © 华为技术有限公司

graphStatus TryGetInputDesc(const string& name, TensorDesc& tensor_desc) const

graphStatus TryGetInputDesc(const char *name, TensorDesc &tensor_desc) const

参数说明

参数名	输入/输出	描述
name	输入	算子的Input名。
tensor_desc	输出	返回算子端口的当前设置格式，为TensorDesc对象。

返回值

参数名	类型	描述
-	graphStatus	True：有此端口，获取TensorDesc成功。 False：无此端口，出参为空，获取TensorDesc失败。

异常处理

异常场景	说明
无对应name输入	返回False。

约束说明

无。

6.3.3.34 UpdateInputDesc

函数功能

根据算子Input名称更新Input的TensorDesc。

函数原型

须知

数据类型为string的接口后续版本会废弃，建议使用数据类型为非string的接口。

 版权所有 © 华为技术有限公司

graphStatus UpdateInputDesc(const string& name, const TensorDesc& tensor_desc);

graphStatus UpdateInputDesc(const char *name, const TensorDesc &tensor_desc);

参数说明

参数名	输入/输出	描述
name	输入	算子Input名称。
tensor_desc	输入	TensorDesc对象。

返回值

参数名	类型	描述
-	graphStatus	更新TensorDesc成功，返回GRAPH_SUCCESS， 否则，返回GRAPH_FAILED。

异常处理

异常场景	说明
无对应name 输入	函数提前结束，返回GRAPH_FAILED。

约束说明

无。

6.3.3.35 UpdateOutputDesc

函数功能

根据算子Output名称更新Output的TensorDesc。

函数原型

须知

数据类型为string的接口后续版本会废弃，建议使用数据类型为非string的接口。

graphStatus UpdateOutputDesc(const string& name, const TensorDesc& tensor_desc)

 版权所有 © 华为技术有限公司

graphStatus UpdateOutputDesc(const char *name, const TensorDesc &tensor_desc)

参数说明

参数名	输入/输出	描述
name	输入	算子Output名称。
tensor_desc	输入	TensorDesc对象。

返回值

参数名	类型	描述
-	graphStatus	更新TensorDesc成功，返回GRAPH_SUCCESS， 否则，返回GRAPH_FAILED。

异常处理

无。

约束说明

无。

6.3.3.36 UpdateDynamicInputDesc

函数功能

根据name和index的组合更新算子动态Input的TensorDesc。

函数原型

须知

数据类型为string的接口后续版本会废弃，建议使用数据类型为非string的接口。

graphStatus UpdateDynamicInputDesc(const string& name, uint32_t index, const TensorDesc& tensor_desc)

graphStatus UpdateDynamicInputDesc(const char *name, uint32_t index, const TensorDesc &tensor_desc)

 版权所有 © 华为技术有限公司

参数说明

参数名	输入/输出	描述
name	输入	算子动态Input的名称。
index	输入	算子动态Input编号，编号从1开始。
tensor_desc	输入	TensorDesc对象。

返回值

参数名	类型	描述
-	graphStatus	更新动态Input成功，返回GRAPH_SUCCESS， 否则，返回GRAPH_FAILED。

异常处理

无。

约束说明

无。

6.3.3.37 UpdateDynamicOutputDesc

函数功能

根据name和index的组合更新算子动态Output的TensorDesc。

函数原型

须知

数据类型为string的接口后续版本会废弃，建议使用数据类型为非string的接口。

graphStatus UpdateDynamicOutputDesc(const string& name, uint32_t index, const TensorDesc& tensor_desc)

graphStatus UpdateDynamicOutputDesc(const char *name, uint32_t index, const TensorDesc &tensor_desc);

 版权所有 © 华为技术有限公司

参数说明

参数名	输入/输出	描述
name	输入	算子动态Output的名称。
index	输入	算子动态Output编号。
tensor_desc	输入	TensorDesc对象。

返回值

参数名	类型	描述
-	graphStatus	更新动态Output成功，返回GRAPH_SUCCESS， 否则，返回GRAPH_FAILED。

异常处理

无。

约束说明

无。

6.3.3.38 VerifyAllAttr

函数功能

根据disableCommonVerifier值，校验Operator中的属性是否有效，校验Operator的输入输出是否有效。

函数原型

graphStatus VerifyAllAttr(bool disable_common_verifier = false)

参数说明

参数名	输入/输出	描述
disable_common_verifier	输入	当false时，只校验属性有效性，当true时，增加校验Operator所有输入输出有效性。 默认值为false。

 版权所有 © 华为技术有限公司

返回值

参数名	类型	描述
-	graphStatus	推导成功，返回GRAPH_SUCCESS，否则，返回GRAPH_FAILED。

异常处理

无。

约束说明

无。

6.3.4 Tensor 类

Tensor的拷贝构造和赋值操作均共享Tensor信息，Tensor复制需使用Clone方法。

6.3.4.1 Tensor 构造函数和析构函数

函数功能

Tensor构造函数和析构函数。

函数原型

```
Tensor();
explicit Tensor(const TensorDesc &tensorDesc);
Tensor(const TensorDesc &tensorDesc, const std::vector<uint8_t> &data);
Tensor(const TensorDesc &tensorDesc, const uint8_t *data, size_t size);
Tensor(TensorDesc &&tensorDesc, std::vector<uint8_t> &&data);
~Tensor();
```

参数说明

参数名	输入/输出	描述
tensorDesc	输入	TensorDesc对象，需设置的Tensor描述符。
data	输入	需设置的数据。
size	输入	数据的长度，单位为字节。

 版权所有 © 华为技术有限公司

返回值

Tensor构造函数返回Tensor类型的对象。

异常处理

无。

约束说明

无。

6.3.4.2 Clone

函数功能

拷贝Tensor。

函数原型

Tensor Clone() const

参数说明

无。

返回值

参数名	类型	描述
-	Tensor	返回拷贝的Tensor对象。

异常处理

无。

约束说明

无。

6.3.4.3 IsValid

函数功能

判断Tensor对象是否有效。

若实际Tensor数据的大小与TensorDesc所描述的Tensor数据大小一致，则有效。

函数原型

graphStatus IsValid()

 版权所有 © 华为技术有限公司

参数说明

无。

返回值

参数名	类型	描述
-	graphStatus	如果Tensor对象有效，则返回GRAPH_SUCCESS，否则，返回GRAPH_FAILED。

异常处理

无。

约束说明

无。

6.3.4.4 GetData

函数功能

获取Tensor中的数据。

const uint8_t* GetData() const返回的数据不可修改，uint8_t* GetData()返回的数据可修改。

函数原型

const uint8_t* GetData() const;

uint8_t* GetData()

参数说明

无。

返回值

参数名	类型	描述
-	const Buffer	Tensor中所存放的数据。

异常处理

无。

 版权所有 © 华为技术有限公司

约束说明

无。

6.3.4.5 GetTensorDesc

函数功能

获取Tensor的描述符。

函数原型

TensorDesc GetTensorDesc() const

参数说明

无。

返回值

参数名	类型	描述
-	TensorDesc	返回当前Tensor的描述符。

异常处理

无。

约束说明

修改返回的TensorDesc信息，不影响Tensor对象中已有的TensorDesc信息。

6.3.4.6 GetSize

函数功能

获取Tensor中的数据的大小。

函数原型

size_t GetSize() const

参数说明

无。

 版权所有 © 华为技术有限公司

返回值

参数名	类型	描述
-	size_t	Tensor中存放的数据的大小，单位为字节。

异常处理

无。

约束说明

无。

6.3.4.7 SetData

函数功能

向Tensor中设置数据。

函数原型

须知

数据类型为string的接口后续版本会废弃，建议使用数据类型为非string的接口。

graphStatus SetData(std::vector<uint8_t> &&data);

graphStatus SetData(const std::vector<uint8_t> &data);

graphStatus SetData(const uint8_t *data, size_t size);

graphStatus SetData(const std::string &data);

graphStatus SetData(const char *data);

graphStatus SetData(const std::vector<std::string> &data);

graphStatus SetData(const std::vector<ge::AscendString> &datas);

graphStatus SetData(uint8_t *data, size_t size, const Tensor::DeleteFunc &deleter_func);

参数说明

参数名	输入/输出	描述
data	输入	需设置的数据。
size	输入	数据的长度，单位为字节。

 版权所有 © 华为技术有限公司

参数名	输入/输出	描述
deleter_func	输入	用于释放data数据。 using DeleteFunc = std::function<void(uint8_t *)>;

返回值

参数名	类型	描述
-	graphStatus	设置成功返回GRAPH_SUCCESS，否则，返回GRAPH_FAILED。

异常处理

无。

约束说明

无。

6.3.4.8 SetTensorDesc

函数功能

设置Tensor的描述符（TensorDesc）。

函数原型

graphStatus SetTensorDesc(const TensorDesc &tensorDesc)

参数说明

参数名	输入/输出	描述
tensorDesc	输入	需设置的Tensor描述符。

返回值

参数名	类型	描述
-	graphStatus	设置成功返回GRAPH_SUCCESS，否则，返回GRAPH_FAILED。

 版权所有 © 华为技术有限公司

异常处理

无。

约束说明

无。

6.3.5 TensorDesc 类

TensorDesc的拷贝构造和赋值操作均为值拷贝，不共享TensorDesc信息。

TensorDesc的Move构造和Move赋值，会将原有TensorDesc信息移动到新的TensorDesc对象中。

6.3.5.1 TensorDesc 构造函数和析构函数

函数功能

TensorDesc构造函数和析构函数。

函数原型

TensorDesc();

explicit TensorDesc(Shape shape, Format format = FORMAT_ND, DataType dt = DT_FLOAT);

TensorDesc(const TensorDesc& desc);

TensorDesc(TensorDesc&& desc);

TensorDesc &operator=(const TensorDesc &desc);

TensorDesc &operator=(TensorDesc &&desc);

~TensorDesc()

参数说明

参数名	输入/输出	描述
shape	输入	Shape对象。
format	输入	Format对象，默认取值FORMAT_ND。 关于Format数据类型的定义，请参见**6.3.10.1 Format**。
dt	输入	DataType对象，默认取值DT_FLOAT。 关于DataType数据类型的定义，请参见**6.3.10.2 DataType**。

 版权所有 © 华为技术有限公司

返回值

TensorDesc构造函数返回TensorDesc类型的对象。

异常处理

无。

约束说明

无。

6.3.5.2 GetDataType

函数功能

获取TensorDesc所描述Tensor的数据类型。

函数原型

DataType GetDataType() const

参数说明

无。

返回值

参数名	类型	描述（参数说明、取值范围等）
-	DataType	TensorDesc所描述的Tensor的数据类型。

异常处理

无。

约束说明

由于返回的DataType信息为值拷贝，因此修改返回的DataType信息，不影响TensorDesc中已有的DataType信息。

6.3.5.3 GetFormat

函数功能

获取TensorDesc所描述的Tensor的Format。

函数原型

Format GetFormat() const

 版权所有 © 华为技术有限公司

参数说明

无。

返回值

参数名	类型	描述
-	Format	TensorDesc所描述的Tensor的format信息。

异常处理

无。

约束说明

由于返回的Format信息为值拷贝，因此修改返回的Format信息，不影响TensorDesc中已有的Format信息。

6.3.5.4 GetName

函数功能

获取TensorDesc所描述Tensor的名称。

函数原型

须知

数据类型为string的接口后续版本会废弃，建议使用数据类型为非string的接口。

std::string GetName() const;

graphStatus GetName(ge::AscendString &name);

参数说明

参数名	输入/输出	描述
name	输出	算子名称。

 版权所有 © 华为技术有限公司

返回值

参数名	类型	描述
-	graphStatus	获取name成功，返回GRAPH_SUCCESS， 否则，返回GRAPH_FAILED。

异常处理

无。

约束说明

无。

6.3.5.5 GetOriginFormat

函数功能

获取TensorDesc所描述Tensor的原始Format。

函数原型

Format GetOriginFormat() const

参数说明

无。

返回值

参数名	类型	描述
-	Format	TensorDesc所描述的Tensor的originFormat信息。 关于Format数据类型的定义，请参见Format。

异常处理

无。

约束说明

无。

 版权所有 © 华为技术有限公司

6.3.5.6 GetOriginShape

函数功能

获取TensorDesc所描述Tensor的原始Shape。

函数原型

Shape GetOriginShape() const

参数说明

无。

返回值

参数名	类型	描述
-	Shape	TensorDesc描述的originShape。

异常处理

无。

约束说明

无。

6.3.5.7 GetPlacement

函数功能

获取Tensor中数据地址的类型。

函数原型

Placement GetPlacement() const

参数说明

无

返回值

参数名	输入/输出	描述
无	Placement	获取Tensor中数据地址的类型。

 版权所有 © 华为技术有限公司

异常处理

无。

约束说明

无。

6.3.5.8 GetRealDimCnt

函数功能

获取TensorDesc所描述Tensor的实际维度个数。

函数原型

int64_t GetRealDimCnt() const

参数说明

无。

返回值

参数名	类型	描述
-	int64_t	TensorDesc所描述的实际维度个数。

异常处理

无。

约束说明

无。

6.3.5.9 GetShape

函数功能

获取TensorDesc所描述Tensor的Shape。

函数原型

Shape GetShape() const

参数说明

无。

 版权所有 © 华为技术有限公司

返回值

参数名	类型	描述
-	Shape	TensorDesc描述的shape。

异常处理

无。

约束说明

由于返回的Shape信息为值拷贝，因此修改返回的Shape信息，不影响TensorDesc中已有的Shape信息。

6.3.5.10 GetShapeRange

函数功能

获取设置的shape变化范围。

函数原型

graphStatus GetShapeRange(std::vector<std::pair<int64_t,int64_t>> &range) const

参数说明

参数名	输入/输出	描述
range	输出	设置过的shape变化范围。

返回值

参数名	类型	描述
-	graphStatus	函数执行结果。若成功，则该值为GRAPH_SUCCESS(即0)，其他值则为执行失败。

异常处理

无。

约束说明

无。

 版权所有 © 华为技术有限公司

6.3.5.11 GetSize

函数功能

获取TensorDesc所描述Tensor的数据大小。

函数原型

int64_t GetSize() const

参数说明

无。

返回值

参数名	类型	描述
-	int64_t	TensorDesc所描述的Tensor的数据大小信息。

异常处理

无。

约束说明

无。

6.3.5.12 SetDataType

函数功能

向TensorDesc中设置Tensor的数据类型。

函数原型

void SetDataType(DataType dt)

参数说明

参数名	输入/输出	描述
dt	输入	需设置的TensorDesc所描述的Tensor的数据类型信息。 关于DataType类型，请参见**DataType**。

 版权所有 © 华为技术有限公司

返回值

无。

异常处理

无。

约束说明

无。

6.3.5.13 SetFormat

函数功能

向TensorDesc中设置Tensor的Format。

函数原型

void SetFormat(Format format)

参数说明

参数名	输入/输出	描述
format	输入	需设置的format信息。 关于Format类型，请参见6.3.10.1 Format。

返回值

无。

异常处理

无。

约束说明

无。

6.3.5.14 SetName

函数功能

向TensorDesc中设置Tensor的名称。

 版权所有 © 华为技术有限公司

函数原型

须知

数据类型为string的接口后续版本会废弃，建议使用数据类型为非string的接口。

void SetName(const std::string &name)

void SetName(const char *name)

参数说明

参数名	输入/输出	描述
name	输入	需设置的Tensor的名称。

返回值

无。

异常处理

无。

约束说明

无。

6.3.5.15 SetOriginFormat

函数功能

向TensorDesc中设置Tensor的原始Format。

该Format是指原始网络模型的Format。

函数原型

void SetOriginFormat(Format originFormat)

参数说明

参数名	输入/输出	描述
originFormat	输入	需设置的原始Format信息。 关于Format数据类型的定义，请参见Format。

 版权所有 © 华为技术有限公司

返回值

无。

异常处理

无。

约束说明

无。

6.3.5.16 SetOriginShape

函数功能

向TensorDesc中设置Tensor的原始Shape。

该Shape是指原始网络模型的Shape。

函数原型

void SetOriginShape(const Shape &originShape)

参数说明

参数名	输入/输出	描述
originShape	输入	向TensorDesc设置原始的originShape对象。

返回值

无。

异常处理

无。

约束说明

无。

6.3.5.17 SetPlacement

函数功能

设置Tensor中数据地址的类型。

函数原型

void SetPlacement(Placement placement)

 版权所有 © 华为技术有限公司

参数说明

参数名	输入/输出	描述

placement	输入	Tensor数据中地址的种类。
		枚举值定义如下：
		enum Placement {
		kPlacementHost = 0, // host data addr
		kPlacementDevice = 1, // device data addr
		};

返回值

无。

异常处理

无。

约束说明

无。

6.3.5.18 SetRealDimCnt

函数功能

向TensorDesc中设置Tensor的实际维度数目。

通过**GetShape**接口返回的Shape的维度可能存在补1的场景，因此可以通过该接口设置Shape的实际维度个数。

函数原型

void SetRealDimCnt(const int64_t realDimCnt)

参数说明

参数名	输入/输出	描述
realDimCnt	输入	需设置的TensorDesc的实际数据维度数目信息。

返回值

无。

异常处理

无。

 版权所有 © 华为技术有限公司

约束说明

无。

6.3.5.19 SetSize

函数功能

向TensorDesc中设置Tensor的数据大小。

函数原型

void SetSize(int64_t size)

参数说明

参数名	输入/输出	描述
size	输入	需设置的Tensor的数据大小信息。

返回值

无。

异常处理

无。

约束说明

无。

6.3.5.20 SetShape

函数功能

向TensorDesc中设置Tensor的Shape。

函数原型

void SetShape(const Shape &shape)

参数说明

参数名	输入/输出	描述
shape	输入	需向TensorDesc设置的shape对象。

 版权所有 © 华为技术有限公司

返回值

无。

异常处理

无。

约束说明

无。

6.3.5.21 SetShapeRange

函数功能

设置shape的变化范围。

函数原型

graphStatus SetShapeRange(const std::vector<std::pair<int64_t,int64_t>> &range)

参数说明

参数名	输入/输出	描述
range	输入	shape代表的变化范围。vector中的每一个元素为一个pair，pair的第一个值为该维度上的dim最小值，第二个值为该维度上dim的最大值。举例如下： 该tensor的shape为{1，1，-1，2}，第三个轴的最大值为100，则range可设置为{ {1,1},{1,1},{1,100},{2,2}}

返回值

参数名	类型	描述
-	graphStatus	函数执行结果。若成功，则该值为GRAPH_SUCCESS(即0)，其他值则为执行失败。

异常处理

无。

 版权所有 © 华为技术有限公司

约束说明

无。

6.3.5.22 SetUnknownDimNumShape

函数功能

设置tensor的shape为{-2}，用来表示tensor是完全未知的。

函数原型

graphStatus SetUnknownDimNumShape()

参数说明

无。

返回值

参数名	类型	描述
-	graphStatus	函数执行结果。执行成功，则该值为GRAPH_SUCCESS(即0)，其他值则为执行失败。

异常处理

无。

约束说明

无。

6.3.5.23 Update

函数功能

更新TensorDesc的shape、format、datatype属性。

函数原型

void Update(const Shape &shape, Format format = FORMAT_ND, DataType dt = DT_FLOAT)

参数说明

参数名	输入/输出	描述
shape	输入	需刷新的shape对象。

 版权所有 © 华为技术有限公司

参数名	输入/输出	描述
format	输入	需刷新的format对象，默认取值FORMAT_ND。
dt	输入	需刷新的datatype对象，默认取值DT_FLOAT。

返回值

无。

异常处理

无。

约束说明

无。

6.3.6 Shape 类

6.3.6.1 Shape 构造函数和析构函数

函数功能

Shape构造函数和析构函数。

函数原型

Shape();

~Shape();

explicit Shape(const std::vector<int64_t>& dims)

参数说明

参数名	输入/输出	描述
dims	输入	Shape的取值内容。 Shape表征张量数据的维度大小，用std::vector<int64_t>表征每一个维度的具体大小。

返回值

Shape构造函数返回Shape类型的对象。

 版权所有 © 华为技术有限公司

异常处理

无。

约束说明

无。

6.3.6.2 GetDim

函数功能

获取Shape第idx维的长度。

函数原型

int64_t GetDim(size_t idx) const

参数说明

参数名	输入/输出	描述
idx	输入	维度索引，索引从0开始。

返回值

参数名	类型	描述
-	int64_t	第idx维的长度。

异常处理

无。

约束说明

无。

6.3.6.3 GetDims

函数功能

获取Shape所有维度组成的向量。

函数原型

std::vector<int64_t> GetDims() const

 版权所有 © 华为技术有限公司

参数说明

无。

返回值

参数名	类型	描述
-	std::vector<int64_t>	Shape的所有维度组成的向量。

异常处理

无。

约束说明

无。

6.3.6.4 GetDimNum

函数功能

获取Shape的维度个数。

函数原型

size_t GetDimNum() const

参数说明

无。

返回值

参数名	类型	描述
-	size_t	Tensor Shape的维度个数。

异常处理

无。

约束说明

无。

 版权所有 © 华为技术有限公司

6.3.6.5 GetShapeSize

函数功能

获取Shape中所有dim的累乘结果。

函数原型

int64_t GetShapeSize() const

参数说明

无。

返回值

参数名	类型	描述
-	int64_t	返回所有dim的累乘结果。

异常处理

无。

约束说明

无。

6.3.6.6 SetDim

函数功能

将Shape中第idx维度的值设置为value。

函数原型

graphStatus SetDim(size_t idx, int64_t value)

参数说明

参数名	输入/输出	描述
idx	输入	Shape维度的索引，索引从0开始。
value	输入	需设置的值。

 版权所有 © 华为技术有限公司

返回值

参数名	类型	描述
-	graphStatus	设置成功返回GRAPH_SUCCESS，否则，返回GRAPH_FAILED。

异常处理

无。

约束说明

使用SetDim接口前，只能使用Shape(const std::vector<int64_t>& dims)构造shape对象。如果使用Shape()构造shape对象，使用SetDim接口将返回失败。

6.3.7 AttrValue 类

6.3.7.1 AttrValue 构造函数和析构函数

函数功能

AttrValue构造函数和析构函数。

函数原型

AttrValue();

~AttrValue()

参数说明

无。

返回值

AttrValue构造函数返回AttrValue类型的对象。

异常处理

无。

约束说明

无。

 版权所有 © 华为技术有限公司

6.3.7.2 CreateFrom

函数功能

将传入的DT类型（支持int64_t、float、std::string类型）的参数转换为对应T类型（支持INT、FLOAT、STR类型）的参数。

- 支持将int64_t类型转换为INT类型
- 支持将float类型转换为FLOAT类型
- 支持将std::string类型转换为STR类型

函数原型

template<typename T, typename DT>

static T CreateFrom(DT&& val)

参数说明

参数名	输入/输出	描述
val	输入	DT类型的参数。

返回值

参数名	类型	描述（参数说明、取值范围等）
-	• INT • FLOAT • STR	T类型的参数。

异常处理

无。

约束说明

无。

6.3.7.3 GetValue

函数功能

获取属性key-value键值对中的value值，并将value值从T类型转换为DT类型。

- 支持将INT类型转换为int64_t类型
- 支持将FLOAT类型转换为float类型

 版权所有 © 华为技术有限公司

- 支持将STR类型转换为std::string类型

函数原型

须知

数据类型为string的接口后续版本会废弃，建议使用数据类型为非string的接口。

template<typename T, typename DT>

graphStatus GetValue(DT& val) const

graphStatus GetValue(ge::AscendString &val)

参数说明

参数名	输入/输出	描述
val	输出	DT类型的参数。

返回值

参数名	类型	描述（参数说明、取值范围等）
-	graphStatus	数据类型转换成功，返回GRAPH_SUCCESS， 否则，返回GRAPH_FAILED。

异常处理

无。

约束说明

无。

6.3.8 Memblock 类

6.3.8.1 MemBlock 构造函数和析构函数

函数功能

MemBlock构造函数和析构函数。

函数原型

MemBlock(Allocator &allocator, void *addr, size_t block_size)

 版权所有 © 华为技术有限公司

```
: allocator_(allocator), addr_(addr), count_(1U), block_size_(block_size) {}
virtual ~MemBlock() = default;
```

参数说明

参数名	输入/输出	描述
allocator	输入	用户根据6.3.9 Allocator类派生的类的引用。
addr	输入	device内存地址。
block_size	输入	device内存addr的大小。

返回值

MemBlock构造函数返回MemBlock类型的对象。

异常处理

无。

约束说明

用户继承6.3.9 Allocator类后，申请内存需要返回MemBlock类型指针，用户只需按构造函数构造MemBlock对象即可，析构函数根据用户需求可以自定义，避免内存泄露。

6.3.8.2 GetAddr

函数功能

获取只读的device内存地址。

函数原型

```
const void *GetAddr() const
```

参数说明

无。

返回值

参数名	类型	描述（参数说明、取值范围等）
-	void*	只读的device内存地址。

 版权所有 © 华为技术有限公司

异常处理

无。

约束说明

无。

6.3.8.3 GetAddr

函数功能

获取可读写的device内存地址。

函数原型

void *GetAddr()

参数说明

无。

返回值

参数名	类型	描述（参数说明、取值范围等）
-	void*	可读写的device内存地址。

异常处理

无。

约束说明

无。

6.3.8.4 GetSize

函数功能

获取device内存对应的大小。

函数原型

size_t GetSize() const

参数说明

无。

 版权所有 © 华为技术有限公司

返回值

参数名	类型	描述（参数说明、取值范围等）
-	size_t	device内存大小。

异常处理

无。

约束说明

无。

6.3.8.5 SetSize

函数功能

设置device内存地址大小。

函数原型

void SetSize(const size_t mem_size)

参数说明

参数名	输入/输出	描述
mem_size	输入	device内存大小。

返回值

无。

异常处理

无。

约束说明

无。

6.3.8.6 Free

函数功能

MemBlock的引用计数减为0时，释放MemBlock到内存池。

 版权所有 © 华为技术有限公司

函数原型

void Free()

参数说明

无。

返回值

无。

异常处理

无。

约束说明

无。

6.3.8.7 AddCount

函数功能

MemBlock引用计数加1。

函数原型

size_t AddCount()

参数说明

无。

返回值

参数名	类型	描述（参数说明、取值范围等）
-	size_t	返回加1后的引用计数。

异常处理

无。

约束说明

无。

 版权所有 © 华为技术有限公司

6.3.8.8 SubCount

函数功能

MemBlock引用计数减1。

函数原型

size_t SubCount()

参数说明

无。

返回值

参数名	类型	描述（参数说明、取值范围等）
-	size_t	返回减1后的引用计数。

异常处理

无。

约束说明

无。

6.3.8.9 GetCount

函数功能

获取MemBlock的引用计数。

函数原型

size_t GetCount() const

参数说明

无。

返回值

参数名	类型	描述（参数说明、取值范围等）
-	size_t	返回引用计数。

 版权所有 © 华为技术有限公司

异常处理

无。

约束说明

无。

6.3.9 Allocator 类

6.3.9.1 Allocator 构造函数和析构函数

函数功能

Allocator构造函数和析构函数。

函数原型

Allocator() = default;

virtual~Allocator() = default;

参数说明

无。

返回值

无。

异常处理

无。

约束说明

纯虚类需要用户派生。

6.3.9.2 Malloc

函数功能

在用户内存池中根据指定size大小申请device内存。

函数原型

virtual MemBlock *Malloc(size_t size) = 0;

 版权所有 © 华为技术有限公司

参数说明

参数名	输入/输出	描述
size	输入	指定需要申请内存大小。

返回值

参数名	类型	描述（参数说明、取值范围等）
-	MemBlock*	返回6.3.8 Memblock类指针。

异常处理

无。

约束说明

纯虚函数用户必须实现。

6.3.9.3 Free

函数功能

根据指定的MemBlock释放内存到内存池。

函数原型

virtual void Free(MemBlock *block) = 0;

参数说明

参数名	输入/输出	描述
block	输入	内存block指针。

返回值

无。

异常处理

无。

约束说明

纯虚函数用户必须实现。

 版权所有 © 华为技术有限公司

6.3.9.4 MallocAdvise

函数功能

在用户内存池中根据指定size大小申请device内存，建议申请的内存地址为addr。

函数原型

virtual MemBlock *MallocAdvise(size_t size, void *addr)

参数说明

参数名	输入/输出	描述
size	输入	指定需要申请内存大小。
addr	输入	建议申请的内存地址为addr。

返回值

参数名	类型	描述（参数说明、取值范围等）
-	MemBlock*	返回6.3.8 Memblock类指针。

异常处理

无。

约束说明

虚函数需要用户实现，如若未实现，默认同6.3.9.2 Malloc功能相同。

6.3.10 数据类型

6.3.10.1 Format

```
enum Format {
  FORMAT_NCHW = 0,   // NCHW
  FORMAT_NHWC,       // NHWC
  FORMAT_ND,         // Nd Tensor
  FORMAT_NC1HWC0,    // NC1HWC0
  FORMAT_FRACTAL_Z,  // FRACTAL_Z
  FORMAT_NC1C0HWPAD,
  FORMAT_NHWC1C0,
  FORMAT_FSR_NCHW,
  FORMAT_FRACTAL_DECONV,
  FORMAT_C1HWNC0,
  FORMAT_FRACTAL_DECONV_TRANSPOSE,
  FORMAT_FRACTAL_DECONV_SP_STRIDE_TRANS,
  FORMAT_NC1HWC0_C04,    // NC1HWC0, C0 =4
  FORMAT_FRACTAL_Z_C04,  // FRACZ, C0 =4
  FORMAT_CHWN,
  FORMAT_FRACTAL_DECONV_SP_STRIDE8_TRANS,
```

 版权所有 © 华为技术有限公司

```
  FORMAT_HWCN,
  FORMAT_NC1KHKWHWC0,  // KH,KW kernel h& kernel w maxpooling max output format
  FORMAT_BN_WEIGHT,
  FORMAT_FILTER_HWCK,  // filter input tensor format
  FORMAT_HASHTABLE_LOOKUP_LOOKUPS = 20,
  FORMAT_HASHTABLE_LOOKUP_KEYS,
  FORMAT_HASHTABLE_LOOKUP_VALUE,
  FORMAT_HASHTABLE_LOOKUP_OUTPUT,
  FORMAT_HASHTABLE_LOOKUP_HITS = 24,
  FORMAT_C1HWNCoC0,
  FORMAT_MD,
  FORMAT_NDHWC,
  FORMAT_FRACTAL_ZZ,
  FORMAT_FRACTAL_NZ,
  FORMAT_NCDHW,
  FORMAT_DHWCN,  // 3D filter input tensor format
  FORMAT_NDC1HWC0,
  FORMAT_FRACTAL_Z_3D,
  FORMAT_CN,
  FORMAT_NC,
  FORMAT_DHWNC,
  FORMAT_FRACTAL_Z_3D_TRANSPOSE, // 3D filter(transpose) input tensor format
  FORMAT_FRACTAL_ZN_LSTM,
  FORMAT_FRACTAL_Z_G,
  FORMAT_RESERVED,
  FORMAT_ALL,
  FORMAT_NULL,
  FORMAT_ND_RNN_BIAS,
  FORMAT_FRACTAL_ZN_RNN,
  FORMAT_NYUV,
  FORMAT_NYUV_A,
  FORMAT_END,
  FORMAT_MAX = 0xff
};
```

IR构图不支持输入以下FORMAT：

```
NC1HWC0
FRACTAL_Z
NC1C0HWPAD
NHWC1C0
FRACTAL_DECONV
C1HWNC0
FRACTAL_DECONV_TRANSPOSE
FRACTAL_DECONV_SP_STRIDE_TRANS
NC1HWC0_C04
FRACTAL_Z_C04
FRACTAL_DECONV_SP_STRIDE8_TRANS
NC1KHKWHWC0
C1HWNCoC0
FRACTAL_ZZ
FRACTAL_NZ
NDC1HWC0
FORMAT_FRACTAL_Z_3D
FORMAT_FRACTAL_Z_3D_TRANSPOSE
FORMAT_FRACTAL_ZN_LSTM
FORMAT_FRACTAL_Z_G
FORMAT_ND_RNN_BIAS
FORMAT_FRACTAL_ZN_RNN
FORMAT_NYUV
FORMAT_NYUV_A
```

6.3.10.2 DataType

```
enum DataType {
    DT_FLOAT = 0,           // float type
    DT_FLOAT16 = 1,         // fp16 type
    DT_INT8 = 2,            // int8 type
    DT_INT16 = 6,     // int16 type
```

 版权所有 © 华为技术有限公司

```
    DT_UINT16 = 7,      // uint16 type
    DT_UINT8 = 4,           // uint8 type
    DT_INT32 = 3,           //
    DT_INT64 = 9,           // int64 type
    DT_UINT32 = 8,          // unsigned int32
    DT_UINT64 = 10,         // unsigned int64
    DT_BOOL = 12,           // bool type
    DT_DOUBLE = 11,         // double type
    DT_STRING = 13,         // string type
    DT_DUAL_SUB_INT8 = 14,    /**< dual output int8 type */
    DT_DUAL_SUB_UINT8 = 15,    /**< dual output uint8 type */
    DT_COMPLEX64 = 16,        // complex64 type
    DT_COMPLEX128 = 17,       // complex128 type
    DT_QINT8 = 18,            // qint8 type
    DT_QINT16 = 19,           // qint16 type
    DT_QINT32 = 20,           // qint32 type
    DT_QUINT8 = 21,           // quint8 type
    DT_QUINT16 = 22,          // quint16 type
    DT_RESOURCE = 23,         // resource type
    DT_STRING_REF = 24,       // string ref type
    DT_DUAL = 25,             // dual output type
    DT_VARIANT = 26,          // dt_variant type
    DT_BF16 = 27,             // bf16 type,the current version does not support this enumerated value.
    DT_UNDEFINED = 28,        // Used to indicate a DataType field has not been set.
    DT_INT4 = 29,             // int4 type
    DT_MAX                    // Mark the boundaries of data types
  };
```

6.3.10.3 TensorType

TensorType类用以定义输入或者输出支持的数据类型，TensorType提供以下接口指定支持的数据类型：

```
struct TensorType {
  explicit TensorType(DataType dt);

  TensorType(const std::initializer_list<DataType> &types);

  static TensorType ALL() {
    return TensorType{DT_BOOL,   DT_COMPLEX128, DT_COMPLEX64, DT_DOUBLE, DT_FLOAT,
DT_FLOAT16, DT_INT16,
                      DT_INT32,  DT_INT64,      DT_INT8,      DT_QINT16, DT_QINT32, DT_QINT8,   DT_QUINT16,
                      DT_QUINT8, DT_RESOURCE,   DT_STRING,    DT_UINT16, DT_UINT32, DT_UINT64,
DT_UINT8};
  }

  static TensorType QuantifiedType() { return TensorType{DT_QINT16, DT_QINT32, DT_QINT8, DT_QUINT16,
DT_QUINT8}; }

  static TensorType OrdinaryType() {
    return TensorType{DT_BOOL,  DT_COMPLEX128, DT_COMPLEX64, DT_DOUBLE, DT_FLOAT,  DT_FLOAT16,
DT_INT16,
                      DT_INT32, DT_INT64,      DT_INT8,      DT_UINT16, DT_UINT32, DT_UINT64,  DT_UINT8};
  }

  static TensorType BasicType() {
    return TensorType{DT_COMPLEX128, DT_COMPLEX64, DT_DOUBLE, DT_FLOAT,  DT_FLOAT16, DT_INT16,
                      DT_INT32,      DT_INT64,     DT_INT8,   DT_QINT16, DT_QINT32,  DT_QINT8,
                      DT_QUINT16,    DT_QUINT8,    DT_UINT16, DT_UINT32, DT_UINT64,  DT_UINT8};
  }

  static TensorType NumberType() {
    return TensorType{DT_COMPLEX128, DT_COMPLEX64, DT_DOUBLE, DT_FLOAT,  DT_FLOAT16, DT_INT16,
DT_INT32,  DT_INT64,
                      DT_INT8,       DT_QINT32,    DT_QINT8,  DT_QUINT8, DT_UINT16,  DT_UINT32, DT_UINT64,
DT_UINT8};
  }

  static TensorType RealNumberType() {
```

 版权所有 © 华为技术有限公司

```
  return TensorType{DT_DOUBLE, DT_FLOAT,  DT_FLOAT16, DT_INT16,  DT_INT32, DT_INT64,
                    DT_INT8,   DT_UINT16, DT_UINT32,  DT_UINT64, DT_UINT8};
 }

 static TensorType ComplexDataType() { return TensorType{DT_COMPLEX128, DT_COMPLEX64}; }

 static TensorType IntegerDataType() {
  return TensorType{DT_INT16, DT_INT32, DT_INT64, DT_INT8, DT_UINT16, DT_UINT32, DT_UINT64,
DT_UINT8};
 }

 static TensorType SignedDataType() { return TensorType{DT_INT16, DT_INT32, DT_INT64, DT_INT8}; }

 static TensorType UnsignedDataType() { return TensorType{DT_UINT16, DT_UINT32, DT_UINT64,
DT_UINT8}; }

 static TensorType FloatingDataType() { return TensorType{DT_DOUBLE, DT_FLOAT, DT_FLOAT16}; }

 static TensorType IndexNumberType() { return TensorType{DT_INT32, DT_INT64}; }

 static TensorType UnaryDataType() { return TensorType{DT_COMPLEX128, DT_COMPLEX64, DT_DOUBLE,
DT_FLOAT, DT_FLOAT16}; }

 static TensorType FLOAT() { return TensorType{DT_FLOAT, DT_FLOAT16}; }

 std::shared_ptr<TensorTypeImpl> tensor_type_impl_;
};
```

6.3.10.4 UsrQuantizeFactor

```
struct UsrQuantizeFactor
{
public:
  //QuantizeScaleMode scale_mode;
  UsrQuantizeScaleMode scale_mode{USR_NORMAL_MODE};
  std::vector<uint8_t> scale_value;
  int64_t scale_offset{0};
  std::vector<uint8_t> offset_data_value;
  int64_t offset_data_offset{0};
  std::vector<uint8_t> offset_weight_value;
  int64_t offset_weight_offset{0};
  std::vector<uint8_t> offset_pad_value;
  int64_t offset_pad_offset{0};

  USR_TYPE_DEC(UsrQuantizeScaleMode, scale_mode);
  USR_TYPE_BYTES_DEC(scale_value);


  USR_TYPE_DEC(int64_t, scale_offset);
  USR_TYPE_BYTES_DEC(offset_data_value);
  USR_TYPE_DEC(int64_t, offset_data_offset);


  USR_TYPE_BYTES_DEC(offset_weight_value);
  USR_TYPE_DEC(int64_t, offset_weight_offset);
  USR_TYPE_BYTES_DEC(offset_pad_value);
  USR_TYPE_DEC(int64_t, offset_pad_offset);


};
```

6.3.10.5 TensorDescInfo

```
  struct TensorDescInfo {
    Format format_ = FORMAT_RESERVED;       /* tbe op register support format */
    DataType dataType_ = DT_UNDEFINED;      /* tbe op register support datatype */
  };
```

Format为枚举类型，定义请参考**6.3.10.1 Format**。

 版权所有 © 华为技术有限公司

DataType为枚举类型，定义请参考**6.3.10.2 DataType**。

6.3.10.6 GetSizeByDataType

函数功能

根据传入的data_type，获取该data_type所占用的内存大小。

函数原型

int GetSizeByDataType(DataType data_type)

参数说明

参数	输入/输出	说明
data_type	输入	数据类型，请参见**6.3.10.2 DataType**。

返回值

该data_type所占用的内存大小（单位为bytes），如果传入非法值或不支持的数据类型，返回-1。

异常处理

无。

约束说明

无。

6.3.10.7 GetFormatName

函数功能

根据传入的format类型，获取format的字符串描述。

函数原型

const char *GetFormatName(Format format)

参数说明

参数	输入/输出	说明
format	输入	format枚举值。

 版权所有 © 华为技术有限公司

返回值

该format所对应的字符串描述，若format不合法或不被识别，则返回nullptr。

异常处理

无。

约束说明

返回的字符串不可被修改。

6.3.10.8 GetFormatFromSub

函数功能

根据传入的主format和子format信息得到实际的format。

实际format为4字节大小，第1个字节的高四位为预留字段，低四位为c0 fromat，第2-3字节为子format信息，第4字节为主format信息，如下：

/*

* --

* | 4 bits | 4bits | 2 bytes | 1 byte |

* |------------|-------------|----------------|--------|

* | reserved | c0 -format | sub-format | format |

* --

*/

函数原型

int32_t GetFormatFromSub(int32_t primary_format, int32_t sub_format)

参数说明

参数	输入/输出	说明
primary_format	输入	主format信息，值不超过0xff。
sub_format	输入	子format信息，值不超过0xffff。

返回值

指定的主format和子format对应的实际format。

异常处理

无。

 版权所有 © 华为技术有限公司

约束说明

无。

6.3.10.9 GetPrimaryFormat

函数功能

从实际format中解析出主format信息。

函数原型

int32_t GetPrimaryFormat(int32_t format)

参数说明

参数	输入/输出	说明
format	输入	实际format（4字节大小，第1个字节的高四位为预留字段，低四位为c0 fromat，第2-3字节为子format信息，第4字节为主format信息）。

返回值

实际format中包含的主format。

异常处理

无。

约束说明

无。

6.3.10.10 GetSubFormat

函数功能

从实际format中解析出子format信息。

函数原型

int32_t GetSubFormat(int32_t format)

 版权所有 © 华为技术有限公司

参数说明

参数	输入/输出	说明
format	输入	实际format（4字节大小，第1个字节的高四位为预留字段，低四位为c0 fromat段，第2-3字节为子format信息，第4字节为主format信息）。

返回值

实际format中包含的子format。

异常处理

无。

约束说明

无。

6.3.10.11 HasSubFormat

函数功能

判断实际format中是否包含子format。

函数原型

bool HasSubFormat(int32_t format)

参数说明

参数	输入/输出	说明
format	输入	实际format（4字节大小，第1个字节的高四位为预留字段，低四位为c0 fromat，第2-3字节为子format信息，第4字节为主format信息）。

返回值

true：实际format中包含子format；false：实际format中不包含子format。

异常处理

无。

 版权所有 © 华为技术有限公司

约束说明

无。

6.3.10.12 HasC0Format

函数功能

判断实际format中是否包含C0 format。

函数原型

bool HasC0Format(int32_t format)

参数说明

参数	输入/输出	说明
format	输入	实际format（4字节大小，第1个字节的高四位为预留字段，低四位为c0 fromat，第2-3字节为子format信息，第4字节为主format信息）。

返回值

- true：实际format中包含c0 format。
- false：实际format中不包含c0 format。

异常处理

无。

约束说明

无。

6.3.10.13 GetC0Value

函数功能

从实际format中解析出c0 format信息。

函数原型

int64_t GetC0Value(int32_t format)

 版权所有 © 华为技术有限公司

参数说明

参数	输入/输出	说明
format	输入	实际format（4字节大小，第1个字节的高四位为预留字段，低四位为c0 fromat，第2-3字节为子format信息，第4字节为主format信息）。

返回值

- 如果包含c0 format，返回实际format中包含的c0 format。
- 如果不包含c0 format，返回-1。

异常处理

无。

约束说明

设置实际format格式时，第一个字节低四位的c0 format的范围只支持x=(0001~1111)，实际获取的c0 value为2^x-1。

 版权所有 © 华为技术有限公司