第四节课-构建计算图关系和执行顺序

> 本课程赞助方：`Datawhale`
>
> 作者：傅莘莘
>
> 主项目：https://github.com/zjhellofss/KuiperInfer 欢迎大家点赞和PR.
>
> 课程代码：https://github.com/zjhellofss/kuiperdatawhale/course4

上节课回忆

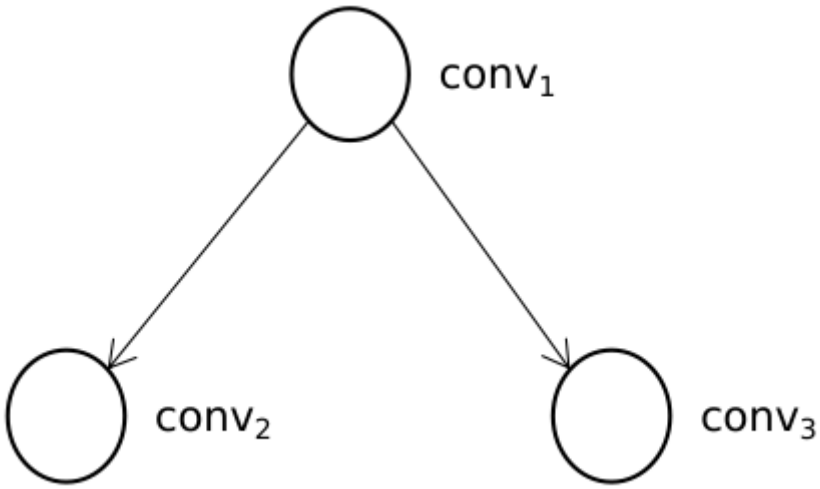
在上一节课中，我们完成了`KuiperInfer`中计算图类的实例化，主要包括了**构建计算图**中每个计算节点(`RuntimeOperator`)的权重信息(`RuntimeAttribute`)、参数信息(`RuntimeParameter`)以及输入输出张量(`input tensor`,`output tensor`)等信息。但是与一个完整的计算图相比，上节课中完成的计算图还相差以下的两个部分：

- 计算图中所有计算节点的执行顺序；
- 计算节点相关的输入输出张量初始化。

所以，我们在课程的以下内容中将着重讲解以上的两点。

计算节点的执行顺序

我们知道，深度学习模型是一个有向无环图。对于**有向图结构中的节点**，可以认为是深度学习模型中的**计算节点（算子）**，而**有向图结构中的边**可以认为是算子之间**连接和前后依赖关系**。



上图的深度学习模型同样也是一个有向无环图，而在这个图中共有三个计算节点(`conv1`, `conv2`和`conv3`). 另外在节点和节点之间也有边连接，`conv1`和`conv2`之间以及`conv1`和`conv3`之间。这些连接的边指定了节点执行的先后顺序，**必须先执行`conv1`，再执行`conv2`或`conv3`**.

那么我们怎么才能确定一个深度学习模型中节点之间的执行顺序呢？我们以下将会讲解一个比较有意思的算法**拓扑排序**。

拓扑排序

对于一个有向无环图，拓扑排序总能够找到一个**节点序列**，在这个序列中，**每个节点的前驱节点都能排在这个节点的前面**。什么是前驱节点呢，也就是对于**有向图中任意一条边的起点，我们可以认为它是终点节点的前驱节点。**

对于上图例子中的深度学习模型，`conv1`是`conv2`的前驱节点，`conv1`也是`conv3`的前驱节点，所以在计算执行节点序列时，`conv1`必须出现在`conv2`和`conv3`的前面，而`conv2`和`conv3`之间的顺序因为没有节点之间的连接而不做要求。因此执行节点顺序有以下两种：

- conv1->conv2->conv3
- conv1->conv3->conv2

基于深度优先的拓扑排序计算步骤

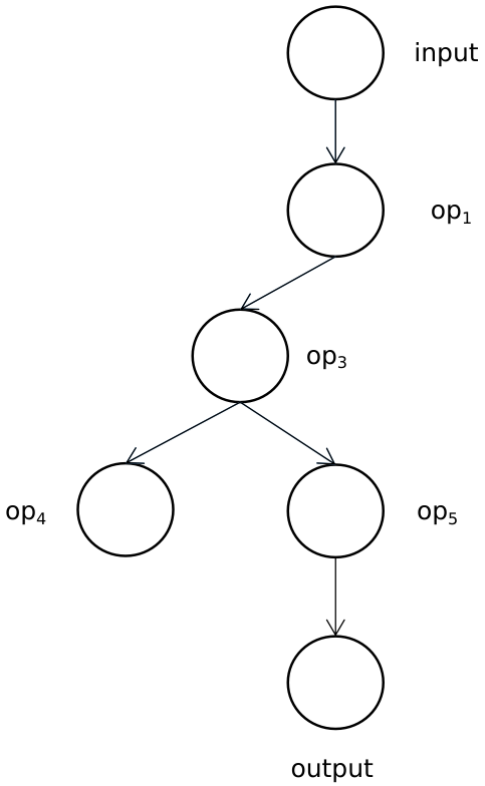
有计算排序的函数为`ReverseTopo`. `ReverseTopo`有参数`current_op`.

1. 选定一个入度为零的节点(`current_op`)，入度为零指的是**该节点没有前驱节点或所有前驱节点已经都被执行过**，在选定的同时将该节点的已执行标记置为`True`，并将该节点传入到`ReverseTopo`函数中；
2. 遍历1步骤中节点的后继节点(`current_op->output_operators`)；
3. 如果1的某个后继节点没有被执行过（已执行标记为`False`），则递归将**该后继节点**传入到`ReverseTopo`函数中；
4. 第2步中的遍历结束后，我们将当前节点放入到执行队列(`topo_operators_`)中。

当该函数结束后，我们对**执行队列中的排序结果做逆序就得到了最终拓扑排序的结果**，我们来看看具体的代码：

```
void RuntimeGraph::ReverseTopo(
    const std::shared_ptr<RuntimeOperator>& current_op) {
  CHECK(current_op != nullptr) << "current operator is
nullptr";
  current_op->has_forward = true;
  const auto& next_ops = current_op->output_operators;
  for (const auto& [_, op] : next_ops) {
    if (op != nullptr) {
      if (!op->has_forward) {
        this->ReverseTopo(op);
      }
    }
  }
  for (const auto& [_, op] : next_ops) {
    CHECK_EQ(op->has_forward, true);
  }
  this->topo_operators_.push_back(current_op);
}
```

一个复杂点的例子



对于上图的这个例子，`input`节点是其中唯一的输入节点，`output`节点是其中唯一的输出节点，其他都是计算节点（`RuntieOperator`），表示深度学习模型中的一个算子。我们下面就对这个模型进行拓扑排序来求得它正确的执行顺序。要注意的是，**拓扑排序可以有多种结果的序列，不同种序列之间的顺序也可能不完全相同，但是不同种序列均不会影响模型最终的预测输出。**

我们对上图的深度学习模型进行手工计算，计算步骤符合上一小节中的代码`ReverseTopo`函数。

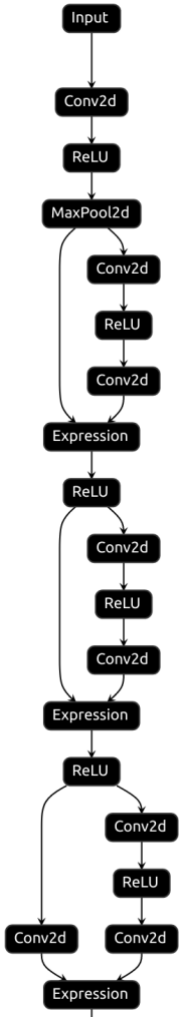
1. 首选，传入到`ReverseTopo`函数中的是`input`节点；
2. 对`input`节点进行遍历，`input`节点的后继节点中没有执行过的只有`op1`，所以我们**以递归的方式**将`op1`传入到`ReverseTopo`函数中；
3. 对`op1`节点进行遍历，`op1`有后继节点为`op3`没有被访问过，所以我们以递归的形式将`op3`传入到`ReverseTopo`函数中；
4. 对`op3`节点进行遍历，分别有后继节点`op4`和`op5`，均没有被访问过，所以我们都以递归的形式将`op4`传入到`ReverseTopo`函数当中；
5. 因为`op4`没有后继节点，会跳出**for (const auto& [_, op] : next_ops)**循环，所以会**将本节点放入到执行队列中**(`op4`)。随后该栈函数返回，**返回到4步中以递归的形式将`op5`传入到`ReverseTopo`函数中；**

6. `op5`的后继节点进行遍历，后继节点为`output`节点，`output`没有被访问过，所以再以递归的形式将`op5`传入到`Reversetopo`函数中。
7. 最后由于`output`节点没有后继节点，我们将`output`节点**放入到执行队列中**(`op4`,`output`)。随后，该栈函数返回，返回到6中，因为`op5`的后继节点已经都被访问过，所以将`op5`放入到执行队列中(`op4`,`output`,`op5`)。
8. 从6再返回到4中，因为`op3`的所有后继都已经被访问过，所以将`op3`**放入到执行队列中**(`op4`,`output`, `op5`, `op3`).
9. 从4返回到3中，因为`op1`的后继节点已经都被访问过，所以将`op1`放入到执行队列中(`op4`,`output`, `op5`, `op3`,`op1`).
10. 从3返回到2中，将`input`节点**放入到执行队列中**，有(`op4`,`output`,`op5`, `op3`,`op1`,`input`).
11. 最后需要对执行队列来一个逆序并得到最后的结果：(`input`,`op1`,`op3`,`op5`,`output`,`op4`).

单元测试

> 我们用单元来测试来理解拓扑排序的过程，深度学习模型结构选用了Resnet18网络。

本小节进行拓扑排序的深度学习模型的部分结构如下图所示，不难看出是一个比较典型的残差网络结构图。但是由于篇幅的关系，我们下图中只截取了部分的模型结构图。



计算拓扑排序的单元测试代码

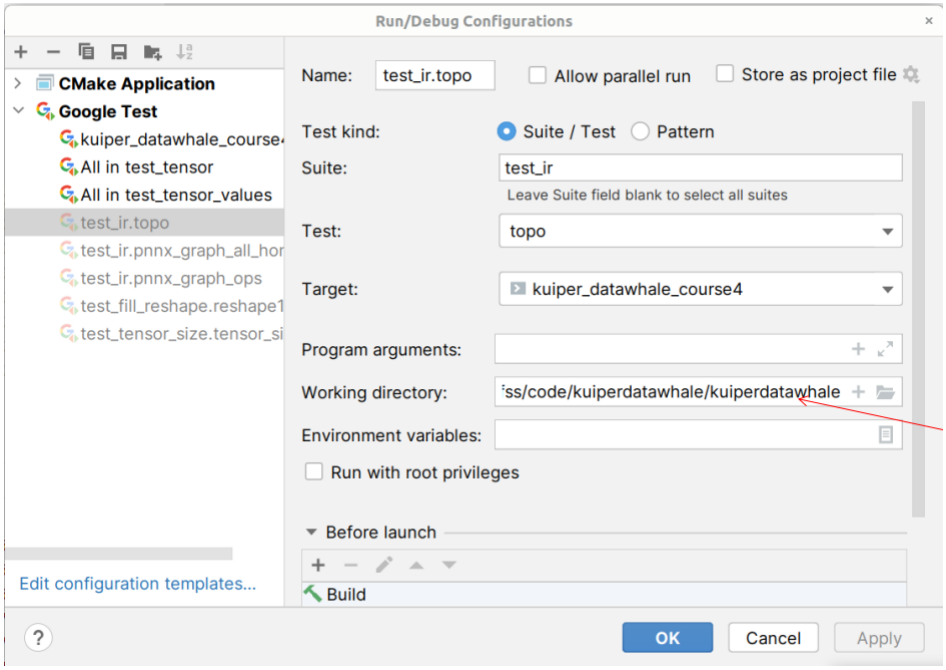
详细代码位于`course4/test`文件夹下，这里的代码调用了`Build`函数来构建模型中算子之间的拓扑顺序。

```
TEST(test_ir, topo) {
  using namespace kuiper_infer;
  std::string
bin_path("course4/model_file/resnet18_batch1.pnnx.bin");
  std::string
param_path("course4/model_file/resnet18_batch1.param");
  RuntimeGraph graph(param_path, bin_path);
  const bool init_success = graph.Init();
  ASSERT_EQ(init_success, true);
  graph.Build("pnnx_input_0", "pnnx_output_0");
  ...
  ...
```

```
  const auto &topo_queues = graph.get_topo_queues();

  int index = 0;
  for (const auto &operator_ : topo_queues) {
    LOG(INFO) << "Index: " << index << " Type: " <<
operator_->type
              << " Name: " << operator_->name;
    index += 1;
  }
}
```

如果出现模型路径找不到的问题，可以按下图的方式进行检查：



因为上方代码中的`param_path`和`bin_path`是相对路径，所以我们程序在执行的时候需要设置`working directory`.

模型的Build(构建)

我们在上小节中看到了`model.build`的调用，所以我们将在这一节中对`build`函数做一个补充调用。

模型的状态

`RuntimeGraph`共有三个状态，表示**不同状态下的同一个模型**（待初始化、待构建和构建完成），分别如下：

```
  enum class GraphState {
    NeedInit = -2,
    NeedBuild = -1,
    Complete = 0,
  };
```

在`RuntimeGraph`类中有一个变量会记录此刻模型的状态：

```
  GraphState graph_state_ = GraphState::NeedInit;
```

三者的状态变换如下，**依次表示待初始化，待构建和模型构建完成。**

```
NeedInit --> NeedBuild --> Complete
```

在初始情况下模型的状态`graph_state_`为`NeedInit`，表示模型目前**待初始化**。因此我们不能在此刻直接调用`Build`函数中的功能，**而是需要在此之前先调用模型的`Init`函数**，这一过程在下方的代码中也有体现，在初始化函数(`Init`)调用成功后会将模型的状态调整为`NeedBuild`.

```
void RuntimeGraph::Build(const std::string& input_name,
                         const std::string& output_name) {

 if (graph_state_ == GraphState::Complete) {
    LOG(INFO) << "Model has been built already!";
    return;
  }

  if (graph_state_ == GraphState::NeedInit) {
    bool init_graph = Init();
    LOG_IF(FATAL, !init_graph) << "Init graph failed!";
  }

  CHECK(graph_state_ >= GraphState::NeedBuild)
```

以上构建(`Build`)函数中代码的目的是为了检查模型是否已经构建完成。也就是检查`graph_state_ == GraphState::Complete`. 如果是这样的话，表示模型已经构建完成，`Build`函数直接返回。如果模型此刻的状态是`NeedInit`, 那我们首先**需要先对这个模型进行初始化**（先调用`Init`函数），再进行构建。

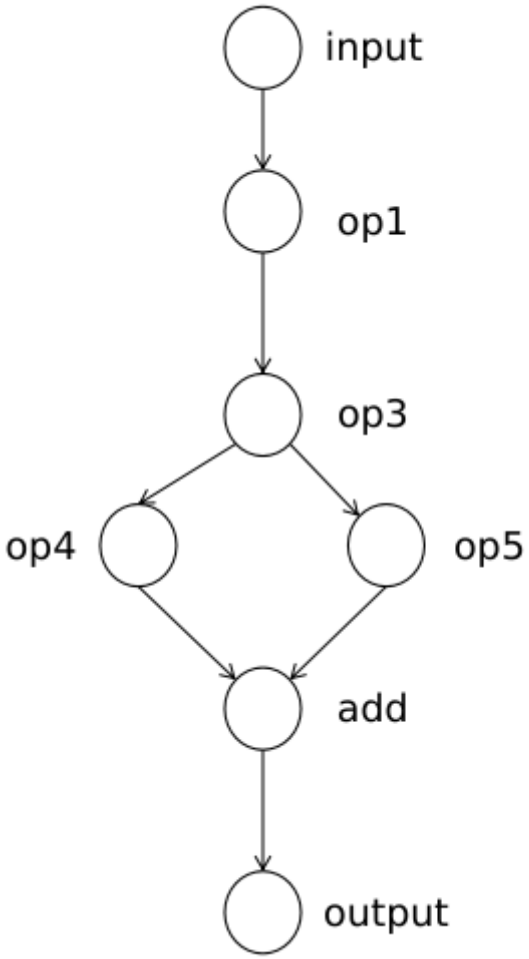
我们再来看一下在`Init`函数在本节课中新增加的状态转换，可以看到在`Init`函数的最后，我们将模型的状态从`NeedInit`调整到`NeedBuild`(需要被构建)，所以从`Init`函数返回之后，`Build`函数便可以继续执行其中的代码。

```
bool RuntimeGraph::Init() {
    ...
    ...
    graph_state_ = GraphState::NeedBuild;
    return true;
```

继续构建图关系

```
void RuntimeGraph::Build(const std::string& input_name,
                         const std::string& output_name){
 ...
 ...
 for (const auto& current_op : this->operators_) {
    const std::vector<std::string>& output_names =
current_op->output_names;
    for (const auto& kOutputName : output_names) {
      if (const auto& output_op = this-
>operators_maps_.find(kOutputName);
          output_op != this->operators_maps_.end()) {
        current_op->output_operators.insert({kOutputName,
output_op->second});
      }
    }
  }
```

我们继续往下看`Build`函数中的内容，这段代码主要用于**构建算子之间的联系**。例如对于以下的深度学习模型结构，我们需要使用最外层的`for`循环遍历模型中的每一个算子`current_op`.



随后我们程序会拿到每个算子中的`output_names`，例如对于`op3`来说，它的`output_names`包括了`op4`和`op5`.

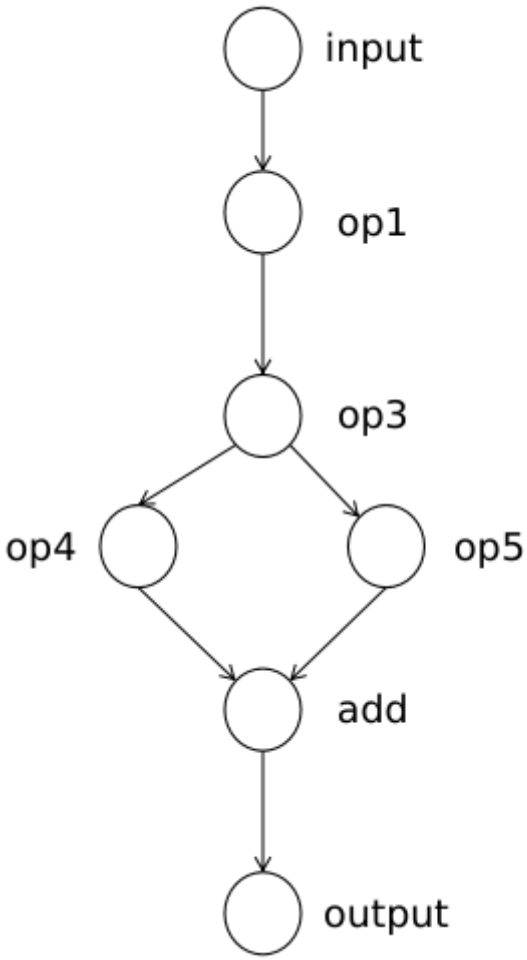
我们会对该算子(`op3`)的`output_names`进行遍历，并会根据`output_name`找到对应的后继算子并插入到`op3`的`output_operators`中。例如对于`op3`算子，我们会根据`output_names`中的`"op4"`和`"op5"`来寻找它的两个后继节点，寻找到后放入到当前计算节点`current_op`的后继节点列表`output_operators`中。

当最外层的循环结束时，`op1`中的`output_operators`中存放了**它的一个后继节点**`{"op3":op3}`,而`op3`中的`output_operators`会存放**它的两个后继节点**`{"op4":op4, "op5":op5}`. 以此类推。不难看出，`output_operators`中存放的是该节点所有后继算子节点的`name`到后继算子本身的映射。

单元测试

> 在这个单元测试中，我们会对以上构建图关系的代码进行调试。

我们先来看看针对如下图模型的拓扑排序序列计算。



在运行时同样需要注意相对路径的配置问题

```
TEST(test_ir, build_output_ops) {
  using namespace kuiper_infer;
  std::string
bin_path("course4/model_file/simple_ops.pnnx.bin");
  std::string
param_path("course4/model_file/simple_ops.pnnx.param")
  RuntimeGraph graph(param_path, bin_path);
  const bool init_success = graph.Init();
  ASSERT_EQ(init_success, true);
  graph.Build("pnnx_input_0", "pnnx_output_0");
  const auto &topo_queues = graph.get_topo_queues();
```

```
  int index = 0;
  for (const auto &operator_ : topo_queues) {
    LOG(INFO) << "Index: " << index << " Type: " <<
operator_->type
              << " Name: " << operator_->name;
    index += 1;
  }
}
```

拓扑排序序列的输出如下，不难看出程序拓扑排序的结果**完全符合模型结构图中的定义**，其中`pnnx_expr_0`是一个表达式层，相当于以上结构图中的`add`层。

```
I20230626 11:58:22.928081  1943 test_topo.cpp:40] Index: 0
Name: pnnx_input_0
I20230626 11:58:22.928135  1943 test_topo.cpp:40] Index: 1
Name: op1
I20230626 11:58:22.928176  1943 test_topo.cpp:40] Index: 2
Name: op3
I20230626 11:58:22.928211  1943 test_topo.cpp:40] Index: 3
Name: op5
I20230626 11:58:22.928251  1943 test_topo.cpp:40] Index: 4
Name: op4
I20230626 11:58:22.928287  1943 test_topo.cpp:40] Index: 5
Name: pnnx_expr_0
I20230626 11:58:22.928324  1943 test_topo.cpp:40] Index: 6
Name: pnnx_output_0
```

随后，我们再来看下对模型结构中所有算子的后继节点输出：

```
TEST(test_ir, build_output_ops2) {
  using namespace kuiper_infer;
  std::string
bin_path("course4/model_file/simple_ops.pnnx.bin");
  std::string
param_path("course4/model_file/simple_ops.pnnx.param");
  RuntimeGraph graph(param_path, bin_path);
  const bool init_success = graph.Init();
  ASSERT_EQ(init_success, true);
  graph.Build("pnnx_input_0", "pnnx_output_0");
```

```
  const auto &topo_queues = graph.get_topo_queues();

  int index = 0;
  for (const auto &operator_ : topo_queues) {
    LOG(INFO) << "operator name: " << operator_->name;
    for (const auto &pair : operator_->output_operators) {
      LOG(INFO) << "output: " << pair.first;
    }
    LOG(INFO) << "-------------------------";
    index += 1;
  }
}
```

```
I20230626 12:14:21.149619  2184 test_topo.cpp:57] operator
name: pnnx_input_0
I20230626 12:14:21.149658  2184 test_topo.cpp:59] output:
op1
I20230626 12:14:21.149684  2184 test_topo.cpp:61] --------
-----------------
I20230626 12:14:21.149709  2184 test_topo.cpp:57] operator
name: op1
I20230626 12:14:21.149735  2184 test_topo.cpp:59] output:
op3
I20230626 12:14:21.149760  2184 test_topo.cpp:61] --------
-----------------
I20230626 12:14:21.149785  2184 test_topo.cpp:57] operator
name: op3
I20230626 12:14:21.149811  2184 test_topo.cpp:59] output:
op4
I20230626 12:14:21.149835  2184 test_topo.cpp:59] output:
op5
I20230626 12:14:21.149863  2184 test_topo.cpp:61] --------
-----------------
I20230626 12:14:21.149888  2184 test_topo.cpp:57] operator
name: op5
I20230626 12:14:21.149914  2184 test_topo.cpp:59] output:
pnnx_expr_0
I20230626 12:14:21.149938  2184 test_topo.cpp:61] --------
-----------------
```

```
I20230626 12:14:21.149963  2184 test_topo.cpp:57] operator
name: op4
I20230626 12:14:21.149988  2184 test_topo.cpp:59] output:
pnnx_expr_0
I20230626 12:14:21.150014  2184 test_topo.cpp:61] --------
-----------------
I20230626 12:14:21.150038  2184 test_topo.cpp:57] operator
name: pnnx_expr_0
I20230626 12:14:21.150064  2184 test_topo.cpp:59] output:
pnnx_output_0
I20230626 12:14:21.150089  2184 test_topo.cpp:61] --------
-----------------
I20230626 12:14:21.150115  2184 test_topo.cpp:57] operator
name: pnnx_output_0
I20230626 12:14:21.150139  2184 test_topo.cpp:61] --------
-----------------
Process finished with exit code 0
```

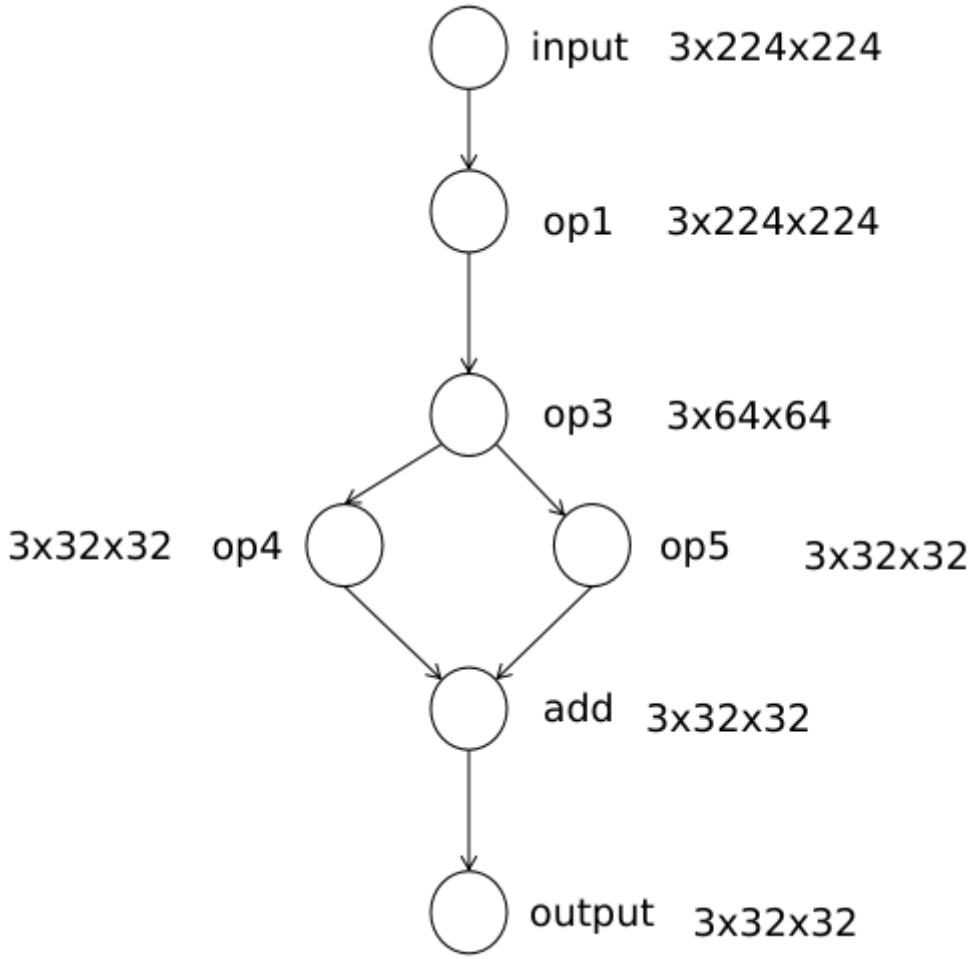
从输出中可以看出`input`节点仅有一个输出节点`op1`,`op1`也仅有一个输出节点`op3`,`op3`有两个输出节点分别为`op4`和`op5`.`add`计算节点(`pnnx_expr_0`)对应的输出节点`pnnx_output_0`也是全模型中的输出节点。

节点输出张量的初始化

我们在`Build`函数中还需要**完成计算节点中输出张量空间的初始化**，从下图中可以看出每个节点都有一个形状不同的输出张量，用来存放该节点的计算输出。那我们为什么要在构建(`Build`)阶段就初始化这些数据，而不是在算子计算的时候再对输出空间初始化呢？这是因为**申请较大字节数的内存空间也需要一定的时间，如果放在构建阶段进行提前申请可以在【运行时】省下这段时间。**

另外，我们需要对算子的**输出空间**做初始化和提前申请，那么我们需要对一个算子的**输入空间初始化**吗？**其实是不用的**，我们借用下方的这张图，**节点旁的维度表示该节点输出维度大小**。

在`op1`节点中我们需要申请`3 x 224 x 224`大小的输出空间，对于它的后继节点`op3`，我们需要申请`3 x 64 x 64`的内存作为输出空间，**而`op3`的输入空间和`op1`的输出空间大小相同，所以`op3`的输入空间可以复用前驱节点(`op3`)中的输出张量空间。**同理对于`add`节点，我们需要为它申请`3 x 32 x 32`大小的输出空间，**但是对于`add`节点的两个输入，我们可以复用前驱节点`op4`和`op5`中的输出。**



以上的张量空间预分配的叙述过程对应有以下的代码：

```
void RuntimeOperatorUtils::InitOperatorOutput(
    const std::vector<pnnx::Operator*>& pnnx_operators,
    const std::vector<std::shared_ptr<RuntimeOperator>>&
operators) {
  CHECK(!pnnx_operators.empty() && !operators.empty());
  CHECK(pnnx_operators.size() == operators.size());
```

我们为`RuntimeOperator`初始化空间时，还是需要用到`PNNX`中的`Operator`结构，所以我们首先**需要判断两个数组是否是等长的**，它们具有一一对应的关系。

```
 void RuntimeOperatorUtils::InitOperatorOutput(...){
 ...
 ...
 for (uint32_t i = 0; i < pnnx_operators.size(); ++i) {
    // 得到pnnx原有的输出空间
    const std::vector<pnnx::Operand*> operands =
pnnx_operators.at(i)->outputs;
    CHECK(operands.size() <= 1) << "Only support one node
one output yet!";
    if (operands.empty()) {
      continue;
    }
    CHECK(operands.size() == 1) << "Only support one
output in the KuiperInfer";

    pnnx::Operand* operand = operands.front();
    const auto& runtime_op = operators.at(i);
    CHECK(operand != nullptr) << "Operand output is null";
    const std::vector<int32_t>& operand_shapes = operand-
>shape;
    const auto& output_tensors = runtime_op-
>output_operands;

    CHECK(operand != nullptr) << "Operand output is null";
    const std::vector<int32_t>& operand_shapes = operand-
>shape;
```

在以上的代码中，我们首先使用`pnnx_operators.at(i)->outputs`获得第i个计算节点中的所有输出计算数`operand`，**我们需要根据这个`pnnx`计算数`Operand`中记录的`Shape`和`Type`信息来初始化我们`runtime_op`中输出数据存储的空间`output_tensors = runtime_op->output_operands.`**

换句话说，我们就是要根据`pnnx::operand`中记录的输出节点大小(`operand->shape`)来**初始化该计算节点的输出张量**`output_tensors = runtime_op->output_operands`也就是**计算节点中的具体存储空间**，以及输出张量中的其他变量。

```
    const int32_t batch = operand_shapes.at(0);
    CHECK(batch >= 0) << "Dynamic batch size is not
supported!";
    CHECK(operand_shapes.size() == 2 ||
operand_shapes.size() == 4 ||
          operand_shapes.size() == 3)
        << "Unsupported shape sizes: " <<
operand_shapes.size();
```

从上文知道，我们需要根据计算数的形状`operand_shapes = operand->shape`来初始化该计算节点的输出张量`output_tensors`中的存储空间，但是我们输出张量的维度只支持二维的、三维以及四维的，所以需要在以上代码上做`check`.

```
    if (!output_tensors) {
      // 需要被初始化的输出张量
      std::shared_ptr<RuntimeOperand> output_operand =
          std::make_shared<RuntimeOperand>();
      // 将输出操作数赋变量
      output_operand->shapes = operand_shapes;
      output_operand->type =
RuntimeDataType::kTypeFloat32;
      output_operand->name = operand->name + "_output";
```

如果输出张量`output_tensors`没有被初始化，我们首先需要根据`pnnx`中的信息来**初始化一个保存输出张量的结构(`output_operand`)**, 在这个结构中需要**保存输出张量相关的类型、名字以及维度信息**。该结构的具体定义，我们已经在第二节中讲述过了，不清楚的同学可以去翻翻那节课的视频和代码。

```
struct RuntimeOperand {
  std::string name;
/// 操作数的名称
  std::vector<int32_t> shapes;
 /// 操作数的形状
  std::vector<std::shared_ptr<Tensor<float>>> datas;
 /// 存储操作数
  RuntimeDataType type = RuntimeDataType::kTypeUnknown;
/// 操作数的类型
};
```

在上面的代码中，我们已经初始化了`RuntimeOperand`结构中的名字、类型和维度等信息。下面我们要去初始化结构中**存放输出数据的`datas`变量**，它是一个张量的数组类型，数组的长度等于该计算节点的`batch_size`大小。

```
      for (int j = 0; j < batch; ++j) {
        if (operand_shapes.size() == 4) {
          sftensor output_tensor = TensorCreate(
              operand_shapes.at(1), operand_shapes.at(2),
operand_shapes.at(3));
          output_operand->datas.push_back(output_tensor);
        } else if (operand_shapes.size() == 2) {
          sftensor output_tensor = TensorCreate(
              std::vector<uint32_t>
{(uint32_t)operand_shapes.at(1)});
          output_operand->datas.push_back(output_tensor);
        } else {
          // current shape is 3
          sftensor output_tensor =
TensorCreate(std::vector<uint32_t>{
              (uint32_t)operand_shapes.at(1),
(uint32_t)operand_shapes.at(2)});
          output_operand->datas.push_back(output_tensor);
        }
      }
      runtime_op->output_operands =
std::move(output_operand);
```

对于一个计算算子`runtime_op`来说，它的输出张量数组的长度等于`batch_size`个，所以我们在循环中需要对`batch_size`个**输出张量进行创建（创建的时候需要依据`operand_shapes`）**。

在创建完成后还需要放入到`output_operand`的`datas`变量中。在循环后结束后，**我们会将初始化好的`output_operands`绑定到对应的计算节点中用于保存计算节点的输出数据**。至此，我们完成了计算节点中输出张量的初始化。

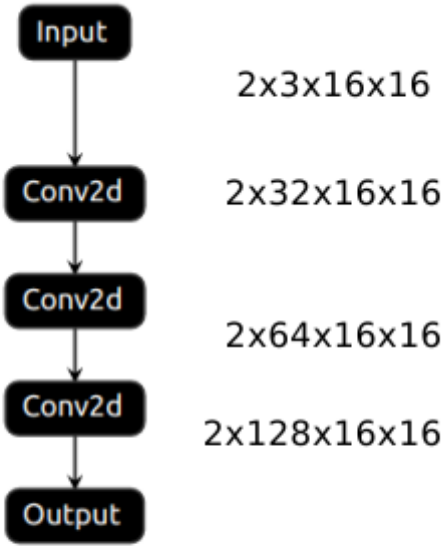
单元测试

```
  enum class GraphState {
    NeedInit = -2,
    NeedBuild = -1,
    Complete = 0,
  };
```

以下的单元测试检查各时刻模型的状态，在`graph.init()`之间，模型的状态是`NeedInit(-2)`. 模型在`init`之后和`build`之前，模型的状态是`NeedBuild(-1)`. 在成功`build`之后，模型的状态是`Complete = 0`.

```
TEST(test_ir, build1_status) {
  using namespace kuiper_infer;
  std::string
bin_path("course4/model_file/simple_ops.pnnx.bin");
  std::string
param_path("course4/model_file/simple_ops.pnnx.param");
  RuntimeGraph graph(param_path, bin_path);
  ASSERT_EQ(int(graph.graph_state()), -2);
  const bool init_success = graph.Init();
  ASSERT_EQ(init_success, true);
  ASSERT_EQ(int(graph.graph_state()), -1);
  graph.Build("pnnx_input_0", "pnnx_output_0");
  ASSERT_EQ(int(graph.graph_state()), 0);
}
```

我们已知该模型的各个节点中，有如下的输出张量维度大小：



节点的输出大小依次为`2x3x16x16`, `2x32x16x16`. 其中第一维表示输出的`batch size`, 第二维表示输出的`channel`, 第三维和第四维依次是输出的`height`和`width`. 我们在以下的单元测试中会输出某个节点的输出张量大小、维度是否符合上图。

```
TEST(test_ir, build1_output_tensors) {
  using namespace kuiper_infer;
  std::string
bin_path("course4/model_file/simple_ops2.pnnx.bin");
  std::string
param_path("course4/model_file/simple_ops2.pnnx.param");
  RuntimeGraph graph(param_path, bin_path);
  ASSERT_EQ(int(graph.graph_state()), -2);
  const bool init_success = graph.Init();
  ASSERT_EQ(init_success, true);
  ASSERT_EQ(int(graph.graph_state()), -1);
  graph.Build("pnnx_input_0", "pnnx_output_0");
  ASSERT_EQ(int(graph.graph_state()), 0);

  const auto &ops = graph.operators();
  for (const auto &op : ops) {
    LOG(INFO) << op->name;
    // 打印op输出空间的张量
```

```
    const auto &operand = op->output_operands;
    if (operand->datas.empty()) {
      continue;
    }
    const uint32_t batch_size = operand->datas.size();
    LOG(INFO) << "batch: " << batch_size;

    for (uint32_t i = 0; i < batch_size; ++i) {
      const auto &data = operand->datas.at(i);
      LOG(INFO) << "channel: " << data->channels()
                << " height: " << data->rows() << " cols:
" << data->cols();
    }
  }
}
```

课堂作业

1. 调试本节课所有的单元测试，观察各个时刻下变量的状态变化；
2. 用另一种思路或方法来实现拓扑排序，并写出相关的代码。