TensorFlow 遇上 Kubernetes

中兴通讯人工智能计算平台的技术实践

刘光聪[1] 韩炳涛[2]

[1]liu.guangcong@zte.com.cn
[2]han.bingtao@zte.com.cn

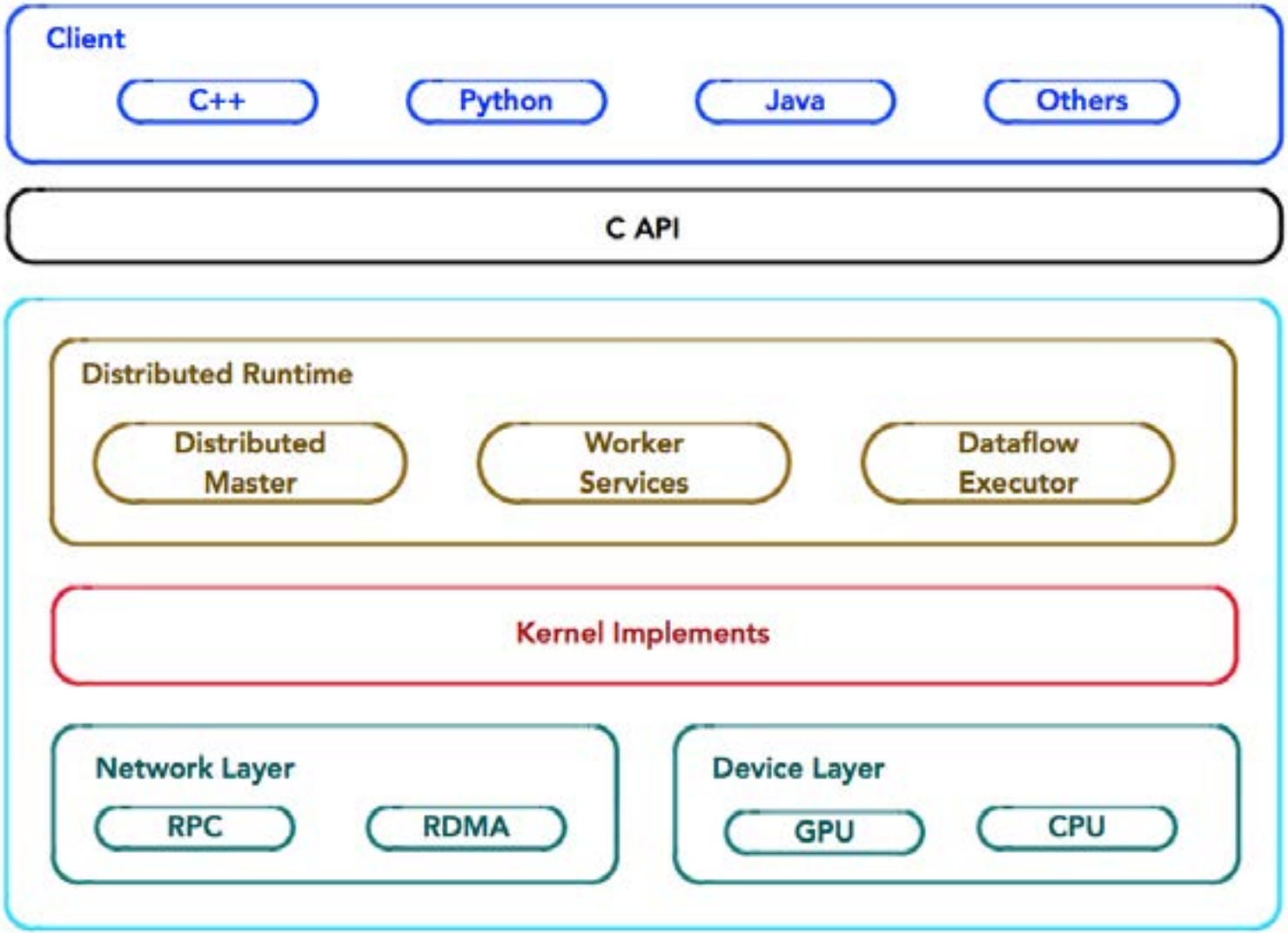
2017-08-05

内容

架构概述

1. 系统架构
2. 图控制

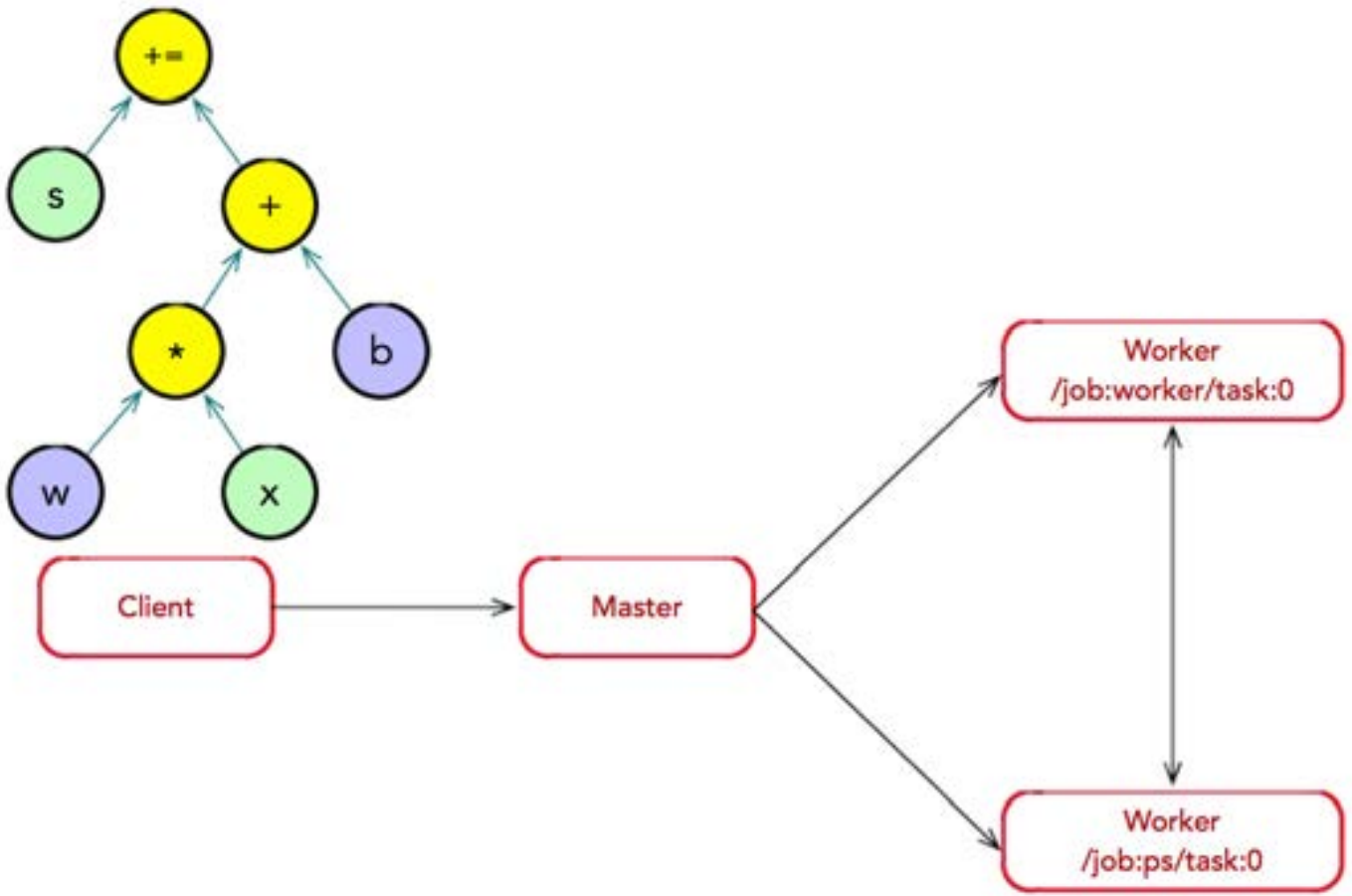
系统架构


Client
C++
Python
Java
Others
C API
Distributed Runtime
Distributed Master
Worker Services
Dataflow Executor
Kernel Implements
Network Layer
RPC
RDMA
Device Layer
GPU
CPU

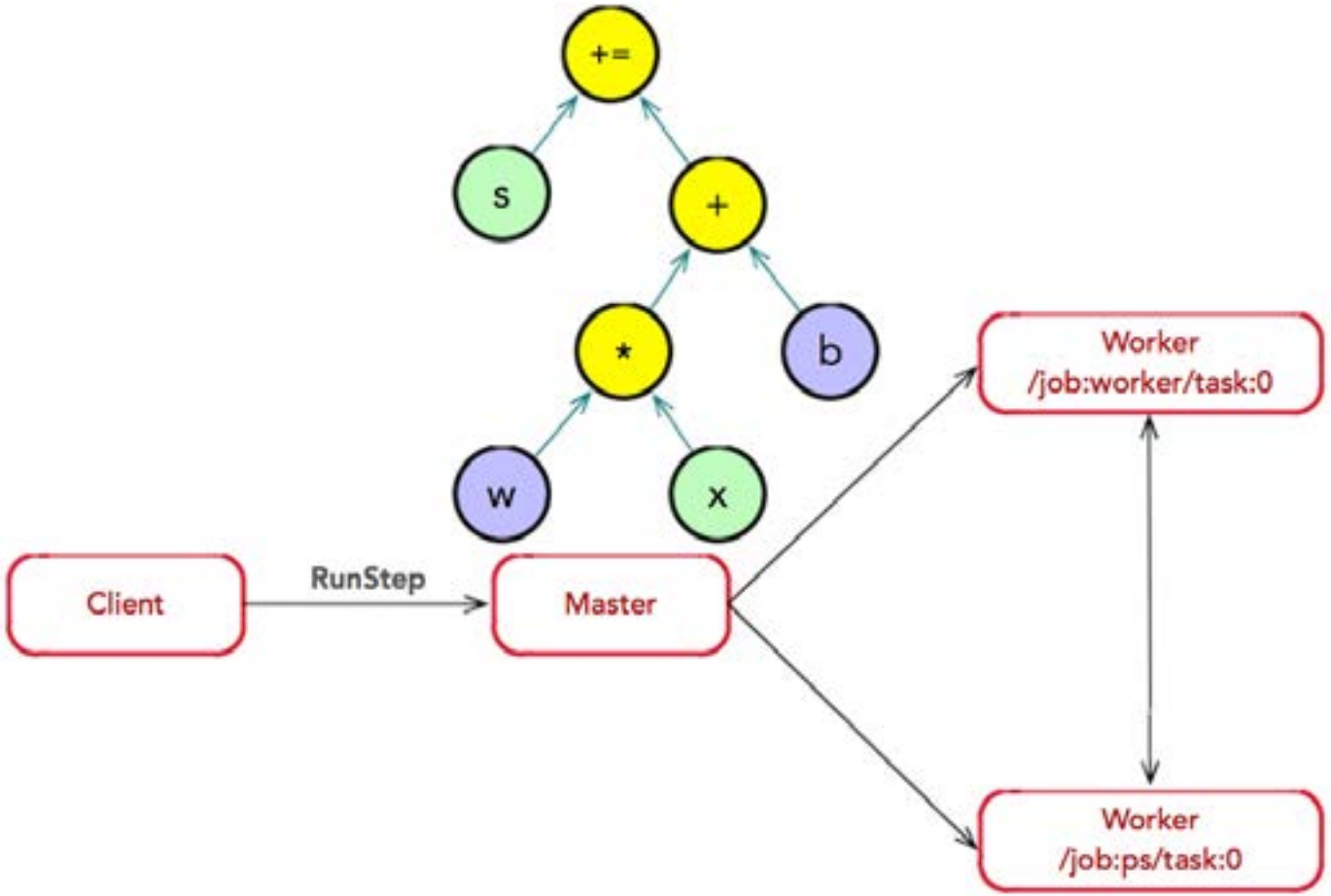
设计原则

- 延迟计算：图的构造与执行分离，并推迟计算图的执行过程
- 原子 OP：OP 是最小的抽象计算单元，支持构造复杂的网络模型
- 抽象设备：支持 CPU, GPU, ASIC 多种异构设备类型
- 抽象任务：基于 Task 的 PS，支持优化算法和网络模型的扩展

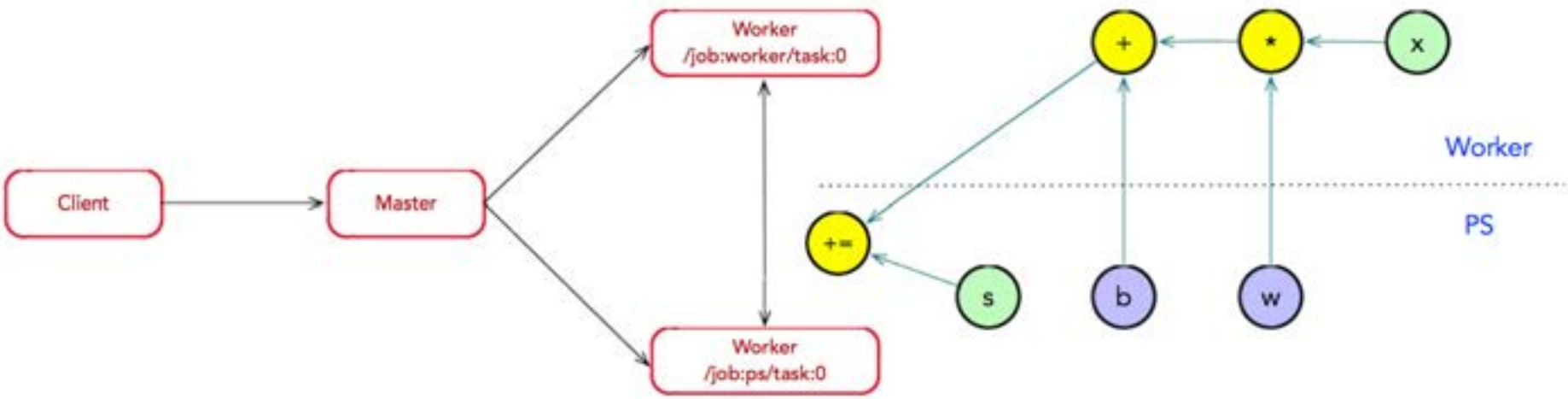
构造计算图


+=
s
+
*
b
w
x
Client
Master
Worker
/job:worker/task:0
Worker
/job:ps/task:0

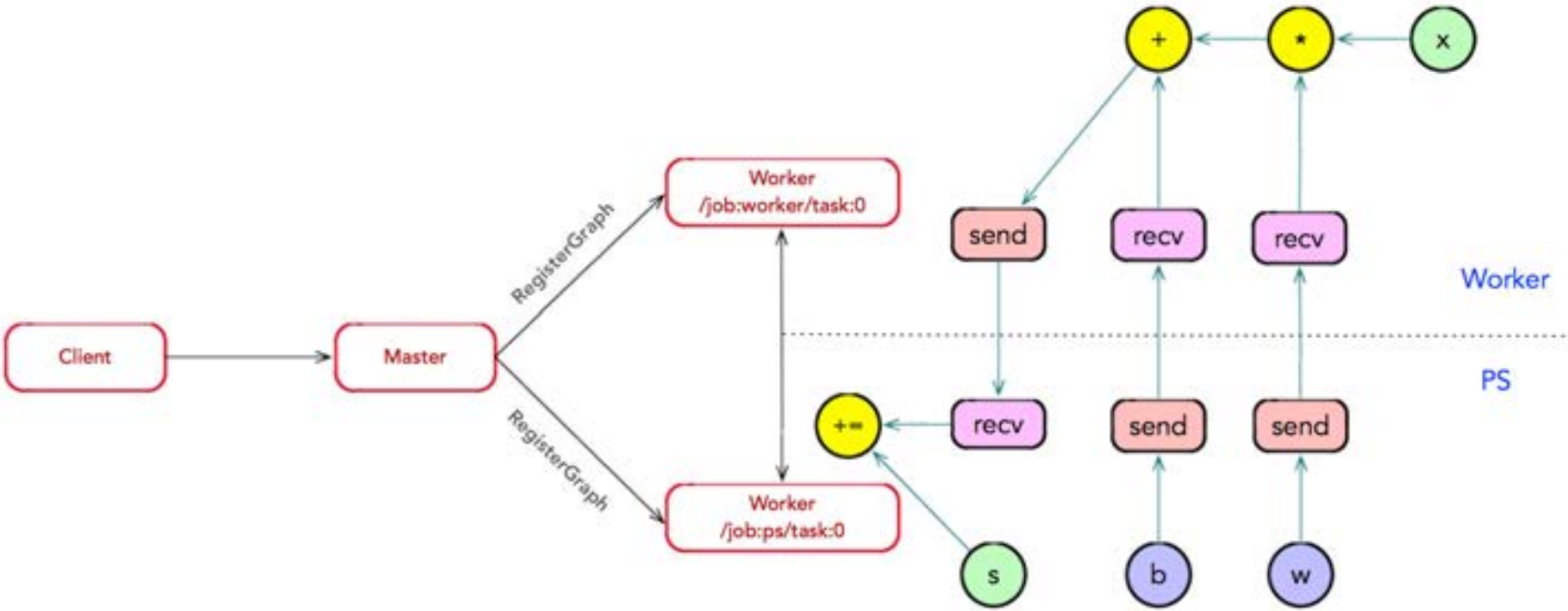
执行计算图


+=
s
+
*
b
w
x
Client
RunStep
Master
Worker
/job:worker/task:0
Worker
/job:ps/task:0

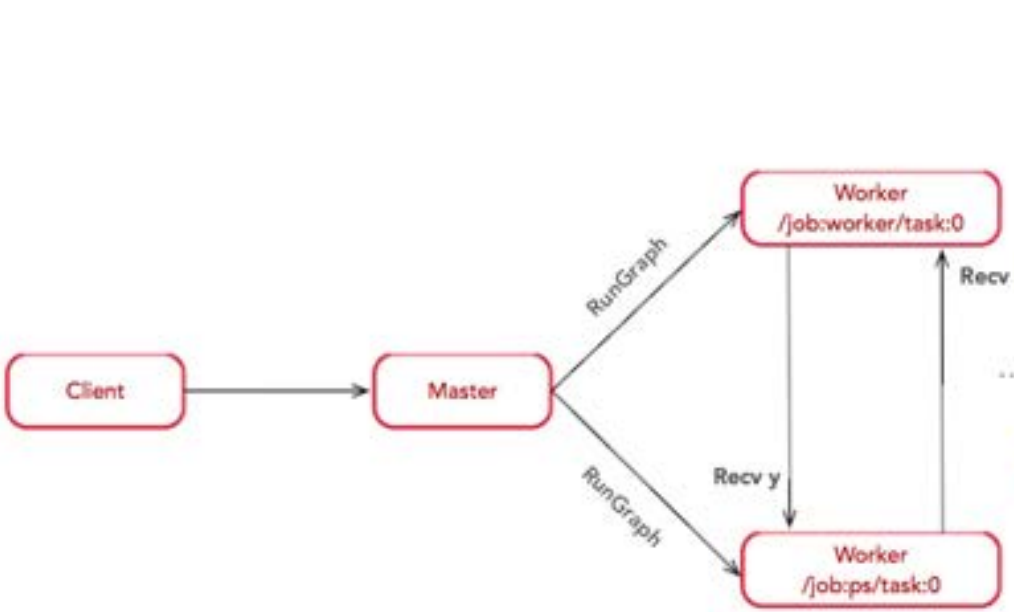
图分解：按 Task 分解

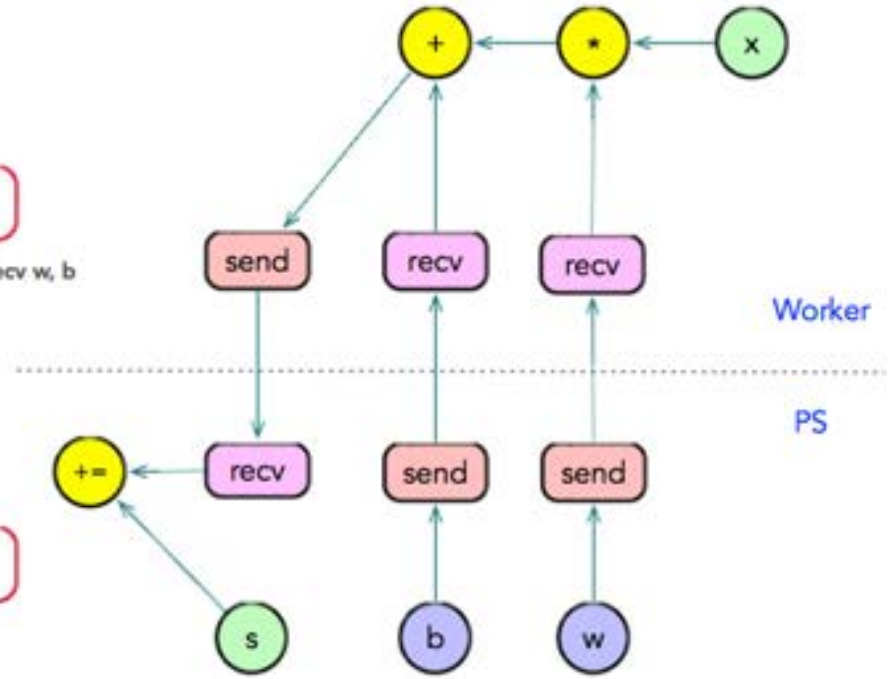

Client
Master
Worker
/job:worker/task:0
Worker
/job:ps/task:0
+
*
x
+=
s
b
w
Worker
PS

注册图


Client
Master
RegisterGraph
RegisterGraph
Worker
/job:worker/task:0
Worker
/job:ps/task:0
+
*
x
send
recv
recv
Worker
PS
+=
recv
send
send
s
b
w

执行图

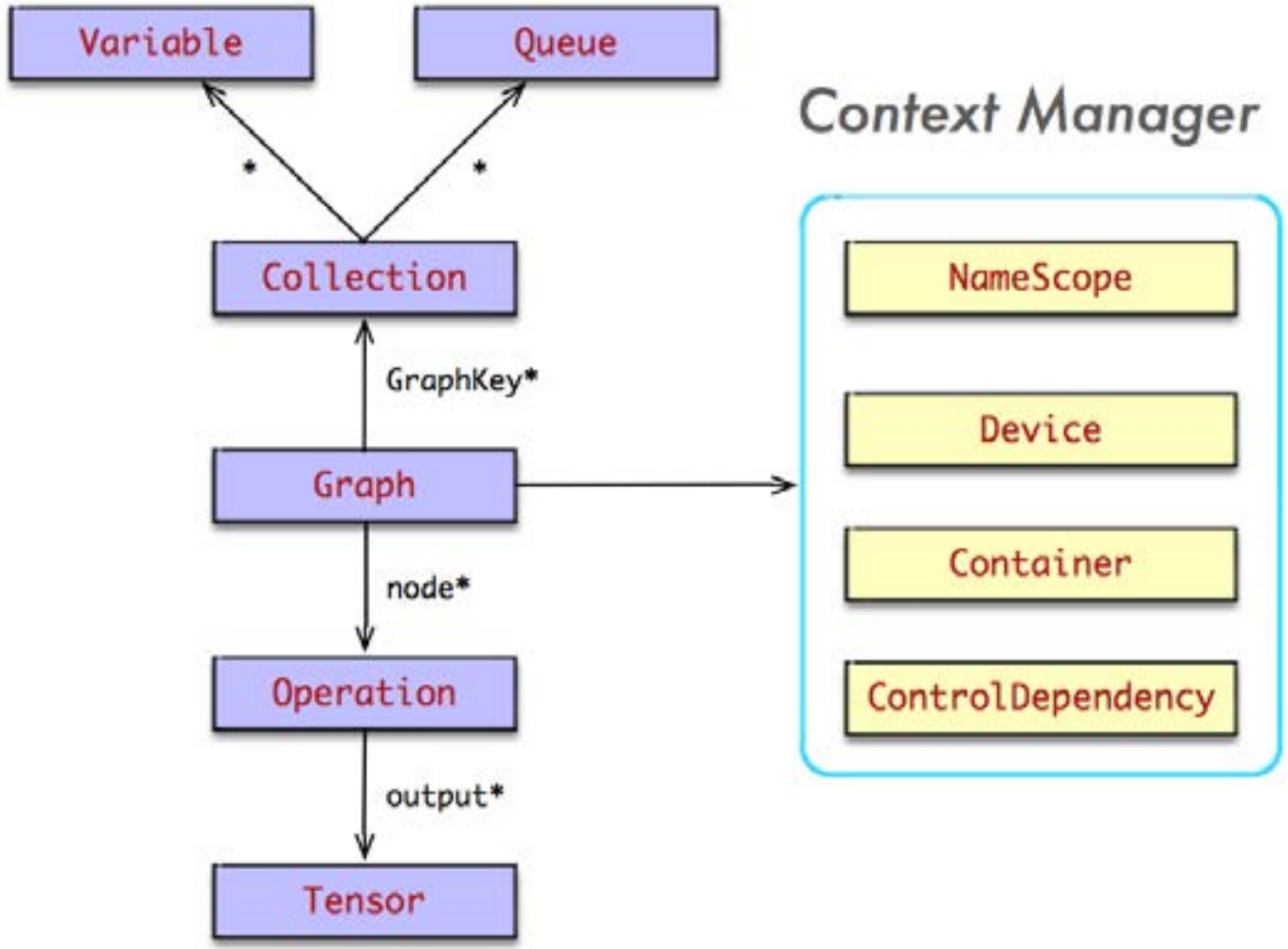

Client
Master
RunGraph
RunGraph
Worker
/job:worker/task:0
Worker
/job:ps/task:0
Recv w, b
Recv y


+
*
x
send
recv
recv
Worker
PS
+=
recv
send
send
s
b
w

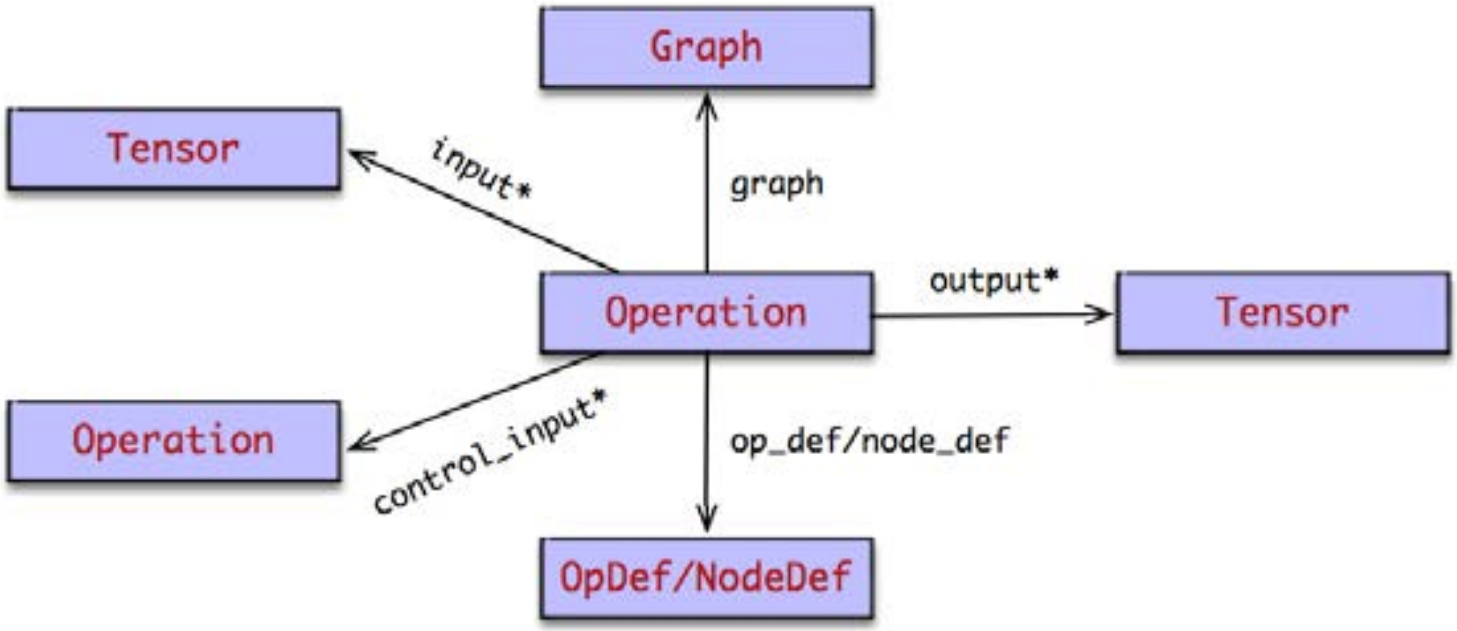
编程模型

1. 计算图
2. 变量
3. 会话
4. 图构造

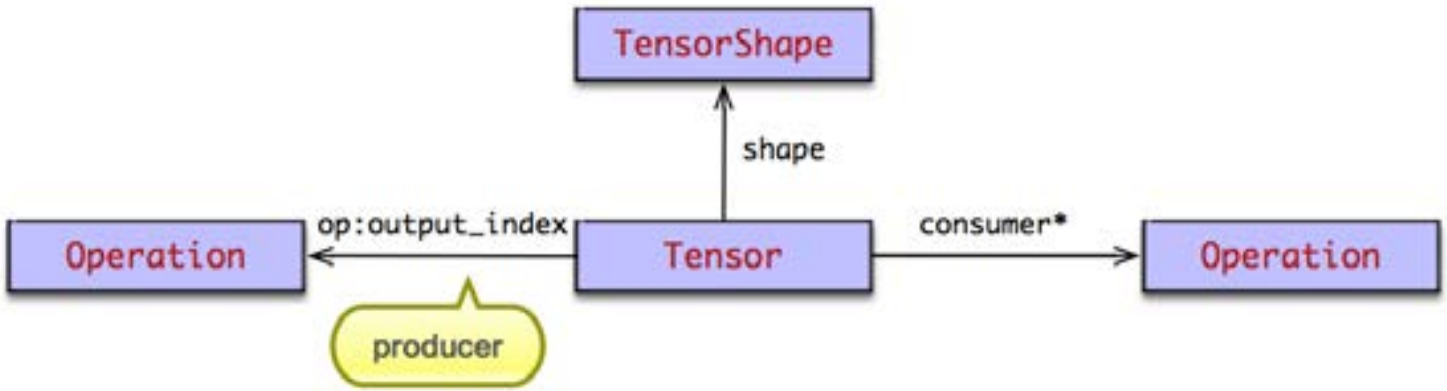
$Graph = Set\{OP\} + Set\{Tensor\}$



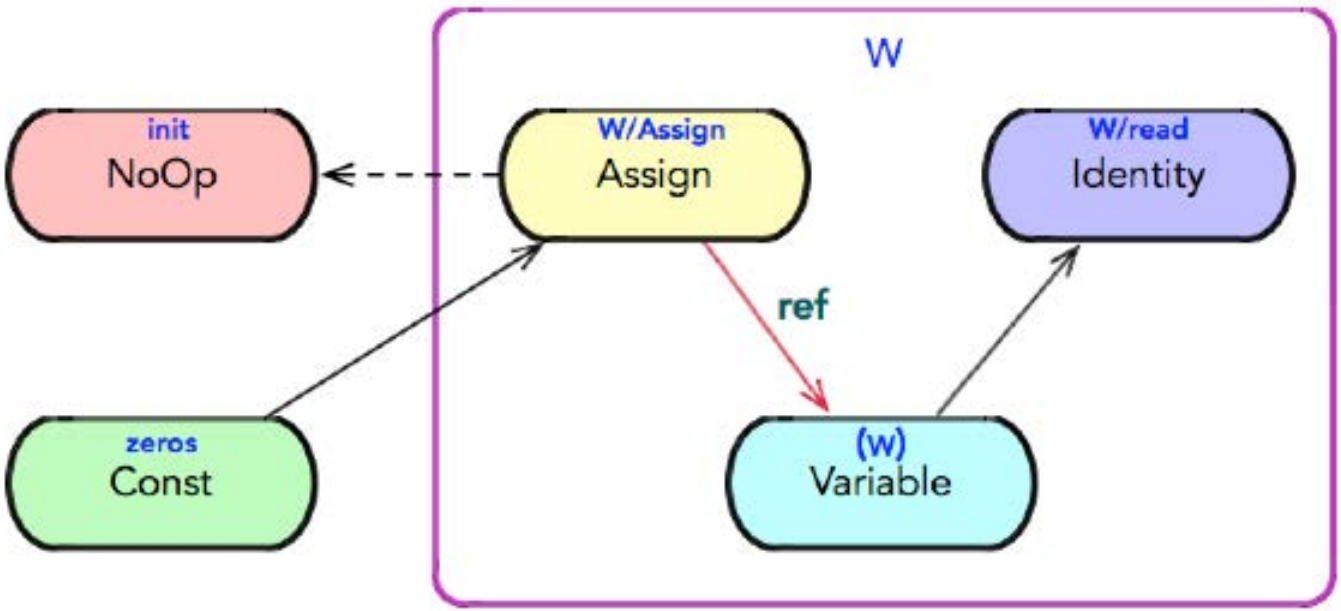
OP: 抽象计算


Graph
Tensor
input*
graph
Operation
output*
Tensor
Operation
control_input*
op_def/node_def
OpDef/NodeDef

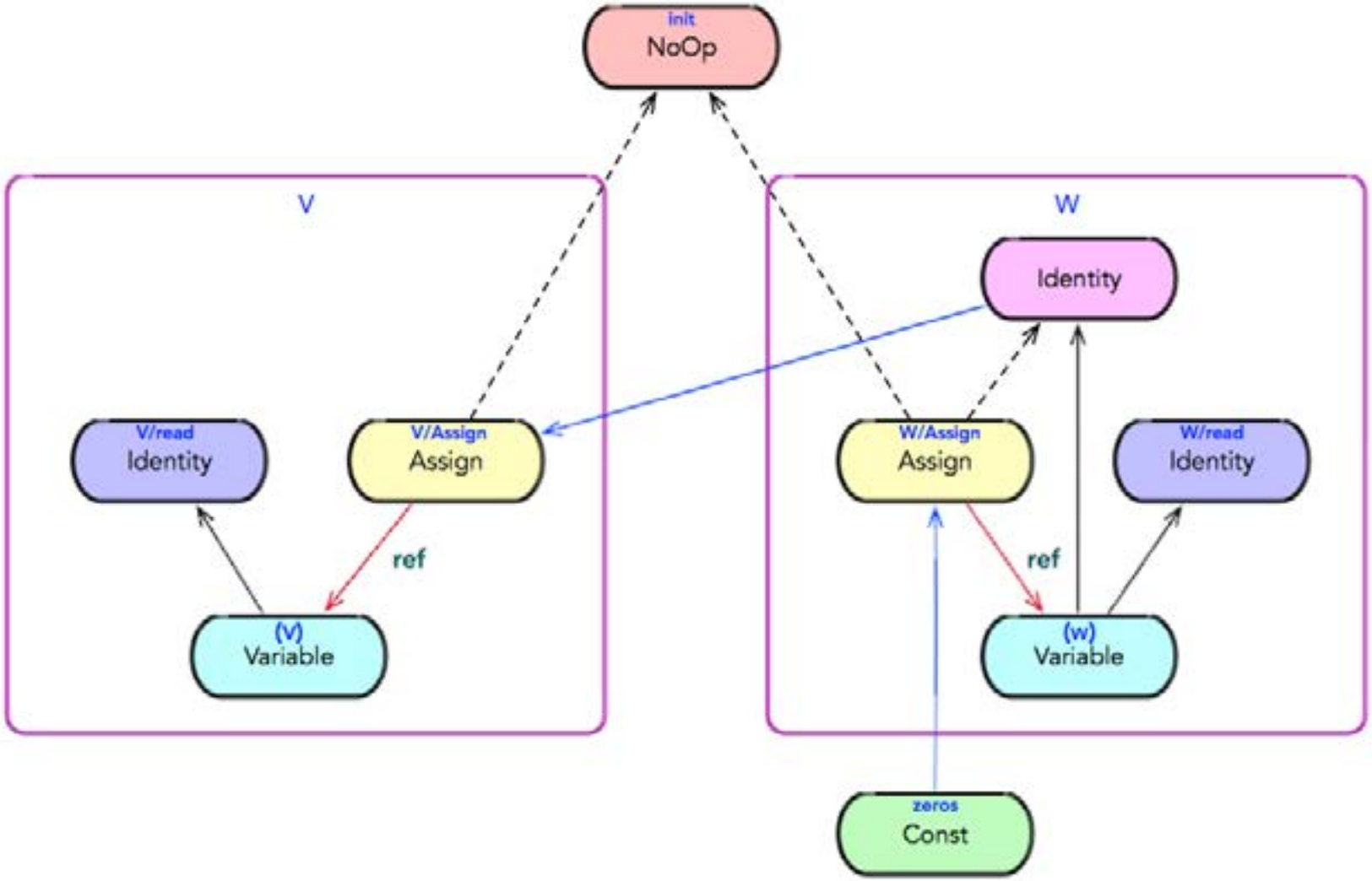
Tensor: 承载数据


TensorShape
shape
Operation
op:output_index
Tensor
consumer*
Operation
producer

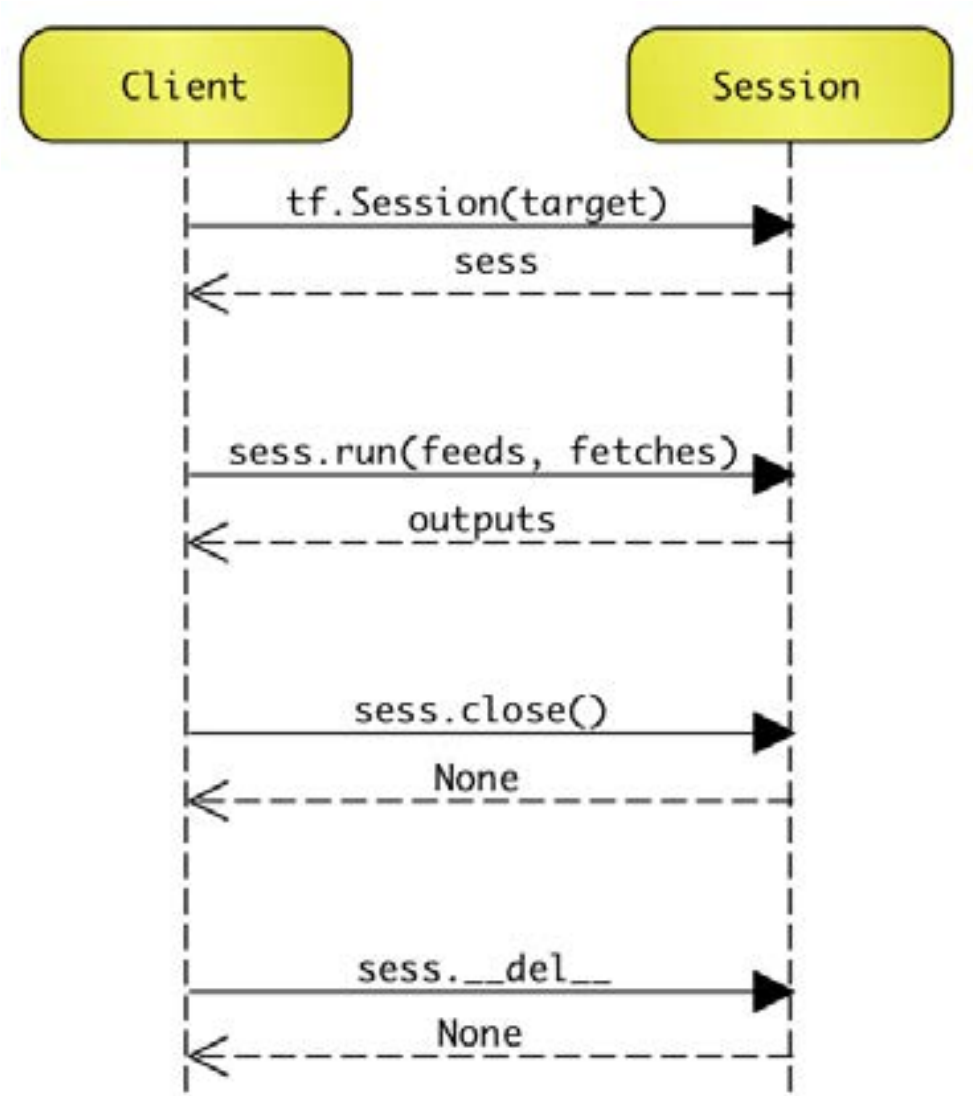
初始化模型


W
init
NoOp
W/Assign
Assign
W/read
Identity
ref
zeros
Const
(w)
Variable

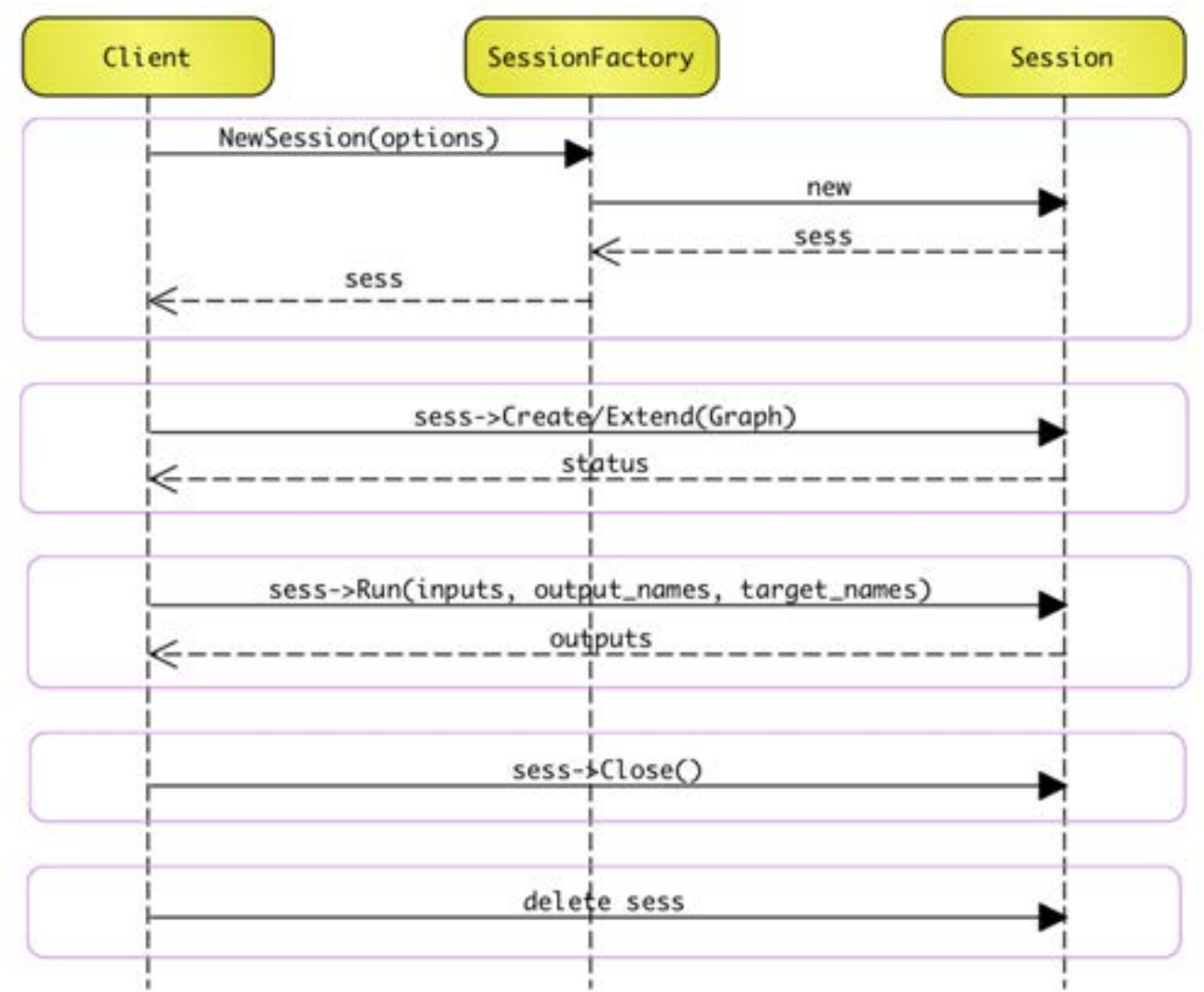
初始化依赖


init
NoOp
V
W
Identity
V/read
Identity
V/Assign
Assign
W/Assign
Assign
W/read
Identity
ref
ref
(V)
Variable
(w)
Variable
zeros
Const

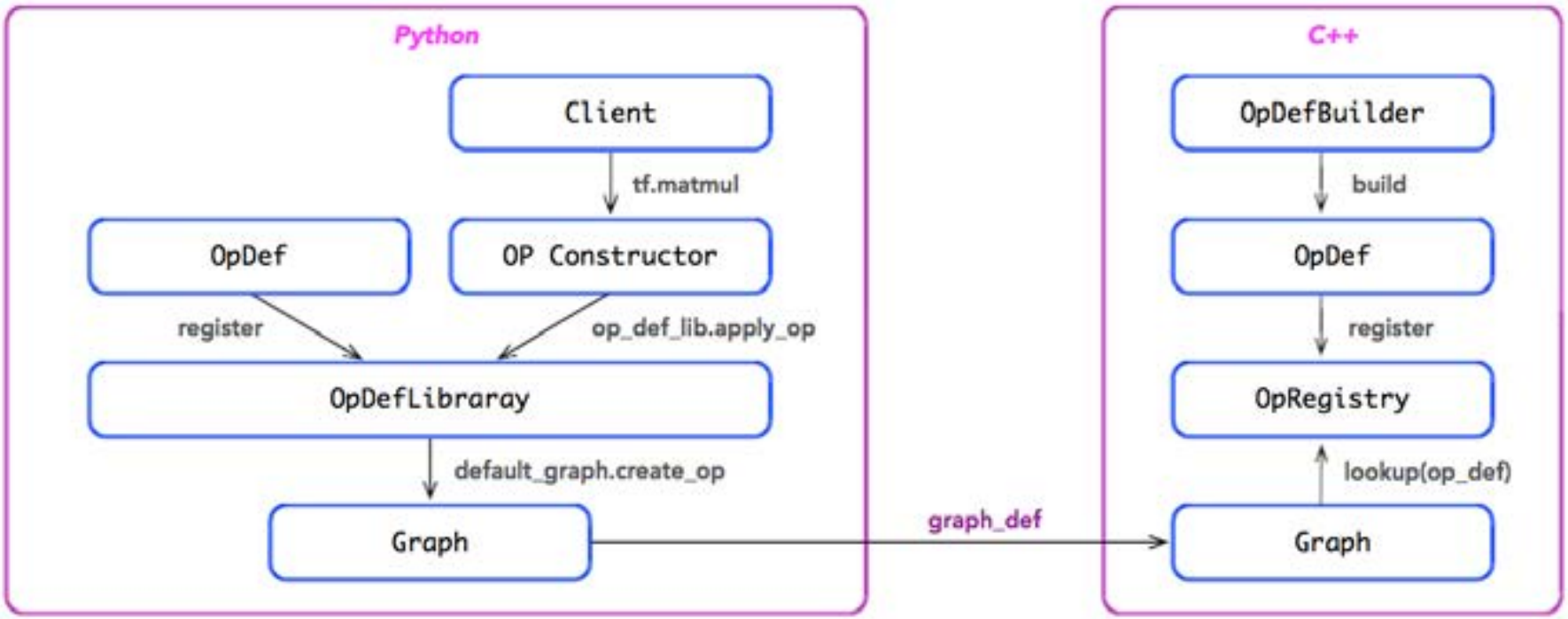
生命周期: Python


Client
Session
tf.Session(target)
sess
sess.run(feeds, fetches)
outputs
sess.close()
None
sess.__del__
None

生命周期: C++


Client
SessionFactory
Session
NewSession(options)
new
sess
sess
sess->Create/Extend(Graph)
status
sess->Run(inputs, output_names, target_names)
outputs
sess->Close()
delete sess

图构造与传递


Python
Client
tf.matmul
OpDef
OP Constructor
register
op_def_lib.apply_op
OpDefLibraray
default_graph.create_op
Graph
graph_def
C++
OpDefBuilder
build
OpDef
register
OpRegistry
lookup(op_def)
Graph

实例: OP 构造器

OP Constructor

```
def zeros_like(tensor, name=None):
  gen_array_ops._zeros_like(tensor, name=name)

tensor = tf.constant([1, 2], name="n1")
zeros = tf.zeros_like(tensor, name="n2")
```

Code Generator

```
def _zeros_like(dtype, shape=None, name=None):
 return _op_def_lib.apply_op("ZerosLike", x=x, name=name)
```

实例：构造 OP

OpDef Repository

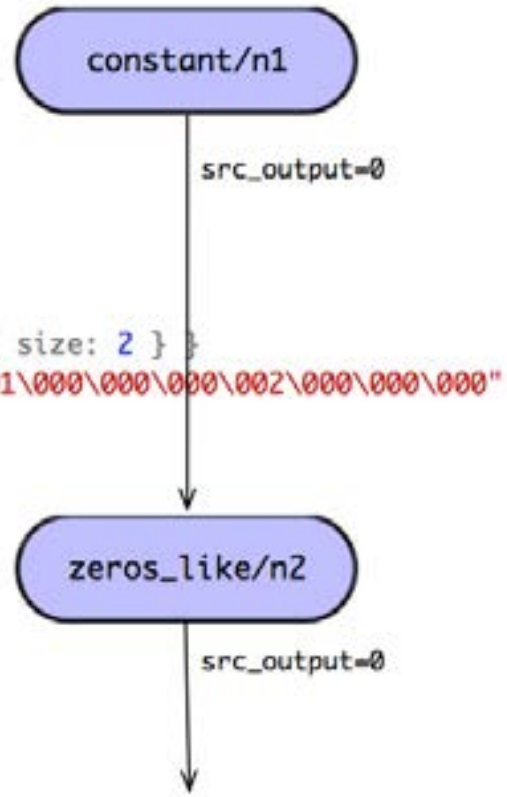
```
class OpDefLibrary(object):
  def apply_op(self, op_name, name=None, **keywords):
    inputs, input_types, output_types, attr_protos, op_def =
    with graph.as_default(), ops.name_scope(name) as scope:
      return graph.create_op(op_name, inputs, output_types, name=scope,
                             input_types=input_types, attrs=attr_protos, op_def=op_def)
```

Graph

```
class Graph(object):
  def create_op(self, op_type, inputs, dtypes, input_types=None,
                name=None, attrs=None, op_def=None):
    node_def, control_inputs = ...
    return Operation(node_def, self, inputs=inputs, output_types=output_types,
                     control_inputs=control_inputs, input_types=input_types,
                     op_def=op_def)
```

实例：图构造

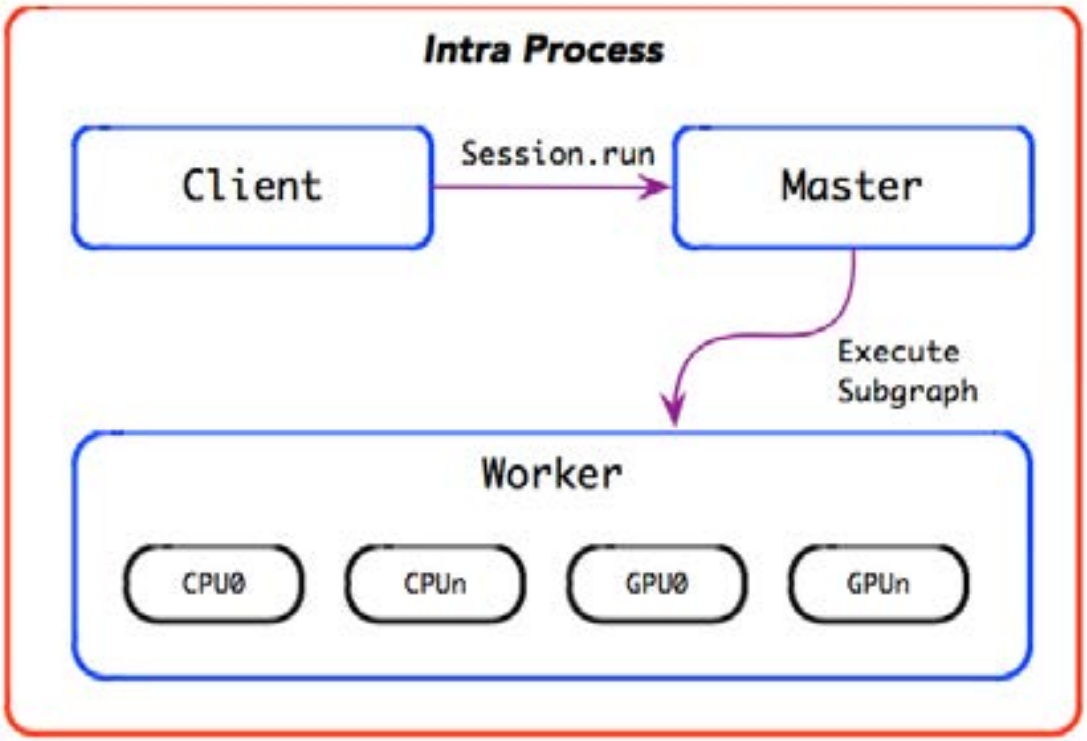
```
node {
  name: "n1"
  op: "Const"
  attr {
    key: "dtype"
    value { type: DT_INT32 }
  }
  attr {
    key: "value"
    value {
      tensor {
        dtype: DT_INT32
        tensor_shape { dim { size: 2 } }
        tensor_content: "\001\000\000\000\002\000\000\000"
      }
    }
  }
}
node {
  name: "n2"
  op: "ZerosLike"
  input: "node1"
  attr {
    key: "T"
    value { type: DT_INT32 }
  }
}
```



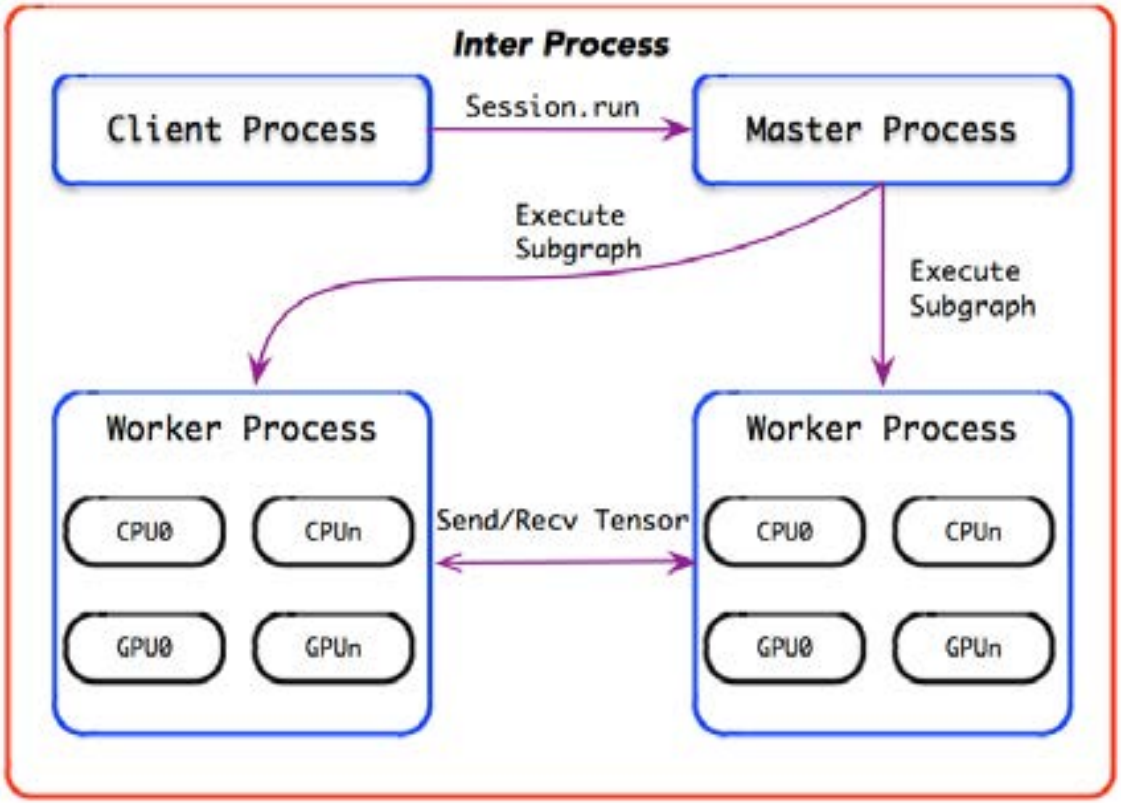
运行模型

1. 运行模式
2. 创建会话
3. 迭代执行

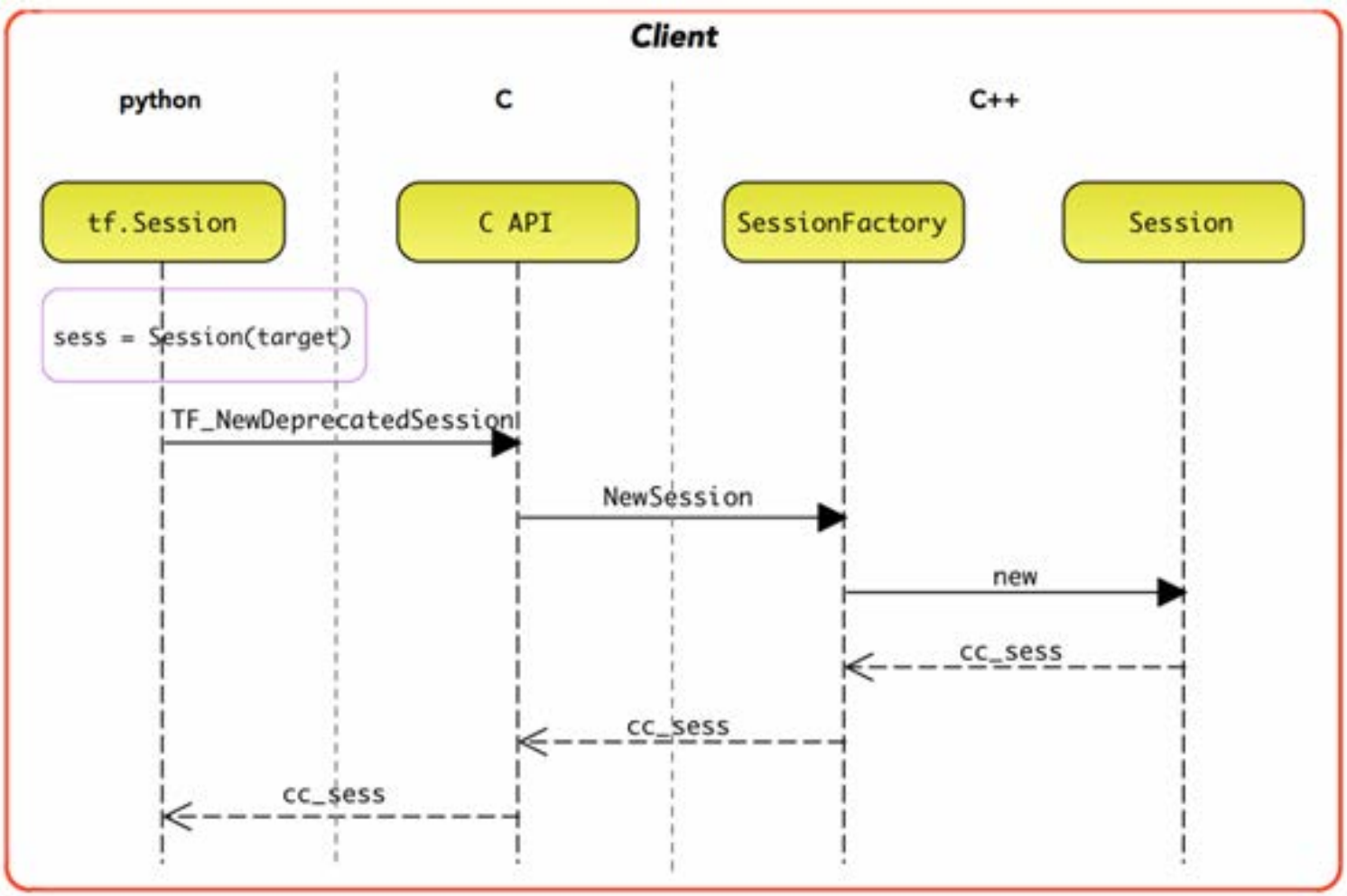
本地模式


Intra Process
Client
Session.run
Master
Execute
Subgraph
Worker
CPU0
CPUn
GPU0
GPUn

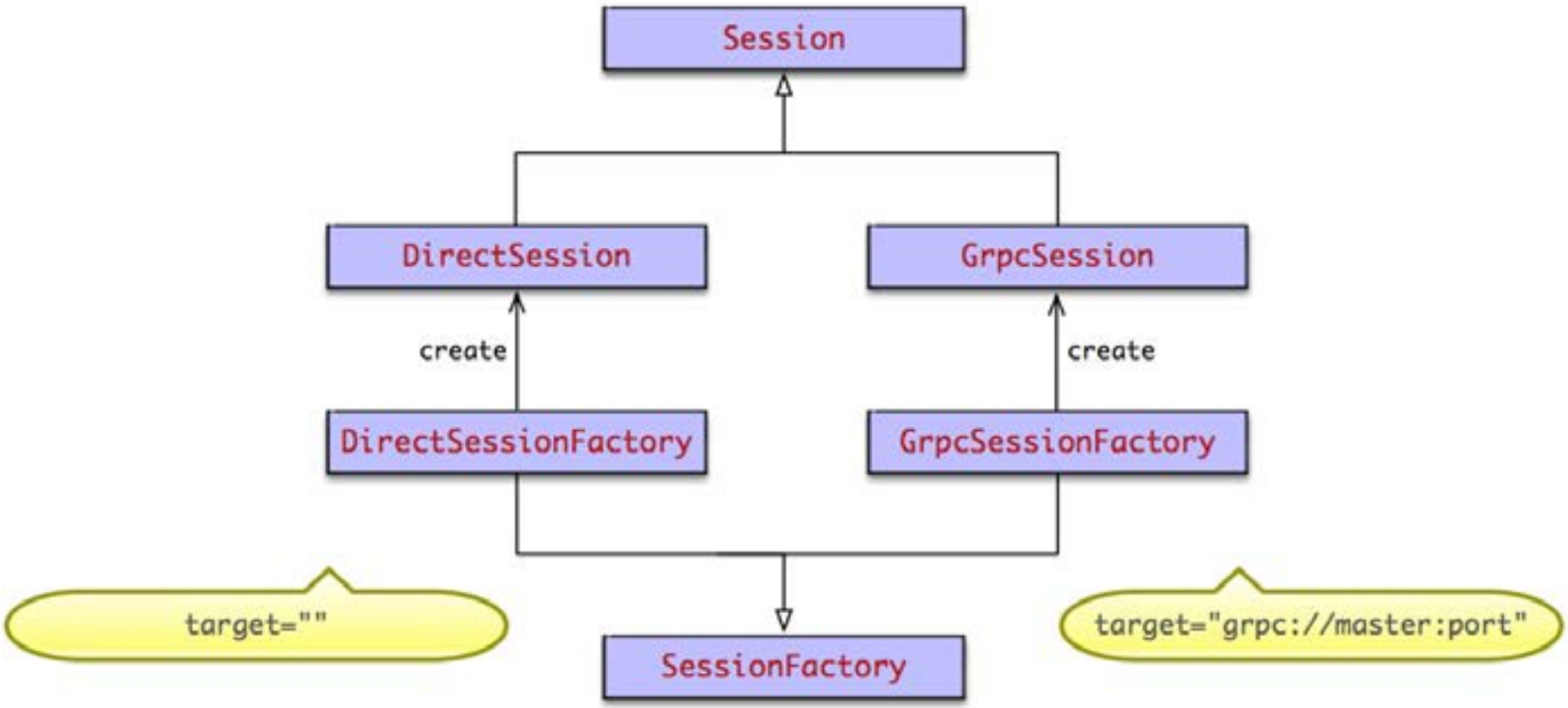
分布式模式


Inter Process
Client Process
Session.run
Master Process
Execute Subgraph
Execute Subgraph
Worker Process
CPU0
CPUn
GPU0
GPUn
Send/Recv Tensor
Worker Process
CPU0
CPUn
GPU0
GPUn

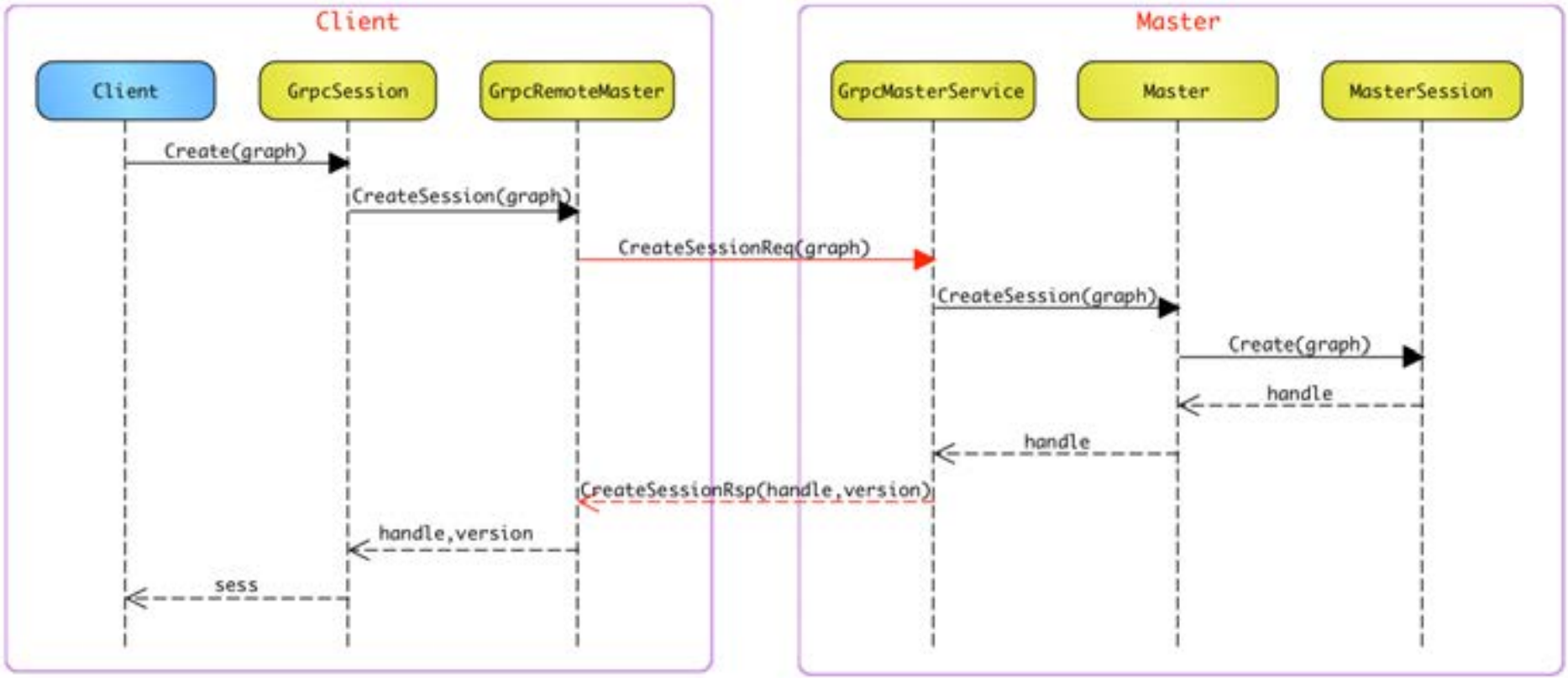
创建 ClientSession



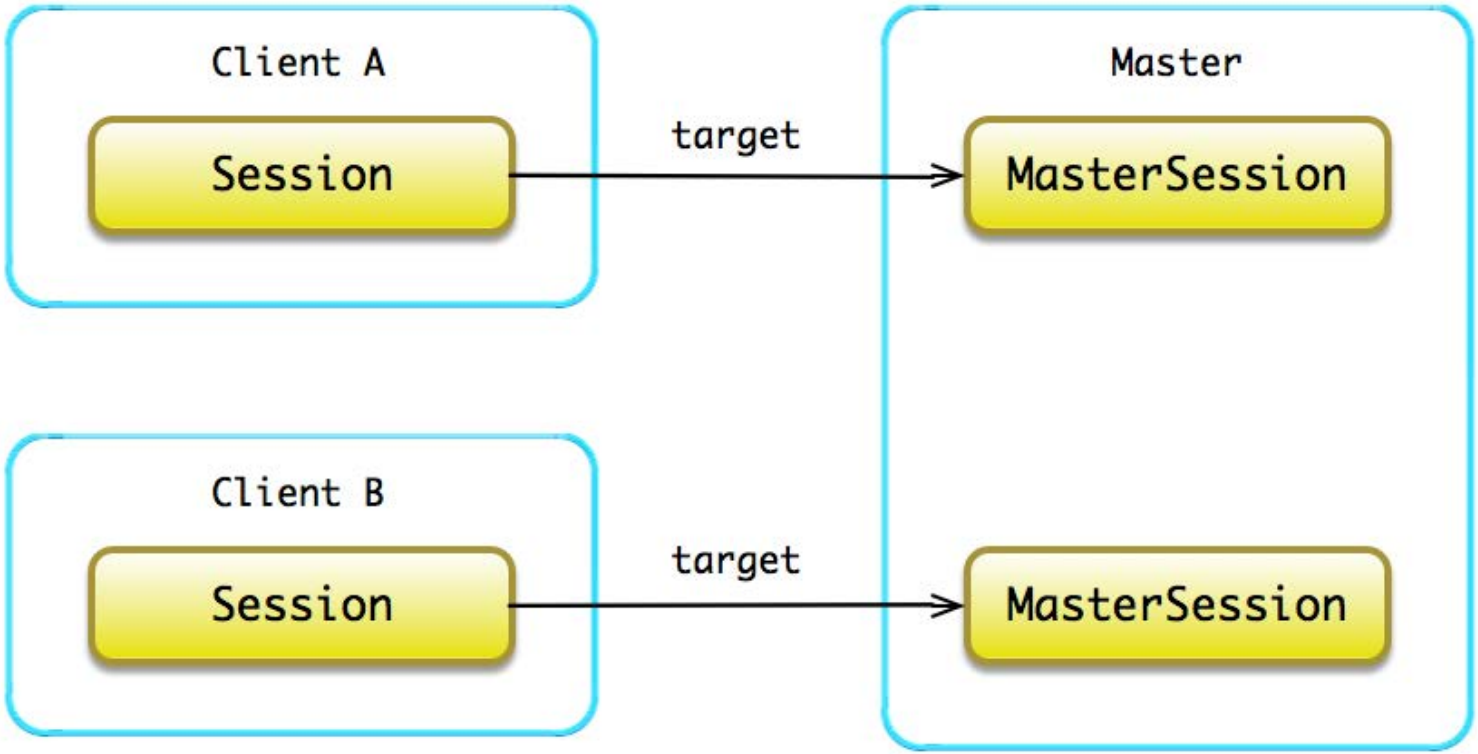
多态创建


Session
DirectSession
GrpcSession
create
create
DirectSessionFactory
GrpcSessionFactory
target=""
target="grpc://master:port"
SessionFactory

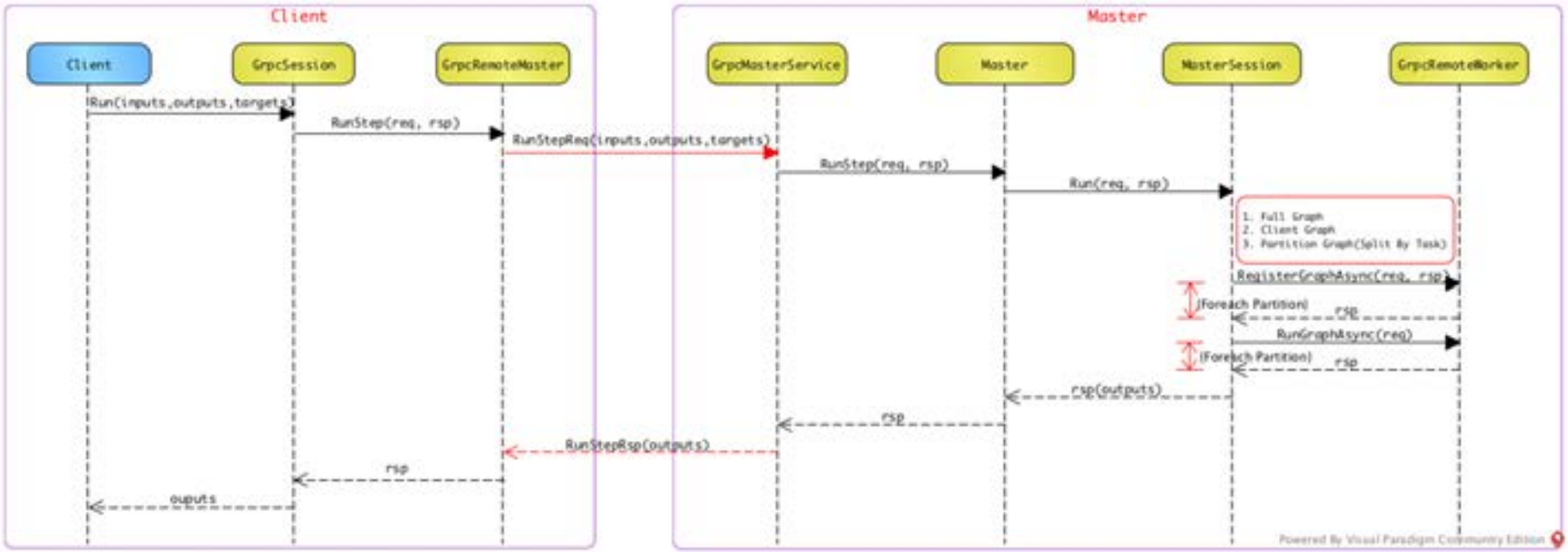
创建 MasterSession



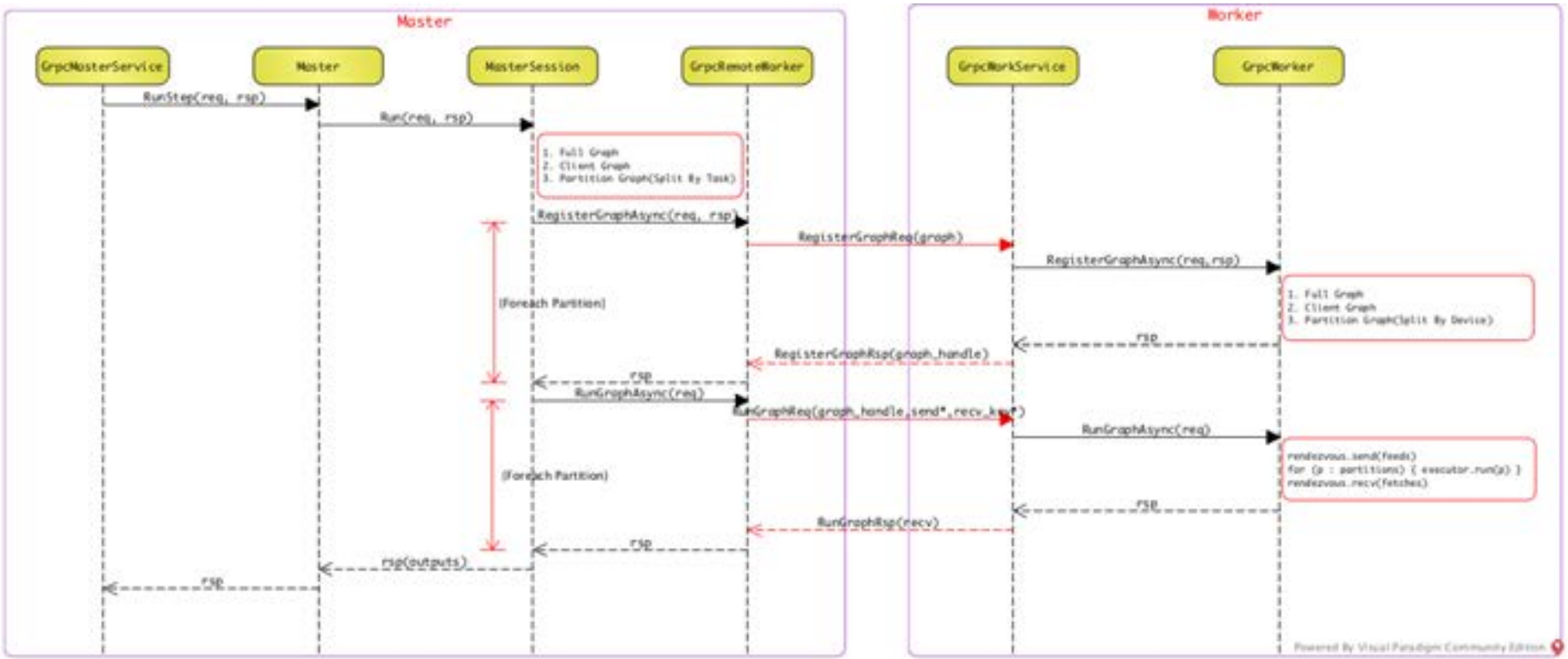
MasterSession 模型



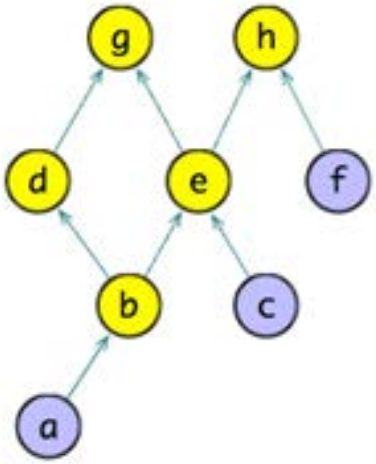
一级图分裂

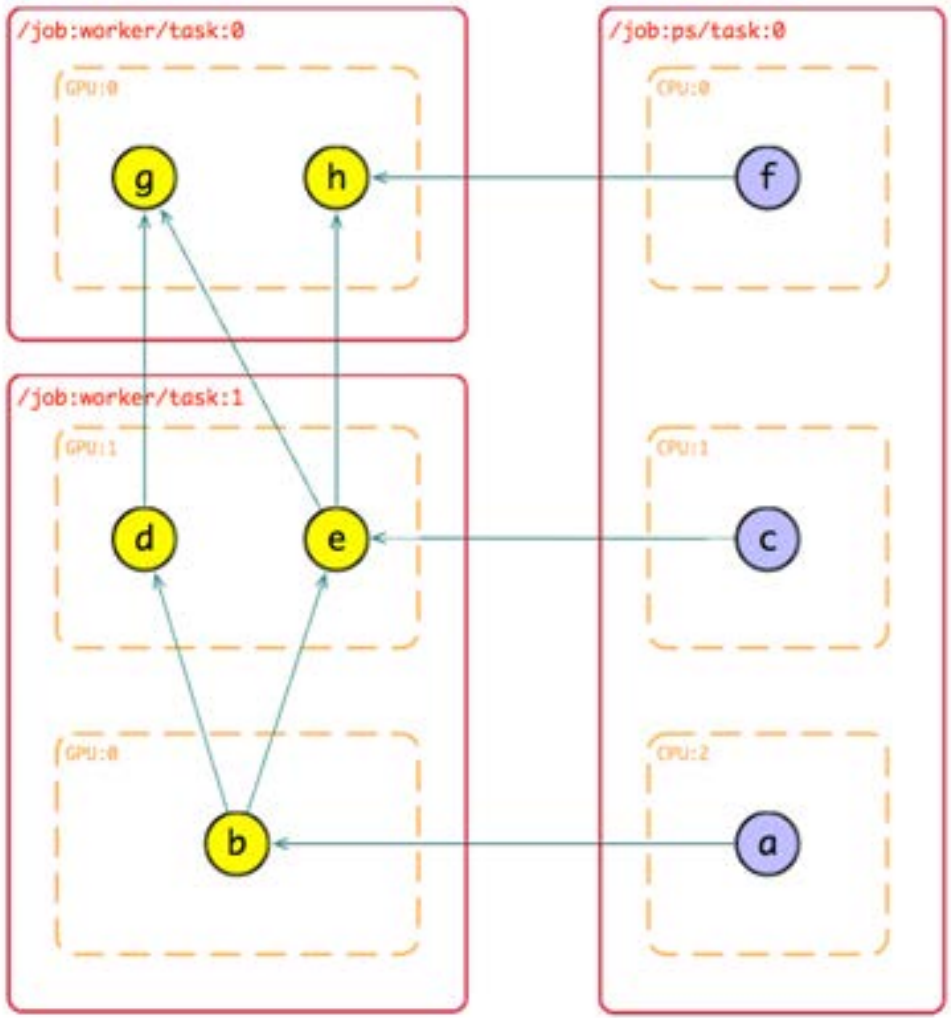

Client
Client
GrpcSession
GrpcRemoteMaster
Run(inputs,outputs,targets)
RunStep(req, rsp)
RunStepReq(inputs,outputs,targets)
rsp
ouputs
Master
GrpcMasterService
Master
MasterSession
GrpcRemoteWorker
RunStep(req, rsp)
Run(req, rsp)
1. Full Graph
2. Client Graph
3. Partition Graph(Split By Task)
RegisterGraphAsync(req, rsp)
(Foreach Partition)
rsp
RunGraphAsync(req)
(Foreach Partition)
rsp
rsp(outputs)
rsp
RunStepRsp(outputs)
Powered By Visual Paradigm Community Edition

二级图分裂

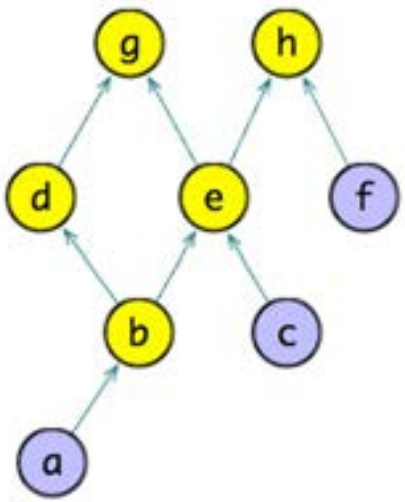

Master
GrpcMasterService
Master
MasterSession
GrpcRemoteWorker
RunStep(req, rsp)
Run(req, rsp)
1. Full Graph
2. Client Graph
3. Partition Graph(Split By Task)
RegisterGraphAsync(req, rsp)
RegisterGraphReq(graph)
[Foreach Partition]
rsp
RegisterGraphRsp(graph_handle)
rsp
RunGraphAsync(req)
RunGraphReq(graph_handle,send*,recv_key*)
[Foreach Partition]
rsp
RunGraphRsp(recv)
rsp
rsp(outputs)
rsp
Worker
GrpcWorkService
GrpcWorker
RegisterGraphAsync(req,rsp)
1. Full Graph
2. Client Graph
3. Partition Graph(Split By Device)
rsp
RunGraphAsync(req)
rendezvous.send(feeds)
for (p : partitions) { executor.run(p) }
rendezvous.recv(fetches)
rsp
Powered By Visual Paradigm Community Edition

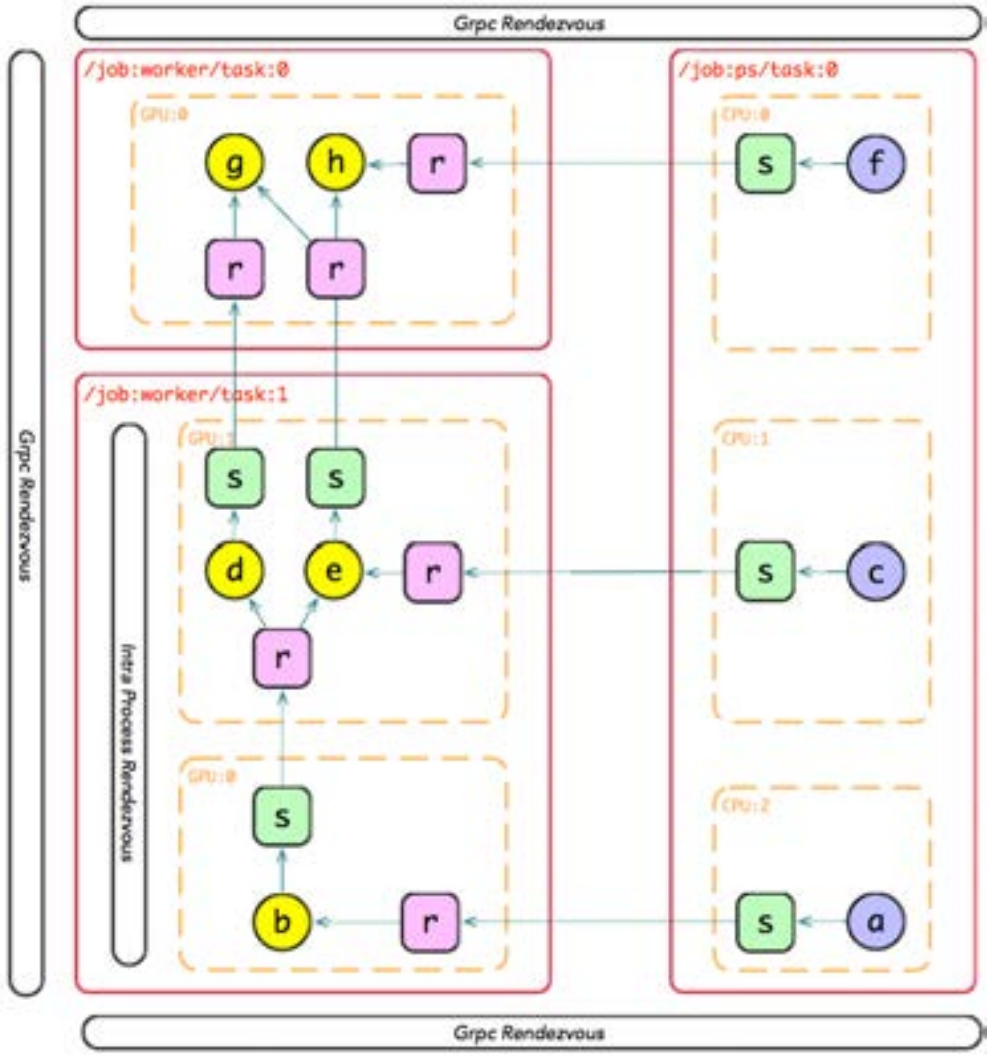
实例：图分裂


g
h
d
e
f
b
c
a


/job:worker/task:0
GPU:0
g
h
/job:ps/task:0
CPU:0
f
/job:worker/task:1
GPU:1
d
e
CPU:1
c
GPU:0
b
CPU:2
a

实例：交换数据


g
h
d
e
f
b
c
a


Grpc Rendezvous
/job:worker/task:0
/job:ps/task:0
/job:worker/task:1
Intra Process Rendezvous
Grpc Rendezvous
Grpc Rendezvous

训练模型

1. 计算梯度
2. 应用梯度
3. 工作流

优化器: 计算梯度

```
class Optimizer(object):
  def minimize(self, loss, var_list=None, global_step=None):
    grads_and_vars = self.compute_gradients(
      loss, var_list=var_list)
    return self.apply_gradients(
      grads_and_vars,
      global_step=global_step)

  def computea_gradients(loss, var_list):
    grads = gradients(loss, var_list, grad)
    return list(zip(grads, var_list))

  def gradients(loss, var_list, grads=1):
    ops_and_grads = {}
    for op in reversed_graph(loss).topological_sort():
      grad = op.grad_fn(grad)
      ops_and_grads[op] = grad
    return [ops_and_grads.get(var) for var in var_list]
```

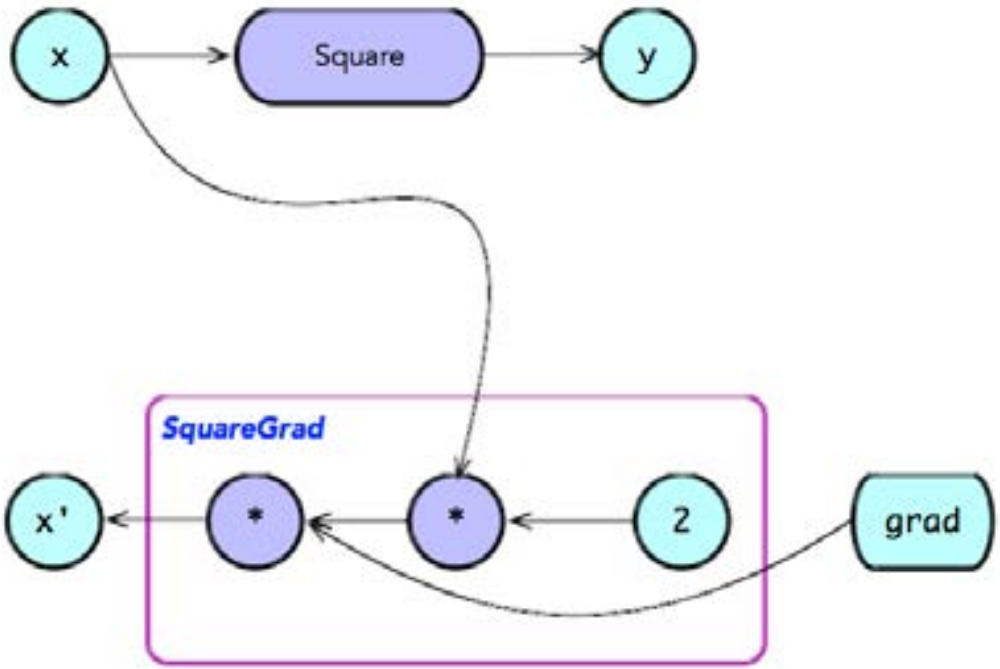
梯度函数

```
@ops.RegisterGradient("op_name")
def grad_func(op, grad):
  """construct gradient subgraph for an op type.
  Returns:
    A list of gradients, one per each input of op.
  """
  return cons_grad_subgraph(op, grad)
```

$$(y_1, y_2, ..., y_m) = f(x_1, x_2, ..., x_n)$$
$$(\partial L/\partial x_1, \partial L/\partial x_2, ..., \partial L/\partial x_n) = g\left(x_1, x_2, ..., x_n; \partial L/\partial y_1, \partial L/\partial y_2, ..., \partial L/\partial y_n\right)$$

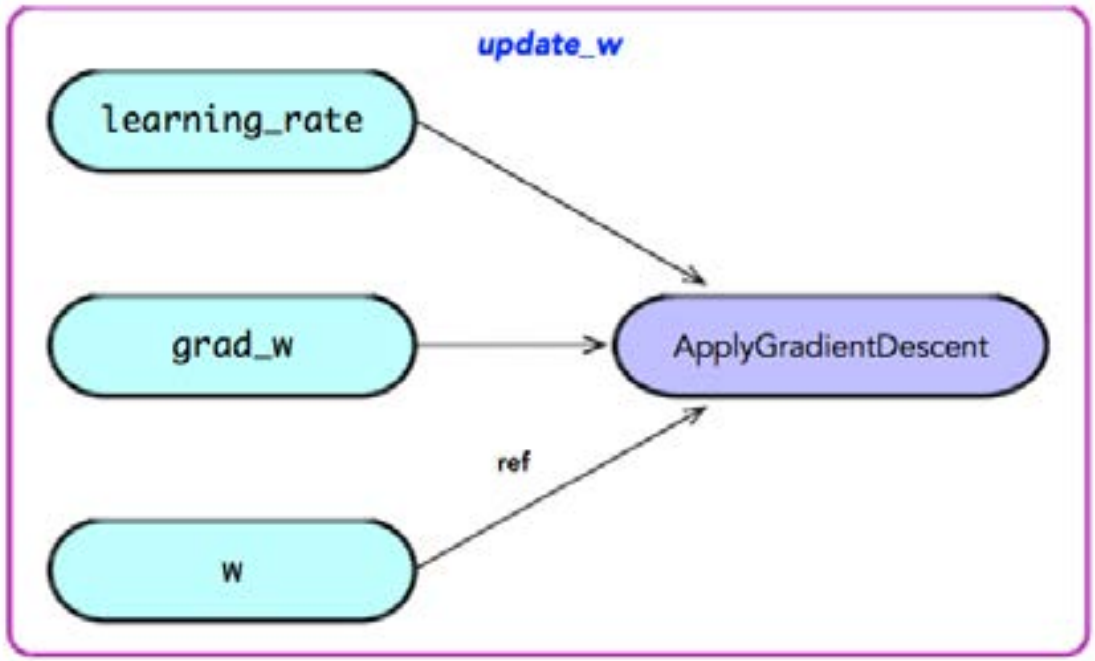
例子：SquareGrad 函数

```
@ops.RegisterGradient("Square")
def SquareGrad(op, grad):
  x = op.inputs[0]
  with ops.control_dependencies([grad.op]):
    return grad * (2.0 * x)
```

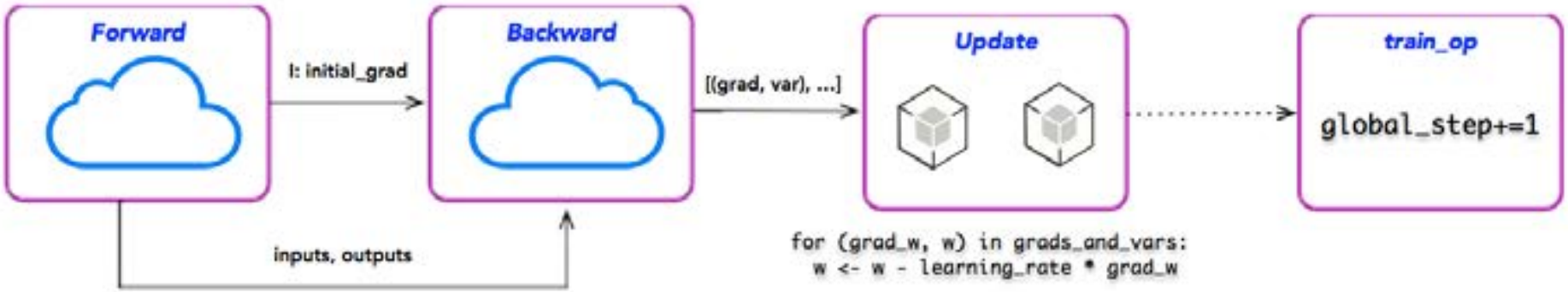


应用梯度

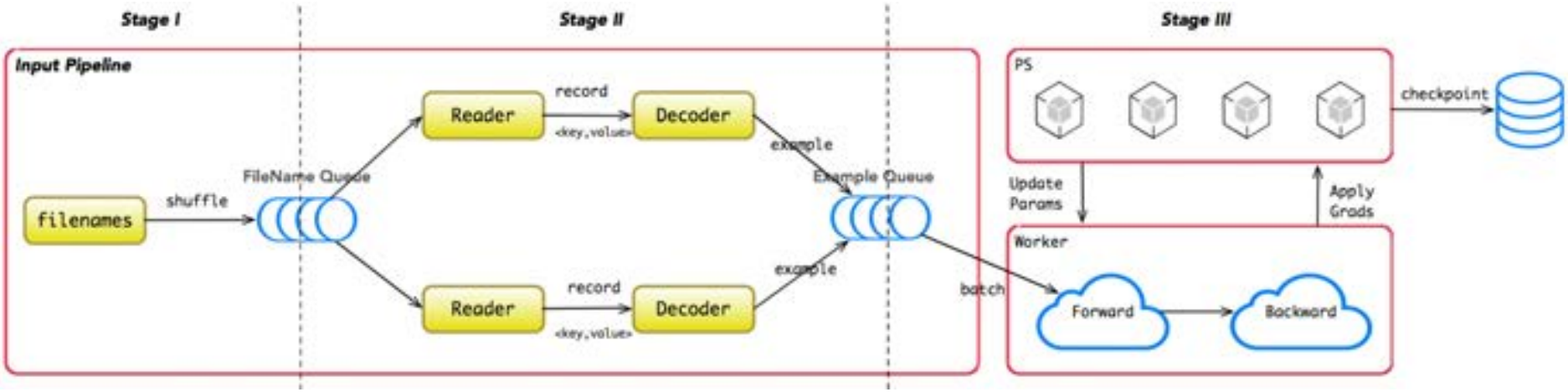
```
def apply_gradients(grads_and_vars, learning_rate):
  for (grad, var) in grads_and_vars:
    apply_gradient_descent(learning_rate, grad, var)
```



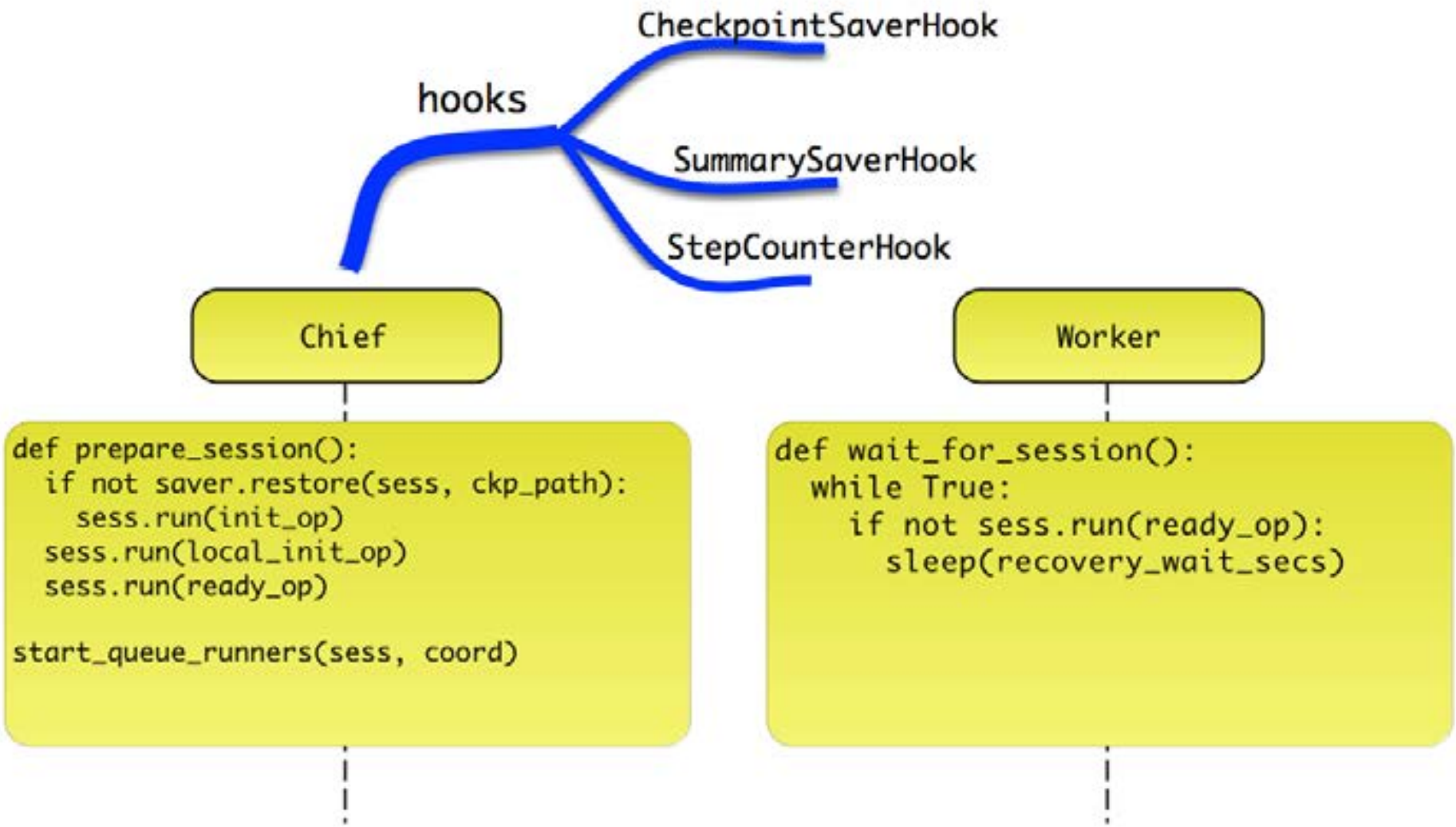
RunStep 过程


Forward
I: initial_grad
Backward
[(grad, var), ...]
Update
train_op
global_step+=1
inputs, outputs
for (grad_w, w) in grads_and_vars:
w <- w - learning_rate * grad_w

大规模模型训练

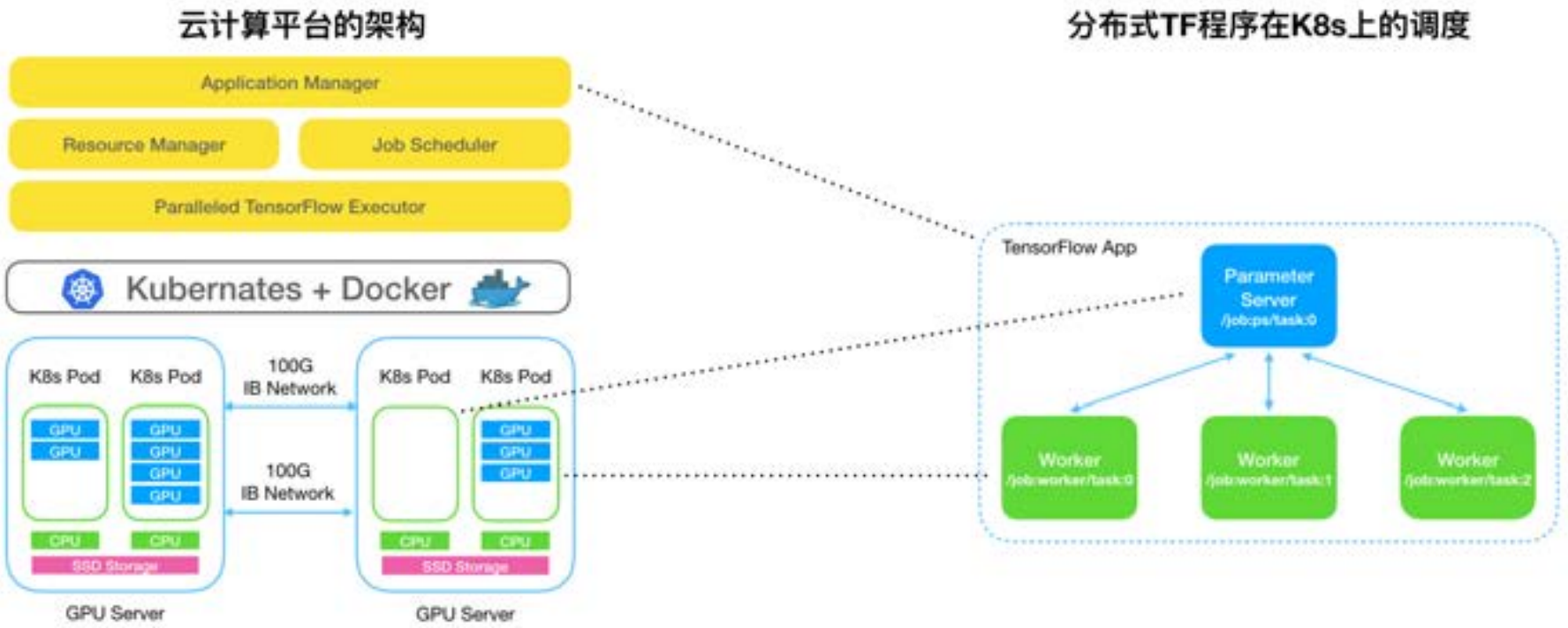

Stage I
Stage II
Stage III
Input Pipeline
filenames
shuffle
FileName Queue
Reader
Reader
record
record
Decoder
Decoder
example
example
Example Queue
batch
PS
checkpoint
Update Params
Apply Grads
Worker
Forward
Backward

初始化模型


CheckpointSaverHook
hooks
SummarySaverHook
StepCounterHook
Chief
Worker
def prepare_session():
if not saver.restore(sess, ckp_path):
sess.run(init_op)
sess.run(local_init_op)
sess.run(ready_op)
start_queue_runners(sess, coord)
def wait_for_session():
while True:
if not sess.run(ready_op):
sleep(recovery_wait_secs)

Tensorflow 遇上 Kubernetes

容器化高性能计算集群



- 使用Nvidia DGX-1搭建高性能计算集群
- 基于Docker容器使用GPU资源，在一个容器中映射多个GPU设备
- 使用Kubernates管理集群资源，为每个训练/推理作业创建多个POD

- Application Manager：管理每个TensorFlow App的Pod List
- Resource Manager：决定为一个App分配哪些Pod
- Job Scheduler：决定在Pod上部署哪些TF进程(Worker or PS)
- Executor：运行并管理容器内的TF进程

扩展 K8s 支持 GPU 和 RMDA 网络

原生K8s不支持GPU和RDMA网络资源的管理，我们对K8s进行了如下扩展：

- 能够从K8s查询节点的GPU设备信息，如GPU个数、显存大小等
- 能够在Pod List中指定映射哪个GPU
- 能够查询节点RDMA网络的信息
- 支持－cpuset-cpus选项，在某些场景能够提升100%性能。
- 资源分配策略是最容易变化的部分。为了减少对K8s的影响，我们把K8s看做是执行器而不是资源调度器。分离了单独的RM子系统来实现多种调度策略（为了简化RM，假设K8s集群只运行TF业务）。

```
apiVersion: v1kind: Pod
metadata:
  name: s1
spec:
  containers:
  - name: s1
    image: library/nginx:v1.0
    imagePullPolicy: IfNotPresent
    resources:
      limits:
        alpha.kubernetes.io/gpuset: 0,1,3
  nodeSelector:
    kubernetes.io/hostname: 10.114.51.204
```

在Pod中指定GPU

```
"labels": {
  "beta.kubernetes.io/arch": "amd64",
  "beta.kubernetes.io/os": "linux",
  "kubernetes.io/hostname": "10.114.51.204",
  "alpha.kubernetes.io/infiniBand": "true"
}
```

RDMA能力发现，通过给相应的节点打标签方式实现

在云上执行分布式 TF 计算

在云环境运行TF并不轻松

- 云环境下每次分配的资源是动态的

```
# On ps0.example.com:
$ python trainer.py \
    --ps_hosts=ps0.example.com:2222,ps1.example.com:2222 \
    --worker_hosts=worker0.example.com:2222,worker1.example.com:2222 \
    --job_name=ps --task_index=0
# On ps1.example.com:
$ python trainer.py \
    --ps_hosts=ps0.example.com:2222,ps1.example.com:2222 \
    --worker_hosts=worker0.example.com:2222,worker1.example.com:2222 \
    --job_name=ps --task_index=1
# On worker0.example.com:
$ python trainer.py \
    --ps_hosts=ps0.example.com:2222,ps1.example.com:2222 \
    --worker_hosts=worker0.example.com:2222,worker1.example.com:2222 \
    --job_name=worker --task_index=0
# On worker1.example.com:
$ python trainer.py \
    --ps_hosts=ps0.example.com:2222,ps1.example.com:2222 \
    --worker_hosts=worker0.example.com:2222,worker1.example.com:2222 \
    --job_name=worker --task_index=1
```

- 硬编码分布式方案不适合动态的环境（希望弹性计算）

```
with tf.device("/job:ps/task:0"):
  weights_1 = tf.Variable(...)
  biases_1 = tf.Variable(...)

with tf.device("/job:ps/task:1"):
  weights_2 = tf.Variable(...)
  biases_2 = tf.Variable(...)
```

不灵活，写起来也很烦！

- 希望最大化的优化性能（怎样分布式是最合理的）

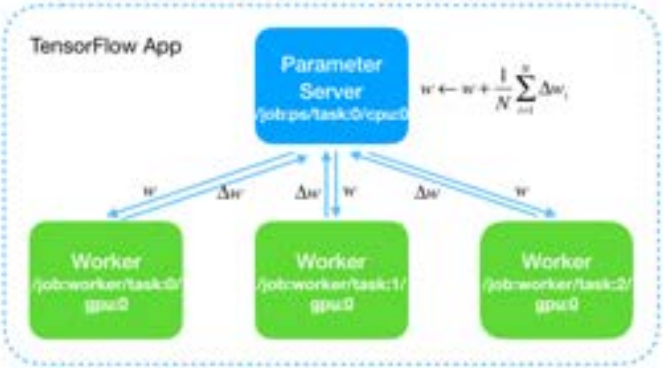
解决之道：自动化实现TF的分布式执行

TensorFlow App 自动化分布式执行流程

机器学习算法预测+模拟退火算法优化，相对于简单贪心策略提升20%-25%的性能

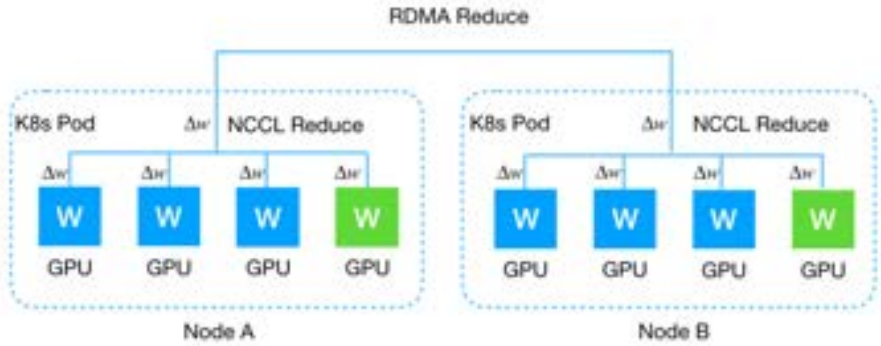
计算模型优化

经典的分布式计算模型效率低下



- 跨Pod之间的通信需要走容器网络，带来额外开销
- GPU-CPU之间的数据传输非常慢

全GPU计算模型



- 如果TF app部署在一个节点，只创建一个容器，映射多个GPU。利用NCCL all-reduce在GPU之间完成参数更新，不经过CPU，也不使用Parameter Server。
- 如果TF app部署跨多个节点，则在每个节点上创建一个容器。每个节点内，先将参数更新one-reduce到一张GPU卡上（主卡），再在每个节点的主卡之间用RDMA网络执行all-reduce。最后主卡将更新后的参数利用NCCL broadcast下发给本节点的所有GPU。

参考文献

文献

- TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems, Google Inc.
- TensorFlow: A System for Large-Scale Machine Learning, Google Inc.

Q&A

致谢

Thanks