第三课-计算图的定义

> 赞助方： datawhale
>
> 作者：傅莘莘、散步、陈前
>
> 本章的代码：https://github.com/zjhellofss/kuiperdatawhale.git

计算图的概念

`KuiperInfer`使用的模型格式是`PNNX`. `PNNX`是`PyTorch Neural Network Exchange`的缩写，其愿景是能够将`PyTorch`模型文件直接导出为高效、简洁的计算图。计算图包括了以下几个部分，作为一种计算图形式，`PNNX`自然也不例外，正如第一章所述：

1. `Operator`: 深度学习计算图中的计算节点。
2. `Graph`: 有多个`Operator`串联得到的有向无环图，规定了各个计算节点（`Operator`）执行的流程和顺序。

3. `Layer`: **计算节点中**运算的具体执行者，`Layer`类先读取输入张量中的数据，然后对输入张量进行计算，得到的结果存放到计算节点的输出张量中，**当然，不同的算子中`Layer`的计算过程会不一致**。
4. `Tensor`: 用于存放**多维数据**的数据结构，方便数据在计算节点之间传递，同时该结构也封装矩阵乘、点积等与矩阵相关的基本操作。

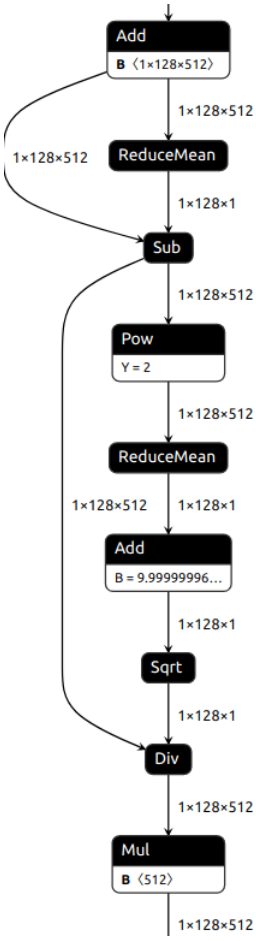
如果你不记得上述内容了，可以回顾一下本课程第一章。接下来，让我们探讨一下为什么在出现了`ONNX`计算图之后还需要`PNNX`，以及它解决了什么问题。

PNNX计算图的优势

> 参考资料：https://zhuanlan.zhihu.com/p/427620428

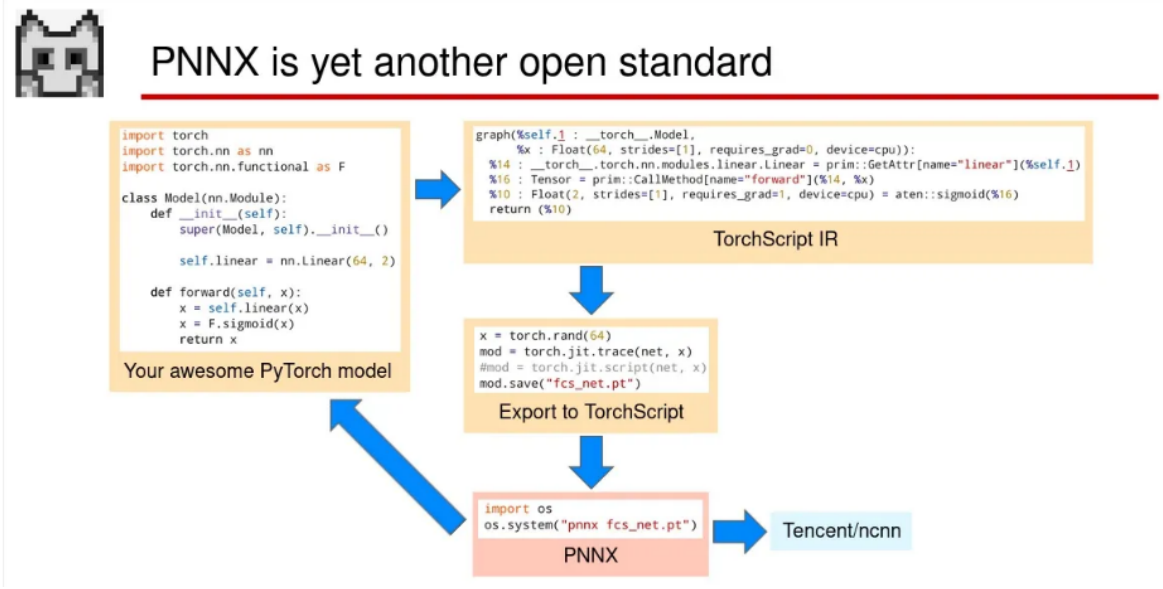
以往我们将训练好的模型导出为`ONNX`结构之后，模型中的一个复杂算子不仅经常会被拆分成多个细碎的算子，而且为了将这些细碎的算子拼接起来完成原有算子的功能，通常还需要一些称之为“胶水算子”的辅助算子，例如`Gather`和`Unsqueeze`等。

你还认得出下图的算子，其实是被拆分后`LayerNorm`算子吗？当然`ONNX`使用多个小算子去组合、等价一个复杂算子的设计，也是为了用尽可能少的算子去兼容更多的训练框架。



但是过于细碎的计算图会不仅不利于推理的优化。另外，拆分的层次过于细致，也会导致算法工程师难以将导出的模型和原始模型进行结构上的相互对应。为了解决以上说到的问题，我们选用`NCNN`推理框架的计算图格式之一`PNNX`，那么`PNNX`给我们带来了什么呢?

下图是它在模型部署中的位置，可以看到，一个模型文件从`PyTorch`先经历了`TorchScript`的导出，随后再经过`PNNX`的优化得到最终的模型文件（无视最后导出为`NCNN`的部分）。



> 图片来源：https://zhuanlan.zhihu.com/p/427620428

1. 使用模板匹配（`pattern matching`）的方法将匹配到的子图用对应等价的大算子替换掉，例如可以将上图子图中的多个小算子（可能是在`TorchScript`中被拆分的）重新替换为`LayerNorm`算子。或者在对`PyTorch`模型导出时，也可以自定义某个`nn.Module`不被拆分；
2. 在`PyTorch`中编写的简单算术表达式在转换为`PNNX`后，会保留表达式的整体结构，而不会被拆分成许多小的加减乘除算子。例如表达式`add(mul(@0, @1),add(@2, @3))`不会被拆分为两个`add`算子和一个`mul`算子，而是会生成一个表达式算子`Expression`；
3. `PNNX`项目中有大量图优化的技术，包括了算子融合，常量折叠和消除，公共表达式消除等技术。

- 算子融合优化是一种针对深度学习神经网络的优化策略，通过将多个相邻的计算算子合并为一个算子来减少计算量和内存占用。以卷积层和批归一化层为例，我们可以把两个算子合并为一个新的算子，也就是将卷积的公式带入到批归一化层的计算公式中：

$$Conv = w * x_1 + b$$

$$BN = \gamma \frac{x_2 - \hat{u}}{\sigma^2 + \epsilon} + \beta$$

其中x_1和x_2依次是卷积和批归一化层的输入，w是卷积层的权重，b是卷积层的偏移量，$\hat{u}$和σ依次是样本的均值和方差，ϵ为一个极小值。带入后有：

$$Fused = \gamma \frac{(w * x + b) - \hat{u}}{\sigma^2 + \epsilon} + \beta$$

- 常量折叠是将**在编译时期间将表达式中的常量计算出来，然后将结果替换为一个等价的常量**，以减少模型在运行时的计算量。
- 常量移除就是将计算图中不需要的常数（**计算图推理的过程中未使用**）节点删除，从而减少计算图的文件和加载后的资源占用大小。
- 公共表达式消除优化是一种针对计算图中重复计算的优化策略，**它可以通过寻找并合并重复计算的计算节点，减少模型的计算量和内存占用**。

公共子表达式检测是指**查找计算图中相同的子表达式**，公共子表达式消除是指**将这些重复计算的计算节点合并为一个新的计算节点**，从而减少计算和内存开销。举个例子：

```
X = input(3,224,224)
A = Conv(X)
B = Conv(X)
C = A + B
```

在上方的代码中，`Conv(X)`这个结果被计算了两次，公共子表达式消除可以将它优化为如下代码，这样一来就少了一次卷积的计算过程。

```
X = input(3, 224,224)
T = Conv(X)
C = T + T
```

综上所述，如果在我们推理框架的底层用`PNNX`计算图，就可以吸收图优化和算子融合的结果，使得推理速度更快更高效。

PNNX计算图的格式

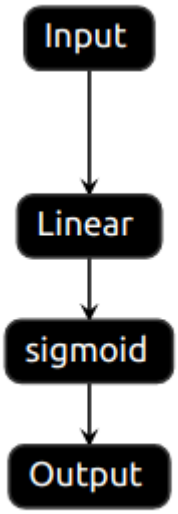
`PNNX`由图结构(`Graph`)，运算符(`Operator`)和操作数(`Operand`)这三种结构组成的，设计非常简洁。

用于测试的PyTorch模型

为了帮助同学们更好地掌握计算图格式，我们准备了一对较小的模型文件`linear.param`和`linear.bin`来做单步调试，它们分别是网络的**结构定义**和**权重文件**。该模型在`PyTorch`中的定义如下：

```
class Model(nn.Module):
    def __init__(self):
        super(Model, self).__init__()
        self.linear = nn.Linear(32, 128)

    def forward(self, x):
        x = self.linear(x)
        x = F.sigmoid(x)
        return x
```

这是一个非常简单的模型，其作用是对输入`x`进行线性映射（从32维到128维），并对输出进行`sigmoid`计算，从而得到最终的计算结果。该模型的网络结构如下图所示（使用`Netron`软件打开`linear.param`），我们以其中的`Linear`层为例：



- `Linear`层有`#0`和`#1`两个操作数，分别为输入和输出张量，形状依次为`(1, 128)`和`(1, 32)`；
- `Linear`层有两个属性参数：`@weight`和`@bias`，用于存储该层的权重数据信息，分别对应权重（即`weight`）和偏置（即`bias`）。可以看到这两个权重的形状分别为`(1, 128)`和`(1, 32)`，在后续过程中可以根据需要进行权重加载。
- `Linear`层有三个属性：`bias`，`in_features`和`out_features`，分别表示是否使用偏置项、线性连接层的输入维度和输出维度。

NODE PROPERTIES

type	nn.Linear
name	linear

ATTRIBUTES

#0	(1,32)f32
#1	(1,128)f32
@bias	(128)f32
@weight	(128,32)f32
bias	True
in_features	32
out_features	128

INPUTS

input	name: **0**

OUTPUTS

output	name: **1**

PNNX中的图结构(Graph)

```
class Graph
{
    Operator* new_operator(const
std::string& type, const std::string& name);
    Operator* new_operator_before(const
std::string& type, const std::string& name,
const Operator* cur);

    Operand* new_operand(const
torch::jit::Value* v);
    Operand* new_operand(const std::string&
name);
    Operand* get_operand(const std::string&
name);

    std::vector<Operator*> ops;
    std::vector<Operand*> operands;
};
```

`Graph`的核心作用是**管理计算图中的运算符和操作数**。下面我们将对这两个概念进行说明:

1. `Operator`类用来**表示计算图中的运算符（算子）**，比如一个模型中的`Convolution`,`Pooling`等算子;
2. `Operand`类用来**表示计算图中的操作数**，即**与一个运算符有关的输入和输出张量**;

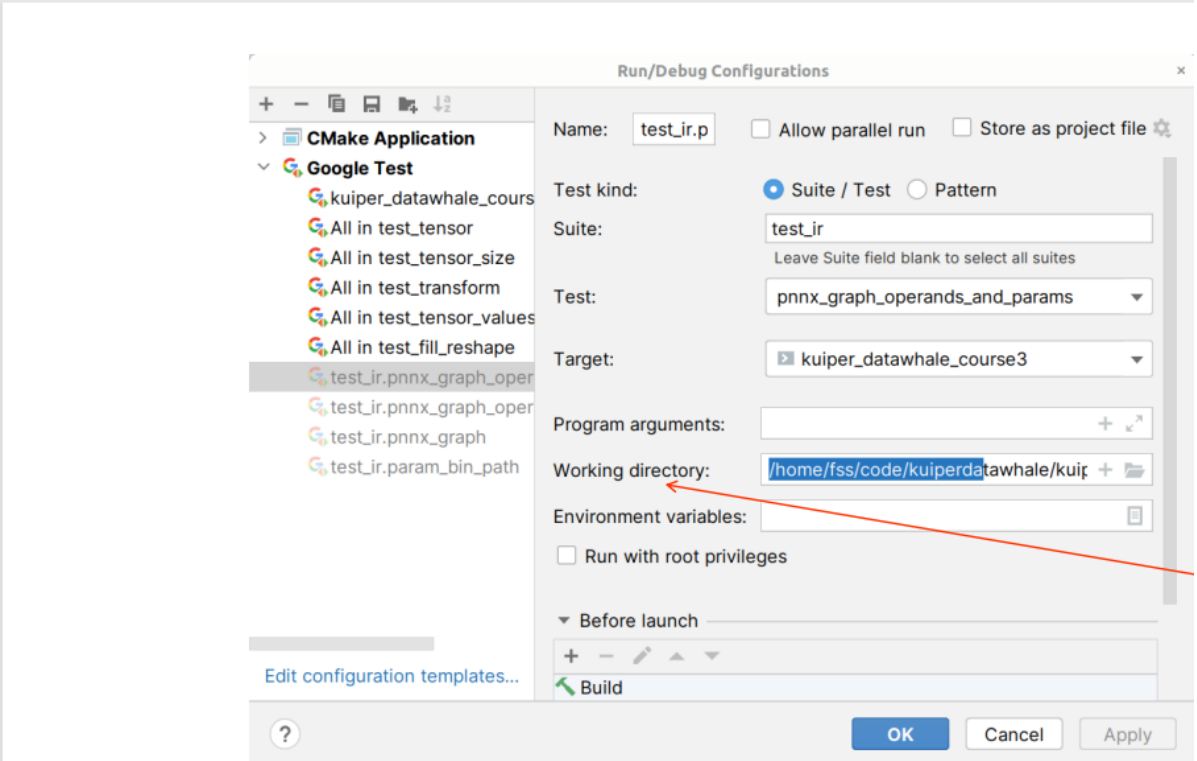
3. `Graph`类的成员函数提供了方便的接口用来**创建和访问操作符和操作数**，以构建和遍历计算图。同时，它也是模型中**运算符（算子）和操作数的集合**。

PNNX中图结构相关的单元测试

```
TEST(test_ir, pnnx_graph_ops) {
  using namespace kuiper_infer;
  /**
   * 如果这里加载失败，请首先考虑相对路径的正确性问题
   */
  std::string bin_path("course3/model_file/test_linear.pnnx.bin");
  std::string param_path("course3/model_file/test_linear.pnnx.param");
  std::unique_ptr<pnnx::Graph> graph = std::make_unique<pnnx::Graph>();
  int load_result = graph->load(param_path, bin_path);
  // 如果这里加载失败，请首先考虑相对路径(bin_path和param_path)的正确性问题
  ASSERT_EQ(load_result, 0);
  const auto &ops = graph->ops;
  for (int i = 0; i < ops.size(); ++i) {
    LOG(INFO) << ops.at(i)->name;
  }
```

```
}
```

以上代码使用`pnnx::Graph`类的`load`函数传递相应参数来加载模型，并使用单元测试来检查模型是否被成功加载。请注意，如果出现`open failed`的问题，可以在`Clion`中设置程序工作的目录。



随后我们在单元测试的`for`循环中打印相关的运算符名字(`name`)，得到了与模型结构图上一致的输出结果。可以看到，这里输出的运算符和模型结构图上的运算符是一致的。

```
I20230606 13:52:36.940599  3651
test_ir.cpp:23] pnnx_input_0
I20230606 13:52:36.940616  3651
test_ir.cpp:23] linear
I20230606 13:52:36.940624  3651
test_ir.cpp:23] F.sigmoid_0
I20230606 13:52:36.940632  3651
test_ir.cpp:23] pnnx_output_0
```

在以下的代码中，我们**除了输出运算符，还输出了运算符相关的输入输出张量，包括张量相关的名字(name)，形状(shape)等属性**，可以看到输出的结果和`Netron`中的可视化模型结构图也保持了一致。

```
TEST(test_ir, pnnx_graph_operands) {
    ...
    ...
    for (int j = 0; j < op->inputs.size();
++j) {
      LOG(INFO) << "Input name: " << op-
>inputs.at(j)->name
                << " shape: " <<
ShapeStr(op->inputs.at(j)->shape);
    }

    LOG(INFO) << "OP Output";
    for (int j = 0; j < op->outputs.size();
++j) {
```

```
      LOG(INFO) << "Output name: " << op-
>outputs.at(j)->name
                << " shape: " <<
ShapeStr(op->outputs.at(j)->shape);
    ...
    ...
}
```

输出结果:

```
...
...
I20230606 14:05:59.942880  3897
test_ir.cpp:52] OP Name: linear
I20230606 14:05:59.942904  3897
test_ir.cpp:53] OP Inputs
I20230606 14:05:59.942926  3897
test_ir.cpp:55] Input name: 0 shape: 1 x 32
I20230606 14:05:59.942950  3897
test_ir.cpp:59] OP Output
I20230606 14:05:59.942974  3897
test_ir.cpp:61] Output name: 1 shape: 1 x
128
...
...
```

PNNX中的运算符结构(Operator)

有了上面的直观认识，我们来聊聊`PNNX`中的运算符结构。

```
class Operator
{
public:
    std::vector<Operand*> inputs;
    std::vector<Operand*> outputs;

    std::string type;
    std::string name;

    std::vector<std::string> inputnames;
    std::map<std::string, Parameter> params;
    std::map<std::string, Attribute> attrs;
};
```

在PNNX中，`Operator`用来表示一个算子，它由以下几个部分组成：

1. `inputs`：类型为`std::vector<operand>`，表示这个算子在计算过程中所需要的**输入操作数**`operand`；
2. `outputs`：类型为`std::vector<operand>`，表示这个算子在计算过程中得到的**输出操作数**`operand`；
3. `type`和`name`类型均为`std::string`，分别表示**该运算符号的类型和名称**；
4. `params`，类型为`std::map`，用于存放**该运算符的所有参数**（例如卷积运算符中的`params`中将存放`stride`，`padding`，`kernel size`等信息）；

5. `attrs`, 类型为`std::map`, 用于存放**该运算符所需要的具体权重属性**（例如卷积运算符中的`attrs`中就存放着卷积的权重和偏移量，通常是一个`float32`数组）。

PNNX中运算符结构相关的单元测试

```
TEST(test_ir,
pnnx_graph_operands_and_params) {
    ...
    ...

    LOG(INFO) << "Params";
    for (const auto &attr : op->params) {
      LOG(INFO) << attr.first << " type " <<
attr.second.type;
    }

    LOG(INFO) << "Weight: ";
    for (const auto &weight : op->attrs) {
      LOG(INFO) << weight.first << " : " <<
ShapeStr(weight.second.shape)
                << " type " <<
weight.second.type;
    }
    LOG(INFO) << "-------------------------
--------------------";
  }
}
```

以上代码使用`pnnx::Graph`类的`load`函数传递相应参数来加载模型，并使用单元测试来检查模型是否被成功加载。

随后我们在`for`循环中打印`Linear`运算符，**得到了与可视化模型结构图上一致的权重(`Weight`)信息和参数信息(`bias`, `in_features`, `out_features`)**.

```
...
...
I20230607 12:16:47.349155  1445
test_ir.cpp:86] OP Name: linear
I20230607 12:16:47.349190  1445
test_ir.cpp:87] OP Inputs
I20230607 12:16:47.349215  1445
test_ir.cpp:89] Input name: 0 shape: 1 x 32
I20230607 12:16:47.349241  1445
test_ir.cpp:93] OP Output
I20230607 12:16:47.349265  1445
test_ir.cpp:95] Output name: 1 shape: 1 x
128

I20230607 12:16:47.349290  1445
test_ir.cpp:99] Params
I20230607 12:16:47.349314  1445
test_ir.cpp:101] bias type 1
I20230607 12:16:47.349339  1445
test_ir.cpp:101] in_features type 2
```

```
I20230607 12:16:47.349364  1445
test_ir.cpp:101] out_features type 2
I20230607 12:19:01.993947  1550
test_ir.cpp:104] Weight:
I20230607 12:19:01.993985  1550
test_ir.cpp:106] bias : 128 type 1
I20230607 12:19:01.994024  1550
test_ir.cpp:106] weight : 128 x 32 type 1
...
...
```

PNNX中的Attribute和Param结构

在PNNX中，**权重数据结构(Attribute)和参数数据结构(Param)**定义如下。它们通常与一个运算符相关联，例如`Linear`算子的`in_features`属性和`weight`权重。

```
class Parameter
{
    // 0=null 1=b 2=i 3=f 4=s 5=ai 6=af 7=as
8=others
    int type;
    ...
    ...
}
```

```
class Attribute
{
public:
    Attribute()
```

```
        : type(0)
    {
    }

    Attribute(const
std::initializer_list<int>& shape, const
std::vector<float>& t);

    // 0=null 1=f32 2=f64 3=f16 4=i32 5=i64
6=i16 7=i8 8=u8 9=bool
    int type;
    std::vector<int> shape;
    ...
};
```

PNNX中的操作数结构(Operand)

```
class Operand
{
public:
    void remove_consumer(const Operator* c);
    Operator* producer;
    std::vector<Operator*> consumers;

    int type;
    std::vector<int> shape;

    std::string name;
    std::map<std::string, Parameter> params;
};
```

重点值得分析的是操作数结构中的`producer`和`customers`，分别表示**产生这个操作数的算子**和**使用这个操作数的算子**。

值得注意的是产生这个操作数的算子只能有一个，而使用这个操作数的算子可以有很多个。

PNNX中操作数结构相关的单元测试

```
TEST(test_ir,
pnnx_graph_operands_customer_producer) {
  ...
  ...
  const auto &operands = graph->operands;
  for (int i = 0; i < operands.size(); ++i)
{
    const auto &operand = operands.at(i);
    LOG(INFO) << "Operand Name: #" <<
operand->name;
    LOG(INFO) << "Customers: ";
    for (const auto &customer : operand-
>consumers) {
      LOG(INFO) << customer->name;
    }

    LOG(INFO) << "Producer: " << operand-
>producer->name;
  }
}
```

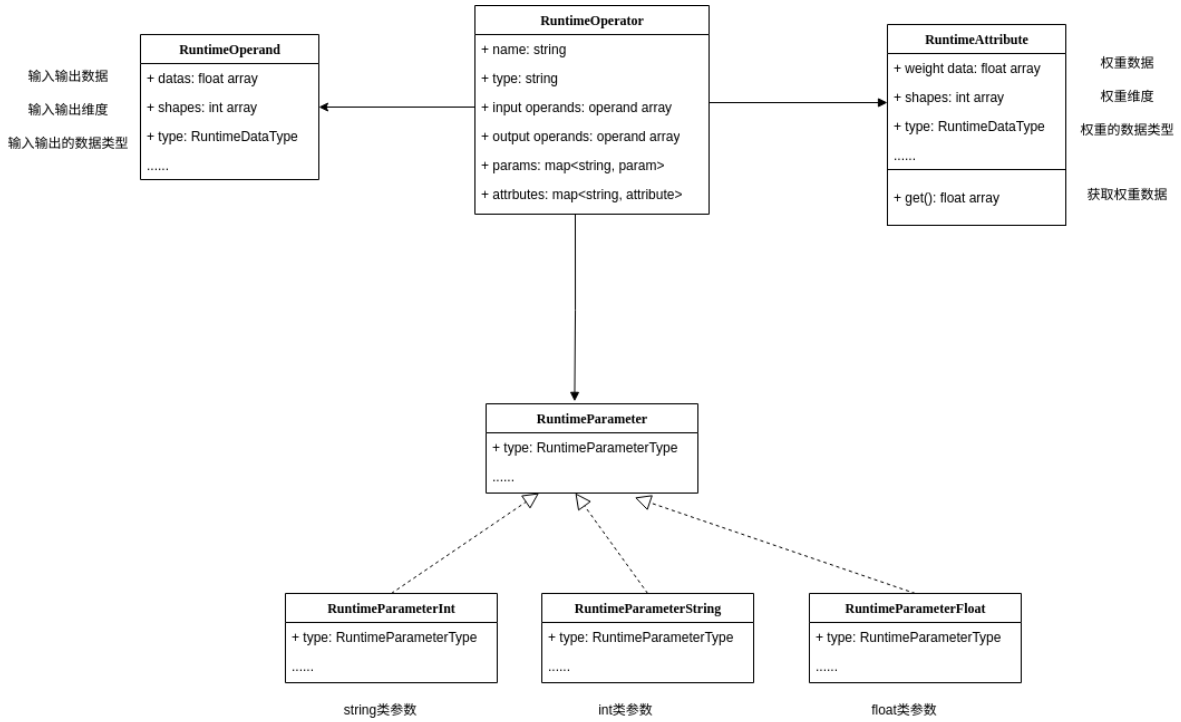
运行后得到以下的输出，以`#1`为例，它是`Linear`层的输出，同时它也是`F.Sigmoid0`层的输入。

```
I20230607 02:48:26.558606  2075
test_ir.cpp:126] Operand Name: #0
I20230607 02:48:26.558667  2075
test_ir.cpp:127] Customers: linear
I20230607 02:48:26.558724  2075
test_ir.cpp:132] Producer: pnnx_input_0

I20230607 02:48:26.558749  2075
test_ir.cpp:126] Operand Name: #1
I20230607 02:48:26.558773  2075
test_ir.cpp:127] Customers: F.sigmoid_0
 I20230607 02:48:26.558821  2075
test_ir.cpp:132] Producer: linear

I20230607 02:48:26.558846  2075
test_ir.cpp:126] Operand Name: #2
I20230607 02:48:26.558869  2075
test_ir.cpp:127] Customers: pnnx_output_0
I20230607 02:48:26.558926  2075
test_ir.cpp:132] Producer: F.sigmoid_0
```

KuiperInfer对计算图的封装

> 为了更好的使用底层PNNX计算图，我们会在项目中对它进行再次封装，使得PNNX更符合我们的使用需求。

UML整体结构图



对Operator的封装

不难从上图看出，`RuntimeOperator`是`KuiperInfer`计算图中的核心数据结构，是对`PNNX::Operator`的再次封装，它有如下的定义：

```
struct RuntimeOperator {
  virtual ~RuntimeOperator();
```

```
  bool has_forward = false;
  std::string name;       /// 计算节点的名称
  std::string type;       /// 计算节点的类型
  std::shared_ptr<Layer> layer;  /// 节点对应的计算Layer

  std::vector<std::string> output_names;  /// 节点的输出节点名称
  std::shared_ptr<RuntimeOperand> output_operands;  /// 节点的输出操作数

  std::map<std::string, std::shared_ptr<RuntimeOperand>>
      input_operands;  /// 节点的输入操作数

  std::vector<std::shared_ptr<RuntimeOperand>>
      input_operands_seq;  /// 节点的输入操作数，顺序排列
  std::map<std::string, std::shared_ptr<RuntimeOperator>>
      output_operators;  /// 输出节点的名字和节点对应

  std::map<std::string, RuntimeParameter*> params;  /// 算子的参数信息
  std::map<std::string, std::shared_ptr<RuntimeAttribute>>
```

```
        attribute;  /// 算子的属性信息，内含权重信息
};
```

以上这段代码定义了一个名为`RuntimeOperator`的结构体。结构体包含以下成员变量：

1. `name`：**运算符节点的名称**，可以用来区分一个唯一节点，例如`Conv_1`，`Conv_2`等；
2. `type`：**运算符节点的类型**，例如`Convolution`，`Relu`等类型；
3. `layer`：**负责完成具体计算的组件**，例如在`Convolution Operator`中，`layer`对输入进行卷积计算，即计算其相应的卷积值；
4. `input_operands`和`output_operands`分别表示**该运算符的输入和输出操作数**。

   如果一个运算符(`RuntimeOperator`)的输入大小为`(4, 3, 224, 224)`，那么在`input_operands`变量中，`datas`数组的长度为 4，数组中每个元素的张量大小为`(3, 224, 224)`；
5. `params`是运算符(`RuntimeOperator`)的**参数信息**，包括卷积层的卷积核大小、步长等信息；
6. `attribute`是运算符(`RuntimeOperator`)的**权重、偏移量信息**，例如`Matmul`层或`Convolution`层需要的权重数据；

7. 其他变量的含义可参考注释。

从Operator到Kuiper::RuntimeOperator

在这个过程中，需要先从 `PNNX::Operator` 中提取数据信息（包括我们上文提到的 `Operand` 和 `Operator` 结构），并依次填入到 `KuiperInfer` 对应的数据结构中。

相应的代码如下所示，由于篇幅原因，在课件中省略了一部分内容，完整的代码可以在配套的 `course3` 文件夹中查看。

```
bool RuntimeGraph::Init() {
  if (this->bin_path_.empty() || this-
>param_path_.empty()) {
    LOG(ERROR) << "The bin path or param
path is empty";
    return false;
  }

  this->graph_ =
std::make_unique<pnnx::Graph>();
  int load_result = this->graph_-
>load(param_path_, bin_path_);
  if (load_result != 0) {
    LOG(ERROR) << "Can not find the param
path or bin path: " << param_path_
               << " " << bin_path_;
    return false;
  }
```

```
  std::vector<pnnx::Operator *> operators =
this->graph_->ops;
  for (const pnnx::Operator *op : operators)
{
      std::shared_ptr<RuntimeOperator>
runtime_operator =
          std::make_shared<RuntimeOperator>
();
      // 初始化算子的名称
      runtime_operator->name = op->name;
      runtime_operator->type = op->type;

      // 初始化算子中的input
      const std::vector<pnnx::Operand *>
&inputs = op->inputs;
      InitGraphOperatorsInput(inputs,
runtime_operator);

      // 记录输出operand中的名称
      const std::vector<pnnx::Operand *>
&outputs = op->outputs;
      InitGraphOperatorsOutput(outputs,
runtime_operator);

      // 初始化算子中的attribute(权重)
      const std::map<std::string,
pnnx::Attribute> &attrs = op->attrs;
      InitGraphAttrs(attrs,
runtime_operator);
```

```
      // 初始化算子中的parameter
      const std::map<std::string,
pnnx::Parameter> &params = op->params;
      InitGraphParams(params,
runtime_operator);
      this-
>operators_.push_back(runtime_operator);
      this-
>operators_maps_.insert({runtime_operator-
>name, runtime_operator});
  }
  return true;
}
```

和上文中的单元测试相同，需要先打开一个 `PNNX` 模型文件，并在返回错误时记录日志并退出。

```
  this->graph_ =
std::make_unique<pnnx::Graph>();
  int load_result = this->graph_-
>load(param_path_, bin_path_);
  if (load_result != 0) {
    LOG(ERROR) << "Can not find the param
path or bin path: " << param_path_
               << " " << bin_path_;
    return false;
  }
```

在`for`循环中**依次对每个运算符进行处理**：

```
for (const pnnx::Operator *op : operators)
```

提取`PNNX`运算符中的名字(`name`)和类型(`type`).

```
runtime_operator->name = op->name;
runtime_operator->type = op->type;
```

提取PNNX中的操作数Operand到RuntimeOperand

此处的过程对应于以上代码中的`InitGraphOperatorsInput`和`InitGraphOperatorsOutput`函数。

```
for (const pnnx::Operator *op : operators){
    inputs = op->inputs;
    InitGraphOperatorsInput(inputs,
runtime_operator);
    ...
```

```
void RuntimeGraph::InitGraphOperatorsInput(
    const std::vector<pnnx::Operand *>
&inputs,
    const std::shared_ptr<RuntimeOperator>
&runtime_operator) {

  // 遍历所有的输入张量
  for (const pnnx::Operand *input : inputs)
{
    if (!input) {
```

```
      continue;
    }
    const pnnx::Operator *producer = input-
>producer;
    std::shared_ptr<RuntimeOperand>
runtime_operand =
        std::make_shared<RuntimeOperand>();
    // 搬运name和shape
    runtime_operand->name = producer->name;
    runtime_operand->shapes = input->shape;

    switch (input->type) {
    case 1: {
      // 搬运类型
      runtime_operand->type =
RuntimeDataType::kTypeFloat32;
      break;
    }
    case 0: {
      runtime_operand->type =
RuntimeDataType::kTypeUnknown;
      break;
    }
    default: {
      LOG(FATAL) << "Unknown input operand
type: " << input->type;
    }
    }
```

```
    runtime_operator-
>input_operands.insert({producer->name,
runtime_operand});
    runtime_operator-
>input_operands_seq.push_back(runtime_operan
d);
  }
}
```

这段代码的两个参数分别是来自 `PNNX` 中的一个运算符的所有输入操作数（`Operand`）和待初始化的 `RuntimeOperator`。在以下的循环中：

```
  for (const pnnx::Operand *input : inputs)
```

我们需要依次将每个 `Operand` 中的**数据信息搬运到新初始化的 `RuntimeOperand` 中**，包括 `type`，`name`，`shapes` 等信息，并记录输出这个操作数(`Operand`)的运算符(`producer`)。

搬运完成后，再将数据完备的 `RuntimeOperand` 插入到待初始化的 `RuntimeOperator` 中。

```
const std::vector<pnnx::Operand*>& outputs =
op->outputs;
InitGraphOperatorsOutput(outputs,
runtime_operator);
```

```
void RuntimeGraph::InitGraphOperatorsOutput(
    const std::vector<pnnx::Operand *>
&outputs,
    const std::shared_ptr<RuntimeOperator>
&runtime_operator) {
  for (const pnnx::Operand *output :
outputs) {
    if (!output) {
      continue;
    }
    const auto &consumers = output-
>consumers;
    for (const auto &c : consumers) {
      runtime_operator-
>output_names.push_back(c->name);
    }
  }
}
```

这段代码的两个参数分别是来自 `PNNX` 中的**一个运算符的所有输出操作数**（`Operand`）和待初始化的 `RuntimeOperator`.

在这里，我们只需要记录操作数的消费者的名字（`customer.name`）即可。在之后的课程中，我们才会对 `RuntimeOperator` 中的输出操作数（`RuntimeOperand`）进行构建，到时再讲。

提取PNNX中的权重(Attribute)到RuntimeAttribute

```
const std::map<std::string,
pnnx::Attribute>& attrs = op->attrs;
InitGraphAttrs(attrs, runtime_operator);
```

```
void RuntimeGraph::InitGraphAttrs(
    const std::map<std::string,
pnnx::Attribute>& attrs,
    const std::shared_ptr<RuntimeOperator>&
runtime_operator) {
  for (const auto& [name, attr] : attrs) {
    switch (attr.type) {
      case 1: {
        std::shared_ptr<RuntimeAttribute>
runtime_attribute =

 std::make_shared<RuntimeAttribute>();
        runtime_attribute->type =
RuntimeDataType::kTypeFloat32;
        runtime_attribute->weight_data =
attr.data;
        runtime_attribute->shape =
attr.shape;
        runtime_operator-
>attribute.insert({name,
runtime_attribute});
        break;
      }
```

```
      default: {
        LOG(FATAL) << "Unknown attribute
type: " << attr.type;
      }
    }
  }
}
```

这段代码的两个参数分别是来自 `PNNX` 中的**一个运算符的所有权重数据结构**(`Attribute`)和待初始化的 `RuntimeOperator`. 在以下的循环中,

```
for (const auto& [name, attr] : attrs)
```

我们需要依次将 `Attribute` 中的**数据信息搬运到新初始化的 `RuntimeAttribute` 中**，包括 `type`, `weight_data`, `shapes` 等信息。搬运完成后，再将数据完备的 `RuntimeAttribute` 插入到待初始化的 `RuntimeOperator` 中，同时也记录这个权重的名字。

在`Linear`层中这里的`name`就是`weight`或`bias`, 对于前文测试模型中的`Linear`层，它的`weight shape`是(32, 128)，`weight_data`就是32×128个`float`数据。

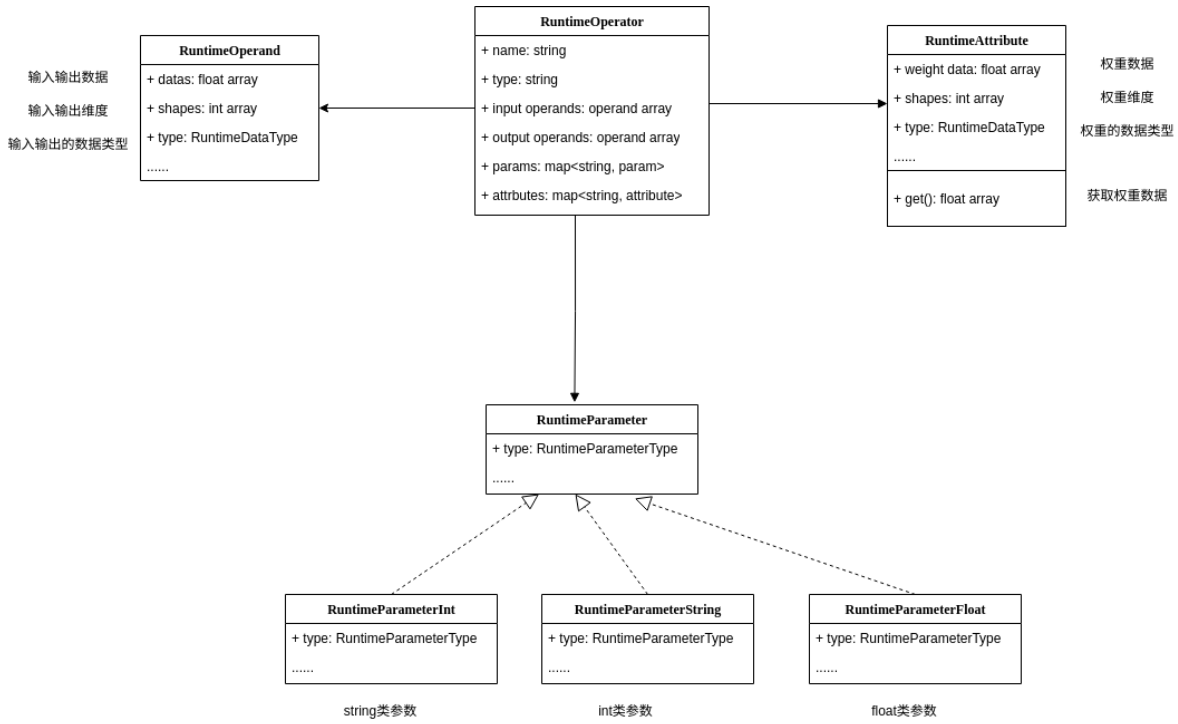
提取PNNX中的参数(Param)到RuntimeParam

Todo, 留做作业

最后的回顾

当一个 `RuntimeOperator` 的所有数据都已经初始化完成后，我们将其插入到 `operators` 数组中。

```
this->operators_.push_back(runtime_operator);
```



从图中不难看出，我们对任意一个`RuntimeOperator`对象都初始化了它的输入输出张量(`RuntimeOperand`)，以及权重数据(`RuntimeAttrinute`).

对KuiperInfer计算图的验证

验证的方法同样是使用单元测试的方式。我们对`RuntimeOperator`数组进行遍历，并打印其中的关键元素。测试代码见`TEST(test_ir, pnnx_graph_all)`，输出如下。不难发现，输出结果和原本的`PNNX`模型结构图保持了一致。

```
I20230607 13:21:19.385279  1956
test_ir.cpp:149] op name: pnnx_input_0 type:
pnnx.Input
I20230607 13:21:19.385316  1956
test_ir.cpp:150] attribute:
I20230607 13:21:19.385339  1956
test_ir.cpp:157] inputs:
I20230607 13:21:19.385363  1956
test_ir.cpp:163] outputs: name: linear
I20230607 13:21:19.385412  1956
test_ir.cpp:167] ----------------------------
-----------
I20230607 13:21:19.385437  1956
test_ir.cpp:149] op name: linear type:
nn.Linear
I20230607 13:21:19.385493  1956
test_ir.cpp:152] bias type: 1 shape: 128
I20230607 13:21:19.385519  1956
test_ir.cpp:152] weight type: 1 shape: 128 x
32
```

```
I20230607 13:21:19.385545  1956
test_ir.cpp:157] inputs: name: pnnx_input_0
shape: 1 x 32
I20230607 13:21:19.385594  1956
test_ir.cpp:163] outputs: name: F.sigmoid_0
I20230607 13:21:19.385643  1956
test_ir.cpp:167] ----------------------------
-----------
I20230607 13:21:19.385666  1956
test_ir.cpp:149] op name: F.sigmoid_0 type:
F.sigmoid
I20230607 13:21:19.385713  1956
test_ir.cpp:157] inputs: name: linear shape:
1 x 128
I20230607 13:21:19.385763  1956
test_ir.cpp:163] outputs: name:
pnnx_output_0
I20230607 13:21:19.385811  1956
test_ir.cpp:167] ----------------------------
-----------
I20230607 13:21:19.385835  1956
test_ir.cpp:149] op name: pnnx_output_0
type: pnnx.Output
I20230607 13:21:19.385883  1956
test_ir.cpp:157] inputs: name: F.sigmoid_0
shape: 1 x 128
I20230607 13:21:19.385932  1956
test_ir.cpp:163] outputs:
```

```
I20230607 13:21:19.385957  1956
test_ir.cpp:167] ----------------------------
----
```

课堂作业

1. `Debug`本节课中的所有单元测试，并观察每个时刻数据状态的变化情况；
2. 编写项目`course3`文件夹下的`RuntimeGraph::InitGraphParams`函数，并通过以下的单元测试。

```
TEST(test_ir, pnnx_graph_all_homework)
```