机器学习参数服务器ps-lite 之(1) -- PostOffic

0x00 摘要

参数服务器是机器学习训练一种范式，是为了解决分布式机器学习问题的一个编程框架，其主要包括服务器端，客户端和调度器，与其他范式相比，参数服务器把模型参数存储和更新提升为主要组件，并且使用多种方法提高了处理能力。

本文是参数服务器系列第一篇，介绍ps-lite的总体设计和基础模块 Postoffice。

0x01 概要

1.1 参数服务器是什么

如果做一个类比，参数服务器是机器学习领域的分布式内存数据库，其作用是存储模型和更新模型。

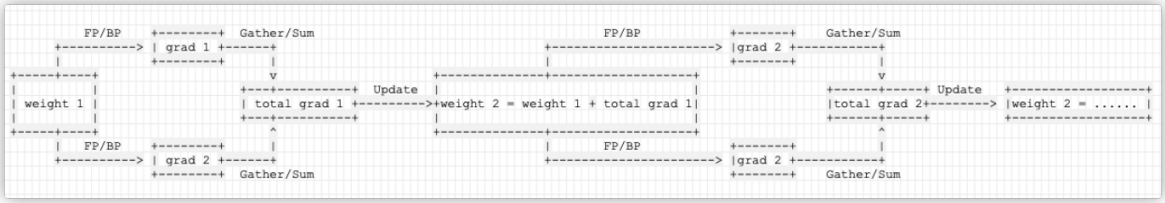
我们来看看机器学习的几个步骤，这些步骤不断循环往复。

1. 准备数据：训练进程拿到权重 weight 和数据（data + label）；
2. 前向计算：训练进程使用数据进行前向计算，得到 loss = f(weight, data & label)；
3. 反向求导：通过对 loss 反向求导，得到导数 grad = b(loss, weight, data & label)；
4. 更新权重：weight -= grad * lr；
5. 来到1，进行下一次迭代；

如果使用参数服务器训练，我们可以把如上步骤对应如下：

1. 参数下发：参数服务器服务端 将 weight 发给 每个worker（或者worker自行拉取），worker就是参数服务器Client端;
2. 并行计算：每个worker 分别完成自己的计算（包括前向计算和反向求导）；
3. grad 收集：参数服务器服务端 从每个 Worker 处得到 grad，完成归并（或者worker自行推送）；
4. 更新权重：参数服务器服务端 自行将 grad 应用到 weight 上;
5. 来到1，进行下一次迭代；

具体如下图：



因此我们可以推导出参数服务器之中各个模块的作用：

- 服务器端（Server ）：存放机器学习模型参数，接收客户端发送的梯度，完成归并，对本地模型参数进行更新。
- 客户端（Client 或者 Worker）：
  - 从服务器端获取当前最新的参数；
  - 使用训练数据和从最新参数计算得到预测值，根据损失函数来计算关于训练参数的梯度；
  - 将梯度发送给服务器端；
- 调度器（Scheduler）：管理服务器/客户端节点，完成节点之间数据同步，节点添加/删除等功能。

1.2 历史溯源

参数服务器属于机器学习训练的一个范式，具体可以分为三代（目前各大公司应该有自己内部最新实现，可以算为第四代）。

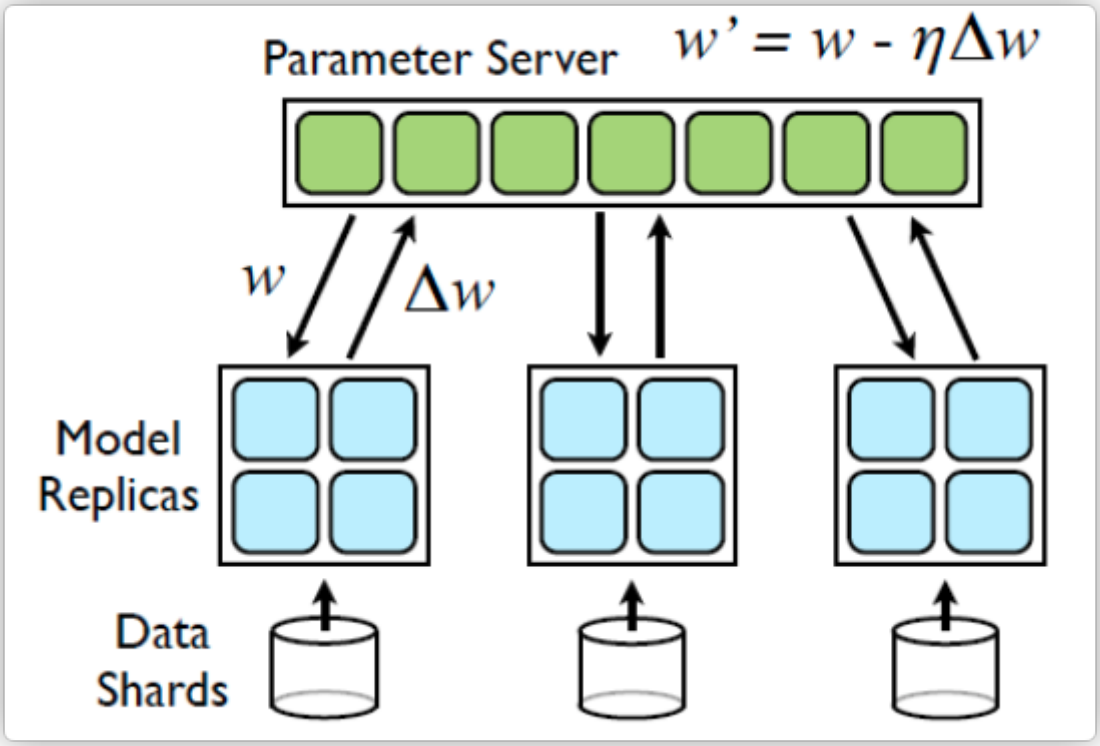
在参数服务器之前，大部分分布式机器学习算法是通过定期同步来实现的，比如集合通信的all-reduce，或者 map-reduce类系统的reduce步骤。但是定期同步有两个问题：

- 同步时期只能做同步，不能训练。
- straggler问题：由于一些软硬件的原因，节点的计算能力往往不尽相同。对于迭代问题来说，每一轮结束时算得快的节点都需等待算得慢的节点算完，再进行下一轮迭代。这种等待在节点数增多时将变得尤为明显，从而拖慢整体的性能。

因此，当async sgd出现之后，就有人提出了参数服务器。

参数服务器的概念最早来自于Alex Smola于2010年提出的并行LDA的框架。它通过采用一个分布式的Memcached作为存放共享参数的存储，这样就提供了有效的机制用于分布式系统中不同的Worker之间同步模型参数，而每个Worker只需要保存他计算时所以来的一小部分参数即可，也避免了所有进程在一个时间点上都停下来同步。但是独立的kv对带来了很大的通信开销，而且服务端端难以编程。

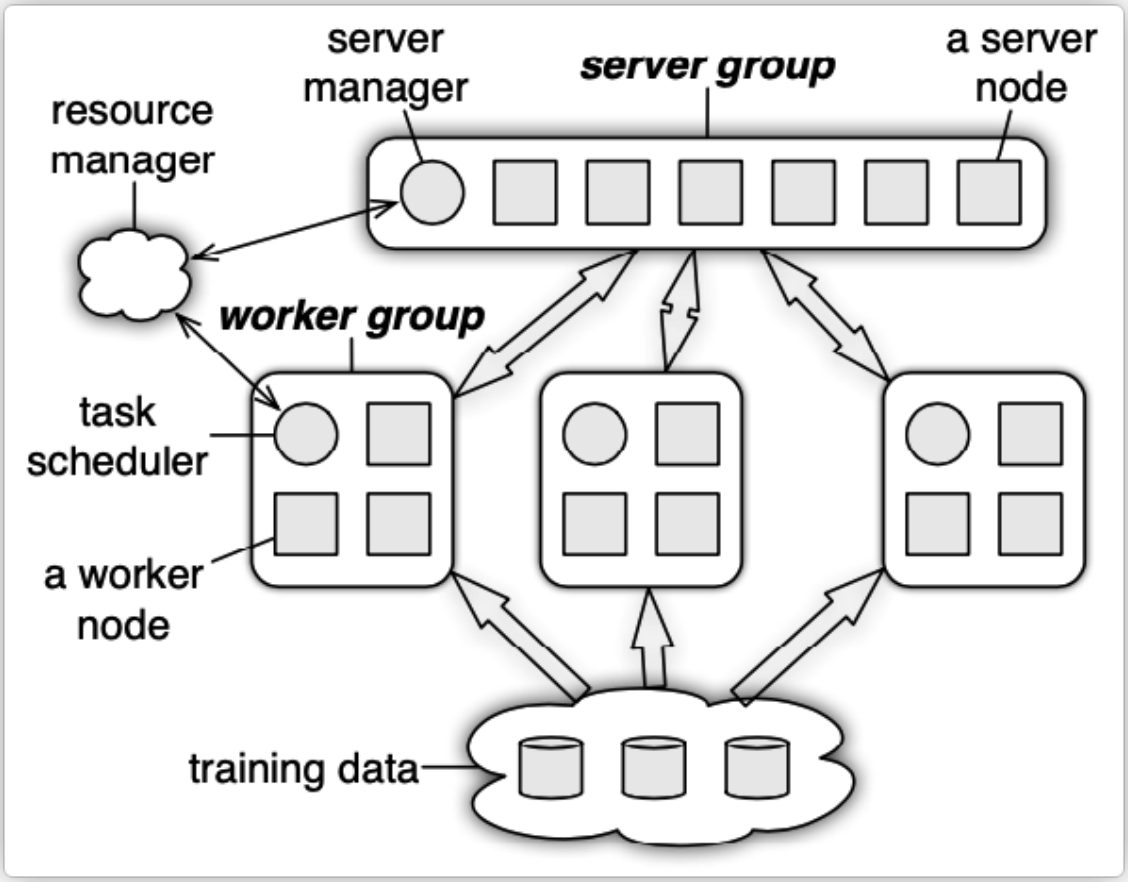
第二代由Google的Jeff Dean进一步提出了第一代Google大脑的解决方案：DistBelief。DistBelief将巨大的深度学习模型分布存储在全局的参数服务器中，计算节点通过参数服务器进行信息传递，很好地解决了SGD和L-BFGS算法的分布式训练问题。



再后来就是李沐所在的DMLC组所设计的参数服务器。根据论文中所写，该parameter server属于第三代参数服务器，就是提供了更加通用的设计。架构上包括一个Server Group和若干个Worker Group。

1.3 论文架构

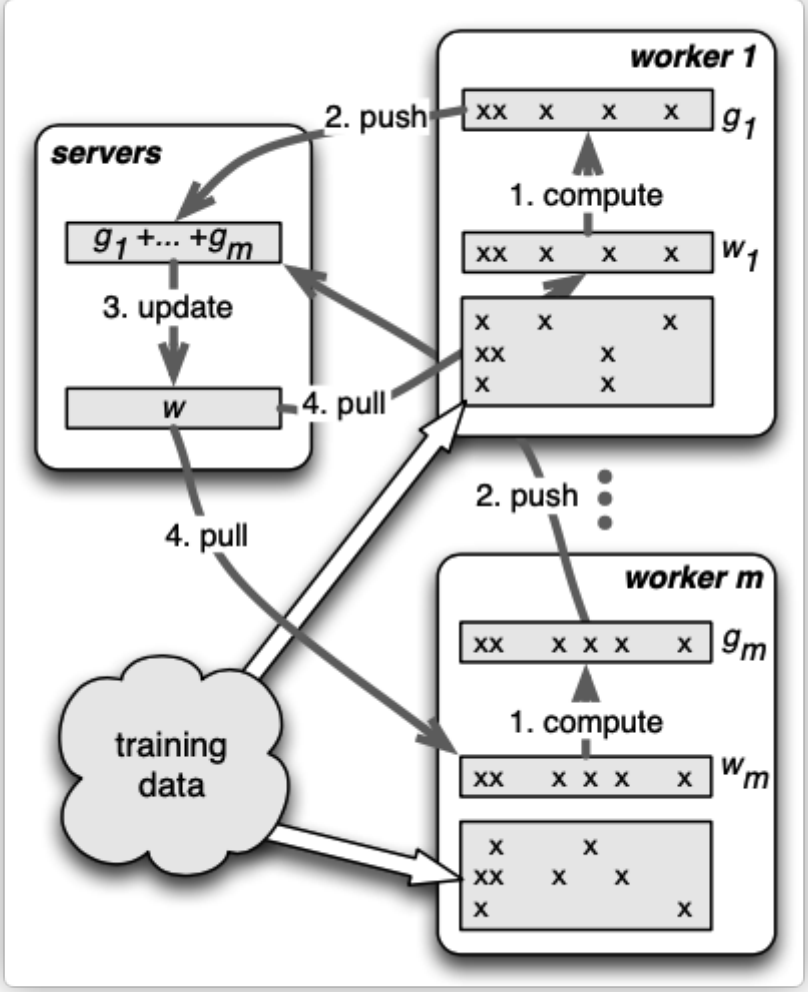
我们首先用沐神论文中的图来看看系统架构。



解释一下图中整体架构中每个模块：

- resource manager：资源分配及管理器。参数服务器使用业界现有的资源管理系统，比如yarn，k8s。
- training data：几十上百亿的训练数据一般存储在分布式文件系统上（比如HDFS），resource manager会均匀的分配到每个worker上。
- 参数服务器的节点被划分到一个 server group 和多个 worker group。
- server group：一次训练任务中申请的servers，用于模型参数的更新和pull应答。
    - server group 中的每个 server 只负责自己分到的部分全局共享参数（server 共同维持一个全局共享参数），一般优化器在此实现。
    - server 之间相互通信以便进行参数的备份/迁移。
    - server group 有一个 server manager node，负责维护 server 元数据的一致性，例如节点状态，参数的分配情况。一般不会有什么逻辑，只有当有server node加入或退出的时候，为了维持一致性哈希而做一些调整。
- worker group：一次训练任务中申请的workers，用于前向过程和梯度计算。
    - 每个 worker group 运行一个计算任务，worker group 中的 每个worker 使用部分数据进行训练。
    - 分成多个group，这样就可以支持多任务的并行计算。
    - 每个 worker group 有一个 task scheduler，负责向 worker 分配任务，并监控他们的运行情况，当有 worker 进入或者退出时，task scheduler 重新分配未完成的任务。
    - worker 之间没有通信，只和对应的 server 通信进行参数更新。

在分布式计算梯度时，系统的数据流如下：



图中每个步骤的作用为：

1. worker 节点 基于该 batch 内的样本计算模型权重的梯度；
2. worker将梯度以key-value的形式推送给server；
3. server按指定的优化器对模型权重进行梯度更新；
4. worker从server中拉取最新的模型权重；

上面两个图的依据是其原始代码。ps-lite 是后来的精简版代码，所以有些功能在 ps-lite 之中没有提供。

1.4 ps-lite发展历程

从网上找到了一些 ps-lite发展历程，可以看到其演进的思路。

第一代是parameter，针对特定算法（如逻辑回归和LDA）进行了设计和优化，以满足规模庞大的工业机器学习任务（数百亿个示例和10-100TB数据大小的功能）。

后来尝试为机器学习算法构建一个开源通用框架。 该项目位于dmlc / parameter_server。

鉴于其他项目的需求不断增长，创建了ps-lite，它提供了一个干净的数据通信API和一个轻量级的实现。该实现基于dmlc / parameter_server，但为不同的项目重构了作业启动器，文件IO和机器学习算法代码，如dmlc-core和wormhole

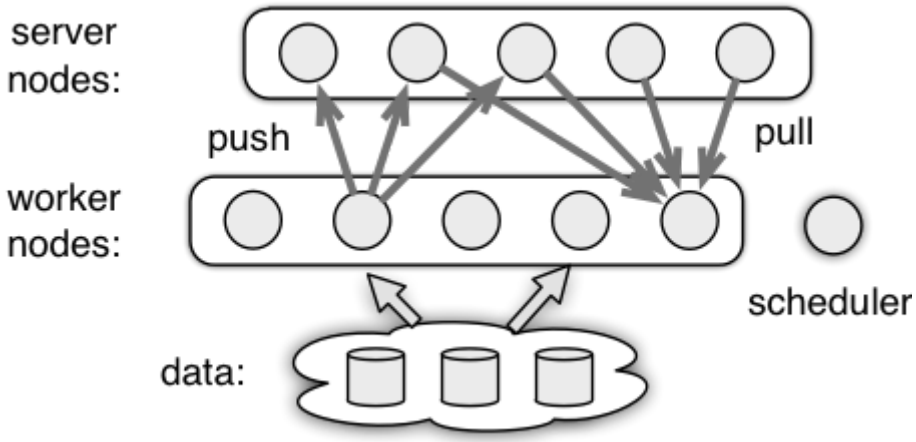
根据在开发dmlc / mxnet期间学到的经验，从v1进一步重构了API和实现。 主要变化包括：

- 库依赖性较少;
- 更灵活的用户定义回调，便于其他语言绑定;
- 让用户（如mxnet的依赖引擎）管理数据一致性;

1.5 ps-lite 系统总体

ps-lite 其实是Paramter Server的实现的一个框架，其中参数处理具体相关策略需用户自己实现。

Parameter Server包含三种角色：Worker，Server，Scheduler。具体关系如下图：



具体角色功能为：

- **worker（工作节点）：** 若干个，执行data pipeline、前向和梯度计算，以key-value的形式将模型权重梯度push到server节点以及从server节点拉取模型最新权重；
- **server（服务节点）：** 若干个，负责对worker的push和pull请求做response，存储，维护和更新模型权重以供各个worker使用（每个server仅维护模型的一部分）；
- **scheduler（控制节点）：** 系统内只有一个。负责所有节点的心跳监测、节点id分配和worker&server间的通信建立，它还可用于将控制信号发送到其他节点并收集其进度。

其中引入scheduler的好处如下：

- 引入一个 scheduler 模块，则会形成一个比较经典的三角色分布式系统架构；worker 和 server 的角色和职责不变，而 scheduler 模块则有比较多的选择：
  - 只承担和下层资源调度系统般若（类似 yarn、mesos）的交互；
  - 额外增加对 worker、server 心跳监控、流程控制的功能；
- 引入 scheduler 模块的另一个好处是给实现模型并行留出了空间；
- scheduler 模块不仅有利于实现模型并行训练范式，还有其他好处：比如通过针对特定模型参数相关性的理解，对参数训练过程进行细粒度的调度，可以进一步加快模型收敛速度，甚至有机会提升模型指标。

熟悉分布式系统的同学可能会担心 scheduler 模块的单点问题，这个通过 raft、zab 等 paxos 协议可以得到比较好的解决。

1.6 基础模块

ps-lite系统中的一些基础模块如下：

- **Environment**：一个单例模式的环境变量类，它通过一个 `std::unordered_map<std::string, std::string> kvs` 维护了一组 kvs 借以保存所有环境变量名以及值；
- **PostOffice**：一个单例模式的全局管理类，一个 node 在生命期内具有一个PostOffice，依赖它的类成员对Node进行管理；
- **Van**：通信模块，负责与其他节点的网络通信和Message的实际收发工作。PostOffice持有一个Van成员；
- **SimpleApp**：KVServer和KVWorker的父类，它提供了简单的Request, Wait, Response，Process功能；KVServer和KVWorker分别根据自己的使命重写了这些功能；

- **Customer**：每个SimpleApp对象持有一个Customer类的成员，且Customer需要在PostOffice进行注册，该类主要负责：
  - 跟踪由SimpleApp发送出去的消息的回复情况；
  - 维护一个Node的消息队列，为Node接收消息；
- **Node**：信息类，存储了本节点的对应信息，每个 Node 可以使用 hostname + port 来唯一标识。

0x02 系统启动

2.1 如何启动

从源码中的例子可以看出，使用ps-lite 提供的脚本 local.sh 可以启动整个系统，这里 test_connection 为编译好的可执行程序。

```
./local.sh 2 3 ./test_connection
```

2.2 启动脚本

具体 local.sh 代码如下。注意，在shell脚本中，有三个shift，这就让脚本中始终使用$1。

针对我们的例子，脚本参数对应了就是

- DMLC_NUM_SERVER 为 2；
- DMLC_NUM_WORKER 为 3；
- bin 是 ./test_connection；

可以从脚本中看到，本脚本做了两件事：

- 每次执行应用程序之前，都会依据本次执行的角色来对环境变量进行各种设定，除了DMLC_ROLE设置得不同外，其他变量在每个节点上都相同。
- 在本地运行多个不同角色。这样 ps-lite 就用多个不同的进程（程序）共同合作完成工作。
  - 首先启动Scheduler节点。这是要固定好Server和Worker数量，Scheduler节点管理所有节点的地址。
  - 启动Worker或Server节点。每个节点要知道Scheduler节点的IP、port。启动时连接Scheduler节点，绑定本地端口，并向Scheduler节点注册自己信息（报告自己的IP，port）。
  - Scheduler等待所有Worker节点都注册后，给其分配id，并把节点信息传送出去（例如Worker节点要知道Server节点IP和端口，Server节点要知道Worker节点的IP和端口）。此时Scheduler节点已经准备好。
  - Worker或Server接收到Scheduler传送的信息后，建立对应节点的连接。此时Worker或Server已经准备好，会正式启动。

具体如下：

```
#!/bin/bash
# set -x
if [ $# -lt 3 ]; then
    echo "usage: $0 num_servers num_workers bin [args..]"
    exit -1;
fi

# 对环境变量进行各种配置，此后不同节点都会从这些环境变量中获取信息
export DMLC_NUM_SERVER=$1
shift
export DMLC_NUM_WORKER=$1
```

```
shift
bin=$1
shift
arg="$@"

# start the scheduler
export DMLC_PS_ROOT_URI='127.0.0.1'
export DMLC_PS_ROOT_PORT=8000
export DMLC_ROLE='scheduler'
${bin} ${arg} &


# start servers
export DMLC_ROLE='server'
for ((i=0; i<${DMLC_NUM_SERVER}; ++i)); do
    export HEAPPROFILE=./S${i}
    ${bin} ${arg} &
done

# start workers
export DMLC_ROLE='worker'
for ((i=0; i<${DMLC_NUM_WORKER}; ++i)); do
    export HEAPPROFILE=./W${i}
    ${bin} ${arg} &
done

wait
```

2.3 示例程序

我们依然使用官方例子看看。

ps-lite 使用的是 C++语言，其中 worker, server, scheduler 都使用同一套代码。这会让习惯于Java，python的同学非常不适应，大家需要适应一个阶段。

针对这个示例程序，起初会让人疑惑，为什么每次程序运行，代码中都会启动 scheduler，worker，server？其实，从下面注释就能看出来，具体执行是依据环境变量来决定。如果环境变量设置了本次角色是 server，则不会启动 scheduler 和 worker。

```
#include <cmath>
#include "ps/ps.h"

using namespace ps;

void StartServer() {
  if (!IsServer()) {
    return;
  }
  auto server = new KVServer<float>(0);
  server->set_request_handle(KVServerDefaultHandle<float>()); //注册functor
  RegisterExitCallback([server](){ delete server; });
}

void RunWorker() {
  if (!IsWorker()) return;
  KVWorker<float> kv(0, 0);
```

```
  // init
  int num = 10000;
  std::vector<Key> keys(num);
  std::vector<float> vals(num);

  int rank = MyRank();
  srand(rank + 7);
  for (int i = 0; i < num; ++i) {
    keys[i] = kMaxKey / num * i + rank;
    vals[i] = (rand() % 1000);
  }

  // push
  int repeat = 50;
  std::vector<int> ts;
  for (int i = 0; i < repeat; ++i) {
    ts.push_back(kv.Push(keys, vals)); //kv.Push()返回的是该请求的timestamp

    // to avoid too frequency push, which leads huge memory usage
    if (i > 10) kv.Wait(ts[ts.size()-10]);
  }
  for (int t : ts) kv.Wait(t);

  // pull
  std::vector<float> rets;
  kv.Wait(kv.Pull(keys, &rets));

  // pushpull
  std::vector<float> outs;
  for (int i = 0; i < repeat; ++i) {
    // PushPull on the same keys should be called serially
    kv.Wait(kv.PushPull(keys, vals, &outs));
  }

  float res = 0;
  float res2 = 0;
  for (int i = 0; i < num; ++i) {
    res += std::fabs(rets[i] - vals[i] * repeat);
    res2 += std::fabs(outs[i] - vals[i] * 2 * repeat);
  }
  CHECK_LT(res / repeat, 1e-5);
  CHECK_LT(res2 / (2 * repeat), 1e-5);
  LL << "error: " << res / repeat << ", " << res2 / (2 * repeat);
}

int main(int argc, char *argv[]) {
  // start system
  Start(0); // Postoffice::start()，每个node都会调用到这里，但是在 Start 函数之中，会依据本次设定的角色来不同处理，只有角色为 scheduler 才会启动 Scheduler。
  // setup server nodes
  StartServer(); // Server会在其中做有效执行，其他节点不会有效执行。
  // run worker nodes
  RunWorker(); // Worker 会在其中做有效执行，其他节点不会有效执行。
  // stop system
  Finalize(0, true); //结束。每个节点都需要执行这个函数。
  return 0;
}
```

其中KVServerDefaultHandle是functor，用与处理server收到的来自worker的请求，具体如下：

```
/**
 * \brief an example handle adding pushed kv into store
 */
template <typename Val>
struct KVServerDefaultHandle { //functor，用与处理server收到的来自worker的请求
    // req_meta 是存储该请求的一些元信息，比如请求来自于哪个节点，发送给哪个节点等等
    // req_data 是发送过来的数据
    // server 是指向当前server对象的指针
  void operator()(
      const KVMeta& req_meta, const KVPairs<Val>& req_data, KVServer<Val>*
server) {
    size_t n = req_data.keys.size();
    KVPairs<Val> res;
    if (!req_meta.pull) { //收到的是pull请求
      CHECK_EQ(n, req_data.vals.size());
    } else { //收到的是push请求
      res.keys = req_data.keys; res.vals.resize(n);
    }
    for (size_t i = 0; i < n; ++i) {
      Key key = req_data.keys[i];
      if (req_meta.push) { //push请求
        store[key] += req_data.vals[i]; //此处的操作是将相同key的value相加
      }
      if (req_meta.pull) {  //pull请求
        res.vals[i] = store[key];
      }
    }
    server->Response(req_meta, res);
  }
  std::unordered_map<Key, Val> store;
};
```

0x03 Postoffice

Postoffice 是一个单例模式的全局管理类，其维护了系统的一个全局信息，具有如下特点：

- 三种Node角色都依赖 Postoffice 进行管理，每一个 node 在生命期内具有一个单例 PostOffice。
- 如我们之前所说，ps-lite的特点是 worker, server, scheduler 都使用同一套代码，Postoffice也是如此，所以我们最好分开描述。
- 在 Scheduler侧，顾名思义，Postoffice 是邮局，可以认为是一个地址簿，一个调控中心，其记录了系统（由scheduler，server， worker 集体构成的这个系统）中所有节点的信息。具体功能如下：
  - 维护了一个Van对象，负责整个网络的拉起、通信、命令管理如增加节点、移除节点、恢复节点等等；
  - 负责整个集群基本信息的管理，比如worker、server数的获取，管理所有节点的地址，server 端 feature分布的获取，worker/server Rank与node id的互转，节点角色身份等等；
  - 负责 Barrier 功能；
- 在 Server / Worker 端，负责：
  - 配置当前node的一些信息，例如当前node是哪种类型(server，worker)，nodeid是啥，以及worker/server 的rank 到 node id的转换。
  - 路由功能：负责 key 与 server 的对应关系。
  - Barrier 功能；

请注意：这些代码都是在 Postoffice 类内，没有按照角色分开成多个模块。

3.1 定义

类 UML 图如下：

下面我们只给出关键变量和成员函数说明，因为每个节点都包含一个 PostOffice，所以 PostOffice 的数据结构中包括了各种节点所需要的变量，会显得比较繁杂。

主要变量作用如下：

- van_ ： 底层通讯对象；
- customers_ ： 本节点目前有哪些 customer；
- node_ids_ ： node id 映射表；
- server_key_ranges_ ： Server key 区间范围对象
- is_worker, *is_server*, is_scheduler_ ： 标注了本节点类型；
- heartbeats_ ： 节点心跳对象；
- barrier_done_ ： Barrier 同步变量；

主要函数作用如下：

- InitEnvironment ： 初始化环境变量，创建 van 对象；
- Start ： 建立通信初始化；
- Finalize ： 节点阻塞退出；
- Manage ： 退出 barrier 阻塞状态；
- Barrier ： 进入 barrier 阻塞状态；
- UpdateHeartbeat ：
- GetDeadNodes ： 根据 heartbeats_ 获取已经 dead 的节点；

具体如下：

```
class Postoffice {
  /**
   * \brief start the system
   *
   * This function will block until every nodes are started.
   * \param argv0 the program name, used for logging.
   * \param do_barrier whether to block until every nodes are started.
   */
  void Start(int customer_id, const char* argv0, const bool do_barrier);
  /**
   * \brief terminate the system
   *
   * All nodes should call this function before existing.
   * \param do_barrier whether to do block until every node is finalized,
default true.
   */
  void Finalize(const int customer_id, const bool do_barrier = true);
  /**
   * \brief barrier
   * \param node_id the barrier group id
   */
  void Barrier(int customer_id, int node_group);
  /**
   * \brief process a control message, called by van
   * \param the received message
   */
  void Manage(const Message& recv);
```

```
  /**
   * \brief update the heartbeat record map
   * \param node_id the \ref Node id
   * \param t the last received heartbeat time
   */
  void UpdateHeartbeat(int node_id, time_t t) {
    std::lock_guard<std::mutex> lk(heartbeat_mu_);
    heartbeats_[node_id] = t;
  }
  /**
   * \brief get node ids that haven't reported heartbeats for over t seconds
   * \param t timeout in sec
   */
  std::vector<int> GetDeadNodes(int t = 60);
 private:
 void InitEnvironment();
  Van* van_;
  mutable std::mutex mu_;
  // app_id -> (customer_id -> customer pointer)
  std::unordered_map<int, std::unordered_map<int, Customer*>> customers_;
  std::unordered_map<int, std::vector<int>> node_ids_;
  std::mutex server_key_ranges_mu_;
  std::vector<Range> server_key_ranges_;
  bool is_worker_, is_server_, is_scheduler_;
  int num_servers_, num_workers_;
  std::unordered_map<int, std::unordered_map<int, bool> > barrier_done_;
  int verbose_;
  std::mutex barrier_mu_;
  std::condition_variable barrier_cond_;
  std::mutex heartbeat_mu_;
  std::mutex start_mu_;
  int init_stage_ = 0;
  std::unordered_map<int, time_t> heartbeats_;
  Callback exit_callback_;
  /** \brief Holding a shared_ptr to prevent it from being destructed too early
*/
  std::shared_ptr<Environment> env_ref_;
  time_t start_time_;
  DISALLOW_COPY_AND_ASSIGN(Postoffice);
};
```

3.2 ID 映射功能

首先我们介绍下 node id 映射功能，就是如何在逻辑节点和物理节点之间做映射，如何把物理节点划分成各个逻辑组，如何用简便的方法做到给组内物理节点统一发消息。

- 1，2，4分别标识Scheduler, ServerGroup, WorkerGroup。
- SingleWorker：rank * 2 + 9；SingleServer：rank * 2 + 8。
- 任意一组节点都可以用单个id标识，等于所有id之和。

3.2.1 概念

- Rank 是一个逻辑概念，是每一个节点（scheduler，work，server）内部的唯一逻辑标示。
- Node id 是物理节点的唯一标识，可以和一个 host + port 的二元组唯一对应。
- Node Group 是一个逻辑概念，每一个 group 可以包含多个 node id。ps-lite 一共有三组 group ： scheduler 组，server 组，worker 组。

- Node group id 是 是节点组的唯一标示。
  - ps-lite 使用 1，2，4 这三个数字分别标识 Scheduler，ServerGroup，WorkerGroup。每一个数字都代表着一组节点，等于所有该类型节点 id 之和。比如 2 就代表server 组，就是所有 server node 的组合。
  - 为什么选择这三个数字？因为在二进制下这三个数值分别是 "001, 010, 100"，这样如果想给多个 group 发消息，直接把 几个 node group id 做 或操作 就行。
  - 即 1-7 内任意一个数字都代表的是Scheduler / ServerGroup / WorkerGroup的某一种组合。
    - 如果想把某一个请求发送给所有的 worker node，把请求目标节点 id 设置为 4 即可。
    - 假设某一个 worker 希望向所有的 server 节点 和 scheduler 节点同时发送请求，则只要把请求目标节点的 id 设置为 3 即可，因为 3 = 2 + 1 = kServerGroup + kScheduler。
    - 如果想给所有节点发送消息，则设置为 7 即可。

3.2.2 逻辑组的实现

三个逻辑组的定义如下：

```
/** \brief node ID for the scheduler */
static const int kScheduler = 1;
/**
 * \brief the server node group ID
 *
 * group id can be combined:
 * - kServerGroup + kScheduler means all server nodes and the scheuduler
 * - kServerGroup + kWorkerGroup means all server and worker nodes
 */
static const int kServerGroup = 2;
/** \brief the worker node group ID */
static const int kWorkerGroup = 4;
```

3.2.3 Rank vs node id

node id 是物理节点的唯一标示，rank 是每一个逻辑概念（scheduler，work，server）内部的唯一标示。这两个标示由一个算法来确定。

如下面代码所示，如果配置了 3 个worker，则 worker 的 rank 从 0 ~ 2，那么这几个 worker 实际对应的 物理 node ID 就会使用 WorkerRankToID 来计算出来。

```
    for (int i = 0; i < num_workers_; ++i) {
      int id = WorkerRankToID(i);
      for (int g : {id, kWorkerGroup, kWorkerGroup + kServerGroup,
                    kWorkerGroup + kScheduler,
                    kWorkerGroup + kServerGroup + kScheduler}) {
        node_ids_[g].push_back(id);
      }
    }
```

具体计算规则如下：

```
  /**
   * \brief convert from a worker rank into a node id
   * \param rank the worker rank
   */
  static inline int WorkerRankToID(int rank) {
    return rank * 2 + 9;
```

```
  }
  /**
   * \brief convert from a server rank into a node id
   * \param rank the server rank
   */
  static inline int ServerRankToID(int rank) {
    return rank * 2 + 8;
  }
  /**
   * \brief convert from a node id into a server or worker rank
   * \param id the node id
   */
  static inline int IDtoRank(int id) {
#ifdef _MSC_VER
#undef max
#endif
    return std::max((id - 8) / 2, 0);
  }
```

这样我们可以知道，1-7 的id表示的是node group，单个节点的id 就从 8 开始。

而且这个算法保证server id为偶数，node id为奇数。

- SingleWorker：rank * 2 + 9；
- SingleServer：rank * 2 + 8；

3.2.4 Group vs node

因为有时请求要发送给多个节点，所以ps-lite用了一个 map 来存储**每个 node group / single node 对应的实际的node节点集合**，即 确定每个id值对应的节点id集。

```
std::unordered_map<int, std::vector<int>> node_ids_
```

如何使用这个node_ids_？我们还是需要看之前的代码：

```
    for (int i = 0; i < num_workers_; ++i) {
      int id = WorkerRankToID(i);
      for (int g : {id, kWorkerGroup, kWorkerGroup + kServerGroup,
                    kWorkerGroup + kScheduler,
                    kWorkerGroup + kServerGroup + kScheduler}) {
        node_ids_[g].push_back(id);
      }
    }
```

我们回忆一下之前的节点信息：

- 1 ~ 7 的 id 表示的是 node group；
- 后续的 id（8，9，10，11 ...）表示单个的 node。其中双数 8，10，12... 表示 worker 0, worker 1, worker 2，... 即（2n + 8），9，11，13，...，表示 server 0, server 1，server 2，...，即（2n + 9）；

所以，为了实现 “设置 1-7 内任意一个数字 可以发送给其对应的 所有node” 这个功能，对于每一个新节点，需要将其对应多个id（node，node group）上，这些id组就是本节点可以与之通讯的节点。例如对于 worker 2 来说，其 node id 是 2 * 2 + 8 = 12，所以需要将它与

- 12（本身）
- 4（kWorkerGroup）li

- 4+1（kWorkerGroup + kScheduler）
- 4+2（kWorkerGroup + kServerGroup）
- 4+1+2,（kWorkerGroup + kServerGroup + kScheduler ）

这 5 个id 相对应，即需要在 node_ids_ 这个映射表中对应的 4, 4 + 1, 4 + 2, 4 +1 + 2, 12 这五个 item 之中添加。就是上面代码中的内部 for 循环条件。即，node_ids_ [4], node_ids_ [5],node_ids_ [6],node_ids_ [7] ,node_ids_ [12] 之中，都需要把 12 添加到 vector 最后。

3.3 参数表示

workers 跟 servers 之间通过 **push** 跟 **pull** 来通信。worker 通过 push 将计算好的梯度发送到 server，然后通过 pull 从server更新参数。

3.3.1 KV格式

parameter server 中，参数都是可以被表示成（key, value）的集合，比如一个最小化损失函数的问题，key就是feature ID，而value就是它的权值。对于稀疏参数来说，value不存在的key，就可以认为value是0。

把参数表示成 k-v， 形式更自然，易于理解和编程实现。

3.3.2 key-values

分布式算法有两个额外成本：数据通信成本，负载均衡不理想和机器性能差异导致的同步成本。

对于高维机器学习训练来说，因为高频特征更新极为频繁，所会导致网络压力极大。如果每一个参数都设一个key并且按key更新，那么会使得通信变得更加频繁低效，为了抹平这个问题，就需要有折衷和平衡，即，
利用机器学习算法的特性，给每个key对应的value赋予一个向量或者矩阵，这样就可以一次性传递多个参数，权衡了融合与同步的成本。

做这样的操作的前提是假设参数是有顺序的。缺点是在对于稀疏模型来说，总会在向量或者矩阵里会有参数为0，这在单个参数状态下是不用存的，所以，造成了数据的冗余。

但这样做有两点好处：

- 降低网络通信
- 使得向量层面的操作变得可行，从而很多线性库的优化特性可以利用的上，比如BLAS、LAPACK、ATLAS等。

3.3.3 Range 操作

为了提高计算性能和带宽效率，参数服务器也会采用批次更新的办法，来减轻高频 key 的压力。比如把minibatch之中高频key合并成一个minibatch进行更新。

ps-lite 允许用户使用 **Range Push** 跟 **Range Pull** 操作。

3.4 路由功能（keyslice）

路由功能指的就是：Worker 在做 Push/Pull 时候，如何知道把消息发送给哪些 Servers。

我们知道，ps-lite 是多 Server 架构，一个很重要的问题是如何分布多个参数。比如给定一个参数的键，如何确定其存储在哪一台 Server 上。所以必然有一个路由逻辑用来确立 key与server的对应关系。

PS Lite 将路由逻辑放置在 Worker 端，采用范围划分的策略，即每一个 Server 有自己固定负责的键的范围。这个范围是在 Worker 启动的时候确定的。细节如下：

- 根据编译 PS Lite 时是否设定的宏 USE_KEY32 来决定参数的键的数据类型，要么是 32 位无符号整数，要么是 64 位的。
- 根据键的数据类型，确定其值域的上界。例如 uint32_t 的上界是 4294967295。
- 根据键域的上界和启动时获取的 Server 数量（即环境变量 DMLC_NUM_SERVER 的值）来划分范围。
- 每个server维护的key范围按 uint32_t / uint64_t 从小到大等距分区间。给定上界 MAX 和 Server 数量 N，第 i 个 Server 负责的范围是 `[MAX/N*i, MAX/N*(i+1))`。
- 对key的hash值构造有一定的要求以避免server间的key倾斜（如32位、16位、8位、4位、2位高低位对调）。
- Worker push和pull的key按升序排列进行slice以实现zero copy。

需要注意的是，在不能刚好整除的情况下，键域上界的一小段被丢弃了。

具体实现如下：

首先，ps-lite的key只支持int类型。

```
#if USE_KEY32
/*! \brief Use unsigned 32-bit int as the key type */
using Key = uint32_t;
#else
/*! \brief Use unsigned 64-bit int as the key type */
using Key = uint64_t;
#endif
/*! \brief The maximal allowed key value */
static const Key kMaxKey = std::numeric_limits<Key>::max();
```

其次，将int范围均分即可

```
const std::vector<Range>& Postoffice::GetServerKeyRanges() {
  if (server_key_ranges_.empty()) {
    for (int i = 0; i < num_servers_; ++i) {
      server_key_ranges_.push_back(Range(
          kMaxKey / num_servers_ * i,
          kMaxKey / num_servers_ * (i+1)));
    }
  }
  return server_key_ranges_;
}
```

3.5 初始化环境

从之前分析中我们可以知道，ps-lite 是通过环境变量来控制具体节点。

具体某个节点属于哪一种取决于启动节点之前设置了哪些环境变量以及其数值。

环境变量包括：节点角色，worker&server个数、ip、port等。

InitEnvironment 函数就是创建了 Van，得到了 worker 和 server 的数量，得到了本节点的类型。

```
void Postoffice::InitEnvironment() {
  const char* val = NULL;
  std::string van_type = GetEnv("DMLC_PS_VAN_TYPE", "zmq");
  van_ = Van::Create(van_type);
  val = CHECK_NOTNULL(Environment::Get()->find("DMLC_NUM_WORKER"));
  num_workers_ = atoi(val);
  val =  CHECK_NOTNULL(Environment::Get()->find("DMLC_NUM_SERVER"));
  num_servers_ = atoi(val);
  val = CHECK_NOTNULL(Environment::Get()->find("DMLC_ROLE"));
  std::string role(val);
  is_worker_ = role == "worker";
  is_server_ = role == "server";
  is_scheduler_ = role == "scheduler";
  verbose_ = GetEnv("PS_VERBOSE", 0);
}
```

3.6 启动

主要就是：

- 调用 InitEnvironment() 来初始化环境，创建 VAN 对象；
- node_ids_初始化。根据worker和server节点个数，确定每个id值对应的节点id集。具体逻辑我们前面有分析。
- 启动 van，这里会进行各种交互（有一个 ADD_NODE 同步等待，与后面的 barrier 等待不同）；
- 如果是第一次调用PostOffice::Start，初始化start_time_成员；
- 如果设置了需要 barrier，则调用 Barrier 来进行 等待/处理 最终系统统一启动。即 所有Node准备并向Scheduler发送要求同步的Message，进行第一次同步；

具体代码如下：

```
void Postoffice::Start(int customer_id, const char* argv0, const bool
do_barrier) {
  start_mu_.lock();
  if (init_stage_ == 0) {
    InitEnvironment();

    // init node info.
    // 对于所有的worker，进行node设置
    for (int i = 0; i < num_workers_; ++i) {
      int id = WorkerRankToID(i);
      for (int g : {id, kWorkerGroup, kWorkerGroup + kServerGroup,
                    kWorkerGroup + kScheduler,
                    kWorkerGroup + kServerGroup + kScheduler}) {
        node_ids_[g].push_back(id);
      }
    }
        // 对于所有的server，进行node设置
    for (int i = 0; i < num_servers_; ++i) {
      int id = ServerRankToID(i);
      for (int g : {id, kServerGroup, kWorkerGroup + kServerGroup,
                    kServerGroup + kScheduler,
                    kWorkerGroup + kServerGroup + kScheduler}) {
        node_ids_[g].push_back(id);
      }
```

```
    }
        // 设置scheduler的node
    for (int g : {kScheduler, kScheduler + kServerGroup + kWorkerGroup,
                  kScheduler + kWorkerGroup, kScheduler + kServerGroup}) {
      node_ids_[g].push_back(kScheduler);
    }
    init_stage_++;
  }
  start_mu_.unlock();

  // start van
  van_->Start(customer_id);

  start_mu_.lock();
  if (init_stage_ == 1) {
    // record start time
    start_time_ = time(NULL);
    init_stage_++;
  }
  start_mu_.unlock();
  // do a barrier here
  if (do_barrier) Barrier(customer_id, kWorkerGroup + kServerGroup +
kScheduler);
}
```

3.7 Barrier

3.7.1 同步

总的来讲，schedular节点通过计数的方式实现各个节点的同步。具体来说就是：

- 每个节点在自己指定的命令运行完后会向schedular节点发送一个Control::BARRIER命令的请求并自己阻塞直到收到schedular对应的返回后才解除阻塞；
- schedular节点收到请求后则会在本地计数，看收到的请求数是否和barrier_group的数量是否相等，相等则表示每个机器都运行完指定的命令了，此时schedular节点会向barrier_group的每个机器发送一个返回的信息，并解除其阻塞。

3.7.2 初始化

ps-lite 使用 Barrier 来控制系统的初始化，就是大家都准备好了再一起前进。这是一个可选项。具体如下：

- Scheduler等待所有的worker和server发送BARRIER信息；
- 在完成**ADD_NODE**后，各个节点会进入指定 group 的**Barrier**阻塞同步机制（发送 BARRIER 给 Scheduler），以保证上述过程每个节点都已经完成；
- 所有节点（worker和server，包括scheduler） 等待scheduler收到所有节点 BARRIER 信息后的应答；
- 最终所有节点收到scheduler 应答的**Barrier** message后退出阻塞状态；

3.7.2.1 等待 BARRIER 消息

Node会调用 Barrier 函数 告知Scheduler，随即自己进入等待状态。

注意，调用时候是

```
if (do_barrier) Barrier(customer_id, kWorkerGroup + kServerGroup + kScheduler);
```

这就是说，等待所有的 group，即 scheduler 节点也要给自己发送消息。

```
void Postoffice::Barrier(int customer_id, int node_group) {
  if (GetNodeIDs(node_group).size() <= 1) return;
  auto role = van_->my_node().role;
  if (role == Node::SCHEDULER) {
    CHECK(node_group & kScheduler);
  } else if (role == Node::WORKER) {
    CHECK(node_group & kWorkerGroup);
  } else if (role == Node::SERVER) {
    CHECK(node_group & kServerGroup);
  }

  std::unique_lock<std::mutex> ulk(barrier_mu_);
  barrier_done_[0][customer_id] = false;
  Message req;
  req.meta.recver = kScheduler;
  req.meta.request = true;
  req.meta.control.cmd = Control::BARRIER;
  req.meta.app_id = 0;
  req.meta.customer_id = customer_id;
  req.meta.control.barrier_group = node_group; // 记录了等待哪些
  req.meta.timestamp = van_->GetTimestamp();
  van_->Send(req); // 给 scheduler 发给 BARRIER
  barrier_cond_.wait(ulk, [this, customer_id] { // 然后等待
      return barrier_done_[0][customer_id];
    });
}
```

3.7.2.2 处理 BARRIER 消息

处理等待的动作在 Van 类之中，我们提前放出来。

具体ProcessBarrierCommand逻辑如下：

- 如果 msg->meta.request 为true，说明是 scheduler 收到消息进行处理。
  - Scheduler会对Barrier请求进行增加计数。
  - 当 Scheduler 收到最后一个请求时（计数等于此group节点总数），则将计数清零，发送结束Barrier的命令。这时候 meta.request 设置为 false；
  - 向此group所有节点发送 `request==false` 的 `BARRIER` 消息。
- 如果 msg->meta.request 为 false，说明是收到消息这个 respones，可以解除barrier了，于是进行处理，调用 Manage 函数 。
  - Manage 函数 将app_id对应的所有costomer的 `barrier_done_` 置为true，然后通知所有等待条件变量 `barrier_cond_.notify_all()` 。

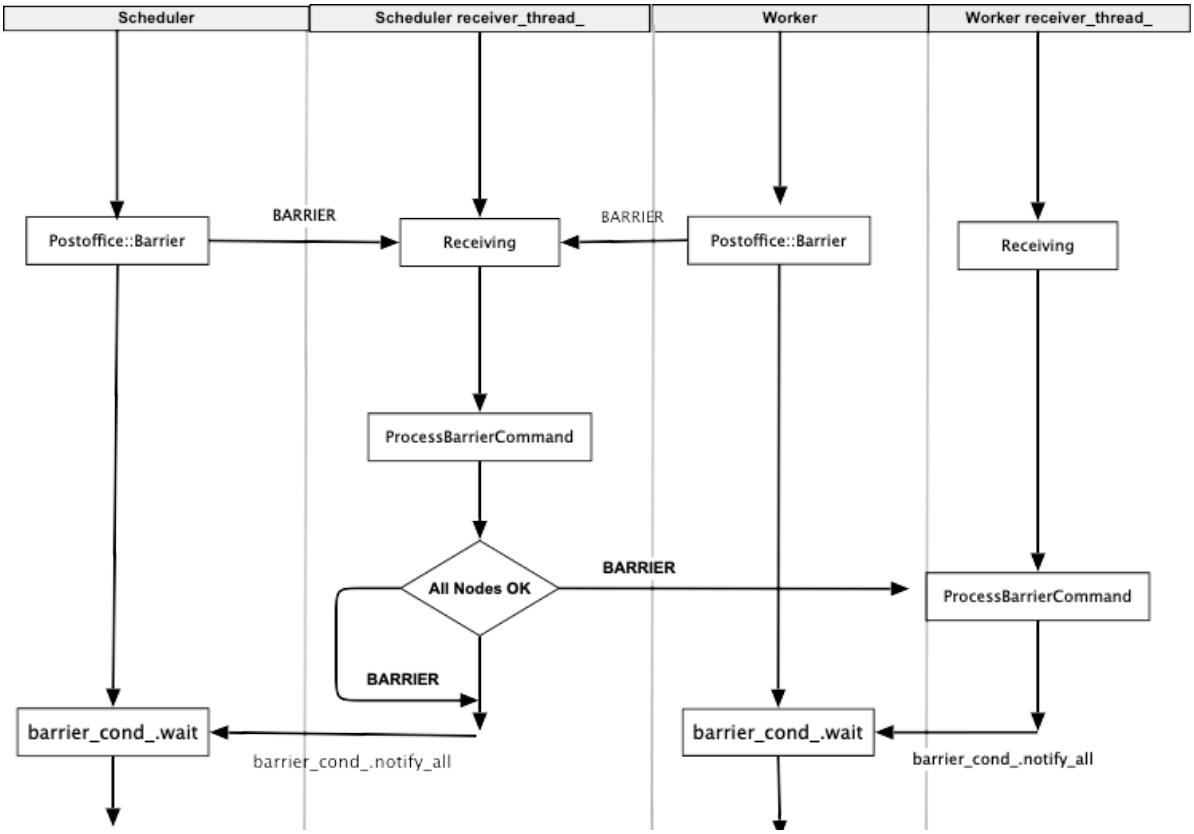
```
void Van::ProcessBarrierCommand(Message* msg) {
  auto& ctrl = msg->meta.control;
  if (msg->meta.request) {  // scheduler收到了消息，因为 Postoffice::Barrier函数 会在发送时候做设置为true。
    if (barrier_count_.empty()) {
      barrier_count_.resize(8, 0);
    }
    int group = ctrl.barrier_group;
    ++barrier_count_[group]; // Scheduler会对Barrier请求进行计数
    if (barrier_count_[group] ==
```

```
        static_cast<int>(Postoffice::Get()->GetNodeIDs(group).size())) { // 如果相
等，说明已经收到了最后一个请求，所以发送解除 barrier 消息。
      barrier_count_[group] = 0;
      Message res;
      res.meta.request = false; // 回复时候，这里就是false
      res.meta.app_id = msg->meta.app_id;
      res.meta.customer_id = msg->meta.customer_id;
      res.meta.control.cmd = Control::BARRIER;
      for (int r : Postoffice::Get()->GetNodeIDs(group)) {
        int recver_id = r;
        if (shared_node_mapping_.find(r) == shared_node_mapping_.end()) {
          res.meta.recver = recver_id;
          res.meta.timestamp = timestamp_++;
          Send(res);
        }
      }
    }
  } else { // 说明这里收到了 barrier respones，可以解除 barrier了。具体见上面的设置为
false处。
    Postoffice::Get()->Manage(*msg);
  }
}
```

Manage 函数就是解除了 barrier。

```
void Postoffice::Manage(const Message& recv) {
  CHECK(!recv.meta.control.empty());
  const auto& ctrl = recv.meta.control;
  if (ctrl.cmd == Control::BARRIER && !recv.meta.request) {
    barrier_mu_.lock();
    auto size = barrier_done_[recv.meta.app_id].size();
    for (size_t customer_id = 0; customer_id < size; customer_id++) {
      barrier_done_[recv.meta.app_id][customer_id] = true;
    }
    barrier_mu_.unlock();
    barrier_cond_.notify_all(); // 这里解除了barrier
  }
}
```

具体示意如下：



至此，Postoffice的分析我们初步完成，其余功能我们将会结合 Van 和 Customer 在后续文章中分析。