Tensorflow 分布式训练

1，PS-worker 架构

将模型维护和训练计算解耦合，将模型训练分为两个作业（job）：

- 模型相关作业，模型参数存储、分发、汇总、更新，有由 PS 执行
- 训练相关作业，包含推理计算、梯度计算（正向/反向传播），由 worker 执行

该架构下，所有的 woker 共享 PS 上的参数，并按照相同的数据流图传播不同 batch 的数据，计算出不同的梯度，交由 PS 汇总、更新新的模型参数，大体逻辑如下：

1. pull：各个 woker 根据数据流图拓扑结构从 PS 获取最新的模型参数
2. feed：各个 worker 根据定义的规则填充各自 batch 的数据
3. compute：各个 worker 使用第一步的模型参数计算各自的 batch 数据，求出各自 batch 的梯度
4. push：各个 worker 将各自的梯度推送到 PS
5. update：PS 汇总来自 n 个 worker 的 n 份梯度，求出平均值后更新模型参数

分布式经典架构 PS-worker 会重复上面步骤，直到损失到达阈值或者轮数到达阈值。

2，数据并行模式分类

根据数据流图构建模式分类：

- 图内复制：单进程、‘单机多卡’的数据并行训练，需要用户自己实现梯度汇总和均值计算。实例，models/tutorials/image/cifar10/cifer10_multi_gpu-train.py（见下节）
- 图间复制：多进程、跨多机的分布式训练，使用同步优化器（SyncReplicasOptimizer）实现分布式梯度计算和模型参数更新。实例，tensorflow/tools/dist_test/python/mnist_replica.py（分布式同步训练实践，见下下节）

根据参数更新机制分类：

- 异步训练：各个 worker 独立训练，计算出梯度后即刻更新参数，不需要等待其他 worker 完成计算
- 同步训练：所有 worker 完成本轮计算后，汇总梯度，更新模型，计算能力强的 worker 需要阻塞等待其他 worker

两种训练机制同时支持上面两周数据流图构建模式。一般来说同步机制收敛快，异步单步计算快，但易受单批数据影响，不稳定。

3，同步优化器

tensorflow 进行**同步**（同步训练模式专用）各个 worker 梯度并进行优化时，会使用特殊的优化器即同步优化器，tf.train.SyncReplicasOptimizer，其**第一个参数为普通优化器**，我们可以定义一个普通的优化器传入，后续参数如下：

参数名称	功能说明	默认值
replicas_to_aggragate	并行副本数	num_workers
total_num_replicas	实际副本数（worker 数目）	num_workers

并行副本数指期望的每一步中并行的 batch 数据数目，实际副本数指参与的 workers 数目，

- 并行=实际：全民参与，一个 worker 领取一个 batch 数据
- 并行>实际：能者多劳，先完成自己 batch 的 worker 会继续领取未训练数据，PS 会等到梯度份数到达并行数后进行模型参数计算
- 并行<实际：替补等位，存在空闲的 worker，取代可能出现的异常 worker，确保训练过程高可用

运算过程

- 计算梯度过程同普通优化器，调用基类的 Optimizer 的 compute_gradients 成员方法
- 更新参数时重写了 Optimizer 的 apply_gradients 方法，见 tensorflow/python/training/sync_replicas_optimizer.py

讲解同步优化器工作逻辑之前，介绍两个概念，

梯度聚合器

每一个模型参数有一个自己队列，收集来自不同 worker 的梯度值，梯度聚合器包含 M 个队列对应 M 个模型参数，每个队列收集来自 N 个 worker 计算出来的 N 个梯度值。

同步标记队列

存储同步标记，实际上就是 N 个 global_step 值，每个 worker 领取一个，用于控制同步

以全民参与模式为例

worker 工作模式如下：

1. 从同步标记队列领取一个 global_step，表示全局训练步数的同步标记
2. 将同步标记值赋予 worker 的本地训练步数 local_step

3. 从 PS 获取最新模型参数
4. 计算出 M 个梯度值
5. 将 M 个梯度值推送到 PS 上的 M 个梯度队列中

PS 工作模式如下：

1. 从梯度聚合器上收集 worker 推送过来的梯度值，每个队列收集 N 份（对应 N 个 global_step 下训练值）后，计算均值，收集齐 M 个均值后，得到 M 对{模型参数，梯度值}的聚合元组
2. 更新模型参数
3. 向同步标记队列推送 N 个 global_step+1 标记

聚合器收集梯度值并校验 local_step 是否符合 global_step，是则接收梯度值，计算能力强的 worker 提交梯度后由于没有同步标记可以领取所以被阻塞，PS 集齐 N 份后更新参数，发布下次 N 个同步标记，开始下一步训练。

由于初始 PS 不会更新参数发布同步标记，所以需要初始化同步标记队列——sync_init_op，直接向队列注入 N 个 0 标记。

分布式模型训练需要的主要初始化操作如下（opt.tf.train.SyncReplicasOptimizer）：

操作名称	常用变量名	功能说明
opt.local_step_init_op	local_init_op	loacl_step 初始值
pot.chief_init_op	local_init_op	gobal_step 初始值
opt.ready_for_local_init_op	ready_for_local_init_op	为未初始化的 Variable 设置初始值
opt.get_chief_queue_runner	chief_queue_runner	同步标记队列启动 QueueRunner 实例
opt.get_init_tockens_op	sync_init_op	同步标记队列初始化
tf.global_variables_initializer	init_op	全局 Variable 设置初始值

如果使用模型管理类 Supervsor，可以将大部分工作交由其代劳。

以能者多劳模式对比

模型参数个数 M，worker 个数 N，并行副本数 R（R>N），此时

梯度聚合器仍然有 M 个参数收集队列，每一个队列要收集 R 份才进行汇总，R>N 所以会存在某个 worker 领取多份数据的情况。

同步标记队列存储 R 个同步标记，以确保每一步中梯度聚合器可以收集到 R 份数据。

4，异步优化器

异步优化器没有很多附加参量，和单机训练几乎一致，只是每个 worker 获取参数需要从另一个进程 PS 中得到而已。

5，模型管理类 Supervsor

本质上是对 Saver（模型参数存储恢复）、Coordinator（多线程服务生命周期管理）、SessionManager（单机以及分布式会话管理）三个类的封装，Coordinator 会监测程序的线程是否运行正常，任何异常的出现都会向 Supervisor 报告，此时 Coordinator 讲程序的停止条件设置为 True，Supervisor 停止训练并清理工作（关闭会话、回收内存等），其他服务检测到 True 后会各自关闭服务，终止线程。

SessionManager 帮助用户创建管理单机或是分布式会话，以便简化数据流图的生命周期和维护逻辑，同事负责将 checkpoint 文件中加载出的数据恢复到数据流图中。

流程逻辑如下：

1. 创建 Supervisor 实例，构造方法需要传入 checkpoint 文件和 summary 文件存储目录（Supervisor 的 logdir 参数）
2. 调用 tf.train.Supervisor.managed_session，从 Supervisor 实例获取会话实例
3. 使用该会话执行训练，训练中需要检查停止条件，保证训练正确性

获取 managed_session 时，Supervisor 会通过 QueueRunner 同时启动一下三个服务：

- 检查点服务：将数据流图中的参数定期保存，默认 10min 保存一次，且会识别 global_step（Supervisor 的 global_step 参数）
- 汇总服务：默认 2min 一次
- 步数计数器服务：向汇总添加 global_step/sec，2min 一次

使用 managed_session 创建会话时，会自动恢复上一次的结果并继续训练

一、tensorflow GPU 设置

GPU 指定占用

```
gpu_options = tf.GPUOptions(per_process_gpu_memory_fraction=0.7)
sess = tf.Session(config=tf.ConfigProto(gpu_options=gpu_options))
```

上面分配给 tensorflow 的 GPU 显存大小为：GPU 实际显存*0.7。

GPU 模式禁用

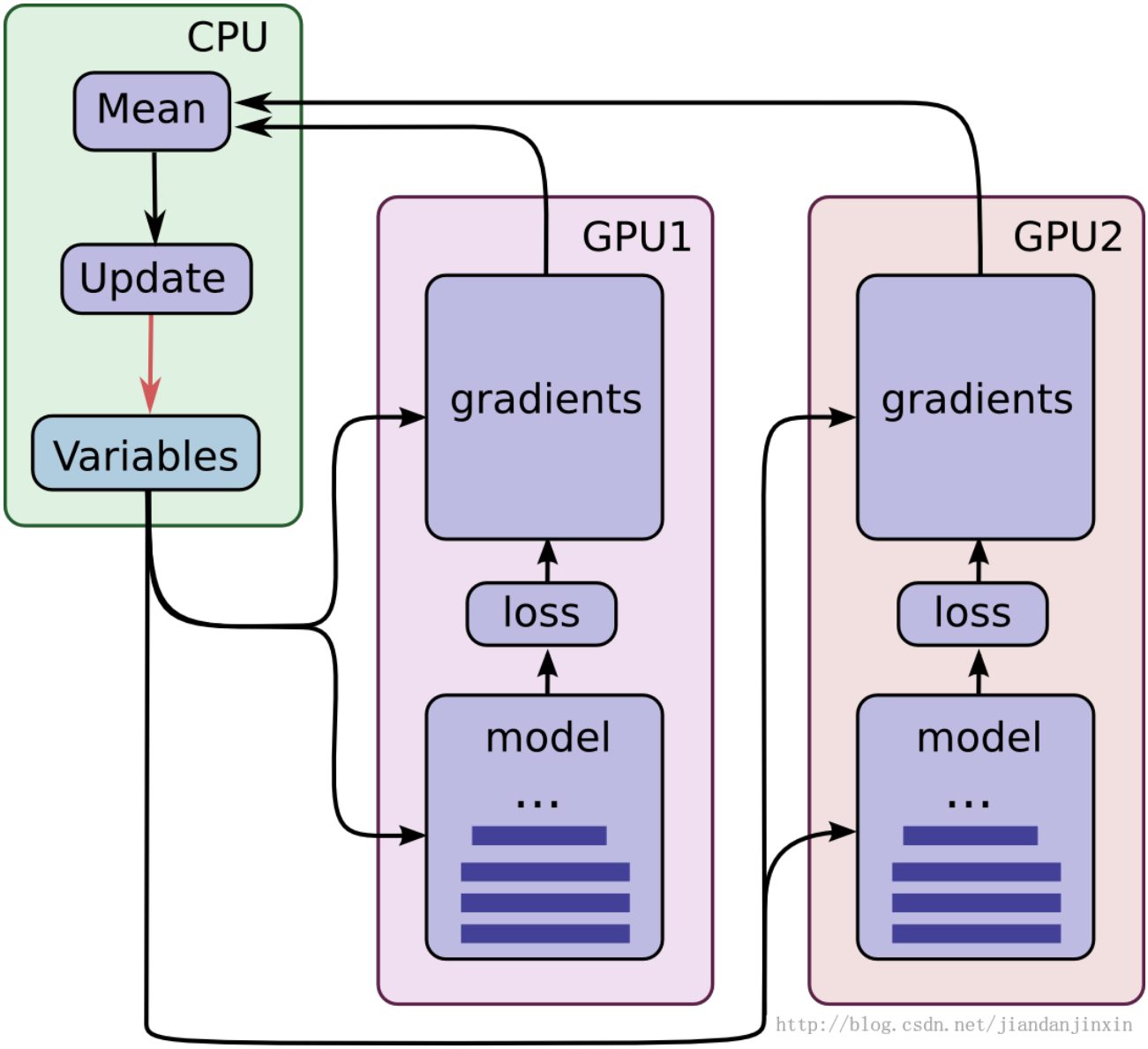
```
import os
os.environ["CUDA_VISIBLE_DEVICES"]="-1"
```

GPU 资源申请规则

```
# 设置 GPU 按需增长
config = tf.ConfigProto()
config.gpu_options.allow_growth = True
sess = tf.Session(config=config)
```

二、单机多 GPU 工作原理

以一篇 csdn 博客（出处见水印）上的图说明多 GPU 工作原理：

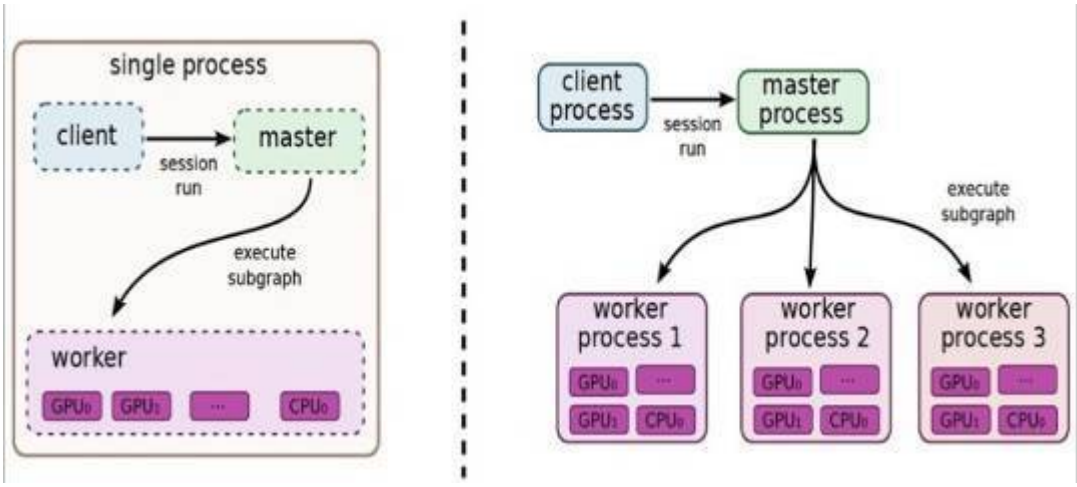


想让 TensorFlow 在多个 GPU 上运行, 需要建立 multi-tower 结构, 在这个结构里每个 tower 分别被指配给不同的 GPU 运行，汇总工作一般交由 CPU 完成，示意如下，

```
# 新建一个 graph.
c = []
for d in ['/gpu:2', '/gpu:3']:
  with tf.device(d):
    a = tf.constant([1.0, 2.0, 3.0, 4.0, 5.0, 6.0], shape=[2, 3])
    b = tf.constant([1.0, 2.0, 3.0, 4.0, 5.0, 6.0], shape=[3, 2])
    c.append(tf.matmul(a, b))
with tf.device('/cpu:0'):
  sum = tf.add_n(c)
# 新建 session with log_device_placement 并设置为 True.
sess = tf.Session(config=tf.ConfigProto(log_device_placement=True))
# 运行这个 op.
print sess.run(sum)
```

分布式原理

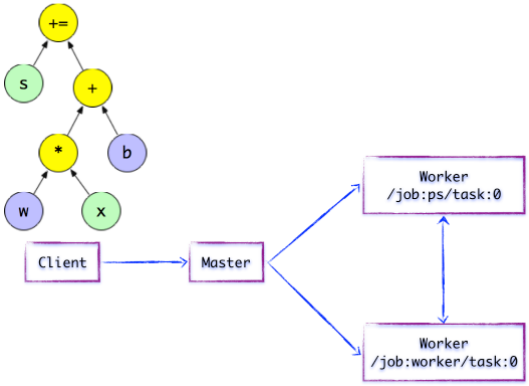
TensorFlow 有一个重要组件 client，顾名思义，就是客户端，它通过 Session 的接口与 master 及多个 worker 相连。其中每一个 worker 可以与多个硬件设备（device）相连，比如 CPU 或 GPU，并负责管理这些硬件。而 master 则负责指导所有 worker 按流程执行计算图。TensorFlow 有单机模式和分布式模式两种实现，其中单机指 client、master、worker 全部在一台机器上的同一个进程中；分布式的版本允许 client、master、worker 在不同机器的不同进程中，同时由集群调度系统统一管理各项任务。



TensorFlow 计算图的运行机制

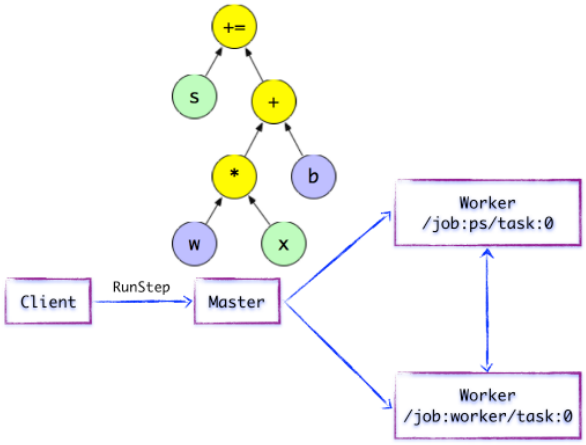
Client 基于 TensorFlow 的编程接口，构造计算图。此时，TensorFlow 并未执行任何计算。直至建立 Session 会话，并以 Session 为桥梁，建立 Client 与后端运行时的通道，将 Protobuf 格式的 GraphDef 发送至 Distributed Master。也就是说，当 Client 对 OP 结果进行求值时，将触发 Distributed Master 的计算图的执行过程。如下图所示，Client 构建了一个简单计算图。它首先将 w 与 x 进行矩阵相乘，再与截距 b 按位相加，最后更新至 s。

Client



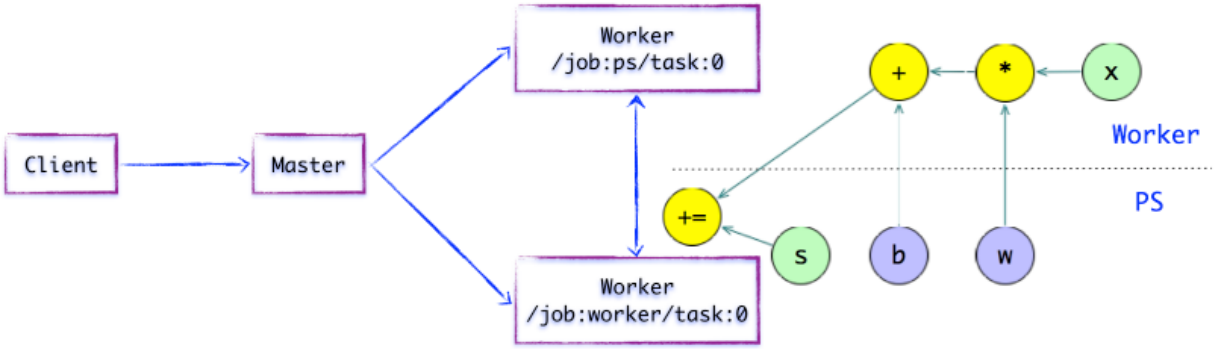
Distributed Master

在分布式的运行时环境中，Distributed Master 根据 Session.run 的 Fetching 参数，从计算图中反向遍历，找到所依赖的最小子图。然后 Distributed Master 负责将该子图再次分裂为多个「子图片段」，以便在不同的进程和设备上运行这些「子图片段」。最后，Distributed Master 将这些图片段派发给 Work Service。随后 Work Service 启动「本地子图」的执行过程。Distributed Master 将会缓存「子图片段」，以便后续执行过程重复使用这些「子图片段」，避免重复计算。



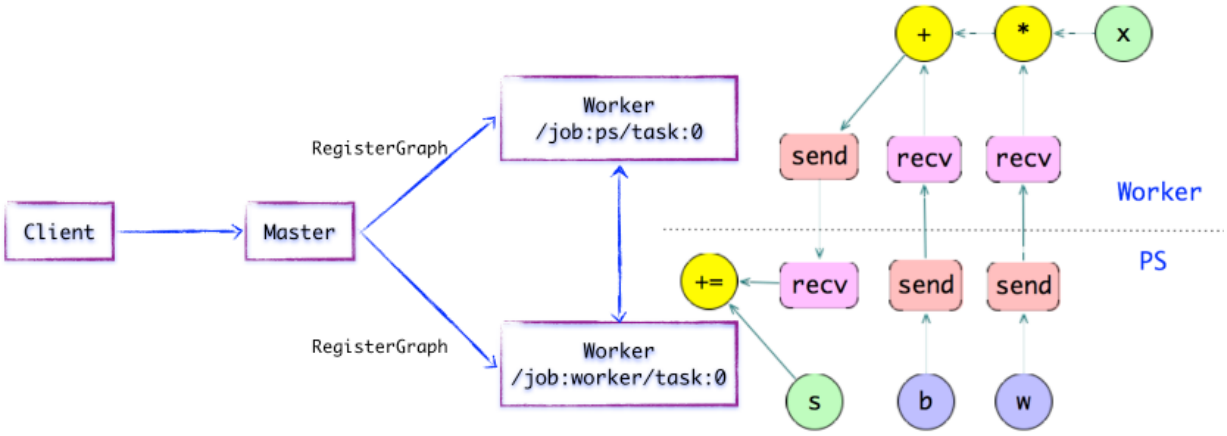
执行图计算

Distributed Master 开始执行计算子图。在执行之前，Distributed Master 会实施一系列优化技术，例如「公共表达式消除」，「常量折叠」等。随后，Distributed Master 负责任务集的协同，执行优化后的计算子图。



子图片段

如上图所示，存在一种合理的「子图片段」划分算法。Distributed Master 将模型参数相关的 OP 进行分组，并放置在 PS 任务上。其他 OP 则划分为另外一组，放置在 Worker 任务上执行。



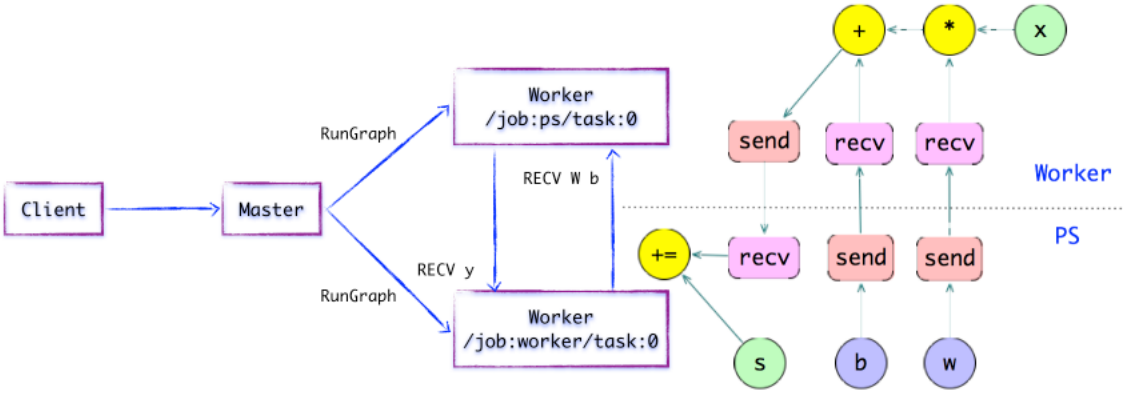
如上图所示，如果计算图的边被任务节点分割，Distributed Master 将负责将该边进行分裂，在两个分布式任务之间插入 SEND 和 RECV 节点，实现数据的传递。

随后，Distributed Master 将「子图片段」派发给相应的任务中执行，在 Worker Service 成为「本地子图」，它负责执行该子图的上的 OP。

Worker Service

对于每个任务，都将存在相应的 Worker Service，它主要负责如下 3 个方面的职责：

- 处理来自 Master 的请求；
- 调度 OP 的 Kernel 实现，执行本地子图；
- 协同任务之间的数据通信。



执行本地子图
Worker Service 派发 OP 到本地设备，执行 Kernel 的特定。它将尽最大可能地利用多 CPU/GPU 的处理能力，并发地执行 Kernel 实现。

另外，TensorFlow 根据设备类型，对于设备间的 SEND/RECV 节点进行特化实现：

- 使用 cudaMemcpyAsync 的 API 实现本地 CPU 与 GPU 设备的数据传输；
- 对于本地的 GPU 之间则使用端到端的 DMA，避免了跨 host CPU 昂贵的拷贝过程。

对于任务之间的数据传递，TensorFlow 支持多协议，主要包括：

- gRPC over TCP
- RDMA over Converged Ethernet

问题 1. master 在哪里？
问题 2. 每一个节点的任务怎么分配？
对于问题 1，在 **In-graph replication** 方式中 **master** 其实就是 `with tf.Session("grpc://172.17.0.2:2222") as sess:` 这句话中指定的 target，也就是说 172.17.0.2:2222 这台机器就是 master，在运行的时候会在这台机器上打印如下日志：

```
I tensorflow/core/distributed_runtime/master_session.cc:1012] Start master session 254ffd62801d1bee with config:
```

对于问题 2，把计算已经从单机多 GPU，已经扩展到了多机多 GPU 了， 不过数据分发还是在一个节点。 这些计算节点暴露出来的网络接口，使用起来就跟本机的一个 GPU 的使用一样， 只要在操作的时候指定 tf.device("/job:worker/task:n")， 就可以向指定 GPU 一样，把操作指定到一个计算节点上计算，使用起来和多 GPU 的类似。

客户端

建立 TensorFlow 计算图，建立与集群交互会话层。也就是包含 session 的代码，一个客户端可同时与多个服务端相连，一个服务端也可与多个客户端相连。

服务端

运行 tf.train.Server 实例进程，TensroFlow 执行任务集群(cluster)的节点。有主节点服务(Master service)和工作节点服务(Worker service)。运行中，一个主节点进程和数个工作节点进程，主节点进程和工作接点进程通过接口通信。单机多卡和分布式结构相同，只需要更改通信接口实现切换。

主节点服务

实现 tensorflow::Session 接口。通过 RPC 服务程序连接工作节点，与工作节点服务进程工作任务通信。TensorFlow 服务端，task_index 为 0 作业(job)。

工作节点服务

实现 worker_service.proto 接口，本地设备计算部分图。TensorFlow 服务端，所有工作节点包含工作节点服务逻辑。每个工作节点负责管理一个或多个设备。工作节点可以是本地不同端口不同进程，或多台服务多个进程。运行 TensorFlow 分布式执行任务集，一个或多个作业(job)。每个作业，一个或多个相同目的任务(task)。每个任务，一个工作进程执行。作业是任务集合，集群是作业集合。

跨多个参数服务器的分片变量

在分布式设置上训练神经网络时，常见模式是将模型参数存储在一组参数服务器上（即`"ps"`作业中的任务），而其他任务则集中在计算上（即，`"worker"`工作中的任务）。 对于具有数百万参数的大型模型，在多个参数服务器上分割这些参数非常有用，可以降低饱和单个参数服务器网卡的风险。 如果您要将每个变量手动固定到不同的参数服务器，那将非常繁琐。 幸运的是，TensorFlow 提供了`replica_device_setter()`函数，它以循环方式在所有`"ps"`任务中分配变量。 例如，以下代码将五个变量引入两个参数服务器：

```
with tf.device(tf.train.replica_device_setter(ps_tasks=2):
    v1 = tf.Variable(1.0)  # pinned to /job:ps/task:0
    v2 = tf.Variable(2.0)  # pinned to /job:ps/task:1
    v3 = tf.Variable(3.0)  # pinned to /job:ps/task:0
    v4 = tf.Variable(4.0)  # pinned to /job:ps/task:1
    v5 = tf.Variable(5.0)  # pinned to /job:ps/task:0
```

您不必传递 `ps_tasks` 的数量，您可以传递集群 `spec = cluster_spec`，TensorFlow 将简单计算`"ps"`作业中的任务数。

如果您在块中创建其他操作，则不仅仅是变量，TensorFlow 会自动将它们连接到`"/job:worker"`，默认为第一个由`"worker"`作业中第一个任务管理的设备。 您可以通过设置 `worker_device` 参数将它们固定到其他设备，但更好的方法是使用嵌入式设备块。 内部设备块可以覆盖在外部块中定义的作业，任务或设备。 例如：

```
      with tf.device(tf.train.replica_device_setter(ps_tasks=2)):
        v1 = tf.Variable(1.0)  # pinned to /job:ps/task:0 (+ defaults to
/cpu:0)
        v2 = tf.Variable(2.0)  # pinned to /job:ps/task:1 (+ defaults to
/cpu:0)
```

```
        v3 = tf.Variable(3.0)  # pinned to /job:ps/task:0 (+ defaults to
/cpu:0)
        [...]
        s = v1 + v2            # pinned to /job:worker (+ defaults to
task:0/gpu:0)
        with tf.device("/gpu:1"):
            p1 = 2 * s         # pinned to /job:worker/gpu:1 (+ defaults to
/task:0)
            with tf.device("/task:1"):
                p2 = 3 * s     # pinned to /job:worker/task:1/gpu:1
```

这个例子假设参数服务器是纯 CPU 的，这通常是这种情况，因为它们只需要存储和传送参数，而不是执行密集计算。

Rethink session

Session 就是用来执行构造好的计算图，就是 tensorflow 的执行引擎
Client 通过 session 接口与 master 和 worker 相连
Master 是负责管理所有 worker 的计算图执行的，也就是创建会话的
Worker 由一个或多个计算设备的 device 组成的，如 cpu，gpu