本来想起一个更搏人眼球的标题，比如《都2202年了，再不知道PS，你就out了》或者《面试算法工程师，面试官问了我一个PS的问题，结果挂了，......》，后来想想作罢，一是有点贩卖焦虑，二是容易被人当成卖课的广告。不过，**对于推荐系统来说，Parameter Server（PS）的重要性是再怎么强调也不为过的**。你想想，面对源源不断的、上T的训练数据，再加上以亿计、以十亿计的特征空间，我们是如何做到每隔X分钟就训练出一版模型，推到线上部署？背后的利器就是一个功能强大、鲁棒可靠的Paramter Server。因此，作为在推荐、广告、搜索算法里卷得不亦乐乎的我们，不了解点PS的技术原理是说不过去的，否则，搞不定PS，再nb的模型也训练不出来。

今天要讲的ps-lite，就是Parameter Server的早期代表，首开ps大规模应用之先河。看名字里的lite就能猜出，“简洁轻巧”是它的一大特点。

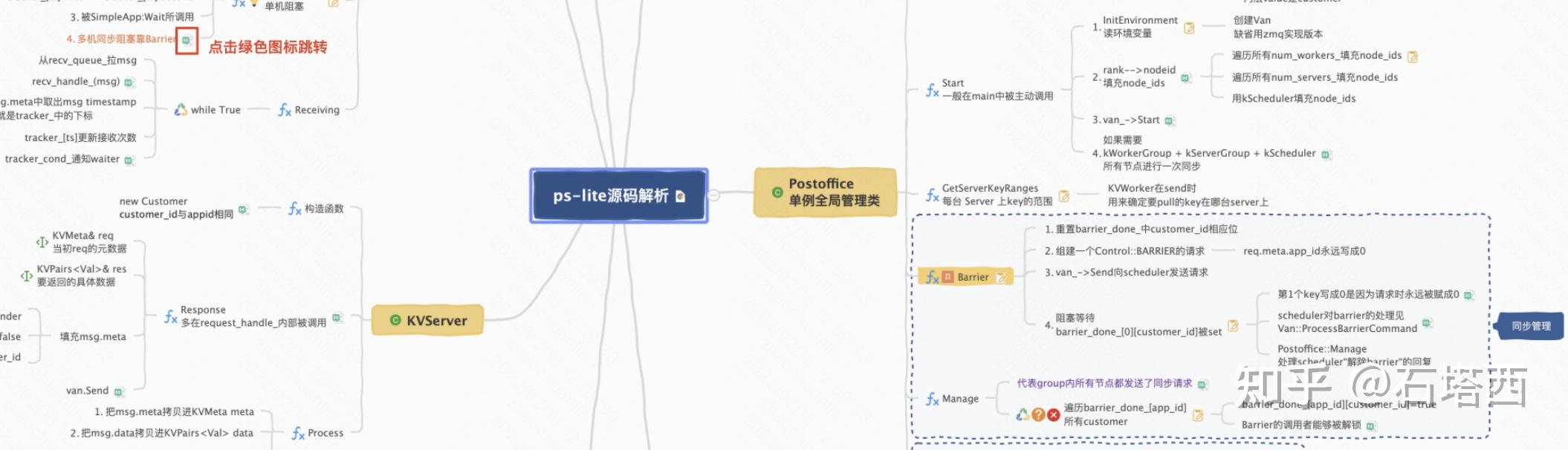
* 相比于它的那些大块头后辈，比如阿里的XDL、腾讯的angel、uber的hovord等，pslite所提供的功能是相对简单和有限的。
  + 我感觉，pslite主要是针对分布式LR或分布式FM而设计的，要基于pslite分布式训练DNN，自己要写的代码就不少。
  + 另外，pslite也缺乏，现代PS针对推荐系统特色而专门做的一些优化，比如embedding和dnn weight使用不同的通信+同步模式，为了避免ps膨胀而引入的特征准入+逐出机制、...等。
* 但是，“麻雀虽小，五脏俱全”，得益于pslite的简洁轻巧，给了我们一个深入ps内部、一探ps运行机理的机会。也为我们在实际工作中，调优pslit那些更强大也更复杂的后辈，奠定了基础。

趁着春节放假期间，我重新研读了一遍pslite的代码，彻底搞清楚了几个之前一直含糊的问题。知乎上有几篇解析pslite源码的文章，但是写得太简单，缺乏细节讲述。虽说是lite，但是也绝非一篇文章所能涵盖的。所以，我打算分4期文章，仔细解析一下pslite源码。

1. 第一章（*本章*），算是热身，先讲一下我们为什么需要Parameter Server，然后再分析一下pslite的入门demo程序，最后介绍pslite中的几个基本概念，为后续阅读打好基础。
2. [第二章](https://zhuanlan.zhihu.com/p/467693949)，讲解pslite中的PostOffice和Van两个类，涉及pslite中节点管理、消息分发、多机同步等方面的知识。
3. [第三章](https://zhuanlan.zhihu.com/p/467697961)，讲解pslite中的Customer, Worker和Server三个类，介绍pslite最常用的两个动作Push/Pull是如何实现的。
4. [第四章](https://zhuanlan.zhihu.com/p/467702941)，掌握了pslite的运行原理后，让我们看看基于pslite如何实现一个分布式的FM算法。

在正式开始解析之前，我首先声明两点：

1. 我的解析，是从PS的使用者的视角出发的。如果从ps架构开发的角度，我的解析还有不少遗漏的地方，敬请读者谅解。
2. 用文字来描述代码流程，其实是苍白无力的。由于我没时间做视频，因此，我提供了一份ps-lite源码思维导图。感兴趣的同学，**关注我的微信公众号“推荐道”，回复pslite领取**。下载免费版MindMaster打开后，读我的解析文章与看思维导图相结合，更容易理解。看思维导图时，重点是节点中的绿色图标，比如点击下图中的“多机同步要靠Barrier”后面的绿色图标，就能跳转到讲Barrier函数的笔记部分。



好，接下来正式开始我们的pslite学习之旅。

**我们为什么需要Parameter Server?**

**传统分布式计算的不足**

推荐系统对Parameter Server的需求并不是凭空产生的，而是针对推荐系统的两大痛点：**海量数据+高维稀疏特征空间**，应运而生。

其实推荐系统对海量数据并不陌生，我们有Hadoop/Spark这样的工具对付它们。所以很直觉的想法就是，能不能拿Hadoop/Spark来训练模型？想想也很简单，

* 数据分散到slave机器上
* master将模型的参数广播到所有slave节点
* 每个slave节点收到最新的参数后，用本地数据，计算出梯度，并上传至master
* master收集齐各slave节点发来的梯度后，聚合梯度，更新模型参数，准备下一轮训练。

so far so good。但是该方案忽略了推荐系统中数据的第二个特点，就是“高维稀疏的特征空间”，给以上方案造成两个困难：

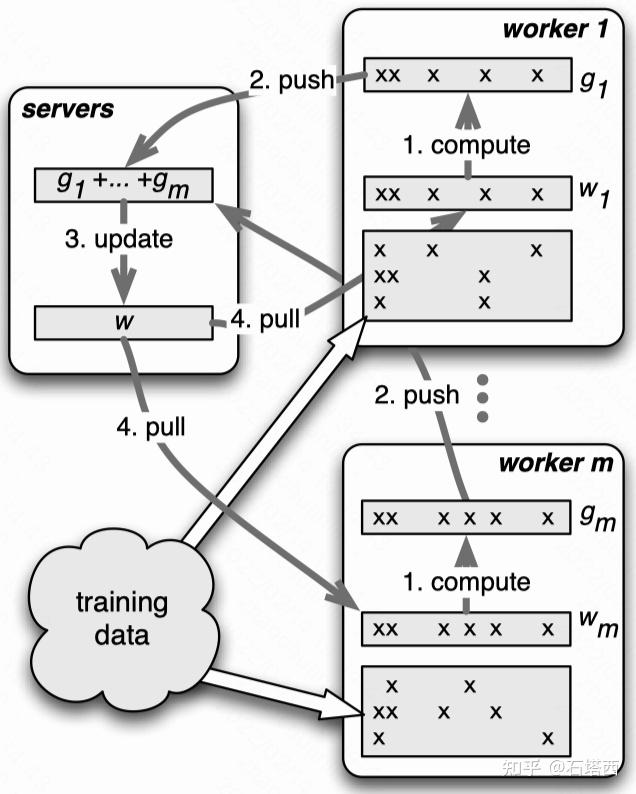
* 推荐系统的特征动辄上亿、上十亿，每个特征的embedding也可能是16位、32位也不短，这么大的参数量是一台master所容纳不下的。
* 要将这么大的参数量“广播”到各slave节点，占据的带宽、造成的时延也是不敢想像的。

**基于PS的分布式训练**

为了解决推荐系统中大规模分布式训练的难题，经典论文《Scaling Distributed Machine Learning with the Parameter Server》提出了Parameter Server架构。简单来说，Parameter Server就是一个分布式的KV数据库，两点设计使它能够克服Master/Slave架构应用于大规模分布式训练时的困难：

* 所有参数不再存储于单一的master节点，而是由一群ps server节点负责存储、读写；
* 得益于推荐系统的特征是超级特征的特点，一个batch的训练数据所包含的feature数目是有限的，因此，我们没必要训练每个batch时都将整个模型（i.e., 上亿的embedding）在server/worker之间传来传去，而只传递当前batch所涵盖的有限几个参数就可以了，从而大大节省了带宽。

再稍微具体一点，基于Parameter Server的训练流程如下图所示：



* 系统中主要有两大类节点
  + Work节点，负责读取本地的局部训练数据，计算梯度
  + Server节点，负责存储局部的参数空间，响应worker请求，对自己负责的局部参数进行读写。
* **[Worker]** 训练时，每个worker先归纳一下当前batch需要涉及到哪些feature\_id。
  + 观察图中的x，其所在行代表样本，其所在列代表feature\_id，可以发现训练数据的确是非常稀疏的。
  + 因此一个batch所涵盖的特征相对于整个特征空间，仍然是十分渺小的。
* **[Worker]** 每个worker向整个ps server集群请求“当前batch涵盖的feature”的最新参数（e.g., 一阶权重或embedding）。即图中的Pull。
* **[Server]** 因为每台ps server只拥有一部分feature的参数，所以尽管图中没有表现，但是中间会有一个路由机制，将worker的请求拆分、路由到合适的ps server上。
* **[Server]** 收到“拉数据”请求的ps server，从本地存储中找到这些feature\_id的最新参数，回复。
* **[Worker]** 收到ps server发送回来的各feature的最新参数，worker就可以在当前batch的数据上进行前代+回代，得到这些feature上的梯度。
* **[Worker]** Worker将计算好的梯度发往Server，即图中的Push。
* **[Server]** 接到worker发来的梯度后，Server汇总聚合梯度，再用各种SGD的变体（e.g., FTRL、Adam、Adagrad等）更新本地参数。

**比较Tricky的问题**

Parameter Server的基本运行流程就如上描述。读到这里，一些有经验的同学可能会问到如下问题：

**各worker之间如何同步？**

同步的确是Parameter Server里面比较tricky的一个问题，控制协议有好几种，比如：BSP（Bulk Synchronous Parallel），SSP（Stalness Synchronous Parallel）和ASP（Asynchronous Parallel）。详细情况，感兴趣的同学可以移步pslite论文原文。

在纯异步的ASP模式中，每台worker在发送完自己的梯度后，不用等其他worker，就可以开始训练下一个batch的数据。由于无须同步，ASP的性能优势比较明显。但是，的确存在可能性，一个非常慢的worker基于老参数计算出过时梯度，传到server端会覆盖一些参数的最新进展。**但是在实际场景下，由于推荐系统的特征空间是超级稀疏的，因此两个worker同时读写同一feature造成冲突的可能性还是较低的，因此纯异步ASP模式的应用还是比较普遍的**。

**DNN Weight怎么办？**

的确，以上对PS“合理性”的很多描述，都基于一个前提，即推荐系统中的参数空间是超级稀疏的，因此我们pull时只需要pull一小部分参数，而push时忽略冲突的可能性，也没那么严重。 以上假设对于feature embedding是成立的，但是模型中DNN各层权重可怎么办？worker要完成训练，必须要拿到完整的、所有各层dnn weight不可。

而在如何处理dnn weight的问题上，各大厂推出的Parameter Server各自形成了独自的风格：

* 有的框架还是用PS来处理dnn weight，而有的框架基于dnn weight的稠密本质，用AllReduce来完成dnn weight在各worker节点的同步。
* 如前所述，由于feature embedding是超级稀疏的，忽略冲突也没啥太大问题，因此训练feature embedding往往采用纯异步ASP模式；而dnn weight是各台worker都需要的，不同步必然造成冲突，因此采用纯同步SSP模式。

**入门ps-lite**

从[**GitHub**](https://link.zhihu.com/?target=https%3A//github.com/dmlc/ps-lite)下载源码后，进到ps-lite根目录下，运行make -j4可编译。

编译完成后，进到test文件夹，运行以下命令，执行demo程序

*# DMLC\_NUM\_SERVER 为 2；*

*# DMLC\_NUM\_WORKER 为 3；*

./local\_multi\_workers.sh 2 3 ./test\_kv\_app\_multi\_workers

想知道demo程序究竟做了什么，打开tests/test\_kv\_app\_multi\_workers.cc

**main函数**

**int** **main**(**int** argc, **char** **\***argv[]) {

**bool** isWorker **=** (strcmp(argv[1], "worker") **==** 0);

*// ------------ 启动server*

**if** (**!**isWorker) {

Start(0); *// 启动PostOffice和Van*

StartServer();*// 内部启动了app\_id=0的server*

......

}

*// ------------ 启动worker*

std**::thread** t0(RunWorker, 0);*// 启动app\_id=0，customer\_id=0的worker thread*

std**::thread** t1(RunWorker, 1);*// 启动app\_id=0，customer\_id=1的worker thread*

t0.join();

t1.join();

**return** 0;

}

**Server逻辑**

**void** **StartServer**() {

**if** (**!**IsServer()) **return**;

**auto** server **=** **new** KVServer**<float>**(0);*// app\_id=0*

*// 注册对消息的处理真正的处理逻辑发生在KVServerDefaultHandle*

server**->**set\_request\_handle(KVServerDefaultHandle**<float>**());

RegisterExitCallback([server](){ **delete** server; });

}

以上代码只是启动server而已，真正的server端逻辑在KVServerDefaultHandle类中

***KVServerDefaultHandle***

**struct** **KVServerDefaultHandle** {

*// req\_meta:请求的一些元信息*

*// req\_data: 请求的具体数据*

**void** **operator**()(

**const** KVMeta**&** req\_meta, **const** KVPairs**<**Val**>&** req\_data, KVServer**<**Val**>\*** server) {

size\_t n **=** req\_data.keys.size();

KVPairs**<**Val**>** res;*//要发回worker的response*

......

**for** (size\_t i **=** 0; i **<** n; **++**i) {

Key key **=** req\_data.keys[i];*// 遍历worker发送来的每个key*

**if** (req\_meta.push) {*// 如果是push请求，就将worker发来的数字累加在key指定位置*

store[key] **+=** req\_data.vals[i];

}

**if** (req\_meta.pull) {*//如果是pull请求，将key指定的数据塞进response*

res.vals[i] **=** store[key];

}

}

server**->**Response(req\_meta, res);*//回复worker*

}

*// 参数保存进一个map中*

std**::**unordered\_map**<**Key, Val**>** store;

};

**Worker逻辑**

由于是demo程序，worker的逻辑也非常简单，制造一些随机数发送给server，再从server拉取每个key对应数字的和（见KVServerDefaultHandle），进行检验。

**void** **RunWorker**(**int** customer\_id) {

Start(customer\_id);

**if** (**!**IsWorker()) {

**return**;

}

KVWorker**<float>** kv(0, customer\_id);

*// init*

**int** num **=** 10000;

std**::**vector**<**Key**>** keys(num);

std**::**vector**<float>** vals(num);

*// ------------ 生成key和要发送的随机数*

**int** rank **=** MyRank();

srand(rank **+** 7);

**for** (**int** i **=** 0; i **<** num; **++**i) {

keys[i] **=** kMaxKey **/** num **\*** i **+** customer\_id;

vals[i] **=** (rand() **%** 1000);

}

*// ------------ 将以上生成的key和value，向server发送50遍（server将累加50次）*

**int** repeat **=** 50;

std**::**vector**<int>** ts;*//ts保存每次异步发送的request\_id*

**for** (**int** i **=** 0; i **<** repeat; **++**i) {

ts.push\_back(kv.Push(keys, vals));

......

}

*// 等待所有async push request完成*

**for** (**int** t : ts) kv.Wait(t);

*// ------------ 从server端拉取累加结果*

std**::**vector**<float>** rets;*//存放pull来的结果*

kv.Wait(kv.Pull(keys, **&**rets));

......

**float** res **=** 0;

**for** (**int** i **=** 0; i **<** num; **++**i) {

*// 因为每个key/value pair都被重复发送了repeat遍*

*// server端也累加了repeat遍*

*// rets[i]和vals[i] \* repeat，理论上应该相等*

res **+=** fabs(rets[i] **-** vals[i] **\*** repeat);

......

}

CHECK\_LT(res **/** repeat, 1e-5);

LL **<<** "error: " **<<** res **/** repeat **<<** ", " **<<** res2 **/** (2 **\*** repeat);

*// stop system*

Finalize(customer\_id, true);

}

**ps-lite基础概念**

**三种节点类型**

在使用SGD进行训练的场景下：

* worker：从server pull最新的待优化参数，读本地数据，计算出梯度，向server push梯度
* server： 其实就是一个key/value数据库，
  + 应对pull请求，将worker请求的参数的最新值发送回去
  + 应对push请求，聚合各worker发送过来的gradient，再利用各种SGD的变体（e.g., Adam，Adagrad）更新weight
* scheduler: 一个系统中只有一个scheduler节点，主要负责
  + 其他server/worker节点的管理：有节点加入或退出时，都需要将本集群内最新的节点名单广播给群内的所有节点
  + 多节点之间的数据同步：统计有多少节点已经运行了到“同步”点；如果所有节点都“到齐”了，向所有参与节点的节点发“解除阻塞”的信息

***Node Id***

每个node全局唯一的整数标识，在通信模块Van的实现中，每个node id能够与其ip地址一一对应起来。每个节点的node\_id是在每个节点向scheduler通过ADD\_NODE命令注册之后，由scheduler统一分配的。

看代码时，还发现有一个rank的概念。rank是节点在group内部的逻辑顺序编号。比如，如果有3个worker，那么三个worker的rank分别是0,1,2。二者之间的转换见PostOffice:WorkerRankToID和PostOffice:ServerRankToID。

***Node Group***

base.h中还定义了三个特殊的node id，代表整个group（e.g., 所有server节点）。如代码中所示，这三个group id可以任意组合，表示跨group的node集合。比如，kServerGroup+kWorkerGroup=6，代表所有server+所有worker节点。

**static** **const** **int** kScheduler **=** 1;

**static** **const** **int** kServerGroup **=** 2;

**static** **const** **int** kWorkerGroup **=** 4;

**五个重要角色**

* PostOffice： 每个进程（无论scheduler/worker/server），有且只有一个“邮局”，是该进程的消息集散中心。它掌管着在它那里等消息的“顾客”的名单。 |
* Van：每个邮局只有一辆“邮车”，负责收发信件的具体工作。派发信件时，Van通过PostOffice的名单，找到某个“顾客”，将信件交到他手上。 |
* Customer：邮局的顾客，其实就是个中介。worker和server为了更好地专心业务工作，将“在邮局排队等消息”的工作外包给了“顾客”。Customer从Van手中接过信件后，转手就交给它的顾主，worker或server。 |
* Worker：Customer的顾主，训练时主要负责处理样本、计算梯度等工作。
* Server：Customer的顾主，训练时主要负责接收梯度，汇总梯度（同步模式下），更新权重等工作。

***App Id vs. Customer Id***

对比KVServer与KVWorker的构造函数，

**explicit** **KVServer**(**int** app\_id) **:** SimpleApp() {

......

*// server创建customer时，第2个参数customer\_id也赋成了app\_id*

obj\_ **=** **new** Customer(app\_id, app\_id, std**::**bind(**&**KVServer**<**Val**>::**Process, **this**, \_1));

}

......

**explicit** KVWorker(**int** app\_id, **int** customer\_id) **:** SimpleApp() {......}

可以看到，建立一个server只需要app\_id，而建立一个worker需要app\_id+customer\_id。这从一个侧面反映出，一个app可以对应多个customer。

另一个证据是在Postoffice::GetCustomer函数中，这个函数先根据app\_id找到与某个app对应的所有customer，再在其中根据customer\_id查找。这也说明了，app和customer是一对多的关系。

Customer**\*** Postoffice**::**GetCustomer(**int** app\_id, **int** customer\_id, **int** timeout) **const** {

Customer**\*** obj **=** **nullptr**;

............

**const** **auto** it **=** customers\_.find(app\_id);

**if** (it **!=** customers\_.end()) {

std**::**unordered\_map**<int**, Customer**\*>** customers\_in\_app **=** it**->**second;

obj **=** customers\_in\_app[customer\_id];

............

**return** obj;

}

可以将app与customer的关系总结如下：

* app\_id
  + 其实可以认为是server id（注意不是server node id）或者database id
  + 比如，一个完整的FM程序，在每个ps server节点都需要两个“数据库”。一个库存放feature的一阶权重，另一个库存放feature的embedding。
  + 这时，我们就需要分别用两个app\_id，定义两个KVServer，分别处理一阶权重和embedding。
* customer\_id
  + 看上面的KVServer的构造函数，创建customer时所需要的customer\_id也赋成了app\_id，也就是server端，一个app只对应一个customer。
  + 而在worker端，如果你想获得多worker并发之外，在一个worker node上也启动多个线程，以进一步提高计算效率。
    - 这时，在每个worker node上，可以创建多个KVWorker，每个占据一个线程。
    - 这些worker有相同的app\_id，因为他们要读取、更新同一类KV数据库
    - 但是他们有不同的customer id，表示不同的工作线程
  + 多worker的详细代码，可以参考tests/test\_kv\_app\_multi\_workers.cc中的RunWorker函数

**辅助类**

有几个基本的辅助类先了解一下它们的用处

***SArray***

一个智能数组，功能上类似vector。但是有两点改进：一是当引用为0时，能够自动回收内存; 二是，实现零拷贝，以提高传递内容时的效率。

***KVPairs***

一个容器类，用于承载worker/server之间要传递的数据，包含三个成员

* SArray<Key> keys, key其实就是int64，表明一次通讯中传递了哪些key
* SArray<Val> vals，表明一次通讯中要传递的值
  + 举例在LR中，每个特征的feature id就是key，该feature对应的weight就是value，
    - 所以len(keys)==len(vals)，
    - 第j个特征对应的feature id是keys[j]，其weight就是vals[j]
  + 举例在FM中，每个特征的feature id就是key，其对应的embedding就是value。
    - len(vals)=len(keys)\*embed\_dim，embed\_dim是embedding的长度
    - 第j个feature对应的feature\_id就是keys[j]，其embedding对应[vals[(j-1) \* embed\_dim : j \* embed\_dim]
* SArray<int> lens，表示每个key对应的value的长度
  + 实际场景中，未必每个key都对应相同的长度，比如，我们将dnn中每层的权重表示为多个kv pair，显然每层权重对应的float array的长度是不同的，这时就需要lens来帮忙。
  + 比如：keys=[11,55]，lens=[2,3]，
    - keys[0] (feature\_id = 11) 的embedding长度是2，内容是[values[0] 和 values[1]]
    - keys[1] (feature\_id = 55) 的embedding长度是3, 内容是[values[2]，values[3]，values[5]]
  + 像上面所举的FM的例子是一个特殊情况，这时lens可以为空，表示每个key对应的value的长度都相等

***Meta***

通讯信息的元数据

* sender & recver: 通讯双方的node id
* app\_id & customer\_id: 即使在一个node上，也可能有多个server和worker，node会根据app\_id+customer\_id来找到具体的信息接收方
* bool request & response：收到的消息是别人发来的request，还是别人对我的response
* bool pull & push：比如对于ps server来说，同样是request，那么这个request到底是写数据（push）还是读数据（pull）

***Control***

主要用于描述在scheduler ~ server和scheduler ~ worker之间传递的控制命令，主要成员有

* barrier\_group：标识在哪组节点内部需要同步（e.g., 所有worker节点需要同步，或，所有server节点需要同步，等）
* vector<Node> node: 当有节点加入或退出时，scheduler会将最新的节点列表写入这个vector，广播到群内所有worker+server节点

***Message***

worker/server之间通讯的信息

* 包含Meta，如前所述，是元数据
* vec<SArray<char>> data，长度是2或3，和KVPairs的转化关系如下：

KVPairs**<**Val**>** kvs;

kvs.keys **=** msg.data[0];

kvs.vals **=** msg.data[1];

**if** (msg.data.size() **>** (size\_t)2) {

kvs.lens **=** msg.data[2];

}

下一章见《[ps-lite源码解析(2)：邮局和邮车](https://zhuanlan.zhihu.com/p/467693949)》，讲解pslite中的PostOffice和Van两个类，涉及pslite中节点管理、消息分发、多机同步等方面的知识。

- END -

本系列是对Parameter Server的经典实现ps-lite的源码解析，共分4章：

1. [第一章](https://zhuanlan.zhihu.com/p/467650462)，算是热身，先讲一下我们为什么需要Parameter Server，然后再分析一下pslite的入门demo程序，最后介绍pslite中的几个基本概念，为后续阅读打好基础。
2. 第二章（*本章*），讲解pslite中的PostOffice和Van两个类，涉及pslite中节点管理、消息分发、多机同步等方面的知识。
3. [第三章](https://zhuanlan.zhihu.com/p/467697961)，讲解pslite中的Customer, Worker和Server三个类，介绍pslite最常用的两个动作Push/Pull是如何实现的。
4. [第四章](https://zhuanlan.zhihu.com/p/467702941)，掌握了pslite的运行原理后，让我们看看基于pslite如何实现一个分布式的FM算法。

另外，用文字来描述代码流程，是苍白无力的。 为此，我提供了一份ps-lite源码思维导图。感兴趣的同学，**关注我的微信公众号“推荐道”，回复pslite领取**。下载免费版MindMaster打开后，读我的解析文章与看思维导图相结合，更容易理解

**PostOffice：单例全局管理类**

PostOffice是一个全局单例类，也就是一个进程（无论是scheduler/worker/server进程）里只有一个PostOffice实例，掌握全局信息

**重要成员变量**

* Van\* van，负责对外通信。即，一个进程只有一个邮局（PostOffice），而一个邮局只有一辆邮车（Van）
* map<int, vec<int>> node\_ids, key是group\_id，value是这一group下所有node id的集合。我们可以通过该结构找到，比如说，所有worker节点
* map<int, map<int, Customer>> customers\_, 存储本地的customer
  + 如前所述，特别是worker进程中，允许有多个worker thread（用customer id标识）向同一个远端KV数据库（用app id标识）读写
  + 外层map的key是app id，内层map的key是customer id，二者结合起来，能够标识标识一个server/worker thread（这里的Customer）
* map<int, map<int,bool>> barrier\_done\_,
  + 外层map的key是app id，内层map的key是customer id，通过app\_id+customer\_id两级检索，能够唯一定位一个server/worker thread
  + 内层map的bool值，表示这个server/worker thread“同步是否完成”
* vector<Range> server\_key\_ranges\_, 每个ps server上所存储的key的区间范围，被worker用于确定在pull/push时该向哪些server发请求。接下来讲GetServerKeyRanges函数时会提到。

**启动**

见PostOffice:Start函数，这个函数一般是在main函数时被主动调用

**void** Postoffice**::**Start(**int** customer\_id, **const** **char\*** argv0, **const** **bool** do\_barrier) {

......

*// -------------- 从环境变量中读取配置，生成Van*

InitEnvironment();

*// -------------- 把每个节点添加到相应的group的节点列表中*

**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);

}

}

**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);

}

}

**for** (**int** g : {kScheduler, kScheduler **+** kServerGroup **+** kWorkerGroup,

kScheduler **+** kWorkerGroup, kScheduler **+** kServerGroup}) {

node\_ids\_[g].push\_back(kScheduler);

}

*// -------------- 启动Van，其中会向Scheduler注册当前节点*

van\_**->**Start(customer\_id);

*// -------------- 如果有必要做一次同步，等待所有进程中的PostOffice:Start都执行完毕*

**if** (do\_barrier) Barrier(customer\_id, kWorkerGroup **+** kServerGroup **+** kScheduler);

}

删除一些不太重要的代码后，可以看到PostOffice的启动流程如下：

1. InitEnvironment从环境变量中读取配置，比如系统里面有多少server，有多少worker，本机的身份是什么 （scheduler? worker? server?），还创建了Van
2. 填充node\_ids\_，关键逻辑是，如果某个节点是server，那么它必须加入所有涉及server的group对应的node列表中，比如“所有server”这个group（代码中的kServerGroup）里面有它；比如，“所有worker+所有server”（代码里的kWorkerGroup + kServerGroup）这个group里也有它。
3. Van->Start，接下来会讲到，启动邮车的过程中，会启动消息处理线程，还会将自己向scheduler注册

**确定每台Server上Key的范围**

每个worker在开始训练某个batch之前，首先要统计一下这批数据中一共包含哪些unique feature ids， 即keys。但是这些keys分布在哪些ps server上，worker该向哪些ps server发送push/pull请求？这时就需要Postoffice::GetServerKeyRanges提前将每台server上的key的分布计算好，保存进server\_key\_ranges\_成员变量中。

其实这个过程非常简单，因为key都是int64类型的，pslite假设int64能够表示的所有key在所有ps server上平均分布，所以每台ps server上的key的范围只要拿int64最大值和num\_servers简单计算一下各分段的上下界就行了。

**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\_;

}

**跨节点同步管理**

PostOffice最重要的功能就是提供了多节点之间的同步功能，由Postoffice::Barrier和Postoffice::Manage两个函数配合完成。

注意这个同步功能，不要和代码中的SimpleApp:Wait相混淆，

* Wait提供的**单节点同步阻塞**。pslite所有IO操作都是异步的，某些场景下需要等待异步IO完成（比如worker等待pull embedding彻底结束），才能进行下一步的操作。这时就要用到SimpleApp:Wait阻塞等待异步IO完成。
* PostOffice提供的“同步”功能是指**多节点之间的同步**，比如**同步SGD**中，ps server将所有worker发送的gradient都收集齐之后才更新权重，这期间所有worker需要阻塞住，等待彼此。

真正的等待逻辑是在scheduler节点的Van中完成的，PostOffice的Barrier和Manage只是负责向scheduler发送请求，和处理scheduler的回复。

PostOffice:Barrier简单来说，就是做两件事，先向scheduler发送barrier请求，再阻塞等待scheduler的回复。

*// 第一个参数customer\_id，表明发送barrier请求的发送方的身份*

*// 第二个参数node\_group，表明在多大范围内同步，比如node\_group=kWorkerGroup，代表所有worker参与同步*

**void** Postoffice**::**Barrier(**int** customer\_id, **int** node\_group) {

......

*// 重置标志位，表明自己(customer\_id)没有收到“同步完成”消息*

barrier\_done\_[0][customer\_id] **=** false;

*// 准备同步请求*

Message req;

req.meta.recver **=** kScheduler;*// 向scheduler发送*

req.meta.request **=** true;

req.meta.control.cmd **=** Control**::**BARRIER;*// 请求类型是“同步”*

req.meta.app\_id **=** 0;

req.meta.customer\_id **=** customer\_id;

req.meta.control.barrier\_group **=** node\_group;*//哪些节点需要同步*

req.meta.timestamp **=** van\_**->**GetTimestamp();

*// 通过Van发送请求*

van\_**->**Send(req);

*// 条件变量阻塞等待barrier\_done\_[0][customer\_id]被设置*

*// 代表自己(i.e., customer\_id)收到了scheduler“同步完成”的消息*

barrier\_cond\_.wait(ulk, [**this**, customer\_id] { **return** barrier\_done\_[0][customer\_id]; });

}

scheduler收到“同步请求”后，在Van::ProcessBarrierCommand中进行处理。简单来说，就是统计有多少节点已经向自己发送了barrier请求，如果请求barrier的节点数已经等于同步范围（见Barrier函数的node\_group参数）内所有节点数，认为同步完成，向同步范围（i.e., Barrier的node\_group参数）内所有节点回复“同步完成”。具体细节，我们接下来ProcessBarrierCommand再细看。

各调用了PostOffice:Barrier等待同步完成的节点，在收到scheduler“同步完成”的信息后，调用PostOffice:Manage函数进行处理，如下所示。在这个函数中，根据app\_id找到其包含的所有customer\_id，设置barrier\_done\_，从而释放正在阻塞的Barrier函数。

**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) {

......

**auto** size **=** barrier\_done\_[recv.meta.app\_id].size();

**for** (size\_t customer\_id **=** 0; customer\_id **<** size; customer\_id**++**) {

*// Barrier正在阻塞等待这个变量被设置*

barrier\_done\_[recv.meta.app\_id][customer\_id] **=** true;

}

......

barrier\_cond\_.notify\_all();*//通知Barrier可能解锁*

}

}

其实这里，我有一个**疑问**，看scheduler的代码Van::ProcessBarrierCommand，scheduler是根据请求barrier的节点数已经等于同步范围（ctrl.barrier\_group）内所有节点数来判断同步是否完成，注意这里根据的是节点数，而不是customer数。而**同一个worker节点上可以有多个customer thread同时工作**，我感觉节点数match，其实并不能说明同步完成。先把这个疑问列出来，有时间再仔细看一下代码。

**Customer管理**

如前所述，PostOffice掌握着在它那里等消息的顾客，由GetCustomer和AddCustomer两个函数完成。

在介绍PostOffice的customers\_成员变量时也提到过，PostOffice是按app\_id+customer\_id两个层次管理customer。

Customer**\*** Postoffice**::**GetCustomer(**int** app\_id, **int** customer\_id, **int** timeout) **const** {

Customer**\*** obj **=** **nullptr**;

......

**const** **auto** it **=** customers\_.find(app\_id);*//先按app\_id查找*

**if** (it **!=** customers\_.end()) {

std**::**unordered\_map**<int**, Customer**\*>** customers\_in\_app **=** it**->**second;

obj **=** customers\_in\_app[customer\_id];*//再按customer\_id查找*

**break**;

}

}

......

**return** obj;

}

理论上，一个worker process里面可以有多个worker thread，每个worker thread有自己的customer id，但它们的读写目标是同一类参数（e.g., 一阶权重，或，embedding），所以它们有相同的app\_id。

**Van：通信模块**

定义在van.h中的Van是一个基类，定义了收发消息的接口。但是具体的IO逻辑(比如：)，是在子类中完成的。比如，缺省的基于ZMQ实现IO的ZMQVan，还有字节实现的IBVerbsVan。应用时创建哪个子类的实例，可以参考Postoffice::InitEnvironment()中对Van::Create的调用。

**启动**

被PostOffice->Start所调用，主要做如下几件事：

1. 和scheduler建立联接
2. 启动线程处理接受到的消息
3. 向scheduler发送ADD\_NODE信息注册自己

**void** Van**::**Start(**int** customer\_id) {

*// get scheduler info*

start\_mu\_.lock();

**if** (init\_stage **==** 0) {

......

*// connect to the scheduler*

Connect(scheduler\_);

......

*// start receiver*

receiver\_thread\_ **=**

std**::**unique\_ptr**<**std**::thread>**(**new** std**::thread**(**&**Van**::**Receiving, **this**));

init\_stage**++**;

}

......

*// --------------- 向Scheduler注册自己*

**if** (**!**is\_scheduler\_) {

*// let the scheduler know myself*

Message msg;

Node customer\_specific\_node **=** my\_node\_;

customer\_specific\_node.customer\_id **=** customer\_id;

msg.meta.recver **=** kScheduler;

msg.meta.control.cmd **=** Control**::**ADD\_NODE;

msg.meta.control.node.push\_back(customer\_specific\_node);

msg.meta.timestamp **=** timestamp\_**++**;

Send(msg);

}

*// wait until ready*

**while** (**!**ready\_.load()) { std**::**this\_thread**::**sleep\_for(std**::**chrono**::**milliseconds(100));}

......

}

**Send：发送消息**

具体逻辑调用SendMsg，由子类override实现。

**Receiving：接收消息**

在Van->Start中，创建了一个独立的线程运行Van::Receiving，执行“接收+处理”的循环。注意这里面调用的ProcessXXX的函数是不区分节点类型的，在一个函数里，如果当前节点是scheduler，执行“处理命令请求”的逻辑；否则，执行“处理命令回复”的逻辑。

**void** Van**::**Receiving() {

*// !!!!!! 注意这个nodes定义在while循环外面，相当于一个global variable*

*// 在while循环中处理了新节点加入或老节点退出，都反映在对这个nodes的修改上*

Meta nodes;

......

**while** (true) {

Message msg;

**int** recv\_bytes **=** RecvMsg(**&**msg);

......

**if** (**!**msg.meta.control.empty()) {

*// ------------------- 处理命令*

*// control msg*

**auto&** ctrl **=** msg.meta.control;

**if** (ctrl.cmd **==** Control**::**TERMINATE) {

ProcessTerminateCommand();

**break**;

} **else** **if** (ctrl.cmd **==** Control**::**ADD\_NODE) {

ProcessAddNodeCommand(**&**msg, **&**nodes, **&**recovery\_nodes);

} **else** **if** (ctrl.cmd **==** Control**::**BARRIER) {

ProcessBarrierCommand(**&**msg);

} **else** **if** (ctrl.cmd **==** Control**::**HEARTBEAT) {

ProcessHearbeat(**&**msg);

} **else** {

LOG(WARNING) **<<** "Drop unknown typed message " **<<** msg.DebugString();

}

} **else** {

*// ------------------- 处理数据信息*

ProcessDataMsg(**&**msg);

}

}

}

**ProcessBarrierCommand：处理同步命令**

与PostOffice:Start和PostOffice:Manage配合，完成多节点同步的功能。

**void** Van**::**ProcessBarrierCommand(Message**\*** msg) {

**auto&** ctrl **=** msg**->**meta.control;

**if** (msg**->**meta.request) {

*// ================ 如果是barrier request，一定是执行在scheduler上*

......

**int** group **=** ctrl.barrier\_group;*// 哪个范围内进行同步*

**++**barrier\_count\_[group];*// 发出barrier request的节点加1*

*// 如果已经发出barrier request的节点数目，已经等于*

*// 同步范围（由ctrl.barrier\_group指定）的所有节点，说明同步已经完成*

*// 对于一个worker node上有多个worker thread的情况怎么办？*

**if** (barrier\_count\_[group] **==**

**static\_cast<int>**(Postoffice**::**Get()**->**GetNodeIDs(group).size())) {

barrier\_count\_[group] **=** 0;*// reset for next time*

*// 回复ctrl.barrier\_group中的所有节点，通知它们“同步完成”*

Message res;

res.meta.request **=** 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** {

*// ================ 否则就是barrier response，一定是执行在worker/server上*

*// PostOffice:Manage负责设置标志位，翻译还在阻塞等待中的Barrier*

Postoffice**::**Get()**->**Manage(**\***msg);

}

}

从以上代码中，我们可以看到，scheduler是根据请求barrier的节点数已经等于同步范围（ctrl.barrier\_group）内所有节点数来判断同步是否完成。这里，我有疑问，**对于同一个worker节点上有多个worker thread同时工作的情况**，我感觉仅仅是worker节点数match了，其实并不能说明同步完成。先把这个疑问抛出来，有时间再仔细读一下源码。

**ProcessAddNodeCommand：集群内节点管理**

这个函数

* 在scheduler端，负责处理worker/server发送来的ADD\_NODE请求，维护全局节点列表
* 在worker/server端，用scheduler发送过来的全局节点列表，更新自己本地的节点列表

**void** Van**::**ProcessAddNodeCommand(Message**\*** msg, Meta**\*** nodes, Meta**\*** recovery\_nodes) {

......

**if** (is\_scheduler\_) {

*// ============ scheduler处理ADD\_NODE请求*

*// Meta\* nodes就是在Van::Receiving的while true循环外面的那个Meta nodes*

*// 由于定义在while true循环外，相当于一个全局变量*

*// 存放着向scheduler注册的所有节点*

ProcessAddNodeCommandAtScheduler(msg, nodes, recovery\_nodes);

} **else** {

*// ============ worker/server处理ADD\_NODE回复*

*// ctrl.node是scheduler发送过来的集群所有节点的列表*

**for** (**const** **auto&** node : ctrl.node) {

std**::**string addr\_str **=** node.hostname **+** ":" **+** std**::**to\_string(node.port);

**if** (connected\_nodes\_.find(addr\_str) **==** connected\_nodes\_.end()) {

*// !!! 注意，这里的Connect也并非任意两两节点之间都要建立联接*

*// 在子类实现中，会对要连接的节点进行筛选*

*// 比如worker只会和server建立联接，而不是会和其他worker建立联接*

Connect(node);

*// 用scheduler发送来的节点列表更新worker/server本地保存的节点列表*

connected\_nodes\_[addr\_str] **=** node.id;

}

**if** (**!**node.is\_recovery **&&** node.role **==** Node**::**SERVER) **++**num\_servers\_;

**if** (**!**node.is\_recovery **&&** node.role **==** Node**::**WORKER) **++**num\_workers\_;

}

ready\_ **=** true;

}

}

scheduler处理ADD\_NODE的函数在ProcessAddNodeCommandAtScheduler

*// Meta\* nodes就是在Van::Receiving的while true循环外面的那个Meta nodes*

*// 由于定义在while true循环外，相当于一个全局变量*

*// 存放着向scheduler注册的所有节点*

**void** Van**::**ProcessAddNodeCommandAtScheduler(Message**\*** msg, Meta**\*** nodes,

Meta**\*** recovery\_nodes) {

recovery\_nodes**->**control.cmd **=** Control**::**ADD\_NODE;

time\_t t **=** time(NULL);

size\_t num\_nodes **=**

Postoffice**::**Get()**->**num\_servers() **+** Postoffice**::**Get()**->**num\_workers();

**if** (nodes**->**control.node.size() **==** num\_nodes) {

*// 如果向scheduler注册的节点总数 == 预置里面指定的num\_worker+num\_server，说明所有节点都注册了*

......

nodes**->**control.node.push\_back(my\_node\_);

nodes**->**control.cmd **=** Control**::**ADD\_NODE;

Message back;

back.meta **=** **\***nodes;*// 向worker/server回复的内容就是scheduler已知的全部*

*// 遍历每个worker和每个server，返回scheduler已知的全部节点*

**for** (**int** r : Postoffice**::**Get()**->**GetNodeIDs(kWorkerGroup **+** kServerGroup)) {

**int** recver\_id **=** r;

**if** (shared\_node\_mapping\_.find(r) **==** shared\_node\_mapping\_.end()) {

back.meta.recver **=** recver\_id;

back.meta.timestamp **=** timestamp\_**++**;

Send(back);

}

}

ready\_ **=** true;

} **else** **if** (**!**recovery\_nodes**->**control.node.empty()) {

......

}

}

**ProcessDataMsg：处理数据消息**

如果传来的message的control部分为空，说明来者是一个数据包，由ProcessDataMsg处理。可以看到，Van只能简单地找到合适的Customer，将数据消息转交。

**void** Van**::**ProcessDataMsg(Message**\*** msg) {

*// 通过app\_id+customer\_id二层查找出相应的Customer*

**int** app\_id **=** msg**->**meta.app\_id;

**int** customer\_id **=**

Postoffice**::**Get()**->**is\_worker() **?** msg**->**meta.customer\_id : app\_id;

**auto\*** obj **=** Postoffice**::**Get()**->**GetCustomer(app\_id, customer\_id, 5);

*// 调用customer->Accept转交消息*

obj**->**Accept(**\***msg);

}

下一章见《[ps-lite源码解析(3)：顾客、工人和服务器](https://zhuanlan.zhihu.com/p/467697961)》，讲解pslite中的Customer, Worker和Server三个类，介绍pslite最常用的两个动作Push/Pull是如何实现的

- END –

[**呆头鹅**](https://www.zhihu.com/people/d79ef4a3cabc6a0082db86184e08b447)

关于ProcessBarrierCommand中同步的问题，我感觉这里应该是实现了同步，一个worker节点上有几个worker线程就对应几个customer发几次barrier请求。在scheduler节点上这几个worker线程共用同一个node id（node id应该对应post office 这些线程也公用一个post office）但是被算作不同的node，因此可以根据收到的barrier请求数是否等于节点数判断是否完成同步， 如果完成同步, schdeuler节点会向这一个post office发送一个barrier请求，这一个post office把所有的customer的阻塞解除掉，完成同步。

2022-04-13

​回复​1

[](https://www.zhihu.com/people/2f5cf415549e054983a504f0a38632df)

[**Apricity**](https://www.zhihu.com/people/2f5cf415549e054983a504f0a38632df)

作者和上面的评论大概忽略了 shared\_node\_mapping\_、init\_stage\_ 这些东西。每个线程或说 customer 都有自己的 node\_id，并且分别作为一个 worker 计入 num\_workers\_，所以根据 node\_id 判数量是 ok 的。

启动过程其实没讲清楚。PostOffice 和 Van 是每个进程有一个，保存节点的信息和与其它节点的连接状态，且由各个 customer 共享。每个 customer 启动时都要调 Postoffice::Start 且传入自己的 customer\_id，并通过 init\_stage\_ 保证 customer 之间共享的节点信息或工作线程只会被读取或启动一次。之后会调 Van::Start(customer\_id)，即每个 customer 都会发一个独立的 ADD\_NODE 信息给 scheduler，然后死循环等 ready 为 true。

在 scheduler 累计收到 num\_workers + num\_servers 个 ADD\_NODE 后，为它们分配 node\_id。如果一个 node 有多个 customer，则枚举到第一个 customer 时，connect 该节点，枚举到后面 customer 时不必再 connect，而是把它保存到 shared\_node\_mapping\_ 中。后面再给其它节点发 ADD\_NODE 信息时，也不必给 shared\_node\_mapping\_ 里的节点发，因为一个物理节点只需要收到一次信息就可以了（节点的信息是 customer 共享的）。

worker/server 收到 scheduler 的 ADD\_NODE 后就可以连接其他节点，然后设置 ready 为 true 唤醒所有 customer，然后系统开始运行。

01-10

本系列是对Parameter Server的经典实现ps-lite的源码解析，共分4章：

1. [第一章](https://zhuanlan.zhihu.com/p/467650462)，算是热身，先讲一下我们为什么需要Parameter Server，然后再分析一下pslite的入门demo程序，最后介绍pslite中的几个基本概念，为后续阅读打好基础。
2. [第二章](https://zhuanlan.zhihu.com/p/467693949)，讲解pslite中的PostOffice和Van两个类，涉及pslite中节点管理、消息分发、多机同步等方面的知识。
3. 第三章（*本章*），讲解pslite中的Customer, Worker和Server三个类，介绍pslite最常用的两个动作Push/Pull是如何实现的。本章标题是向电影《锅匠，裁缝，士兵，间谍》致敬。
4. [第四章](https://zhuanlan.zhihu.com/p/467702941)，掌握了pslite的运行原理后，让我们看看基于pslite如何实现一个分布式的FM算法。

另外，用文字来描述代码流程，是苍白无力的。 为此，我提供了一份ps-lite源码思维导图。感兴趣的同学，**关注我的微信公众号“推荐道”，回复pslite领取**。下载免费版MindMaster打开后，读我的解析文章与看思维导图相结合，更容易理解

**Customer：消息中介**

如前所述，Customer其实就是个中介。worker和server为了更好地专心业务工作，将“在邮局排队等消息”的工作外包给了它。Customer从Van手中接过信件后，转手就交给它的顾主，worker或server。

**重要成员**

* ThreadsafePQueue recv\_queue\_ + recv\_thread\_：队列+线程的组合，很容易让我们想到“Producer+Consumer”模式。Customer的上游（Van）把消息插入队列，recv\_thread\_线程循环从队列的另一端拉取消息并处理之。
* recv\_handle\_: 由创建Customer的雇主(worker/server)指定的消息处理函数。因为Customer只是一个中介，本身并不知道如何处理消息，一接到消息，就通过recv\_handle\_转交给雇主去处理。
* vec<pair<int, int>> tracker\_：记录每个request的完成情况
  + 训练时，worker所需的参数可能分布在若干server上，所以worker pull一次要访问多个节点
  + vector内部元素是一个pair，pair[0]记录一共向多少节点发出了请求，pair[1]记录多少节点已经回复。当二者相等时，代表请求完成。
  + tracker\_的下标相当于request\_id，也是NewRequest函数的返回值。

**构造函数**

Customer**::**Customer(**int** app\_id, **int** customer\_id, **const** Customer**::**RecvHandle**&** recv\_handle)

**:** app\_id\_(app\_id), customer\_id\_(customer\_id), recv\_handle\_(recv\_handle) {

Postoffice**::**Get()**->**AddCustomer(**this**);

recv\_thread\_ **=** std**::**unique\_ptr**<**std**::thread>**(**new** std**::thread**(**&**Customer**::**Receiving, **this**));

}

* 首先是两大id
  + app\_id其实就是database id，表明这个customer对哪类消息感兴趣。比如，在FM场景下，是对一阶权重还是embedding。
  + customer\_id才是这个customer自身的id。如前所述，一个worker node可能包含多个worker thread，它们各自有独立的customer\_id。
* 输入参数recv\_handle，由该customer的雇主传入。因为customer只是个消息中介，它自己并不知道该如何处理收到的消息，因此只能通过recv\_handle将消息转交给
* 构造函数中，通过调用Postofficer->AddCustomer将自己向邮局注册。这样当Van有了消息时，才知道要投递给哪个customer。（见[[ps-lite#ProcessDataMsg：处理数据消息|Van:ProcessDataMsg]]）
* 启动线程recv\_thread\_，运行Customer::Receiving

**消息相关API**

***Accept：接受消息***

如前所述，Customer的消息处理遵循“producer+consumer”模式。Accept就是producer接口，被上游Van->ProcessDataMsg调用。逻辑非常简单，就是插入消息队列，等待consumer线程拉取处理。cpp

**inline** **void** **Accept**(**const** Message**&** recved) {recv\_queue\_.Push(recved);}

***Receiving：处理消息***

这个函数实现"producer+consumer"模式中的consumer角色。

**void** Customer**::**Receiving() {

**while** (true) {

Message recv;

recv\_queue\_.WaitAndPop(**&**recv);*//从队列中取消息*

......

recv\_handle\_(recv);*//调用雇主指定的函数处理消息*

**if** (**!**recv.meta.request) {*//如果是回复*

......

*// recv.meta.timestamp其实就是request\_id，给某次request的“回复节点”数目加1*

tracker\_[recv.meta.timestamp].second**++**;

tracker\_cond\_.notify\_all();*//通知WaitRequest，有可能会解锁它*

}

}

}

循环从消息队列中拉取消息，拿到后做两件事：

1. 调用雇主(worker/server)提供的recv\_handle\_处理消息，完成业务相关逻辑
2. 更新某次请求（由recv.meta.timestamp指定，即某次请求的request\_id）的“已经回复”节点数，并由tracker\_cond\_通知监听方，可能会解锁阻塞中的WaitRequest函数。

**请求相关API**

***NewRequest***

主要是在KVWorker生成pull请求时被调用，在tracker\_中占一位，并且设置“请求节点总数”。

**int** Customer**::**NewRequest(**int** recver) {

......

*// recver一般是一个node group id，比如KServerGroup*

*// num是该group下一共有多少节点*

**int** num **=** Postoffice**::**Get()**->**GetNodeIDs(recver).size();

*// tracker\_占一位*

*// pair[0]=num，代表向recver group中的所有节点发送了请求*

*// pair[1]=0，代表尚未收到任何回复*

tracker\_.push\_back(std**::**make\_pair(num, 0));

**return** tracker\_.size() **-** 1;*//返回这个请求的下标，作为request\_id*

}

***WaitRequest***

就是用tracker\_cond\_阻塞等待“请求节点数”和“回复节点数”相等。而tracker\_cond\_是在Customer::Receiving每次接受到消息时通知一下。

**void** Customer**::**WaitRequest(**int** timestamp) {

*// timestamp，其实就是某次请求的request\_id，就是NewRequest返回的tracker中的下标*

......

tracker\_cond\_.wait(lk, [**this**, timestamp]{

**return** tracker\_[timestamp].first **==** tracker\_[timestamp].second;*//请求节点数 == 回复节点数？*

});

}

注意这个阻塞是“单机阻塞”，适用于比如，worker必须在从server拉取完最新参数之后，才能开始下一轮的训练。由于pull是异步的，所以worker需要调用WaitRequest阻塞等待那个pull request完成。多机之间的阻塞同步，要用PostOffice:Barrier。

**KVServer**

**构造函数**

KVServer(**int** app\_id) **:** SimpleApp() {

obj\_ **=** **new** Customer(app\_id, app\_id, std**::**bind(**&**KVServer**<**Val**>::**Process, **this**, \_1));

}

从以上代码中，我们可以看到：

* 构建Customer时，第一参数（app\_id）和第二参数（customer\_id）设置的都是app\_id。说明在server端，一个server只有一个customer，不会有多个server thread并发处理。
* 把KVServer:Process作为消息处理接口赋予Customer, 即customer中的recv\_thread\_收到消息后，就交给KVServer:Process来处理。

**Process**

收到消息后（由Customer转交的，而Customer的消息来自Van），做一些格式上的转化，再外包给用户自定义函数request\_handle\_去处理。

* request\_handle\_通过KVServer：set\_request\_handle由用户设定
* 简单demo可以参考KVServerDefaultHandle
* 实际场景下，用户自定义的request\_handle\_要实现各种SGD变体，汇总梯度，更新参数。

**void** KVServer**<**Val**>::**Process(**const** Message**&** msg) {

......

*// -------------- 填充元信息*

KVMeta meta;

meta.cmd **=** msg.meta.head;

meta.push **=** msg.meta.push;

meta.pull **=** msg.meta.pull;

meta.sender **=** msg.meta.sender;

meta.timestamp **=** msg.meta.timestamp;

meta.customer\_id **=** msg.meta.customer\_id;

*// -------------- 填充数据部分KVPairs*

KVPairs**<**Val**>** data;

**int** n **=** msg.data.size();

**if** (n) {

CHECK\_GE(n, 2);

data.keys **=** msg.data[0];

data.vals **=** msg.data[1];

**if** (n **>** 2) {

CHECK\_EQ(n, 3);

data.lens **=** msg.data[2];

CHECK\_EQ(data.lens.size(), data.keys.size());

}

}

*// -------------- 其实KVServer自己也不知道该如何处理消息，*

*// 只好再次外包给用户自定义的消息处理函数*

request\_handle\_(meta, data, **this**);

}

**Response**

一个辅助函数，多在用户自定义的request\_handle\_被调用，回复KVWorker。demo可参考KVServerDefaultHandle。

*// KVMeta& req：当初request的元信息*

*// KVPairs<Val>& res:要返回给kvworker的response数据*

**void** KVServer**<**Val**>::**Response(**const** KVMeta**&** req, **const** KVPairs**<**Val**>&** res) {

Message msg;

*// ------------ 填充回复的元数据*

msg.meta.app\_id **=** obj\_**->**app\_id();

msg.meta.customer\_id **=** req.customer\_id;*//返回给worker node上的哪个worker thread*

msg.meta.request **=** false;*//是一次response*

msg.meta.push **=** req.push;

msg.meta.pull **=** req.pull;

msg.meta.head **=** req.cmd;

msg.meta.timestamp **=** req.timestamp;

msg.meta.recver **=** req.sender;*//回复的recver就是当初请求的sender*

*// ------------ 填充回复的数据*

**if** (res.keys.size()) {

msg.AddData(res.keys);

msg.AddData(res.vals);

**if** (res.lens.size()) {

msg.AddData(res.lens);

}

}

*// ------------ 交给邮车去发送*

Postoffice**::**Get()**->**van()**->**Send(msg);

}

**KVWorker**

**成员变量**

* map<int, vec<KVPairs>> recv\_kvs\_
  + key是request\_id，在pslite中被填充成message.meta.timestamp。其实就是Customer:tracker\_中的下标，被Customer:NewRequst分配。
  + value是该请求涉及的所有server返回数据的集合。
* map<int, Callback> callbacks\_
  + key是request\_id，在pslite中被填充成message.meta.timestamp
  + value是当某次request完成时（i.e., 所有server都回复了）需要执行的回调函数

**构造函数**

KVWorker(**int** app\_id, **int** customer\_id) **:** SimpleApp() {

slicer\_ **=** std**::**bind(**&**KVWorker**<**Val**>::**DefaultSlicer, **this**, \_1, \_2, \_3);

obj\_ **=** **new** Customer(app\_id, customer\_id, std**::**bind(**&**KVWorker**<**Val**>::**Process, **this**, \_1));

}

* 先是创建了slicer\_。worker一次请求的keys可能分布在不同的server上，slicer\_被用于将请求的每个key划分到拥有它的server上。
* 可以看到与KVServer创建customer不同，KVWorker创建customer的app\_id和customer\_id是分开的。也就是说，一个worker节点可以有多个worker thread，它们向server读写的是同一类数据，所以app\_id相同；但是分属不同线程，所以各自拥有独立的customer\_id。
* 把KVWorker:Process作为消息处理接口赋予Customer, 即customer中的recv\_thread\_收到消息后，就交给KVWorker:Process来处理。

**Send**

无论是Push还是Pull，都有一些共同的逻辑要实现在Send函数中。

**void** KVWorker**<**Val**>::**Send(**int** timestamp, **bool** push, **bool** pull, **int** cmd, **const** KVPairs**<**Val**>&** kvs) {

*// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 决定要向哪些server发送请求*

SlicedKVs sliced;*// 存储分配结果*

slicer\_(kvs, Postoffice**::**Get()**->**GetServerKeyRanges(), **&**sliced);

*// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 有些server不包含本次请求要求的keys，提前处理*

**int** skipped **=** 0;*// 本次请求不涉及的servers的总数*

**for** (size\_t i **=** 0; i **<** sliced.size(); **++**i) {

**if** (**!**sliced[i].first) **++**skipped;

}

*// 内部不过是tracker\_[timestamp].second += skipped*

*// 假设这些不涉及的servers已经返回了*

obj\_**->**AddResponse(timestamp, skipped);

......

*// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 向所有涉及到的server发送请求*

**for** (size\_t i **=** 0; i **<** sliced.size(); **++**i) {

**const** **auto&** s **=** sliced[i];

**if** (**!**s.first) **continue**;*//本次请求不涉及该节点*

Message msg;

msg.meta.app\_id **=** obj\_**->**app\_id();

msg.meta.customer\_id **=** obj\_**->**customer\_id();

msg.meta.request **=** true;

msg.meta.push **=** push;

msg.meta.pull **=** pull;

msg.meta.head **=** cmd;

msg.meta.timestamp **=** timestamp;

msg.meta.recver **=** Postoffice**::**Get()**->**ServerRankToID(i);

msg.meta.priority **=** kvs.priority;

**const** **auto&** kvs **=** s.second;*//分配到当前节点上的key-value pairs*

**if** (kvs.keys.size()) {

msg.AddData(kvs.keys);

msg.AddData(kvs.vals);

**if** (kvs.lens.size()) {

msg.AddData(kvs.lens);

}

}

Postoffice**::**Get()**->**van()**->**Send(msg);

}

}

**AddPullCB**

由于一次pull要涉及到多台server，而接下来的动作要等到所有server都回复了才能进行，所以AddPullCB的职能就是注册一个**等所有server都回复后才执行**的回调函数。

**int** KVWorker**<**Val**>::**AddPullCB(

*// C\* vals和D\* lens指向由调用者指定的结构体。*

*// 等所有server都返回后，从所有server拉来的数据*

**const** SArray**<**Key**>&** keys, C**\*** vals, D**\*** lens, **int** cmd,

*// Callback& cb代表在所有server回复后要执行的额外的回调*

*// 一般我们都是在pull后就立刻阻塞等待，所以cb一般为空*

**const** Callback**&** cb) {

*// \*\*\*\*\*\*\*\*\*\*\*\*\*\* 创建request，返回的ts是该request\_id*

**int** ts **=** obj\_**->**NewRequest(kServerGroup);

*// \*\*\*\*\*\*\*\*\*\*\*\*\*\* 添加callback，等所有server都回复后再执行*

AddCallback(ts, [**this**, ts, keys, vals, lens, cb]() **mutable** {

......

*// 容纳ts（即request\_id）所接受数据的缓冲区*

**auto&** kvs **=** recv\_kvs\_[ts];

......

*// total\_keys是根据kvs统计出来的接收到的key的总数*

*// keys是当初请求的所有keys，检查二者是否相等*

......

CHECK\_EQ(total\_key, keys.size()) **<<** "lost some servers?";

*// \*\*\*\*\*\*\*\*\*\*\*\*\*\* 将所有server返回的数据，合并，填充到用户指定的输出位置*

*// vals和lens都指向调用者传入的结构体*

*// p\_vals和p\_lens都是指向输出区的指针*

Val**\*** p\_vals **=** vals**->**data();

......

p\_lens **=** lens**->**data();

......

*// 遍历从各台server接收到的内容，填充到输出区p\_vals和p\_lens*

**for** (**const** **auto&** s : kvs) {

memcpy(p\_vals, s.vals.data(), s.vals.size() **\*** **sizeof**(Val));

p\_vals **+=** s.vals.size();

**if** (p\_lens) {

memcpy(p\_lens, s.lens.data(), s.lens.size() **\*** **sizeof**(**int**));

p\_lens **+=** s.lens.size();

}

}

......

recv\_kvs\_.erase(ts);*//清空本次请求的接收缓冲区*

......

**if** (cb) cb();*// 如果有额外的callback，执行之*

});

**return** ts;

}

* C\* vals和D\* lens 都是输出，指向由调用者指定的结构体。等所有server都返回后，将从所有server拉来的数据填充到vals和lens。
* 先调用NewRequest创建了一个请求，返回的ts就是本次请求的request\_id，将作为追踪本次请求完成情况的凭证。
* 添加的回调函数，只有在所有server都返回后才调用。触发条件见KVWorker:Process。这时auto& kvs = recv\_kvs\_[ts]包含所有server返回的数据。
* 回调中先做了一个sanity check，即所有返回的keys的个数应该和请求的keys的个数相等。
  + 其实这里我也是有疑问的，因为ps server为了节省空间，一般都有参数的准入和逐出功能，因此请求某个key在ps server端找不到，也属正常。
  + 当然对于这种ps server找不到key的情况，pslite也能支持，response中的key还必须与request keys保持一致，只要lens相应位置为0即可，表示这个key返回空。
* 然后就是将各server返回的数据，合并，填充至用户指定的结构体
* 最后，如果提供了额外的回调cb，执行之。不过，如果选择Pull之后就Wait，同步阻塞等待Pull完成，没必要再提供额外回调。

**Pull & ZPull**

这两个函数差不多，只不过ZPull（Z代表zero-copy）实现了“零拷贝”，效率更高。

**int** **ZPull**(**const** SArray**<**Key**>&** keys,*//要拉取哪些keys*

SArray**<**Val**>\*** vals,*// 输出参数，指向存放pull结果的位置*

SArray**<int>\*** lens **=** **nullptr**,*// 输出参数，指向存放pull结果的位置*

**int** cmd **=** 0,

**const** Callback**&** cb **=** **nullptr**,

**int** priority **=** 0) {

*// 内部新建了一个请求，request\_id为ts*

*// 并且注册了一个回调，等所有server都回复后，把pull的结果存放在vals和lens*

**int** ts **=** AddPullCB(keys, vals, lens, cmd, cb);

KVPairs**<**Val**>** kvs;

kvs.keys **=** keys;

kvs.priority **=** priority;

*// 内部会把请求的keys划分到合适的server上，发送请求*

Send(ts, false, true, cmd, kvs);

**return** ts;

}

**Push & ZPush**

两个函数一样，Push只不过委托ZPush执行

**int** **ZPush**(**const** SArray**<**Key**>&** keys,*//要push的keys*

**const** SArray**<**Val**>&** vals,*//要push的vals*

**const** SArray**<int>&** lens **=** {},*//要push的每个key对应的value的长度*

**int** cmd **=** 0,

**const** Callback**&** cb **=** **nullptr**,*//额外回调*

**int** priority **=** 0) {

**int** ts **=** obj\_**->**NewRequest(kServerGroup);*//创建请求*

AddCallback(ts, cb);

KVPairs**<**Val**>** kvs;

kvs.keys **=** keys;

kvs.vals **=** vals;

kvs.lens **=** lens;

kvs.priority **=** priority;

*// 内部会决定每个key要push到哪台server去*

Send(ts, true, false, cmd, kvs);

**return** ts;*// request\_id*

}

**接收数据**

收到消息后（由Customer转交的，而Customer的消息来自Van）

1. 把message里的数据填充到KVPairs，插入ts（即某次request\_id）指定的接收数据缓冲区recv\_kvs\_[ts]
2. 如果所有server都回复了，通过ts找到该request完成时的回调，执行之。

**void** KVWorker**<**Val**>::**Process(**const** Message**&** msg) {

......

**int** ts **=** msg.meta.timestamp;*// timestamp存储的是request\_id*

**if** (msg.meta.pull) {

KVPairs**<**Val**>** kvs;

kvs.keys **=** msg.data[0];

kvs.vals **=** msg.data[1];

**if** (msg.data.size() **>** (size\_t)2) {

kvs.lens **=** msg.data[2];

}

mu\_.lock();

recv\_kvs\_[ts].push\_back(kvs);*//某个server的回复插入缓冲区*

mu\_.unlock();

}

*// 如果所有server都返回了结果，调用回调，如果是pull的话，就是AddPullCB指定的回调*

*// 注意：这里之所以写obj\_->NumResponse(ts) == Postoffice::Get()->num\_servers() - 1*

*// 而不是obj\_->NumResponse(ts) == Postoffice::Get()->num\_servers()*

*// 是因为Customer::Receiving中*

*// 先执行recv\_handle\_(recv), recv\_handle\_就是这里的kvworker:process*

*// 再执行tracker\_[recv.meta.timestamp].second++*

*// 所以，在执行kvworker:process时，NumResponse还没有更新，*

*// 所以，必须和num\_servers-1比较，而不是和num\_servers比较*

**if** (obj\_**->**NumResponse(ts) **==** Postoffice**::**Get()**->**num\_servers() **-** 1) {

RunCallback(ts);

}

}

**总结**

简单总结一下pslite中的调用流程：

* 启动
  + 每个节点上的PostOffice->Start启动，并调用Van->Start
  + 在Van->Star中，
    - Van启动了消息处理线程receiver\_thread\_
    - worker/server节点向scheduler发送ADD\_NODE消息注册自己。
  + scheduler在Van:ProcessAddNodeCommandAtScheduler中处理worker/server的ADD\_NODE请求。等所有worker和server节点都注册了之后，将本集群内所有节点的名单广播给所有worker+server。
  + 各worker+server节点，收到scheduler的ADD\_NODE的回复后，用scheduler发来的全局节点名单更新本地名单。
* 传递数据
  + 创建KVWorker，worker内部创建了customer，并向PostOffice注册
  + 创建KVServer，server内部创建了customer，并向PostOffice注册
  + Van中的消息处理线程receiver\_thread\_，运行Van:Receiving，循环接收消息。如果接到数据信息，就调用Van:ProcessDataMsg
  + Van:ProcessDataMsg根据消息中的app\_id+customer\_id找到相应的Customer，调用customer->Accept转交消息。
  + Customer内部也有一个接收线程recv\_thread\_和消息队列recv\_queue\_，Accept只是将消息插入消息队列
  + Customer:Receiving运行在接收线程上，循环从消息队列中拉取消息，通过调用者预先设置的接口，将消息转交给它的雇主（worker/server）
  + KVServer的消息处理函数是KVServer:Process，其实也是调用用户自定义的接口实现真正的处理逻辑，比如使用SGD更新权重。
  + KVWorker的消息处理函数是KVWorker:Process，由于worker的一次pull请求会涉及多台server，所以process要将某一台server的response数据缓存。等到所有server都回复后，将所有server的回复数据，合并拷贝至调用者指定的地址。

下一章见《[ps-lite源码解析(4)：基于ps-lite实现分布式FM](https://zhuanlan.zhihu.com/p/467702941)》, 让我们看看基于pslite如何实现一个分布式的FM算法。

- END -

本系列是对Parameter Server的经典实现ps-lite的源码解析，共分4章：

1. [第一章](https://zhuanlan.zhihu.com/p/467650462)，算是热身，先讲一下我们为什么需要Parameter Server，然后再分析一下pslite的入门demo程序，最后介绍pslite中的几个基本概念，为后续阅读打好基础。
2. [第二章](https://zhuanlan.zhihu.com/p/467693949)，讲解pslite中的PostOffice和Van两个类，涉及pslite中节点管理、消息分发、多机同步等方面的知识。
3. [第三章](https://zhuanlan.zhihu.com/p/467697961)，讲解pslite中的Customer, Worker和Server三个类，介绍pslite最常用的两个动作Push/Pull是如何实现的。
4. 第四章（*本章*），掌握了pslite的运行原理后，让我们看看基于pslite如何实现一个分布式的FM算法。

另外，用文字来描述代码流程，是苍白无力的。 为此，我提供了一份ps-lite源码思维导图。感兴趣的同学，**关注我的微信公众号“推荐道”，回复pslite领取**。下载免费版MindMaster打开后，读我的解析文章与看思维导图相结合，更容易理解。

本章要分析的分布式FM源码见[**xflow**](https://link.zhihu.com/?target=https%3A//github.com/xswang/xflow)。这个github项目有如下看点：

* 相比于分布式的LR，xflow中的分布式FM中每个key对应的是一个向量，而非一个浮点数。在embedding大行其道的今天，更具实战价值。
* FM里既有一阶权重，也有embedding，两类数据，需要两类处理方式。
* 强调一下，这套代码中的分布式训练采用“**异步**”模式，不要被代码中的Wait所迷惑。

**Server代码**

定义在server.h中，可以看到在构造函数中先建立并启动了了对一阶权重w和对embedding的两个KVServer。注意，由于要读写两类数据，所以两个KVServer分别拥有独立app\_id，针对一阶权重w的app\_id=0，针对embedding "v"的app\_id=1。

Server() {

server\_w\_ **=** **new** ps**::**KVServer**<float>**(0);*// app\_id=0*

server\_w\_**->**set\_request\_handle(FTRL**::**KVServerFTRLHandle\_w());

server\_v\_ **=** **new** ps**::**KVServer**<float>**(1);*// app\_id=1*

server\_v\_**->**set\_request\_handle(FTRL**::**KVServerFTRLHandle\_v());

}

具体逻辑，即接收到worker发来的梯度后，用FTRL算法更新一阶权重"w"和embedding "v"，分别实现在KVServerFTRLHandle\_w和KVServerFTRLHandle\_v中。两个类的逻辑差不多，KVServerFTRLHandle\_w只不过是将一阶权重w看成embed\_dim=1的embedding。所以接下来重点看KVServerFTRLHandle\_v是如何更新embedding的。

**Embedding存储**

ps server用KVServerFTRLHandle\_v:map<ps::Key, ftrlentry\_v> store来存储各feature的embedding。其中使用到的ftrlentry\_v定义如下。

**typedef** **struct** **FTRLEntry\_v** {

FTRLEntry\_v(**int** k **=** v\_dim) {*// v\_dim在代码里定义成10*

w.resize(k, 0.0);

n.resize(k, 0.0);

z.resize(k, 0.0);

}

std**::**vector**<float>** w;*//embedding向量*

std**::**vector**<float>** n;*//FTRL要用到的中间变量*

std**::**vector**<float>** z;*//FTRL要用到的中间谜题*

} ftrlentry\_v;

从以上代码可以看到，为了使用FTRL算法来更新权重，对于每个feature，server端不仅要存储其embedding，还要存储n和z两个中间变量。在推荐系统这样一个动辄特征量就上亿的场景下，这会占用极大量的内存。因此，**现在很多分布式优化算法在下功夫，减少中间变量存储**。

**KVServerFTRLHandle\_v**

**struct** **KVServerFTRLHandle\_v** {

**void** **operator**()(**const** ps**::**KVMeta**&** req\_meta,**const** ps**::**KVPairs**<float>&** req\_data,ps**::**KVServer**<float>\*** server) {

size\_t keys\_size **=** req\_data.keys.size();

ps**::**KVPairs**<float>** res;

**if** (req\_meta.push) {

size\_t vals\_size **=** req\_data.vals.size();

*// 做sanity check，因为每个embedding都有v\_dim那么长，所以vals\_size == keys\_size \* v\_dim*

CHECK\_EQ(keys\_size, vals\_size **/** v\_dim);

} **else** {

res.keys **=** req\_data.keys;

res.vals.resize(keys\_size **\*** v\_dim);

}

*// \*\*\*\*\*\*\*\*\*\*\*\*\*\* 遍历请求的每个key*

**for** (size\_t i **=** 0; i **<** keys\_size; **++**i) {

ps**::**Key key **=** req\_data.keys[i];

*// 如果第一次访问某个key，随机初始化*

*// 更先进的算法是设立准入规则，没必要为只出现一两次的特征浪费存储空间*

**if** (store.find(key) **==** store.end()) {

FTRLEntry\_v val(v\_dim);;

**for** (**int** k **=** 0; k **<** v\_dim; **++**k) {

val.w[k] **=** Base**::**local\_normal\_real\_distribution**<double>**(0.0, 1.0)(Base**::**local\_random\_engine()) **\*** 1e-2;

}

store[key] **=** val;

}

FTRLEntry\_v**&** val **=** store[key];

*// \*\*\*\*\*\*\*\*\*\*\*\*\*\* 遍历embedding的每一位*

**for** (**int** j **=** 0; j **<** v\_dim; **++**j) {

**if** (req\_meta.push) {

*// ---------- 处理push请求，用FTRL算法根据梯度更新embedding*

**float** g **=** req\_data.vals[i **\*** v\_dim **+** j];

**float** old\_n **=** val.n[j];

**float** n **=** old\_n **+** g **\*** g;

val.z[j] **+=** g **-**

(std**::**sqrt(n) **-** std**::**sqrt(old\_n)) **/** alpha **\*** val.w[j];

val.n[j] **=** n;

**if** (std**::**abs(val.z[j]) **<=** lambda1) {

val.w[j] **=** 0.0;*//更新embedding*

} **else** {

**float** tmpr **=** 0.0;

**if** (val.z[j] **>** 0.0) tmpr **=** val.z[j] **-** lambda1;

**if** (val.z[j] **<** 0.0) tmpr **=** val.z[j] **+** lambda1;

**float** tmpl **=** **-**1 **\*** ((beta **+** std**::**sqrt(val.n[j]))**/**alpha **+** lambda2);

val.w[j] **=** tmpr **/** tmpl;*//更新embedding*

}

} **else** {

*// ---------- 处理pull请求，把embedding按位复制进response*

*// vals是将所有key对应的embedding拼接并打平成一个大数组*

res.vals[i **\*** v\_dim **+** j] **=** val.w[j];

}

}

}

server**->**Response(req\_meta, res);

}

**private:**

std**::**unordered\_map**<**ps**::**Key, ftrlentry\_v**>** store;*//存储参数*

};

**Worker代码**

FM的worker端代码定义在fm\_worker.h和fm\_worker.cc中

**构造函数**

FMWorker(**const** **char** **\***train\_file, **const** **char** **\***test\_file) **:**train\_file\_path(train\_file),test\_file\_path(test\_file) {

kv\_w **=** **new** ps**::**KVWorker**<float>**(0);*// app\_id=0*

kv\_v **=** **new** ps**::**KVWorker**<float>**(1);*// app\_id=1*

......

}

同样是由于FM要读写一阶权重w和embeding "v"两种类型的数据，所以在worker侧也定义了两个KVWorker，各自拥有独立的app\_d，针对一阶权重w的app\_id=0，针对embedding "v"的app\_id=1。拥有相同app\_id的worker/server之间才能通信。

**batch\_training：训练入口**

worker侧训练的入口在FMWorker::batch\_training。

**void** FMWorker**::**batch\_training(ThreadPool**\*** pool) {

......

**for** (**int** epoch **=** 0; epoch **<** epochs; **++**epoch) {*//把数据过若干epoch*

xflow**::**LoadData train\_data\_loader(train\_data\_path, block\_size **<<** 20);

train\_data **=** **&**(train\_data\_loader.m\_data);

**int** block **=** 0;

**while** (1) {*// 循环直到将本次的训练数据都读完*

*// 读取一个mini-batch的数据存放在train\_data->fea\_matrix中*

train\_data\_loader.load\_minibatch\_hash\_data\_fread();

**if** (train\_data**->**fea\_matrix.size() **<=** 0) **break**;

*// 把这个mini-batch的训练数据平分到各个thread上*

*// 每个thread分到thread\_size条训练数据*

**int** thread\_size **=** train\_data**->**fea\_matrix.size() **/** core\_num;

gradient\_thread\_finish\_num **=** core\_num;

**for** (**int** i **=** 0; i **<** core\_num; **++**i) {*//遍历启动各训练线程*

**int** start **=** i **\*** thread\_size;*// 当前线程分到的数据的起始位置*

**int** end **=** (i **+** 1)**\*** thread\_size;*// 当前线程分到的数据的截止位置*

*// 启动线程运行FMWorker::update，训练当前mini-batch的局部数据*

pool**->**enqueue(std**::**bind(**&**FMWorker**::**update, **this**, start, end));

}

**while** (gradient\_thread\_finish\_num **>** 0) {*//等待所有thread结束*

usleep(5);

}

**++**block;

}

**if** ((epoch **+** 1) **%** 30 **==** 0) std**::**cout **<<** "epoch : " **<<** epoch **<<** std**::**endl;

train\_data **=** NULL;

}

}

**update：真正执行训练**

**void** FMWorker**::**update(**int** start, **int** end) {*//运行在独立线程中，真正执行一部分数据的训练*

size\_t idx **=** 0;

**auto** all\_keys **=** std**::**vector**<**Base**::**sample\_key**>**();

**auto** unique\_keys **=** std**::**vector**<**ps**::**Key**>**();

**int** line\_num **=** 0;

*// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 遍历分配给自己的数据，重新组织feature*

**for** (**int** row **=** start; row **<** end; **++**row) {

*// sample\_size是第row行样本内部包含的特征个数*

**int** sample\_size **=** train\_data**->**fea\_matrix[row].size();

Base**::**sample\_key sk;

sk.sid **=** line\_num;*// sid，即sample\_id，用于标识特征属于哪条样本*

**for** (**int** j **=** 0; j **<** sample\_size; **++**j) {*//遍历当前样本的每个特征*

idx **=** train\_data**->**fea\_matrix[row][j].fid;

sk.fid **=** idx;*// fid，即feature\_id*

all\_keys.push\_back(sk);

(unique\_keys).push\_back(idx);

}

**++**line\_num;

}

*// 把分配给自己的那部分数据中的所有特征，按feature\_id重新排序*

*// 相同feature\_id，但分属不同sample的feature排列在一起*

std**::**sort(all\_keys.begin(), all\_keys.end(), base\_**->**sort\_finder);

*// 把这部分数据中出现的所有feature\_id去重*

*// 因为向ps server pull的时候，没必要重复pull相同的key*

std**::**sort((unique\_keys).begin(), (unique\_keys).end());

(unique\_keys).erase(unique((unique\_keys).begin(), (unique\_keys).end()),

(unique\_keys).end());

**int** keys\_size **=** (unique\_keys).size();*//去重后的feature\_id个数*

*// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 用去重后的feature\_ids从ps server拉取最新的参数*

*// 拉取一阶权重"w"*

**auto** w **=** std**::**vector**<float>**();

kv\_w**->**Wait(kv\_w**->**Pull(unique\_keys, **&**w));

*// 定义空间，等待容纳各feature的一阶权重'w'上的导数*

**auto** push\_w\_gradient **=** std**::**vector**<float>**(keys\_size);

*// 拉取embedding "v"*

**auto** v **=** std**::**vector**<float>**();

kv\_v**->**Wait(kv\_v**->**Pull(unique\_keys, **&**v));

*// 定义空间，等待容纳各feature的embedding上的导数*

*// 由于embedding是个向量，所以需要定义的空间是keys\_size(去重后有多少feaute)\*v\_dim\_(每个embedding的长度)*

**auto** push\_v\_gradient **=** std**::**vector**<float>**(keys\_size **\*** v\_dim\_);

*// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 前代*

**auto** loss **=** std**::**vector**<float>**(end **-** start);

**auto** v\_sum **=** std**::**vector**<float>**(end **-** start);

*// loss这个名字取得不好，其实里面存储的是loss针对final logit的层数*

calculate\_loss(w, v, all\_keys, unique\_keys, start, end, v\_sum, loss);

*// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 回代*

*// 输出push\_w\_gradient: loss对各feature的一阶权重'w'上的导数*

*// 输出push\_v\_gradient: loss对各feature的embedding上的导数*

calculate\_gradient(all\_keys, unique\_keys, start, end, v, v\_sum, loss,

push\_w\_gradient, push\_v\_gradient);

*// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 向Sever push梯度，让server端更新优化参数*

*// 注意！！！这里是每个worker thread各push各自的，完全没有与其他worker同步，因此这里实现的是异步模式。*

kv\_w**->**Wait(kv\_w**->**Push(unique\_keys, push\_w\_gradient));

kv\_v**->**Wait(kv\_v**->**Push(unique\_keys, push\_v\_gradient));

**--**gradient\_thread\_finish\_num;*//代表有一个线程完成了*

}

***同步还是异步？***

需要注意的是，代码最后的kv\_w->Wait(kv\_w->Push和kv\_v->Wait(kv\_v->Push，**尽管调用了Wait，但并不是在实现Synchronous SGD**。这个distributed FM的实现，本质上仍然是**Asynchronous SGD**。 如前所述，Wait只作用于单机，这里只是等待异步的Push完成，但是这个worker process并没有等待其他worker process，就开始训练下一个batch，即，没有多机间的协调。（这个代码写得有点复杂，其实代码中是将一个worker process中的多个worker thread进行了同步，但是多个worker process之间仍然缺乏同步）

如果要基于pslite实现同步SGD，还有点复杂：

* worker侧，各worker在kv->Wait(kv->Push之后，还要调用Postoffice::Get()->Barrier(ps::kWorkerGroup);，让所有worker节点同步
* server侧，用户自定义的消息处理handle中还要开辟额外的缓冲区，缓存各worker发来的梯度。等待所有worker都发来梯度之后，先平均梯度，再更新权重。

而现代针对推荐系统优化的PS，往往需要针对embedding用异步更新，面对dnn weight用同步更新。想想就复杂，此间先按下不表。

**前代 & 回代**

前代发生在FMWorker::calculate\_loss，回代发生在FMWorker::calculate\_gradient。

理解两个函数的关键在于，**记住这两个函数的调用者，已经将训练数据中的所有feature信息（代码里的sample\_key，记录某feature的feature\_id和所属sample），根据feature\_id排好序了，拥有相同feature\_id但分属不同sample的feature信息（i.e., sample\_key）排列在一起**。

这么做的目的，是为了能够高效访问拉来的数据。**由于具备相同feature\_id的特征信息排列在一起，那个feature\_id对应的一阶权重"w"和embedding "v"，只需要查找并提取一次**（由于拉的数据也是按照key排好的序的，查找都免了，顺序访问就好），**避免根据feature\_id在拉来数据中反复查找带来的时间浪费**。

两个函数的结构非常类似，只看FMWorker::calculate\_loss，另一个留待读者自己查看源码。

**void** FMWorker**::**calculate\_loss(std**::**vector**<float>&** w,

std**::**vector**<float>&** v,

std**::**vector**<**Base**::**sample\_key**>&** all\_keys,

std**::**vector**<**ps**::**Key**>&** unique\_keys,

size\_t start, size\_t end,

std**::**vector**<float>&** v\_sum,

std**::**vector**<float>&** loss) {

*// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 每条样本，累加一阶权重*

**auto** wx **=** std**::**vector**<float>**(end **-** start);

**for** (**int** j **=** 0, i **=** 0; j **<** all\_keys.size(); ) {

*// j: 遍历每个feature信息 （sample\_key）的游标*

*// i: 遍历拉下来的每个feature一阶权重的游标*

size\_t allkeys\_fid **=** all\_keys[j].fid;

size\_t weight\_fid **=** (unique\_keys)[i];

**if** (allkeys\_fid **==** weight\_fid) {*//还使用当前的w*

*// 把权重w[i]累加到第[all\_keys[j].sid]个样本的logit上*

wx[all\_keys[j].sid] **+=** (w)[i];

**++**j;

} **else** **if** (allkeys\_fid **>** weight\_fid) {*//移到下一个w*

**++**i;

}

}

*// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 每条样本，累加所有embedding的两两交叉*

**auto** v\_pow\_sum **=** std**::**vector**<float>**(end **-** start);

**for** (size\_t k **=** 0; k **<** v\_dim\_; k**++**) {*//遍历embedding的各位*

**for** (size\_t j **=** 0, i **=** 0; j **<** all\_keys.size(); ) {

*// j: 遍历每个feature信息 （sample\_key）的游标*

*// i: 遍历拉下来的每个feature一阶权重的游标*

size\_t allkeys\_fid **=** all\_keys[j].fid;

size\_t weight\_fid **=** unique\_keys[i];

**if** (allkeys\_fid **==** weight\_fid) {*//还使用当前embedding*

size\_t sid **=** all\_keys[j].sid;

**float** v\_weight **=** v[i **\*** v\_dim\_ **+** k];

v\_sum[sid] **+=** v\_weight;

v\_pow\_sum[sid] **+=** v\_weight **\*** v\_weight;

**++**j;

} **else** **if** (allkeys\_fid **>** weight\_fid) {*//移到下一个embedding*

**++**i;

}

}

}

**auto** v\_y **=** std**::**vector**<float>**(end **-** start);

**for** (size\_t i **=** 0; i **<** end **-** start; **++**i) {

v\_y[i] **=** v\_sum[i] **\*** v\_sum[i] **-** v\_pow\_sum[i];

}

*// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 每条样本，计算loss对logit的梯度*

**for** (**int** i **=** 0; i **<** wx.size(); i**++**) {*// 遍历每条样本*

**float** pctr **=** base\_**->**sigmoid(wx[i] **+** v\_y[i]);

*// loss这个名字起得太有误导性，loss[i]不是每i条样本上的loss*

*// loss[i]是第i条样本上的loss对第i条样本预测logit的梯度*

loss[i] **=** pctr **-** train\_data**->**label[start**++**];

}

}

- END -