[cxxnet学习笔记1](http://blog.csdn.net/u010333076/article/details/46473015)

分类： [机器学习](http://blog.csdn.net/u010333076/article/category/5584677)2015-06-12 16:22 51人阅读 [评论](http://blog.csdn.net/u010333076/article/details/46473015#comments)(0) [收藏](javascript:void(0);) [举报](http://blog.csdn.net/u010333076/article/details/46473015#report)

cxxnet/src目录是cxxnet部分的源码，在这里有如下几个文件与文件夹

cxxnet\_main.cpp  **io**     local\_main.cpp  **plugin**     **updater**

global.h         **layer**  **nnet**            README.md  **utils**

其中，各部分依赖关系如下：

nnet->updater->layer；

所有模块依赖于global.h和utils；

io是一个独立的模块；

layer是神经网络层的实现，并且定义了前向和后向的propagation；

updater是参数上传模型，它定义了权制上传规则；

nnet将神经网络层结合在一起，组建成一个神经网络；

io是一个用于读入各类数据和预处理的输入模型。

我们将整个计划分为如下几部分展开，每一单行我们用一天时间去完成理解。

local\_main.cpp,global.h,cxxnet\_main.cpp

io

utils

nnet

updater

layer

现在我们从第一行（local\_main.cpp,global.h,cxxnet\_main.cpp）开始。

global.h包含着一些预编译，没什么好说的，local\_main.cpp是main函数所在，其只包含一个接口，指向真正main函数所在地cxxnet\_main.cpp。

在cxxnet\_main.cpp中：

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

  cxxnet::CXXNetLearnTask tsk;

  return tsk.Run(argc, argv);

}

这就是整个程序真正的main函数，实例化了一个CXXNetLearnTask的对象，调用了其run方法。

整个cxxnet\_main的主体是也是类——CXXNetLearnTask的定义与实现，我们来看一下其结构：

public:

 CXXNetLearnTask(void);

 ~CXXNetLearnTask(void);

 inline int Run(int argc, char \*argv[]);

 inline void SetParam(const char \*name , const char \*val);

private:

 inline void Init(void);

 inline int SyncLastestModel(void);

 inline void LoadModel(void);

 inline void SaveModel(void);

 inline nnet::INetTrainer\* CreateNet(void);

 inline void InitIter(IIterator<DataBatch>\* itr,const std::vector< std::pair< std::string, std::string> > &defcfg);

 inline void CreateIterators(void);

 inline void TaskPredict(void);

 inline void TaskGetWeight(void);

 inline void TaskExtractFeature();

 inline void TaskTrain(void);

 inline void CopyModel(void);

 Run函数是其主体，它会在此处调用Init()初始化，其可以调用TaskPredict(), TaskGetWeight(),TaskExtractFeature(),TaskTrain()这四个函数以完成其他具体不同的功能，而主要的计算任务，也是由这四个不同的函数分别启动的。

 不管是上述四种哪一函数的启动，都会涉及到net\_trainer的调用，net\_trainer是nnet模块在CXXNetLearnTask类中的一个实例，它也可以调用nnet模块的不同功能，关于nnet模块我们会在以后详叙。

[cxxnet学习笔记2](http://blog.csdn.net/u010333076/article/details/46504995)

分类： [机器学习](http://blog.csdn.net/u010333076/article/category/5584677)2015-06-15 16:20 59人阅读 [评论](http://blog.csdn.net/u010333076/article/details/46504995#comments)(0) [收藏](javascript:void(0);) [举报](http://blog.csdn.net/u010333076/article/details/46504995#report)

cxxnet——io模块学习

io模块是一个独立模块，它被设计用来读取各类输入数据和完成预处理。

－io使用迭代器模式来处理数据流管道；

－管道可以使用线程寄存器实现多线程。

在io模块中，有如下这些源码文件：

[data.h](https://github.com/dmlc/cxxnet/blob/master/src/io/data.h)

[data.cpp](https://github.com/dmlc/cxxnet/blob/master/src/io/data.cpp)

[image\_augmenter-inl.hpp](https://github.com/dmlc/cxxnet/blob/master/src/io/image_augmenter-inl.hpp)

[image\_recordio.h](https://github.com/dmlc/cxxnet/blob/master/src/io/image_recordio.h)

[inst\_vector.h](https://github.com/dmlc/cxxnet/blob/master/src/io/inst_vector.h)

[iter\_attach\_txt-inl.hpp](https://github.com/dmlc/cxxnet/blob/master/src/io/iter_attach_txt-inl.hpp)

[iter\_augment\_proc-inl.hpp](https://github.com/dmlc/cxxnet/blob/master/src/io/iter_augment_proc-inl.hpp)

[iter\_batch\_proc-inl.hpp](https://github.com/dmlc/cxxnet/blob/master/src/io/iter_batch_proc-inl.hpp)

[iter\_csv-inl.hpp](https://github.com/dmlc/cxxnet/blob/master/src/io/iter_csv-inl.hpp)

[iter\_image\_recordio-inl.hpp](https://github.com/dmlc/cxxnet/blob/master/src/io/iter_image_recordio-inl.hpp)

[iter\_img-inl.hpp](https://github.com/dmlc/cxxnet/blob/master/src/io/iter_img-inl.hpp)

[iter\_mem\_buffer-inl.hpp](https://github.com/dmlc/cxxnet/blob/master/src/io/iter_mem_buffer-inl.hpp)

[iter\_mnist-inl.hpp](https://github.com/dmlc/cxxnet/blob/master/src/io/iter_mnist-inl.hpp)

[iter\_thread\_imbin-inl.hpp](https://github.com/dmlc/cxxnet/blob/master/src/io/iter_thread_imbin-inl.hpp)

[iter\_thread\_imbin\_x-inl.hpp](https://github.com/dmlc/cxxnet/blob/master/src/io/iter_thread_imbin_x-inl.hpp)

[iter\_thread\_iminst-inl.hpp](https://github.com/dmlc/cxxnet/blob/master/src/io/iter_thread_iminst-inl.hpp)

data.h是其提供给外部的接口，其中声明了一个模版类IIterator和其他几种数据结构以及一个DataBatch迭代器创建函数声明，如下所示：

template<typename DType> class IIterator {};

struct DataInst {};

//一种单独的数据实例

struct SparseInst {};

//一种存储于稀疏容器中的稀疏数据类型

struct DataBatch {};

//一种用于迭代器的常见批量数据类型。

IIterator<DataBatch> \*CreateIterator(const std::vector<std::pair<std::string, std::string> > &cfg);

//从配置函数创建迭代器

IIterator是一个典型的迭代器类型，它定义了几个虚函数交由子类实现：

  virtual void SetParam(const char \*name, const char \*val) = 0;

//设置参数

  virtual void Init(void) = 0;

//初始化迭代器

  virtual void BeforeFirst(void) = 0;

//在第一项前设置

  virtual bool Next(void) = 0;

//迭代器游标移动到下一项

  virtual const DType &Value(void) const = 0;

//获得当前数据

在data.cpp中，实现了data.h中的CreateIterator，该方法返回迭代器实例。

在image\_augmenter-inl.hpp中，定义了一个类ImageAugmenter，该类是图像增强的辅助类。

在image\_recordio.h中，定义了一个结构体ImageRecordIO，该结构体用来存储图像IO记录。

在inst\_vector.h中，定义了两个类，一个是模版类TensorVector，它是张量序列容器，在内存中紧凑存储，另一个类是InstVector，它是实例容器，可以以一种高效的方式存储非均匀形状数据实例。

在iter\_attach\_txt-inl.hpp中，定义了一个IIterator<DataBatch>的子类AttachTxtIterator，该迭代器用来读取存储于txt文件中的额外数据。

在iter\_augment\_proc-inl.hpp中，该单元用于数据增强，定义了一个IIterator<DataInst>的子类AugmentIterator，用于从单实例迭代器中创建一个批量迭代器。

在iter\_batch\_proc-inl.hpp中，定义了俩个IIterator<DataBatch>的子类，一个是BatchAdaptIterator，用于从单实例迭代器中创建一个批量迭代器，另一个是ThreadBufferIterator，用来作为线程迭代器寄存器。

在iter\_csv-inl.hpp中，定义了一个IIterator<DataInst>的子类CSVIterator，它是一个简单的csv迭代器，只被用来读取数据实例。

在iter\_image\_recordio-inl.hpp中，类ImgaeLabelMap用来存储图像标签，类ImageRecordIOParser用来解析图像IO记录，类ImageRecordIOIterator是IIterator<DataInst>的子类，作为图像IO记录的迭代器。

在iter\_img-inl.hpp中，定义了一个IIterator< DataInst >的子类ImageIterator，它是一个简单的图像迭代器，只被用来读取数据实例。

在iter\_mem\_buffer-inl.hpp中，定义了一个IIterator<DataBatch>的子类DenseBufferIterator，该迭代器用于将限定数量的batch存入memory中。

在iter\_mnist-inl.hpp中，定义了一个IIterator<DataBatch>的子类MNISTIterator，它被设计用来获取mnist数据集。

在iter\_thread\_imbin-inl.hpp中，定义了一个IIterator<DataInst>的子类ThreadImagePageIterator，它是线程寄存器迭代器的page版本。

在iter\_thread\_imbin\_x-inl.hpp中，定义了一个IIterator<DataInst>的子类ThreadImagePageIteratorX，暂时不太清楚它和ThreadImagePageIterator的区别，以后会更新。

在iter\_thread\_iminst-inl.hpp中，定义了一个IIterator<DataInst>的子类ThreadImageInstIterator，同上。

观察该部分所有代码可以发现，IIterator类是所有子类的父类，其类型可能是DataBatch或者DataInst，根据不同的需求，可以用IItertor不同的子类来完成，或者我们根据这一模式独自编写符合我们要求的代码。但有一点值得注意，SparseInst类型只在data.h出现过，在所有子类中均为出现。

该学习笔记还处在初步阶段，也只对代码做一个大致的功能讲解，笔者会在以后有更深一步的了解后更新该文档。

[cxxnet学习笔记3](http://blog.csdn.net/u010333076/article/details/46517619)

分类： [机器学习](http://blog.csdn.net/u010333076/article/category/5584677)2015-06-16 14:11 57人阅读 [评论](http://blog.csdn.net/u010333076/article/details/46517619#comments)(0) [收藏](javascript:void(0);) [举报](http://blog.csdn.net/u010333076/article/details/46517619#report)

cxxnet－utils模块学习笔记

utils模块被所有模块所依赖，其包含一些基础的功能模块。

utils模块包含以下源码文件：

[config.h](https://github.com/dmlc/cxxnet/blob/master/src/utils/config.h)

[decoder.h](https://github.com/dmlc/cxxnet/blob/master/src/utils/decoder.h)

[io.h](https://github.com/dmlc/cxxnet/blob/master/src/utils/io.h)

[metric.h](https://github.com/dmlc/cxxnet/blob/master/src/utils/metric.h)

[random.h](https://github.com/dmlc/cxxnet/blob/master/src/utils/random.h)

[thread.h](https://github.com/dmlc/cxxnet/blob/master/src/utils/thread.h)

[utils.h](https://github.com/dmlc/cxxnet/blob/master/src/utils/utils.h)

utils.h是一个接口，实现了一些简单基础的功能函数，应该是直接从xgboost扒过来的，接口包括：

#ifndef CXXNET\_CUSTOMIZE\_MSG\_

inline void HandleAssertError(const char \*msg) {}

//输出由于不当输入造成的Assert error

inline void HandleCheckError(const char \*msg) {}

//输出由于不当输入造成的Check errot

inline void HandlePrint(const char \*msg) {}

//用于输出msg消息

#else

//假如包含如下声明，必须实现这些接口。

void HandleAssertError(const char \*msg);

//处理由于不当输入造成的Assert error

void HandleCheckError(const char \*msg);

//处理由于不当输入造成的Check errot

void HandlePrint(const char \*msg);

//用于输出msg消息

#endif

inline void Printf(const char \*fmt, ...) {}

//输出消息到控制台

inline int SPrintf(char \*buf, size\_t size, const char \*fmt, …){}

//向字符串中打印数据

inline void Check(bool exp, const char \*fmt, ...) {}

//和Assert相同，但可以当作消息为用户使用。

inline void Error(const char \*fmt, ...) {}

//用于报错，和check相同。

inline std::FILE \*FopenCheck(const char \*fname, const char \*flag){}

//替换fopen，当fopen失败时报相应错误。

inline void TrackerPrint(const std::string msg, bool root\_only = true) {}

//输出消息给tracker。

inline void TrackerPrintf(const char \*fmt, ...) {}

//先vsnprintf消息，再交由TrackerPrint输出。

uitls提供的接口就如上这些所示。

config.h用于读取配置参数，它包含三个类，类的继承关系如下所示：

ConfigReaderBase -> ConfigStreamReader -> ConfigIterator

ConfigReaderBase是参数读取类的基类，包含了一些基本的实现；ConfigStreamReader是一个使用了stream基类的迭代器，允许使用所有类型的IO流；ConfigIterator是一个迭代器，用于迭代配置文件并读取配置信息。

io.h定义了IO流的抽象接口，包含了一个结构体和两个类。

结构体MemoryBufferStream继承了类ISeekStream，它是一个内存缓冲区，用作流接口被读写。

类StdFile继承ISeekStream，实现了文件IO流操作。BinaryPage是基本页面类。

decoder.h定义了两个图像解码器结构体，用于图像解码，一个是JpegDecoder，另一个是OpenCVDecoder，根据预定义的不同，选择不同的解码器。

metric.h定义了一个IMetric类和几个结构体，该单元用于评估Metric。

类IMetric是Metric失真评估器，结构体MetricBase是metric基类，结构体——MetricRMSE、MetricError、MetricRecall、MetricSet根据其命名，可以看出它们分别用作Metric的不同用途。

random.h定义了一个类RandomSampler，主要用于随机取样。

thread.h该单元是多线程编程基础模块，分为windows版本和linux版本两种实现方法，不管哪种方法，定义了两个同名类，分别是Samphore和Thread。Samphore用于线程间同步，Thread表示一个线程类。

thread\_buffer.h与thread.h互相协作，该单元实现了多线程缓冲器、迭代器，用于创建并行管道。其中定义了一个模版类ThreadBuffer，它可以使用多线程加载缓存迭代器，而且它采用了模板，可以加载Elem类型的元素，可以实现ElemFactory的factory类型以便使用线程缓冲区。

总体上来说，utils模块作为基本模块，完成了许多其他模块的基础功能，这使后面的开发可以更加高效，代码更加简练，这部分的理解也对后面的学习必不可少。

[cxxnet学习笔记4](http://blog.csdn.net/u010333076/article/details/46518721)

分类： [机器学习](http://blog.csdn.net/u010333076/article/category/5584677)2015-06-16 15:27 54人阅读 [评论](http://blog.csdn.net/u010333076/article/details/46518721#comments)(0) [收藏](javascript:void(0);) [举报](http://blog.csdn.net/u010333076/article/details/46518721#report)

cxxnet—nnet模块学习笔记

nnet模块抽象的表示了神经网络结构，它结合layer模块共同构筑了神经网络。

该模块包含的源码文件如下所示：

[neural\_net-inl.hpp](https://github.com/dmlc/cxxnet/blob/master/src/nnet/neural_net-inl.hpp)

[nnet.h](https://github.com/dmlc/cxxnet/blob/master/src/nnet/nnet.h)

[nnet\_config.h](https://github.com/dmlc/cxxnet/blob/master/src/nnet/nnet_config.h)

[nnet\_impl-inl.hpp](https://github.com/dmlc/cxxnet/blob/master/src/nnet/nnet_impl-inl.hpp)

[nnet\_impl.cpp](https://github.com/dmlc/cxxnet/blob/master/src/nnet/nnet_impl.cpp)

[nnet\_impl.cu](https://github.com/dmlc/cxxnet/blob/master/src/nnet/nnet_impl.cu)

[nnet\_ps\_server.cpp](https://github.com/dmlc/cxxnet/blob/master/src/nnet/nnet_ps_server.cpp)

在nnet模块中，依赖关系如下所示：

CXXNetThreadTrainer->NeuralNetThread->NeuralNet

nnet.h是其接口，其他.hpp文件实现了其接口，.cpp文件使用了.hpp文件实现的接口。

在nnet.h中，定义了类INetTrainer，它是trainer网络的接口，还定义了一个带模版的创建函数，用来根据所选择设备，网络类型不同，创建所需trainer网络。

nnet\_config.h实现网络结构配置功能，它定义了一个结构体NetConfig，它记录了神经网络的配置，用以存储网络结构，并可以读取该配置，它和每个层相关联。

nerual\_net-inl.hpp单元实现了常用神经网络，它定义了一个带模版的结构体NerualNet，实现了抽象神经网络，它还定义了一个模版类NerualNetThread，表达了神经网络的独立线程结构。模版提供的类型有CPU和GPU可供选择。

nnet\_impl-inl.hpp单元定义了一个类CXXNetThreadTrainer，它继承了INetTrainer，该类使用多线程实现了神经网络训练器，它同样包括一个模版，类型可选CPU或GPU。此外，它还定义了一个CXXNetThreadTrainer创建函数，可根据不同net\_type和设备创建所需CXXNetThreadTrainer，nnet\_impl.cpp文件唯一的函数Create也指向这里。

nnet\_ps\_sever.cpp单元涉及cxxnet和mshadow两个部分，其一是nnet下的CXXNetUpdater类实现，它继承于mshadow::ps::IModelUpdater<reanl\_t>，还有一部分是mshadow下ps内 IModelUpdater<cxxnet::real\_t>类型的CreateModelUpdater函数返回CXXNetUpdater类。

CXXNetUdater用于上传参数到参数服务器。

nnet\_impl.cu是nnet\_impl.cpp的GPU版本。

该模块主要用于构建神经网络，在下一步理解其内部逻辑关系以及网络构建流程才是关键。

[cxxnet学习笔记5](http://blog.csdn.net/u010333076/article/details/46533229)

分类： [机器学习](http://blog.csdn.net/u010333076/article/category/5584677)2015-06-17 14:10 36人阅读 [评论](http://blog.csdn.net/u010333076/article/details/46533229#comments)(0) [收藏](javascript:void(0);) [举报](http://blog.csdn.net/u010333076/article/details/46533229#report)

updater是参数更新模块，它定义了权值更新规则，其中AsyncUpdater是一个特殊的updater，用于处理异步通信和更新。该模块使用mshadow-ps完成异步通信。

updater模块包含的源码文件如下所示：

[updater.h](https://github.com/dmlc/cxxnet/blob/master/src/updater/updater.h)

[param.h](https://github.com/dmlc/cxxnet/blob/master/src/updater/param.h)

[adam\_updater-inl.hpp](https://github.com/dmlc/cxxnet/blob/master/src/updater/adam_updater-inl.hpp)

[async\_updater-inl.hpp](https://github.com/dmlc/cxxnet/blob/master/src/updater/async_updater-inl.hpp)

[nag\_updater-inl.hpp](https://github.com/dmlc/cxxnet/blob/master/src/updater/nag_updater-inl.hpp)

[sgd\_updater-inl.hpp](https://github.com/dmlc/cxxnet/blob/master/src/updater/sgd_updater-inl.hpp)

[updater\_impl-inl.hpp](https://github.com/dmlc/cxxnet/blob/master/src/updater/updater_impl-inl.hpp)

[updater\_impl.cpp](https://github.com/dmlc/cxxnet/blob/master/src/updater/updater_impl.cpp)

[updater\_impl.cu](https://github.com/dmlc/cxxnet/blob/master/src/updater/updater_impl.cu)

updater.h包含其外部接口，该模块定义了两个类与几个函数，如下所示：

类IUpdater参数更新模型的接口，它定义了一系列关于参数更新过程的操作，如下所示：

注：ILayer并不负责参数更新，IUpdater采用由ILayer和权值累加的梯度值来更新权值。

class IUpdater {

 public:

  typedef typename layer::ILayer<xpu>::IVisitor IVisitor;

//复用layer的vistor类型，可以用来获得updater的权值。

  virtual ~IUpdater(void) {}

  virtual void SetStream(mshadow::Stream<xpu> \*stream) = 0;

//设置内部计算流为流，参数中的stream为所用的流。

  virtual void Init(void) = 0;

//初始化，如果有响应则输出updater信息。

  virtual void ApplyVisitor(IVisitor \*pvisitor) = 0;

//应用visitor到updater，用于获取updater内容。

  virtual void StartRound(int round) = 0;

//通知updater我们开始了在数据上的新一轮迭代，参数round表示第几轮迭代

  virtual void Update(long epoch) = 0;

//更新参数。参数epoch为当前epoch值，epoch是当每一轮训练结束后mini-batches传过来的数。

  virtual void Update(long epoch, mshadow::Tensor<xpu, 2> grad) = 0;

//更新参数，从外部提供梯度值。参数epoch属性不变，参数grad是为了简化接口的指向梯度值的指针，在传递梯度值之前FlatTo2D应该被调用。

  virtual void SetParam(const char \*name, const char \*val) = 0;

//用来设定updater参数

};

类IAsyncUpdater继承IUpdater，BeforeBackprop和AfterBackprop是异步调用函数，用户需要调用UpdateWait来等待更新过程完成。下面展示IAsyncUpdater结构：

template<typename xpu>

class IAsyncUpdater : public IUpdater<xpu> {

 public:

  virtual void BeforeBackprop(const std::vector<layer::Node<xpu>\*> &nodes\_in,const std::vector<layer::Node<xpu>\*> &nodes\_out) = 0;

//在BP前调用此函数，当updater想要通过其自身恢复梯度值时使用此函数，无需ILayer计算此过程。

  virtual void AfterBackprop(bool do\_update, long epoch) = 0;

//在BP后调用此函数，参数do\_update决定是否在这个迭代器执行更新，如果更新，参数epoch为更新所用epoch值。

  virtual void BeforeAllForward(void) = 0;

//当所有layer调用forwardprop时，该函数会被调用。

  virtual void UpdateWait(void) = 0;

//阻塞直至更新完成，如果没有更新或更新早已完成，该函数会直接返回。

//禁用Update功能

  virtual void Update(long epoch) {

    utils::Error("IAsyncUpdater.Update call AfterBackprop instead");

  }

  virtual void Update(long epoch, mshadow::Tensor<xpu, 2> grad) {

    utils::Error("IAsyncUpdater.Update call AfterBackprop instead");

  }

};

根据指定类型创建updater，参数type指定updater类型，p\_rnd为产生的随机数，weight是要更新的权值，wgrad为张量的梯度值，tag是weight的类型。

template<typename xpu>

IUpdater<xpu>\* CreateUpdater(const char \*type,

                             mshadow::Random<xpu> \*p\_rnd,

                             mshadow::Tensor<xpu, 2> weight,

                             mshadow::Tensor<xpu, 2> wgrad,

                             const char \*tag);

为指定layer创建多个updater，将它们推回out\_updaters。参数layer\_index为layer索引，device\_id为异步updater所在设备id，param\_server为异步updater可用的参数服务器，type指明updater类型，p\_rnd指针为产生的随机数，layer\_type为layer的类型，p\_layer指针指向数据流出的layer对象，out\_updaters容器存储输出，如果它里面已包含成员，就在容器尾部加入新的updater。

template<typename xpu>

void CreateAsyncUpdaters(int layer\_index,

          int device\_id,

          mshadow::ps::ISharedModel<xpu, real\_t> \*param\_server,

          const char \*type,

          mshadow::Random<xpu> \*p\_rnd,

          layer::LayerType layer\_type,

          layer::ILayer<xpu> \*p\_layer,

          std::vector<IAsyncUpdater<xpu>\*> \*out\_updaters);

//kDataKeyStep是一常数，用于参数服务器的编码键索引。计算方式如下：

 \*   data\_key = layer\_index \* kDataKeyStep

 \*   key(layer[i].bias) == i \* kDataKeyStep + 1

 \*   key(layer[i].bias) == i \* kDataKeyStep + 1

static const int kDataKeyStep = 4;

//将layer索引和weight标签编码为唯一键，参数layer\_index为layer索引，参数tag为weight类型

inline int EncodeDataKey(int layer\_index, const char \*tag) {…}

//由参数key解码tag。

inline const char \*DecodeTag(int key) {…}

updater.h的结构就如上所示。

param.h单元完成常见的参数更新操作，支持复杂的学习速率调度。只包含一个结构体UpdaterParam的定义，它表示每一个layer的潜在参数。

adam\_updater-inl.hpp单元以momentum实现SGD，它包含一个模版类AdmUpdater的定义，继承于IUpdater ，该类实现所有相关操作。

async\_updater-inl.hpp单元使用SGD实现异步更新，定义了一个类AsyncUpdater继承于IAsyncUpdater。

nag\_updater-inl.hpp单元以momentum实现NAG，它也只定义了一个类NAGUpdater，其继承于IUpdater。

sgd\_updater-inl.hpp单元以momentum实现SGD，它定义了一个结构体－clip（用于梯度划分和nan监测）和一个类SGDUpdater（继承于IUpdater）

updater\_impl-inl.hpp单元一同编译所有updater的实现。该单元包括两个创建函数，分别是：CreateUpdater\_和CreateAsyncUpdater\_，这俩个函数根据指定类型创建指定updater。

它还定义了一个结构体CreateAsyncUpdaterVisitor，继承于IUpdater。

updater\_impl.cpp实现了CreateUpdater函数和CreateAsyncUpdaters函数的CPU版本，updater\_impl.cu实现了其GPU版本。

以上是该模块的讲解，后续会继续补充。该模块主要是参数更新模型，下一步我们会开始最后一个模块layer的学习。

[cxxnet学习笔记6](http://blog.csdn.net/u010333076/article/details/46536621)

分类： [机器学习](http://blog.csdn.net/u010333076/article/category/5584677)2015-06-17 18:08 53人阅读 [评论](http://blog.csdn.net/u010333076/article/details/46536621#comments)(0) [收藏](javascript:void(0);) [举报](http://blog.csdn.net/u010333076/article/details/46536621#report)

layer模块实现了神经网络层并定义了Forward和Backprop。

layer包括以下这些源码文件：

[loss](https://github.com/dmlc/cxxnet/tree/master/src/layer/loss)

[activation\_layer-inl.hpp](https://github.com/dmlc/cxxnet/blob/master/src/layer/activation_layer-inl.hpp)

[batch\_norm\_layer-inl.hpp](https://github.com/dmlc/cxxnet/blob/master/src/layer/batch_norm_layer-inl.hpp)

[bias\_layer-inl.hpp](https://github.com/dmlc/cxxnet/blob/master/src/layer/bias_layer-inl.hpp)

[concat\_layer-inl.hpp](https://github.com/dmlc/cxxnet/blob/master/src/layer/concat_layer-inl.hpp)

[convolution\_layer-inl.hpp](https://github.com/dmlc/cxxnet/blob/master/src/layer/convolution_layer-inl.hpp)

[cudnn\_convolution\_layer-inl.hpp](https://github.com/dmlc/cxxnet/blob/master/src/layer/cudnn_convolution_layer-inl.hpp)

[cudnn\_pooling\_layer-inl.hpp](https://github.com/dmlc/cxxnet/blob/master/src/layer/cudnn_pooling_layer-inl.hpp)

[dropout\_layer-inl.hpp](https://github.com/dmlc/cxxnet/blob/master/src/layer/dropout_layer-inl.hpp)

[fixconn\_layer-inl.hpp](https://github.com/dmlc/cxxnet/blob/master/src/layer/fixconn_layer-inl.hpp)

[flatten\_layer-inl.hpp](https://github.com/dmlc/cxxnet/blob/master/src/layer/flatten_layer-inl.hpp)

[fullc\_layer-inl.hpp](https://github.com/dmlc/cxxnet/blob/master/src/layer/fullc_layer-inl.hpp)

[insanity\_layer-inl.hpp](https://github.com/dmlc/cxxnet/blob/master/src/layer/insanity_layer-inl.hpp)

[insanity\_pooling\_layer-inl.hpp](https://github.com/dmlc/cxxnet/blob/master/src/layer/insanity_pooling_layer-inl.hpp)

[layer.h](https://github.com/dmlc/cxxnet/blob/master/src/layer/layer.h)

[layer\_impl-inl.hpp](https://github.com/dmlc/cxxnet/blob/master/src/layer/layer_impl-inl.hpp)

[layer\_impl.cpp](https://github.com/dmlc/cxxnet/blob/master/src/layer/layer_impl.cpp)

[layer\_impl.cu](https://github.com/dmlc/cxxnet/blob/master/src/layer/layer_impl.cu)

[lrn\_layer-inl.hpp](https://github.com/dmlc/cxxnet/blob/master/src/layer/lrn_layer-inl.hpp)

[op.h](https://github.com/dmlc/cxxnet/blob/master/src/layer/op.h)

[pairtest\_layer-inl.hpp](https://github.com/dmlc/cxxnet/blob/master/src/layer/pairtest_layer-inl.hpp)

[param.h](https://github.com/dmlc/cxxnet/blob/master/src/layer/param.h)

[pooling\_layer-inl.hpp](https://github.com/dmlc/cxxnet/blob/master/src/layer/pooling_layer-inl.hpp)

[prelu\_layer-inl.hpp](https://github.com/dmlc/cxxnet/blob/master/src/layer/prelu_layer-inl.hpp)

[split\_layer-inl.hpp](https://github.com/dmlc/cxxnet/blob/master/src/layer/split_layer-inl.hpp)

[visitor.h](https://github.com/dmlc/cxxnet/blob/master/src/layer/visitor.h)

[xelu\_layer-inl.hpp](https://github.com/dmlc/cxxnet/blob/master/src/layer/xelu_layer-inl.hpp)

其中loss是一个文件夹，展开其内部如下所示：

[loss\_layer\_base-inl.hpp](https://github.com/dmlc/cxxnet/blob/master/src/layer/loss/loss_layer_base-inl.hpp)

[lp\_loss\_layer-inl.hpp](https://github.com/dmlc/cxxnet/blob/master/src/layer/loss/lp_loss_layer-inl.hpp)

[multi\_logistic\_layer-inl.hpp](https://github.com/dmlc/cxxnet/blob/master/src/layer/loss/multi_logistic_layer-inl.hpp)

[softmax\_layer-inl.hpp](https://github.com/dmlc/cxxnet/blob/master/src/layer/loss/softmax_layer-inl.hpp)

我们还是从layer.h开始。

layer.h是该模块的接口，它主要定义了如下这些内容：

template<typename xpu>

struct Node {};

//该结构体用于存储网络中的Forward激励与BP梯度。

struct LabelRecord {};

//一个label记录可被一个损失函数采用，使用结构体保障未来的可扩展性。

struct LabelInfo {}

//用于保存label实例的额外信息，这些信息可被layer用来计算在目标函数上的梯度，这个数据结构会被升级，以满足不同种类的监督信号的需求。

template<typename xpu>

struct ConnectState {};

//表示连接状态，时间空间信息可被Forward和Backprop之间共享。并不是所有layer需要它。

//ILayer是该模块的接口，它是一个纯粹的接口，里面没有数据成员，然而，在

一个layer中可以存在具有相同模式的成员，具体规则如下：

Connection 和 Layer:

在当前CXXnet设计中，connection和layer的概念为：

一个layer定义了Forward和Backprop功能并给定了输入输出节点；

一个layer不依附于特定的节点，而Connection是；

Connection是节点间的连接，这些节点的功能由layer支持；

不同的connection可共享一个相同的layer；

这意味着layer不能包含特定状态（如：标记为dropout）的节点类；

Connection指定其状态由connection维护，在Forward/Backprop过程中传递给layer。

权重和梯度：

一些layer可以有参数为权重和梯度的connection；

这些权重由特定实现持有；

它们可以用一个vistor获取，见IVisitor；

SaveModel和LoadModel用来序列化和反序列化它们。

参数：

参数与每一个layer相关（如：隐层节点个数），可以调用SetParam设置。

template<typename xpu>

class ILayer {

 public:

  struct IVisitor {};

//定义layer的vistor，用于访问layer的权重和梯度值。

  virtual ~ILayer(void) {}

  virtual void InitConnection(

  const std::vector<Node<xpu>\*> &nodes\_in,

            const std::vector<Node<xpu>\*> &nodes\_out,

            ConnectState<xpu> \*p\_cstate) = 0;

//初始化connection，该函数负责俩件事

1、设置输出节点的形状；

2、在p\_cstate分配必要的时间状态空间。

  virtual void OnBatchSizeChanged(

const std::vector<Node<xpu>\*> &nodes\_in,

const std::vector<Node<xpu>\*> &nodes\_out,

ConnectState<xpu> \*p\_cstate) {}

//当batch大小改变时更新p\_cstate，

  virtual void Forward(bool is\_train,

                       const std::vector<Node<xpu>\*> &nodes\_in,

                       const std::vector<Node<xpu>\*> &nodes\_out,

                       ConnectState<xpu> \*p\_cstate) = 0;

//由输入节点向输出节点做前向传播

  virtual void Backprop(bool prop\_grad,

                        const std::vector<Node<xpu>\*> &nodes\_in,

                        const std::vector<Node<xpu>\*> &nodes\_out,

                        ConnectState<xpu> \*p\_cstate) = 0;

//由输出节点向输入节点做后向传播，在函数调用前，确保输出节点包含梯度值。

  virtual bool AllowSharing(void) const {

    return true;

  }

//询问是否可跨多个connection共享这一layer，对于大多数layer应当返回true。

  virtual void SetStream(mshadow::Stream<xpu> \*stream) {}

//设置内部计算的stream为stream。

  virtual void ApplyVisitor(IVisitor \*pvisitor) {}

//应用vistor到layer，vistor用来访问layer的内容。

  virtual void SetParam(const char \*name, const char\* val) {}

//设置layer的参数。

  virtual void InitModel(void) {}

//初始化Model参数，只在model参数未被初始化时调用。

  virtual void SaveModel(utils::IStream &fo) const {}

//将model存入二进制文件。

  virtual void LoadModel(utils::IStream &fi) {}

//从二进制文件中读取model

};

上面这些就是ILayer类的定义。下面的是一些枚举，省略其中一部分。

const int kSharedLayer = 0;

const int kFullConnect = 1;

const int kSoftmax = 2;

…

const int kFixConnect = 31;

const int kBatchNorm\_no\_ma = 32;

const int kPairTestGap = 1024;

typedef int LayerType;

inline LayerType GetLayerType(const char \*type) {}

//给出type名，返回其对应的整数值，对应关系参考上面的枚举。

template<typename xpu>

ILayer<xpu>\* CreateLayer(LayerType type,

                         mshadow::Random<xpu> \*p\_rnd,

                         const LabelInfo \*label\_info);

//由给定类型创建一个updater，它与节点很像，但包含了label信息。

//指定connection的数据结构，定义了节点间的连接。

template<typename xpu>

struct Connection {

  ILayer<xpu> \*layer;

//指向后端连接的layer

  LayerType type;

//后端layer的类型

  ConnectState<xpu> state;

//表示当前的共享信息

  std::vector<Node<xpu>\*> nodes\_in;

//输入节点容器

  std::vector<Node<xpu>\*> nodes\_out;

//输出节点容器

  inline void SetStream(mshadow::Stream<xpu> \*stream) {}

//设置内部计算流

};

layer.h的定义到此为止，其重点在于node、ILayer和Connection这三部分。

op.h定义了mshadow针对cxxnet的额外操作。

在每一个layer中几乎都用到了参数，param.h提供的结构体LayerParam并不强制使用，但在layer中使用它会很有帮助。LayerParam定义了任意layer所有可能会用到的参数。

visitor.h给出了visitor关于layer权重操作的实现，提供给utils模块从一个layer设置或得到权重的方法。它实现了俩个类GetWeightVisitor和SetWeightVisitor，都继承于IVisitor，如名字所示，一个用来从layer获得权重，一个用于给layer设置权重，使用举例如下：

 Get Example:

      GetWeightVisitor vs("weight");

      layer->ApplyVisitor(&vs);

      std::vector<mshadow::Tensor<xpu, 2> > weights = vs.data;

 Set Example:

      GetSetVisitor vs(data, "weight");

      layer->ApplyVisitor(&vs);

由于Layer模块最为复杂庞大，所以我们分为两节来讲，本节到此结束，下一节会对所有.hpp文件学习。

[cxxnet学习笔记78](http://blog.csdn.net/u010333076/article/details/46618097)

分类： [机器学习](http://blog.csdn.net/u010333076/article/category/5584677)2015-06-24 09:32 43人阅读 [评论](http://blog.csdn.net/u010333076/article/details/46618097#comments)(0) [收藏](javascript:void(0);) [举报](http://blog.csdn.net/u010333076/article/details/46618097#report)

cxxnet整体架构

1、cxxnet-main.cpp是入口，它会实例化一个CXXNetLearnTask类的实例tsk，并运行tsk的Run方法。

2、在Run方法中，先读入配置参数到程序中，然后调用Init()完成初始化，Init会创建INetTrainer对象net\_trainer，还会创建迭代器，具体流程如下：

 a、在 Init中，如果当前task为train且continue\_training为true，需要先读取模型（会调用CreateNet()方法），然后创建迭代器，并返回；如果模型为空，则转到b。

 b、现在停止训练，设置continue\_training为false，检查模型是否为空，如果没有，则创建神经网络（调用自身的CreateNet()方法），然后初始化模型；如果模型不为空，则判断当前task是否为finetune，如果是则拷贝模型，不是则读取模型。

 c、最后创建迭代器，Init过程结束。

3、现在回到Run中，在Init过程结束后，根据task的不同（可供选择的有5种，train | finetune 、 pred 、extract 、get\_weight，默认初始化为train）选择四种任务其一启动，任务最开始会从train开始启动，我们下面讲述TaskTrain()流程：

 a、itr\_train为训练数据迭代器；start\_counter表示从第几层神经开始；num\_round为神经网络层数，默认为10；整个训练过程会由第start\_counter层神经开始，到num\_round层神经为止，也可以指定max\_round最大训练层数，取俩者中最小的为准。

 b、在每一层训练过程中，针对每一个训练数据的值，都会执行net\_trainner的Update方法。

 c、在迭代完所有训练数据之后，会执行net\_trainner的Evaluate方法一次，然后SaveModel，这一层神经的训练结束，回到过程a。

4、接下来讲述TaskPredict()过程；

 a、整个过程会遍历一遍所有itr\_pred，对每一个itr\_pred打包为DataBatch类型的batch，然后调用net\_trainer的Predict方法，直至所有被遍历。

5、TaskExtractFeature()：将每一个itr\_pred打包为batch，采用net\_trainer的ExtractFeature方法对提取其特征。

6、TaskGetWeight()：执行一次net\_trainer的GetWeight方法。

红字中标注的都是net\_trainer方法的调用，下一步我们探究nnet模块下的INetTrainer类，继续挖它的层次结构。

[xxnet学习笔记9](http://blog.csdn.net/u010333076/article/details/46618111)

分类： [机器学习](http://blog.csdn.net/u010333076/article/category/5584677)2015-06-24 09:33 42人阅读 [评论](http://blog.csdn.net/u010333076/article/details/46618111#comments)(0) [收藏](javascript:void(0);) [举报](http://blog.csdn.net/u010333076/article/details/46618111#report)

cxxnet-内部三层架构之间的逻辑关系。

我们在前面简述过，cxxnet的实现主要靠三个模块nnet、updater和layer，我们也探究过每个模块具体的功能和其内部结构，那么我们这一节的目标是搞清楚这三层之间是如何协同工作的。

前面我们提到过，这三层有着如下的依赖关系：nnet->updater->layer。前一节中，我们也指出cxxnet\_main.cpp向下调用了nnet模块中的几个功能，接下来，本节就从nnet开始，自顶向下的展示cxxnet其整体逻辑结构。

上一节中讲到Init过程会创建nnet模块指针net\_trainer，调用CreateNet创建INetTrainer实例，最终创建其实是INetTrainer的子类CXXNetThreadTrainer，即net\_trainer是CXXNetThreadTrainer对象，之后在cxxnet\_main.cpp中，net\_trainer作为nnet提供给上层的接口，使其可以调用其内部功能（如前文4个task分别对应的功能）。如图所示：

针对每个功能模块，我们再依次展开，首先展开train部分，查看Update和Evaluate，前面说过，update函数是用来根据训练集来更新模型参数的，Evaluate用来在每一轮update后评估训练结果。

CXXNetThreadTrainer用到多线程，其线程存储于nets\_，nets\_是NerualNetThread类，在Update中，通过调用所有线程的TrainForwardBackprop方法来完成更新。类似的，Evaluate执行的是SyncParam。之后，跳入线程内，在单个线程内部，由NeuralNet结构体声明的net\_变为其基本执行单位，所有的操作又针对其执行。在NeuralNet中，调用layer层对象做forwardbackprop，针对每一个权重，会创建一个updater用来更新其权值。关于layer和updater的具体细节可以看本笔记之前关于俩个模块的文档。

这样，单机版本的cxxnet的流程及其内部结构和各部分完成的功能我们已经大致清晰了，下一步，我们开始rabit和参数服务器的学习，来看一看分布式的cxxnet是如何运行的。

# [Rabit学习笔记](http://blog.csdn.net/u010333076/article/details/46618129)

分类： [机器学习](http://blog.csdn.net/u010333076/article/category/5584677)2015-06-24 09:34 40人阅读 [评论](http://blog.csdn.net/u010333076/article/details/46618129#comments)(0) [收藏](javascript:void(0);) [举报](http://blog.csdn.net/u010333076/article/details/46618129#report)

关于Rabit的一篇论文：<http://homes.cs.washington.edu/~icano/projects/rabit.pdf>

Rabit有着较完善的文档，理解Rabit重点在于它的**Allreduce和Broadcast**，还有就是它的容错和恢复机制。

首先要理解的是Allreduce这个概念，Allreduce相较于mapreduce，通过允许程序员轻松的将模型（这些模型将被复制于每个节点）维护于内存中，使它避免了不必要的map过程、重新分配内存步骤以及迭代器之间的硬盘读写过程。如图所示：

Rabit的Allreduce和Broadcast过程，可以用下图很具象的来解释：

每个节点向上传递自己的值，每个节点执行一次求和操作，直到根节点，这就是Allreduce过程；根节点告诉所有叶子节点结果，这就是Broadcast过程。

那么这俩个过程有何作用呢，往下看。

Rabit作为一个同步的分布式机器学习通信库，其最重要的特性就是其可靠性，它的可靠性体现在俩点：容错性和可恢复性。

那么是如何达到容错的呢？下图可以解释其容错机制：

从1 到 5是其执行步骤：

1. Pause every node until the failed node is fully recov- ered.
2. Detect the model version we need to recover by ob- taining the minimum operation number. This is done using the Consensus Protocol explained in section 3.2.2.
3. Transfer the model to the failed node using the Rout- ing Protocol explained in section 3.2.2.
4. Resume the execution of the failed node using the received model.
5. Resume the execution of the other nodes as soon as the failed node catches up.

容易理解，当蓝点down 机后，所有节点在下一个Allreduce前等待该节点恢复，在蓝点启动后，从最近相邻节点读取模型，然后和其他节点一同继续下去。

那么，是如何得知哪个点down 机了呢，就是通过模型版本加前面提到过的Allreduce，这里采用了一个一致性协议，协议规定所有模型的版本必须一致，版本号根据Allreduce轮数依次增加，所以找到有版本号比大家低的，就说明那个机器down掉了，恢复即可。如图所示：

关于如何选择最近节点，参考下图：

这里采用的是路由协议，一共俩轮，第一轮计算距离，第二轮用最近节点恢复。

# [ps－lite源码学习笔记](http://blog.csdn.net/u010333076/article/details/46649149)

分类： [机器学习](http://blog.csdn.net/u010333076/article/category/5584677)2015-06-26 13:36 30人阅读 [评论](http://blog.csdn.net/u010333076/article/details/46649149#comments)(0) [收藏](javascript:void(0);) [举报](http://blog.csdn.net/u010333076/article/details/46649149#report)

**ps-lite代码解析**

首先查看ps－lit的源码结构：

[base](https://github.com/dmlc/ps-lite/tree/master/src/base)

[filter](https://github.com/dmlc/ps-lite/tree/master/src/filter)

[kv](https://github.com/dmlc/ps-lite/tree/master/src/kv)

[parameter](https://github.com/dmlc/ps-lite/tree/master/src/parameter)

[proto](https://github.com/dmlc/ps-lite/tree/master/src/proto)

[ps](https://github.com/dmlc/ps-lite/tree/master/src/ps)

[system](https://github.com/dmlc/ps-lite/tree/master/src/system)

[ps.h](https://github.com/dmlc/ps-lite/blob/master/src/ps.h)

[ps\_main.cc](https://github.com/dmlc/ps-lite/blob/master/src/ps_main.cc)

其中，ps\_main.cc是其程序入口，ps.h是其接口，其他文件夹则包含各个功能模块。下面详细阐释各部分。

ps\_main.cc代码很短，如下所示：

#include "ps.h"

namespace ps {

App\* App::Create(int argc, char \*argv[]) {

  App\* app = new App();

  if (ps::IsServerNode()) {

    CreateServerNode(argc, argv);

  }

  return app;

}

}  // namespace ps

int main(int argc, char \*argv[]) {

  ps::StartSystem(&argc, &argv);

  int ret = ps::IsWorkerNode() ? WorkerNodeMain(argc, argv) : 0;

  ps::StopSystem();

  return ret;

}

首先，只看main函数，main函数的流程很明晰，可以展示为下图：

可以发现，main函数主要用于创建WorkerNode，在main函数上面的代码中，我们注意到有一个app类，关于app类的定义，可见ps/app.h，这个类，在这里，这个Create用于创建SeverNode。

那么，main函数的功能不难理解，它就是用来创建WorkerNode和SeverNode的。

ps.h：接下来，我们展开接口函数ps.h，理解了它，结合main函数中的流程，我们就不难理解整个ps－lite的架构，首先观察其源码，笔者将整个接口分为4部分，分别是：

Worker node APIs

Server node APIs

Scheduler Node APIs（目前为空）

More Advanced APIs

由于其中Scheduler Node APIs 为空，所以我们分三部分来研究。

**第一部分——Worker node APIs ：**

//Syncopts是push和pull操作的选项集

struct Syncopts{

  std::vector<int> deps;

//deps是push操作的时间戳，只有当该操作被参数服务器处理时，才会修改其时间戳。

  std::function<void()> callback;

//当收到参数服务器返回的响应时执行callback。

  std::vector<Filter> filters;

//key－value过滤器用于减少通信开销

  Filter\* AddFilter(Filter::Type type);

//用于添加过滤器，使用举例：AddFilter(Filter::COMPRESSING);

  int cmd = -1;

//－1表示无命令。

};

//KVWorker是用来收发参数服务器key-value对的缓存器。

template<typename Val>

class KVWorker{

  explicit KVWorker(int id = 0) {

    cache\_ = CHECK\_NOTNULL((new KVCache<Key, Val>(id)));

  }

//id是用来在参数服务器中寻找KVStore的唯一标识，负数ID的KVWorker为系统专用。//注：声明为explicit表示其不可被隐式转换。

  ~KVWorker() {

delete cache\_;  }

// Basic Push and Pull

int Push(const std::vector<Key>& keys,

           const std::vector<Val>& vals,

           const SyncOpts& opts = SyncOpts()) {

    // copy the data, then use the zero copy push

    return ZPush(std::make\_shared<std::vector<Key>>(keys),

                 std::make\_shared<std::vector<Val>>(vals), opts);

  }

//push操作将一串key-value对传到参数服务器。

//它是一个非阻塞调用，当要发送的信息进入系统的发送队列立刻返回。真正的push操作只在wait返回时或者提供的callback被调用时才结束。

int Pull(const std::vector<Key>& keys,

           std::vector<Val>\* vals,

           const SyncOpts& opts = SyncOpts()) {

    // copy the data, then use the zero copy pull

    return ZPull(std::make\_shared<std::vector<Key>>(keys), vals, opts);

  }

//pull操作根据key将value从参数服务器取出。

//它是一个非阻塞调用，当要发送的信息进入系统的发送队列立刻返回。真正的pull操作只在wait返回时或者提供的callback被调用时才结束。

void Wait(int timestamp) {

    cache\_->Wait(timestamp);

  }

//Wait操作用来阻塞下一个请求，直到当前请求完成。

//  Zero-copy Push and Pull

//默认情况下，Push和pull操作都会先拷贝数据，然后用户程序可以重写或者删除数据。然而，有些情况下memcpy十分耗费时间，所以我们用ZPush和ZPull以减少时间，但需要注意的是，用户需要自己保持key和value的值不变直到请求完成。

int ZPush(const std::shared\_ptr<std::vector<Key> >& keys,

           const std::shared\_ptr<std::vector<Val> >& vals,

            const SyncOpts& opts = SyncOpts()) {

return cache\_->Push(GetTask(opts), SArray<Key>(keys),SArray<Val>(vals), SArray<int>(),opts.callback);

  }

  int ZPull(const std::shared\_ptr<std::vector<Key> >& keys,

            std::vector<Val>\* vals,

            const SyncOpts& opts = SyncOpts()) {

    return cache\_->Pull(GetTask(opts), SArray<Key>(keys),

                        CHECK\_NOTNULL(vals), NULL, opts.callback);

  }

…

private:

  Task GetTask(const SyncOpts& opts);

  KVCache<Key, Val>\* cache\_;

};

int WorkerNodeMain(int argc,char \*argv[]);

//WorkerNodeMain是一个WorkerNode的Main函数。

**第二部分——Server node APIs ：**

//IOnlineHandle是一个用于处理请求的句柄例子。

template <typename Val, typename SyncVal>

class IOnlineHandle {

public:

  IOnlineHandle() { }

  virtual ~IOnlineHandle() { }

  inline void Start(bool push, int timestamp, int cmd, void\* msg) { }

//开始处理一个Worker的请求。

  inline void Finish() { }

//请求已被处理。

  inline void Init(Key key, Val& val) { }

//Handle初始化。

  inline void Push(Key recv\_key, Blob<const SyncVal> recv\_val, Val& my\_val) {…}

//处理来自worker节点的push请求

  inline void Pull(Key recv\_key, const Val& my\_val, Blob<SyncVal>& send\_val){…}

//处理来自worker节点的pull请求

  inline void SetCaller(void \*obj) { }

//接收caller

};

//Sever节点用OnlineServer存储key－value对，key－value对从worker发过来，被Handle依次接收，然后存储进KVStore中。

template <typename Val, typename SyncVal = Val,

          typename Handle = IOnlineHandle<Val, SyncVal> >

class OnlineServer {

public:

  OnlineServer(const Handle& handle = Handle(), int pull\_val\_len = 1, int id = 0) {…};

  ~OnlineServer() { }

  KVStore\* server() { return server\_; }

 private:

  KVStore\* server\_ = NULL;

};

//Sever节点的main函数。

int CreateServerNode(int argc, char \*argv[]);

**第三部分——More Advanced APIs ：**

inline App\* MyApp()

{ return Postoffice::instance().manager().app(); }

//返回当前节点运行的App

inline Node MyNode()

{ return Postoffice::instance().manager().van().my\_node(); }

//返回当前节点的全局ID

inline std::string MyNodeID() { return MyNode().id(); }

//返回我的所有节点的ID

inline int IsWorkerNode()

{ return MyNode().role() == Node::WORKER; }

//判断当前节点是否为Worker节点

inline int IsServerNode()

{ return MyNode().role() == Node::SERVER; }

//判断当前节点是否为Sever节点

inline int IsSchedulerNode()

{ return MyNode().role() == Node::SCHEDULER; }

//判断当前节点是否为scheduler节点

inline std::string SchedulerID() {

  return Postoffice::instance().manager().van().scheduler().id();}

//返回SchedulerID

inline int MyRank() { return MyNode().rank(); }

//返回节点所在组的rankID，

inline int RankSize() {

  auto& mng = Postoffice::instance().manager();

  return IsWorkerNode() ? mng.num\_workers() : (IsServerNode() ? mng.num\_servers() : 1);}

//返回当前组的节点数。

inline int NumWorkers() { return FLAGS\_num\_workers; }

inline int NumServers() { return FLAGS\_num\_servers; }

inline void StartSystem(int\* argc, char \*\*\*argv) {

  ps::Postoffice::instance().Run(argc, argv);

}

inline void StopSystem() {

  ps::Postoffice::instance().Stop();

}

inline int RunSystem(int\* argc, char \*\*\*argv) {

  StartSystem(argc, argv); StopSystem();

  return 0;

}

# [mshaodw－ps模块学习](http://blog.csdn.net/u010333076/article/details/46711299)

分类： [机器学习](http://blog.csdn.net/u010333076/article/category/5584677)2015-07-01 16:01 23人阅读 [评论](http://blog.csdn.net/u010333076/article/details/46711299#comments)(0) [收藏](javascript:void(0);) [举报](http://blog.csdn.net/u010333076/article/details/46711299#report)

shadow－ps模块学习笔记

mshadow-ps实现了一个双层机制的参数服务器，架构如下图所示。通常一个GPU或一个CPU运行一个工作节点，一个一层服务器节点和工作节点在同一台机器上通信。不同机器之间的通信依赖二层服务器节点。

因为同机上的通信开销远远小于不同机器之间，所以通过在不同的层次之间使用不同的模型可以更好的平衡算法效率和系统性能。比如在一层使用顺序一致性模性来保障局部一致性，在二层使用最终一致性模型从而隐藏通信延时。