**技术文档说明：**

**class Fleet(object)：**

Fleet是一个基类，transpiler和pslib都是Fleet的实现。

**参数：**

mode(Mode)：实现Fleet模式。

**返回：**None

**主要函数方法：**

* **init(role\_maker=None)**

fleet初始化，需在使用其他接口前先调用，用于定义多机环境配置

* **is\_worker()**

Parameter Server训练中使用，判断当前节点是否为Worker节点，是则返回True，否则返回False

* **is\_server(model\_dir=None)**

Parameter Server训练中使用，判断当前节点是否是Server节点，是则返回True，否则返回False

* **init\_server()**

Parameter Server训练中，fleet加载model\_dir中保存的模型相关参数进行parameter server的初始化

* **run\_server()**

Parameter Server训练中使用，用来启动server端服务

* **init\_worker()**

Parameter Server训练中使用，用来启动worker端服务

* **stop\_worker()**

训练结束后，停止worker

* **distributed\_optimizer(optimizer, strategy=None)**

分布式优化算法装饰器，用户可带入单机optimizer，并配置分布式训练策略，返回一个分布式的optimizer

**class DistributedTranspiler(Fleet)：**

是Fleet基类的一个子类（参考上一个类的文档第一句话），我们在代码中进行分布式配置时使用的fleet是该类的实例化：

1. 初始化一个role\_maker对象，生成本机里面的role

2. 进行optimizer的config 和 program的config

3. 如果本机是server，还能初始化server，控制server的运行

4. 如果本机是worker，创建worker进程进行训练

**主要函数方法：**

在继承并重写了父类相关函数外还定义了\_transpile()函数

* **init\_server()**

Parameter Server训练中，fleet需要执行executor以运行初始化startup program，当然，如果model\_dir中已经有保存好的模型相关参数，就加载之并用于parameter server的初始化

* **init\_worker()**

首先，worker需要等待所有的参数服务器pserver启动；第二，运行executor的初始化startup program；第三，等待所有的worker全部初始化

检查所有的服务器是否正常

创建一个Communicator()类，Communicator用于分布式训练的信息传递，是CPP类Communicator()的封装

* **run\_server()**

运行executor来执行server的主程序main program

* **stop\_worker()**

训练结束后，关闭executor，停止worker

* **distributed\_optimizer(optimizer, strategy=None)**

分布式优化算法装饰器，用户可带入单机optimizer，并配置分布式训练策略，返回一个分布式的optimizer

* **\_transpile()**

传入之前设置的config（同步异步模式/是否使用GEO……）并根据config设置main\_program和transpiler；

设置startup\_program 、init\_server、init\_worker运行的program

**输入:** config、DistributedStrategy类的对象，包含训练信息

**输出：**无

**功能：**

根据config的信息和参数，将输入的program转化为分布式的program

**class DistributeTranspilerConfig(object):**

**说明：**

存储配置的class文件，为transpiler分布式任务提供配置

**重要的参数：**

slice\_var\_up (bool)

为多个Pserver（parameter server）将tensor切片, 默认为True。

split\_method (PSDispatcher)

可使用 RoundRobin 或者 HashName。

注意: 尝试选择最佳方法来达到Pserver间负载均衡。

min\_block\_size (int)

block中分割(split)出的元素个数的最小值。

**返回：**

初始化后的DistributeTranspilerConfig实例

**返回类型：**

实例（DistributeTranspilerConfig）

**代码：**

from paddle.fluid.transpiler.ps\_dispatcher import RoundRobin

import paddle.fluid as fluid

config = fluid.DistributeTranspilerConfig()

config.slice\_var\_up = True

config.split\_method = RoundRobin

config.min\_block\_size = 81920

**class DistributeTranspiler(object):**

该类可以把fluid program转变为分布式数据并行计算的program, 有PServer和NCCL2两种模式。 在Pserver（全称：parameter server）模式下， 通过 transpile 将用于单机训练的 program 转译为可用于parameter server的分布式架构(即PServer,参数服务器)来进行训练的program。 在NCCL2模式下, 通过 transpile 将用于单机训练的 program 转译为可用于NCCL2的分布式架构来进行训练的program。在NCCL2模式下，transpiler会在 startup\_program 中附加一个 NCCL\_ID 广播算子（broadcasting operators）来实现在该集群中所有工作结点共享``NCCL\_ID`` 。 调用 transpile\_nccl2 后， 你 必须 将 trainer\_id , num\_trainers 参数提供给 Executor 来启动NCCL2分布式模式。

**参数：**

config （DistributeTranspilerConfig） DistributeTranspiler属性配置实例，定义了program转变所需要的属性, 请参考：DistributeTranspilerConfig 相关文档。

**返回：**

初始化后的DistributeTranspiler实例

**返回类型：**

实例（DistributeTranspiler）

**代码：**

x = fluid.layers.data(name='x', shape=[13], dtype='float32')

y = fluid.layers.data(name='y', shape=[1], dtype='float32')

y\_predict = fluid.layers.fc(input=x, size=1, act=None)

cost = fluid.layers.square\_error\_cost(input=y\_predict, label=y)

avg\_loss = fluid.layers.mean(cost)

sgd\_optimizer = fluid.optimizer.SGD(learning\_rate=0.001)

sgd\_optimizer.minimize(avg\_loss)

# pserver 模式下

pserver\_endpoints = "192.168.0.1:6174,192.168.0.2:6174"

trainer\_endpoints = "192.168.0.1:6174,192.168.0.2:6174"

current\_endpoint = "192.168.0.1:6174"

trainer\_id = 0

trainers = 4

role = "PSERVER"

t = fluid.DistributeTranspiler()

t.transpile(

trainer\_id, pservers=pserver\_endpoints, trainers=trainers)

if role == "PSERVER":

pserver\_program = t.get\_pserver\_program(current\_endpoint)

pserver\_startup\_program = t.get\_startup\_program(current\_endpoint,

pserver\_program)

elif role == "TRAINER":

trainer\_program = t.get\_trainer\_program()

# nccl2 模式下

trainer\_num = 2

trainer\_id = 0

config = fluid.DistributeTranspilerConfig()

config.mode = "nccl2"

trainer\_endpoints = "192.168.0.1:6174,192.168.0.2:6174"

t = fluid.DistributeTranspiler(config=config)

t.transpile(trainer\_id=trainer\_id, trainers=trainer\_endpoints, current\_endpoint="192.168.0.1:6174")

exe = fluid.ParallelExecutor(

use\_cuda=True,

loss\_name=avg\_loss.name,

num\_trainers=trainer\_num,

trainer\_id=trainer\_id

)

**class TranspilerOptimizer (DistributedOptimizer):**

**输入类DistributedOptimizer:**

DistributedOptimizer是paddle.fluid.optimizer的封装器, 用户应该将paddle.fluid. optimizer转化为DistributedOptimizer从而开始分布式训练.

DistributedOptimizer是运行分布式训练的基础,优化信息将存储在Fleet()实例中,该实例保存当前分布式训练的全局信息.

**参数:**

optimizer(Optimizer): Optimizer类的子类.

Strategy(DistributeTranspilerConfig) : DistributeTranspilerConfig的实例

**返回:** 无.

**类的初始化:**

使输入的strategy为self.\_strategy, 若输入的strategy为None则默认为sync\_strategy, 若self.\_strategy为DistributedStrategy的一个实例, 则根据输入的strategy进行参数设置.

**类的方法:**

backward(self, loss, startup\_program=None, parameter\_list=None, no\_grad\_set=None, callbacks =None):

minimize的第一部分,进行auto-diff来为当前的program附加反向传播的运算符.

**参数:**

loss (Variable): 用于进行优化的损失

startup\_program (Program): 用于对“parameter\_list”中的参数进行优化的启动程序.

parameter\_list (list): 需要更新的参数列表.

no\_grad\_set (set | None): 应忽略的变量集合.

callbacks (list | None): 为一个参数追加反向传播运算符时要运行的可调用列表.

**返回:**

list: (param, grad)对的列表, 其中grad是反向传播的输出.

apply\_gradients(self, params\_grads):

“minimize”的第二部分, 为给定的“param \_grads”对添加优化运算符.

**参数:**

params\_grads (list):要进行优化的(param，grad)对列表.

**返回:**

list:附加到当前程序的操作符列表.

**例:**

.. code-block:: python

loss = network()

optimizer = fluid.optimizer.SGD(learning\_rate=0.1)

params\_grads = optimizer.backward(loss)

# you may append operations for params\_grads here

# ...

optimizer.apply\_gradients(params\_grads)

minimize(self, loss, scopes=None, startup\_program=None, parameter\_list=None, no\_grad\_set = None):

通过更新parameter\_list来添加操作以最小化loss, 该方法把该方法将接口“backward()”和“apply\_gradients()”合二为一.

**参数:**

loss (Variable):运行优化的损失变量.

scopes (None)::TranspilerOptimizer不需要作用域参数.

startup\_program (Program):用于初始化在“parameter\_list”中的参数的启动程序.

parameter\_list (list):要更新的变量列表.

no\_grad\_set (set | None):应忽略的变量集合.

**返回:**

tuple:附加的运算符列表(optimize\_ops，params\_grads)和用于优化的变量对列表(param，grad).

**class StrategyFactory(object)**

通过静态方法将HalfAsyncStrategy, AsyncStrategy, SyncStrategy, GeoStrategy四个类写入这个大类中, 便于在代码里通过大类直接调用四种策略对应的类.

**class SyncStrategy(DistributedStrategy):**

**class AsyncStrategy(DistributedStrategy):**

**class HalfAsyncStrategy(DistributedStrategy):**

**class GeoStrategy(DistributedStrategy):**

这四个类根据四种不同的分布式策略对program, trainer\_runtime, server\_runtime, execute\_strategy, build\_strategy进行设置.

**类的方法:**

**def check\_trainer\_runtime\_config(self):**

根据当前策略对trainer\_runtime进行设置.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Sync | Async | HalfAsync | Geo |
| self.\_trainer\_runtime\_config.mode | DistributedMode.SYNC | DistributedMode.ASYNC | DistributedMode.HALF\_ASYNC | DistributedMode.GEO |

**def check\_program\_config(self):**

根据当前策略对program\_config进行设置, pass指不进行设置而跳过.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Sync | Async | HalfAsync | Geo |
| self.\_program\_config.sync\_mode | False | False | False | False |
| self.\_program\_config.runtime\_split\_send\_recv | True | True | True | True |
| self.\_program\_config.half\_async | True | pass | True | pass |
| self.\_program\_config.completely\_not\_async | True | pass | pass | pass |
| self.\_program\_config.geo\_sgd\_mode | pass | pass | pass | True |

**def check\_server\_runtime\_config(self):**

根据当前策略对server\_runtime\_config进行设置, pass指不进行设置而跳过. 四种策略在此处均跳过设置.

**def check\_execute\_strategy(self):**

根据当前策略对execute\_strategy进行设置, pass指不进行设置而跳过.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Sync | Async | HalfAsync | Geo |
| self.\_execute\_strategy.use\_thread\_barrier | True | pass | True | pass |

**def check\_build\_strategy(self):**

根据当前策略对build\_strategy进行设置, pass指不进行设置而跳过.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Sync | Async | HalfAsync | Geo |
| self.\_build\_strategy.async\_mode | True | True | True | True |

**class DistributedOptimizer(object):**

DistributedOptimizer是paddle.fluid.optimizer的包装器，用户应将实现了minimum函数的paddle.fluid.optimizer类的实例传递给DistributedOptimizer。DistributedOptimizer是分布式训练的起点。优化的具体信息将存储在Fleet实例当中，该实例保存有关当前分布式训练的全局信息。

**参数：**

optimizer(Optimizer)-被封装的优化器。Optimizer的子类。

strategy(any)-用户定义的策略信息。

**返回：**

无

**返回类型：**

无

**代码示例：**

import paddle.fluid as fluid

from paddle.fluid.transpiler.distribute\_transpiler import DistributeTranspilerConfig

from paddle.fluid.incubate.fleet.base.fleet\_base import DistributedOptimizer

optimizer = fluid.optimizer.Adam(0.01)

strategy = DistributeTranspilerConfig()

DistributedOptimizer(optimizer, strategy)

**class TranspilerOptimizer(DistributedOptimizer):**

DistributedOptimizer的子类，区别在于strategy参数的区别

**参数：**

optimizer(Optimizer)-被封装的优化器。Optimizer的子类。

strategy(DistributeTranspilerConfig)-分布式优化器额外特性

**返回：**

TranspilerOptimizer 自身实例

**返回类型：**

TranspilerOptimizer

**代码示例：**

import paddle.fluid as fluid

from paddle.fluid.transpiler.distribute\_transpiler import DistributeTranspilerConfig

from paddle.fluid.incubate.fleet.parameter\_server.distribute\_transpiler import TranspilerOptimizer

optimizer = fluid.optimizer.Adam(0.01)

strategy = DistributeTranspilerConfig()

to = TranspilerOptimizer(optimizer, strategy)

**class DistributedStrategy(object):**

提供设置和修改一系列分布式config的接口

**参数：**

无

**返回：**

DistributedStrategy自身实例

**返回类型：**

DistributedStrategy

**代码示例：**

from paddle.fluid.incubate.fleet.parameter\_server.distribute\_transpiler.distributed\_strategy import DistributedStrategy

dist\_strategy = DistributedStrategy()

**Class RoleMakerBase(object):**

**类说明：**

RoleMakerBase作为一个基类，用于给当前分布式训练中的进程安排一个role

**函数：**

**def \_\_init\_\_(self)**

创建RoleMakerBase对象，对实例的worker节点、server节点列表进行初始化。

参数：无

返回：无

返回类型：无

**def is\_worker(self)**

Parameter Server训练中使用，判断当前节点是否是Worker节点，是则返回True，否则返回False

参数：无

返回：是否为worker节点

返回类型：bool

**def is\_server(self)**

Parameter Server训练中使用，判断当前节点是否是Server节点，是则返回True，否则返回False

参数：无

返回：是否为Server节点

返回类型：bool

**def is\_first\_worker(self):**

判断当前节点是否是第一个worker节点，是则返回True，否则返回False

参数：无

返回：是否为第一个worker节点

返回类型：Bool

**def worker\_num(self):**

返回当前总共的节点数

参数：无

返回：总共的节点数

返回类型：Int

**def worker\_index(self):**

返回当前worker节点id

参数：无

返回：当前节点id

返回类型：int

**def server\_index(self):**

返回当前server节点id

参数：无

返回：当前server节点id

返回类型：int

**def get\_trainer\_endpoints (self):**

返回trainer endpoints

参数：无

返回：trainer endpoints

返回类型：self.\_server\_endpoints 实例

**def to\_string (self):**

按照特定格式返回RoleMaker实例信息

参数：无

返回：实例信息

返回类型：string

**def all\_gather(self, input):**

根据input值，返回trainers和pservers之间的所有gather

参数：

input: int 或者 float

返回：trainers和pservers之间的所有gather

返回类型：values组成的list

**def all\_reduce\_worker(self, input, output, mode="sum"):**

trainers间所有的reduce

参数：

input: list

output:list

mode:string, 有sum, min, max三种模式

返回：无

返回类型：无

**def barrier\_worker(self):**

当前trainer之间的barrier

参数: 无

返回： 无

返回类型：无

**def barrier\_all (self):**

所有trainer之间的barrier

参数：无

返回：无

返回类型：无

**Class PaddleCloudRoleMaker (RoleMakerBase):**

**类说明：**

PaddleCloudRoleMaker是一个高级封装，支持使用paddle.distributed.launch或者paddle.distributed.launch\_ps启动脚本

**函数：**

def \_\_init\_\_(self, is\_collective=False)

创建PaddleCloudRoleMaker类实例

参数：is\_collective：bool类型，是否为collective模式

返回：无

返回类型：无

**def generate\_role (self)**

在PaddleCloudRoleMaker类实例实例中，根据类属性创建对应的分布式role

参数：无

返回：无

返回类型：无

示例：

from paddle.fluid.incubate.fleet.parameter\_server.distribute\_transpiler import fleet

from paddle.fluid.incubate.fleet.base import role\_maker

role = role\_maker.PaddleCloudRoleMaker()

fleet.init(role)