Tensorflow中所有的Tensor、Operation均拥有自己的名字 `name` ，是其唯一标识符。在Python中，一个变量可以绑定一个对象，同样的也可以绑定一个Tensor或Operation，但这个变量并不是标识符。Tensorflow使用 `name` 的好处是可以使我们定义的图在不同的编程环境中均可以使用，例如再C++，Java，Go等API中也可以通过 `name` 获得Tensor、Variable、Operation，所以 `name` 可以看做是它们的唯一标识符。Tensorflow使用 `name` 也可以更方便的可视化。

实际中，使用Python API（或别的API）时，我们通常使用程序语言中的变量来代表Tensor或Operation，而没有指定它们的 `name` 。这是因为Tensorflow自己命名了 `name` 。

Tensorflow也有作用域(scope)，用来管理Tensor、Operation的name。Tensorflow的作用域分为两种，一种是variable*scope，另一种是name*scope。简言之，variable*scope主要给variable*name加前缀，也可以给op*name加前缀；name*scope是给op_name加前缀。

1. name

Tensor与Operation均有 `name` 属性，但我们只给Operation进行主动命名，Tensor的 `name` 由Operation根据自己的 `name` 与输出数量进行命名（所有的Tensor均由Operation产生）。

例如，我们定义一个常量，并赋予其 `name` ：

```
a = tf.constat([1, 2, 3], name='const')
```

这里我们给常量Op定义了一个 `name` 为 `const` 。 `a` 是常量Op的返回值（输出），是一个张量Tensor对象，所以 `a` 也有自己的 `name` ，为 `const:0` 。

可以看到：Operation的name是我们进行命名的，其输出的张量在其后增加了冒号与索引，是TF根据Operation的name进行命名的。

1.1 Op的name命名规范

首先Tensor对象的name我们并不能直接进行操作，我们只能给Op设置name。其规范是：**由数字、字母、下划线组成，不能以下划线开头**。

正确的命名方式如下：

```
a1 = tf.constant([1, 2, 3], name='const')
a2 = tf.constant([1, 2, 3], name='123')
a3 = tf.constant([1, 2, 3], name='const_')
```

错误的命名方式如下：

```
a1 = tf.constant([1, 2, 3], name='_const')
a2 = tf.constant([1, 2, 3], name='/123')
a3 = tf.constant([1, 2, 3], name='const:0')
```

1.2 Op的name构成

对于一个Op，其 `name` 就是我们设置的 `name` ，所以也是由数字、字母、下划线构成的。我们可以通过查看Operation的 `name` 属性来获得其 `name` 。

例如：

```
# 返回一个什么都不做的Op
op = tf.no_op(name='hello')

print(op.name)  # hello
```

1.3 Tensor的name构成

有些Op会有返回值，其返回值是一个Tensor。Tensor也有 `name` ，但与Op不同的是，我们无法直接设置Tensor的 `name` ，Tensor的 `name` 是由Op的 `name` 所决定的。Tensor的 `name` 相当于在Op的 `name` 之后加上输出索引。Tensor的 `name` 由以下三部分构成：

1. op的名称，也就是我们指定的op的name；
2. 冒号；
3. op输出内容的索引，默认从0开始。

例如：

```
a = tf.constant([1, 2, 3], name='const')

print(a.name)  # const:0
```

这里，我们设置了常量Op的 `name` 为 `const` ，这个Op会返回一个Tensor，所以返回的Tensor的 `name` 就是在其后加上冒号与索引。由于只有一个输出，所以这个输出的索引就是0。

对于两个或多个的输出，其索引依次增加：如下：

```
key, value = tf.ReaderBase.read(..., name='read')

print(key.name)  # read:0
print(value.name)  # read:1
```

1.4 Op与Tensor的默认name

当我们不去设置Op的 `name` 时，Tensorflow也会默认设置一个 `name` ，这也正是 `name` 为可选参数的原因。默认 `name` 往往与Op的类型相同（默认的 `name` 并无严格的规律）。

例如：

```
a = tf.add(1, 2)
# op name为 `Add`
# Tensor name 为 `Add:0`
```

还有一些特殊的Op，我们没法指定其 `name` ，只能使用默认的 `name` ，例如：

```
init = tf.global_variables_initializer()
print(init.name)  # init
```

1.5 重复name的处理方式

Tensorflow并不会严格的规定我们必须设置完全不同的 `name` ，但Tensorflow同时也不允许存在 `name` 相同的Op或Tensor，所以当出现了两个Op设置相同的 `name` 时，Tensorflow会自动给 `name` 加一个后缀。如下：

```
a1 = tf.add(1, 2, name='my_add')
a2 = tf.add(3, 4, name='my_add')

print(a1.name)  # my_add:0
print(a2.name)  # my_add_1:0
```

后缀由下划线与索引组成（注意区分Tensor的name后缀与冒号索引）。从重复的第二个 `name` 开始加后缀，后缀的索引从1开始。

当我们不指定 `name` 时，使用默认的 `name` 也是相同的处理方式：

```
a1 = tf.add(1, 2)
a2 = tf.add(3, 4)

print(a1.name)  # Add:0
print(a2.name)  # Add_1:0
```

1.6 不同图中相同操作name

当我们构建了两个或多个图的时候，如果这两个图中有相同的操作或者相同的 `name` 时，并不会互相影响。如下：

```
g1 = tf.Graph()
with g1.as_default():
    a1 = tf.add(1, 2, name='add')
    print(a1.name)  # add:0

g2 = tf.Graph()
with g2.as_default():
    a2 = tf.add(1, 2, name='add')
    print(a2.name)  # add:0
```

可以看到两个图中的 `name` 互不影响。并没有关系。

2. 通过name获取Op与Tensor

上面，我们介绍了 `name` 可以看做是Op与Tensor的标识符，其实 `name` 不仅仅只是标识其唯一性的工具，也可以利用 `name` 获取到Op与Tensor。（我们可以不借助Python中的变量绑定对象的方式操作Op与Tensor，但是写法很复杂。）

例如，一个计算过程如下：

```
g1 = tf.Graph()
with g1.as_default():
    a = tf.add(3, 5)
    b = tf.multiply(a, 10)

with tf.Session(graph=g1) as sess:
    sess.run(b)  # 80
```

我们也可以使用如下方式，两种方式结果一样：

```
g1 = tf.Graph()
with g1.as_default():
    tf.add(3, 5, name='add')
    tf.multiply(g1.get_tensor_by_name('add:0'), 10, name='mul')

with tf.Session(graph=g1) as sess:
    sess.run(g1.get_tensor_by_name('mul:0'))  # 80
```

这里使用了 `tf.Graph.get_tensor_by_name` 方法。可以根据 `name` 获取Tensor。其返回值是一个Tensor对象。这里要注意Tensor的 `name` 必须写完整。

利用 `name` 也可以获取到相应的Op，这里需要使用 `tf.Graph.get_operation_by_name` 方法。上述例子中，我们在会话中运行的是乘法操作的返回值 `b` 。运行 `b` 的时候，与其相关的依赖，包括乘法Op也运行了，当我们不需要返回值时，我们在会话中可以直接运行Op，而不是Tensor。

例如：

```
g1 = tf.Graph()
with g1.as_default():
    tf.add(3, 5, name='add')
    tf.multiply(g1.get_tensor_by_name('add:0'), 10, name='mul')

with tf.Session(graph=g1) as sess:
    sess.run(g1.get_operation_by_name('mul'))  # None
```

在会话中，fetch一个Tensor，会返回一个Tensor，fetch一个Op，返回 `None` 。

3. name_scope

name*scope可以用来给op*name、tensor_name加前缀。其目的是区分功能模块，可以更方便的在TensorBoard中可视化，同时也便于管理name，以及便于持久化和重新加载。

name*scope使用* `tf.name_scope()` *创建，返回一个上下文管理器。name*scope的参数 `name` 可以是字母、数字、下划线，不能以下划线开头。类似于Op的 `name` 的命名方式。

`tf.name_scope()` 的详情如下：

```
tf.name_scope(
    name,  # 传递给Op name的前缀部分
    default_name=None,  # 默认name
    values=None)  # 检测values中的tensor是否与下文中的Op在一个图中
```

注意： `values` 参数可以不填。当存在多个图时，可能会出现在当前图中使用了在别的图中的Tensor的错误写法，此时如果不在Session中运行图，并不会报错，而填写到了 `values` 参数的中的Tensor都会检测其所在图是否为当前图，提高安全性。

使用 `tf.name_scope()` 的例子：

```
a = tf.constant(1, name='const')
print(a.name)  # >> const:0

with tf.name_scope('scope_name') as name:
    print(name)  # >> scope_name/
    b = tf.constant(1, name='const')
    print(b.name)  # >> scope_name/const:0
```

在一个name_scope的作用域中，可以填写name相同的Op，但Tensorflow会自动加后缀，如下：

```
with tf.name_scope('scope_name') as name:
    a1 = tf.constant(1, name='const')
    print(b.name)  # scope_name/const:0
    a2 = tf.constant(1, name='const')
    print(c.name)  # scope_name/const_1:0
```

多个name_scope

我们可以指定任意多个name*scope，并且可以填写相同* `name` *的两个或多个name*scope，但Tensorflow会自动给name_scope的name加上后缀：

如下：

```
with tf.name_scope('my_name') as name1:
    print(name1)  # >> my_name/

with tf.name_scope('my_name') as name2:
    print(name2)  #>> my_name_1/
```

3.1 多级name_scope

name*scope可以嵌套，嵌套之后的name包含上级name*scope的name。通过嵌套，可以实现多样的命名，如下：

```
with tf.name_scope('name1'):
    with tf.name_scope('name2') as name2:
        print(name2)  # >> name1/name2/
```

不同级的name*scope可以填入相同的name（本质上不同级的name*scope不存在同名），如下：

```
with tf.name_scope('name1') as name1:
    print(name1)  # >> name1/
    with tf.name_scope('name1') as name2:
        print(name2)  # >> name1/name1/
```

在多级name*scope中，op的name会被加上一个前缀，这个前缀取决于所在的name*scope。不同级中的name因为其前缀不同，所以不可能重名，如下：

```
with tf.name_scope('name1'):
    a = tf.constant(1, name='const')
    print(a.name)  # >> name1/const:0
    with tf.name_scope('name2'):
        b = tf.constant(1, name='const')
        print(b.name)  # >> name1/name2/const:0
```

3.2 name_scope的作用范围

使用name*scope可以给op*name加前缀，但不包括 `tf.get_variable()` 创建的变量Op，如下所示：

```
with tf.name_scope('name'):
    var = tf.Variable([1, 2], name='var')
    print(var.name)  # >> name/var:0
    var2 = tf.get_variable(name='var2', shape=[2, ])
    print(var2.name)  # >> var2:0
```

3.3 注意事项

1. 从外部传入的Tensor，并不会在name_scope中加上前缀。例如：

   ```
   a = tf.constant(1, name='const')
   with tf.name_scope('my_name', values=[a]):
      print(a.name)  # >> const:0
   ```

2. Op与name_scope的 `name` 中可以使用 `/` ，但 `/` 并不是 `name` 的构成，还是区分命名空间的符号，不推荐直接使用 `/` 。

3. name*scope的* `default_name` *参数可以在函数中使用。name*scope返回的str类型的scope可以作为 `name` 传给Op的 `name` ，这样做的好处是返回的Tensor的name反映了其所在的模块。例如：

   ```
   def my_op(a, b, c, name=None):
      with tf.name_scope(name, "MyOp", [a, b, c]) as scope:
          a = tf.convert_to_tensor(a, name="a")
          b = tf.convert_to_tensor(b, name="b")
          c = tf.convert_to_tensor(c, name="c")
          # Define some computation that uses `a`, `b`, and `c`.
          return foo_op(..., name=scope)
   ```

4. variable_scope

variable*scope也可以用来给name加前缀，包括variable*name与op*name都可以。与name*scope相比，variable*scope功能要更丰富，最重要的是其可以给get*variable()创建的变量加前缀。

variable*scope使用* `tf.variable_scope()` *创建，返回一个上下文管理器。name*scope的参数 `name` 可

以是字母、数字、下划线，不能以下划线开头。类似于变量的参数 `name` 以及name_scope的命名。

`tf.variable_scope` 的详情如下:

```
variable_scope(name_or_scope,  # 可以是name或者别的variable_scope
               default_name=None,
               values=None,
               initializer=None,
               regularizer=None,
               caching_device=None,
               partitioner=None,
               custom_getter=None,
               reuse=None,
               dtype=None,
               use_resource=None):
```

4.1 给op_name加前缀

variable*scope包含了name*scope的功能，默认的variable*scope的* `name` *等于其中的name*scope的 `name` 。如下:

```
g = tf.Graph()
with g.as_default():
    with tf.variable_scope('abc') as scope:
        # 输出variable_scope的`name`
        print(scope.name)  # >> abc

        n_scope = g.get_name_scope()
        # 输出name_scope的`name`
        print(n_scope)  # >> abc
```

在variable_scope下也可以给Op与Tensor加前缀:

```
with tf.variable_scope('abc') as scope:
    a = tf.constant(1, name='const')
    print(a.name)  # >> abc/const:0
```

4.2 给variable_name加前缀

variable*scope与name*scope最大的不同就是，variable_scope可以给使用 `tf.get_variable()` 创建的变量加前缀，如下:

```
with tf.variable_scope('my_scope'):
    var = tf.get_variable('var', shape=[2, ])
    print(var.name)  # >> my_scope/var:0
```

`tf.get_variable()` 创建变量时，必须有 `name` 与 `shape` 。 `dtype` 可以不填，默认是 `tf.float32` 。同时，使用 `tf.get_variable()` 创建变量时，**name不能填入重复的**。

以下写法是错误的：

```
a = tf.get_variable('abcd', shape=[1])
b = tf.get_variable('abcd', shape=[1])  # ValueError
```

4.3 同名variable_scope

创建两个或多个variable*scope时可以填入相同的* `name` *，此时相当于创建了一个variable*scope与两个name_scope。

```
g = tf.Graph()
with g.as_default():
    with tf.variable_scope('abc') as scope:
        print(scope.name)  # >> abc
        n_scope = g.get_name_scope()
        print(n_scope)  # >> abc

    with tf.variable_scope('abc') as scope:
        print(scope.name)  # >> abc
        n_scope = g.get_name_scope()
        print(n_scope)  # >> abc_1
```

同名的variable*scope，本质上属于一个variable*scope，不允许通过 `tf.get_variable` 创建相同name的Variable。下面的代码会抛出一个ValueError的错误：

```
with tf.variable_scope('s1'):
    vtf.get_variable('var')

with tf.variable_scope('s1'):
    # 抛出错误
    tf.get_variable('var')
```

使用一个variable*scope初始化另一个variable*scope。这两个variable*scope的name一样，name*scope的name不一样。相当于是同一个variable_scope。如下：

```
with tf.variable_scope('my_scope') as scope1:
    print(scope1.name)  # >> my_scope

with tf.variable_scope(scope1) as scope2:
    print(scope2.name)  # >> my_scope
```

variable_scope的reuse参数

创建variable*scope时，默认的* `reuse` *参数为* `None` *，当设置其为* `True` *时，此处的variable*scope成为了共享变量scope，即可以利用 `tf.get_variable()` 共享其它同名的但 `reuse` 参数为 `None` 的variable_scope中的变量。此时 `tf.get_variable()` 的作用成为了“获取同名变量”，而不能够创建变量（尝试创建一个新变量会抛出ValueError的错误）。

注意： `reuse` 参数的取值是 `None` 或者 `True` ，不推荐使用 `False` 代替 `None` 。

`tf.get_variable()` 配合variable_scope使用，才能够发挥其create与get变量的双重能力。

例如：

```
with tf.variable_scope('my_scope') as scope1:
    # 默认情况下reuse=None

    # 创建变量
    var = tf.get_variable('var', shape=[2, 3])

with tf.variable_scope('my_scope', reuse=True) as scope2:

    # 使用tf.get_variable()获取变量
    var2 = tf.get_variable('var')
    var3 = tf.get_variable('var')

print(var is var2)  # >> True
print(var is var3)  # >> True
```

使用scope.reuse*variables分割variable*scope

使用 `scope.reuse_variables()` 可以将一个variable_scope分割成为可以创建变量与可以重用变量的两个块。例如：

```
with tf.variable_scope('my_scope') as scope:
    a1 = tf.get_variable('my_var', shape=[1, 2])
    scope.reuse_variables()
    a2 =tf.get_variable('my_var')

    print(a1 is a2)  # >> True
```

4.4 多级变量作用域

我们可以在一个作用域中，使用另一个作用域，这时候作用域中的name也会叠加在一起，如下：

```
with tf.variable_scope('first_scope') as first_scope:
    print(first_scope.name)  # >> first_scope
    with tf.variable_scope('second_scope') as second_scope:
        print(second_scope.name)  # >> first_scope/second_scope
        print(tf.get_variable('var', shape=[1, 2]).name)
        # >> first_scope/second_scope/var:0
```

跳过作用域

如果在开启的一个变量作用域里使用之前预定义的一个作用域，则会跳过当前变量的作用域，保持预先存在的作用域不变：

```
with tf.variable_scope('outside_scope') as outside_scope:
    print(outside_scope.name)  # >> outside_scope

with tf.variable_scope('first_scope') as first_scope:
    print(first_scope.name)  # >> first_scope
    print(tf.get_variable('var', shape=[1, 2]).name)   # >> first_scope/var:0

    with tf.variable_scope(outside_scope) as second_scope:
        print(second_scope.name)  # >> outside_scope
        print(tf.get_variable('var', shape=[1, 2]).name)  # >> outside_scope/var:0
```

多级变量作用域中的reuse

在多级变量作用域中，规定外层的变量作用域设置了 `reuse=True`，内层的所有作用域的 `reuse` 必须设置为 `True`（设置为其它无用）。

多级变量作用域中，使用 `tf.get_variable()` 的方法如下：

```
# 定义
with tf.variable_scope('s1') as s1:
    tf.get_variable('var', shape=[1,2])
    with tf.variable_scope('s2') as s2:
        tf.get_variable('var', shape=[1,2])
        with tf.variable_scope('s3') as s3:
            tf.get_variable('var', shape=[1,2])

# 使用
with tf.variable_scope('s1', reuse=True) as s1:
    v1 = tf.get_variable('var')
    with tf.variable_scope('s2', reuse=None) as s2:
        v2 = tf.get_variable('var')
        with tf.variable_scope('s3', reuse=None) as s3:
            v3 = tf.get_variable('var')
```

4.5 变量作用域的初始化

variable_scope可以在创建时携带一个初始化器。其作用是将在其中的变量自动使用初始化器的方法进行初始化。方法如下：

```
# 直接使用tf.get_variable()得到的是随机数
var1 = tf.get_variable('var1', shape=[3, ])
var1.eval()
# 输出的可能值：
# array([-0.92183685, -0.078825  , -0.61367416], dtype=float32)
```

```
# 使用variable_scope的初始化器
with tf.variable_scope(
                      'scope',
                      initializer=tf.constant_initializer(1)):
    var2 = tf.get_variable('var2', shape=[3, ])
    var1.eval()  # 输出 [ 1.  1.  1.]
```

常见的初始化器有：

```
# 常数初始化器
tf.constant_initializer(value=0, dtype=tf.float32)
# 服从正态分布的随机数初始化器
tf.random_normal_initializer(mean=0.0, stddev=1.0, seed=None, dtype=tf.float32)
# 服从截断正态分布的随机数初始化器
tf.truncated_normal_initializer(mean=0.0, stddev=1.0, seed=None, dtype=tf.float32)
# 服从均匀分布的随机数初始化器
tf.random_uniform_initializer(minval=0, maxval=None, seed=None, dtype=tf.float32)
# 全0初始化器
tf.zeros_initializer(dtype=tf.float32)
# 全1初始化器
tf.ones_initializer(dtype=tf.float32)
```