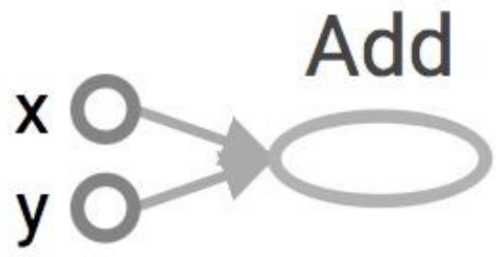
Tensorflow使用**数据流图(Data Flow Graphs)**来定义计算流程。数据流图在定义阶段并不会执行出计算结果。数据流图使得计算的定义与执行分离开来。Tensorflow构建的数据流图是有向无环图。而图的执行需要在**会话(Session)**中完成。

1. 数据流图

我们尝试构建一个简单的图，这个图描述了两个数的加法运算，如下：

```
import tensorflow as tf

a = tf.add(3, 5)
```

使用TensorBoard可视化图：



此处的x、y是被TensorBoard自动命名的，分别代表3与5这两个量。这里需要注意的是数据流图含了**边（edge）**和**节点（node）**，正好与Tensorflow中的tensor与flow对应。tensor代表了数据流图中的边，而flow这个动作代表了数据流图中的节点。节点也被称之为**操作（operation, op）**，一个 op 获得 0 个或多个 `Tensor`, 执行计算, 产生 0 个或多个 `Tensor` 。 每个 Tensor 是一个类型化的多维数组 。关于节点与边，我们会在后续的内容当中详细描述。

这时候如果我们直接输出a，会得到如下结果：

```
print(a)

>> Tensor("Add:0", shape=(), dtype=int32)
```

根据我们的推断，执行加法之后，应该得到一个值，为‘8’，然而并非如此。我们得到的是一个Tensor对象的描述。之所以如此是因为我们定义的变量a代表的是一个图的定义，这个图定义了一个加法运算，但并没有执行。要想得到计算结果，必须在会话中执行。

2. 会话

启动会话的第一步是创建一个session对象。会话提供在图中执行op的一些列方法。一般的模式是，建立会

话，在会话中添加图，然后执行。建立会话可以有两种方式，一种是 `tf.Session()` ，通常使用这种方法创建会话；另一种是 `tf.InteractiveSession()` ，这种方式更多的用在iPython notebook等交互式Python环境中。

在会话中执行图：

```
import tensorflow as tf

# 创建图
a = tf.add(3, 5)
# 创建会话
sess = tf.Session()
# 执行图
res = sess.run(a)  # print(res) 可以得到 8
# 关闭会话
sess.close()
```

为了简便，我们可以使用上下文管理器来创建Session，所以也可以写成：

```
a = tf.add(3, 5)
with tf.Session() as sess:
    print(sess.run(a))
```

2.1 feed与fetch

在调用Session对象的run()方法来执行图时，传入一些张量，这一过程叫做**填充（feed）**，返回的结果类型根据输入的类型而定，这个过程叫**取回（fetch）**。

`tf.Session.run()` 方法可以运行并评估传入的张量：

```
# 运行fetches中的所有张量
run(fetches, feed_dict=None, options=None, run_metadata=None)
```

除了使用 `tf.Session.run()` 以外，在sess持有的上下文中还可以使用 `eval()` 方法。

```
a = tf.add(3, 5)
with tf.Session() as sess:
    print(a.eval())
```

这里需要注意， `a.eval()` 必须在sess持有的上下文中执行才可行。有时候，在交互式的Python环境中，上下文管理器使用不方便，这时候我们可以使用交互式会话来代替会话，这样 `eval()` 方法便可以随时使用：

```
a = tf.add(3, 5)
sess = tf.InteractiveSession()
print(a.eval())
sess.close()
```

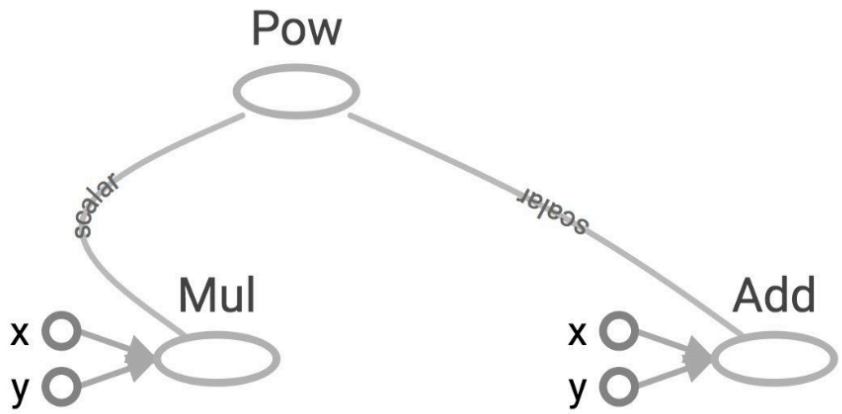
两种会话的区别就是是否支持直接使用 `eval()` 。

2.2 节点依赖

通常一个图会有较多的边与节点，这时候在会话中执行图时，所有依赖的节点均参与计算，如：

```
import tensorflow as tf

x = 2
y = 3

op1 = tf.add(x, y)
op2 = tf.multiply(x, y)
op3 = tf.pow(op2, op1)

with tf.Session() as sess:
    res = sess.run(op3)
```

利用TensorBoard可视化图，如下：



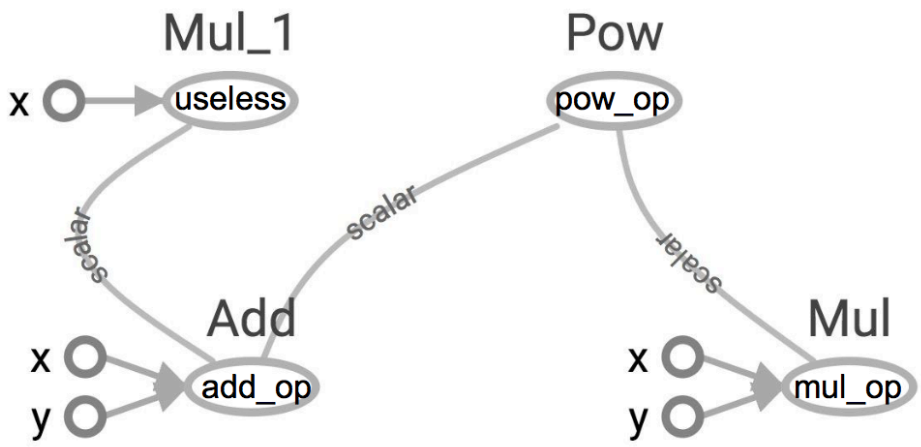
虽然 `session` 中运行的是节点 `op3` ，然而与之相关联的 `op1` 、 `op2` 也参与了运算。

3. 子图

图的构建可以是多种多样的，以上构建的图中数据只有一个流动方向，事实上我们可以构建多个通路的图，每一个通路可以称之为其子图。如下：

```
import tensorflow as tf

x = 2
y = 3

add_op = tf.add(x, y)
mul_op = tf.multiply(x, y)

useless = tf.mul(x, add_op)
pow_op = tf.pow(add_op, mul_op)

with tf.Session() as sess:
    res = sess.run(pow_op)
```

利用TensorBoard可视化图：



可以看到此图中有两个子图，即运行完整的图我们可以得到两个结果 `useless` 和 `pow_op` 。当我们执行上述代码时，相当于执行了一个子图 `pow_op` 。这里需要注意由于得到 `pow_op` 用不到 `useless` 的值，即没有依赖关系，所以 `useless` 这个node并没有执行，如果我们需要得到 `useless` 和 `pow_op` 两个节点的值，则需要稍微改进代码：

```
import tensorflow as tf

x = 2
y = 3

add_op = tf.add(x, y)
mul_op = tf.multiply(x, y)

useless = tf.multiply(x, add_op)
pow_op = tf.pow(add_op, mul_op)

with tf.Session() as sess:
    res1, res2 = sess.run([pow_op, useless])
```

4. 多个图

一个图包含了一组op对象和张量，描述了一个计算流程。但有时候我们需要建构多个图，在不同的图中定义不同的计算，例如一个图描述了算法的训练，另一个图描述了这个算法的正常运行过程，这时候需要在不同的会话中调用不同的图。

创建图使用 `tf.Graph()` 。将图设置为默认图使用 `tf.Graph.as_default()` ，此方法会返回一个上下文管理器。如果不显示的添加一个默认图，系统会自动设置一个全局的默认图。所设置的默认图，在模块范围内定义的节点都将自动加入默认图中。

```
# 构建一个图
g = tf.Graph()
# 将此图作为默认图
with g.as_default():
    op = tf.add(3, 5)
```

上述代码也可以写成：

```
with tf.Graph().as_default():
    op = tf.add(3, 5)
```

`tf.Graph()` *不能用作为上下文管理器，必须调用其方法* `as_default()` 。

构建多个图：

```
g1 = tf.Graph()
with g1.as_default():
    pass

g2 = tf.Graph()
with g2.as_default():
    pass
```

在我们加载了Tensorflow包时，就已经加载了一个图，这也是我们可以不创建图而直接构建边和节点的原因，这些边与节点均加入了默认图。我们可以使用 `tf.get_default_graph()` 方法获取到当前环境中的图，如下：

```
# 获取到了系统的默认图
g1 = tf.get_default_graph()
with g1.as_default():
    pass

g2 = tf.Graph
with g2.as_default():
    pass
```

`tf.get_default_graph()` 获取到的图时当前所在图的环境。如果在上面的代码中在 `g2` 的图环境中执行 `tf.get_default_graph()` ，则得到的图是 `g2` 。

在使用notebook时，由于notebook的一些特性，通常需要指定默认图，否则程序可能报错。为了书写规范，推荐尽量书写完整默认图。

当使用了多个图时，在 `session` 中就需要指定当前 `session` 所运行的图，不指定时，为默认图。如下：

```
# 指定当前会话处理g这个图
with tf.Session(graph=g) as sess:
    sess.run(op)
```

5. 分布式计算

当我们有多个计算设备时，可以将一个计算图的不同子图分别使用不同的计算设备进行运算，以提高运行速度。一般，我们不需要显示的指定使用CPU或者GPU进行计算，Tensorflow能自动检测，如果检测到GPU，Tensorflow会尽可能的利用找到的第一个GPU进行操作。如果机器上有超过一个可用的GPU，出第一个外，其它的GPU默认是不参与计算的。为了让Tensorflow使用这些GPU，我们必须明确指定使用何种设备进行计算。

TensorFlow用指定字符串 `strings` 来标识这些设备. 比如:

- `"/cpu:0"` : 机器中的 CPU
- `"/gpu:0"` : 机器中的 GPU, 如果你有一个的话.
- `"/gpu:1"` : 机器中的第二个 GPU, 以此类推...

这里需要注意的是，CPU可以支持所有类型的运行，而GPU只支持部分运算，例如矩阵乘法。下面是一个使用第3个GPU进行矩阵乘法的运算：

```
with tf.device('/gpu:2'):
    a = tf.constant([[1.0, 2.0, 3.0], [4.0, 5.0, 6.0]], name='a')
    b = tf.constant([[1.0, 2.0], [3.0, 4.0], [5.0, 6.0]], name='b')
    c = tf.matmul(a, b)
```

为了查看节点和边被指派给哪个设备运算，可以在 `session` 中配置“记录设备指派”为开启，如下：

```
with tf.Session(config=tf.ConfigProto(log_device_placement=True)) as sess:
    print(sess.run(c))
```

当指定的设备不存在时，会出现 `InvalidArgumentError` 错误提示。为了避免指定的设备不存在这种情况，可以在创建的 `session` 里把参数 `allow_soft_placement` 设置为 `True`，这时候TF会在设备不存在时自动分配设备。如下：

```
config = tf.ConfigProto(
    log_device_placement=True,
    allow_soft_placement=True)
with tf.Session(config=config) as sess:
    pass
```

6. 使用图与会话的优点

对于图与会话：

- 多个图需要多个会话来运行，每个会话会尝试使用所有可用的资源。
- 不同图、不同会话之间无法直接通信。可以通过python等进行间接通信。
- 最好在一个图中有多个不连贯的子图。

使用图的优点：

- 节省计算资源。我们仅仅只需要计算用到的子图就可以了，无需完整计算整个图。
- 可以将复杂计算分成小块，进行自动微分等运算。
- 促进分布式计算，将工作分布在多个CPU、GPU等设备上。
- 很多机器学习模型可以使用有向图进行表示，与计算图一致。