[Nvidia-docker](https://benpaodewoniu.github.io/2020/04/24/docker13/)

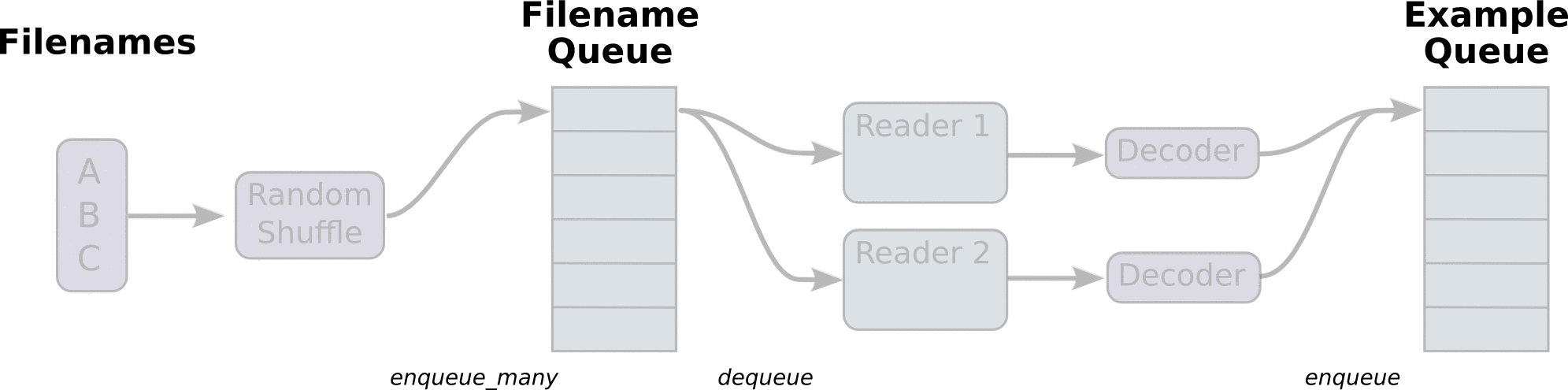
为了在Docker中使用GPU，就需要在container中安装主机上使用GPU的driver，然后把主机上的GPU设备（例如：/dev/nvidia0）映射到container中。

(确保您已经为您的 Linux 发行版安装了 NVIDIA 驱动程序和 Docker 19.03 注意您不需要在主机上安装 CUDA 工具包，但需要安装驱动程序)

(幸运的是服务器之前已经有人安装成功了：执行这段话会产生驱动信息:

sudo docker run --rm --gpus all nvidia/cuda:10.2-base nvidia-smi)

# [tf.TFRecordReader()函数解析](https://blog.csdn.net/TeFuirnever/article/details/90271862)

从TFRecords文件中读取数据， 首先需要用tf.train.string\_input\_producer生成一个解析队列。之后调用tf.TFRecordReader的tf.parse\_single\_example解析器

解析器首先读取解析队列，返回serialized\_example对象，之后调用tf.parse\_single\_example操作将Example协议缓冲区(protocol buffer)解析为张量。简单来说，一旦生成了TFRecords文件，接下来就可以使用队列（queue）读取数据了

def read\_and\_decode(filename):

    #根据文件名生成一个队列

    filename\_queue = tf.train.string\_input\_producer([filename])

    reader = tf.TFRecordReader()

    \_, serialized\_example = reader.read(filename\_queue)   #返回文件名和文件

    features = tf.parse\_single\_example(serialized\_example,

                                       features={

                                           'label': tf.FixedLenFeature([], tf.int64),

                                           'img\_raw' : tf.FixedLenFeature([], tf.string),

                                       })

    img = tf.decode\_raw(features['img\_raw'], tf.uint8)

    img = tf.reshape(img, [224, 224, 3])

    img = tf.cast(img, tf.float32) \* (1. / 255) - 0.5

    label = tf.cast(features['label'], tf.int32)

    return img, label



# [tf.train.shuffle\_batch函数解析](https://blog.csdn.net/u013555719/article/details/77679964)

通过随机打乱张量的顺序创建批次

tf.train.shuffle\_batch(tensor\_list, batch\_size, capacity, min\_after\_dequeue, num\_threads=1, seed=None, enqueue\_many=False, shapes=None, name=None)

参数：

tensor\_list: 入队的张量列表

batch\_size: 表示进行一次批处理的tensors数量.

capacity:一个整数,队列中的最大的元素数.

min\_after\_dequeue: 当一次出列操作完成后,队列中元素的最小数量,往往用于定义元素的混合级别.

num\_threads:设置num\_threads的值大于1,使用多个线程在tensor\_list中读取文件,这样保证了同一时刻只在一个文件中进行读取操作(但是读取速度依然优于单线程),而不是之前的同时读取多个文件,

seed:打乱tensor队列的随机数种子

enqueue\_many:定义tensor\_list中的tensor是否冗余.

shapes:用于改变读取tensor的形状,默认情况下和直接读取的tensor的形状一致.

name: 可选

Returns:默认返回一个和读取tensor\_list数据和类型一个tensor列表.

# [@staticmethod作用及用法](https://cloud.tencent.com/developer/article/1597015)

类名调用：事先声明@staticmethod，可以在不创建类实例的情况下调用类内方法，

静态调用：用实例调用该方法

class Time():

    def \_\_init\_\_(self,sec):

        self.sec = sec

    #声明一个静态方法

    @staticmethod

    def sec\_minutes(s1,s2):

        #返回两个时间差

        return abs(s1-s2)

t = Time(10)

print(Time.sec\_minutes(10,5))#类名调用

print(t.sec\_minutes(t.sec,5))#静态调用

#结果为5 5

# with...as[构建计算图](http://c.biancheng.net/view/1883.html)



# [tf.name\_scope()函数解析](https://www.jianshu.com/p/635d95b34e14)

1.tf.name\_scope()命名空间的实际作用

（1）在某个tf.name\_scope()指定的区域中定义的所有对象及各种操作，他们的“name”属性上会增加该命名区的区域名，用以区别对象属于哪个区域；

（2）将不同的对象及操作放在由tf.name\_scope()指定的区域中，便于在tensorboard中展示清晰的逻辑关系图，这点在复杂关系图中特别重要。

import tensorflow as tf;

tf.reset\_default\_graph()

# 无tf.name\_scope()

a = tf.constant(1,name='my\_a') #定义常量

b = tf.Variable(2,name='my\_b') #定义变量

c = tf.add(a,b,name='my\_add') #二者相加（操作）

print("a.name = "+a.name)

print("b.name = "+b.name)

print("c.name = "+c.name)

# 有tf.name\_scope()

# with tf.name\_scope('cgx\_name\_scope'): #定义一块名为cgx\_name\_scope的区域，并在其中工作

#     a = tf.constant(1,name='my\_a')

#     b = tf.Variable(2,name='my\_b')

#     c = tf.add(a,b,name='my\_add')

# print("a.name = "+a.name)

# print("b.name = "+b.name)

# print("c.name = "+c.name)

# 保存graph用于tensorboard绘图

with tf.Session() as sess:

    sess.run(tf.global\_variables\_initializer())

    writer = tf.summary.FileWriter("./test",sess.graph)

    print(sess.run(c))

writer.close()

# 输出结果

# 无tf.name\_scope()

a.name = my\_a:0

b.name = my\_b:0

c.name = my\_add:0

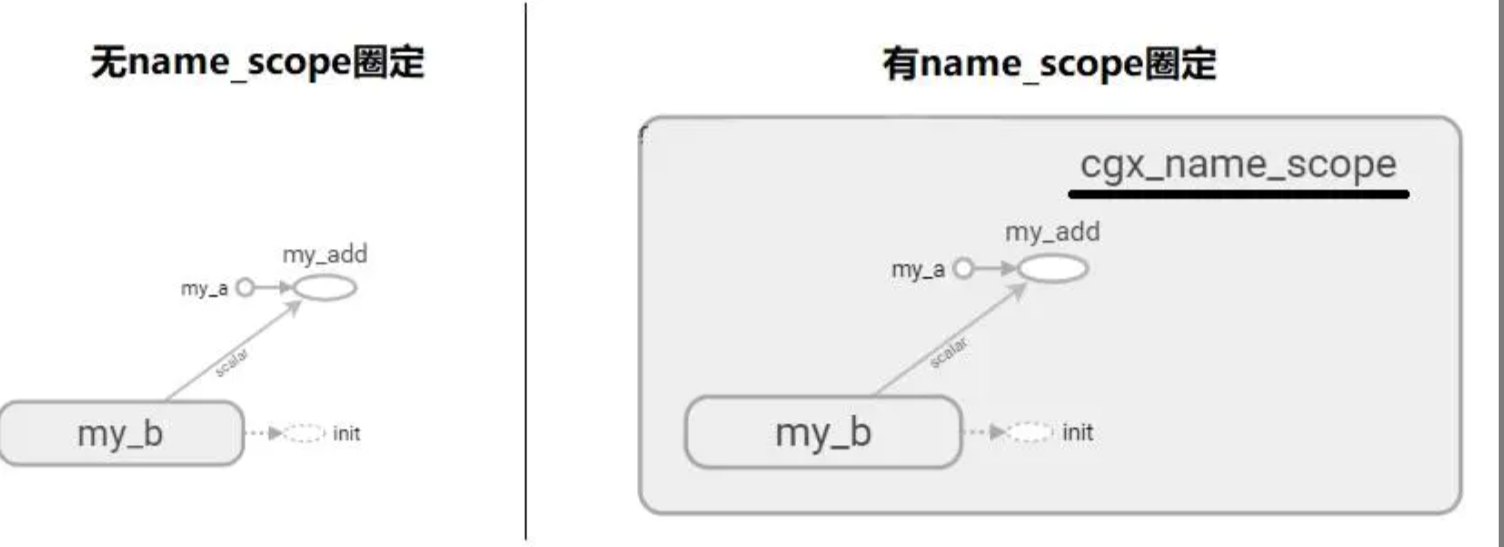
# 有tf.name\_scope()

a.name = cgx\_name\_scope/my\_a:0

b.name = cgx\_name\_scope/my\_b:0

c.name = cgx\_name\_scope/my\_add:0

从输出结果可以看出，在tf.name\_scope()下的所有对象和操作，其name属性前都加了cgx\_name\_scope，用以表示这些内容全在其范围下。  
下图展示了两种情况的tensorboard差异，差别一目了然。



2. name\_scope()只决定“对象”属于哪个范围，并不会对“对象”的“作用域”产生任何影响。

tf.name\_scope()只是规定了对象和操作属于哪个区域，但这并不意味着他们的作用域也只限于该区域（with的这种写法很容易让人产生这种误会）实际上他是具有cgx\_1前缀的全局变量

#name\_scope 的影响范围

with tf.name\_scope('cgx\_1'): #指定cgx\_1区域

    a = tf.Variable(tf.constant(4),name='my\_a') #定义变量a

    print("case1: a.name="+a.name) #cgx\_1区域内输出

print("case2: a.name="+a.name) #cgx\_1区域外输出

# 保存graph用于tensorboard绘图

with tf.name\_scope('cgx\_2'): #新指定cgx\_2区域

    print("case3: a.name="+a.name) #在cgx\_2内输出cgx\_1中定义a.name

    a = tf.Variable(tf.constant(4),name='my\_a') #★★重新定义a这个变量

    print("case4: a.name="+a.name)

print("case5: a.name="+a.name)

# 输出结果：

case1: a.name = cgx\_1/my\_a:0

case2: a.name = cgx\_1/my\_a:0

case3: a.name = cgx\_1/my\_a:0

case4: a.name = cgx\_2/my\_a:0

case5: a.name = cgx\_2/my\_a:0

3. tf.name\_scope('cgx\_scope')语句重复执行几次，就会生成几个独立的命名空间，尽管表面上看起来都是“cgx\_scope”，实际上tensorflow在每一次执行相同语句都会在后面加上“\_序数”，加以区别。

with tf.name\_scope('cgx\_scope'):

    a = tf.Variable(1,name='my\_a')

with tf.name\_scope('cgx\_scope'):

    b = tf.Variable(2,name='my\_b')

c = tf.add(a,b,name='my\_add')

print("a.name = "+a.name)

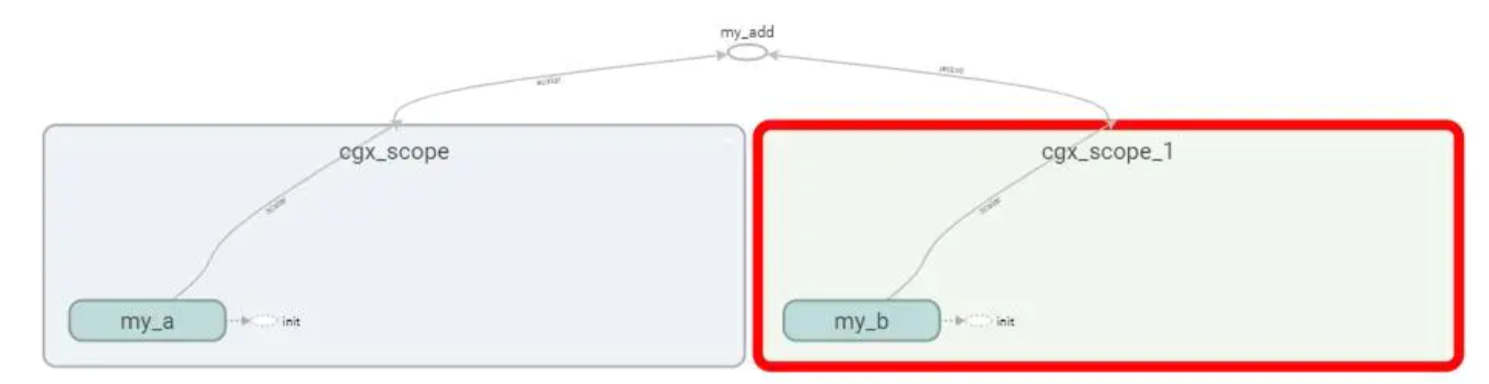
print("b.name = "+b.name)

# 保存graph用于tensorboard绘图程序（同上）

# 输出结果

a.name = cgx\_scope/my\_a:0

b.name = cgx\_scope\_1/my\_b:0 #自动增加了“\_1”



4. 如何在不同的地方将对象指定到相同的name\_scope中？

with tf.name\_scope('cgx\_scope') as cgxscope:

    a = tf.Variable(1,name='my\_a')

with tf.name\_scope(cgxscope):

    b = tf.Variable(2,name='my\_b')

c = tf.add(a,b,name='my\_add')

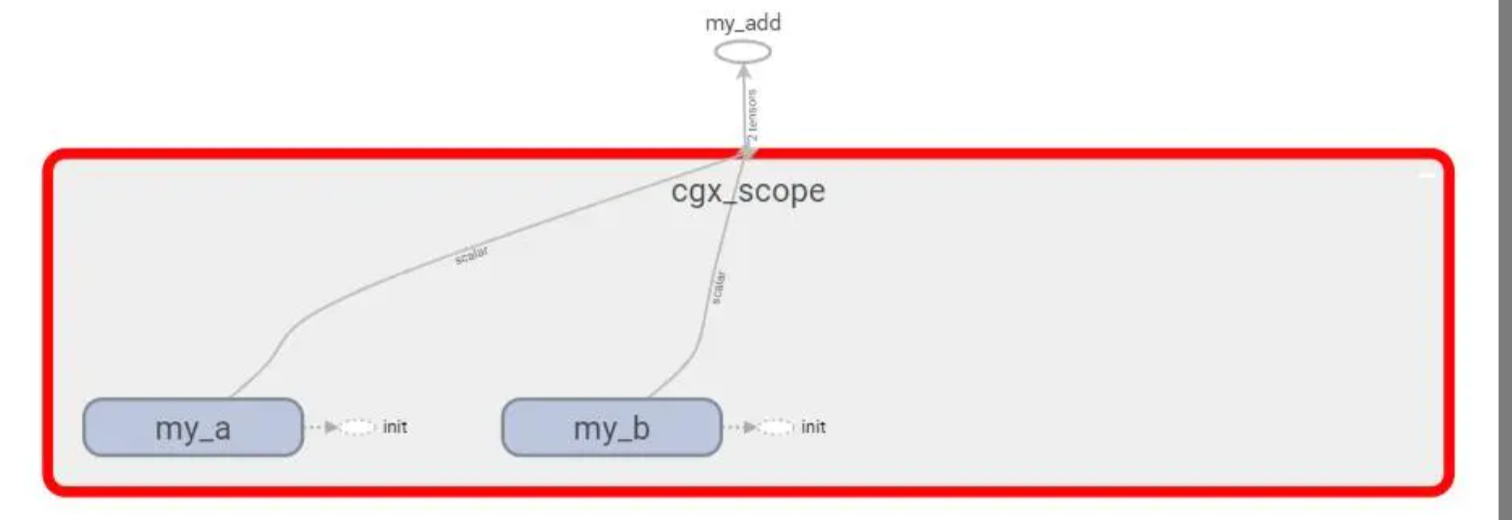
print("a.name = "+a.name)

print("b.name = "+b.name)

# 输出结果

a.name = cgx\_scope/my\_a:0

b.name = cgx\_scope/my\_b:0



5. 多重name\_scope命名空间

with tf.name\_scope('cgx\_scope\_1'): #第一重命名空间

    with tf.name\_scope('cgx\_scope\_2'): #第二重命名空间

        a = tf.Variable(1,name='my\_a')

with tf.name\_scope('cgx\_scope\_3'):

    b = tf.Variable(2,name='my\_b')

c = tf.add(a,b,name='my\_add')

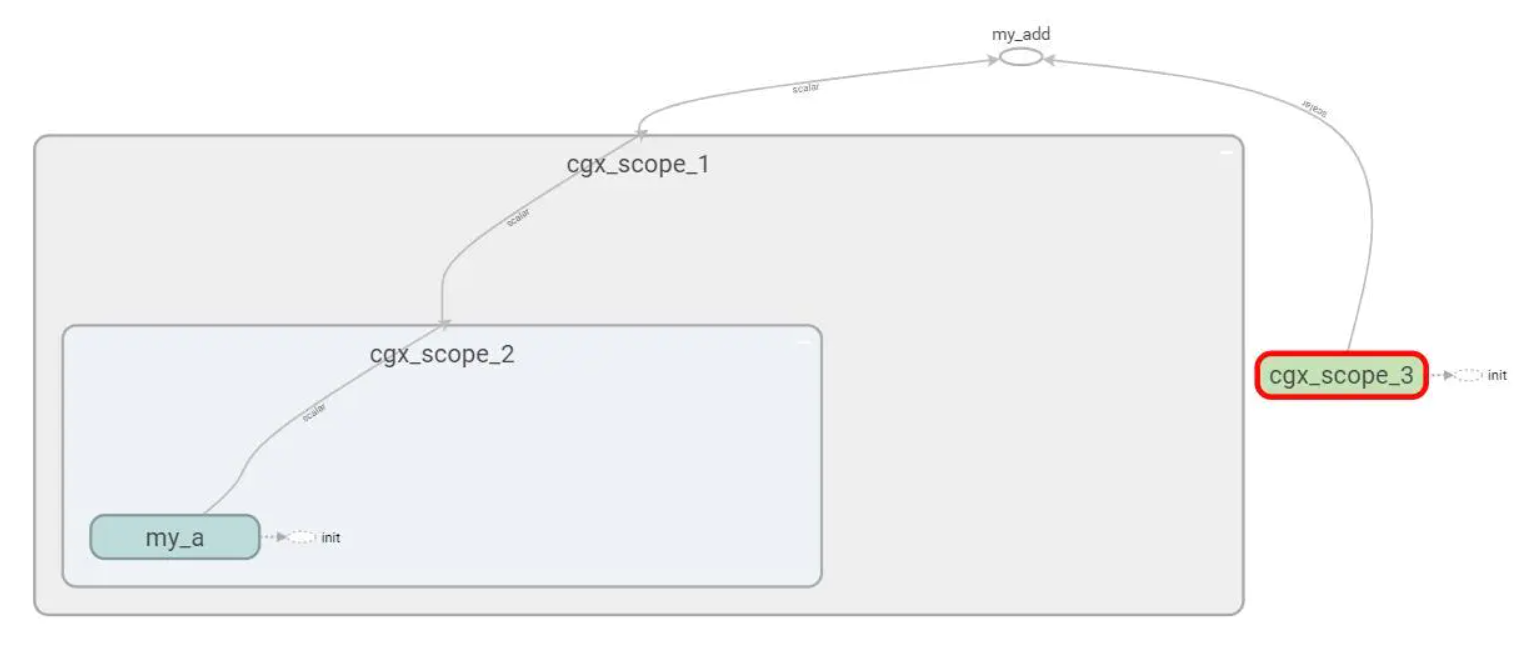
print("a.name = "+a.name)

print("b.name = "+b.name)

# 输出结果

a.name = cgx\_scope\_1/cgx\_scope\_2/my\_a:0

b.name = cgx\_scope\_3/my\_b:0



6. 常量tf.constant()即使在某个name\_scope下被定义，在tensorboard中也不会被绘制到该name\_scope下，主要考虑到图像的结构问题。

with tf.name\_scope('cgx\_scope'):

    a = tf.constant(1,name='my\_a')

    b = tf.Variable(2,name='my\_b')

c = tf.add(a,b,name='my\_add')

print("a.name = "+a.name)

print("b.name = "+b.name)

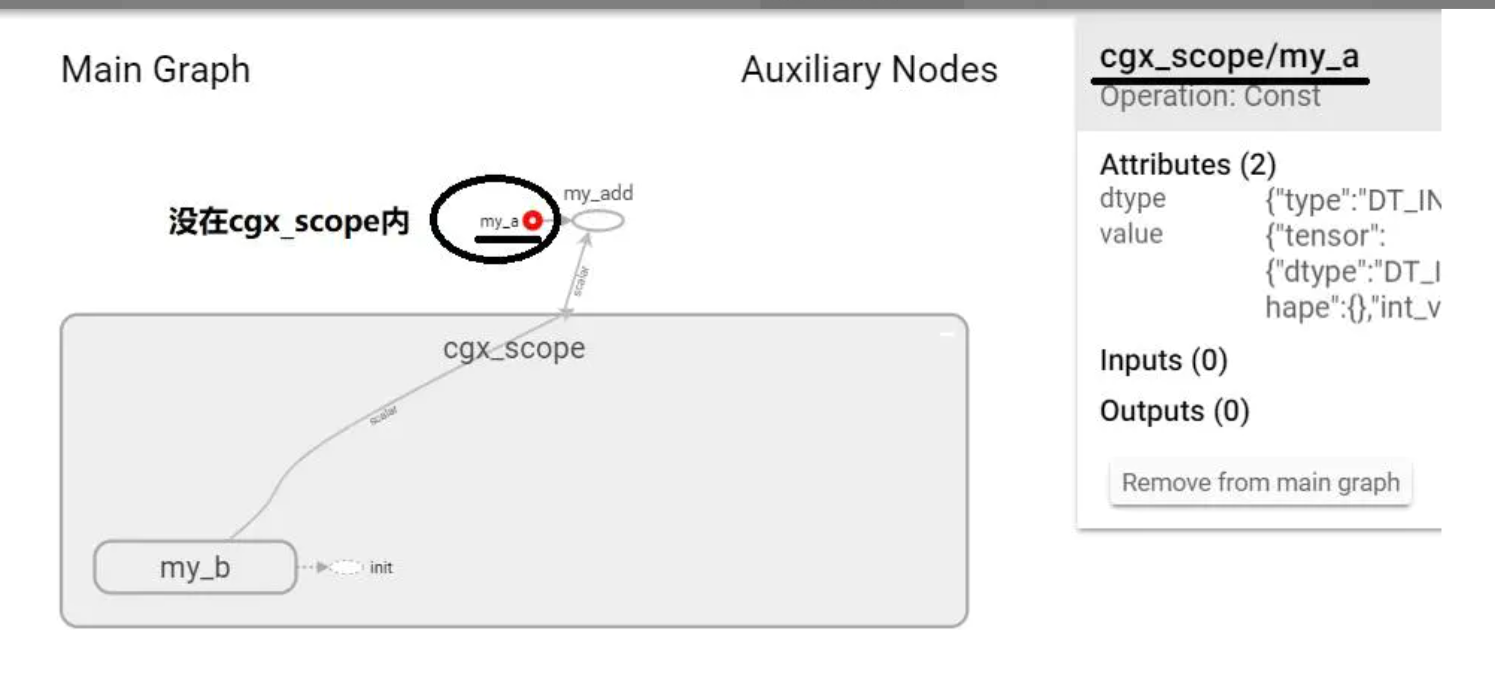
print("c.name = "+c.name)

# 输出结果

a.name = cgx\_scope/my\_a:0

b.name = cgx\_scope/my\_b:0

c.name = my\_add:0



7. tf.name\_scope()对由tf.get\_variable()产生的变量没有影响

with tf.name\_scope('cgx\_scope'):

    a = tf.get\_variable('my\_a',initializer=4)

    b = tf.Variable(2,name='my\_b')

c = tf.add(a,b,name='my\_add')

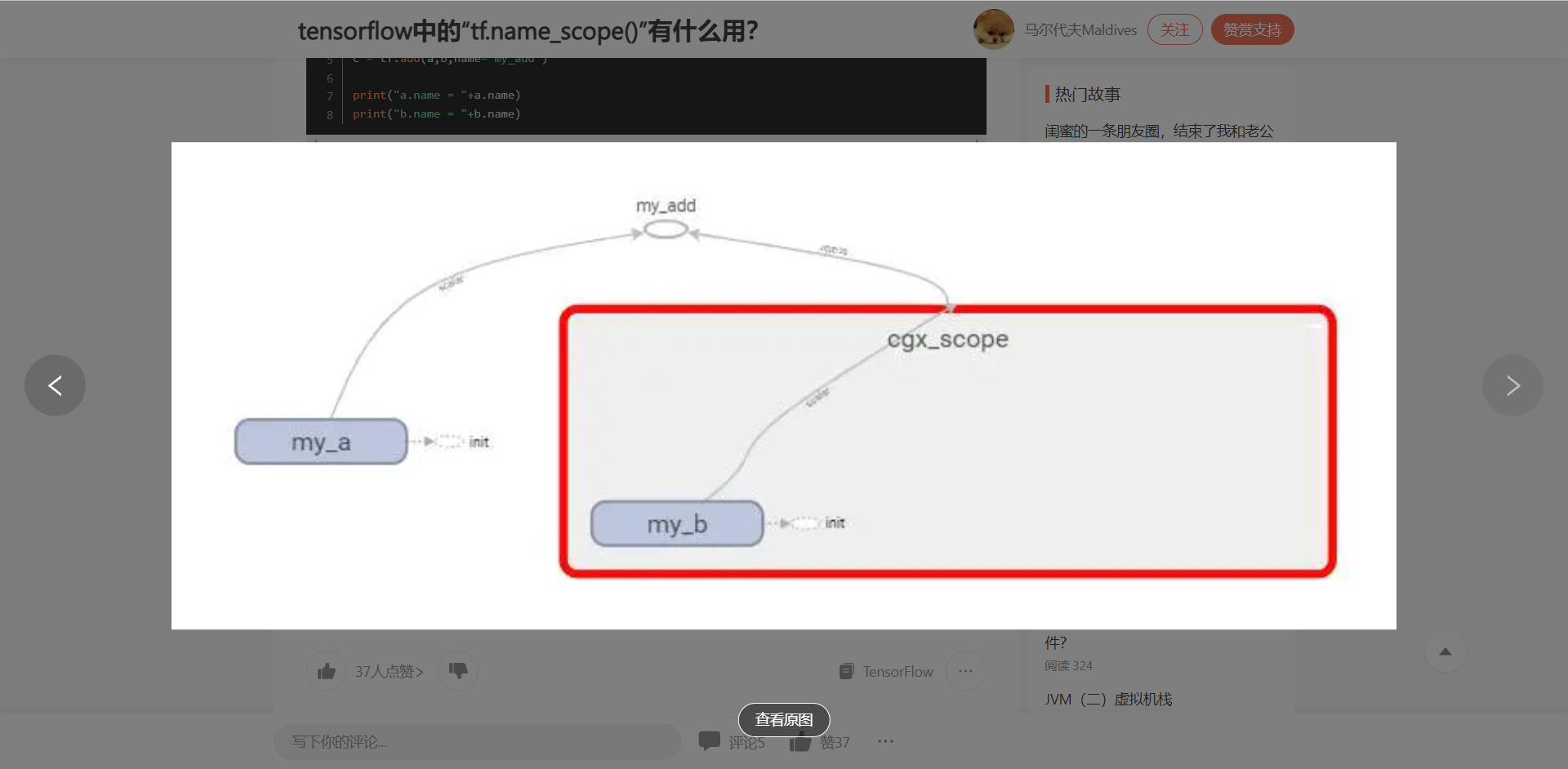
print("a.name = "+a.name)

print("b.name = "+b.name)

# 输出结果

a.name = my\_a:0

b.name = cgx\_scope/my\_b:0



# [tf.slice()函数详解](https://www.cnblogs.com/Kobaayyy/p/12540510.html#:~:text=tf.slice%20%28%29%E5%87%BD%E6%95%B0%E7%9A%84%E4%BD%9C%E7%94%A8%E5%B0%B1%E6%98%AF%E4%BB%8E%E5%BC%A0%E9%87%8F%E4%B8%AD%E6%8F%90%E5%8F%96%E6%83%B3%E8%A6%81%E7%9A%84%E5%88%87%E7%89%87%E3%80%82%20%E6%AD%A4%E6%93%8D%E4%BD%9C%E4%BB%8E%E7%94%B1begin%E6%8C%87%E5%AE%9A%E4%BD%8D%E7%BD%AE%E5%BC%80%E5%A7%8B%E7%9A%84%E5%BC%A0%E9%87%8Finput%E4%B8%AD%E6%8F%90%E5%8F%96%E4%B8%80%E4%B8%AA%E5%B0%BA%E5%AF%B8size%E7%9A%84%E5%88%87%E7%89%87.%E5%88%87%E7%89%87size%E8%A2%AB%E8%A1%A8%E7%A4%BA%E4%B8%BA%E5%BC%A0%E9%87%8F%E5%BD%A2%E7%8A%B6%2C%E5%85%B6%E4%B8%ADsize,%5Bi%5D%E6%98%AF%E4%BD%A0%E6%83%B3%E8%A6%81%E5%88%86%E5%89%B2%E7%9A%84input%E7%9A%84%E7%AC%ACi%E7%BB%B4%E7%9A%84%E5%85%83%E7%B4%A0%E7%9A%84%E6%95%B0%E9%87%8F.%E5%88%87%E7%89%87%E7%9A%84%E8%B5%B7%E5%A7%8B%E4%BD%8D%E7%BD%AE%20%28begin%29%E8%A1%A8%E7%A4%BA%E4%B8%BA%E6%AF%8F%E4%B8%AAinput%E7%BB%B4%E5%BA%A6%E7%9A%84%E5%81%8F%E7%A7%BB%E9%87%8F.%E6%8D%A2%E5%8F%A5%E8%AF%9D%E8%AF%B4%2Cbegin%20%5Bi%5D%E6%98%AF%E4%BD%A0%E6%83%B3%E4%BB%8E%E4%B8%AD%E5%88%86%E5%89%B2%E5%87%BA%E6%9D%A5%E7%9A%84input%E7%9A%84%E2%80%9C%E7%AC%ACi%E4%B8%AA%E7%BB%B4%E5%BA%A6%E2%80%9D%E7%9A%84%E5%81%8F%E7%A7%BB%E9%87%8F%E3%80%82)

tf.slice()函数的作用就是从张量中提取想要的切片。

def slice(input\_, begin, size, name=None):

...

return gen\_array\_ops.\_slice(input\_, begin, size, name=name)

input\_类型为一个tensor，表示的是输入的tensor，也就是被切的那个

begin是一个int32或int64类型的tensor，表示的是每一个维度的起始位置

size是一个int32或int64类型的tensor，表示的是每个维度要拿的元素数

name是操作的名称，可写可不写

return

返回一个和输入类型一样的tensor

t = tf.constant([[[1, 1, 1], [2, 2, 2]],

                 [[3, 3, 3], [4, 4, 4]],

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

tf.slice(t, [1, 0, 0], [1, 1, 3]) # begin = [1, 0, 0]

#结果

[[[3, 3, 3]]]