Tensorflow Python API 翻译（array_ops）

该章介绍有关张量转换的API

数据类型投射

Tensorflow提供了很多的数据类型投射操作，你能将数据类型投射到一个你想要的数据类型上去。

`tf.string_to_number(string_tensor, out_type = None, name = None)`

解释：这个函数是将一个`string`的`Tensor`转换成一个数字类型的`Tensor`。但是要注意一点，如果你想转换的数字类型是`tf.float32`，那么这个`string`去掉引号之后，里面的值必须是一个合法的浮点数，否则不能转换。如果你想转换的数字类型是`tf.int32`，那么这个`string`去掉引号之后，里面的值必须是一个合法的浮点数或者整型，否则不能转换。

使用例子：

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

import tensorflow as tf
import numpy as np

sess = tf.Session()
data = tf.constant('123')
print sess.run(data)
d = tf.string_to_number(data)
print sess.run(d)
```

输入参数：

- `string_tensor`: 一个`string`类型的`Tensor`。
- `out_type`: 一个可选的数据类型`tf.DType`，默认的是`tf.float32`，但我们也可以选择`tf.int32`或者`tf.float32`。
- `name`:（可选）为这个操作取一个名字。

输出参数：

- 一个`Tensor`，数据类型是`out_type`，数据维度和`string_tensor`相同。

`tf.to_double(x, name = 'ToDouble')`

解释：这个函数是将一个`Tensor`的数据类型转换成`float64`。

使用例子：

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

import tensorflow as tf
import numpy as np

sess = tf.Session()
data = tf.constant(123)
print sess.run(data)
d = tf.to_double(data)
print sess.run(d)
```

输入参数：

- `x`: 一个`Tensor`或者是`SparseTensor`。

- name:（可选）为这个操作取一个名字。

输出参数：

- 一个Tensor或者SparseTensor，数据类型是float64，数据维度和x相同。

提示：

- 错误: 如果x是不能被转换成float64类型的，那么将报错。

---

tf.to_float(x, name = 'ToFloat')

解释：这个函数是将一个Tensor的数据类型转换成float32。

使用例子：

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

import tensorflow as tf
import numpy as np

sess = tf.Session()
data = tf.constant(123)
print sess.run(data)
d = tf.to_float(data)
print sess.run(d)
```

输入参数：

- x: 一个Tensor或者是SparseTensor。
- name:（可选）为这个操作取一个名字。

输出参数：

- 一个Tensor或者SparseTensor，数据类型是float32，数据维度和x相同。

提示：

- 错误: 如果x是不能被转换成float32类型的，那么将报错。

---

tf.to_bfloat16(x, name = 'ToBFloat16')

解释：这个函数是将一个Tensor的数据类型转换成bfloat16。

译者注：这个API的作用不是很理解，但我测试了一下，输入的x必须是浮点型的，别的类型都不行。

使用例子：

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

import tensorflow as tf
import numpy as np

sess = tf.Session()
data = tf.constant([x for x in range(20)], tf.float32)
print sess.run(data)
d = tf.to_bfloat16(data)
print sess.run(d)
```

输入参数：

- x: 一个Tensor或者是SparseTensor。
- name:（可选）为这个操作取一个名字。

输出参数：

- 一个`Tensor`或者`SparseTensor`，数据类型是`bfloat16`，数据维度和`x`相同。

提示：

- 错误: 如果`x`是不能被转换成`bfloat16`类型的，那么将报错。

`tf.to_int32(x, name = 'ToInt32')`

解释：这个函数是将一个`Tensor`的数据类型转换成`int32`。

使用例子：

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

import tensorflow as tf
import numpy as np

sess = tf.Session()
data = tf.constant([x for x in range(20)], tf.float32)
print sess.run(data)
d = tf.to_int32(data)
print sess.run(d)
```

输入参数：

- `x`: 一个`Tensor`或者是`SparseTensor`。
- `name`:（可选）为这个操作取一个名字。

输出参数：

- 一个`Tensor`或者`SparseTensor`，数据类型是`int32`，数据维度和`x`相同。

提示：

- 错误: 如果`x`是不能被转换成`int32`类型的，那么将报错。

`tf.to_int64(x, name = 'ToInt64')`

解释：这个函数是将一个`Tensor`的数据类型转换成`int64`。

使用例子：

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

import tensorflow as tf
import numpy as np

sess = tf.Session()
data = tf.constant([x for x in range(20)], tf.float32)
print sess.run(data)
d = tf.to_int64(data)
print sess.run(d)
```

输入参数：

- `x`: 一个`Tensor`或者是`SparseTensor`。
- `name`:（可选）为这个操作取一个名字。

输出参数：

- 一个`Tensor`或者`SparseTensor`，数据类型是`int64`，数据维度和`x`相同。

提示：

- 错误: 如果`x`是不能被转换成`int64`类型的，那么将报错。

`tf.cast(x, dtype, name = None)`

解释：这个函数是将一个`Tensor`或者`SparseTensor`的数据类型转换成`dtype`。

使用例子：

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

import tensorflow as tf
import numpy as np

sess = tf.Session()
data = tf.constant([x for x in range(20)], tf.float32)
print sess.run(data)
d = tf.cast(data, tf.int32)
print sess.run(d)
```

输入参数：

- `x`: 一个`Tensor`或者是`SparseTensor`。
- `dtype`: 目标数据类型。
- `name`:（可选）为这个操作取一个名字。

输出参数：

- 一个`Tensor`或者`SparseTensor`，数据维度和`x`相同。

提示：

- 错误: 如果`x`是不能被转换成`dtype`类型的，那么将报错。

数据维度转换

Tensorflow提供了很多的数据维度转换操作，你能改变数据的维度，将它变成你需要的维度。

`tf.shape(input, name = None)`

解释：这个函数是返回`input`的数据维度，返回的`Tensor`数据维度是一维的。

使用例子：

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

import tensorflow as tf
import numpy as np

sess = tf.Session()
data = tf.constant([[[1, 1, 1], [2, 2, 2]], [[3, 3, 3], [4, 4, 4]]])
print sess.run(data)
d = tf.shape(data)
print sess.run(d)
```

输入参数：

- `input`: 一个`Tensor`。
- `name`:（可选）为这个操作取一个名字。

输出参数：

- 一个`Tensor`，数据类型是`int32`。

`tf.size(input, name = None)`

解释：这个函数是返回`input`中一共有多少个元素。

使用例子：

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

import tensorflow as tf
import numpy as np

sess = tf.Session()
data = tf.constant([[[1, 1, 1], [2, 2, 2]], [[3, 3, 3], [4, 4, 4]]])
print sess.run(data)
d = tf.size(data)
print sess.run(d)
```

输入参数：

- input: 一个Tensor。
- name:（可选）为这个操作取一个名字。

输出参数：

- 一个Tensor，数据类型是int32。

---

tf.rank(input, name = None)

解释：这个函数是返回Tensor的秩。

注意：Tensor的秩和矩阵的秩是不一样的，Tensor的秩指的是元素维度索引的数目，这个概念也被成为order, degree或者ndims。比如，一个Tensor的维度是[1, 28, 28, 1]，那么它的秩就是4。

使用例子：

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

import tensorflow as tf
import numpy as np

sess = tf.Session()
data = tf.constant([[[1, 1, 1], [2, 2, 2]], [[3, 3, 3], [4, 4, 4]]])
print sess.run(data)
d = tf.rank(data)
print sess.run(tf.shape(data))
print sess.run(d)
```

输入参数：

- input: 一个Tensor。
- name:（可选）为这个操作取一个名字。

输出参数：

- 一个Tensor，数据类型是int32。

---

tf.reshape(tensor, shape, name = None)

解释：这个函数的作用是对tensor的维度进行重新组合。给定一个tensor，这个函数会返回数据维度是shape的一个新的tensor，但是tensor里面的元素不变。

如果shape是一个特殊值[-1]，那么tensor将会变成一个扁平的一维tensor。

如果shape是一个一维或者更高的tensor，那么输入的tensor将按照这个shape进行重新组合，但是重新组合的tensor和原来的tensor的元素是必须相同的。

使用例子：

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

import tensorflow as tf
import numpy as np
```

```
sess = tf.Session()
data = tf.constant([[[1, 1, 1], [2, 2, 2]], [[3, 3, 3], [4, 4, 4]]])
print sess.run(data)
print sess.run(tf.shape(data))
d = tf.reshape(data, [-1])
print sess.run(d)
d = tf.reshape(data, [3, 4])
print sess.run(d)
```

输入参数：

- tensor: 一个Tensor。
- shape: 一个Tensor，数据类型是int32，定义输出数据的维度。
- name:（可选）为这个操作取一个名字。

输出参数：

- 一个Tensor，数据类型和输入数据相同。

---

tf.squeeze(input, squeeze_dims = None, name = None)

解释：这个函数的作用是将input中维度是1的那一维去掉。但是如果你不想把维度是1的全部去掉，那么你可以使用squeeze_dims参数，来指定需要去掉的位置。

使用例子：

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

import tensorflow as tf
import numpy as np

sess = tf.Session()
data = tf.constant([[1, 2, 1], [3, 1, 1]])
print sess.run(tf.shape(data))
d_1 = tf.expand_dims(data, 0)
d_1 = tf.expand_dims(d_1, 2)
d_1 = tf.expand_dims(d_1, -1)
d_1 = tf.expand_dims(d_1, -1)
print sess.run(tf.shape(d_1))
d_2 = d_1
print sess.run(tf.shape(tf.squeeze(d_1)))
print sess.run(tf.shape(tf.squeeze(d_2, [2, 4])))

# 't' is a tensor of shape [1, 2, 1, 3, 1, 1]
# shape(squeeze(t)) ==> [2, 3]

# 't' is a tensor of shape [1, 2, 1, 3, 1, 1]
# shape(squeeze(t, [2, 4])) ==> [1, 2, 3, 1]
```

输入参数：

- input: 一个Tensor。
- squeeze_dims:（可选）一个序列，索引从0开始，只移除该列表中对应位的tensor。默认下，是一个空序列[]。
- name:（可选）为这个操作取一个名字。

输出参数：

- 一个Tensor，数据类型和输入数据相同。

---

tf.expand_dims(input, dim, name = None)

解释：这个函数的作用是向input中插入维度是1的张量。

我们可以指定插入的位置dim，dim的索引从0开始，dim的值也可以是负数，从尾部开始插入，符合 python

的语法。

这个操作是非常有用的。举个例子，如果你有一张图片，数据维度是`[height, width, channels]`，你想要加入“批量”这个信息，那么你可以这样操作`expand_dims(images, 0)`，那么该图片的维度就变成了`[1, height, width, channels]`。

这个操作要求：

`-1-input.dims() <= dim <= input.dims()`

这个操作是`squeeze()`函数的相反操作，可以一起灵活运用。

使用例子：

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

import tensorflow as tf
import numpy as np

sess = tf.Session()
data = tf.constant([[1, 2, 1], [3, 1, 1]])
print sess.run(tf.shape(data))
d_1 = tf.expand_dims(data, 0)
print sess.run(tf.shape(d_1))
d_1 = tf.expand_dims(d_1, 2)
print sess.run(tf.shape(d_1))
d_1 = tf.expand_dims(d_1, -1)
print sess.run(tf.shape(d_1))
```

输入参数：

- `input`: 一个`Tensor`。
- `dim`: 一个`Tensor`，数据类型是`int32`，标量。
- `name`:（可选）为这个操作取一个名字。

输出参数：

- 一个`Tensor`，数据类型和输入数据相同，数据和`input`相同，但是维度增加了一维。

数据抽取和结合

Tensorflow提供了很多的数据抽取和结合的方法。

`tf.slice(input_, begin, size, name = None)`

解释：这个函数的作用是从输入数据`input`中提取出一块切片，切片的尺寸是`size`，切片的开始位置是`begin`。切片的尺寸`size`表示输出`tensor`的数据维度，其中`size[i]`表示在第`i`维度上面的元素个数。开始位置`begin`表示切片相对于输入数据`input_`的每一个偏移量，比如数据`input_`是

`[[[1, 1, 1], [2, 2, 2]],`
`[[33, 3, 3], [4, 4, 4]],`
`[[5, 5, 5], [6, 6, 6]]]`，

`begin`为`[1, 0, 0]`，那么数据的开始位置是33。因为，第一维偏移了1，其余几位都没有偏移，所以开始位置是33。

操作满足：

`size[i] = input.dim_size(i) - begin[i]`
`0 <= begin[i] <= begin[i] + size[i] <= Di for i in [0, n]`

使用例子：

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

import tensorflow as tf
import numpy as np

sess = tf.Session()
input = tf.constant([[[1, 1, 1], [2, 2, 2]],
```

```
                    [[3, 3, 3], [4, 4, 4]],
                    [[5, 5, 5], [6, 6, 6]]])
data = tf.slice(input, [1, 0, 0], [1, 1, 3])
print sess.run(data)
data = tf.slice(input, [1, 0, 0], [1, 2, 3])
print sess.run(data)
data = tf.slice(input, [1, 0, 0], [2, 1, 3])
print sess.run(data)
data = tf.slice(input, [1, 0, 0], [2, 2, 2])
print sess.run(data)
```

输入参数：

- input_: 一个Tensor。
- begin: 一个Tensor，数据类型是int32或者int64。
- size: 一个Tensor，数据类型是int32或者int64。
- name:（可选）为这个操作取一个名字。

输出参数：

- 一个Tensor，数据类型和input_相同。

---

tf.split(split_dim, num_split, value, name = 'split')

解释：这个函数的作用是，沿着split_dim维度将value切成num_split块。要求，num_split必须被value.shape[split_dim]整除，即value.shape[split_dim] % num_split == 0。

使用例子：

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

import tensorflow as tf
import numpy as np

sess = tf.Session()
input = tf.random_normal([5,30])
print sess.run(tf.shape(input))[0] / 5
split0, split1, split2, split3, split4 = tf.split(0, 5, input)
print sess.run(tf.shape(split0))
```

输入参数：

- split_dim: 一个0维的Tensor，数据类型是int32，该参数的作用是确定沿着哪个维度进行切割，参数范围 [0, rank(value))。
- num_split: 一个0维的Tensor，数据类型是int32，切割的块数量。
- value: 一个需要切割的Tensor。
- name:（可选）为这个操作取一个名字。

输出参数：

- 从value中切割的num_split个Tensor。

---

tf.tile(input, multiples, name = None)

解释：这个函数的作用是通过给定的tensor去构造一个新的tensor。所使用的方法是将input复制multiples次，输出的tensor的第i维有input.dims(i) * multiples[i]个元素，input中的元素被复制multiples[i]次。比如，input = [a b c d], multiples = [2]，那么tile(input, multiples) = [a b c d a b c d]。

使用例子：

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
```

```
import tensorflow as tf
import numpy as np

sess = tf.Session()
data = tf.constant([[1, 2, 3, 4], [9, 8, 7, 6]])
d = tf.tile(data, [2,3])
print sess.run(d)
```

输入参数：

- input_: 一个Tensor，数据维度是一维或者更高维度。
- multiples: 一个Tensor，数据类型是int32，数据维度是一维，长度必须和input的维度一样。
- name:（可选）为这个操作取一个名字。

输出参数：

- 一个Tensor，数据类型和input相同。

---

tf.pad(input, paddings, name = None)

解释：这个函数的作用是向input中按照paddings的格式填充0。paddings是一个整型的Tensor，数据维度是[n, 2]，其中n是input的秩。对于input的中的每一维D，paddings[D, 0]表示增加多少个0在input之前，paddings[D, 1]表示增加多少个0在input之后。举个例子，假设paddings = [[1, 1], [2, 2]]和input的数据维度是[2,2]，那么最后填充完之后的数据维度如下：

$$\begin{matrix} 1 & 2 & 1 \\ 2 & 2 & 2 \end{matrix} + \begin{matrix} 1 & 2 & 1 \\ 2 & 2 & 2 \end{matrix} = \begin{matrix} 4 \\ 6 \end{matrix}$$

填充之后的数据维度

也就是说，最后的数据维度变成了[4,6]。

使用例子：

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

import tensorflow as tf
import numpy as np

sess = tf.Session()
t = tf.constant([[[3,3,],[2,2]]])
print sess.run(tf.shape(t))
paddings = tf.constant([[3,3],[1,1],[2,2]])
print sess.run(tf.pad(t, paddings)).shape
```

输入参数：

- input: 一个Tensor。
- paddings: 一个Tensor，数据类型是int32。
- name:（可选）为这个操作取一个名字。

输出参数：

- 一个Tensor，数据类型和input相同。

---

tf.concat(concat_dim, value, name = 'concat')

解释：这个函数的作用是沿着concat_dim维度，去重新串联value，组成一个新的tensor。

使用例子：

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

import tensorflow as tf
```

```
import numpy as np

sess = tf.Session()
t1 = tf.constant([[1, 2, 3], [4, 5, 6]])
t2 = tf.constant([[7, 8, 9], [10, 11, 12]])
d1 = tf.concat(0, [t1, t2])
d2 = tf.concat(1, [t1, t2])
print sess.run(d1)
print sess.run(tf.shape(d1))
print sess.run(d2)
print sess.run(tf.shape(d2))

# output
[[ 1  2  3]
 [ 4  5  6]
 [ 7  8  9]
 [10 11 12]]

[[ 1  2  3  7  8  9]
 [ 4  5  6 10 11 12]]

# tips
从直观上来看，我们取的concat_dim的那一维的元素个数肯定会增加。比如，上述例子中的d1的第0维增加了，而且d1.shape[0] = t1.shape[0]+t2.shape[0]。
```

输入参数：

- `concat_dim`: 一个零维度的`Tensor`，数据类型是`int32`。
- `values`: 一个`Tensor`列表，或者一个单独的`Tensor`。
- `name`:（可选）为这个操作取一个名字。

输出参数：

- 一个重新串联之后的`Tensor`。

`tf.pack(values, name = 'pack')`

解释：这个函数的作用是将秩为`R`的`tensor`打包成一个秩为`R+1`的`tensor`。具体的公式可以表示为：

```
tf.pack([x, y, z]) = np.asqrray([x, y, z])
```

使用例子：

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

import tensorflow as tf

x = tf.constant([1,2,3])
y = tf.constant([4,5,6])
z = tf.constant([7,8,9])

p = tf.pack([x,y,z])

sess = tf.Session()
print sess.run(tf.shape(p))
print sess.run(p)
```

输入参数：

- `values`: 一个`Tensor`的列表，每个`Tensor`必须有相同的数据类型和数据维度。
- `name`:（可选）为这个操作取一个名字。

输出参数：

- `output`: 一个打包的`Tensor`，数据类型和`values`相同。

`tf.unpack(value, num = None, name = 'unpack')`

解释：这个函数的作用是将秩为R+1的tensor解压成一些秩为R的tensor。其中，num表示要解压出来的tensor的个数。如果，num没有被指定，那么num = value.shape[0]。如果，value.shape[0]无法得到，那么系统将抛出异常ValueError。具体的公式可以表示为：

```
tf.unpack(x, n) = list(x)
```

使用例子：

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

import tensorflow as tf

x = tf.constant([1,2,3])
y = tf.constant([4,5,6])
z = tf.constant([7,8,9])

p = tf.pack([x,y,z])

sess = tf.Session()
print sess.run(tf.shape(p))
pp = tf.unpack(p,3)
print sess.run(pp)
```

输入参数：

- value: 一个秩大于0的Tensor。
- num: 一个整型，value的第一维度的值。
- name:（可选）为这个操作取一个名字。

输出参数：

- 从value中解压出来的一个Tensor数组。

异常：

- ValueError: 如果num没有被正确指定，那么将抛出异常。

---

tf.reverse_sequence(input, seq_lengths, seq_dim, name = None)

解释：将input中的值沿着第seq_dim维度进行翻转。

这个操作先将input沿着第0维度切分，然后对于每个切片，将切片长度为seq_lengths[i]的值，沿着第seq_dim维度进行翻转。

向量seq_lengths中的值必须满足seq_lengths[i] < input.dims[seq_dim]，并且其长度必须是input_dims(0)。

对于每个切片i的输出，我们将第seq_dim维度的前seq_lengths[i]的数据进行翻转。

比如：

```
# Given this:
seq_dim = 1
input.dims = (4, 10, ...)
seq_lengths = [7, 2, 3, 5]

# 因为input的第0维度是4，所以先将input切分成4个切片；
# 因为seq_dim是1，所以我们按着第1维度进行翻转。
# 因为seq_lengths[0] = 7，所以我们第一个切片只翻转前7个值，该切片的后面的值保持不变。
# 因为seq_lengths[1] = 2，所以我们第一个切片只翻转前2个值，该切片的后面的值保持不变。
# 因为seq_lengths[2] = 3，所以我们第一个切片只翻转前3个值，该切片的后面的值保持不变。
# 因为seq_lengths[3] = 5，所以我们第一个切片只翻转前5个值，该切片的后面的值保持不变。
output[0, 0:7, :, ...] = input[0, 7:0:-1, :, ...]
output[1, 0:2, :, ...] = input[1, 2:0:-1, :, ...]
output[2, 0:3, :, ...] = input[2, 3:0:-1, :, ...]
output[3, 0:5, :, ...] = input[3, 5:0:-1, :, ...]

output[0, 7:, :, ...] = input[0, 7:, :, ...]
output[1, 2:, :, ...] = input[1, 2:, :, ...]
output[2, 3:, :, ...] = input[2, 3:, :, ...]
```

```
output[3, 2:, :, ...] = input[3, 2:, :, ...]
```

使用例子：

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

import tensorflow as tf

sess = tf.Session()
input = tf.constant([[1, 2, 3, 4], [3, 4, 5, 6]], tf.int64)
seq_lengths = tf.constant([3, 2], tf.int64)
seq_dim = 1
output = tf.reverse_sequence(input, seq_lengths, seq_dim)
print sess.run(output)
sess.close()

# output
[[3 2 1 4]
 [4 3 5 6]]
```

输入参数：

- `input`: 一个`Tensor`，需要反转的数据。
- `seq_lengths`: 一个`Tensor`，数据类型是`int64`，数据长度是`input.dims(0)`，并且`max(seq_lengths) < input.dims(seq_dim)`。
- `seq_dim`: 一个`int`，确定需要翻转的维度。
- `name`:（可选）为这个操作取一个名字。

输出参数：

- 一个`Tensor`，数据类型和`input`相同，数据维度和`input`相同。

`tf.reverse(tensor, dims, name = None)`

解释：将指定维度中的数据进行翻转。

给定一个`tensor`和一个`bool`类型的`dims`，`dims`中的值为`False`或者`True`。如果`dims[i] == True`，那么就将`tensor`中这一维的数据进行翻转。

`tensor`最多只能有`8`个维度，并且`tensor`的秩必须和`dims`的长度相同，即`rank(tensor) == size(dims)`。

使用例子：

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

import tensorflow as tf

sess = tf.Session()
input_data = tf.constant([[
[
[ 0,  1,  2,  3],
[ 4,  5,  6,  7],
[ 8,  9, 10, 11]
],
[
[12, 13, 14, 15],
[16, 17, 18, 19],
[20, 21, 22, 23]
]
]])
print 'input_data shape : ', sess.run(tf.shape(input_data))
dims = tf.constant([False, False, False, True])
print sess.run(tf.reverse(input_data, dims))
print "============================"
dims = tf.constant([False, True, False, False])
print sess.run(tf.reverse(input_data, dims))
print "============================"
dims = tf.constant([False, False, True, False])
```

```
print sess.run(tf.reverse(input_data, dims))
sess.close()
```

输入参数：

- tensor: 一个Tensor，数据类型必须是以下之一：uint8，int8，int32，bool，float32或者float64，数据维度不超过8维。
- dims: 一个Tensor，数据类型是bool。
- name:（可选）为这个操作取一个名字。

输出参数：

- 一个Tensor，数据类型和tensor相同，数据维度和tensor相同。

---

tf.transpose(a, perm = None, name = 'transpose')

解释：将a进行转置，并且根据perm参数重新排列输出维度。

输出数据tensor的第i维将根据perm[i]指定。比如，如果perm没有给定，那么默认是perm = [n-1, n-2, ..., 0]，其中rank(a) = n。默认情况下，对于二维输入数据，其实就是常规的矩阵转置操作。

比如：

```
input_data.dims = (1, 4, 3)
perm = [1, 2, 0]

# 因为 output_data.dims[0] = input_data.dims[ perm[0] ]
# 因为 output_data.dims[1] = input_data.dims[ perm[1] ]
# 因为 output_data.dims[2] = input_data.dims[ perm[2] ]
# 所以得到 output_data.dims = (4, 3, 1)
output_data.dims = (4, 3, 1)
```

使用例子：

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

import tensorflow as tf

sess = tf.Session()
input_data = tf.constant([[1,2,3],[4,5,6]])
print sess.run(tf.transpose(input_data))
print sess.run(input_data)
print sess.run(tf.transpose(input_data, perm=[1,0]))
input_data = tf.constant([[[1,2,3],[4,5,6],[7,8,9],[10,11,12]]])
print 'input_data shape: ', sess.run(tf.shape(input_data))
output_data = tf.transpose(input_data, perm=[1, 2, 0])
print 'output_data shape: ', sess.run(tf.shape(output_data))
print sess.run(output_data)
sess.close()
```

输入参数：

- a: 一个Tensor。
- perm: 一个对于a的维度的重排列组合。
- name:（可选）为这个操作取一个名字。

输出参数：

- 一个经过翻转的Tensor。

---

tf.gather(params, indices, name = None)

解释：根据indices索引，从params中取对应索引的值，然后返回。

indices必须是一个整型的tensor，数据维度是常量或者一维。最后输出的数据维度是indices.shape + params.shape[1:]。

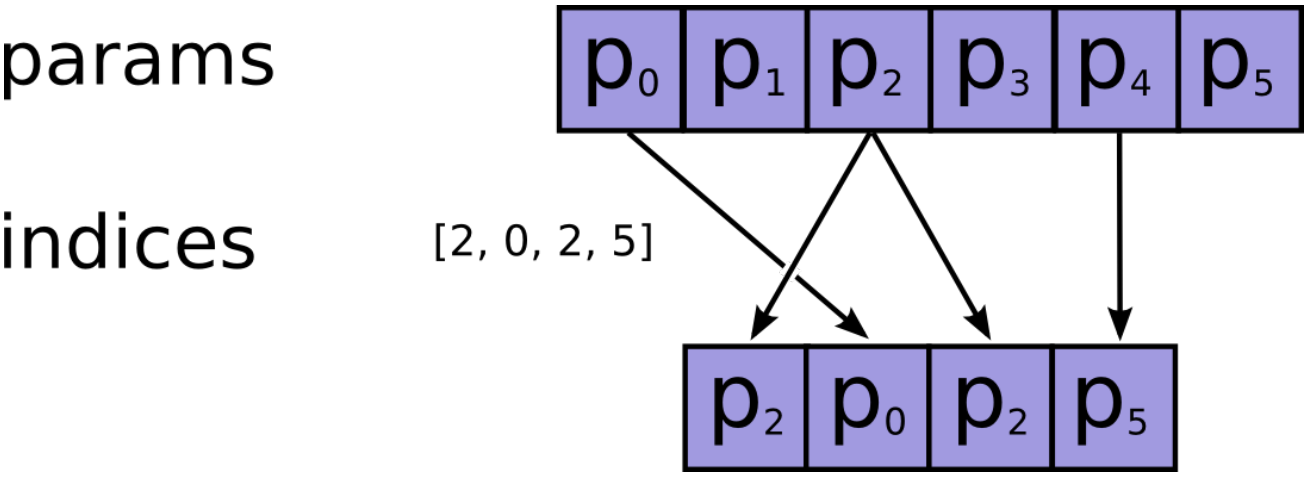
比如：

```
# Scalar indices
output[:, ..., :] = params[indices, :, ... :]

# Vector indices
output[i, :, ..., :] = params[indices[i], :, ... :]

# Higher rank indices
output[i, ..., j, :, ... :] = params[indices[i, ..., j], :, ..., :]
```

如果`indices`是一个从`0`到`params.shape[0]`的排列，即`len(indices) = params.shape[0]`，那么这个操作将把`params`进行重排列。



Gather

使用例子：

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

import tensorflow as tf

sess = tf.Session()
params = tf.constant([6, 3, 4, 1, 5, 9, 10])
indices = tf.constant([2, 0, 2, 5])
output = tf.gather(params, indices)
print sess.run(output)
sess.close()
```

输入参数：

- `params`: 一个`Tensor`。
- `indices`: 一个`Tensor`，数据类型必须是`int32`或者`int64`。
- `name`:（可选）为这个操作取一个名字。

输出参数：

- 一个`Tensor`，数据类型和`params`相同。

`tf.dynamic_partition(data, partitions, num_partitions, name = None)`

解释：根据从`partitions`中取得的索引，将`data`分割成`num_partitions`份。

我们先从`partitions.ndim` 中取出一个元祖`js`，那么切片`data[js, ...]`将成为输出数据`outputs[partitions[js]]`的一部分。我们将`js`按照字典序排列，即`js`里面的值为`(0, 0, ..., 1, 1, ..., 2, 2, ..., ..., num_partitions - 1, num_partitions - 1, ...)`。我们将`partitions[js] = i`的值放入`outputs[i]`。`outputs[i]`中的第一维对应于`partitions.values == i`的位置。更多细节如下：

```
outputs[i].shape = [sum(partitions == i)] + data.shape[partitions.ndim:]

outputs[i] = pack([data[js, ...] for js if partitions[js] == i])
```

data.shape must start with partitions.shape

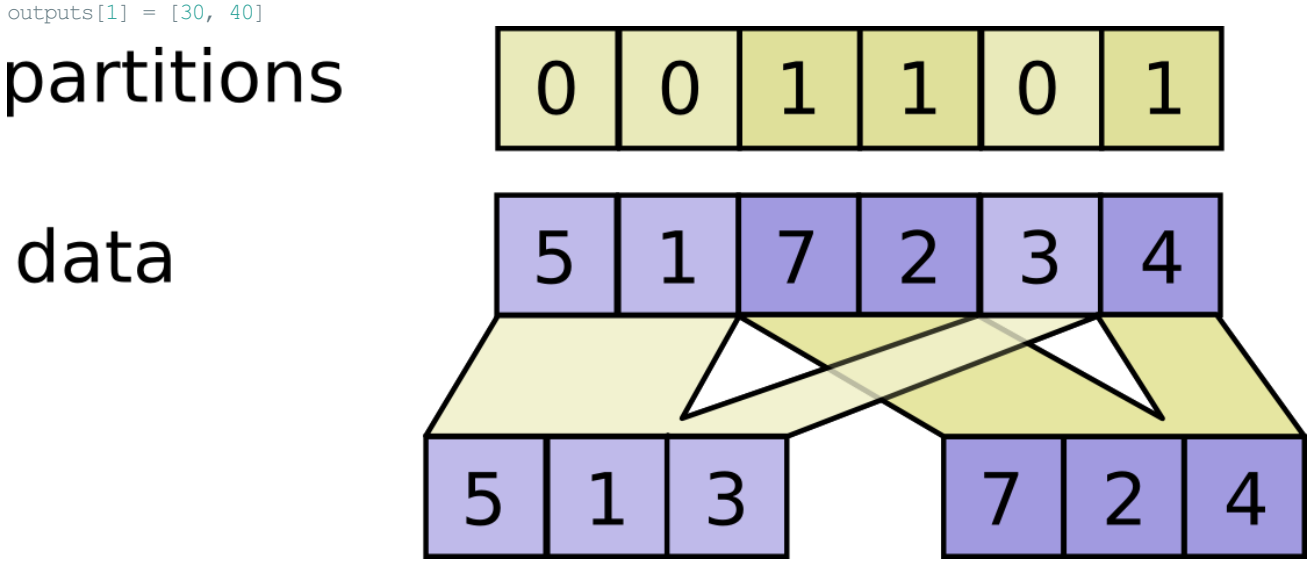
这句话不是很明白，说说自己的理解。

`data.shape(0)`必须和`partitions.shape(0)`相同，即`data.shape[0] == partitions.shape[0]`。

比如：

```
# Scalar partitions
partitions = 1
num_partitions = 2
data = [10, 20]
outputs[0] = []  # Empty with shape [0, 2]
outputs[1] = [[10, 20]]

# Vector partitions
partitions = [0, 0, 1, 1, 0]
num_partitions = 2
data = [10, 20, 30, 40, 50]
outputs[0] = [10, 20, 50]
outputs[1] = [30, 40]
```



DynamicPartition

使用例子：

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

import tensorflow as tf

sess = tf.Session()
params = tf.constant([6, 3, 4, 1, 5, 9, 10])
indices = tf.constant([2, 0, 2, 5])
output = tf.gather(params, indices)
print sess.run(output)
sess.close()
```

输入参数：

- `data`: 一个`Tensor`。
- `partitions`: 一个`Tensor`，数据类型必须是`int32`。任意数据维度，但其中的值必须是在范围`[0, num_partitions)`。
- `num_partitions`: 一个`int`，其值必须不小于`1`。输出的切片个数。
- `name`:（可选）为这个操作取一个名字。

输出参数：

- 一个数组`Tensor`，数据类型和`data`相同。

`tf.dynamic_stitch(indices, data, name = None)`

解释：这是一个交错合并的操作，我们根据`indices`中的值，将`data`交错合并，并且返回一个合并之后

的`tensor`。

如下构建一个合并的`tensor`：

```
merged[indices[m][i, ..., j], ...] = data[m][i, ..., j, ...]
```

其中，`m`是一个从`0`开始的索引。如果`indices[m]`是一个标量或者向量，那么我们可以得到更加具体的如下推导：

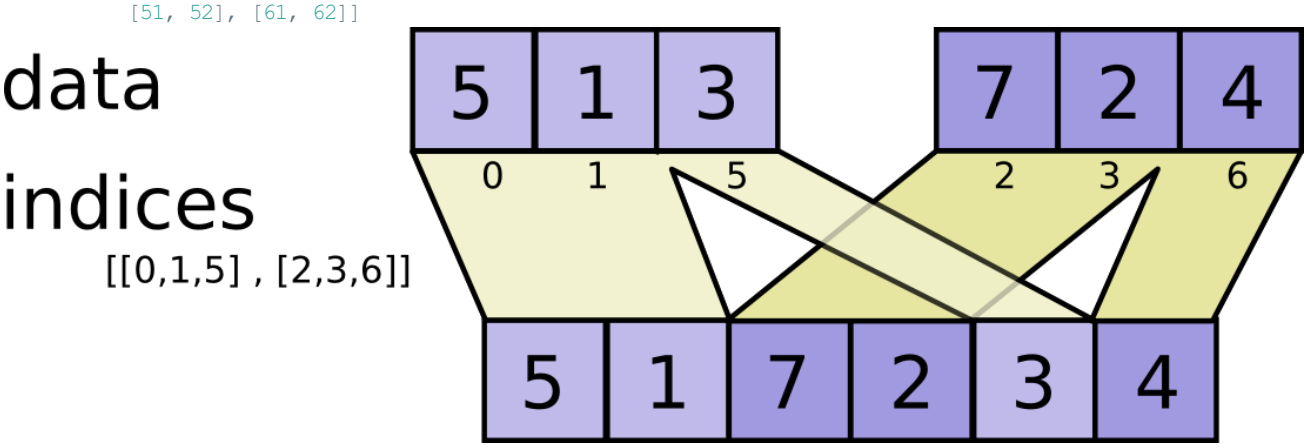
```
# Scalar indices
merged[indices[m], ...] = data[m][...]

# Vector indices
merged[indices[m][i], ...] = data[m][i, ...]
```

从上式的推导，我们也可以看出最终合并的数据是按照索引从小到大排序的。那么会产生两个问题：1）假设如果一个索引同时存在`indices[m][i]`和`indices[n][j]`中，其中`(m, i) < (n, j)`。那么，`data[n][j]`将作为最后被合并的值。2）假设索引越界了，那么缺失的位上面的值将被随机值给填补。

比如：

```
indices[0] = 6
indices[1] = [4, 1]
indices[2] = [[5, 2], [0, 3]]
data[0] = [61, 62]
data[1] = [[41, 42], [11, 12]]
data[2] = [[[51, 52], [21, 22]], [[1, 2], [31, 32]]]
merged = [[1, 2], [11, 12], [21, 22], [31, 32], [41, 42],
          [51, 52], [61, 62]]
```



DynamicStitch

使用例子：

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

import tensorflow as tf

sess = tf.Session()
indices = [6, [4, 1], [[5, 2], [0, 3]]]
data = [[61, 62], [[41, 42], [11, 12]], [[[51, 52], [21, 22]], [[1, 2], [31, 32]]]]
output = tf.dynamic_stitch(indices, data)
print sess.run(output)
# 缺少了第6，第7的位置，索引最后合并的数据中，这两个位置的值会被用随机数代替
indices = [8, [4, 1], [[5, 2], [0, 3]]]
output = tf.dynamic_stitch(indices, data)
# 第一个2被覆盖了，最后合并的数据是第二个2所指的位置
indices = [6, [4, 1], [[5, 2], [2, 3]]]
output = tf.dynamic_stitch(indices, data)
print sess.run(output)
print sess.run(output)
sess.close()
```

输入参数：

- `indices`: 一个列表，至少包含两`Tensor`，数据类型是`int32`。
- `data`: 一个列表，里面`Tensor`的个数和`indices`相同，并且拥有相同的数据类型。
- `name`:（可选）为这个操作取一个名字。

输出参数：

- 一个`Tensor`，数据类型和`data`相同。