Tensorflow本身并非是一门编程语言，而是一种符号式编程库，用来给C++环境中执行算法的主体提供计算流程的描述。这使得Tensorflow拥有了一些编程语言的特征例如拥有变量、常量，却不是编程语言。Tensorflow用图来完成运算流程的描述。一个图是由OP与Tensor构成，即通过OP生成、消费或改变Tensor。

1. 常量

常量是一块只读的内存区域，常量在初始化时就必须赋值，并且之后值将不能被改变。Python并无内置常量关键字，需要我们去实现，而Tensorflow内置了常量方法 `tf.constant()` 。

1.1 普通常量

普通常量使用 `tf.constant()` 初始化得到，其有5个参数。

```
constant(value, dtype=None, shape=None, name="Const", verify_shape=False):
```

- value是必填参数，即常量的初识值。这里需要注意，这个value可以是Python中的list、tuple以及Numpy中的ndarray，但**不可以是Tensor对象**，因为这样没有意义。
- dtype 可选参数，表示数据类型，value中的数据类型应与与dtype中的类型一致，如果不填写则会根据value中值的类型进行推断。
- shape 可选参数，表示value的形状。如果参数 `verify_shape=False` ，shape在与value形状不一致时会修改value的形状。如果参数 `verify_shape=True` ，则要求shape必须与value的shape一致。当shape不填写时，默认为value的shape。

注意： `tf.constant()` 生成的是一个张量。其类型是tf.Tensor。常量本质上就是一个指向固定张量的符号。

1.2 常量存储位置

常量存储在图的定义当中，可以将图序列化后进行查看：

```
const_a = tf.constant([1, 2])
with tf.Session() as sess:
    # tf.Graph.as_graph_def 返回一个代表当前图的序列化的`GraphDef`
    print(sess.graph.as_graph_def()) # 你将能够看到const_a的值
```

当常量包含的数据量较大时，会影响图的加载速度。通常较大的数据使用变量或者在之后读取。

1.3 序列常量

除了使用 `tf.constant()` 生成任意常量以外，我们还可以使用一些方法快捷的生成**序列常量**：

```
# 在指定区间内生成均匀间隔的数字
tf.linspace(start, stop, num, name=None) # slightly different from np.linspace
tf.linspace(10.0, 13.0, 4) ==> [10.0 11.0 12.0 13.0]

# 在指定区间内生成均匀间隔的数字 类似于python中的range
tf.range(start, limit=None, delta=1, dtype=None, name='range')
# 'start' is 3, 'limit' is 18, 'delta' is 3
tf.range(start, limit, delta) ==> [3, 6, 9, 12, 15]
# 'limit' is 5
tf.range(limit) ==> [0, 1, 2, 3, 4]
```

1.4 随机数常量

类似于Python中的random模块，Tensorflow也拥有自己的随机数生成方法。可以生成**随机数常量**：

```
# 生成服从正态分布的随机数
tf.random_normal(shape, mean=0.0, stddev=1.0, dtype=tf.float32, seed=None, name=No
ne)

# 生成服从截断的正态分布的随机数
# 只保留了两个标准差以内的值，超出的值会被丢掉重新生成
tf.truncated_normal(shape, mean=0.0, stddev=1.0, dtype=tf.float32, seed=None,
name=None)

# 生成服从均匀分布的随机值
tf.random_uniform(shape, minval=0, maxval=None, dtype=tf.float32, seed=None,
name=None)

# 将输入张量在第一个维度上进行随机打乱
tf.random_shuffle(value, seed=None, name=None)

# 随机的将张量收缩到给定的尺寸
# 注意：不是打乱，是随机的在某个位置开始裁剪指定大小的样本
# 可以利用样本生成子样本
tf.random_crop(value, size, seed=None, name=None)
```

随机数的生成需要随机数种子seed。Tensorflow默认是加了随机数种子的，如果我们希望生成的随机数是固定值，那么可以指定 `seed` 参数为固定值。

Tensorflow随机数的生成事实上有两个种子在起作用，一个是图级别的，一个是会话级别的。使用 `tf.set_random_seed()` 可以设置图级的随机种子为固定值，这样在两个不同的会话中执行相同的图时，两个会话中得到的随机数一样。

```
t = tf.random_normal(1)
tf.set_random_seed(123)
with tf.Session() as sess:
    t1 = sess.run(t)
with tf.Session() as sess:
    t2 = sess.run(t)
print(t1 == t2)  # True
```

1.5 特殊常量

```
# 生成指定shape的全0张量
tf.zeros(shape, dtype=tf.float32, name=None)
# 生成与输入的tensor相同shape的全0张量
tf.zeros_like(tensor, dtype=None, name=None,optimize=True)
# 生成指定shape的全1张量
tf.ones(shape, dtype=tf.float32, name=None)
# 生成与输入的tensor相同shap的全1张量
tf.ones_like(tensor, dtype=None, name=None, optimize=True)
# 生成一个使用value填充的shape是dims的张量
tf.fill(dims, value, name=None)
```

2. 变量

变量用于存取张量，在Tensorflow中主要使用类 `tf.Variable()` 来实例化一个变量对象，作用类似于Python中的变量。

```
tf.Variable(initial_value=None, trainable=True, collections=None, validate_shape=T
rue, caching_device=None, name=None, variable_def=None, dtype=None, expected_shape
=None, import_scope=None)
```

initial_value是必填参数，即变量的初始值。可以使用Python中的list、tuple、Numpy中的ndarray、Tensor对象或者其他变量进行初始化。

```
# 使用list初始化
var1 = tf.Variable([1, 2, 3])

# 使用ndarray初始化
var2 = tf.Variable(np.array([1, 2, 3]))

# 使用Tensor初始化
var3 = tf.Variable(tf.constant([1, 2, 3]))

# 使用服从正态分布的随机数Tensor初始化
var4 = tf.Variable(tf.random_normal([3, ]))

# 使用变量var1初始化
var5 = tf.Variable(var1)
```

这里需要注意的是：使用 `tf.Variable()` 得到的对象不是Tensor对象，而是承载了Tensor对象的Variable对象。Tensor对象是一个“流动对象”，可以存在于各种操作中，包括存在与Variable中。所以这也涉及到了如何给变量对象赋值、取值问题。

使用 `tf.get_variable()` 创建变量

除了使用 `tf.Variable()` 类实例化一个变量对象以外，还有一种常用的方法来获得一个变量对象： `tf.get_variable()` 这是一个方法，不是类，需要注意的是，默认情况下，使用 `tf.get_variable()` 方法获得一个变量时，其**name不能与已有的name重名**。

```
# 生成一个shape为[3, ]的变量，变量的初值是随机的。
tf.get_variable(name='get_var', shape=[3, ])
# <tf.Variable 'get_var:0' shape=(3,) dtype=float32_ref>
```

关于 `tf.get_variable()` 的更多用法我们会在之后的内容中详解，这里只需要知道这是一种创建变量的方法即可。

2.1变量初始化

变量生成的是一个变量对象，只有初始化之后才能参与计算。我们可以使用变量参与图的构建，但在会话中执行时，必须首先初始化。初始化的方法主要有三种。

- 使用变量的属性 `initializer` 进行初始化：

  例如：

```
var = tf.Variable(tf.constant([1, 2, 3], dtype=tf.float32))
...

with tf.Session() as sess:
  sess.run(var.initializer)
  ...
```

- 使用 `tf.variables_initializer()` 初始化一批变量。

  例如：

```
var1 = tf.Variable(tf.constant([1, 2, 3], dtype=tf.float32))
var2 = tf.Variable(tf.constant([1, 2, 3], dtype=tf.float32))
...

with tf.Session() as sess:
  sess.run(tf.variables_initializer([var1, var2]))
  ...
```

- 使用 `tf.global_variables_initialize()` 初始化所有变量。

  例如：

```
var1 = tf.Variable(tf.constant([1, 2, 3], dtype=tf.float32))
var2 = tf.Variable(tf.constant([1, 2, 3], dtype=tf.float32))
...

with tf.Session() as sess:
  sess.run(tf.global_variables_initialize())
  ...
```

通常，为了简便，第三种方法是首选方法。

在不初始化变量的情况下，也可以使用 `tf.Variable.initialized_value()` 方法获得其中存储的张量，但我们在运行图时，依然需要初始化变量，否则使用到变量的地方依然会出错。

直接获取变量中的张量：

```
var1 = tf.Variable([1, 2, 3])
tensor1 = var1.initialized_value()
```

2.2 变量赋值

变量赋值包含两种情况，第一种情况是初始化时进行赋值，第二种是修改变量的值，这时候需要利用赋值函数：

```
A = tf.Variable(tf.constant([1, 2, 3]), dtype=tf.float32)
with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
    print(sess.run(A))  # >> [1, 2, 3]

    # 赋值方法一
    sess.run(tf.assign(A, [2, 3, 4]))
    print(sess.run(A))  # >> [2, 3, 4]

    # 赋值方法二
    sess.run(A.assign([2, 3, 4]))
    print(sess.run(A))  # >> [2, 3, 4]
```

注意：使用 `tf.Variable.assign()` 或 `tf.assign()` 进行赋值时，必须要求所赋的值的shape与Variable对象中张量的shape一样、dtype一样。

除了使用 `tf.assign()` 以外还可以使用 `tf.assign_add()` 、 `tf.assign_sub()` 。

```
A = tf.Variable(tf.constant([1, 2, 3]))
# 将ref指代的Tensor加上value
# tf.assign_add(ref, value, use_locking=None, name=None)
# 等价于 ref.assign_add(value)
A.assign_add(A, [1, 1, 3])  # >> [2, 3, 6]

# 将ref指代的Tensor减去value
# tf.assign_sub(ref, value, use_locking=None, name=None)
# 等价于 ref.assign_sub(value)
A.assign_sub(A, [1, 1, 3])  # >> [0, 1, 0]
```

2.3 变量操作注意事项

- **注意事项一：**

当我们在会话中运行并输出一个初始化并再次复制的变量时，输出是多少？如下：

```
W = tf.Variable(10)
W.assign(100)
with tf.Session() as sess:
    sess.run(W.initializer)
    sess.run(W)
```

上面的代码将输出 `10` 而不是 `100` ，原因是 `w` 并不依赖于 `W.assign(100)` ， `W.assign(100)` 产生了一个OP，然而在 `sess.run()` 的时候并没有执行这个OP，所以并没有赋值。需要改为：

```
W = tf.Variable(10)
assign_op = W.assign(100)
with tf.Session() as sess:
    sess.run([W.initializer, assign_op])
    sess.run(W)
```

- **注意事项二：**

重复运行变量赋值语句会发生什么?

```
var = tf.Variable(1)
assign_op = var.assign(2 * var)

with tf.Session() as sess:
    sess.run(var.initializer)
    sess.run(assign_op)
    sess.run(var)  # > 2

    sess.run(assign_op)
    sess.run(var)  # > ???
```

这里第二次会输出 `4` ，因为运行了两次赋值op。第一次运行完成时， `var` 被赋值为 `2` ，第二次运行时被赋值为 `4` 。

那么改为如下情况呢?

```
var = tf.Variable(1)
assign_op = var.assign(2 * var)

with tf.Session() as sess:
    sess.run(var.initializer)
    sess.run([assign_op, assign_op])
    sess.run(var)  # ???
```

这里，会输出 `2` 。会话run一次，图执行一次，
而 `sess.run([my_var_times_two, my_var_times_two])` 仅仅相当于查看了两次执行结果，并不是执行了两次。

那么改为如下情况呢?

```
var = tf.Variable(1)
assign_op_1 = var.assign(2 * var)
assign_op_2 = var.assign(3 * var)

with tf.Session() as sess:
    sess.run(var.initializer)
    sess.run([assign_op_1, assign_op_2])
    sess.run(var)  # >> ??
```

这里两次赋值的Op相当于一个图中的两个子图，其执行顺序不分先后，由于两个子图的执行结果会对公共的变量产生影响，当子图A的执行速度快于子图B时，可能是一种结果，反之是另一种结果，所以这样的写法是不安全的写法，执行的结果是可变。但可以通过控制依赖来强制控制两个子图的执行顺序。

- **注意事项三：**

在多个图中给一个变量赋值：

```
W = tf.Variable(10)
sess1 = tf.Session()
sess2 = tf.Session()

sess1.run(W.initializer)
sess2.run(W.initializer)

print(sess1.run(W.assign_add(10)))  # >> 2o
print(sess2.run(W.assign_sub(2)))  # ???

sess1.close()
sess2.close()
```

第二个会打印出 `8` 。因为在两个图中的OP是互不相干的。**每个会话都保留自己的变量副本**，它们分别执行得到结果。

- **注意事项四：**

使用一边量初始化另一个变量时：

```
a = tf.Variable(1)
b = tf.Variable(a)

with tf.Session() as sess:
    sess.run(b.initializer) # 报错
```

出错的原因是 `a` 没有初始化， `b` 就无法初始化。所以使用一个变量初始化另一个变量时，可能会是不安全的。为了确保初始化时不会出错，可以使用如下方法：

```
a = tf.Variable(1)
b = tf.Variable(a.initialized_value())

with tf.Session() as sess:
    sess.run(b.initializer)
```

3. 张量、list、ndarray相互转化

Tensorflow本身的执行环境是C++，Python在其中的角色是设计业务逻辑。我们希望尽可能的将所有的操作都在C++环境中进行，以提高执行效率，然而不可避免的是总有一些地方，需要用到Python和Tensorflow进行数据交互。例如 `a = tf.Variable([1.0, 2.0])` 这句代码事实上使用了Python的list制造了一个原始数据为 `[1.0, 2.0]` 的变量，然后在会话中执行时会将其传递给Tensorflow的C++环境。这就涉及到了Tensorflow中的数据对象Tensor与Python中的数据对象或相关库的数据对象的交互。即如何将Tensor转化为Python数据类型。

Tensorflow的C++底层使用并扩展了numpy的数据结构，例如我们可以使用 `np.float32` 代替 `tf.float32` 等。所以很多时候也会涉及与Numpy中的ndarray的互转。

```
l = [[1.0, 2.0, 3.0], [4.0, 5.0, 6.0]]
print(l)  # >> [[1.0, 2.0, 3.0], [4.0, 5.0, 6.0]]
print(type(l))  # >> list

# 等价的初始化方法一：使用numpy的ndarray初始化
# a = tf.Variable(np.array([[1.0, 2.0, 3.0], [4.0, 5.0, 6.0]]))
# 等价的初始化方法二：使用张量初始化
# a = tf.Variable(tf.constant([[1.0, 2.0, 3.0], [4.0, 5.0, 6.0]]))
# 等价的初始化方法三：使用list初始化
a = tf.Variable([[1.0, 2.0, 3.0], [4.0, 5.0, 6.0]])
print(a)  # >> Tensor("Const_1:0", shape=(2, 3), dtype=int64)
print(type(a))  # <class 'tensorflow.python.framework.ops.Tensor'>

with tf.Session() as sess:
    res = sess.run(a)
    print(res)  # >> [[1.0, 2.0, 3.0], [4.0, 5.0, 6.0]]
    print(type(res))  # >> <class 'numpy.ndarray'>
```

有些数据结构不符合张量的特征，也不存在shape等属性，需要加以区分，例如：

```
t = [1, [2, 3]]
```

`t` 表示一个list，但无法转化为张量。

4. 占位符

我们定义一个图，就类似于定义一个数学函数。当我们定义一个函数$f(x, y)=x*2+y$时不需要知道其中的自变量x,y的值，x,y充当了占位符（placehoders）*。在执行这个函数的时候，再把占位符替换为具体数值。占位符起到了不依赖数据而可以构建函数的目的。使用Tensorflow定义图时，我们也希望图仅仅是用来描述算法的运行过程，而不需要依赖于数据。这样，我们定义的图就更加独立。Tensorflow中，使用 `tf.placeholder` 构建占位符。

在Tensorflow中占位符也是一个节点op，使用如下方法构建占位符：

```
tf.placeholder(dtype, shape=None, name=None)
```

就像使用张量一样，占位符可以直接参与运算：

```
a = tf.placeholder(tf.float32)
b = tf.constant([1, 2, 3])

c = a + b  # 等价于 tf.add(a, b)
```

这里我们没有指定 `shape` ，那么 `shape` 就是 `None` 。这意味着可以使用任意shape的张量输入，但这样做不利于调试，假如我们使用一个 `shape=[2, ]` 的张量输入，那么执行图时一定会报错。所以推荐写上 `shape` 。

`tf.placehoder` 的 `shape` 可以指定部分维度，例如指定 `shape=[None, 10]` ，则需要使用第一个维度为任意长度，第二个维度的长度是10的张量。

4.1 feed

就如同函数的执行，需要传入自变量一样，图构建好之后，运行图时，需要把占位符用张量来替代(feed)，否则这个图无法运行，如下：

```
a = tf.placeholder(tf.int32)
b = tf.constant([1, 2, 3])

c = a + b  # 等价于 tf.add(a, b)

with tf.Session() as sess:
    sess.run(c, feed_dict={a: [2, 3, 4]})  # >> [3 5 7]
```

可以看到建立一个占位符，仅仅只需要输入占位数据的类型即可，即不需要填入具体数据，也不要求写出数据的shape。也就是说可以代替shape可变的数据。

注意： `tf.Variable()` 并不能代替 `tf.placeholder()` 。两者的功能完全不同， `tf.Variable()` 要求必须有shape，也就是规定了shape必须是固定的。

4.2 feed的更多用法

除了 `tf.placeholder` 可以并且必须使用张量替代以外，很多张量均可以使用 `feed_dict` 替代，例如：

```
a = multiply(1, 2)

with tf.Session() as sess:
    sess.run(a, feed_dict={a, 10})  # >> 10
```

为了保证某个张量能够被其它张量替代，可以使用 `tf.Graph.is_feedable(tensor)` 检查 `tensor` 是否可以替代：

```
a = multiply(1, 2)
tf.get_default_graph().is_feedable(a)  # True
```