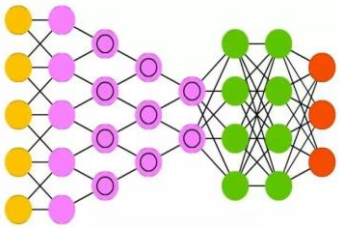
# 智能计算系统

# 第五章 编程框架机理

中国科学院计算技术研究所
陈云霁 研究员
cyj@ict.ac.cn

# Driving Example


输入 ▶ 建模 ▶ 实现 ▶ 运行 ▶ 输出


编程框架


Bang

第五章将学习到实现深度学习算法所使用的编程框架的工作机理

# 提纲

- TensorFlow设计原则
- TensorFlow计算图机制
- TensorFlow系统实现
- 驱动范例
- 编程框架对比

# 1、高性能

- TensorFlow中的算子，设计过程中已经针对底层硬件架构进行了充分的优化
- 针对生成的计算图，TensorFlow又提供了一系列的优化操作，以提升计算图的运行效率
- TensorFlow调度器可以根据网络结构特点，并发运行没有数据依赖的节点

```
1  import tensorflow as tf
2
3  a = tf.constant(1.0)
4  b = tf.constant(2.0)
5  c = tf.sin(a)
6  d = tf.cos(b)
7  e = tf.add(c, d)
8
9  with tf.Session() as sess:
10   sess.run(e)
```

张量c和d可以并发执行

# 2、易开发

- TensorFlow针对现有的多种深度学习算法，提取了大量的共性运算，并封装成算子
- 用户使用TensorFlow进行算法开发时，能够直接调用这些算子，很方便的实现算法

# 3、可移植

- TensorFlow可工作于各种类型的异构系统
- 对每个算子（例如矩阵乘法）需提供在不同设备上的不同底层实现
- 通过上述机制，使得统一的用户程序可以在不同硬件平台上执行。

# 提纲

- TensorFlow设计原则
- TensorFlow计算图机制
- TensorFlow系统实现
- 驱动范例
- 编程框架对比

# 1、计算图的自动求导

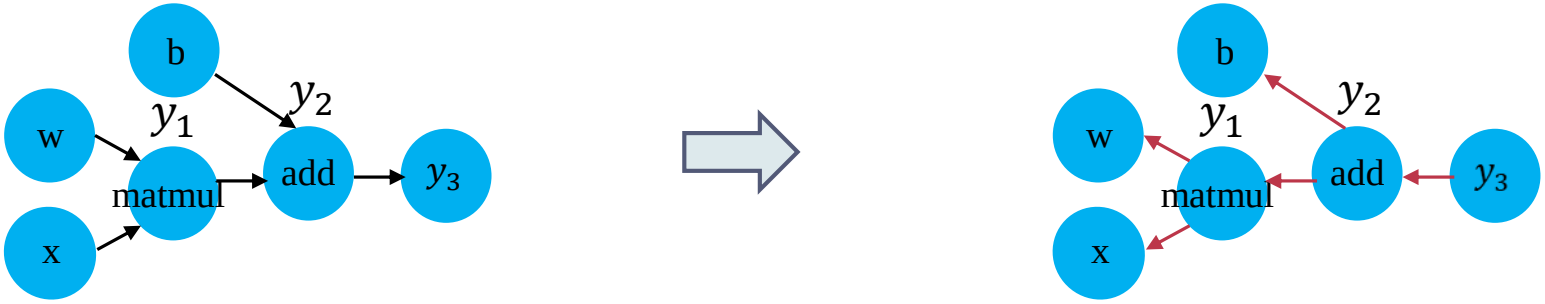
- 深度学习中通常采用梯度下降法来更新模型参数
- 梯度计算比较直观，但对于复杂模型，手动计算梯度非常困难
- 目前大部分深度学习框架均提供自动梯度计算功能
- 用户只需描述前向计算的过程，由编程框架自动推导反向计算图，完成导数计算

# 常用的求导方法

- 手动求解法
- 数值求导法
- 符号求导法
- 自动求导法

# 手动求解法

- 即传统的反向传播算法：手动用链式法则求解出梯度公式，代入数值，得到最终梯度值
- 缺点：
  - 对于大规模的深度学习算法，手动用链式法则进行梯度计算并转换成计算机程序非常困难
  - 需要手动编写梯度求解代码
  - 每次修改算法模型，都要修改对应的梯度求解算法



# 数值求导法

- 利用导数的原始定义求解

$$f'(x) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

- 优点：
  - 易操作
  - 可对用户隐藏求解过程
- 缺点：
  - 计算量大，求解速度慢
  - 可能引起舍入误差和截断误差

# 符号求导法

- 利用求导规则来对表达式进行自动操作，从而获得导数
- 常见求导规则：

$$\frac{d}{dx}\big(f(x)+g(x)\big)=\frac{d}{dx}f(x)+\frac{d}{dx}g(x)$$

$$\frac{d}{dx}f(x)g(x)=\left(\frac{d}{dx}f(x)\right)g(x)+f(x)\left(\frac{d}{dx}g(x)\right)$$

$$\frac{d}{dx}\frac{f(x)}{g(x)}=\frac{f'(x)g(x)-f(x)g'(x)}{g(x)^2}$$

- 缺点：表达式膨胀问题

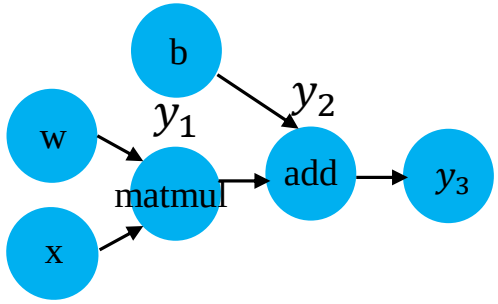
**Table 1** Iterations of the logistic map $l_{n+1}=4l_n(1-l_n)$, $l_1=x$ and the corresponding derivatives of $l_n$ with respect to $x$, illustrating expression swell.

| $n$ | $l_n$ | $\frac{d}{dx}l_n$ | $\frac{d}{dx}l_n$ (Optimized) |
|---|---|---|---|
| 1 | $x$ | 1 | 1 |
| 2 | $4x(1-x)$ | $4(1-x)-4x$ | $4-8x$ |
| 3 | $16x(1-x)(1-2x)^2$ | $16(1-x)(1-2x)^2-16x(1-2x)^2-64x(1-x)(1-2x)$ | $16(1-10x+24x^2-16x^3)$ |
| 4 | $64x(1-x)(1-2x)^2(1-8x+8x^2)^2$ | $128x(1-x)(-8+16x)(1-2x)^2(1-8x+8x^2)+64(1-x)(1-2x)^2(1-8x+8x^2)^2-64x(1-2x)^2(1-8x+8x^2)^2-256x(1-x)(1-2x)(1-8x+8x^2)^2$ | $64(1-42x+504x^2-2640x^3+7040x^4-9984x^5+7168x^6-2048x^7)$ |

# 自动求导法

- 数值求导法：一开始直接代入数值近似求解
- 符号求导法：直接对代数表达式求解，最后才代入问题数字
- 自动求导法：介于数值求导和符号求导的方法

- 计算图结构天然适用于自动求导
  - 计算图将多输入的复杂计算表达成了由多个基本二元计算组成的有向图，并保留了所有中间变量，有助于程序自动利用链式法则进行求导



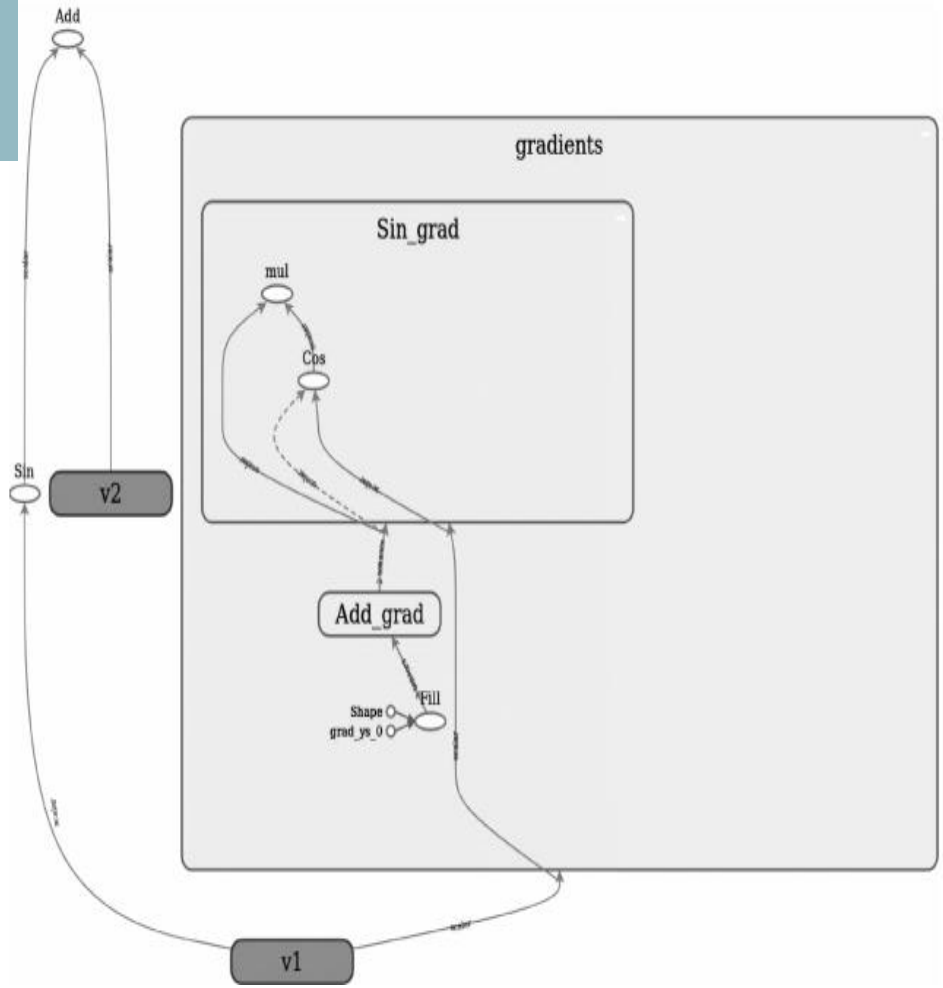

- 优点
  - 灵活，可以完全向用户隐藏求导过程
  - 只对基本函数运用符号求导法，因此可以灵活结合编程语言的循环结构、条件结构等

- TensorFlow中注册Sin(x)函数的反向求导方法

```
@ops.RegisterGradient("Sin")
def _SinGrad(op, grad):
"""Returns grad * cos(x)."""
x = op.inputs[0]
with ops.control_dependencies([grad]):
    x = math_ops.conj(x)
return grad * math_ops.cos(x)
```

- TensorFlow会自动生成对应的反向计算节点，并将其加入到计算图中

```
v1 = tf.Variable(0.0, name="v1")
v2 = tf.Variable(0.0, name='v2')
loss = tf.add(tf.sin(v1), v2)
sgd = tf.train.GradientDescentOptimizer(0.01)
grads_and_vars = sgd.compute_gradients(loss)
```



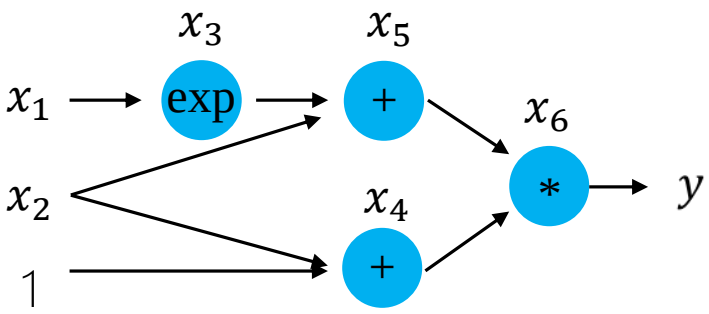
- 计算分两步执行：
  - 1）原始函数建立计算图，数据正向传播，计算出中间节点$x_j$，并记录计算图中的节点依赖关系
  - 2）反向遍历计算图，计算输出对于每个节点的导数

$$\bar{x}_i = \frac{\partial y_j}{\partial x_i}$$

- 对于前向计算中一个数据($x_i$)连接多个输出数据($y_j$、$y_k$)的情况，自动求导中，将这些输出数据相对于该数据的导数累加

$$\bar{x}_i = \bar{y}_j \frac{\partial y_j}{\partial x_i} + \bar{y}_k \frac{\partial y_k}{\partial x_i}$$

# 示例---$f(x_1, x_2) = (e^{x_1} + x_2)(x_2 + 1)$




$x_1 = 3$

$x_2 = 2$

$x_3 = e^{x_1} = 20.086$

$x_5 = x_3 + x_2 = 22.086$

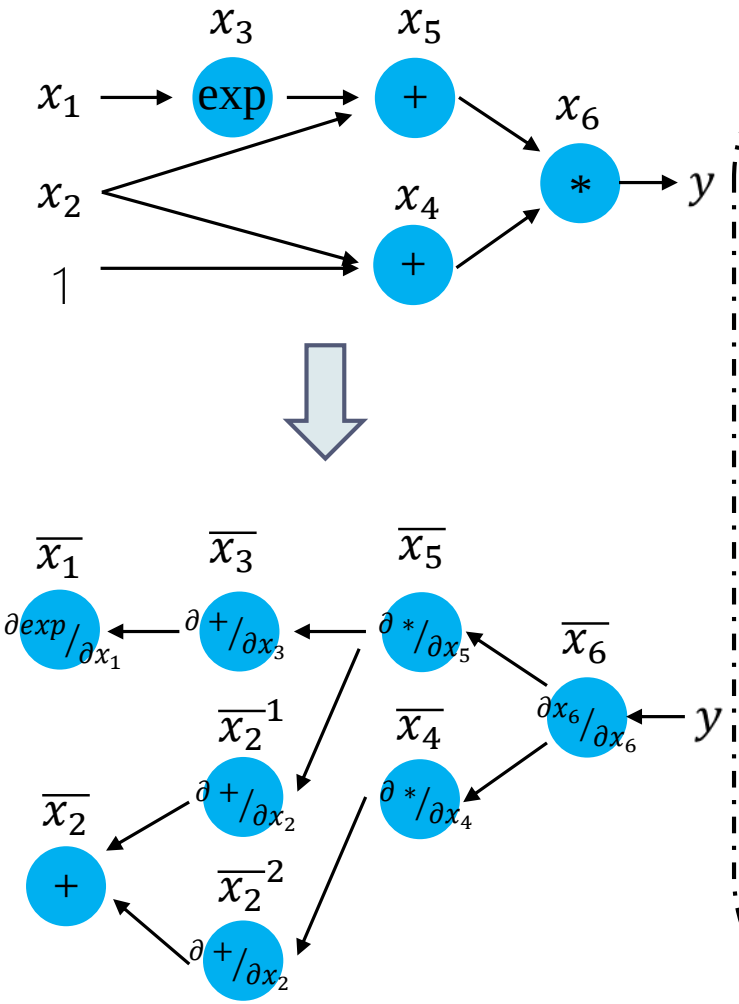
$x_4 = x_2 + 1 = 3$

$x_6 = x_4 * x_5 = 66.258$

$y = x_6 = 66.258$

前向计算

# 示例---$f(x_1, x_2) = (e^{x_1} + x_2)(x_2 + 1)$




$$\overline{x_2} = \overline{x_2}^1 + \overline{x_2}^2 = 25.086$$

$$\overline{x_1} = \overline{x_3} * \frac{\partial x_3}{\partial x_1} = \overline{x_3} * x_3 = 60.258$$

$$\overline{x_2}^2 = \overline{x_4} * \frac{\partial x_4}{\partial x_2} = \overline{x_4} * 1 = 22.086$$

$$\overline{x_2}^1 = \overline{x_5} * \frac{\partial x_5}{\partial x_2} = \overline{x_5} * 1 = 3$$

$$\overline{x_3} = \overline{x_5} * \frac{\partial x_5}{\partial x_3} = \overline{x_5} * 1 = 3$$

$$\overline{x_4} = \overline{x_6} * \frac{\partial x_6}{\partial x_4} = \overline{x_6} * x_5 = 22.086$$

$$\overline{x_5} = \overline{x_6} * \frac{\partial x_6}{\partial x_5} = \overline{x_6} * x_4 = 3$$

$$\overline{x_6} = \frac{\partial y}{\partial x_6} = 1$$

反向计算

# 对比

| 方法 | 对图的遍历次数 | 精度 | 备注 |
| --- | --- | --- | --- |
| 手动求解法 | NA | 高 | 实现复杂 |
| 数值求导法 | $n_I+1$ | 低 | 计算量大，速度慢 |
| 符号求导法 | NA | 高 | 表达式膨胀 |
| 自动求导法 | $n_O+1$ | 高 | 对输入维度较大的情况优势明显 |

其中：

$n_I$：要求导的神经网络层的输入变量数，包括$w$、x、b

$n_O$：神经网络层的输出个数

# 2、检查点机制

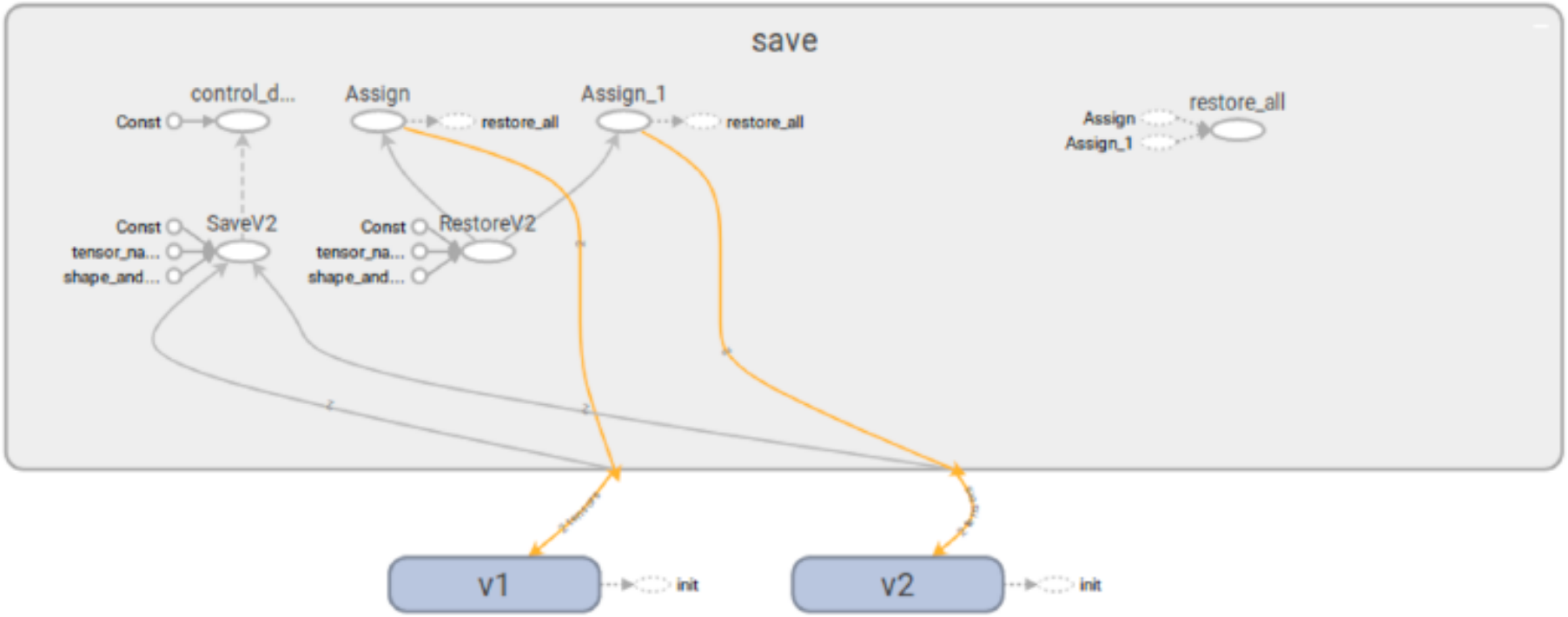
- 在模型训练过程中，使用tf.train.Saver()来保存模型中的所有变量

```
import tensorflow as tf
weights = tf.Variable(tf.random_normal([30,60],stddev=0.35),name="weights")
w2 = tf.Variable(weights.initialized_value(),name="w2")

#实例化saver对象
saver = tf.train.Saver()

with tf.Session() as sess:
  sess.run(tf.global_variables_initializer())
  for step in xrange(1000000):
    #执行模型训练
    sess.run(training_op)
    if step % 1000 == 0:
        # 将训练得到的变量值保存到检查点文件中
        saver.save(sess, './ckpt/my-model')
```

# 恢复模型

- 当需要基于某个checkpoint继续训练模型参数时，需要从.ckpt文件中恢复出已保存的变量
- 同样使用tf.train.Saver()来恢复变量，恢复变量时不需要先初始化变量

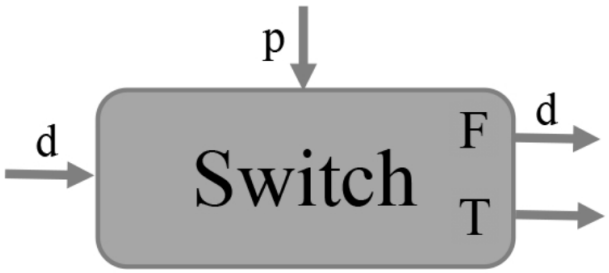
```
import tensorflow as tf

weights = tf.Variable(tf.random_normal([30,60],stddev=0.35),name="weights")
w2 = tf.Variable(weights.initialized_value(),name="w2")

#模型路径只需要给出文件夹名称
model_path = "./ckpt"

#实例化saver对象
saver = tf.train.Saver()

with tf.Session() as sess:
  #找到存储变量值的位置
  ckpt = tf.train.latest_checkpoint(model_path)
  #恢复变量
  saver.restore(sess,ckpt)
  print(sess.run(weights))
  print(sess.run(w2))
```




- TensorFlow 通过向计算图中插入Save 节点及其关联节点来完成保存模型的功能
- 在恢复模型时，也是通过在计算图中插入Restore节点及其关联节点来完成

# 3、TensorFlow中的控制流

- Tensorflow中使用控制流算子来实现不同复杂控制流场景
- 通过引入少量的简单基础操作，为多样的Tensorflow应用提供丰富的控制流表达
- 在Tensorflow中，每一个操作都会在一个执行帧中被执行，控制流操作负责创建和管理这些执行帧




- Switch：一个*Switch*操作根据控制输入p的布尔值，将一个输入张量d推进到某一个输出（二选一）。


- Merge：*Merge*操作将它的其中一个输入推向输出。当一个Merge操作的任意一个输入准备好之后，Merge操作就会执行。

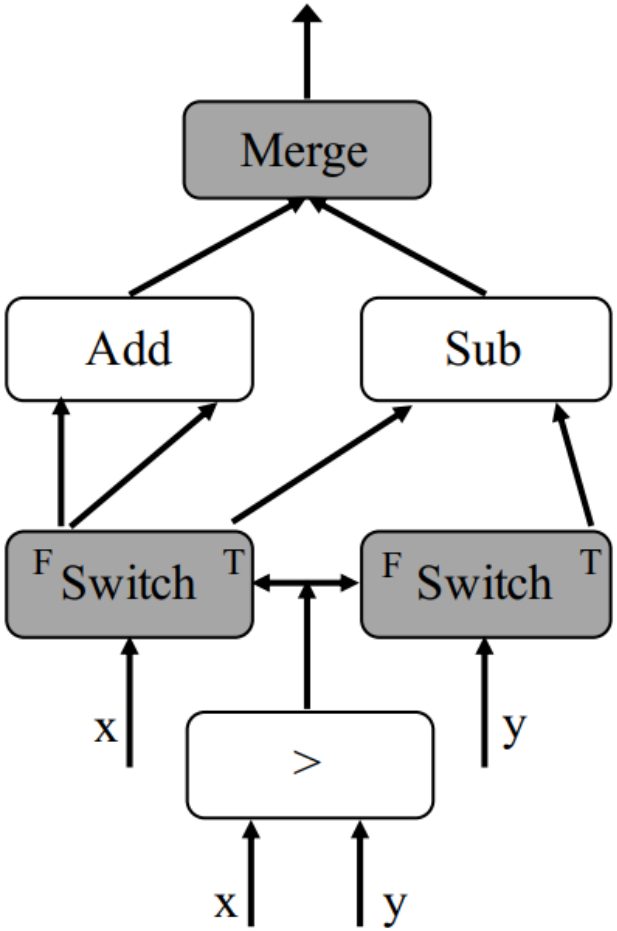

- Enter(name)： *Enter*操作将它的输入推向名为name的执行帧。


- Exit： *Exit*操作，将一个张量从一个子执行帧推向它的父执行帧。它的作用是将张量从子执行帧返回给父执行帧。


- NextIteration： *NextIteration*操作将一个张量从当前执行帧的一轮迭代传递到下一轮迭代。
- 在一个执行帧中可能会有多个NextIteration操作。当执行帧的第N轮执行的第一个NextIteration操作开始执行时，Tensorflow的运行时开始执行第N+1轮的迭代。
- 当更多的张量通过了NextIteration操作进入新的执行轮次时，新执行轮次中更多的操作就会开始运行。当输入准备完成之后，NextIteration操作开始执行。

# 控制流结构的编译---条件表达式

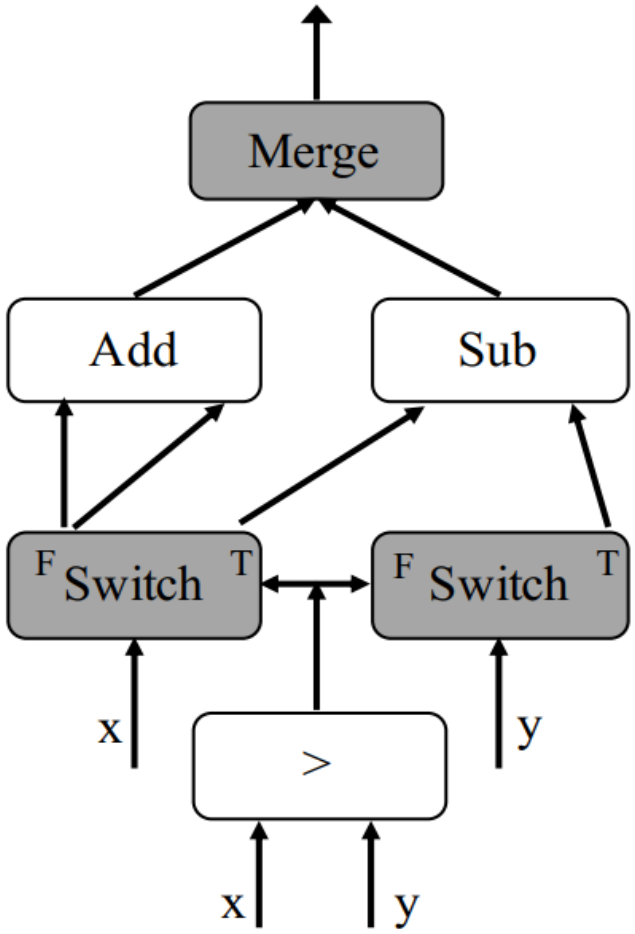
- cond(pred, true_fn, false_fn)




```
tf.cond(x>y, lambda: tf.subtract(x, y),
             lambda: tf.add(x, x))
```

- cond(pred, true_fn, false_fn)

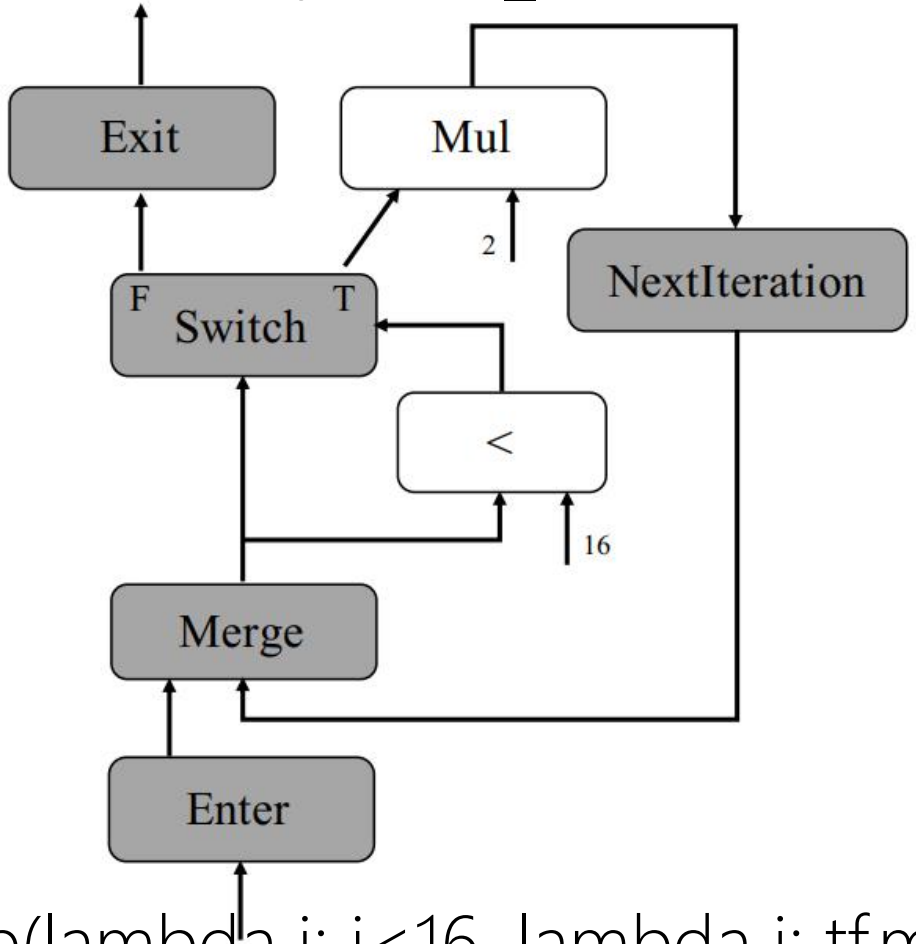
```
# 添加Switch节点
switch_f, switch_t = Switch(pred, pred)
# 创建Switch为真时的环境
ctx_t = MakeCondCtx(pred, switch_t, branch=1)
# 创建Switch为真时的计算图
res_t = ctx_t.Parse(true_fn)
# 创建Switch为假时的环境
ctx_f = MakeCondCtx(pred, switch_f, branch=0)
# 创建Switch为假时的计算图
res_f = ctx_f.Parse(false_fn)
# 将两个分支结果合并到一起
rets = [Merge([f, t]) for (f, t) in zip(res_f, res_t)]
```




tf.cond(x>y, lambda: tf.subtract(x, y),
lambda: tf.add(x,

# 控制流结构的编译---循环操作

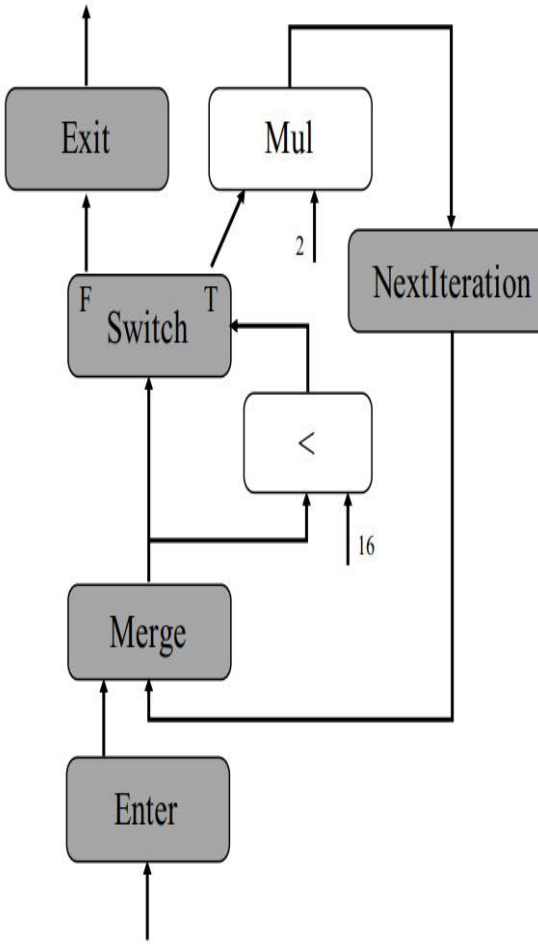
- while_loop(pred, body, loop_vars)




- tf.while_loop(lambda i: i<16, lambda i: tf.multiply(i,2), [4])

- while_loop(pred, body, loop_vars)

```
# 创建环境
while_ctx = WhileContext()
while_ctx.Enter()
# 为每个循环变量添加一个Enter节点
enters = [Enter(x, frame_name) for x in loop_vars]
# 添加Merge节点，Merge节点的第二个输入稍后会被更新
merges = [Merge([x, x]) for x in enters]
# 构建循环子图
pred_results = pred(*merges)
# 添加Switch节点
switchs = [Switch(x, pred_result) for x in merges]
# 构建循环体
body_res = body(*[x[1] for x in switchs])
# 添加NextItreation节点
nexts = [NextIteration(x) for x in body_res]
# 构建循环迭代
for m, n in zip(merge_vars, nexts):
    m.op.update(1, n)
# 添加退出节点
exits = [Exit(x[0]) for x in switchs]
while_ctx.Exit()
```

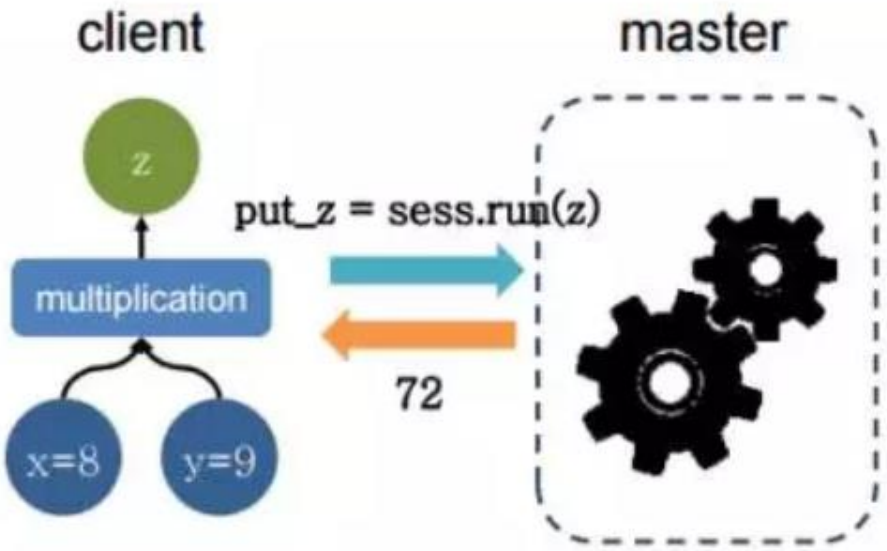


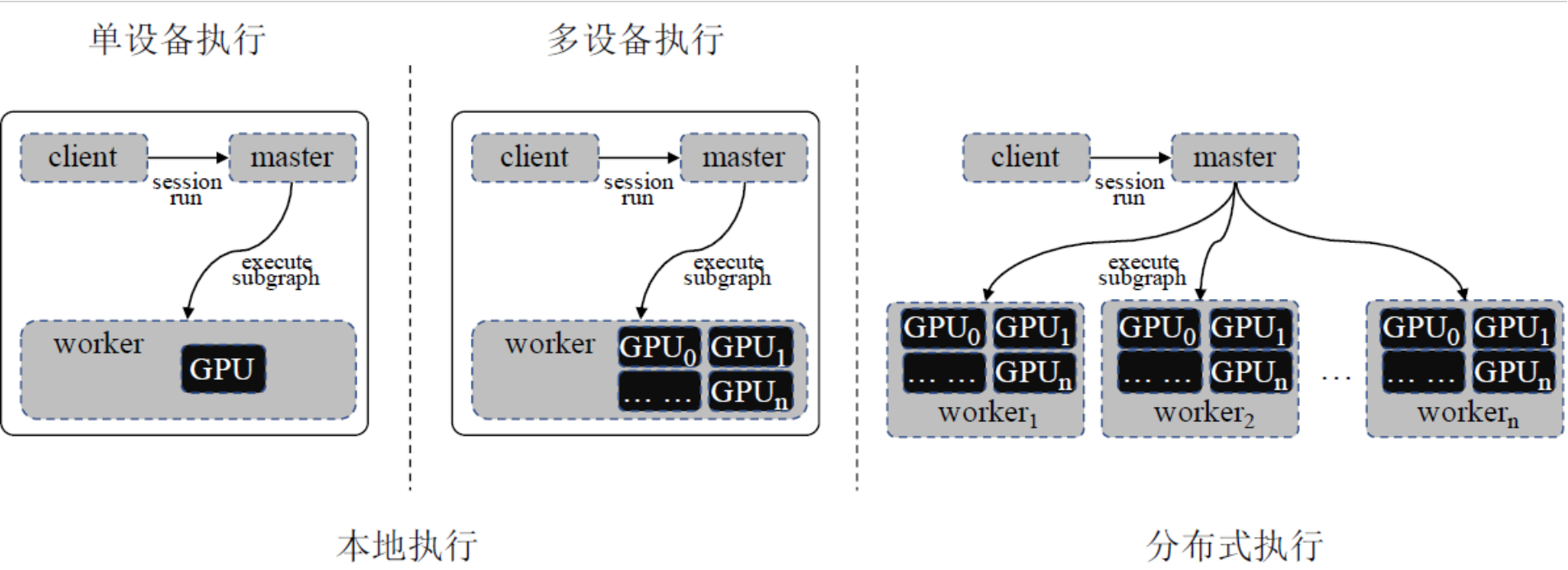
# 4、计算图的执行模式

- client：通过session接口与master和worker接口通信。worker可以是一个，也可以是多个。
- master：控制所有的worker按照计算图执行。
- worker：每一个worker负责一个或多个计算设备的仲裁访问，并根据master的指令，执行这些计算设备中的计算图节点。
- 设备：可以是CPU核或GPU卡。

# 简单示例

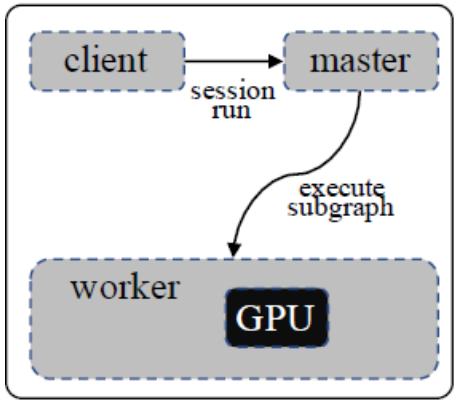
```
1  import tensorflow as tf
2
3  x = tf.constant(8)
4  y = tf.constant(9)
5  z = tf.multiply(x, y)
6
7  with tf.Session() as sess:
8    put_z = sess.run(z)
```




单设备执行
多设备执行
client
master
session run
execute subgraph
worker
GPU
GPU0
GPU1
GPUn
worker1
worker2
workern
本地执行
分布式执行

# 本地单设备执行

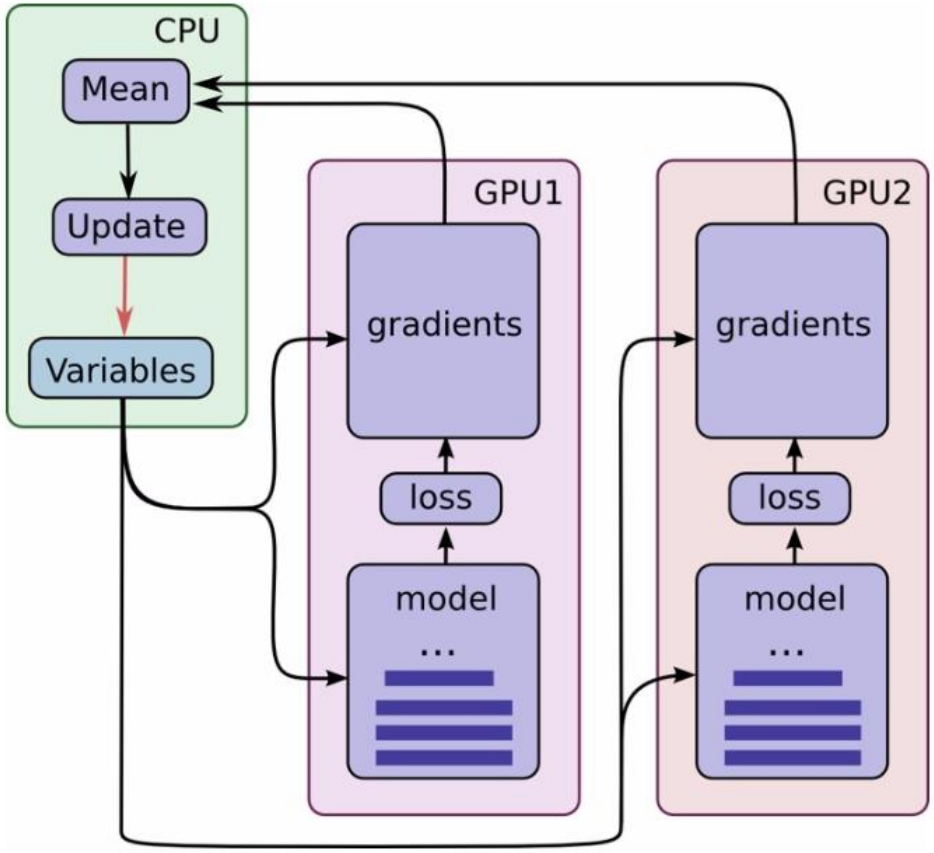
- 首先考虑最简单的执行场景：一个worker进程中仅包含一个设备的情况。
- 在该情况下：
  - 计算图按照节点之间的依赖关系顺序执行
  - 每个节点有一个计数器，记录了其依赖节点中尚未执行的节点数量，当一个节点执行完成，则其所有依赖节点的计数器计数递减
  - 当计数器计数为0时，则该节点可以执行，并将其添加到就绪队列中



# 本地多设备执行

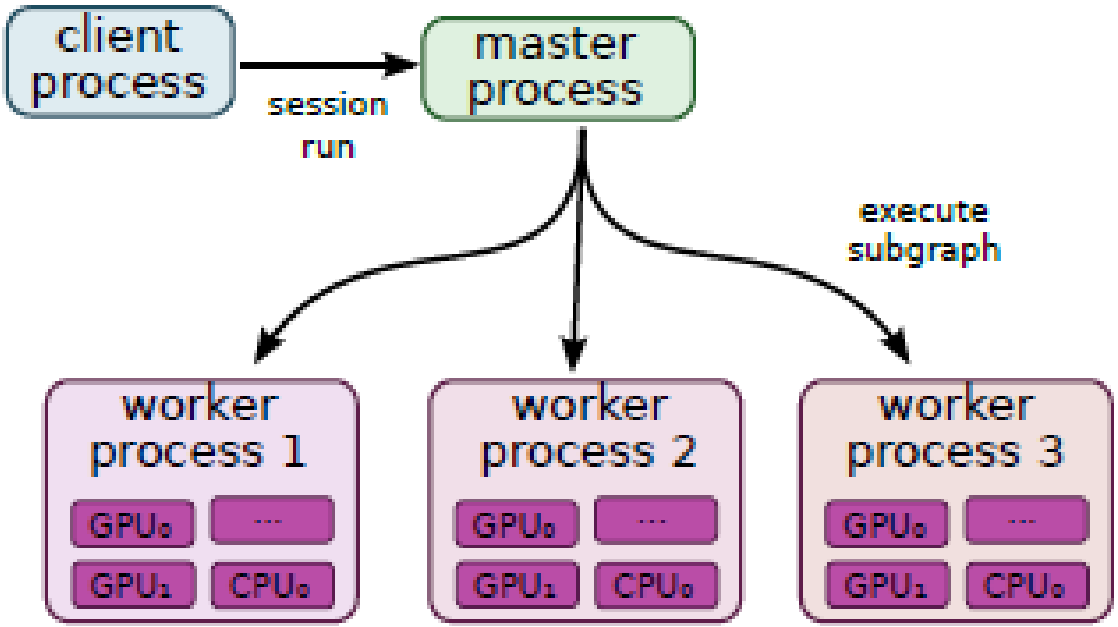
- CPU作为参数服务器，用于保存参数和变量、计算梯度平均等
- GPU作为worker，用于模型训练

  - 1、在本地将数据切分为一个一个batch
  - 2、把数据分别送到多个GPU进行模型训练，每个GPU分配到不同数据
  - 3、每个GPU分别训练，求loss得到梯度，把梯度送回到CPU进行模型平均
  - 4、CPU接收GPU传来的梯度，进行梯度平均，更新参数
  - 5、GPU更新参数
  - 6、重复2-5直到模型收敛



# 分布式执行

- 该模式下，client、master和worker可以工作于不同机器上的不同进程中
- 兼容本地多设备执行模式



# 5、计算图本地执行

- 1、计算图剪枝
- 2、计算图分配
- 3、计算图优化
- 4、计算图切分

# 计算图剪枝

- 目的：得到本地运行的最小子图
- 包括：
  - 为输入输出建立与外界的交互
  - 去除与最终输出节点无关的节点和边

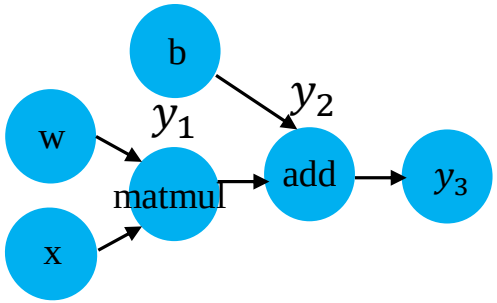
```
1  import tensorflow as tf
2
3  a = tf.placeholder(dtype=tf.float32)
4  b = tf.placeholder(dtype=tf.float32)
5
6  c = tf.add(a, b)
7  d = tf.sin(a)
8  e = tf.multiply(c, d)
9  f = tf.cos(c)
10
11 with tf.Session() as sess:
12   res = sess.run(f, feed_dict={a:2, b:3})
```

- 目的：得到本地运行的最小子图
- 包括：
  - 为输入输出建立与外界的交互
    - 通过FunctionCallFrame函数调用帧来解决输入输出值传递的问题
    - 在每个输入节点前插入Arg节点，所有的输入节点连接到Source节点上，并通过控制依赖边相连；
    - 在每个输出节点后面加入RetVal节点，所有的输出节点连接到Sink节点上，也通过控制依赖边相连，最终形成完整的计算图。

- 目的：得到本地运行的最小子图
- 包括：
  - 为输入输出建立与外界的交互
  - 去除与最终输出节点无关的节点和边
    - 从输出节点开始进行宽度搜索遍历，删除没有接触到的节点和边
    - 将每个连通图中入度为0的节点通过控制依赖边与source节点相连，出度为0的节点通过控制依赖边和sink节点相连

# 计算图分配

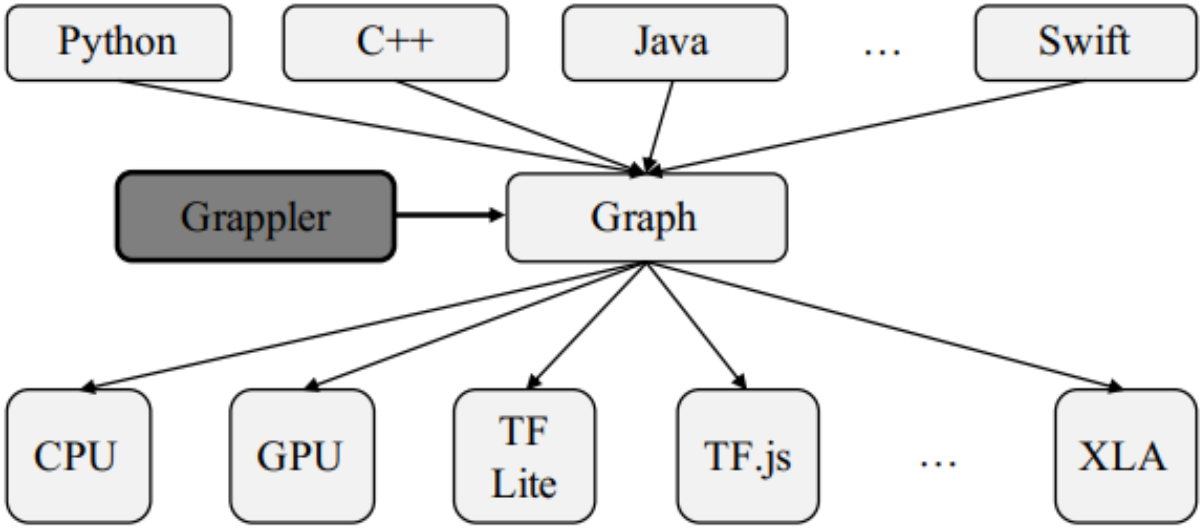
- 问题：多设备运行环境中，对计算图中的每个节点如何分配计算设备
  - 目标：保证计算的快速执行



# 对计算图中的每个节点如何分配计算设备

- 算法输入：cost model。包含图中每个节点的输入输出tensor的数据量、每个节点的预计计算时间
- 算法执行过程：
  - 1、从计算图起始点开始遍历
  - 2、对于遍历中的每个节点，考虑其可行的设备集合
  - 3、如果设备不提供实现特定操作的内核，则设备不可行
  - 4、如果某个节点具有多个可行设备，则采用贪心算法，检查该节点在所有可行设备上的完成时间，将最快完成的设备分配给该节点
  - 5、重复2-4直到遍历完成整个图

# 计算图优化




- TensorFlow中的图优化由Grappler模块来实现
- 通过图优化，可以根据不同的硬件结构调整计算调度策略，从而获得更快的计算速度和更高的硬件利用率
- 也能减少推理过程中所需的峰值内存，从而运行更大的模型

- ConstFold：常量折叠
- Arithmetic：算术简化
- Layout：布局优化
- Remapper：算子融合

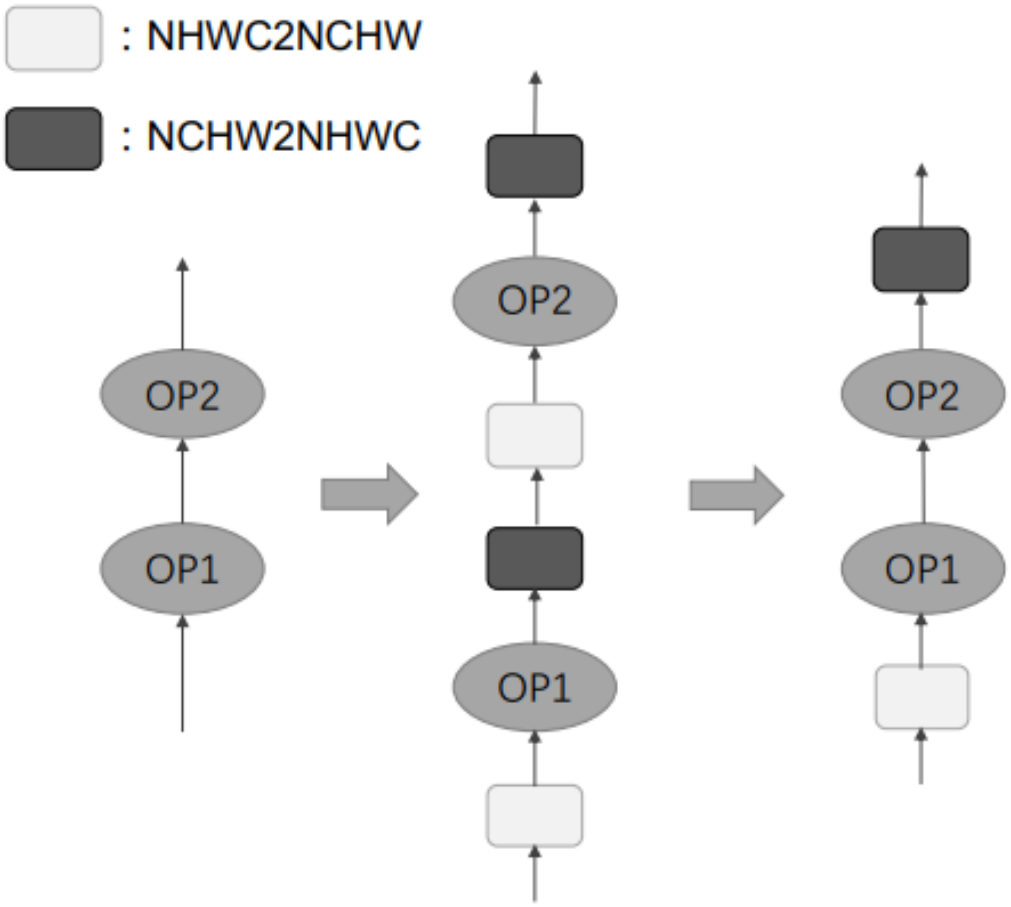
# 常量折叠（ConstFold）

- 有的常数节点可以被提前计算，用得到的结果生成新的节点来代替原来的常数节点
- 主要由三个关键函数组成：
  - MaterializeShapes：处理与Shape相关的节点
  - FoldGraph：对每个节点的输入进行检测，如果均为Const节点，则提前计算出值来完整替换当前节点
  - SimplifyGraph：简化节点中的常量运算，如：
    - Mul(c1, Mul(tensor, c2)) -> Mul(tensor, c1 * c2)，
    - Concat([tensor1, c1, c2, tensor2]) )) -> Concat([tensor1, Concat([c1, c2]), tensor2])，
    - Zeros(tensor_shape) - tensor1 -> Neg(tensor1)

# 算术优化（Arithmetic）

- 包含两个部分：公共子表达式消除、算术简化
  - tensor + tensor + tensor + tensor -> 4 *tensor。
  - AddN(tensor * c1, c2 * tensor, tensor * c3) -> tensor * AddN(c1+c2+c3)
  - (mat1 + s1) + (mat2 + s2) -> (mat1 + mat2) + (s1 + s2)
  - 去除冗余计算： 当g指代取反或取倒数这类操作时， g(g(h)) 转化成 h

# 布局优化（Layout）



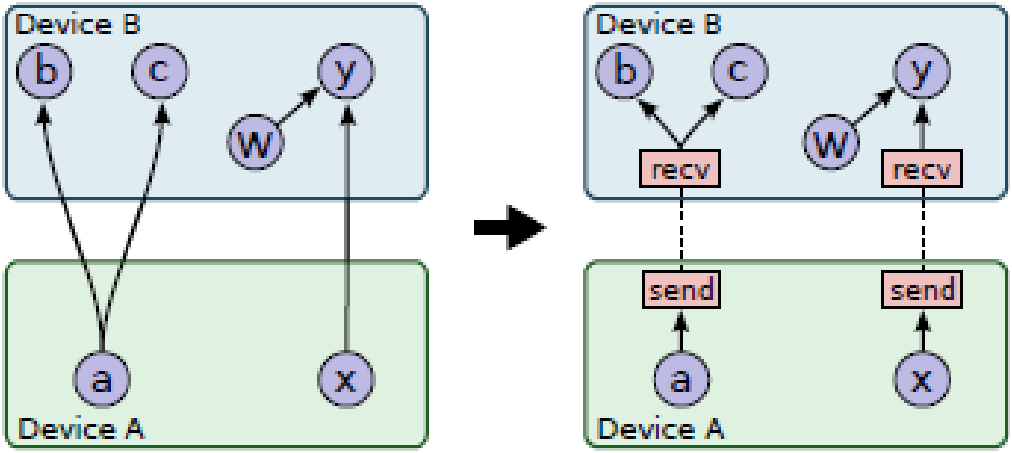

- TensorFlow中默认采用NHWC格式，而GPU中使用NCHW
- 两个连续的GPU计算节点之间的连续NCHW2NHWC和NHWC2NCHW转换应互相抵消去除

# 重映射（Remapper）

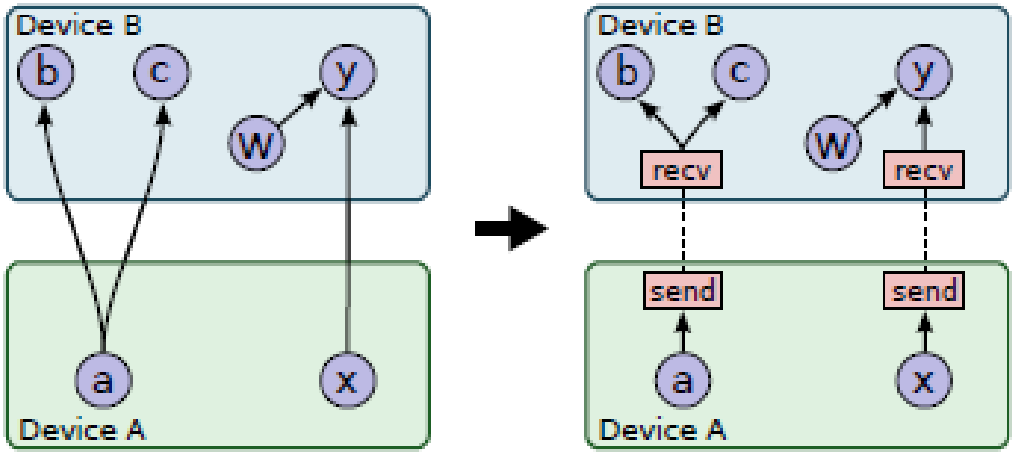
- 算子融合，将出现频率较高的子图用一个单独算子来替代，提高计算效率
- 可以进行单算子替换的例子包括：
  - Conv2D + BiasAdd + Activation
  - Conv2D + FusedBatchNorm + Activation
  - MatMul + BiasAdd + Activation
- 好处：
  - 消除子图调度开销
  - 计算Conv2D + BiasAdd时，Conv2D的数据处理是分块进行的，融合后的BiasAdd也可以在片上存储里进行

# 计算图切分和设备通信

- 完成每个节点的设备分配后，将整个计算图按照所分配设备分成若干子图，每个设备一张子图
- 对于跨设备通信的边，执行如下操作：
  - 1、将跨设备的边删掉
  - 2、在设备边界插入send或recv节点
  - 3、在设备A对应的子图中，增加x节点到send节点的边
  - 4、在设备B对应的子图中，增加recv节点到y节点的边



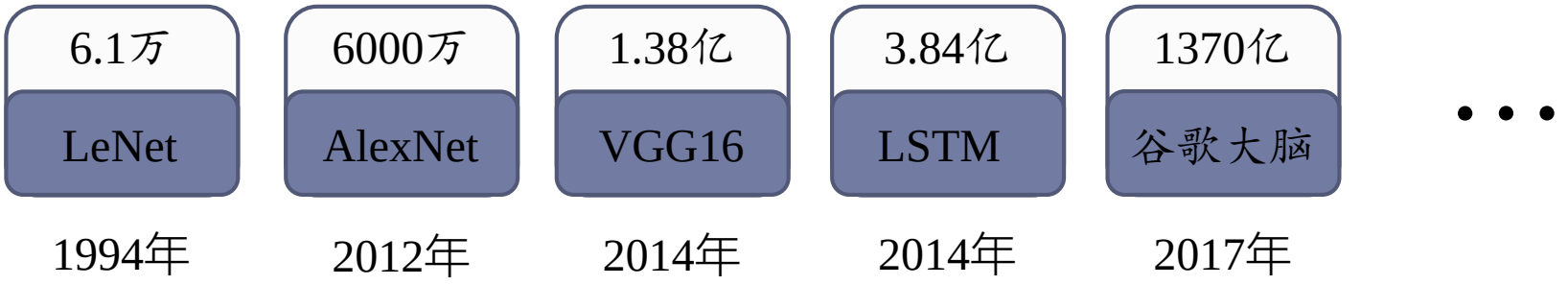
- 5、插入send和recv节点时，规定单个设备上特定张量的多个用户使用单个recv节点，如节点b、c，从而确保所需的张量在设备之间只传输一次




- 6、执行计算图时，通过send和recv节点来实现跨设备的数据传输

# 6、计算图的分布式执行

- 神经网络规模及数据规模指数型增加




- 为了有效提高神经网络训练效率，降低训练时间，在模型训练中普遍采用分布式技术
- 分布式技术：将一个大的神经网络模型，拆分成许多小的部分，同时分配在多个节点上进行计算
- 目前主流的深度学习框架均支持分布式技术

# 分布式通信

- 分为两类：点到点通信（Point-to-Point Communication）和集合通信（Collective Communication）
- TensorFlow中实现了集合通信的基本算子：
  - all_sum：将所有的输入张量进行累加操作，并将累加结果广播给所有的输出张量。
  - all_prod：将所有的输入张量进行乘法操作，并将乘法结果广播给所有的输出张量。
  - all_min：将所有的输入张量进行取最小值操作，并将该结果广播给所有的输出张量。
  - all_max：将所有的输入张量进行取最大值操作，并将该结果广播给所有的输出张量。
  - reduce_sum：将所有的输入张量进行累加操作，并返回这个结果。
  - broadcast：将输入张量广播给所有的设备。

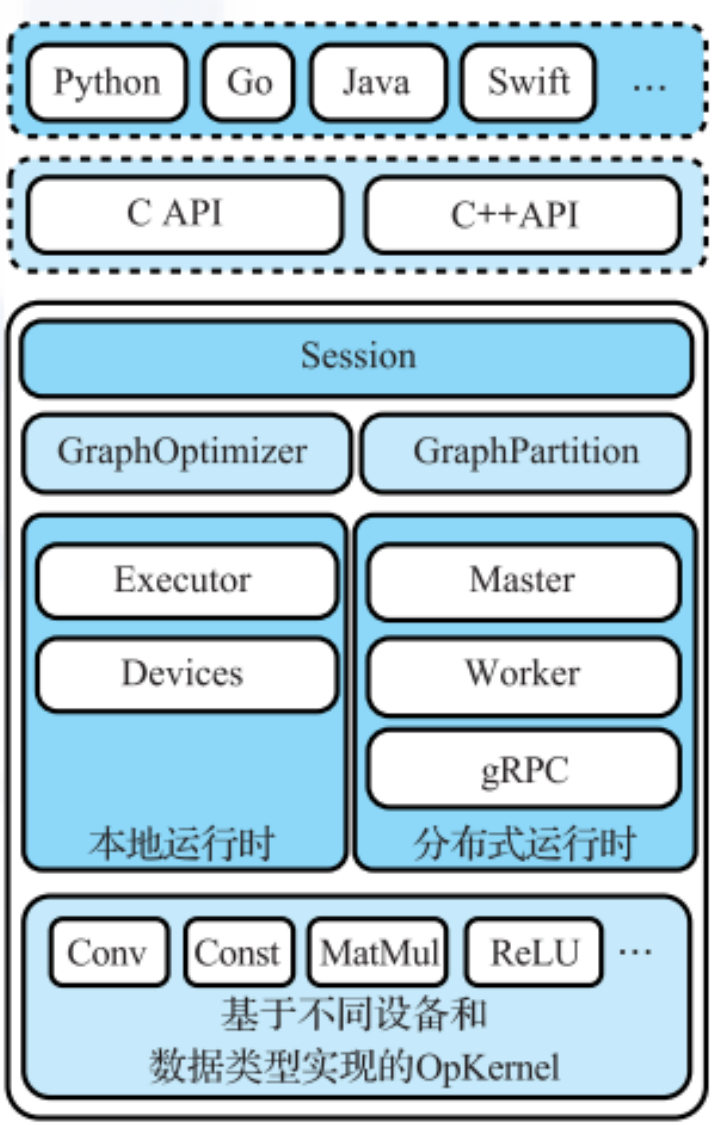
# 容错机制

- 为了确保分布式系统的稳定性，TensorFlow中增加了错误检查和容错机制
- 一方面检查Send和Recv节点传输的正确性，一方面定期检查每个工作机的状态
- 检查到错误时，计算图执行过程会停止并重启
- TensorFlow在训练过程中会保存中间状态，用于立即恢复到出错前的状态。

# 提纲

- TensorFlow设计原则
- TensorFlow计算图机制
- TensorFlow系统实现
- 驱动范例
- 编程框架对比

# 1、整体架构




- 面向各个语言的语言包
- C/C++ API
- 后端代码

# 2、计算图执行模块

- Session 是用户和TensorFlow运行时的接口。在Session接收到输入数据时，便可开始运行
- 一般情况下，每个设备会有一个执行器（Executor），负责本设备上子计算图的执行
- Run 函数是Session 执行的核心逻辑，在其中完成计算图的执行，包括传参、运行和返回

```
1 import tensorflow as tf
2 a = tf.placeholder(tf.int32)
3 b = tf.placeholder(tf.int32)
4 c= tf.multiply(a,b)
5 with tf.Session() as sess:
6     print(sess.run(c,feed_dict = {a:100,b:200}))
7
```

# 计算图执行前的传参过程

```
Status DirectSession::Run(const RunOptions& run_options, const NamedTensorList& inputs,
        const std::vector<string>& output_names, const std::vector<string>& target_nodes,
        std::vector<Tensor>* outputs, RunMetadata* run metadata) {
  // 提取输入张量名字input_tensor_names、input_size
  std::vector<string> input_tensor_names;
  input_tensor_names.reserve(inputs.size());
  size_t input_size = 0;
  for (const auto& it : inputs) {
    input_tensor_names.push_back(it.first);
    input_size += it.second.AllocatedBytes();
  }
  metrics::RecordGraphInputTensors(input_size);

  //创建或者得到执行器。如果已经存在则直接获取，不存在则创建。
  //Executor将执行graph计算操作，多个Executor可以并行计算，在feed批处理计算数据时非常有用。
  ExecutorsAndKeys* executors_and_keys;
  RunStateArgs run_state_args(run_options.debug_options());
  run_state_args.collective_graph_key = run_options.experimental().collective_graph_key();
  TF_RETURN_IF_ERROR(GetOrCreateExecutors(input_tensor_names, output_names,
      target_nodes,  &executors_and_keys, &run_state_args));
  {
    mutex_lock l(collective_graph_key_lock_);
    collective_graph_key_ = executors_and_keys->collective_graph_key; }
```

```
25  // 设置函数调用帧参数，用来处理执行器的输入与输出
26  FunctionCallFrame call_frame(executors_and_keys->input_types, executors_and_keys->output_types);
27
28  // 执行器从feed中获取输入，并将执行结果写入到fetch中
29  gtl::InlinedVector<Tensor, 4> feed_args(inputs.size());
30  for (const auto& it : inputs) {
31    if (it.second.dtype() == DT_RESOURCE) {
32      Tensor tensor_from_handle;
33      TF_RETURN_IF_ERROR(ResourceHandleToInputTensor(it.second, &tensor_from_handle));
34      feed_args[executors_and_keys->input_name_to_index[it.first]] = tensor_from_handle;
35    } else {
36      feed_args[executors_and_keys->input_name_to_index[it.first]] = it.second;
37    }
38  }
39  const Status s = call_frame.SetArgs(feed_args);
40  ... ...
41 }
```

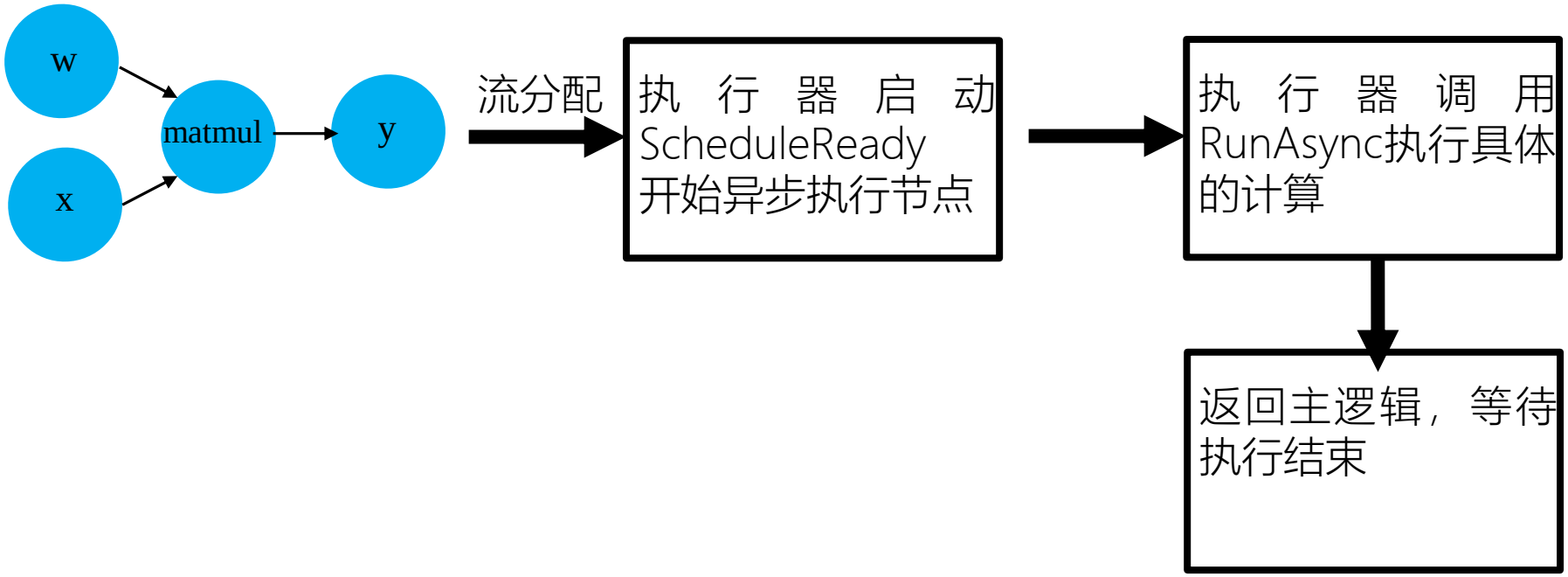
# RunInternal函数

```
Status DirectSession::RunInternal(int64 step_id, const RunOptions& run_options,
        CallFrameInterface* call_frame, ExecutorsAndKeys* executors_and_keys,
        RunMetadata* run_metadata, const thread::ThreadPoolOptions& threadpool_options) {
  // RunState 用于标记运行状态，构建 IntraProcessRendezvous 用于本地Tensor管理
  run_state.rendez = new IntraProcessRendezvous(device_mgr_.get());

  // 开始并行执行器
  const size_t num_executors = executors_and_keys->items.size();
  // 构建 ExecutorBarrier 用于协调多个 Executor 并行计算，保证每个Executor执行graph计算时数据的一致性。
  // 在barrier的协调下，每个executor完成对应的graph计算操作。
  ExecutorBarrier* barrier = new ExecutorBarrier(
  num_executors, run_state.rendez, [&run_state](const Status& ret) {
    {
      mutex_lock l(run_state.mu_);
      run_state.status.Update(ret);
    }
    run_state.executors_done.Notify();
  });
```

```
20  // 异步启动执行器
21  for (const auto& item : executors_and_keys->items) {
22    // 通过线程池实际运行Executor
23    thread::ThreadPool* device_thread_pool =
24    item.device->tensorflow_device_thread_pool();
25    if (!device_thread_pool) {
26      args.runner = default_runner;
27    } else {
28      args.runner = [this, device_thread_pool](Executor::Args::Closure c) {
29        device_thread_pool->Schedule(std::move(c));
30      };
31    }
32    if (handler != nullptr) {
33      args.user_intra_op_threadpool = handler->AsIntraThreadPoolInterface();
34    }
35    item.executor->RunAsync(args, barrier->Get());
36  }

38  // 等待执行器结束
39  WaitForNotification(&run_state, &step_cancellation_manager,
40  run_options.timeout_in_ms() > 0 ? run_options.timeout_in_ms() : operation_timeout_in_ms_);
41  ... ...
42 }
```

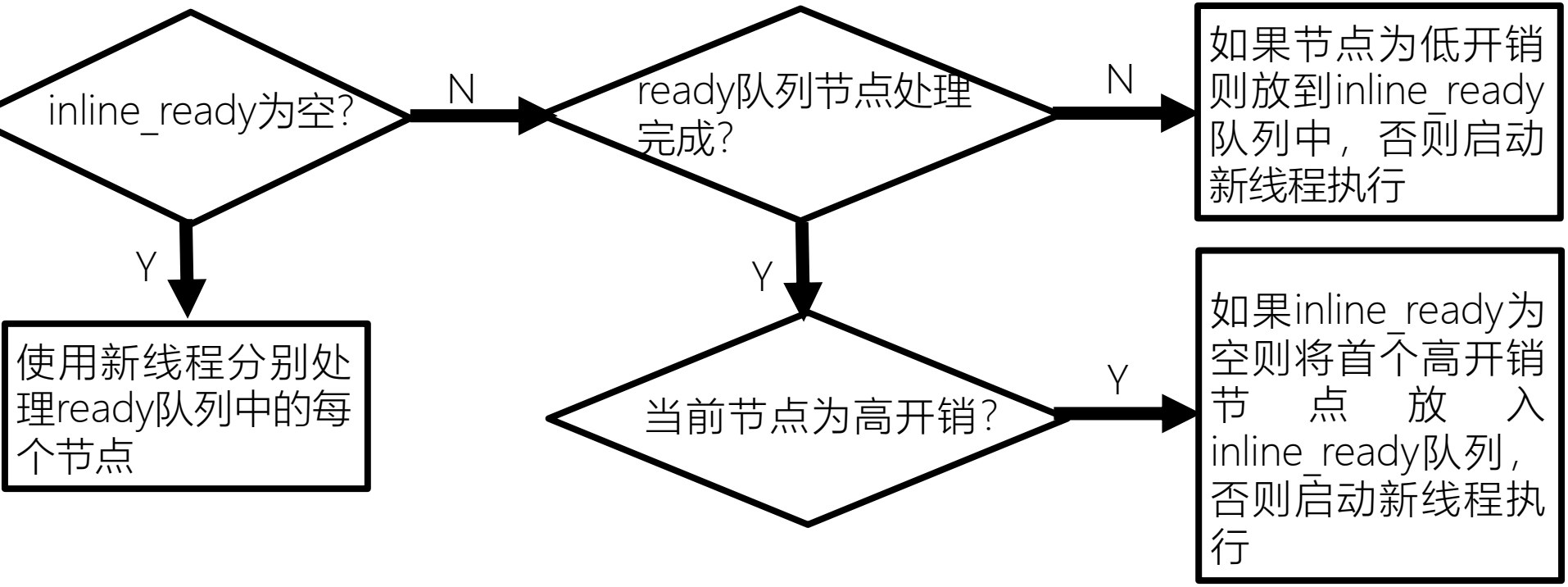
# 执行器逻辑

- 执行流（Stream）：一个能够存储计算任务的队列
- 流间任务可以并行执行，流内任务串行执行



# ScheduleReady逻辑流程

- 输入： ready队列（预执行队列）， inline_ready队列（当前线程要处理的队列）



- ready队列中的每个节点都使用process方法来进行处理，Process函数会真正进行节点计算：
  - 为OpKernel设置运行参数、为OpKernel::Compute准备输入和参数、调用设备计算、处理计算输出、传播输出、更新节点间依赖关系
- 计算核函数有同步和异步两种运行模式。其中绝大部分算子是同步计算模式，而Send/Receive算子则是异步计算模式。
- 类似GPU这种具有执行流（Stream）概念的设备，核函数并不真正同步运行，调用完Compute函数只表示计算任务已经下发到了执行流中

# 3、设备抽象和管理

- TensorFlow将设备分成本地设备和远程设备两类
- TensorFlow使用注册机制来管理设备。每个设备负责一个子图的运算，可以通过注册接口支持自定义设备
- 设备继承自DeviceBase类，其定义了基本的数据结构与接口
- 基于DeviceBase类进一步设计了LocalDevice类
- 本地设备会基于LocalDevice创建自己的设备类，再通过注册机制将设备注册到TensorFlow运行时

```cpp
class BaseDLPDevice : public LocalDevice {
  public:
  BaseDLPDevice(const SessionOptions& options, const string& name,
    Bytes memory_limit, const DeviceLocality& locality, const int device_id,
    const string& physical_device_desc, Allocator* dlp_allocator,
    Allocator* cpu_allocator, bool sync_every_op, int32 max_streams);

  // 是否需要记录访问过的Tensor。
  bool RequiresRecordingAccessedTensors() const override;

  // 为当前设备的计算图分配执行流，尽量地利用硬件资源。
  Status FillContextMap(const Graph* graph, DeviceContextMap* device_context_map) override;

  // 同步：以等到执行流中的计算任务全部结束。
  Status Sync() override;

  // 计算：将计算任务下发到执行流。
  void Compute(OpKernel* op_kernel, OpKernelContext* context) override;

  // 异步计算，直到真正的计算结束执行回调函数。只有Send和Recv是异步核函数。
  void ComputeAsync(AsyncOpKernel* op_kernel, OpKernelContext* context,
        AsyncOpKernel::DoneCallback done) override;
```

```
  // 机器中可能包含多个智能设备,这个函数用于返回当前智能设备ID 。
  int dlp_id() const { return dlp_id_; }

  // 深度学习处理器执行器，用来管理设备、控制深度学习处理器执行流。
  DLPStreamExecutor* executor() const { return executor_; }
  ... ...

  protected:
  // 内存分配器
  Allocator* dlp_allocator_; // not owned
  Allocator* cpu_allocator_;  // not owned

  // 深度学习处理器的执行器
  DLPStreamExecutor* executor_;  //not owned

  private:
  // 执行流数组
  vector<DLPStream*> streams_;
  // 设备环境
  std::vector<DLPDeviceContext*> device_contexts_;
  // 设备信息
  DLPDeviceInfo* dlp_device_info_ = nullptr;
  ... ...
};
```

# 4、网络和通信

- TensorFlow的设备间通信由Send和Receive节点进行，使用Rendezvous机制完成数据交互
- Rendezvous机制对外提供了最基本的Send、Recv和RecvAsync接口和实现，在不同的通信场景下需要提供不同的实现
- 对于本地传输来说，TensorFlow提供了LocalRendezvous实现类，对于使用跨进程通信场景来说，TensorFlow提供了RemoteRendezvous实现系列

- 每个Rendezvous实例拥有一个通道表，其中记录了每对Send/Receive的关系和状态。不同的通道拥有唯一的键值
- 生产者使用Send方法将数据传到特定通道，消费者使用Receive方法从特定通道中获取数据。消费者可以在任意时刻调用Recv方法来获取数据，可以选择使用回调或者阻塞的方法来获取数据。
- 不论哪种方法，消费者都能在数据有效时尽快得到数据。生产者在任何时候都不会被阻塞。

```
class Rendezvous : public core::RefCounted {
  public:
    struct Args {
       DeviceContext* device_context = nullptr;
      AllocatorAttributes alloc_attrs;
      CancellationManager* cancellation_manager = nullptr; // not owned.    };

    // 使用CreateKey函数来构造可以解析的rendezvous键值。
    static string CreateKey(const string& src_device, uint64 src_incarnation,
      const string& dst_device, const string& name,
      const FrameAndIter& frame_iter);

    // 解析rendezvous 键值，得到源和目的设备等信息。
    struct ParsedKey {
      StringPiece src_device;
      DeviceNameUtils::ParsedName src;
      uint64 src_incarnation = 0;
      StringPiece dst_device;
      DeviceNameUtils::ParsedName dst;
      StringPiece edge_name;

      ParsedKey() {}
      ParsedKey(const ParsedKey& b) { *this = b; }
      ... ...    };
      ... ...
}
```

- Rendezvous类中最重要的函数是Send和Recv
  - 发送方法（Send）：将Tensor（val）和状态（is_dead）等信息绑定在一起发送到特定的键值通道上。is_dead通常是由控制流相关算子来设置的变量。参数args是Send传给Recv的一些信息，通常该信息只有当Send和Recv在一个工作进程（worker）中才有效

```
27  //发送函数是纯虚函数
28  virtual Status Send(const ParsedKey& key, const Args& args,
29          const Tensor& val, const bool is_dead) = 0;
30  typedef std::function<void(const Status&, const Args&, const Args&, const Tensor&,
31          const bool)>  DoneCallback;
```

- Rendezvous类中最重要的函数是Send和Recv
  - 接收方法（Recv）：由于Recv不知道何时数据才是有效的，因此采用异步接收模式。一旦读取的Tensor有效时，回调函数便会被调用，完成Recv后续操作。

```
33  //异步接收函数是纯虚函数
34  virtual void RecvAsync(const ParsedKey& key, const Args& args, DoneCallback done) = 0;
35
36  // 使用RecvAsync封装出的同步接收函数
37  Status Recv(const ParsedKey& key, const Args& args, Tensor* val, bool* is_dead, int64 timeout_ms);
38  Status Recv(const ParsedKey& key, const Args& args, Tensor* val, bool* is_dead);
```

# 本地通信：LocalRendezvousImpl

- Send函数：如果队列（即键值通道）为空或者队列中只有Send的信息，则继续把新的信息放入到队列；如果队列中有Recv的信息，则直接把这个Send信息通过Recv的回调函数传给Recv。
- Recv函数：主要逻辑是RecvAsync，如果队列中已经有Send信息，那么直接把该Send信息处理掉；如果队列为空或者只有Recv信息，则继续将本次Recv信息放入到队列。

# 远程通信：RemoteRendezvous

- TensorFlow中使用RPC通信机制实现远程通信，因此设备间使用RpcRemoteRendezvous机制作为远程数据的交互
- RpcRemoteRendezvous和LocalRendezvous在主要逻辑上是一致的，也是使用Send和Recv两个方法进行交互
- RpcRemoteRendezvous类继承于BaseRemoteRendezvous

- 在BaseRemoteRendezvous中定义了RecvAsync的实现

```
1  void BaseRemoteRendezvous::RecvAsync(const ParsedKey& parsed,
2    const Rendezvous::Args& recv_args, DoneCallback done) {
3    ... ...
4    if (IsSameWorker(parsed.src, parsed.dst)) {
5    //如果源和目的是同一个worker，则调用本地的RecvAsync
6      local_->RecvAsync(... ...);
7    }
8    else {
9    //否则调用RpcRemoteRensezvous的RecvFromRemoteAsync方法。
10   RecvFromRemoteAsync(parsed, recv_args, std::move(done));}
11   ... ...
12 }
```

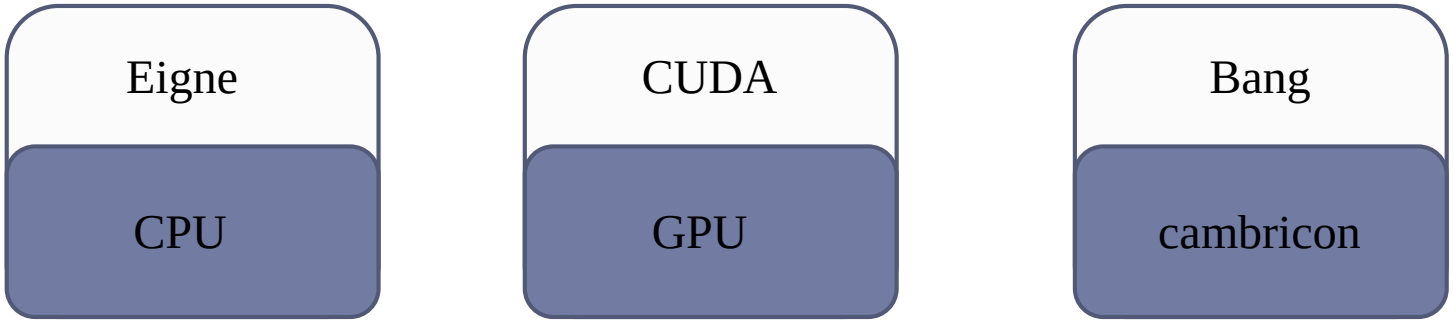
# RecvFromRemoteAsync

- 核心是准备并启动RpcRecvTensorCall类型的过程调用，用于获取远程Tensor。其中最终会调用Worker的RecvTensorAsync发起相应请求

```
void RpcRemoteRendezvous::RecvFromRemoteAsync(const Rendezvous::ParsedKey& parsed,
  const Rendezvous::Args& recv_args, DoneCallback done) {
  CHECK(is_initialized());
  Status s;
  // 准备并启动RpcRecvTensorCall类型的过程调用，用于获取远程Tensor。
  // RpcRecvTensorCall最终会调用Worker的RecvTensorAsync发起相应请求
  RpcRecvTensorCall* call = get_call_freelist()->New();
  ... ...
  WorkerSession* sess = session();
  WorkerInterface* rwi = sess->worker_cache()->GetOrCreateWorker(call->src_worker_);
  Device* dst_device;
  if (s.ok()) {
    s = sess->device_mgr()->LookupDevice(parsed.dst_device, &dst_device);
  }
  ... ...
  // 初始化过程调用
  call->Init(rwi, step_id_, parsed.FullKey(), recv_args.alloc_attrs, dst_device,
    recv_args, std::move(done));

  // 开始过程调用
  Ref();
  call->Start([this, call]() {
    call->ReleaseWorker(session()->worker_cache());
    call->done()(s, Args(), call->recv_args(), call->tensor(), call->is_dead());
    get_call_freelist()->Release(call);
    Unref();
  });
}
```

# 5、算子实现

- 算子是TensorFlow的基本单元，OpKernel是算子的特定执行，依赖于底层硬件
- TensorFlow通过注册机制来支持不同的算子和相应的OpKernel函数



- OpKernel的计算可以是同步的也可以是异步的
- 大部分OpKernel的计算是同步的，"Compute()"返回即认为数据已经被正确处理
  - 继承OpKernel，复写Compute()方法
- 和通信相关的OpKernel（如Send和Receive）需要采用异步执行方式
  - 继承AsyncOpKernel，复写ComputeAsync()方法
- 所有的OpKernel在实现Compute或ComputeAsync方法时，都是通过OpKernelContext得到输入输出信息，并设置运行状态

# 示例

- 1、基于OpKernel定义算子

```
class DLPMaxPoolOp : public OpKernel {
  public:
    explicit DLPMaxPoolOp(OpKernelConstruction* context) : OpKernel(context) {
    // 根据context信息进行初始化参数以及参数检查。
      ... ...
    }
    void Compute(OpKernelContext* context) override {
    // 使用DLP编程语言实现的MaxPool运算逻辑。
      ... ...
    }
    ... ...
};
```

- 2、将算子注册到TensorFlow系统中

```
REGISTER_KERNEL_BUILDER(Name("MaxPool")          // Op名字
                        .Device(DEVICE_DLP)                    // 设备类型
                        .TypeConstraint<T>("T"),               // 数据类型
                        DLPMaxPoolOp<T>);                    // OpKernel对象
```

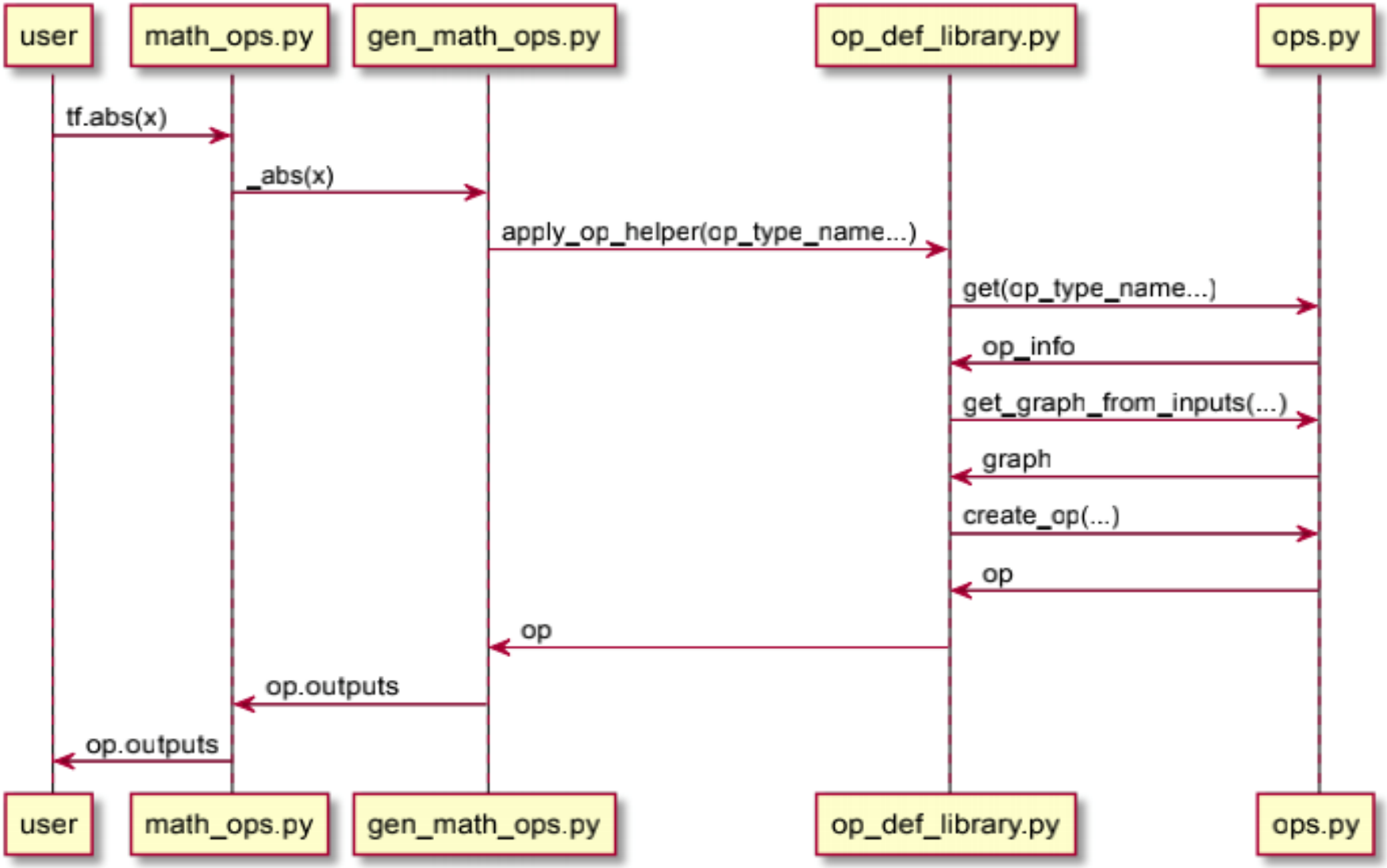
# 提纲

- TensorFlow设计原则
- TensorFlow计算图机制
- TensorFlow系统实现
- 驱动范例
- 编程框架对比

# 构建VGG网络

```
def build_vggnet(vgg19_npy_path):
  models = {}
  models['input'] = tf.Variable(np.zeros((1, 224, 224, 3)).astype('float32'))
  models['conv1_1'] = basic_calc('conv',models['input'],read_wb(vgg19_npy_path,'conv1_1'))
  models['conv1_2'] = basic_calc('conv',models['conv1_1'],read_wb(vgg19_npy_path,'conv1_2'))
  models['pool1']   = basic_calc('pool',models['conv1_2'])
  models['conv2_1'] = basic_calc('conv',models['pool1'],read_wb(vgg19_npy_path,'conv2_1'))
  models['conv2_2'] = basic_calc('conv',models['conv2_1'],read_wb(vgg19_npy_path,'conv2_2'))
  models['pool2']   = basic_calc('pool',models['conv2_2'])
  models['conv3_1'] = basic_calc('conv',models['pool2'],read_wb(vgg19_npy_path,'conv3_1'))
  models['conv3_2'] = basic_calc('conv',models['conv3_1'],read_wb(vgg19_npy_path,'conv3_2'))
  models['conv3_3'] = basic_calc('conv',models['conv3_2'],read_wb(vgg19_npy_path,'conv3_3'))
  models['conv3_4'] = basic_calc('conv',models['conv3_3'],read_wb(vgg19_npy_path,'conv3_4'))
  models['pool3']   = basic_calc('pool',models['conv3_4'])
  models['conv4_1'] = basic_calc('conv',models['pool3'],read_wb(vgg19_npy_path,'conv4_1'))
  models['conv4_2'] = basic_calc('conv',models['conv4_1'],read_wb(vgg19_npy_path,'conv4_2'))
  models['conv4_3'] = basic_calc('conv',models['conv4_2'],read_wb(vgg19_npy_path,'conv4_3'))
  models['conv4_4'] = basic_calc('conv',models['conv4_3'],read_wb(vgg19_npy_path,'conv4_4'))
  models['pool4']   = basic_calc('pool',models['conv4_4'])
  models['conv5_1'] = basic_calc('conv',models['pool4'],read_wb(vgg19_npy_path,'conv5_1'))
  models['conv5_2'] = basic_calc('conv',models['conv5_1'],read_wb(vgg19_npy_path,'conv5_2'))
  models['conv5_3'] = basic_calc('conv',models['conv5_2'],read_wb(vgg19_npy_path,'conv5_3'))
  models['conv5_4'] = basic_calc('conv',models['conv5_3'],read_wb(vgg19_npy_path,'conv5_4'))
  models['pool5']   = basic_calc('pool',models['conv5_4'])
  return models
```

# 内部构图逻辑


user
math_ops.py
gen_math_ops.py
op_def_library.py
ops.py
tf.abs(x)
_abs(x)
apply_op_helper(op_type_name...)
get(op_type_name...)
op_info
get_graph_from_inputs(...)
graph
create_op(...)
op
op
op.outputs
op.outputs
user
math_ops.py
gen_math_ops.py
op_def_library.py
ops.py

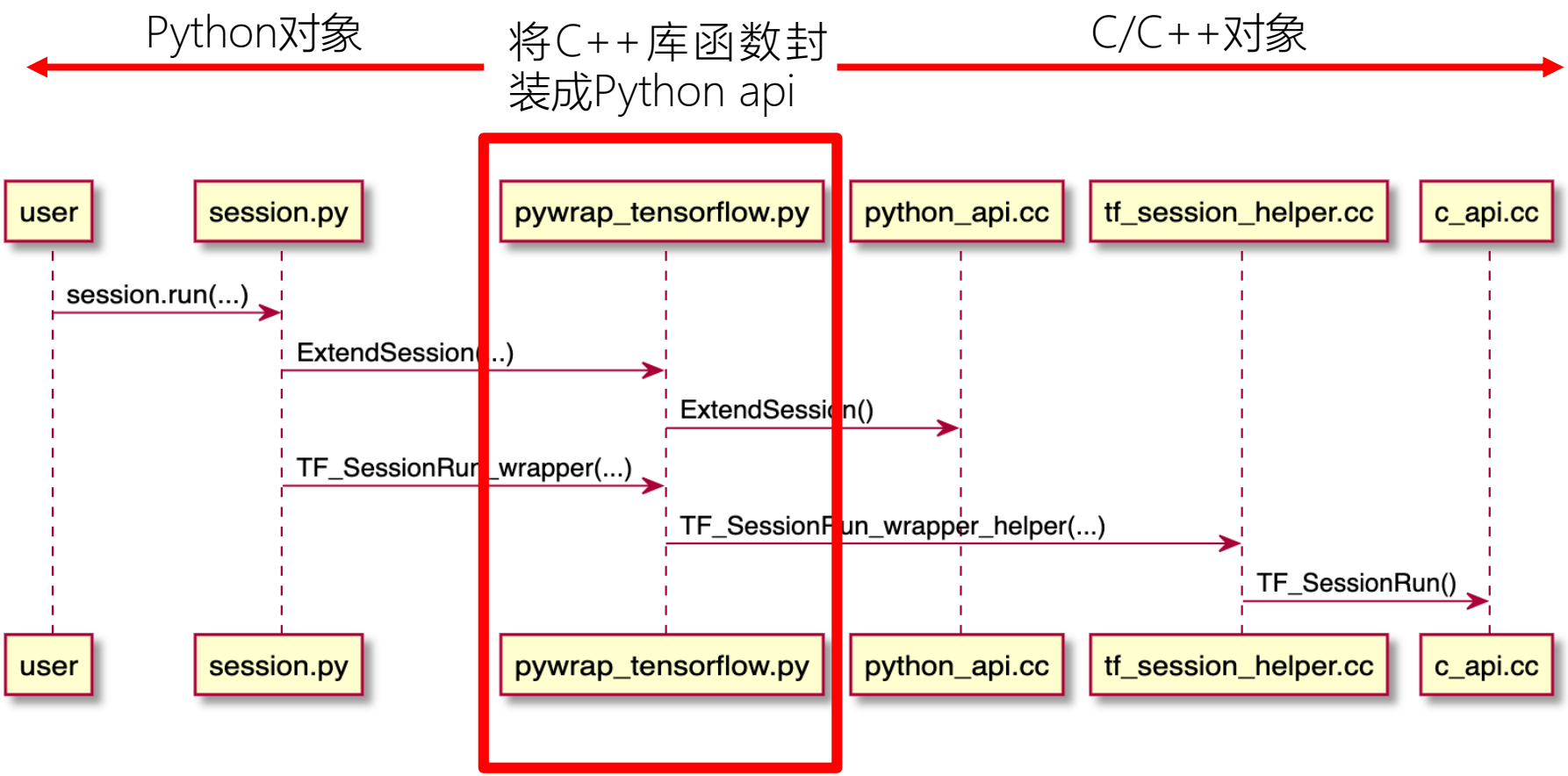
# 训练模型

```
...
#构建模型，使用与模型推理时相同的网络结构
models = build_vggnet(vgg19_npy_path)

#定义损失函数
total_loss = loss(sess, models,img_content,img_style)
#创建优化器
optimizer = tf.train.AdamOptimizer(2.0)
#定义模型训练方法
train_op = optimizer.minimize(total_loss)
#使用噪声图像img_random进行训练
sess.run(models['input'].assign(img_random))
#保存模型
saver = tf.train.Saver(manx_to_keep = 5)
savedir = "model/"

while step < epochs:
  step += 1
  -, loss = sess.run([train_op, total_loss])
  #恢复模型
  saver.save(sess, savedir + "vgg19.ckpt", global_step = epoch)
```

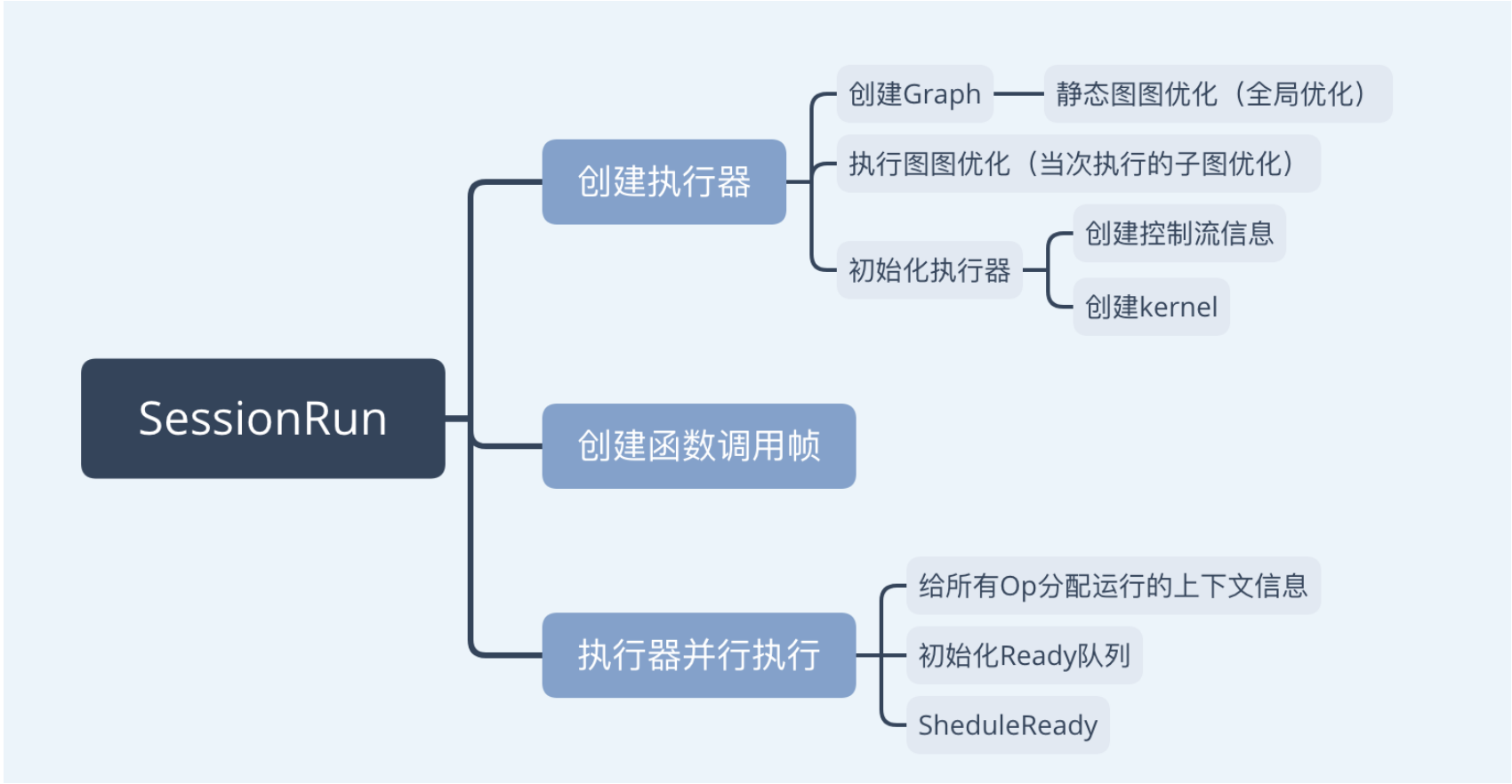
# 加载模型执行预测

```
with tf.Session() as sess:
    models = build_vggnet(vgg19_npy_path)

    sess.run(models['input'].assign(img_content))
    res = sess.run(models['pool5'])
```

# session run由Python到C/C++ API

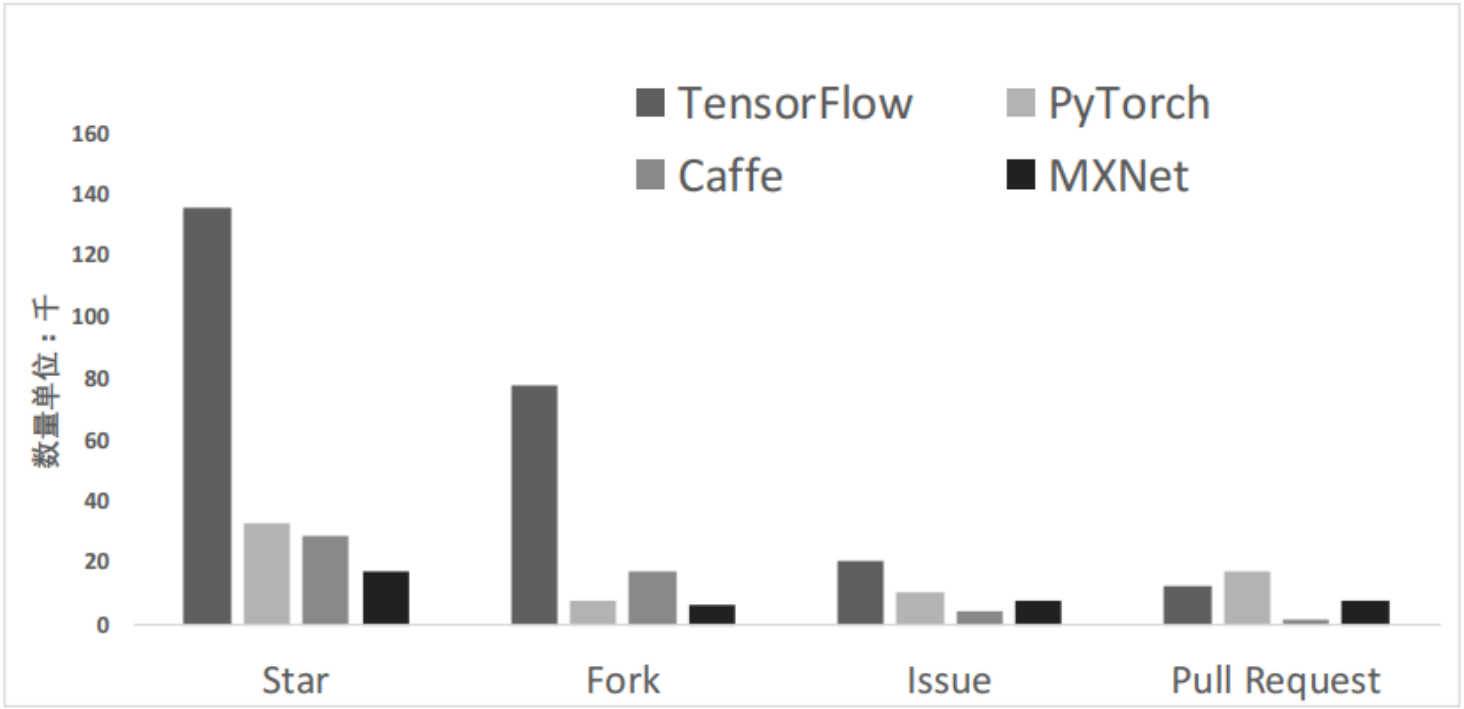


# C++中的SessionRun流程



# 提纲

- TensorFlow设计原则
- TensorFlow计算图机制
- TensorFlow系统实现
- 驱动范例
- 编程框架对比




- 统计各框架星标（Star）数、仓库复制数（Fork）、讨论贴（Issue）数和代码提交请求（Pull Request）数

| 框架名称 | 主要维护团体 | 前端支持语言 | 支持平台 | 编程模式 | 辅助工具生态 |
| --- | --- | --- | --- | --- | --- |
| **TensorFlow** | Google | Python、C/C++、Java、Go、JavaScript、R、Julia、Swift | Linux、MacOS、Windows、iOS、Android | Graph：声明式；Eager：命令式 | TensorBoard、Profiler、TFLite、TF-serving、tfdbg、官方模型库 |
| **PyTorch** | Facebook | Python、C++ | Linux、MacOS、Windows | 命令式 | TorchVision、官方模型库、ONNX 模型交换格式 |
| **MXNet** | Amazon | Python、C++、Go、Julia、Matlab、R、JavaScript、Scala、Perl、Clojure | Linux、MacOS、Windows、iOS、Android | MXNet：声明式、Gluon：命令式 | mxboard、官方模型库 |
| **Caffe** | 官方不再维护，推出 Caffe2 | Python、C++、Matlab | Linux、MacOS、Windows | 声明式 | 官方模型库 |

# TensorFlow

- 目前社区最受欢迎的框架之一
- 支持众多常见的前端语言，覆盖云端到终端几乎所有的平台，同时也有众多的辅助工具来支持多平台多设备使用
- 社区力量强大，文档完善，对初学者较为友好。提供了丰富的教程和开源模型库帮助用户更好地学习和使用
- API较为混乱、声明式编程不方便调试
- TensorFlow 2.0版本中提供命令式和声明式两种编程模式

# PyTorch

- 小而灵活
- 前端支持Python和C++
- 支持动态图命令式的编程模式，在复杂循环网络中更易用及易调试
- 在小规模的使用场景和学术界，Pytorch使用者数量迅猛增长，有赶超TensorFlow的趋势
- 目前PyTorch无法全面支持各种平台，训练好的模型不能很方便地转移到其他平台或设备上使用，因此对生产环境来说，PyTorch目前还不是首选

# MXNet

- 针对效率和灵活性而设计
- 支持声明式编程及命令式编程
- 支持R、Julia和Go等语言

# Caffe

- 计算以层（Layer）为粒度，对应于神经网络中的层，为每一层给出了前向实现和反向实现
- 使用者能很快掌握深度学习基础算法的内部本质和实现方法，并由此开发出自己的Caffe变种，完成自定义功能
- 缺少灵活性、扩展性和复用性。在功能上有很多局限性，对RNN类的网络支持有限
- 早期的Caffe版本已经不再维护更新

# 小结

- TensorFlow设计原则

  高性能、易开发、可移植

- TensorFlow计算图机制

  自动求导、计算图本地执行、分布式执行

- TensorFlow系统实现

  整体架构、执行模块、设备管理、网络和通信、算子实现

- 驱动范例

  以VGG19为例介绍TensorFlow内部的实现机理

- 编程框架对比

  TensorFlow、PyTorch、Caffe、MXNet框架的对比

# 谢谢大家！