

第1章 Paddle中的Tensor

```
In [1]: import paddle  
import numpy as np
```

1.张量的创建

1.1 从指定数据创建

从列表创建

```
In [2]: paddle.to_tensor([[1,2],[3,4]])
```

```
Out[2]: Tensor(shape=[2, 2], dtype=int64, place=Place(cpu), stop_gradient=True,  
[[1, 2],  
 [3, 4]])
```

从ndarray创建

```
In [3]: x1=np.array([[1,2],[3,4]])  
paddle.to_tensor(x1)
```

```
Out[3]: Tensor(shape=[2, 2], dtype=int32, place=Place(cpu), stop_gradient=True,  
[[1, 2],  
 [3, 4]])
```

将tensor转回list和ndarray

```
In [4]: x2=paddle.to_tensor([[1,2],[3,4]])  
x2.tolist(),x2.numpy()
```

```
Out[4]: ([[1, 2], [3, 4]],  
 array([[1, 2],  
 [3, 4]], dtype=int64))
```

1.2 从指定形状创建

```
In [5]: x3=paddle.zeros([2, 3])      # 创建数据全为 0, 形状为 [m, n] 的 Tensor  
x4=paddle.ones([2, 3])      # 创建数据全为 1, 形状为 [m, n] 的 Tensor  
x5=paddle.full([2, 3], 10)    # 创建数据全为 10, 形状为 [m, n] 的 Tensor  
x6=paddle.empty([2,3])  
x3,x4,x5,x6
```

```
Out[5]: (Tensor(shape=[2, 3], dtype=float32, place=Place(cpu), stop_gradient=True,  
[[0., 0., 0.],  
 [0., 0., 0.]]),  
 Tensor(shape=[2, 3], dtype=float32, place=Place(cpu), stop_gradient=True,  
[[1., 1., 1.],  
 [1., 1., 1.]]),  
 Tensor(shape=[2, 3], dtype=float32, place=Place(cpu), stop_gradient=True,  
[[10., 10., 10.],  
 [10., 10., 10.]]),  
 Tensor(shape=[2, 3], dtype=float32, place=Place(cpu), stop_gradient=True,  
[[0., 0., 0.],  
 [0., 0., 0.]]))
```

1.3 从指定区间创建

paddle.arange(start, end, step)
创建以步长step均匀分隔区间[start, end]的Tensor
paddle.linspace(start, stop, num)
创建以元素个数num均匀分隔区间[start, stop]的Tensor

```
In [6]: paddle.arange(start=1, end=5, step=1)
```

```
Out[6]: Tensor(shape=[4], dtype=int64, place=Place(cpu), stop_gradient=True,  
[1, 2, 3, 4])
```

```
In [7]: paddle.linspace(1,5,5)
```

```
Out[7]: Tensor(shape=[5], dtype=float32, place=Place(cpu), stop_gradient=True,  
[1., 2., 3., 4., 5.])
```

1.4 随机矩阵

注：随机种子只在同一个单元格中起作用

0~1上的均匀分布与正太分布

```
In [8]: paddle.seed(102)  
paddle.rand(shape=[2, 3])
```

```
Out[8]: Tensor(shape=[2, 3], dtype=float32, place=Place(cpu), stop_gradient=True,  
[[0.57241243, 0.45361701, 0.77573997],  
 [0.01048208, 0.46195969, 0.10416761]])
```

```
In [9]: paddle.randn(shape=[2, 3])
```

```
Out[9]: Tensor(shape=[2, 3], dtype=float32, place=Place(cpu), stop_gradient=True,  
[[ -1.01171708, -2.08565593, 0.18424854],  
 [ 0.00328017, -1.80662513, 0.47235256]])
```

任意区间上的均匀分布与正态分布

```
In [10]: paddle.uniform(min=5,max=10,shape=[2,3])
```

```
Out[10]: Tensor(shape=[2, 3], dtype=float32, place=Place(cpu), stop_gradient=True,  
[[9.23183632, 7.80339479, 8.69056320],  
 [6.81550121, 6.40685844, 6.66369724]])
```

```
In [11]: paddle.normal(mean=5,std=1,shape=[2,3])
```

```
Out[11]: Tensor(shape=[2, 3], dtype=float32, place=Place(cpu), stop_gradient=True,  
[[6.31953335, 6.15016079, 5.81440353],  
 [5.69808245, 6.92263985, 4.05796194]])
```

1.5 其他情况

复制矩阵

```
In [12]: paddle.clone(x6)
```

```
Out[12]: Tensor(shape=[2, 3], dtype=float32, place=Place(cpu), stop_gradient=True,  
[[0., 0., 0.],  
 [0., 0., 0.]])
```

形状一样，指定元素填充

```
In [13]: paddle.zeros_like(x6),paddle.ones_like(x6),paddle.full_like(x6,8),paddle.empty_like(x6)
```

```
Out[13]: (Tensor(shape=[2, 3], dtype=float32, place=Place(cpu), stop_gradient=True,  
[[0., 0., 0.],  
 [0., 0., 0.]]),  
 Tensor(shape=[2, 3], dtype=float32, place=Place(cpu), stop_gradient=True,  
[[1., 1., 1.],  
 [1., 1., 1.]]),  
 Tensor(shape=[2, 3], dtype=float32, place=Place(cpu), stop_gradient=True,  
[[8., 8., 8.],  
 [8., 8., 8.]]),  
 Tensor(shape=[2, 3], dtype=float32, place=Place(cpu), stop_gradient=True,  
[[0., 0., 0.],  
 [0., 0., 0.]]))
```

2.张量的访问

2.1 索引与切片

一维张量

```
In [14]: x1=paddle.to_tensor([3,2,12,24,5,60,7,8,9])
x1
Out[14]: Tensor(shape=[9], dtype=int64, place=Place(cpu), stop_gradient=True,
[3 , 2 , 12, 24, 5 , 60, 7 , 8 , 9 ])
Tensor[头:尾:步长]

In [15]: x1[1:3],x1[1:-2]
Out[15]: (Tensor(shape=[2], dtype=int64, place=Place(cpu), stop_gradient=True,
[2 , 12]),
Tensor(shape=[6], dtype=int64, place=Place(cpu), stop_gradient=True,
[2 , 12, 24, 5 , 60, 7 ]))

In [16]: # 通过控制头和步长实现倒序
x1[1::-1]
Out[16]: Tensor(shape=[9], dtype=int64, place=Place(cpu), stop_gradient=True,
[9 , 8 , 7 , 60, 5 , 24, 12, 2 , 3 ])
Tensor[列表]

In [17]: x1[[1,2,4]]
Out[17]: Tensor(shape=[3], dtype=int64, place=Place(cpu), stop_gradient=True,
[2 , 12, 5 ])

In [18]: # 利用列表索引实现排序
index=paddle.argsort(-x1) # 获得x1从大到小的元素的索引
index,x1[index]
Out[18]: (Tensor(shape=[9], dtype=int64, place=Place(cpu), stop_gradient=True,
[5, 3, 2, 8, 7, 6, 4, 0, 1]),
Tensor(shape=[9], dtype=int64, place=Place(cpu), stop_gradient=True,
[60, 24, 12, 9 , 8 , 7 , 5 , 3 , 2 ]))
```

二维张量

```
In [19]: x2=paddle.to_tensor([[1,2,3],[4,5,6],[9,8,7],[44,88,66]],dtype=paddle.float32)
x2
Out[19]: Tensor(shape=[4, 3], dtype=float32, place=Place(cpu), stop_gradient=True,
[[1. , 2. , 3. ],
[4. , 5. , 6. ],
[9. , 8. , 7. ],
[44., 88., 66.]])
把第1维的每个行向量看成一个点，按照一维tensor的规则取出行元素
Tensor[列表]取出指定行
Tensor[头:尾:步长]取出指定行

In [20]: x2[[0,3]],x2[0:3:2]
Out[20]: (Tensor(shape=[2, 3], dtype=float32, place=Place(cpu), stop_gradient=True,
[[1. , 2. , 3. ],
[44., 88., 66.]]),
Tensor(shape=[2, 3], dtype=float32, place=Place(cpu), stop_gradient=True,
[[1., 2., 3.],
[9., 8., 7.]]))
Tensor[行, 列]
行和列的取出规则同一维情况

In [21]: x2[:,2,:-1]
Out[21]: Tensor(shape=[2, 3], dtype=float32, place=Place(cpu), stop_gradient=True,
[[3., 2., 1.],
[6., 5., 4.]])
批量索引，比如[0,1]取出第0, 1行，[2,3]取出第2, 3行，那么[[0,1],[2,3]]会将取出的两个矩阵堆叠
注意这里批量所以不能直接使用[[0,1],[2,3]]，要转换为tensor([[0,1],[2,3]])
```

```
In [57]: qq=torch.randn(size=[8,3])
qq
Out[57]: tensor([[-1.7424,  1.1238,  1.7252],
[-1.1183, -0.1578,  0.2605],
[ 0.1480,  0.7043,  0.2383],
[-1.8373, -1.1922, -0.1603],
[-0.6661, -0.3062, -0.3948],
[ 0.3725, -0.6756, -2.0834],
[ 0.6882, -0.5547,  1.5902],
[ 1.0456,  0.7099,  0.9883]])
```

```
In [59]: qq[torch.tensor([[0,1],[2,3]])]
```

```
Out[59]: tensor([[[-1.7424,  1.1238,  1.7252],
[-1.1183, -0.1578,  0.2605]],
[[ 0.1480,  0.7043,  0.2383],
[-1.8373, -1.1922, -0.1603]]])
```

Bool索引

```
In [22]: x2
Out[22]: Tensor(shape=[4, 3], dtype=float32, place=Place(cpu), stop_gradient=True,
[[1. , 2. , 3. ],
[4. , 5. , 6. ],
[9. , 8. , 7. ],
[44., 88., 66.]])
```

取出指定标签行
当bool索引是1维张量时，取出为True的行

```
In [23]: labels=paddle.to_tensor([1,0,1,0])
labels==paddle.to_tensor([1])
Out[23]: Tensor(shape=[4], dtype=bool, place=Place(cpu), stop_gradient=True,
[True , False, True , False])
```

```
In [24]: x2[labels==paddle.to_tensor([1])]
Out[24]: Tensor(shape=[2, 3], dtype=float32, place=Place(cpu), stop_gradient=True,
[[1., 2., 3.],
[9., 8., 7.]]))
```

当bool索引为2维张量时，取出相应元素，返回1维张量

```
In [25]: (x2<20)&(x>2)
Out[25]: Tensor(shape=[4, 3], dtype=bool, place=Place(cpu), stop_gradient=True,
[[False, False, True ],
[True , True , True ],
[True , True , True ],
[False, False, False]])
```

```
In [26]: x2[(x2<20)&(x>2)]
Out[26]: Tensor(shape=[7], dtype=float32, place=Place(cpu), stop_gradient=True,
[3., 4., 5., 6., 9., 8., 7.])
```

使用paddle.where(条件, x, y)可以把x矩阵中的元素按条件变成y中的元素
注意, 这里x, y形状必须一致, 与numpy, torch的api有所不同

```
In [27]: y=paddle.zeros_like(x2)
In [28]: paddle.where(x2>6,x2,y)
Out[28]: Tensor(shape=[4, 3], dtype=float32, place=Place(cpu), stop_gradient=True,
[[0., 0., 0.],
[0., 0., 0.],
[9., 8., 7.],
[44., 88., 66.]])
```

高级操作: 掩码与填充

经典例子: 设计bool矩阵实现掩码操作

假如矩阵x2的每一行都是一个句子, 一共4句话, 真实长度分别为1, 2, 2, 3

请对每句话实现PAD掩码, 即多余的长度部分置0

条件矩阵: 即mask_condition矩阵, 用[0,1,2,3]<[1,1,1,1]获得[True,False,False,False], 即第一句话的掩码条件, 其他几句话雷同

x矩阵: 即句子

y句子: 即PAD

```
In [29]: mask_condition=paddle.arange(x2.shape[1])<paddle.to_tensor([[1],[2],[2],[3]])
mask_condition
```

```
Out[29]: Tensor(shape=[4, 3], dtype=bool, place=Place(cpu), stop_gradient=True,
[[True, False, False],
[True, True, False],
[True, True, False],
[True, True, True]])
```

```
In [30]: PAD=paddle.zeros(x2.shape)
PAD
```

```
Out[30]: Tensor(shape=[4, 3], dtype=float32, place=Place(cpu), stop_gradient=True,
[[0., 0., 0.],
[0., 0., 0.],
[0., 0., 0.],
[0., 0., 0.]])
```

```
In [31]: mask=paddle.where(mask_condition,x2,PAD)
mask
```

```
Out[31]: Tensor(shape=[4, 3], dtype=float32, place=Place(cpu), stop_gradient=True,
[[1., 0., 0.],
[4., 5., 0.],
[9., 8., 0.],
[44., 88., 66.]])
```

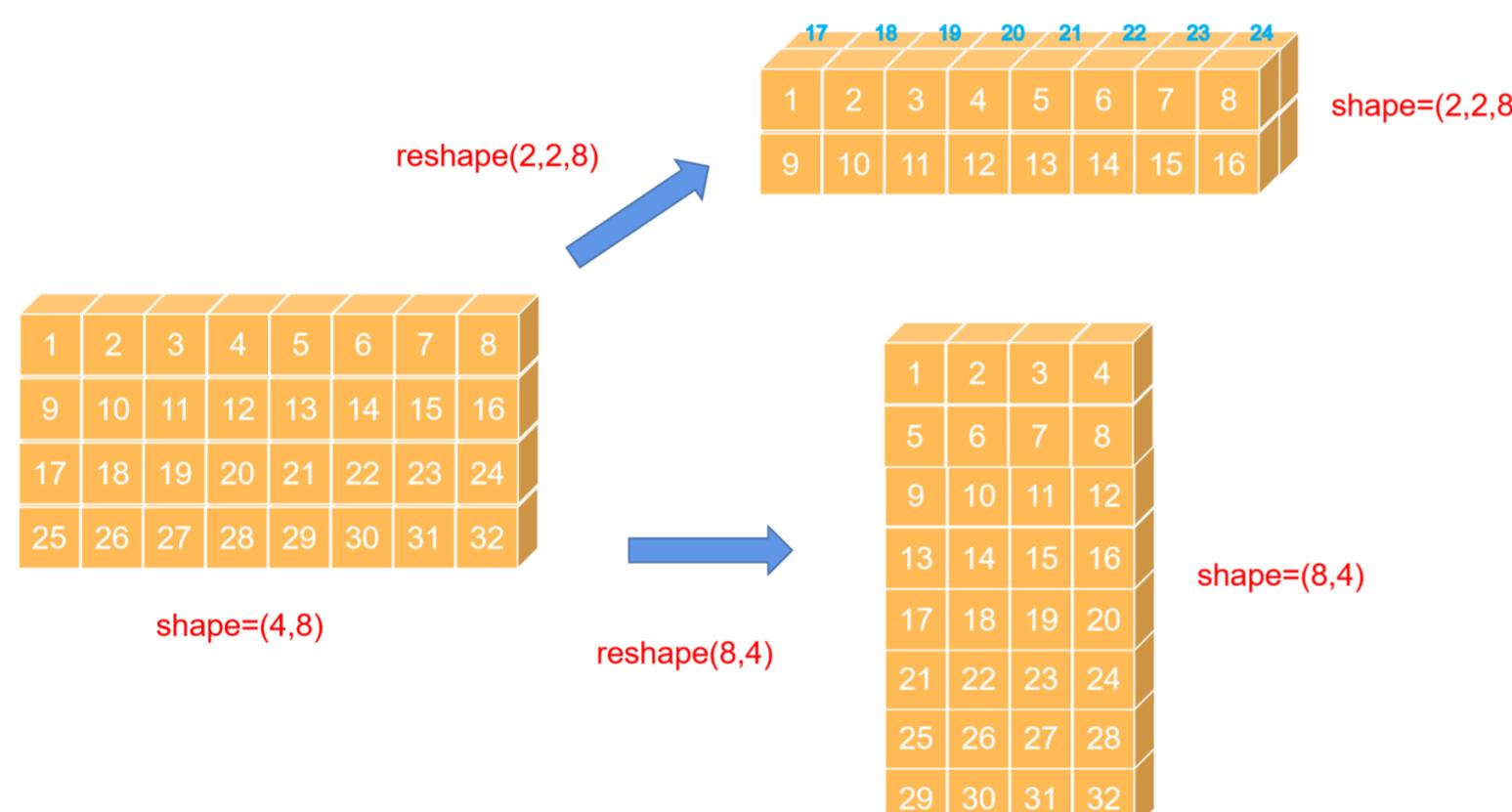
3.张量的变形

3.1 Reshape

reshape的两个常见用法:

- (1) 把某一维切分成两个维度
- (2) 重整二维矩阵的形状

变形规则: 把原张量所有元素按顺序拉平, 再按顺序填充到变形后的张量中 (可以看成魔方)



先做个拉平后的张量, 然后依次变形

```
In [32]: x=paddle.arange(start=1, end=33, step=1,dtype=paddle.float32)
x
Out[32]: Tensor(shape=[32], dtype=float32, place=Place(cpu), stop_gradient=True,
[1., 2., 3., 4., 5., 6., 7., 8.,
15., 16., 17., 18., 19., 20., 21., 22., 23., 24., 25., 26., 27., 28.,
29., 30., 31., 32.])
```

```
In [33]: x.reshape([4,8])
Out[33]: Tensor(shape=[4, 8], dtype=float32, place=Place(cpu), stop_gradient=True,
[[1., 2., 3., 4., 5., 6., 7., 8.],
[9., 10., 11., 12., 13., 14., 15., 16.],
[17., 18., 19., 20., 21., 22., 23., 24.],
[25., 26., 27., 28., 29., 30., 31., 32.]])
```

```
In [34]: x.reshape([2,2,8])
Out[34]: Tensor(shape=[2, 2, 8], dtype=float32, place=Place(cpu), stop_gradient=True,
[[[1., 2., 3., 4., 5., 6., 7., 8.],
[9., 10., 11., 12., 13., 14., 15., 16.],
[17., 18., 19., 20., 21., 22., 23., 24.],
[25., 26., 27., 28., 29., 30., 31., 32.]]])
```

```
In [35]: x.reshape([8,4])
Out[35]: Tensor(shape=[8, 4], dtype=float32, place=Place(cpu), stop_gradient=True,
[[1., 2., 3., 4.],
[5., 6., 7., 8.],
[9., 10., 11., 12.],
[13., 14., 15., 16.],
[17., 18., 19., 20.],
[21., 22., 23., 24.],
[25., 26., 27., 28.],
[29., 30., 31., 32.]])
```

再从4x8的矩阵出发, 变形成另外两种

由于原理是按照拉平后的向量再重整形状的, 所以结果应该与上面保持一致

```
In [36]: x1=x.reshape([4,8])
x1
Out[36]: Tensor(shape=[4, 8], dtype=float32, place=Place(cpu), stop_gradient=True,
[[1., 2., 3., 4.],
[9., 10., 11., 12., 13., 14., 15., 16.],
[17., 18., 19., 20., 21., 22., 23., 24.],
[25., 26., 27., 28., 29., 30., 31., 32.]])
```

```
In [37]: x1.reshape([2,2,8])

```

```
Out[37]: Tensor(shape=[2, 2, 8], dtype=float32, place=Place(cpu), stop_gradient=True,
[[[1., 2., 3., 4., 5., 6., 7., 8.],
[9., 10., 11., 12., 13., 14., 15., 16.]],
[[17., 18., 19., 20., 21., 22., 23., 24.],
[25., 26., 27., 28., 29., 30., 31., 32.]])
```

In [38]: `x1.reshape([8,4])`

```
Out[38]: Tensor(shape=[8, 4], dtype=float32, place=Place(cpu), stop_gradient=True,
[[1., 2., 3., 4.],
[5., 6., 7., 8.],
[9., 10., 11., 12.],
[13., 14., 15., 16.],
[17., 18., 19., 20.],
[21., 22., 23., 24.],
[25., 26., 27., 28.],
[29., 30., 31., 32.]])
```

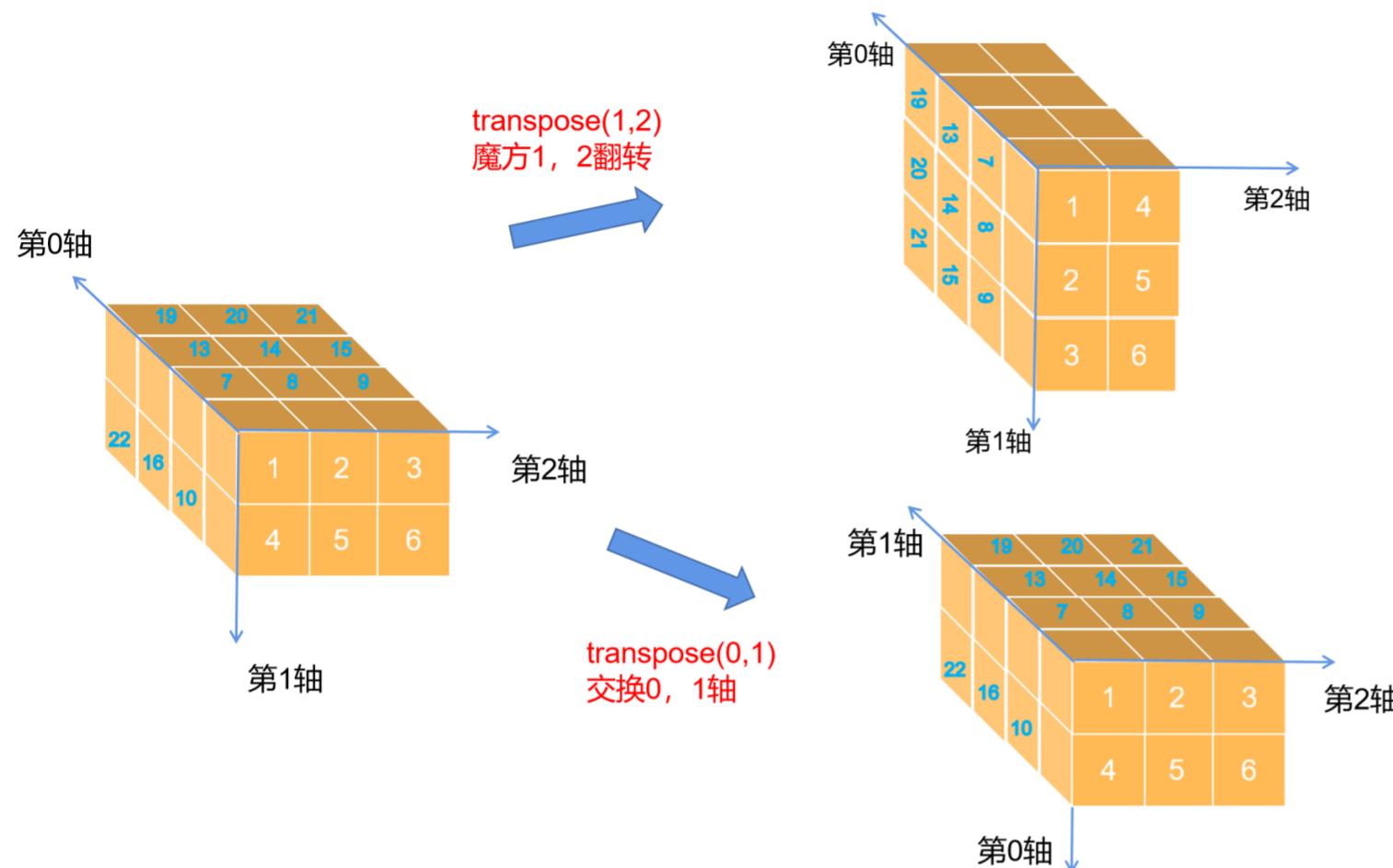
3.2 Transpose

变形规则：

有两种看法，

一是可以看成魔方的旋转90度，注意旋转后会有一根轴方向跟我们画的标准魔方轴的方向不一样，我们需要转换一下
另一种是把轴进行交换，观察角度需要切换到标准角度（从标准角度观察，0轴是前后轴，1轴是上下轴，2轴是左右轴），以下图为例，
标准角度就是看见1, 2轴所在平面的角度，对于transpose(0,1)后的标准角度是从上方往下看

注意：这个接口跟其他框架接口的使用方法也有点不同



In [39]: `x2=paddle.arange(start=1, end=25, step=1,dtype=paddle.float32).reshape([4,2,3])
x2,x2.transpose([0,2,1]),x2.transpose([1,0,2])`

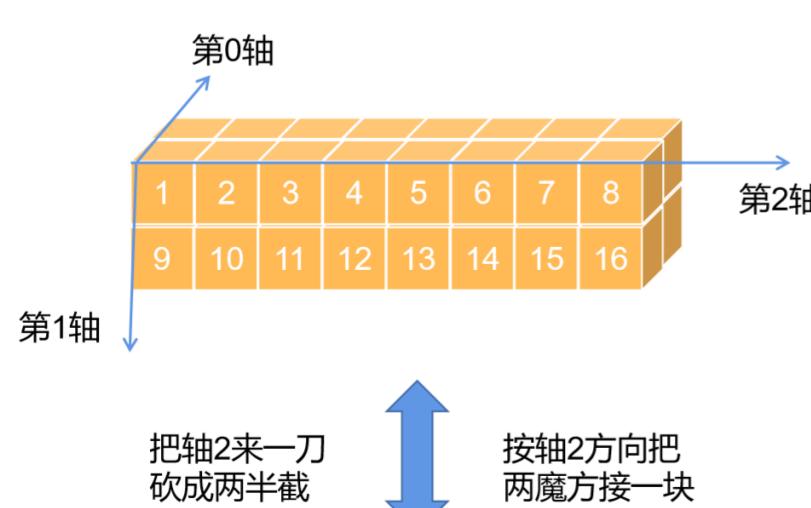
```
Out[39]: (Tensor(shape=[4, 2, 3], dtype=float32, place=Place(cpu), stop_gradient=True,
[[[1., 2., 3.],
[4., 5., 6.]],
[[7., 8., 9.],
[10., 11., 12.]],
[[13., 14., 15.],
[16., 17., 18.]],
[[19., 20., 21.],
[22., 23., 24.]]]),
Tensor(shape=[4, 3, 2], dtype=float32, place=Place(cpu), stop_gradient=True,
[[[1., 4.],
[2., 5.],
[3., 6.]],
[[7., 10.],
[8., 11.],
[9., 12.]],
[[13., 16.],
[14., 17.],
[15., 18.]],
[[19., 22.],
[20., 23.],
[21., 24.]]]),
Tensor(shape=[2, 4, 3], dtype=float32, place=Place(cpu), stop_gradient=True,
[[[1., 2., 3.],
[7., 8., 9.],
[13., 14., 15.],
[19., 20., 21.]],
[[4., 5., 6.],
[10., 11., 12.],
[16., 17., 18.],
[22., 23., 24.]]]))
```

3.3 拼接与切割

变形规则：

就是把魔方来一刀变成两个魔方，或者把两个魔方粘在一起，
原魔方与操作后得到的魔方维数（指的是轴的数量）不变

chunk(砍成几块, axis=某根轴), 对标字符串里面的split
concat((魔方1, 魔方2),axis=某根轴), 对标列表里面的+



In [40]: `x3=paddle.arange(start=1, end=33, step=1,dtype=paddle.float32).reshape([2,2,8])`

```
x3

Out[40]: Tensor(shape=[2, 2, 8], dtype=float32, place=Place(cpu), stop_gradient=True,
[[[1., 2., 3., 4., 5., 6., 7., 8.],
 [9., 10., 11., 12., 13., 14., 15., 16.]]],

In [41]: a,b=x3.chunk(2,axis=2) # pytorch里面需要写成dim=2
a,b

Out[41]: (Tensor(shape=[2, 2, 4], dtype=float32, place=Place(cpu), stop_gradient=True,
[[[1., 2., 3., 4.],
 [9., 10., 11., 12.]]],

[[17., 18., 19., 20.],
 [25., 26., 27., 28.]]]),
Tensor(shape=[2, 2, 4], dtype=float32, place=Place(cpu), stop_gradient=True,
[[[5., 6., 7., 8.],
 [13., 14., 15., 16.]]],

[[21., 22., 23., 24.],
 [29., 30., 31., 32.]]])))

In [42]: c=paddle.concat((a,b),axis=2)
c

Out[42]: Tensor(shape=[2, 2, 8], dtype=float32, place=Place(cpu), stop_gradient=True,
[[[1., 2., 3., 4., 5., 6., 7., 8.],
 [9., 10., 11., 12., 13., 14., 15., 16.]]],

[[17., 18., 19., 20., 21., 22., 23., 24.],
 [25., 26., 27., 28., 29., 30., 31., 32.]]]))
```

3.3 维数增减 unsqueeze()&squeeze()

先说一下这个操作的目的，比如维数增这个操作，假如我有一堆矩阵，我需要增加一个维度，也就是前后方向的那个维度，把这些矩阵装进去，此时我只要把每个矩阵变成一个3维魔方，然后在前后维上拼接在一起即可
增加维数从张量角度来看，就是加一层括号，我们把张量从外向内的括号分别编号0, 1, 2，这个编号正是0, 1, 2维的括号，既然括号有了对应编号，删除相应维就是删除这个编号对应的括号，增加一维，也是从外到内数到那一层加括号，具体请看下面的例子

```
In [43]: x4=paddle.to_tensor([[1],[2],[3],[4],[5],[6]])
x4.shape,x4

Out[43]: ([2, 3, 1],
Tensor(shape=[2, 3, 1], dtype=int64, place=Place(cpu), stop_gradient=True,
[[[1],
 [2],
 [3]],

[[4],
 [5],
 [6]]])))

先把第二维的括号去掉

In [44]: x5=x4.squeeze(2)
x5.shape,x5

Out[44]: ([2, 3],
Tensor(shape=[2, 3], dtype=int64, place=Place(cpu), stop_gradient=True,
[[1, 2, 3],
 [4, 5, 6]]))

再把x5的第0维加个括号

In [45]: x6=x5.unsqueeze(0)
x6.shape,x6

Out[45]: ([1, 2, 3],
Tensor(shape=[1, 2, 3], dtype=int64, place=Place(cpu), stop_gradient=True,
[[[1, 2, 3],
 [4, 5, 6]]])))

创建另一个形状一样的矩阵，并跟x5在深度方向上进行拼接

In [46]: x7=paddle.full_like(x5,100.).unsqueeze(0)
x7

Out[46]: Tensor(shape=[1, 2, 3], dtype=int64, place=Place(cpu), stop_gradient=True,
[[[100, 100, 100],
 [100, 100, 100]]])

In [47]: paddle.concat((x6,x7),axis=0)

Out[47]: Tensor(shape=[2, 2, 3], dtype=int64, place=Place(cpu), stop_gradient=True,
[[[1, 2, 3],
 [4, 5, 6],
 [100, 100, 100],
 [100, 100, 100]]])
```

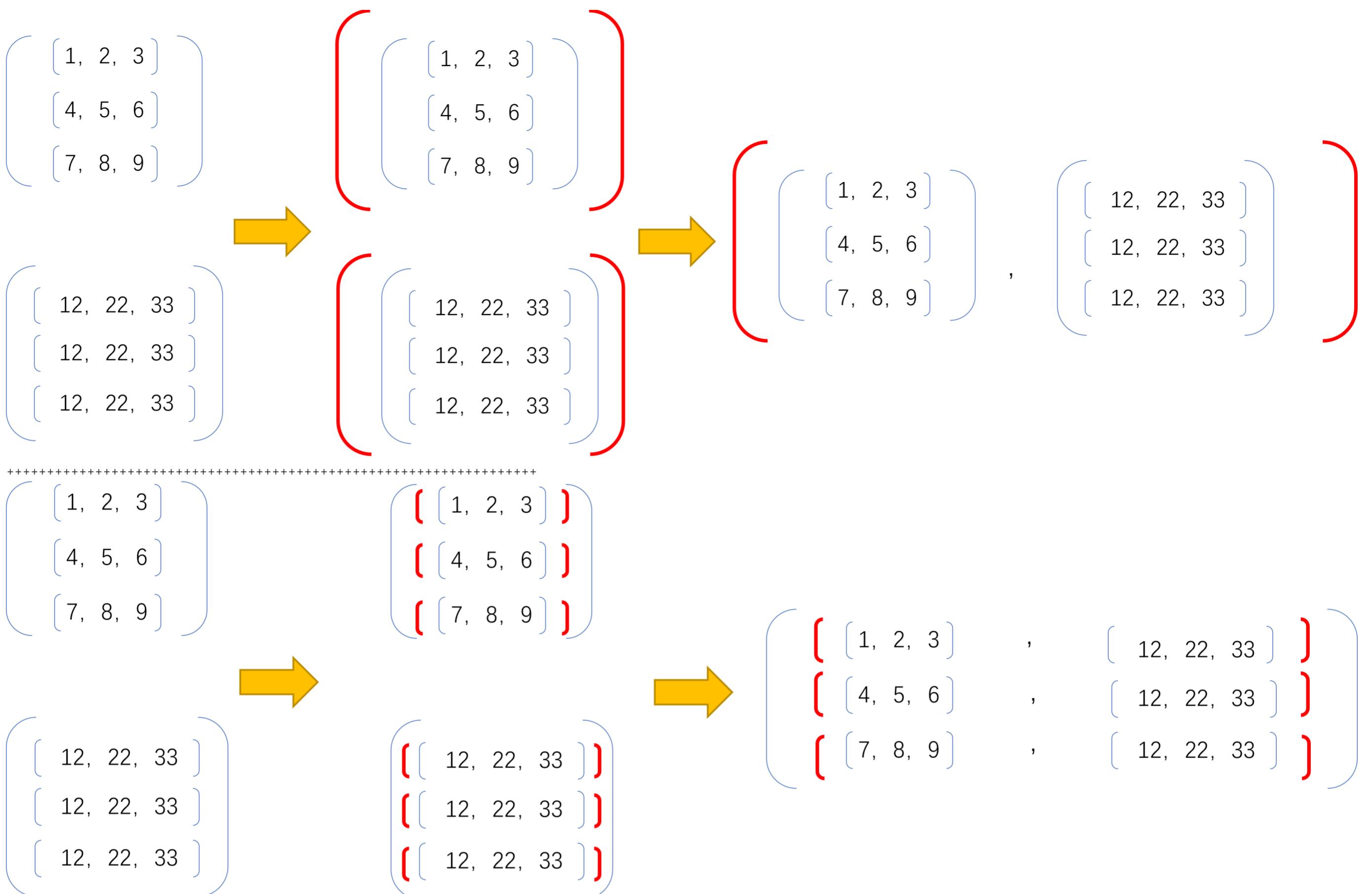
3.4 矩阵堆叠 stack()

就是上面先对矩阵增加维数，再拼接
注：此操作维数会发生改变，而concat操作维数并不会改变

```
In [48]: A = paddle.to_tensor([[1, 2, 3],
 [4, 5, 6],
 [7, 8, 9]])
B = paddle.to_tensor([[12, 22, 33],
 [44, 55, 66],
 [77, 88, 99]])
A,B

Out[48]: (Tensor(shape=[3, 3], dtype=int64, place=Place(cpu), stop_gradient=True,
[[1, 2, 3],
 [4, 5, 6],
 [7, 8, 9]]),
Tensor(shape=[3, 3], dtype=int64, place=Place(cpu), stop_gradient=True,
[[12, 22, 33],
 [44, 55, 66],
 [77, 88, 99]]))

把魔方画出来，然后按下面的方式堆叠：
axis=0: 第0维加个红[], 然后把红[]进行concat连接
axis=1: 第1维加个红[], 然后把红[]进行concat连接
axis=2: 第2维加个红[], 然后把红[]进行concat连接
```



In [49]: `paddle.stack((A,B),axis=0),paddle.stack((A,B),axis=1),paddle.stack((A,B),axis=2)`

```
Out[49]: (Tensor(shape=[2, 3, 3], dtype=int64, place=Place(cpu), stop_gradient=True,
[[[1 , 2 , 3 ],
[4 , 5 , 6 ],
[7 , 8 , 9 ]],

[[[12, 22, 33],
[44, 55, 66],
[77, 88, 99]]]),

Tensor(shape=[3, 2, 3], dtype=int64, place=Place(cpu), stop_gradient=True,
[[[1 , 2 , 3 ],
[12, 22, 33]],

[[4 , 5 , 6 ],
[44, 55, 66]],

[[7 , 8 , 9 ],
[77, 88, 99]]]),

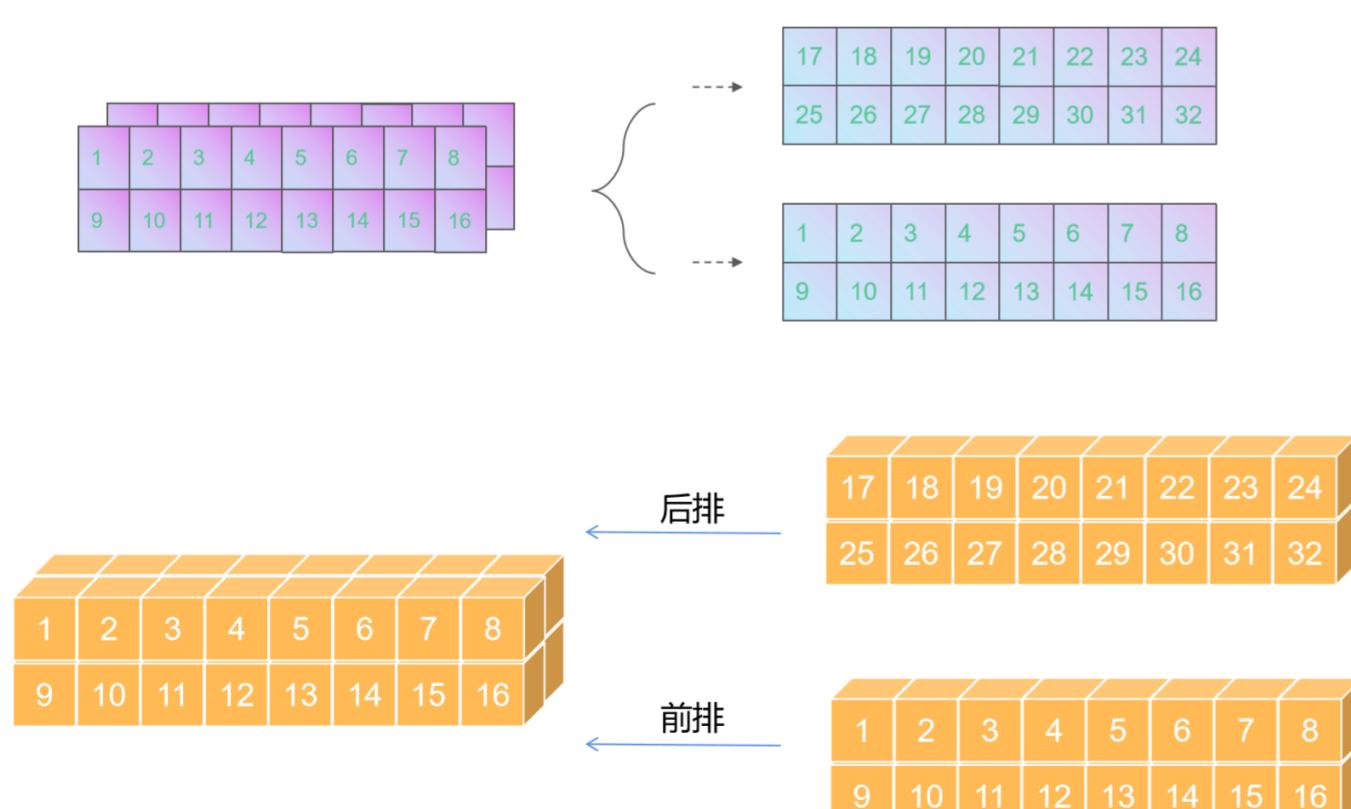
Tensor(shape=[3, 3, 2], dtype=int64, place=Place(cpu), stop_gradient=True,
[[[1 , 12],
[2 , 22],
[3 , 33]],

[[4 , 44],
[5 , 55],
[6 , 66]],

[[7 , 77],
[8 , 88],
[9 , 99]]]))
```

4.张量的并行乘法

由于这个和张量的变形密切相关，所以单独拎出来写几个范例



4.1 数据的表示

如上图，paddle中的三维张量BxD我们用一个三维魔方形象化表示。

其中B是batch size维，LxD是我们的数据矩阵，后面我们以数据是序列为例进行说明。

所以L代表句子的长度，也就是句子中单词的数量，D代表每个单词的word embedding。

比如要处理的第一句话是["上","海"]，上=[1,2,3,4,5,6,7,8]，海=[9,10,11,12,13,14,15,16]。

那么上面魔方的第一排中的第一行表示“上”，第一排第二行表示“海”，第一张矩阵就是一个句子，一个序列。

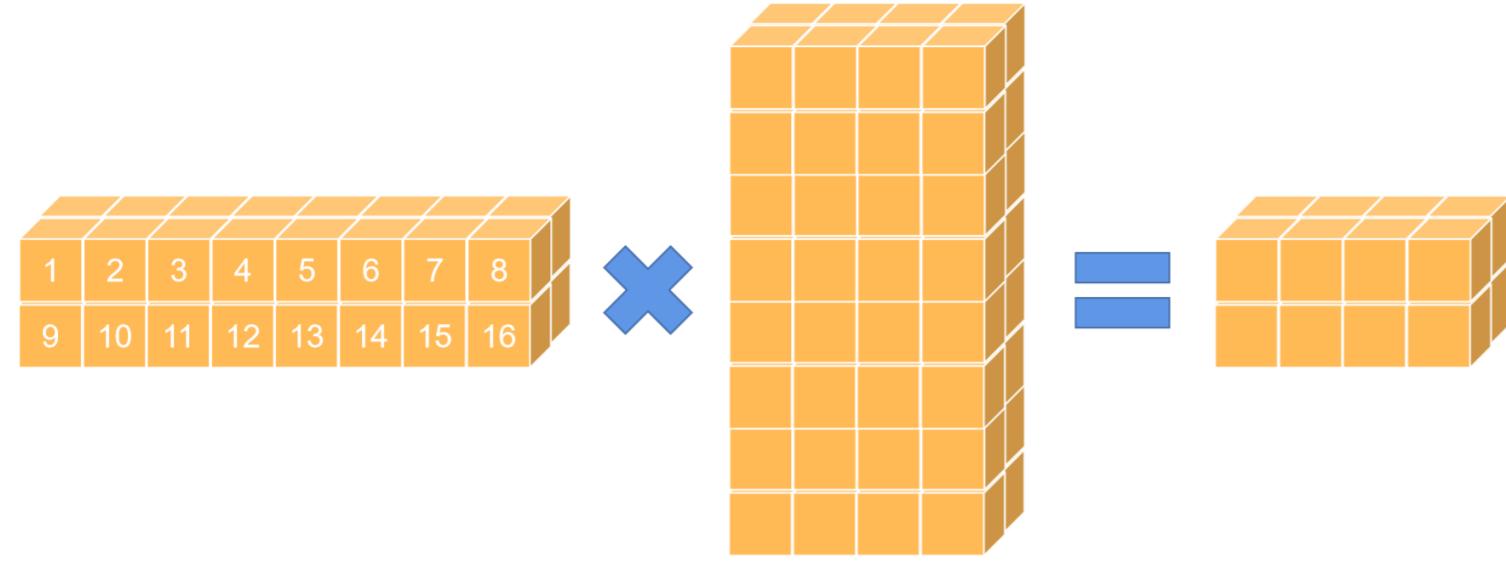
由于paddle可以对batch size维做并行计算，所以我们在后排的矩阵中放置其他句子。

In [50]: `x=paddle.arange(start=1, end=33, step=1,dtype=paddle.float32).reshape([2,2,8])`
x

```
Out[50]: Tensor(shape=[2, 2, 8], dtype=float32, place=Place(cpu), stop_gradient=True,
[[[1., 2., 3., 4., 5., 6., 7., 8.],
[9., 10., 11., 12., 13., 14., 15., 16.]],
[[17., 18., 19., 20., 21., 22., 23., 24.],
[25., 26., 27., 28., 29., 30., 31., 32.]]])
```

4.2 张量乘法的并行计算

情形一：多个不同的矩阵并行乘以同一个矩阵（反过来讲也一样）



设W是一个8x4的矩阵，可以把一个8维的单词向量降维到4维，那么一张句子矩阵乘以W，就变成了另一张矩阵，是句子在4维空间中的表示，第一行还是“上”，第二行还是“海”，只不过编码变成了4维。假如还有另一个句子["狗","蛋"]需要相同的处理，我们不需要用for把上海跟W乘完后，再拿狗蛋跟W乘，因为这样如果有100个句子矩阵要跟W相乘，这样for让句子排队等着跟W相乘很浪费时间，于是可以这样做，我们先把句子的一张张句子按前后顺序排成一排，上图中已经展示，然后把W复制多份，也如法炮制这么排成一排，接着这两个魔方相乘，让每一排的句子矩阵和每一排的W同时相乘，获得一个魔方，这个魔方的每一排就是每一排相乘的结果。如何实现：paddle的底层已经实现了“前后排”这个维度，也就是batch size这个维度并行计算的魔方矩阵乘法，第一个魔方就是我们的多个句子的魔方，第二个魔方由于前后排都是W，所以通过广播机制，W会自动转成上述魔方，我们直接使用W即可。

先不利用并行计算，直接手动把每张矩阵和W相乘

```
In [51]: paddle.seed(102)
seq1=x[0,:,:]
seq2=x[1,:,:]
W=paddle.rand(shape=[8,4])
seq1,seq2,W

Out[51]: (Tensor(shape=[2, 8], dtype=float32, place=Place(cpu), stop_gradient=True,
[[1., 2., 3., 4., 5., 6., 7., 8.],
[9., 10., 11., 12., 13., 14., 15., 16.]]),
Tensor(shape=[2, 8], dtype=float32, place=Place(cpu), stop_gradient=True,
[[17., 18., 19., 20., 21., 22., 23., 24.],
[25., 26., 27., 28., 29., 30., 31., 32.]]),
Tensor(shape=[8, 4], dtype=float32, place=Place(cpu), stop_gradient=True,
[[0.57241243, 0.45361701, 0.77573997, 0.01048208],
[0.46195969, 0.10416761, 0.44305241, 0.38260248],
[0.99569464, 0.50882483, 0.04836850, 0.84598064],
[0.29769155, 0.55289471, 0.84636718, 0.56067896],
[0.73811269, 0.36310029, 0.28137168, 0.33273944],
[0.67521429, 0.65272415, 0.78472847, 0.74406075],
[0.64274055, 0.43006125, 0.61808574, 0.19687471],
[0.95740205, 0.94290531, 0.02224903, 0.72165704]]))
```

```
In [52]: seq1_encoder=seq1.matmul(W)
seq2_encoder=seq2.matmul(W)
seq1_encoder,seq2_encoder
```

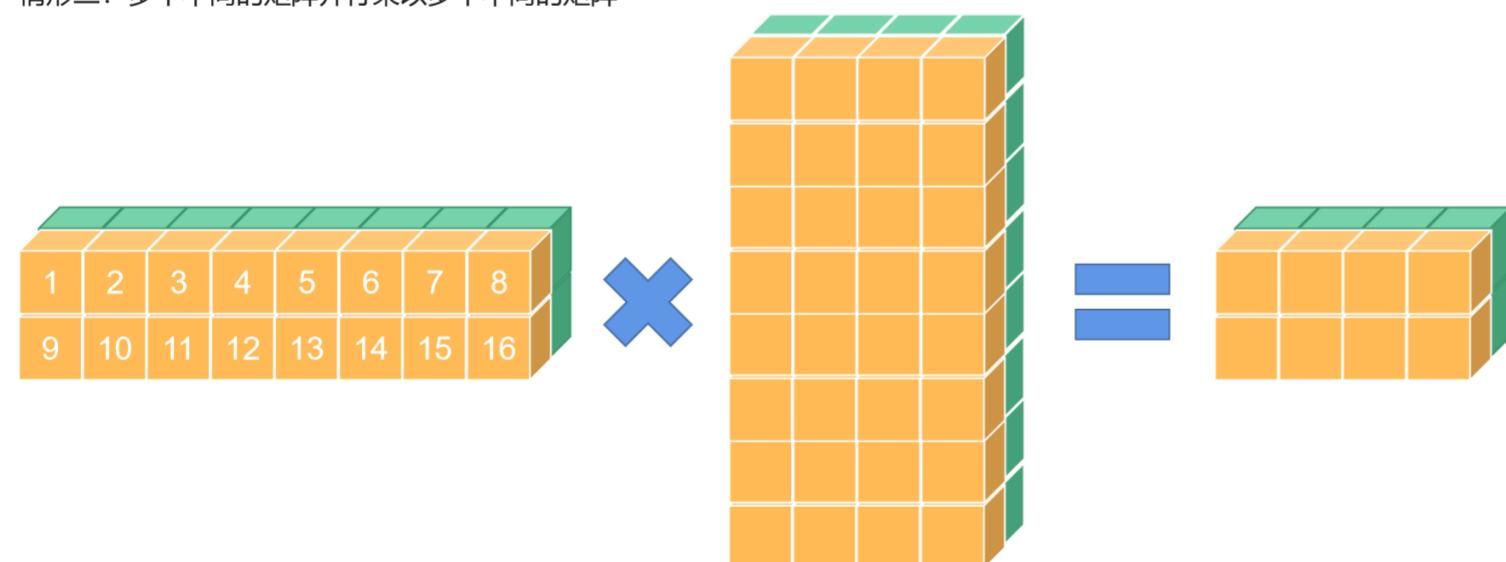
```
Out[52]: (Tensor(shape=[2, 4], dtype=float32, place=Place(cpu), stop_gradient=True,
[[25.57443237, 20.68552399, 15.81224060, 18.83578491],
[68.30426025, 52.75188446, 46.37194061, 49.19639587]], Tensor(shape=[2, 4], dtype=float32, place=Place(cpu), stop_gradient=True,
[[111.03407288, 84.81824493, 76.93164825, 79.55699921],
[153.76390076, 116.88461304, 107.49134827, 109.91761017]]))
```

使用并行计算结果如下

```
In [53]: x.matmul(W)

Out[53]: Tensor(shape=[2, 2, 4], dtype=float32, place=Place(cpu), stop_gradient=True,
[[[25.57443237, 20.68552399, 15.81224060, 18.83578491],
[68.30426025, 52.75188446, 46.37194443, 49.19639587]],
[[111.03407288, 84.81824493, 76.93164825, 79.55699921],
[153.76390076, 116.88461304, 107.49134827, 109.91761017]]])
```

情形二：多个不同的矩阵并行乘以多个不同的矩阵



如上图所示，这次我们弄了两个不同的W，分别跟两个句子相乘，就是第一排跟第一排相乘，第二排跟第二排相乘。

手动把每一排相乘

```
In [54]: paddle.seed(102)
seq1=x[0,:,:]
seq2=x[1,:,:]
W1=paddle.rand(shape=[8,4])
W2=paddle.rand(shape=[8,4])
seq1,seq2,W1,W2
```

```
Out[54]: (Tensor(shape=[2, 8], dtype=float32, place=Place(cpu), stop_gradient=True,
[[1., 2., 3., 4., 5., 6., 7., 8.]),
[9., 10., 11., 12., 13., 14., 15., 16.]]),
Tensor(shape=[2, 8], dtype=float32, place=Place(cpu), stop_gradient=True,
[[17., 18., 19., 20., 21., 22., 23., 24.]),
[25., 26., 27., 28., 29., 30., 31., 32.]]),
Tensor(shape=[8, 4], dtype=float32, place=Place(cpu), stop_gradient=True,
[[0.57241243, 0.45361701, 0.77573997, 0.01048208],
[0.46195969, 0.10416761, 0.44305241, 0.38260248],
[0.99569464, 0.50882483, 0.04836850, 0.84598064],
[0.29769155, 0.55289471, 0.84636718, 0.56067896],
[0.73811269, 0.36310029, 0.28137168, 0.33273944],
[0.67521429, 0.65272415, 0.78472847, 0.74406075],
[0.64274055, 0.43006125, 0.61808574, 0.19687471],
[0.95740205, 0.94290531, 0.02224903, 0.72165704]]),
Tensor(shape=[8, 4], dtype=float32, place=Place(cpu), stop_gradient=True,
[[0.36518422, 0.73727894, 0.12286152, 0.25183380],
[0.42372191, 0.06211283, 0.81855983, 0.18270063],
[0.57215798, 0.58215296, 0.30646491, 0.07521024],
[0.54299659, 0.90746790, 0.72740388, 0.34959465],
[0.75364786, 0.64785343, 0.22374235, 0.36532798],
[0.53331590, 0.68132800, 0.61552483, 0.97442144],
[0.46480274, 0.62408495, 0.33415428, 0.22650568],
[0.61128128, 0.94556141, 0.50532883, 0.75958169]]]))
```

```
In [55]: seq1_encoder=seq1.matmul(W1)
seq2_encoder=seq2.matmul(W2)
seq1_encoder,seq2_encoder
```

```
Out[55]: (Tensor(shape=[2, 4], dtype=float32, place=Place(cpu), stop_gradient=True,
[[25.57443237, 20.68552399, 15.81224060, 18.83578491],
[68.30426025, 52.75188446, 46.37194061, 49.19639587]]),
Tensor(shape=[2, 4], dtype=float32, place=Place(cpu), stop_gradient=True,
[[88.48683167, 108.50360107, 75.24720764, 68.53942871],
[122.62368774, 150.00631714, 104.47953796, 94.02083588]]))
```

并行相乘

```
In [56]: W=paddle.concat((W1.unsqueeze(0),W2.unsqueeze(0)),0) #先别管这行代码怎么用，后面会提到
W
```

```
Out[56]: Tensor(shape=[2, 8, 4], dtype=float32, place=Place(cpu), stop_gradient=True,
[[[0.57241243, 0.45361701, 0.77573997, 0.01048208],
[0.46195969, 0.10416761, 0.44305241, 0.38260248],
[0.99569464, 0.50882483, 0.04836850, 0.84598064],
[0.29769155, 0.55289471, 0.84636718, 0.56067896],
[0.73811269, 0.36310029, 0.28137168, 0.33273944],
[0.67521429, 0.65272415, 0.78472847, 0.74406075],
[0.64274055, 0.43006125, 0.61808574, 0.19687471],
[0.95740205, 0.94290531, 0.02224903, 0.72165704]],
[[0.36518422, 0.73727894, 0.12286152, 0.25183380],
[0.42372191, 0.06211283, 0.81855983, 0.18270063],
[0.57215798, 0.58215296, 0.30646491, 0.07521024],
[0.54299659, 0.90746790, 0.72740388, 0.34959465],
[0.75364786, 0.64785343, 0.22374235, 0.36532798],
[0.53331590, 0.68132800, 0.61552483, 0.97442144],
[0.46480274, 0.62408495, 0.33415428, 0.22650568],
[0.61128128, 0.94556141, 0.50532883, 0.75958169]]]))
```

```
In [57]: x.matmul(W)
```

```
Out[57]: Tensor(shape=[2, 2, 4], dtype=float32, place=Place(cpu), stop_gradient=True,
[[[25.57443237, 20.68552399, 15.81224060, 18.83578491],
[68.30426025, 52.75188446, 46.37194061, 49.19639587]],
[[88.48683167, 108.50360107, 75.24720764, 68.53942871],
[122.62368774, 150.00631714, 104.47953796, 94.02083588]]])
```

5.张量计算

- (1).基本计算
- (2).统计计算

5.1 基本计算

- 加减乘除：按元素逐个运算
 - $a + b = \text{paddle.add}(a, b)$
 - $a - b = \text{paddle.subtract}(a, b)$ pytorch是sub
 - $a * b = \text{paddle.multiply}(a, b)$
 - $a/b = \text{paddle.divide}(a, b)$ pytorch是div
 - $a ** b = \text{paddle.pow}(a, b)$
- 加法的广播机制
 - 维数一致：如shape=[3,5,8]和shape=[1,5,1]运算的结果为shape=[3,5,8]。
 - 维数不一致：如shape=[3,5,8]和shape=[1,8]两个矩阵右对齐，然后维数少的左边补齐，[1,8]-->[1,8]，然后运算，结果shape=[3,5,8]
 - 对比相应维数的长度时，如果不等，必须有一个为1才能进行广播，否则出错。如shape=[3,6]和shape=[2,6]无法运算
- Paddle矩阵乘法matmul
 - 1维乘以1维-->1维
 - $\text{paddle.matmul}(x, y) = xy^T$
 - $\text{paddle.matmul}(y, x) = yx^T$
 - 1维乘以2维-->1维
 - 1维向量转置还是其自己，所以等于没有转置
 - 1维向量只能放在矩阵左边才是矩阵乘法，放在右边会将1维矩阵广播进行点运算
 - 不要试图使用矩阵转置达到你想要的矩阵乘法效果，这里行不通，容易出错
 - $\text{paddle.matmul}(W, x) = W$ 的每一行与x点积
 - $\text{paddle.matmul}(x, W) = xW$
 - $\text{paddle.matmul}(W, x, \text{t}()) = \text{paddle.matmul}(W, x)$
 - 2维乘以2维-->2维(只要1维向量不参与结果永远是二维)
 - $\text{paddle.matmul}(A, B) = AB$
 - $\text{paddle.multiply}(A, B) = A \cdot B$
 - M.shape=[j,1,n,m], N.shape=[k,m,p]: 最后两个维度作矩阵乘法，其他维度进行广播机制
 - [j,1]广播[k]，就是[j,k]，然后[n,m]与[m,p]作矩阵乘法，就是[n,p]，最终结果shape=[j,k,n,p]
 - 实际情况，N有k个shape=[m,p]的矩阵，将M也变成k个shape=[n,m]的矩阵，作运算，最后复制为j份存到dim=0这一维
- Numpy&Torch矩阵乘法matmul
 - 1维乘以1维-->0维(这点和paddle不一样)
 - 1维乘以2维-->1维
 - 2维乘以2维-->2维(只要1维向量不参与结果永远是二维)

```
In [58]: import numpy as np
import torch
import paddle
```

例子：1维乘以1维

numpy&torch:1维乘以1维-->0维
paddle:1维乘以1维-->1维

```
In [59]: x=np.array([0,1,2,3])
y=np.array([3,2,1,1])
x.T,np.matmul(x,y),np.matmul(x,y.T),np.matmul(x.T,y)
```

```
Out[59]: (array([0, 1, 2, 3]), 7, 7, 7)
```

```
In [60]: x=torch.tensor([0,1,2.,3])
y=torch.tensor([3,2,1.,1])
x.T,torch.matmul(x,y),torch.matmul(x,y.T),torch.matmul(x.T,y)

C:\Users\Administrator\AppData\Local\Temp\ipykernel_14900\3098817676.py:3: UserWarning: The use of `x.T` on tensors of dimension other than 2 to reverse their shape is deprecated and it will throw an error in a future release. Consider `x.mT` to transpose batches of matrices or `x.permute(*torch.arange(x.ndim - 1, -1, -1))` to reverse the dimensions of a tensor. (Triggered internally at ..\aten\src\ATen\native\TensorShape.cpp:3575.)
x.T,torch.matmul(x,y),torch.matmul(x,y.T),torch.matmul(x.T,y)

Out[60]: (tensor([0., 1., 2., 3.]), tensor(7.), tensor(7.), tensor(7.))

In [61]: x=paddle.to_tensor([0,1,2.,3])
y=paddle.to_tensor([3,2,1.,1])
x.T,paddle.matmul(x,y),paddle.matmul(x,y.T),paddle.matmul(x.T,y)

Out[61]: (Tensor(shape=[4], dtype=float32, place=Place(cpu), stop_gradient=True,
[0., 1., 2., 3.]),
Tensor(shape=[1], dtype=float32, place=Place(cpu), stop_gradient=True,
[7.]),
Tensor(shape=[1], dtype=float32, place=Place(cpu), stop_gradient=True,
[7.]),
Tensor(shape=[1], dtype=float32, place=Place(cpu), stop_gradient=True,
[7.]))
```

例子：1维乘以2维—>1维

```
In [62]: x=torch.tensor([2.,3])
W=torch.tensor([[1.,2],[3,4]])
a=torch.matmul(x,W)
b=torch.matmul(W,x.t())
c=torch.matmul(W,x.t())
a,b,c

Out[62]: (tensor([11., 16.]), tensor([ 8., 18.]), tensor([ 8., 18.]))

In [63]: x=paddle.to_tensor([2.,3])
W=paddle.to_tensor([[1.,2],[3,4]])
a=paddle.matmul(x,W)
b=paddle.matmul(W,x.t())
c=paddle.matmul(W,x.t())
a,b,c

Out[63]: (Tensor(shape=[2], dtype=float32, place=Place(cpu), stop_gradient=True,
[11., 16.]),
Tensor(shape=[2], dtype=float32, place=Place(cpu), stop_gradient=True,
[8., 18.]),
Tensor(shape=[2], dtype=float32, place=Place(cpu), stop_gradient=True,
[8., 18.]))
```

例子：2维乘以2维—>2维

```
In [64]: x=np.array([[0,1,2,3]])
y=np.array([[3],[2],[1],[1]])
np.matmul(x,y)

Out[64]: array([[7]])

In [65]: x=paddle.to_tensor([[0,1,2,3]])
y=paddle.to_tensor([[3],[2],[1],[1]])
paddle.matmul(x,y)

Out[65]: Tensor(shape=[1, 1], dtype=float32, place=Place(cpu), stop_gradient=True,
[[7.]])
```

例子：矩阵乘法，Batch维自动广播机制，并行计算

```
In [66]: x=paddle.to_tensor([[2.,3],[1,1],[3,8],[33,21]], [[1,1],[1,1],[1,1],[1,1]])      #shape=[2,4,2]
y=paddle.to_tensor([[1.,2,3],[4,5,6]])          #shape=[2,3]
a=paddle.matmul(x,y)                            #shape=[2,4,3], 实际就是将x的每排shape=[4,2]的矩阵分别与矩阵y相乘
a

Out[66]: Tensor(shape=[2, 4, 3], dtype=float32, place=Place(cpu), stop_gradient=True,
[[[14., 19., 24.],
[5., 7., 9.],
[35., 46., 57.],
[117., 171., 225.]],
[[5., 7., 9.],
[5., 7., 9.],
[5., 7., 9.],
[5., 7., 9.]]])
```

5.2 统计计算

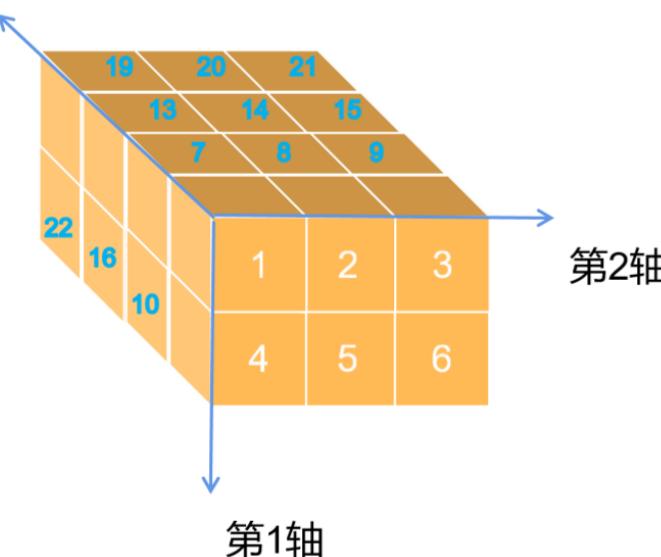
- `torch.sum()`, `torch.mean()`, `torch.var()`, `torch.std()`
- 对全部元素求和, 均值, 方差, 标准差

`torch.sum(axis)`, `torch.mean(axis)`, `torch.var(axis)`, `torch.std(axis)`

- 沿着魔方的轴进行求和, 均值, 方差, 标准差

注: pytorch中dim是axis

第0轴



```
In [67]: x=paddle.arange(start=1,end=25,step=1).reshape([4,2,-1]) # -1这个维度会自动计算, 这里元素24个, 所以-1就是3
x,x.sum(),x.sum(axis=0),x.sum(axis=1),x.sum(axis=2)
```

```

Out[67]: (Tensor(shape=[4, 2, 3], dtype=int64, place=Place(cpu), stop_gradient=True,
[[1 , 2 , 3 ],
[4 , 5 , 6 ]],
[[7 , 8 , 9 ],
[10, 11, 12]],
[[13, 14, 15],
[16, 17, 18]],
[[19, 20, 21],
[22, 23, 24]]),
Tensor(shape=[1], dtype=int64, place=Place(cpu), stop_gradient=True,
[300]),
Tensor(shape=[2, 3], dtype=int64, place=Place(cpu), stop_gradient=True,
[[40, 44, 48],
[52, 56, 60]]),
Tensor(shape=[4, 3], dtype=int64, place=Place(cpu), stop_gradient=True,
[[5 , 7 , 9 ],
[17, 19, 21],
[29, 31, 33],
[41, 43, 45]]),
Tensor(shape=[4, 2], dtype=int64, place=Place(cpu), stop_gradient=True,
[[6 , 15],
[24, 33],
[42, 51],
[60, 69]]))

```

6.张量的基本数据类型

Tensor 的数据类型 `dtype` 可以通过 `Tensor.dtype` 查看，支持类型包括：bool、float16、float32、float64、uint8、int8、int16、int32、int64、complex64、complex128。

同一 Tensor 中所有元素的数据类型均相同，通常通过如下方式指定：

- 通过给定 Python 序列创建的 Tensor，可直接使用 `dtype` 参数指定。如果未指定：
 - 对于 Python 整型数据，默认会创建 int64 型 Tensor；
 - 对于 Python 浮点型数据，默认会创建 float32 型 Tensor，并且可以通过 `paddle.set_default_dtype` 来调整浮点型数据的默认类型。

```
In [68]: # 注意，整型的tensor和浮点型的混合计算经常会出现错误，一般创建tensor的时候记得带小数点，统一设置成float32类型
paddle.to_tensor([1,2,3,4,5]),paddle.to_tensor([1.,2,3,4,5]),paddle.to_tensor([1,2,3,4,5],dtype=paddle.float32)
```

```
Out[68]: (Tensor(shape=[5], dtype=int64, place=Place(cpu), stop_gradient=True,
[1, 2, 3, 4, 5]),
Tensor(shape=[5], dtype=float32, place=Place(cpu), stop_gradient=True,
[1., 2., 3., 4., 5.]),
Tensor(shape=[5], dtype=float32, place=Place(cpu), stop_gradient=True,
[1., 2., 3., 4., 5.]))
```

修改数据类型的方法：`paddle.cast()`

```
In [69]: x=paddle.to_tensor([1,2,3,4,5])
y=x.cast(paddle.float32)
x,y
```

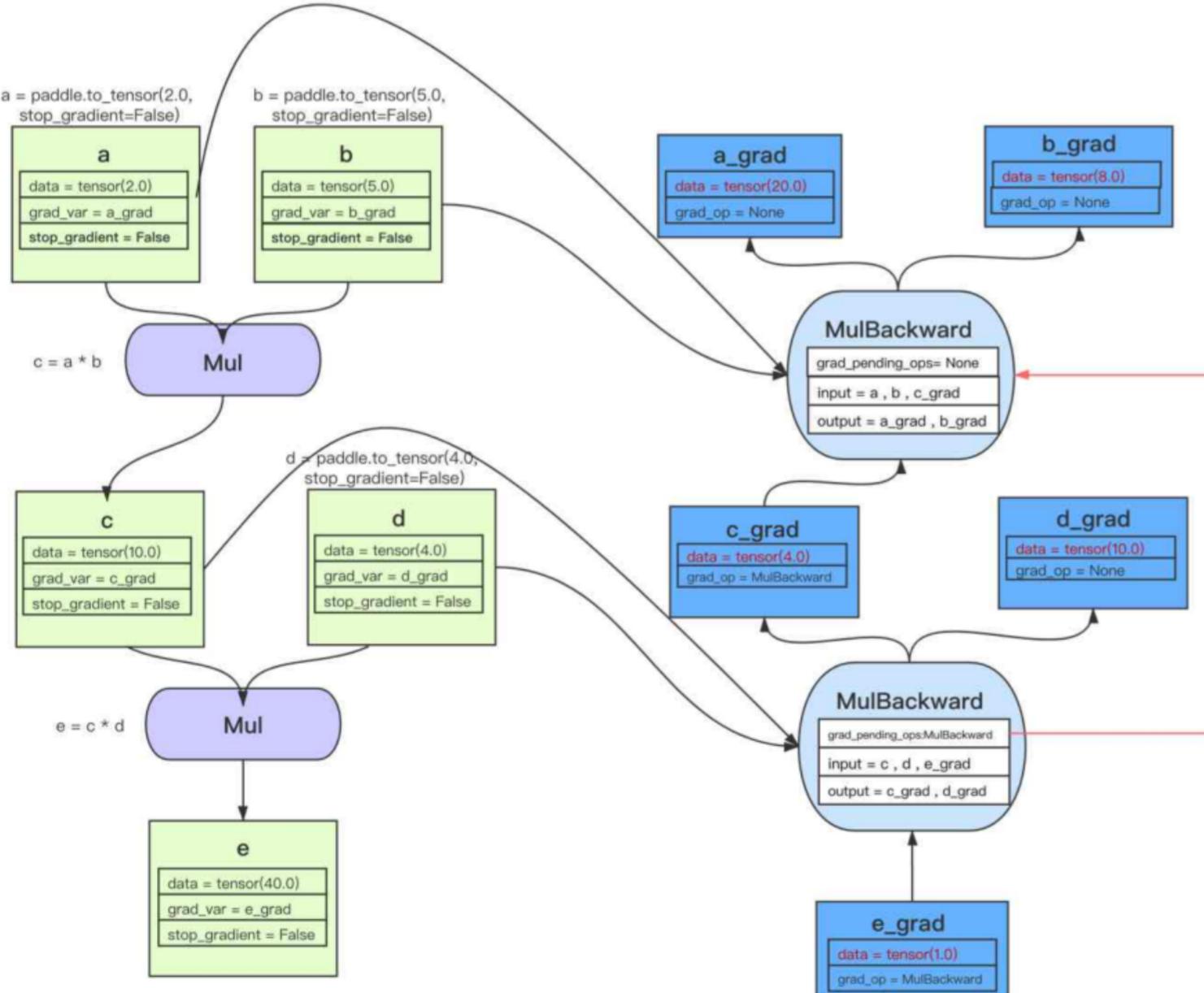
```
Out[69]: (Tensor(shape=[5], dtype=int64, place=Place(cpu), stop_gradient=True,
[1, 2, 3, 4, 5]),
Tensor(shape=[5], dtype=float32, place=Place(cpu), stop_gradient=True,
[1., 2., 3., 4., 5.]))
```

全局默认类型设置：
`paddle.set_default_dtype()` 是用来设置浮点数默认类型的，对整型不起作用
`torch.set_default_tensor_type(torch.FloatTensor)` 才是真正的设置全局默认数据类型

```
In [70]: paddle.set_default_dtype("float64")
paddle.to_tensor([1,2,3,4,5]),paddle.to_tensor([1.,2,3,4,5])
```

```
Out[70]: (Tensor(shape=[5], dtype=int64, place=Place(cpu), stop_gradient=True,
[1, 2, 3, 4, 5]),
Tensor(shape=[5], dtype=float64, place=Place(cpu), stop_gradient=True,
[1., 2., 3., 4., 5.]))
```

7.Paddle中的计算图与自动微分



张量节点：不参与计算图，仅仅作为算子的输入输出

前向算子节点：完成前向计算，创建反向算子

反向算子节点：具备完成反向计算的功能，并且通过红线构建反向计算图

`backward()`:遍历反向计算图

上图可以这么看，首先不要看连线，这些节点对象包括变量节点，正向算子节点，反向算子节点。

变量节点和正向算子节点是我们通过前向传播手动创建的，图中黑色连线表示算子的输入和输出，比如 $c=a*b$ ，就是拿 a , b 丢进 Mul 算子中，然后 Mul 吐出来一个 c 。这就是前向传播所有对象的创建过程，反向张量和反向算子其实也是在前向计算的时候自动创建的，如果这个变量需要求梯度，就创建反向张量（创建变量的时候完成）。反向算子是正向算子执行过程中自动创建的，并且会让它的`input`属性指向3个对象，分别是 a , b , c_grad ，后面计算梯度需要用到。

另外反向算子还有个`grad_pending_ops`属性，这个就是图中的红线，用于构建反向计算图。

比如 $MulBackward$ ，输入连接3个对象，输出连接3个对象，红线构建的反向计算图，只要在`backward`函数中对反向计算图进行遍历，其连接的输出对象的值都能计算出来。

但是不清楚为什么，我输出相关属性会报错，上图来自官方文档，不清楚哪里出了问题。

```
In [71]: a = paddle.to_tensor(2.0, stop_gradient=False)
b = paddle.to_tensor(5.0, stop_gradient=False)
```

```

c = a * b
d = paddle.to_tensor(4.0, stop_gradient=False)
e = c * d
e.backward()
print("Tensor a's grad is: {}".format(a.grad))
print("Tensor b's grad is: {}".format(b.grad))
print("Tensor c's grad is: {}".format(c.grad))
print("Tensor d's grad is: {}".format(d.grad))

Tensor a's grad is: Tensor(shape=[1], dtype=float64, place=Place(cpu), stop_gradient=False,
[20.])
Tensor b's grad is: Tensor(shape=[1], dtype=float64, place=Place(cpu), stop_gradient=False,
[8.])
Tensor c's grad is: None
Tensor d's grad is: Tensor(shape=[1], dtype=float64, place=Place(cpu), stop_gradient=False,
[10.])

```

既然官方文档调用出错，我们按照上面的图手写一个

```

In [72]: class Tensor_grad:
    def __init__(self, data=None, grad_op=None):
        self.data = data
        self.grad_op = grad_op

    class Tensor:
        def __init__(self, data=None, stop_gradient=True):
            self.data = data
            self.stop_gradient = stop_gradient

            if not self.stop_gradient:
                self.grad_var = Tensor_grad() # 创建反向张量

        # 最后一个节点调用，遍历反向计算图，初始梯度为1
        def backward(self):
            self.grad_var.data = 1.0
            self.grad_var.grad_op.backward()

    class Mul:
        """
        1. 产生前向计算的节点，并把结果写进去
        2. 产生反向张量
        3. 产生反向算子
        """

        def __init__(self):
            pass

        def __call__(self, a, b):
            c = Tensor(data=a.data * b.data) # 此时不会立刻创建反向张量
            if not a.stop_gradient or not b.stop_gradient:
                c.stop_gradient = False
            if not c.stop_gradient:
                c.grad_var = Tensor_grad() # 创建输出节点反向张量
                c.grad_var.grad_op = Mulbackward(a, b, c.grad_var) # 创建反向算子
            print(c.stop_gradient, c.grad_var.grad_op)
            return c

    class Mulbackward:
        def __init__(self, a, b, grad_var):
            self.input = [a, b, grad_var]
            self.output = [a.grad_var, b.grad_var]
            if a.grad_var.grad_op or b.grad_var.grad_op:
                self.grad_pending_ops = [a.grad_var.grad_op, b.grad_var.grad_op]
            else:
                self.grad_pending_ops = None

        def __call__(self):
            da = self.input[2].data * self.input[1].data
            db = self.input[2].data * self.input[0].data
            self.output[0].data = da
            self.output[1].data = db

        def backward(self, grads=1):
            Mulbackward.__call__(self) # 求梯度
            if self.grad_pending_ops: # 判断后面的连接是否为空
                if self.grad_pending_ops[0]:
                    self.grad_pending_ops[0].backward(grads=grads)
                if self.grad_pending_ops[1]:
                    self.grad_pending_ops[1].backward(grads=grads)

In [73]: mul = Mul()
a = Tensor(2.0, stop_gradient=False)
b = Tensor(5.0, stop_gradient=False)
c = mul(a, b)
d = Tensor(4.0, stop_gradient=False)
e = mul(c, d)
e.backward()
a.grad_var.data, b.grad_var.data, c.grad_var.data, d.grad_var.data

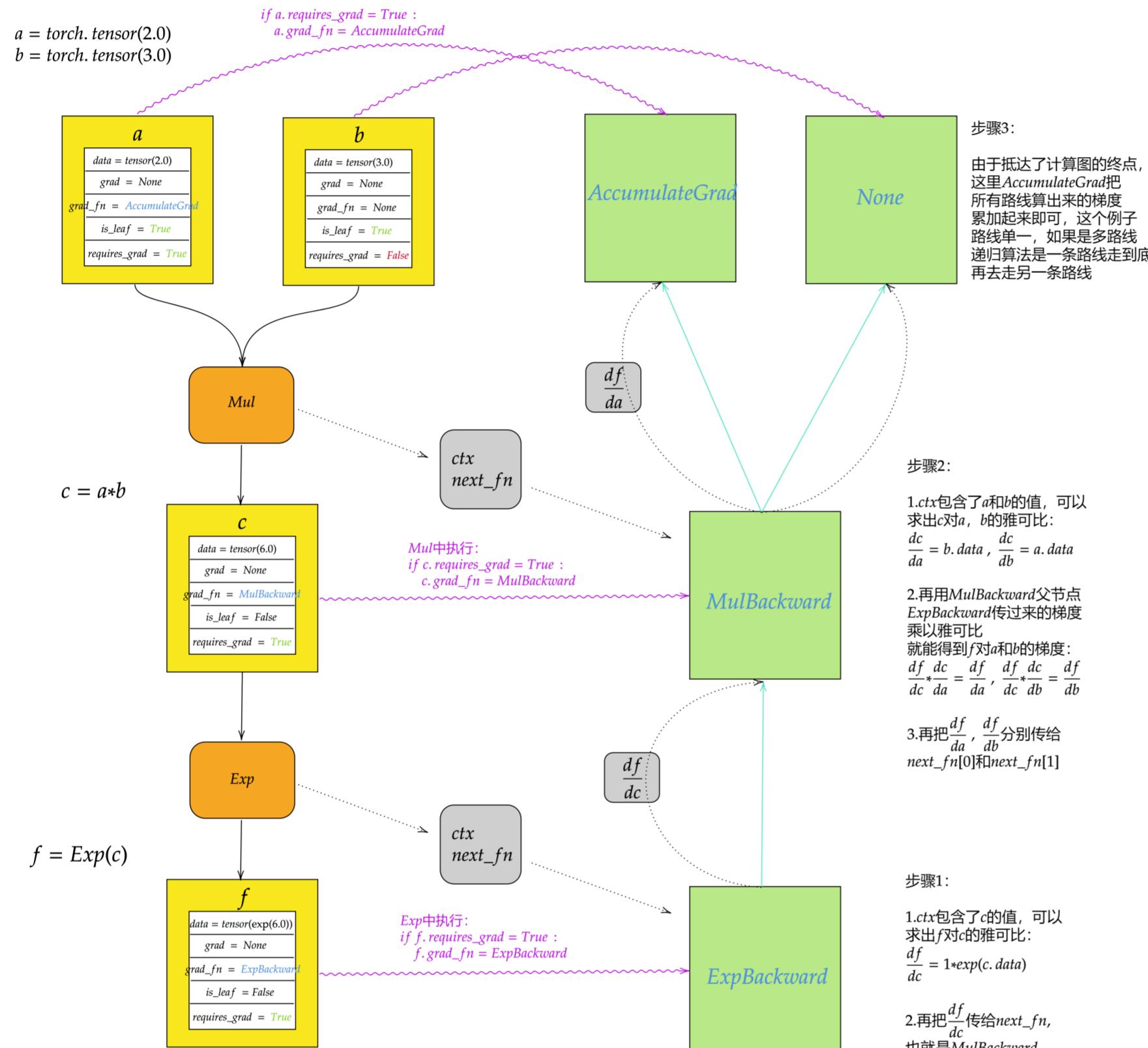
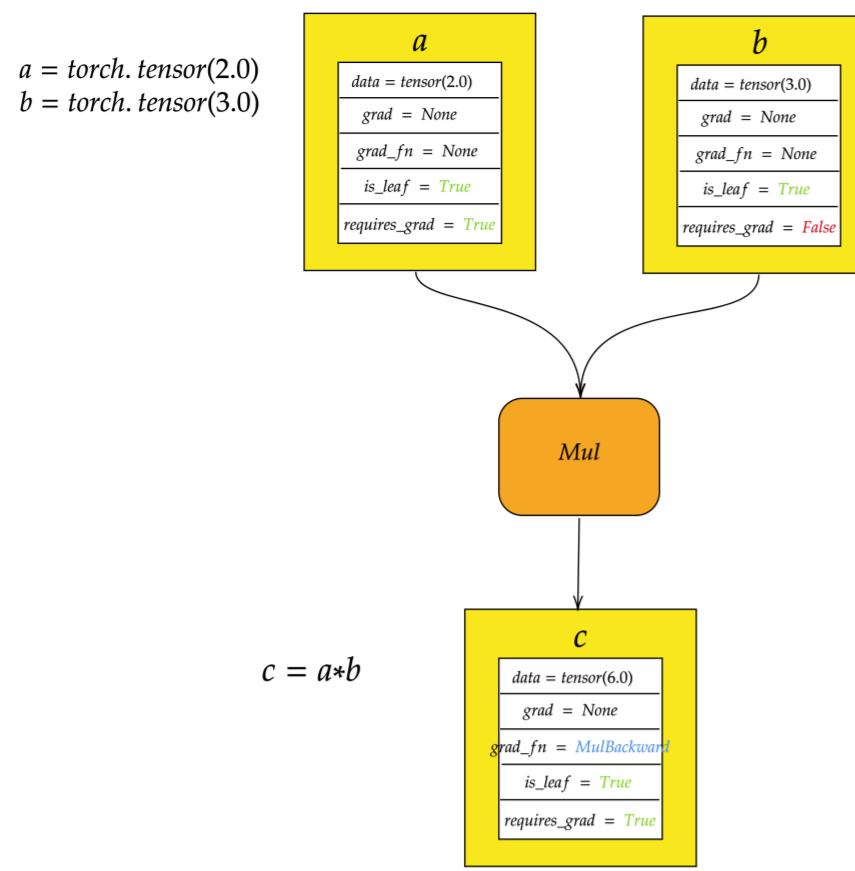
```

```

False <__main__.Mulbackward object at 0x00000239B970B5E0>
False <__main__.Mulbackward object at 0x00000239B970ACE0>
Out[73]: (20.0, 8.0, 4.0, 10.0)

```

8.PyTorch中的计算图与自动微分



上图是我根据Pytorch中的相关属性及方法，写的一个实现（不是完全一样的）

- 黄色节点：就是前向传播过程中产生的节点，不构成计算图
- 连接黄色节点的黑实线：表示前向算子的输入输出
- 绿色节点：反向算子，反向计算图的节点
- 绿色实线：反向计算图的边
- 黑色虚线：用于传递需要用到的属性

```
In [74]: class Tensor:
    def __init__(self, data=None, requires_grad=False):
        self.data = data
        self.requires_grad = requires_grad

        self.grad = None
        self.grad_fn = None
        self.is_leaf = False

    if self.requires_grad and self.is_leaf:
        self.grad_fn = AccumulateGrad(self) # 反向传播的终点节点

    def backward(self):
        self.grad_fn.backward(grads=1)

class to_tensor(Tensor):
    def __init__(self, data=None, requires_grad=False):
        Tensor.__init__(self, data, requires_grad)
        self.is_leaf = True
        if self.requires_grad:
            self.grad_fn = AccumulateGrad(self)

class Mul:
    """
    1.产生前向计算的节点
    2.计算前向计算的结果
    3.记录反向传播的节点以及连接，即构建反向传播图
    """
    def __init__(self):
        pass

    def __call__(self, a, b):
        c = Tensor(data=a.data * b.data)
        if a.requires_grad or b.requires_grad:
```

```

    c.requires_grad = True
if c.requires_grad:
    next_fn=(a.grad_fn, b.grad_fn)
    ctx = (a.data, b.data)
    c.grad_fn = MulBackward(next_fn, ctx=ctx)
return c

class MulBackward:
    def __init__(self, next_fn, ctx=None): # *grad_fn是用来建立连接的, ctx是用来求梯度的
        self.next_functions = list(next_fn)
        self.x, self.y = ctx
        self.dx = self.y
        self.dy = self.x

    def __call__(self, grads=1):
        Dx=grads * self.dx
        Dy=grads * self.dy
        print("哈哈哈")
        return Dx,Dy

    def backward(self, grads=1):
        if self.next_functions[0] is not None:
            self.next_functions[0].backward(grads * self.dx)
        if self.next_functions[1] is not None:
            self.next_functions[1].backward(grads * self.dy)
        return

class AccumulateGrad:
    def __init__(self, node):
        self.node = node # 记录它对应的节点, 以便把求出来的梯度传进去
        self.acc_grads = None

    def __call__(self, grads):
        if self.acc_grads is None:
            self.acc_grads = grads
        else:
            self.acc_grads = self.acc_grads + grads
        self.node.grad = self.acc_grads
        return self.node.grad

    def backward(self, grads):
        if self.acc_grads is None:
            self.acc_grads = grads
        else:
            self.acc_grads = self.acc_grads + grads
        self.node.grad = self.acc_grads
        return

```

先把上面那个乘法的例子重新跑一次, 注意这里中间节点的梯度没有写入, paddle中也一样
只不过我最开始实现的那个保留了中间节点的梯度值

```
In [75]: mul=Mul()
a = to_tensor(data=2, requires_grad=True)
b = to_tensor(data=5, requires_grad=True)
c = mul(a, b)
d = to_tensor(data=4)
e = mul(c, d)
e.backward()
a.grad,b.grad
```

Out[75]: (20, 8)

利用反向算子, 一步一步求

```
In [76]: mul=Mul()
a = to_tensor(data=2, requires_grad=True)
b = to_tensor(data=5, requires_grad=True)
c = mul(a, b)
d = to_tensor(data=4)
e = mul(c, d)
de=1.0
mul_back1=e.grad_fn
dc,dd=mul_back1(de)
print(dc,dd)
mul_back2=mul_back1.next_functions[0]
da,db=mul_back2(dc)
print(da,db)
```

哈哈哈
4.0 10.0
哈哈哈
20.0 8.0