图包含了节点与边。边可以看做是流动着的数据以及相关依赖关系。而节点表示了一种操作，即对当前流动来的数据的运算。

1. 边（edge）

Tensorflow的边有两种连接关系：**数据依赖**和**控制依赖**。其中，实线边表示数据依赖，代表数据，即张量。虚线边表示控制依赖（control dependency），可用于控制操作的运行，这被用来确保happens-before关系，这类边上没有数据流过，但源节点必须在目的节点开始执行前完成执行。

1.1 数据依赖

数据依赖很容易理解，某个节点会依赖于其它节点的数据，如下所示：

```
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)
```

1.2 控制依赖

控制依赖使用这个方法 `tf.Graph.control_dependencies(control_inputs)` ，返回一个可以使用上下文管理器的对象，用法如下：

```
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)

g = tf.get_default_graph()
with g.control_dependencies([c]):
    d = tf.constant([[1.0, 2.0, 3.0], [4.0, 5.0, 6.0]], name='d')
    e = tf.constant([[1.0, 2.0], [3.0, 4.0], [5.0, 6.0]], name='e')
    f = tf.matmul(d, e)
with tf.Session() as sess:
    sess.run(f)
```

上面的例子中，我们在会话中执行了f这个节点，可以看到与c这个节点并无任何数据依赖关系，然而f这个节点必须等待c这个节点执行完成才能够执行f。最终的结果是c先执行，f再执行。

控制依赖除了上面的写法以外还拥有简便的写法： `tf.control_dependencies(control_inputs)` 。其调用默认图的 `tf.Graph.control_dependencies(control_inputs)` 方法。上面的写法等价于：

```
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)

with tf.control_dependencies([c]):
    d = tf.constant([[1.0, 2.0, 3.0], [4.0, 5.0, 6.0]], name='d')
    e = tf.constant([[1.0, 2.0], [3.0, 4.0], [5.0, 6.0]], name='e')
    f = tf.matmul(d, e)
with tf.Session() as sess:
    sess.run(f)
```

注意：有依赖的op必须写在 `tf.control_dependencies` 上下文中，否则不属于有依赖的op。**如下写法是错误的**：

```
def my_fun():
    a = tf.constant(1)
    b = tf.constant(2)
    c = a + b

    d = tf.constant(3)
    e = tf.constant(4)
    f = a + b
    # 此处 f 不依赖于 c
    with tf.control_dependencies([c]):
        return f
```

1.3 张量的阶、形状、数据类型

Tensorflow数据流图中的边用于数据传输时，数据是以张量的形式传递的。张量有阶、形状和数据类型等属性。

Tensor的阶

在TensorFlow系统中，张量的维数来被描述为**阶**。但是张量的阶和矩阵的阶并不是同一个概念。张量的阶是张量维数的一个数量描述。比如，下面的张量（使用Python中list定义的）就是2阶.

```
t = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

你可以认为一个二阶张量就是我们平常所说的矩阵，一阶张量可以认为是一个向量.对于一个二阶张量你可以用语句 `t[i, j]` 来访问其中的任何元素.而对于三阶张量你可以用 `t[i, j, k]` 来访问其中的任何元素。

阶	数学实例	Python 例子
0	纯量 (或标量。只有大小)	`s = 483`
1	向量(大小和方向)	`v = [1.1, 2.2, 3.3]`
2	矩阵(数据表)	`m = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]`
3	3阶张量 (数据立体)	`t = [[[2], [4], [6]], [[8], [10], [12]], [[14], [16], [18]]]`
n	n阶	`....`

Tensor的形状

TensorFlow文档中使用了三种记号来方便地描述张量的维度：阶，形状以及维数.下表展示了他们之间的关系：

阶	形状	维数	实例
0	[ ]	0-D	一个 0维张量. 一个纯量.
1	[D0]	1-D	一个1维张量的形式[5].
2	[D0, D1]	2-D	一个2维张量的形式[3, 4].
3	[D0, D1, D2]	3-D	一个3维张量的形式 [1, 4, 3].
n	[D0, D1, … Dn-1]	n-D	一个n维张量的形式 [D0, D1, … Dn-1].

张量的阶可以使用 `tf.rank()` 获取到：

```
a = tf.constant([[1.0, 2.0, 3.0], [4.0, 5.0, 6.0]])
tf.rank(a)  # <tf.Tensor 'Rank:0' shape=() dtype=int32> => 2
```

张量的形状可以通过Python中的整数、列表或元祖（int list或tuples）来表示，也或者用 `TensorShape` 类来表示。如下：

```
# 指定shape是[2, 3]的常量,这里使用了list指定了shape，也可以使用ndarray和TensorShape来指定shape
a = tf.constant([[1.0, 2.0, 3.0], [4.0, 5.0, 6.0]], shape=[2, 3])

# 获取shape 方法一：利用tensor的shape属性
a.shape  # TensorShape([Dimension(2), Dimension(3)])

# 获取shape 方法二：利用Tensor的方法get_shape()
a.get_shape()  # TensorShape([Dimension(2), Dimension(3)])

# 获取shape 方法三：利用tf.shape()
tf.shape(a) # <tf.Tensor 'Shape:0' shape=(2, 3) dtype=int32>
```

`TensorShape` 有一个方法 `as_list()`，可以将 `TensorShape` 转化为python的list。

```
a.get_shape().as_list() # [2, 3]
```

同样的我们也可以使用list构建一个TensorShape的对象：

```
ts = tf.TensorShape([2, 3])
```

Tensor的数据类型

Tensor有一个数据类型属性。可以为一个张量指定下列数据类型中的任意一个类型：

数据类型	Python 类型	描述
DT_FLOAT	tf.float32	32 位浮点数.
DT_DOUBLE	tf.float64	64 位浮点数.
DT_INT64	tf.int64	64 位有符号整型.
DT_INT32	tf.int32	32 位有符号整型.
DT_INT16	tf.int16	16 位有符号整型.
DT_INT8	tf.int8	8 位有符号整型.(此处符号位不算在数值位当中)
DT_UINT8	tf.uint8	8 位无符号整型.
DT_STRING	tf.string	可变长度的字节数组.每一个张量元素都是一个字节数组.
DT_BOOL	tf.bool	布尔型.(不能使用number类型表示bool类型，但可转换为bool类型)
DT_COMPLEX64	tf.complex64	由两个32位浮点数组成的复数:实部和虚部。
DT_COMPLEX128	tf.complex128	由两个64位浮点数组成的复数:实部和虚部。
DT_QINT32	tf.qint32	用于量化Ops的32位有符号整型.
DT_QINT8	tf.qint8	用于量化Ops的8位有符号整型.
DT_QUINT8	tf.quint8	用于量化Ops的8位无符号整型.

Tensor的数据类型类似于Numpy中的数据类型，但其加入了对string的支持。

设置与获取Tensor的数据类型

设置Tensor的数据类型：

```
# 方法一
# Tensorflow会推断出类型为tf.float32
a = tf.constant([[1.0, 2.0, 3.0], [4.0, 5.0, 6.0]])

# 方法二
# 手动设置
a = tf.constant([[1.0, 2.0, 3.0], [4.0, 5.0, 6.0]], dtype=tf.float32)

# 方法三 (不推荐)
# 设置numpy类型 未来可能会不兼容
# tf.int32 == np.int32  -> True
a = tf.constant([[1.0, 2.0, 3.0], [4.0, 5.0, 6.0]], dtype=np.float32)
```

获取Tensor的数据类型，可以使用如下方法：

```
a = tf.constant([[1.0, 2.0, 3.0], [4.0, 5.0, 6.0]], name='a')
a.dtype  # tf.float32
print(a.dtype)  # >> <dtype: 'float32'>

b = tf.constant(2+3j)  # tf.complex128 等价于 tf.complex(2., 3.)
print(b.dtype)  # >> <dtype: 'complex128'>

c = tf.constant([True, False], tf.bool)
print(c.dtype)  # <dtype: 'bool'>
```

这里需要注意的是一个张量仅允许一种dtype存在，也就是一个张量中每一个数据的数据类型必须一致。

数据类型转化

如果我们需要将一种数据类型转化为另一种数据类型，需要使用 `tf.cast()` 进行：

```
a = tf.constant([[1.0, 2.0, 3.0], [4.0, 5.0, 6.0]], name='a')
# tf.cast(x, dtype, name=None) 通常用来在两种数值类型之间互转
b = tf.cast(a, tf.int16)
print(b.dtype)  # >> <dtype: 'int16'>
```

有些类型利用 `tf.cast()` 是无法互转的，比如string无法转化成为number类型，这时候可以使用以下方法：

```
# 将string转化为number类型 注意：数字字符可以转化为数字
# tf.string_to_number(string_tensor, out_type = None, name = None)
a = tf.constant([['1.0', '2.0', '3.0'], ['4.0', '5.0', '6.0']], name='a')
num = tf.string_to_number(a)
```

实数数值类型可以使用cast方法转化为bool类型。

2. 节点

图中的节点也可以成为**算子**，它代表一个操作(operation, OP)，一般用来表示数学运算，也可以表示数据输入（feed in）的起点以及输出（push out）的终点，或者是读取/写入持久变量（persistent variable）的终点。常见的节点主要包括以下几种类型：变量、张量元素运算、张量塑形、张量运算、检查点操作、队列和同步操作、张量控制等。

当OP表示数学运算时，每一个运算都会创建一个 `tf.Operation` 对象。常见的操作，例如生成一个变量或者常量、数值计算均创建 `tf.Operation` 对象

2.1 变量

变量用于存储张量，可以使用list、Tensor等来进行初始化，例如：

```
# 使用纯量0进行初始化一个变量
var = tf.Variable(0)
```

2.2 张量元素运算

张量元素运算包含几十种常见的运算，比如张量对应元素的相加、相乘等，这里我们介绍以下几种运算：

- **`tf.add()` shape相同的两个张量对应元素相加。等价于 `A + B` 。**

  ```
  tf.add(1, 2)  # 3
  tf.add([1, 2], [3, 4])  # [4, 6]
  tf.constant([1, 2]) + tf.constant([3, 4])  # [4, 6]
  ```

- **`tf.subtract()` shape相同的两个张量对应元素相减。等价于 `A - B` 。**

  ```
  tf.subtract(1, 2)  # -1
  tf.subtract([1, 2], [3, 4])  # [-2, -2]
  tf.constant([1, 2]) - tf.constant([3, 4])  # [-2, -2]
  ```

- **`tf.multiply()` shape相同的两个张量对应元素相乘。等价于 `A * B` 。**

  ```
  tf.multiply(1, 2)  # 2
  tf.multiply([1, 2], [3, 4])  # [3, 8]
  tf.constant([1, 2]) * tf.constant([3, 4])  # [3, 8]
  ```

- **`tf.scalar_mul()` 一个纯量分别与张量中每一个元素相乘。等价于 `a * B`**

  ```
  sess.run(tf.scalar_mul(10., tf.constant([1., 2.])))  # [10., 20.]
  ```

- **`tf.divide()` shape相同的两个张量对应元素相除。等价于 `A / B` 。这个除法操作是Tensorflow推荐使用的方法。此方法不接受Python自身的数据结构，例如常量或list等。**

```
tf.divide(1, 2)  # 0.5
tf.divide(tf.constant([1, 2]), tf.constant([3, 4]))  # [0.33333333, 0.5]
tf.constant([1, 2]) / tf.constant([3, 4])  # [0.33333333, 0.5]
```

- **`tf.div()` shape相同的两个张量对应元素相除，得到的结果。不推荐使用此方法。tf.divide与tf.div相比，tf.divide符合Python的语义。例如：**

```
1/2  # 0.5
tf.divide(tf.constant(1), tf.constant(2))  # 0.5
tf.div(1/2)  # 0
```

- **`tf.floordiv()` shape相同的两个张量对应元素相除取整数部分。等价于 `A // B` 。**

```
tf.floordiv(1, 2)  # 0
tf.floordiv([4, 3], [2, 5])  # [2, 0]
tf.constant([4, 3]) // tf.constant([2, 5])  # [2, 0]
```

- **`tf.mod()` shape相同的两个张量对应元素进行模运算。等价于 `A % B` 。**

```
tf.mod([4, 3], [2, 5])  # [0, 3]
tf.constant([4, 3]) % tf.constant([2, 5])  # [0, 3]
```

上述运算也支持满足一定条件的shape不同的两个张量进行运算。在此不做过多演示。

除此以外还有很多的逐元素操作的函数，例如求平方 `tf.square()` 、开平方 `tf.sqrt` 、指数运算、三角函数运算、对数运算等等。

2.3 张量运算与塑形

`tf.matmul()` 通常用来做矩阵乘法，张量的阶rank>2，均可使用此方法。

`tf.transpose()` 转置张量。

```
a = tf.constant([[1., 2., 3.], [4., 5., 6.0]])
# tf.matmul(a, b, transpose_a=False, transpose_b=False, adjoint_a=False, adjoint_b
=False, a_is_sparse=False, b_is_sparse=False, name=None)
# tf.transpose(a, perm=None, name='transpose')
tf.matmul(a, tf.transpose(a))  # 等价于 tf.matmul(a, a, transpose_b=True)
```

张量的拼接、切割、形变也是常用的操作：

```
# 沿着某个维度对二个或多个张量进行连接
# tf.concat(values, axis, name='concat')
t1 = [[1, 2, 3], [4, 5, 6]]
t2 = [[7, 8, 9], [10, 11, 12]]
tf.concat([t1, t2], 0) ==> [[1, 2, 3], [4, 5, 6], [7, 8, 9], [10, 11, 12]]


# 对输入的张量进行切片
# tf.slice(input_, begin, size, name=None)
'input' is [[[1, 1, 1], [2, 2, 2]],
            [[3, 3, 3], [4, 4, 4]],
            [[5, 5, 5], [6, 6, 6]]]
tf.slice(input, [1, 0, 0], [1, 1, 3]) ==> [[[3, 3, 3]]]
tf.slice(input, [1, 0, 0], [1, 2, 3]) ==> [[[3, 3, 3],
                                            [4, 4, 4]]]
tf.slice(input, [1, 0, 0], [2, 1, 3]) ==> [[[3, 3, 3]],
                                           [[5, 5, 5]]]


# 将张量分裂成子张量
# tf.split(value, num_or_size_splits, axis=0, num=None, name='split')
# 'value' is a tensor with shape [5, 30]
# Split 'value' into 3 tensors with sizes [4, 15, 11] along dimension 1
split0, split1, split2 = tf.split(value, [4, 15, 11], 1)
tf.shape(split0) ==> [5, 4]
tf.shape(split1) ==> [5, 15]
tf.shape(split2) ==> [5, 11]


# 将张量变为指定shape的新张量
# tf.reshape(tensor, shape, name=None)
# tensor 't' is [1, 2, 3, 4, 5, 6, 7, 8, 9]
# tensor 't' has shape [9]
new_t = tf.reshape(t, [3, 3])
# new_t ==> [[1, 2, 3],
#            [4, 5, 6],
#            [7, 8, 9]]
new_t = tf.reshape(new_t, [-1]) # 这里需要注意shape是一阶张量，此处不能直接使用 -1
# tensor 'new_t' is [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

2.4 其它

其它操作，我们会在之后详述。