Caffe安装以及代码结构理解CQIU

2017年4月17日星期一

目 录

[1. 模板标题1 3](#_Toc480191340)

[1.1 模板标题2 3](#_Toc480191341)

[1.1.1 模板标题3 3](#_Toc480191342)

[1.1.2 模板标题3 3](#_Toc480191343)

[1.1.2.1 模板标题4 3](#_Toc480191344)

[1.1.2.2 模板标题4 3](#_Toc480191345)

[2. 模板标题1 3](#_Toc480191346)

[2.1 模板标题2 3](#_Toc480191347)

[2.1.1 模板标题3 3](#_Toc480191348)

[2.1.1.1 模板标题4 3](#_Toc480191349)

## Caffe 的安装问题

模板标题1正文

### Caffe在Eclipse上的安装问题

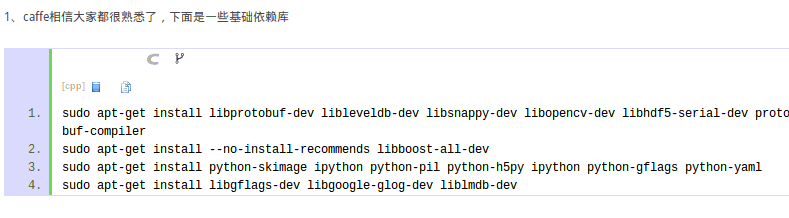
模板标题2正文

#### Caffe在eclipse上安装问题Ubuntu14.04版本问题

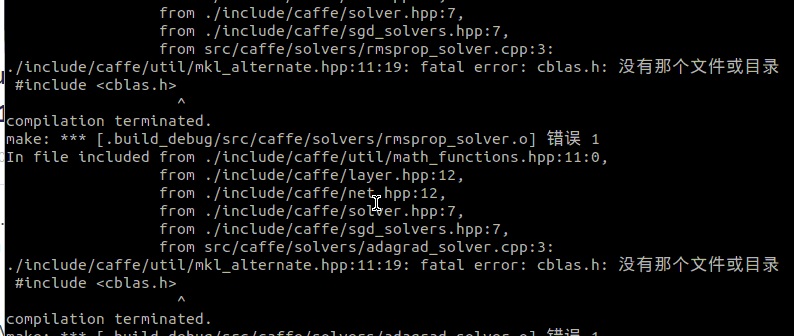
##### 问题1描述，没有flags基础库

提示没有flags库等等东西，就是缺库，要安装基础依赖库

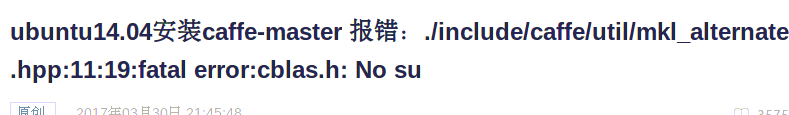
##### 解决方法



##### 问题2描述，cblas.h没有库

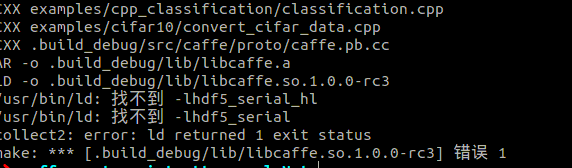


##### 解决



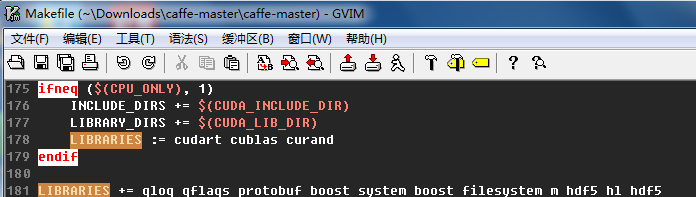


##### 问题3描述，提示hdf5\_serial\_hl库没有



##### 解决

特别注意，对于Ubuntu14.04来说，不需要hdf5\_serial\_hl这个库，库的名字不对，真实的库名字是：就是hdf5\_hl以及hdf5就可以了。

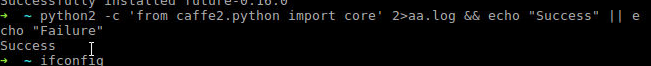


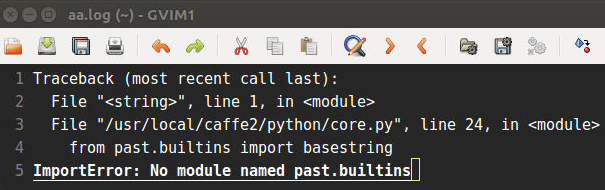
#### Caffe2在eclipse上安装问题汇总

具体的安装步骤,可以参考网上,一大推安装教程

##### 问题描述

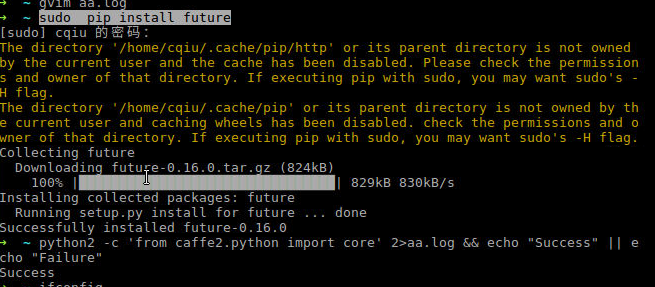
安装失败，打开重定向文件aa.log





##### 解决方法

提示这种错误，上网查后，升级pip，再测试就OK了

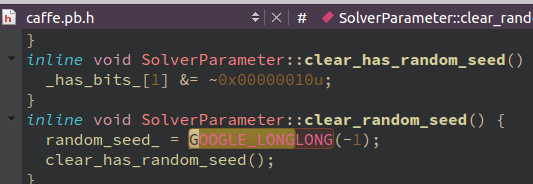


### Caffe的protobuf问题

问题描述见下

#### 问题描述

问题定位发现，总是提示GOOGLE\_LONGLONG这个宏找不到，没办法，就重新安装了protobuf2.6.1就ok了。



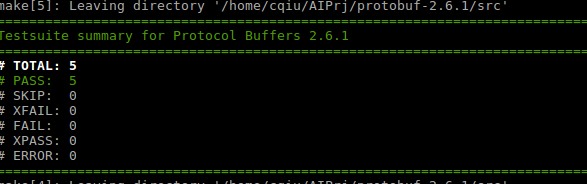
#### 解决方法

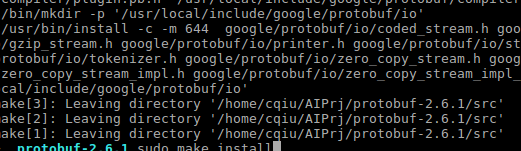
重新安装protobuf2.6.1

按照这个提示安装即可：



中间提示OK的过程





### import caffe中提示no module google.protobuf.internal

解决方法：



## qt 打开caffe工程方法

具体参见正文

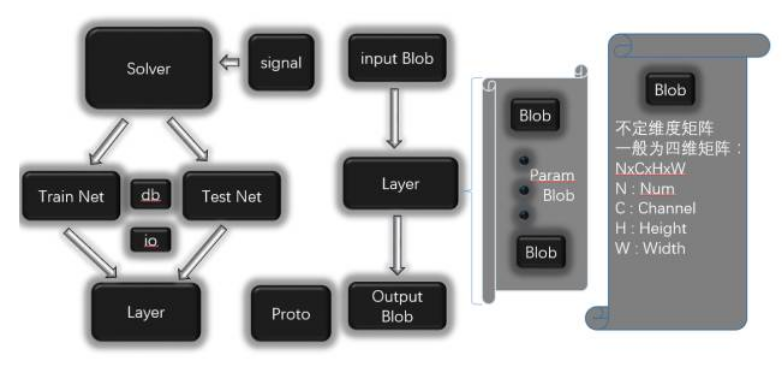
### 模板标题2

模板标题2正文

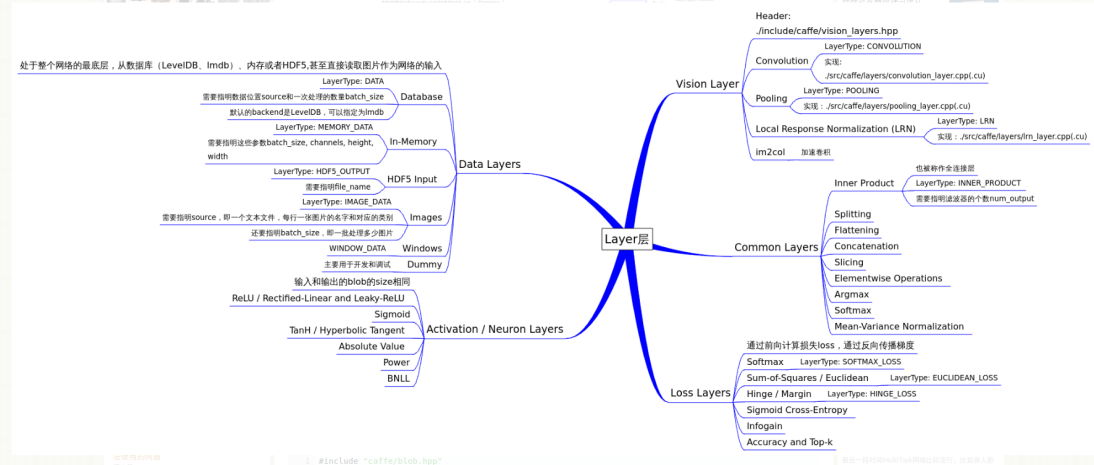
## Caffe 的框架分析

### 整体结构

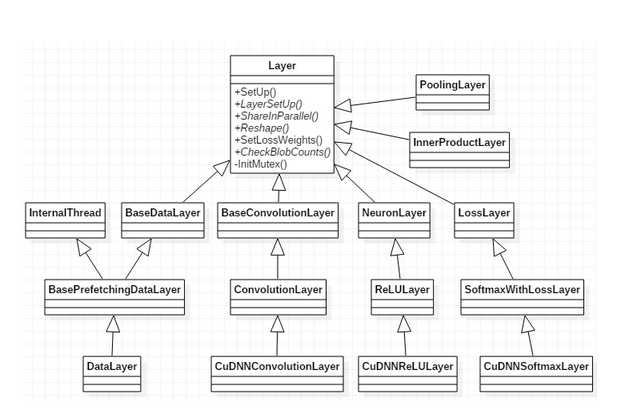
整体框架如图所示：



### 代码思维导图

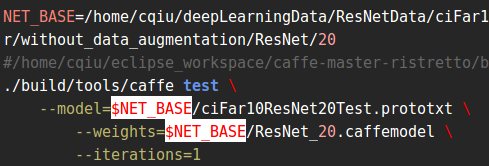


最核心的Layer层的UML图



## Caffe 在Eclipse上的代码研究

运行的命令：以一条测试为主线



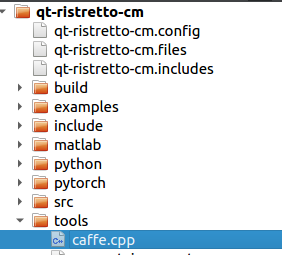
### Caffe.cpp程序入口分析

在源代码里面如下：

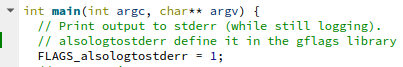
#### Caffe.cpp分析

##### 程序起始入口

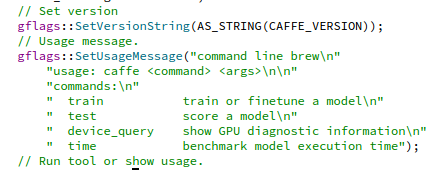
位置



这个应该是google公司的gflags命令行参数库的宏，设置为1了。



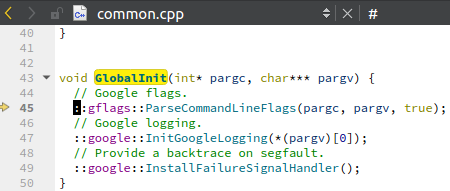
接下来设置Caffe的版本号，在gflags库中为main函数设置usage信息。



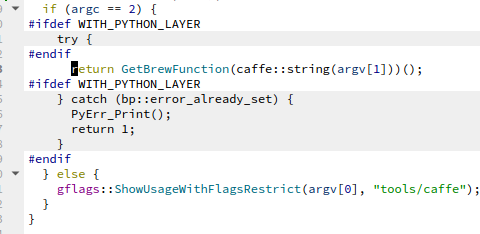
GlobalInit函数是全局初始化函数，用于初始化google开源库gflags(gflags::ParseCommandLineFlags)和glog(google::IniGoogleLogging)



调用如下：



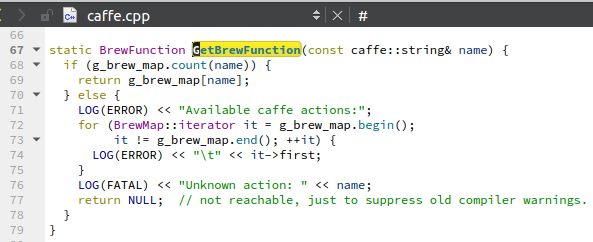
开始传进来时，argc是5个参数，处理过后，变成了2.



然后进入最为关键的GetBrewFunction函数

##### GetBrewFunction

Brew策划之意，此时传进来的string(argv[1])的值是test



这个函数依然在caffe.cpp文件中，为了方便描述，下面给出函数指针类型定义的方法。

##### C++中函数指针类型定义

###### typedef 函数指针类型定义

typedef 定义了一种“函数指针”类型，可以声明很多变量，函数指针定义了一个变量，

比如

Int max(int x,int y) {return (x>y?x:y)}这个是一个普通的函数。

区别比较，以前我们定义个新的类型

typedef int Int32; 我们把int类型起了一个新的名字，叫Int32。

那么现在，我们定义了一个函数指针

typedef int (\*func\_ptr)(int,int);

感觉格式和之前的都对不上，个人理解

Typedef int 任何函数名字(int a, int b) func\_ptr; ,其中func\_ptr就是这样函数的类型名字。

那么我们可以 声明变量并赋值 func\_ptr abc = max;

##### 回到caffe代码



表示一种新的函数类型。

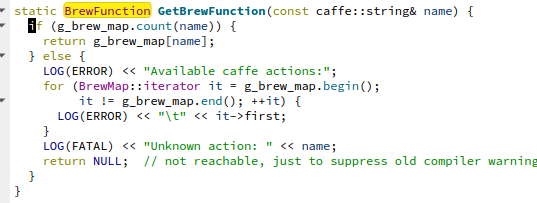


Std::map是STL 标准C++库中的关联容器，它提供一对一的数据处理能力，特点是增加删除节点影响小，可以修改实时值，不能修改Key。

Map特点：自建一套 Key-Value记录，它可以是任何类型的，能快速删除记录，更改记录。

因此这里面Key的类型就是string, Value的类型就是BrewFunction，就是一个函数，然后这样的map类型，就是BrewMap，感觉就是很多函数的容器，比较好理解。

看这一段代码。



GetBrewFunction是一个BrewFunction类型的函数，他是静态函数。

疑问：既然BrewFunction指的是函数入口是()空的形式，为啥GetBrewFunction不是空的呢？

解答：BrewFunction说的是这个函数GetBrewFunction的返回类型，跟GetBrewFunction本身函数入口没关系哦。

比如 float test1(int aa,int bb,int cc){}

这个BrewFunction就相当于float，当然和int aa,...之类的可以不同了。

然后g\_brew\_map就是一个map的全局变量。



知识补充：

map和set两种容器的底层结构都是红黑树，所以容器中不会出现相同的元素，因此count()的结果只能为0和1，可以以此来判断键值元素是否存在(当然也可以使用find()方法判断键值是否存在)。

拿map<key,value>举例，find()方法返回值是一个迭代器，成功返回迭代器指向要查找的元素，失败返回的迭代器指向end。count()方法返回值是一个整数，1表示有这个元素，0表示没有这个元素。

BrewFunction函数执行结束后，将返回test函数入口的指针，这种方法是在牛逼啊。

##### 全局注册器

Caffe中全局变量中有一个特殊变量map，也要赋值，采用注册器的机制。



这个注册器里面，传递的宏func,然后做整体的替换操作。核心就是把func的名字作为string。关键地方 map容器 g\_brew\_map[#func]= $func,函数地址作为值，函数名称作为string，也就是Key，这样写入map容器中进行注册。

G\_brew\_map[test] = &test;

G\_brew\_map[train] = &train;

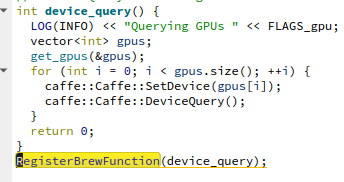
G\_brew\_map[time] = &time;

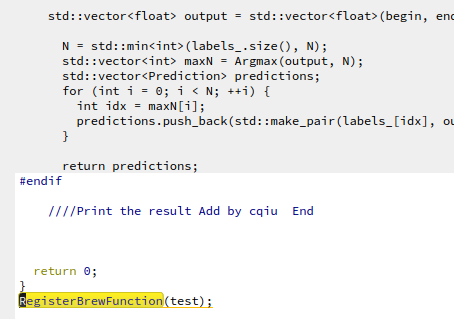
G\_brew\_map[device\_query]=$device\_query;

这种方式实在nibility啊。

宏定义中的#,##，以后再说把。

然后每个函数后面都会注册一把：



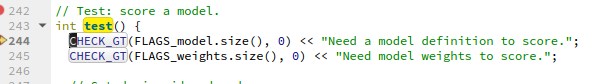


等等，具体本人研究了实例，这个不是Caffe的功能，不需要特殊的库，是C++语法。

具体参见这个小例子。/home/cqiu/qt\_workspace/qt-caffe/qt\_func\_register。至此成功的执行到了test()函数的入口。

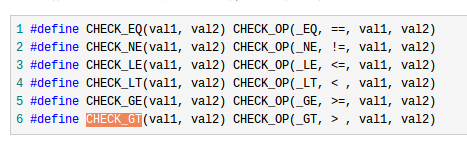
### test()分支函数

在源代码里面如下：

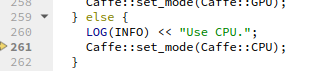


CHECK\_GT是google公司Glog库中的宏，表示第一个参数FLAGS\_modle.size()要大于第二个参数0，才能正常运行，否则就打印Need a model definition to score。

拓展分析：

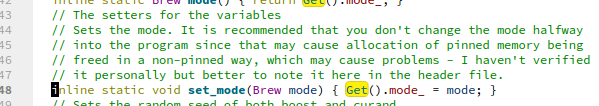


EQ，表示相等，NE表示不等，LE表示小于等于，LT表示小于，GE表示大于等于，GT表示大于。



#### Caffe::set\_mode

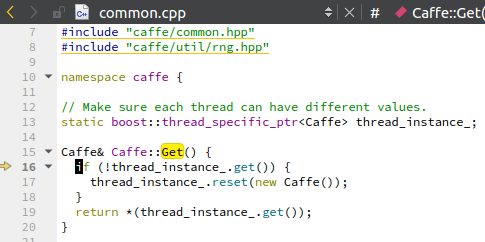
执行到Caffe::set\_mode(Caffe::CPU)，设置CPU模式，设置指针方式，采用了shared\_ptr智能指针的方式。



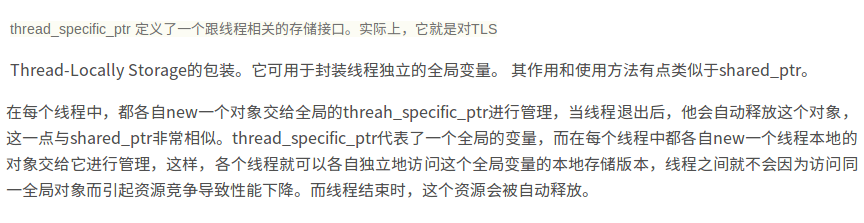
其中Brew是一个枚举



在Caffe的名空间内，有一个Get()函数



这里面调用了boost标准库，封装了线程独立的全局变量，作用和shared\_ptr差不多。

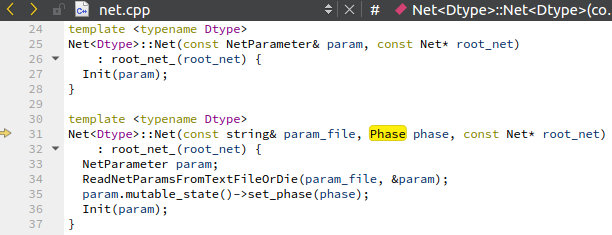


第16行，如果全局变量thread\_instance定义了get方式，那么就直接返回，否则就定义一下，再返回。

#### Caffe\_net(FLAGS\_model,caffe::TEST)



初始化caffe\_net网络，这个是一个最大的网络，Net里面有很多层。



实际调用第31行Phase属于TEST分支，root\_net在Net自己初始化的时候再定义。

##### 小插曲，实例化的时候为啥叫 caffe\_net?

具体问题：

代码: 

而Net的定义是 

小疑问：为啥搞成



怎么是第2个Net对应的是caffe\_net呢？难道名字可以随便修改么？

解答：

首先Net是一个类，也就是一个新的数据类型吧，类似于Int等等的，但是实例化对象的时候可以 Net abc;的这种方式。

第2,Net这个类，是模板的，可以有 Net<float> abc;这种方式。

第3,Net这个类，初始化的时候可以有多个入参，比如31行介绍的这种，就是可以有3个入参，其中第3个入参还有默认的初始值，:root\_net(root\_net)这种方式。

因此就有 Net<float> abc(参数1,参数2);这种形式，因此

就会有：

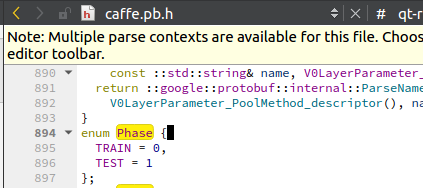
这种形式了，其实就是实例化一个Net的对象，他叫caffe\_net。

##### caffe\_net入口参数分析

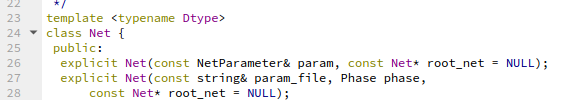
第一个参数string类型的，没啥好说的，第2个参数Phase的，是个枚举类型，

第三个参数，root\_net，初始化的时候指向NULL，也就是0.

Phase 类型说明：在caffe.pb.h文件下，有Phase的类型定义，是一个枚举类型

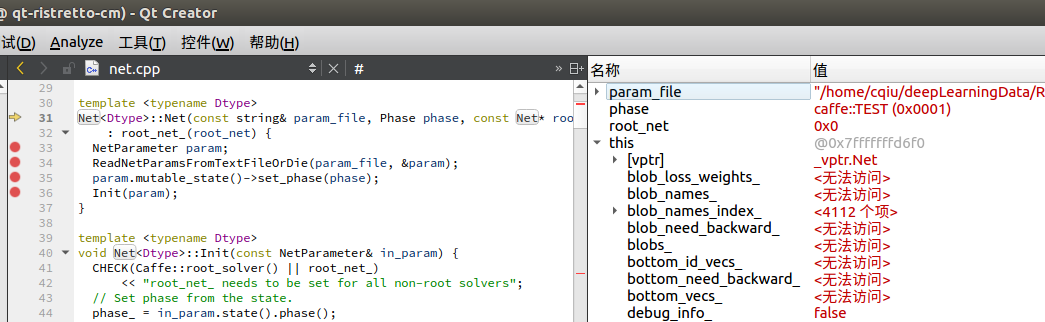


第3个参数，默认调用的时候，如果没有赋值，就是NULL，第26行给出了提示。

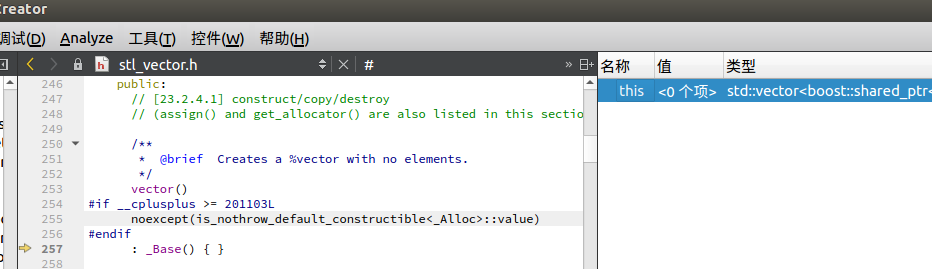


##### caffe\_net入口单步跟踪的时候，为何进入到vector界面

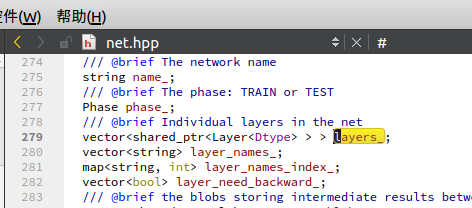
第一步：看this指针，就是caffe\_net对象，里面好多成员变量都未赋值。



第二步：跳入到vector部分，给this指针各个变量赋值，那么为何要跳入到vector模块呢？



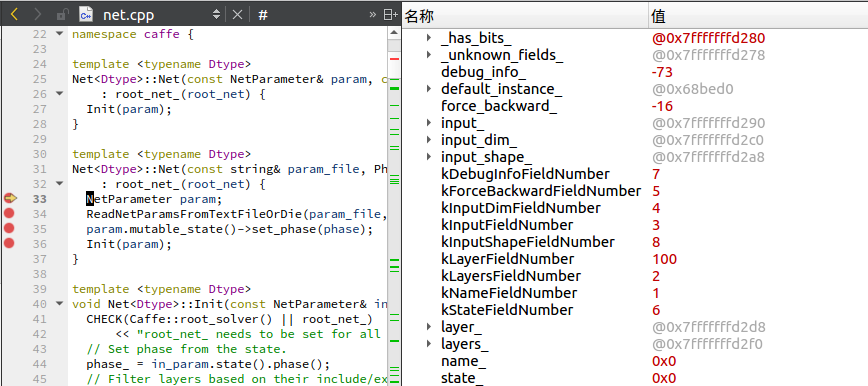
解答：net里面有很多成员变量都是vector的，比如layers\_等等，其实我们自己做一个简单小实例，定义一个vector变量，要是单步调试，都会首先进入到这个stl\_vector.h中\_Base()函数的入口上，后面的分析有关容器的介绍，可以慢慢再研究

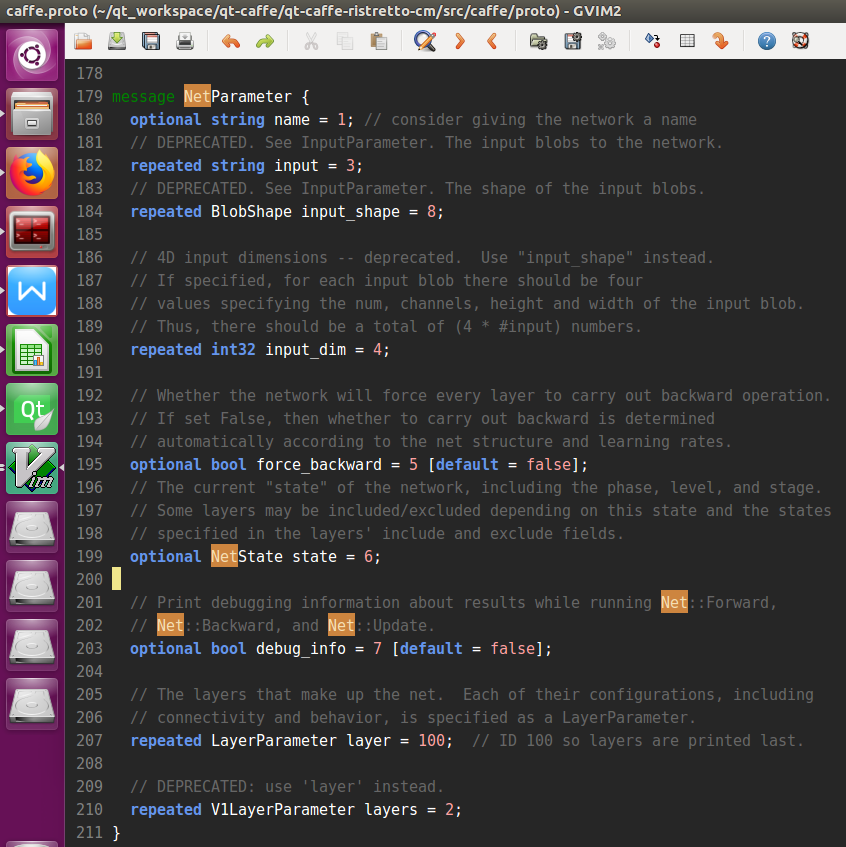


##### C++中标准容器的介绍，迭代器，vector等等后续研究

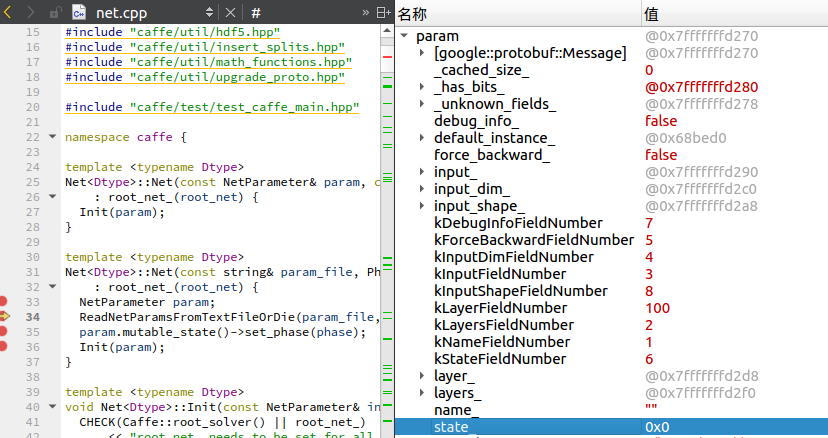
##### 分析NetParameter param;代码

单步跟踪到NetParameter 这个类型，这是一个结构体，只不过出现在的caffe.proto文件内部，但是它的类型是message的，是google公司的一种数据定义方式.





经过一顿初始化后，赋值成如下，这些值是初始化的数据。



##### 分析ReadNetParamsFromTextFileOrDie

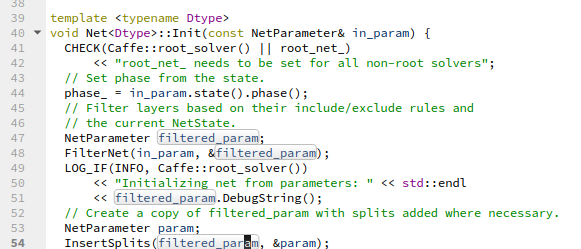
##### 分析param.mutable\_state()->set\_phase(phase)



这行代码完成给param赋值，过滤掉phase不用的信息。

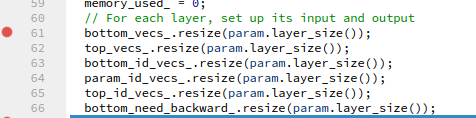
##### 重点Init(param)



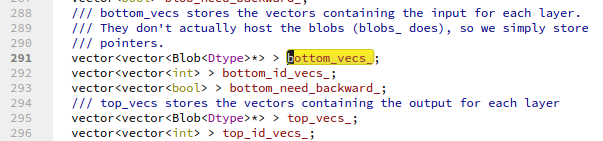


第48行，将in\_param中不符合规则的层过滤掉，比如我们做的是TEST，如果prototxt中有train的层介绍，就把这些参数干掉。

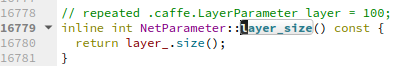
第54行，进行分层，一般比如Inception来说，有些层的结果是由很多其他层共同作用的，需要分裂子层，便于实现网络的通用性。



看bottom\_vecs\_,它作为NetClass的成员变量，它是是向量类型的向量。



第61行中，resize是向量的内建函数，param.size表示层数



###### 知识点补充向量的resize函数



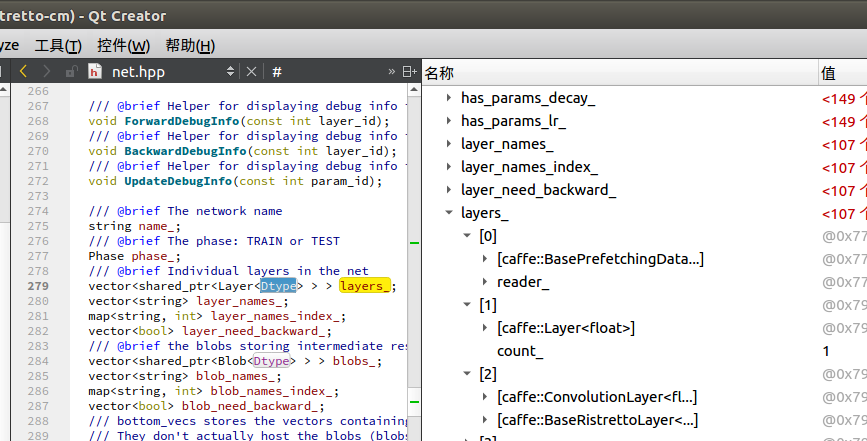
Init初始化过程很复杂，这里不具体研究了，但是要关系初始化后的结果状态。

###### Init初始化结果

初始化结束后，layer\_向量结果如下。

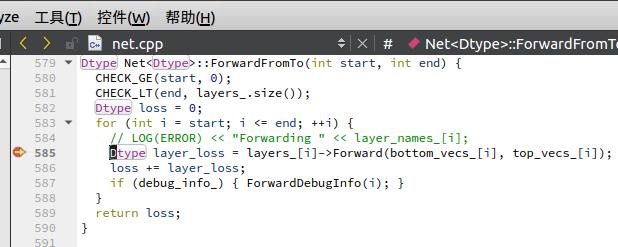
从279行可以看大，layer\_向量本身是一个向量，有vector<>形式，

<>内包裹的是 shared\_ptr<Layer<Dtype> >方式，其中shared\_ptr的<>里面包裹的是数据类型Layer<Dtype>。 Shared\_ptr，表示它是一个指针类型。



#### Layers\_[i]->Forward(bottom\_vecs\_[i],top\_vecs\_[i])分析

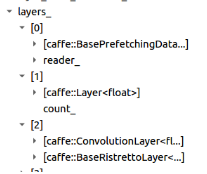
代码如下：



前文介绍过，layer\_是 Net类的一个成员，它是一个向量，向量的里元素是指针类型的。

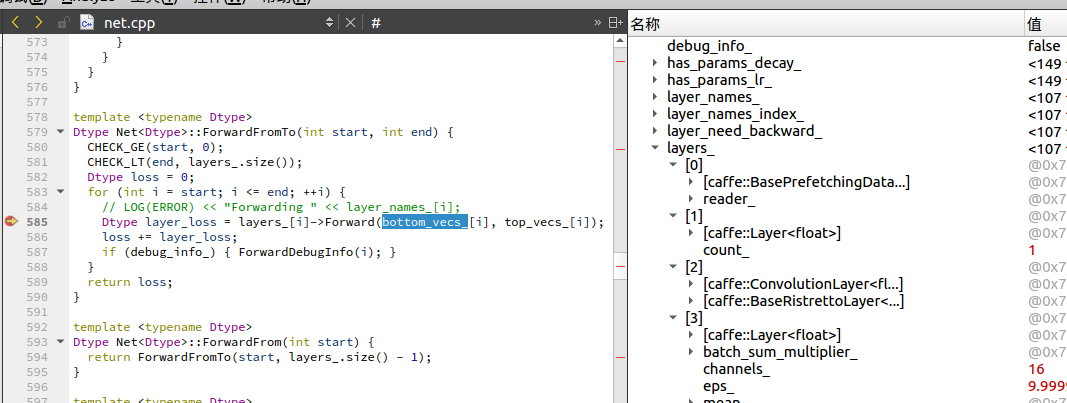


指针是Layer<Dtype>类型的，Dtype可以是float,double等等。这个Layer<Dtype>可以有很多种，比如BasePrefetchingData,Layer<float>,ConvolutionLayer等等。



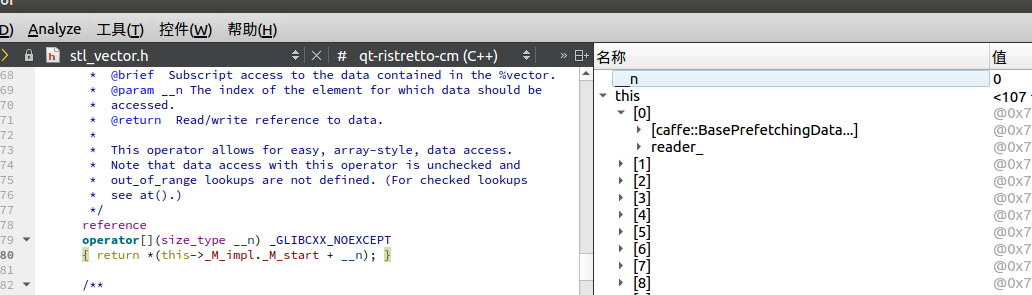
因此他们执行的函数Forward也就是对应指针类型的下的Forward函数。

此处start =0 , end = 106，表示585行要执行106次，bottom\_vecs\_和top\_vecs\_是之前Init时候，分配好地址空间后，的Feature向量

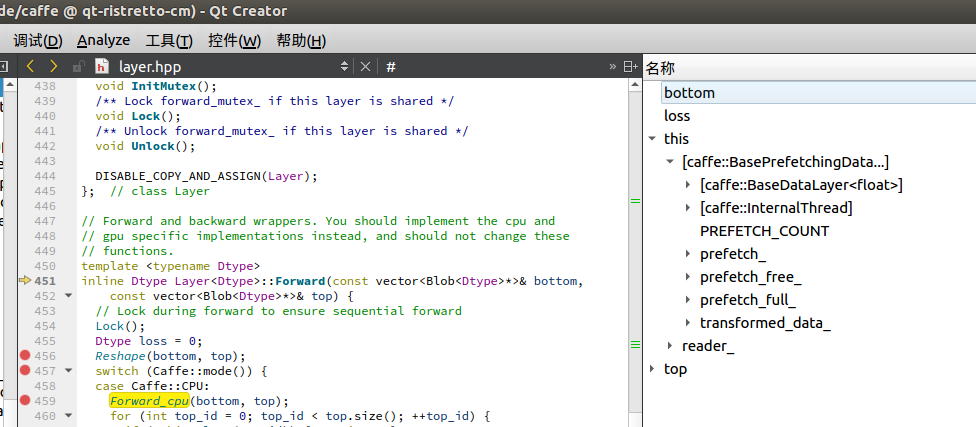


从图中可以看出，对应layers\_向量，每个索引下对应实际的Layer都不相同，比如第0个是BasePrefetchingData，第1个是Layer<float>，第2个是ConvolutionLayer之类的等等。

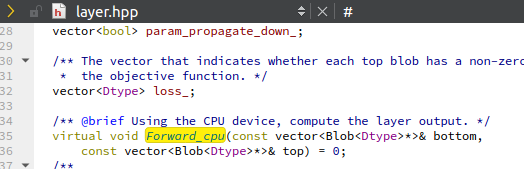
单步跟踪后如下图，



继续跟踪后，进入Layer->Forward函数，执行到了Forward\_cpu函数



此时我们看一下Layer类中的Forward\_cpu函数，发现它 Forward\_cpu()=0，表明它是一个纯虚函数，不干任何事情，由此也直到Layer是一个抽象类，要执行必须依托于它派生的类（具体参见纯虚函数和抽象类的定义）,这里面this指针指向的是BasePrefetchingData，因此接下来将执行这个类的Forward\_cpu方法



##### 纯虚函数Forward\_cpu知识点补充



Forward\_cpu这个虚函数直接 = 0;就表示它是纯虚函数，不是返回值为0,而表示它就是一个纯虚函数，告诉编译器的。

###### 知识补充，纯虚函数作用

补充知识：纯虚函数作用：

纯虚函数只有函数的名字而不具备函数的功能，不能被调用。   
纯虚函数的作用是在基类中为其派生类保留一个函数的名字，以便派生类根据需要对他进行定义。如果在基类中没有保留函数名字，则无法实现多态性。   
如果在一个类中声明了纯虚函数，在其派生类中没有对其函数进行定义，则该虚函数在派生类中仍然为纯虚函数。

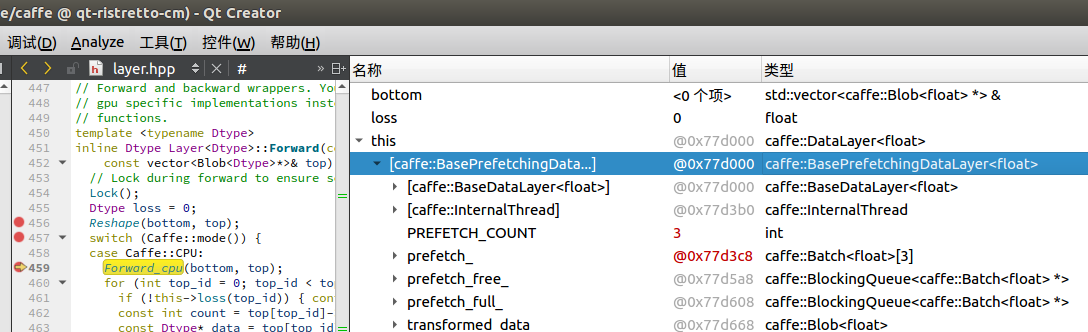
个人理解，纯虚函数不能直接调用，一定是在派生类中进行了调用的，有点注册的意思。

在我们这里，Forward\_cpu这个纯虚函数是在Layer class中的，在Layer中，没有直接调用，Layer是抽象类。

抽象类：就是带有一个或多个纯虚函数的类，Layer就是

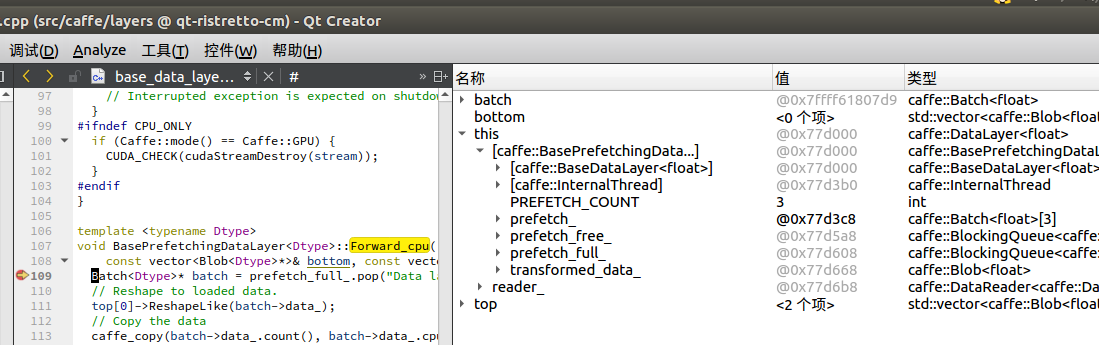
抽象类，不能定义类对象，但是可以定义类指针和引用。

##### 调用BasePrefetchingDataLayer类下的Forward\_cpu



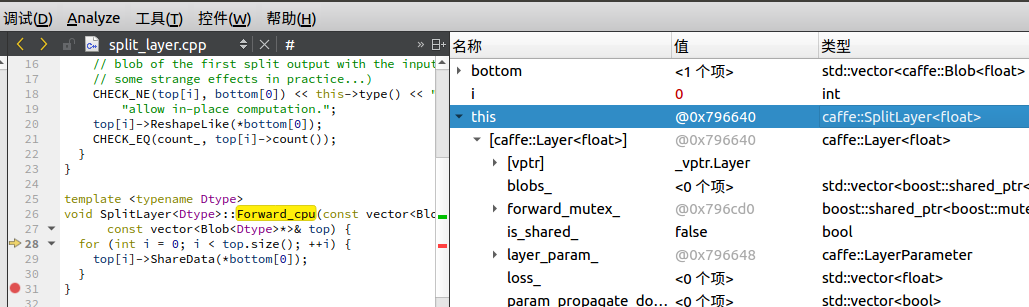
注意Forward\_cpu中this指向的类型，他是BasePrefetchingDataLayer<float>

因此要执行这个类下的Forward\_cpu



##### 第2次调用SplitLayer<float>类下的Forward\_cpu

This的类型就是这样。

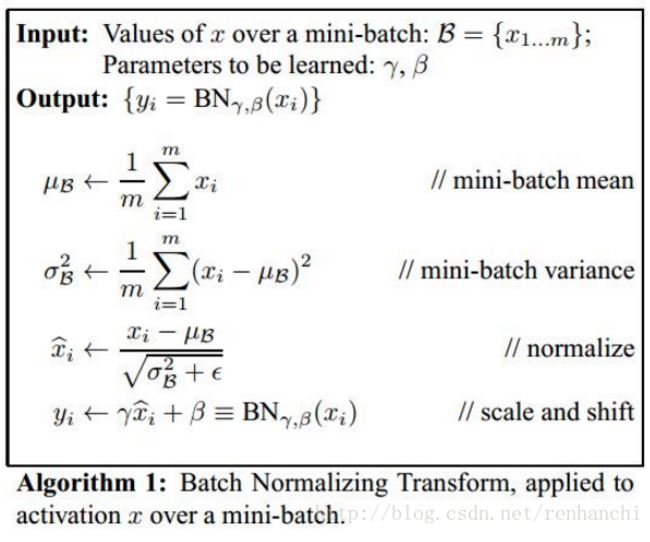


后面就按照同样的办法，每次都调用不同Layer下的Forward\_cpu函数即可执行不同的任务。

#### Batch Normalization 代码分析

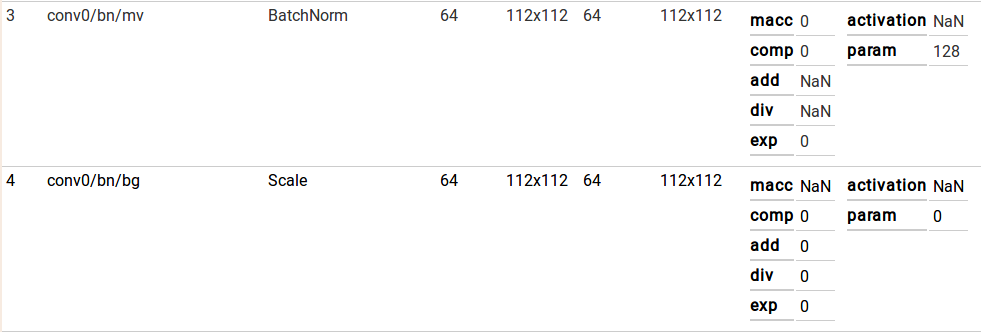
##### BatchNorm归一化公式

一般归一化的公式如下，见第3个公式//normalize带这个注释的行。

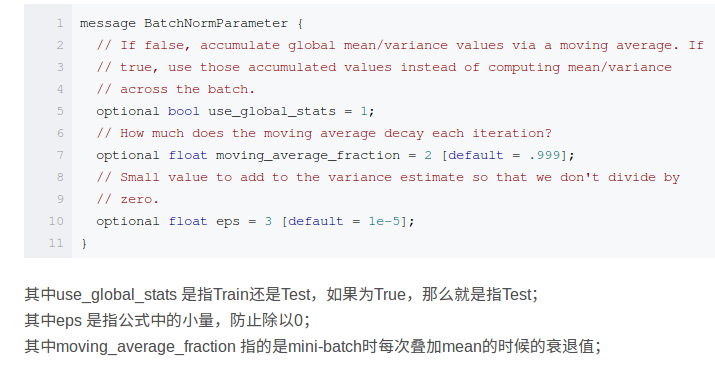


特别注意：对于残差网络，归一化参数是通道的2倍，128=64x2，为啥是2个，根据公式，我们可以把平均值提前算出来，可以把提前算出来，有多少个通道，就有多少个参数，因此是2×Channels，后续代码详细解析里面有BatchNormal具体归一化方法。

Scale不需要参数哦。



##### BatchNormParameter参数



##### LayerSetUp模块





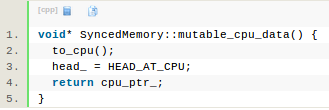
##### Batch Normal中Forward Cpu模块

###### 准备知识0--cpu\_data和mutable\_cpu\_data的区别

一开始看代码的时候会纳闷，为啥caffe里头又一个

cpu\_data，还要有一个mutable\_cpu\_data，其实从字面就可以看出来这个mutable就是易变的意思啦。二话不说翻出源代码看看究竟有啥区别:





他们之间就相差一个

head\_ = HEAD\_AT\_CPU;

这是为什么呢？为啥需要这样做？

我想原因就是

当你想读取数据的时候请使用cpu\_data

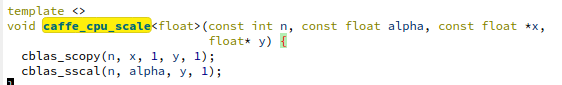
当然想修改数据的时候请你使用mutable\_cpu\_data，mutable\_cpu\_data，这种写法，编译器就不好优化，容易影响效率。

这样就提示系统数据我改过啦，你要小心了的意思(我只知道数据一定在CPU上)。

另外还有一个 gpu\_data和mutable\_gpu\_data，也是这样的原因。

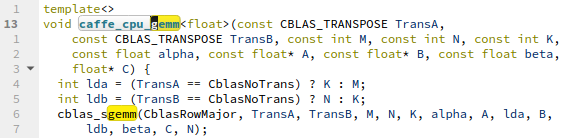
###### 准备知识1--caffe\_cpu\_scale函数

模板标题5正文



这里面，y=x,y=alpha \* y

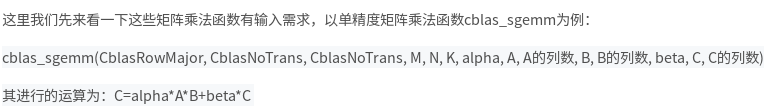
###### 准备知识2--caffe\_cpu\_gemm函数

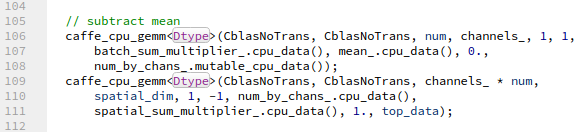


其中 M ----> A的行数

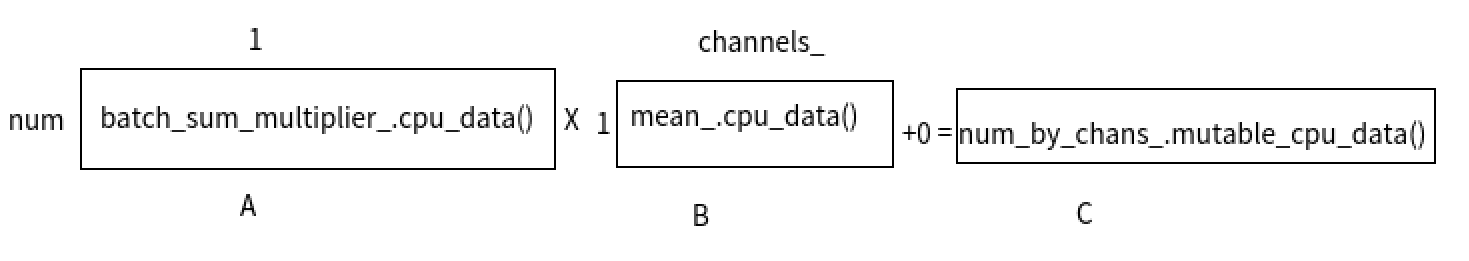
lda ----> A的列数，是B的行数

ldb ----> B的列数

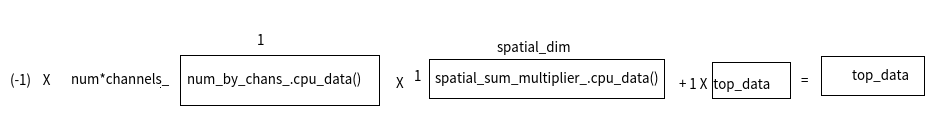




第106行等价为：



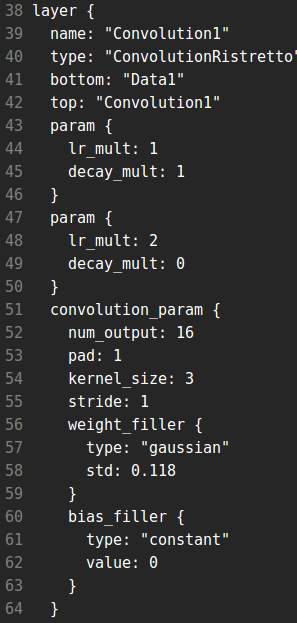
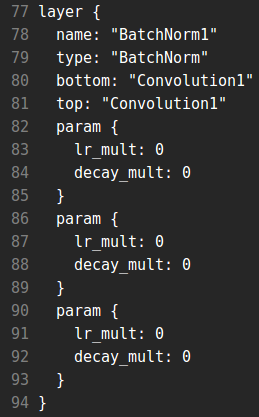
第107行等价为



背景：

以下说明，都是以resNet20网络，ciFar10的数据集来讲的，

第一层Prototxt如下：

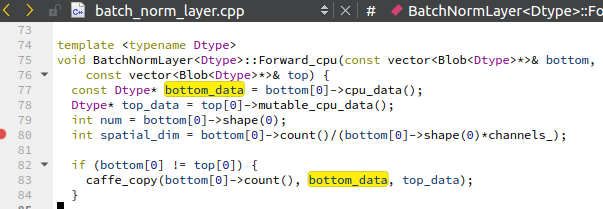
还有一个地方，ciFar10,开始输入的图像尺寸是32x32的

完毕。

###### 代码详细分析

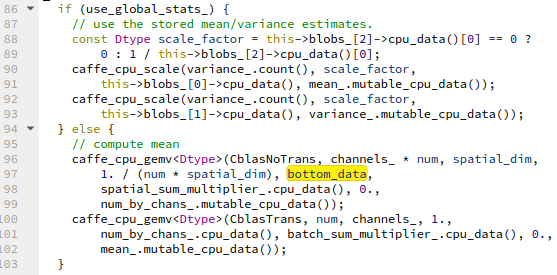
初始化部分

开始分析：

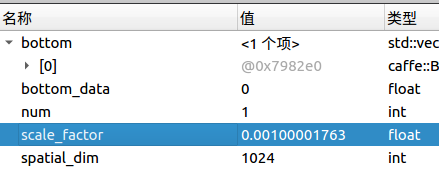


输入是bottom,最后经过batchNormal 归一化之后，就得到了top。

Num=1,表示批处理是1,spatial\_dim是第一层Feature的输出，是32x32=1024， 第82行说的是，如果bottom和top不是指向同一段内存，就把bottom\_data复制到top\_data内存空间上，这里面一般都是相同的。



Use\_global\_status\_ 对于Test来说，他是1,因此走第87-93行的分支。



Scale\_factor只有一个值，

，

mean\_.mutable\_cpu\_data()矩阵

如下：



平均数

 = scale\_factor \* 

因此首先得出了平均数的mean\_值，这里面是16个点。

variance\_.mutable\_cpu\_data()矩阵

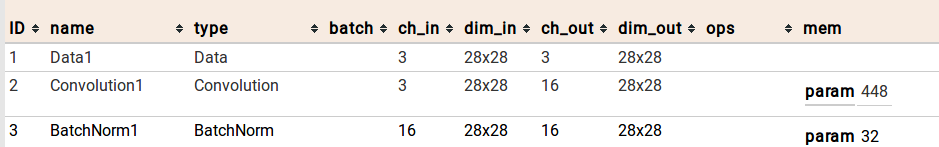
同样的办法



 = scale\_factor \* 

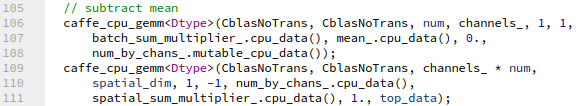
这里面都是16个点，因此总共是32个权重。

和这里吻合，只是还有一个疑问，scale\_factor应该也要算一个数值的，但是感觉权重没有算在内。

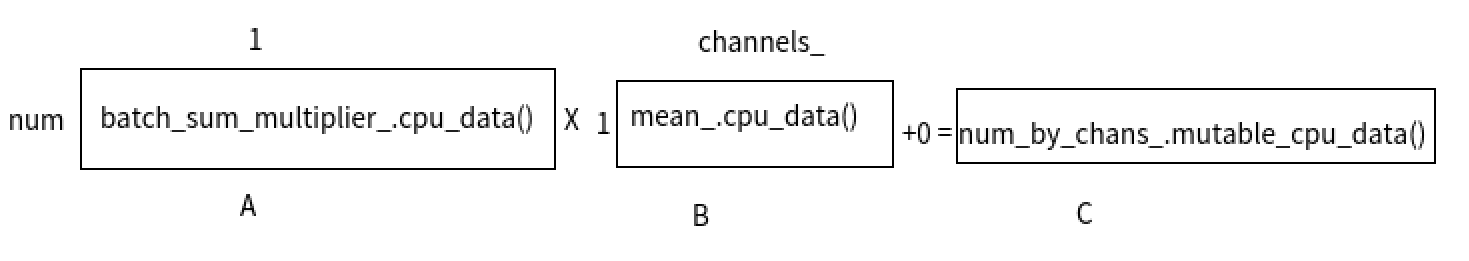


执行减法得到num\_by\_chans\_.mutable\_cpu\_data()矩阵

代码：



第106行等价如下



num=1,channels\_=16

batch\_sum\_multiplier\_.cpu\_data()是A，它是1行，1列的

是B，他是1行，16列

，是C，他是1行16列，

Alpha=1，beta=0

由公式：

经过该操作后：

C=A×B，代入有：



=×

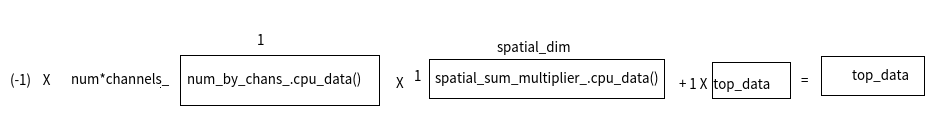
理解为，这个还是均值的操作，只是乘了一个固定系数。

执行减法更新top\_data矩阵，

代码：



代码等价如下：



channels\_是16

 他是32x32图像大小的

A：，行数16，列数1

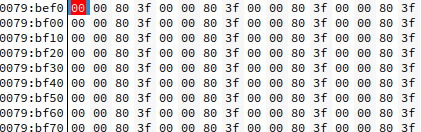
B:行数1,列数：1024,

Alpha=-1，beta=1

C： ，它本身是16行，1024列

C=-A×B+C

其中B，spatial\_sum\_multiplier\_是一副图像的大小空间，一般初始化为1.

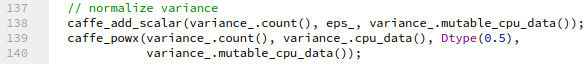


这里面初始化为 00 00 80 3f，这个是浮点数的存储方式，实际的浮点数是“”

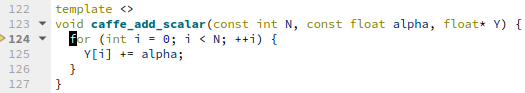
因此这里完成了一次减均值的操作，完毕。

与输入数据无关，将variance变量，对每个值求开方，还是16个点

代码：



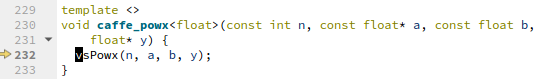
第138行：



实现variance\_.mutable\_cpu\_data()+eps\_的功能。

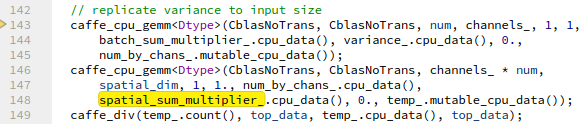
注意，eps是非常小的数值，这条函数其实，可以不用考虑。





复制系数，将放大系数扩充乘矩阵16×1024大小

代码：



A：

行数：num=1,列数1

B:

行数：1, 列数channels\_=16

Alpha=1，beta=0.

C：

根据公式：

有=×

相当于是将variance\_放大了一个系数倍，注意这个num\_by\_chans\_其实可以理解成一个临时变量吧。



代码理解：

A： 行数：16,列数1

B： 行数1,列数1024

Alpha=1,beta=0

C：

有

=×

其实就是之前减完均值后，对top\_data再做除法

代码：

进行除法操作，将top\_data/temp\_.cpu\_data()=top\_data



进行复制

代码：



小提示：

其中Y是目的，X是源

归一化完毕

基本思想总结一下：实际上就是按照各自的通道，减掉一个均值，然后再乘以一个系数。

###### Batch Normal中Log打印显示

输入数据：

输出数据：

其中均值：

系数：

公式变换： = （-）然后在除以，即可得到。

举例说明：



可以均值：0.006143, 系数是127.374954

输入 bottom =0.

代入公式，有 top\_data= （0-0.006143）/127.374954 = -0.000048

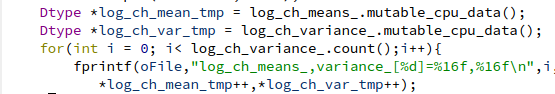
正好是log的打印数值，完毕，其他的应该也一样。



小总结：如果后续需要batchNormal处理，那么只需要把每个通道的means系数，和variance系数提供一下即可，最好是variance系数，取一个倒数，这样对于FPGA来说是非常方便的。

后面应该考虑batchNormal，也要进行Int8的量化操作，这部分工作放在后面再做。

经过batch Normal处理后，附属代码



###### Batch Normal归一化公式

FeatureIn表示输入数据，

FeatureOut表示输出数据。

每个通道的FeatureIn-log\_ch\_means\_各自通道的means,取绝对值，然后再/variance\_(也是各自通道的variance\_)



##### BatchNormal的量化工作

以量化为例，学习在caffe中如何添加新的类

##### Scale的后续问题，也需要int8量化工作

特别提示：Scale，没有权重参数，具体代码没细看，应该不难。

#### Eltwise\_layer代码分析

代码如下：

## Caffe 学习中欠下的知识点

### 容器,迭代器，向量的理解

在源代码里面如下：

### 解

在源代码里面如下：

### 如何在C++下删除文件夹，再创建文件夹

最简单的方法，在C++下调用system（）函数，这个C也是通用的，系统函数的头文件

#include <stdlib.h>

#include <stdio.h>

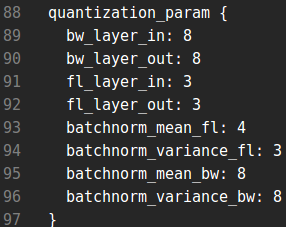


这种方式即可，非常的简单，亲侧有效果。

## Caffe的二次开发工作

### 在Caffe中添加新的参数

比如量化中原来有如下参数，图左，需要添加成如下参数，图右



添加的思路是，先修改caffe.proto文件，增加参数，然后再根据新增加的参数对其他文件的影响进行相关修改。

#### Caffe的proto文件分析

在src/caffe/proto目录下面有个caffe.proto文件，除此之外还有一个caffe.pb.h和caffe.pb.cc文件，这2个文件是通过caffe.proto文件自动生成的。

Caffe.proto文件是消息格式文件，后缀名师proto，也叫消息协议原型定义文件，在该文件中可以通过描述性语言来定义程序中需要用到的数据格式。

##### Protobuf全称-Google Protocol Buffers

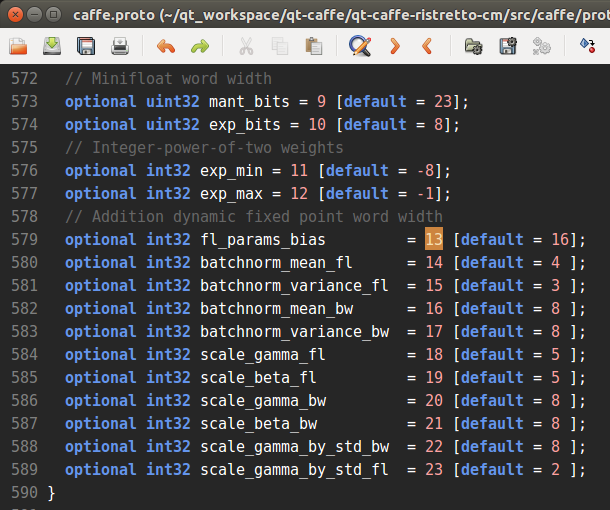
这是一个开源的项目，是一种结构化的数据存储格式。速度快，效率高，如果要使用ProtoBuf库，要自己编一个.proto文件。定义我们程序中需要处理的结构化数据，在Protobuf中，结构化数据被称为Message。在一个.proto文件中可以定义多个消息类型。用Protobuf编译器(protoc.exe)将.proto文件编译成目标语言，会生成对应的.h文件和.cc文件，.proto文件中的每一个消息有(message)一个对应的类

#### 操作方法

操作步骤如下

##### 找到src/caffe/proto/caffe.proto文件如下修改

第579行为以前代码，后面580到589行为新增代码。



##### Blob数据结构

#### 零散

##### Blob数据结构

Blob本质上是一个4维数组，包含channels，num，height,width。

Blobs[0]->channels(相当于channel input)

Blobs[0]->num(相当于channel output)

Blobs[0]->height(用作权重时，就是kernel\_size，比如5,7,3,1等等)

Blobs[0]->width(用作权重时，就是kernel\_size，比如5,7,3,1等等)，采用行优先的存储方式。

##### flag数据结构

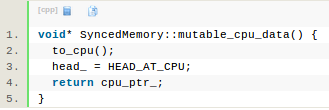
flag是google推出的命令行参数的库。

##### Caffe中mutable\_cpu\_data和cpu\_data的区别

一开始看代码的时候会纳闷，为啥caffe里头又一个

cpu\_data，还要有一个mutable\_cpu\_data，其实从字面就可以看出来这个mutable就是易变的意思啦。二话不说翻出源代码看看究竟有啥区别:





他们之间就相差一个

head\_ = HEAD\_AT\_CPU;

这是为什么呢？为啥需要这样做？

我想原因就是

当你想读取数据的时候请使用cpu\_data

当然想修改数据的时候请你使用mutable\_cpu\_data。

这样就提示系统数据我改过啦，你要小心了的意思(我只知道数据一定在CPU上)。

另外还有一个 gpu\_data和mutable\_gpu\_data，也是这样的原因。

## Makefile总结

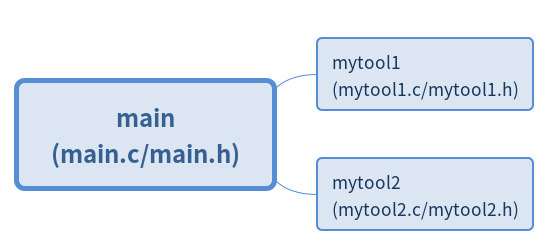
Makefile 的灵魂：



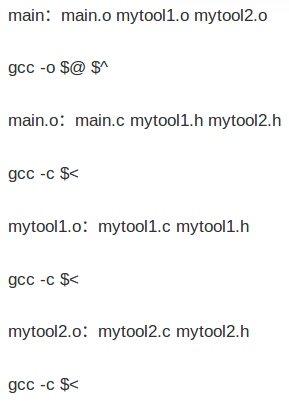
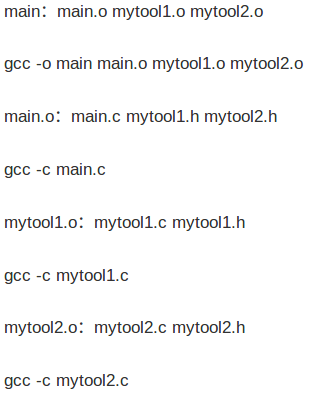
### 举例分析学Makefile

#### 实例1

假设main函数调用了mytool1和mytool2，关系如下：



根据依赖关系可写成如下Makefile



Makefile有三个非常有用的变量。分别是$@，$^，$<代表的意义分别是：

#### $@--目标文件

$@--目标文件

#### $^--所有的依赖文件

$^--所有的依赖文件

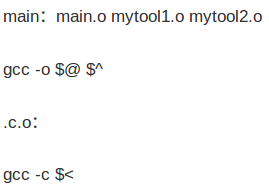
#### $<--第一个依赖文件

$<--第一个依赖文件

#### .c.o:表示所有的.o文件都依赖与相应的.c文件的

上图中，右侧给出了替换方法。

因此还有替换方法3：

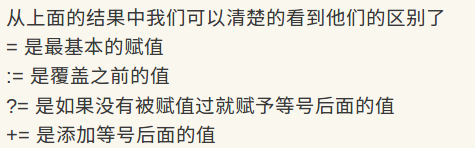


#### 通配符%和\*区别

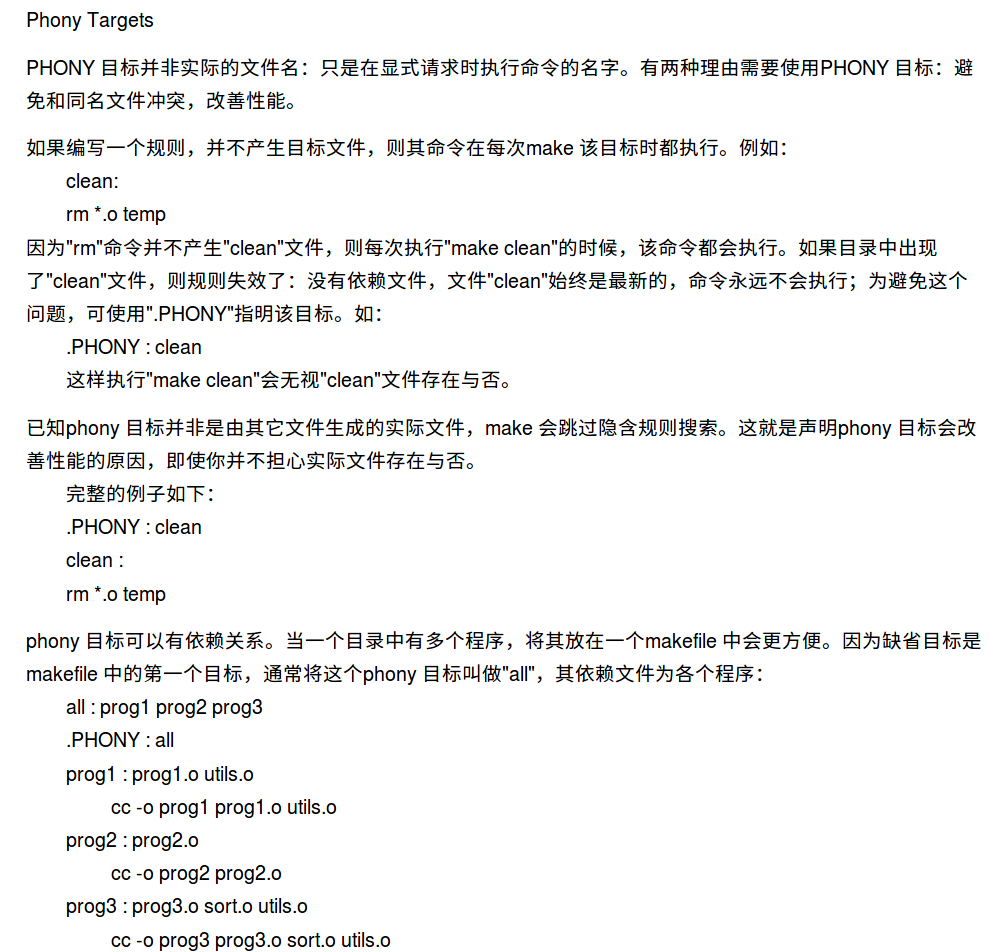
Makefile中的%表示通配符，而\*表示任意字符   
举例，%.o表示匹配所有的.o文件，注意是用于匹配的。   
\*.o是表示所有的.o文件。   
用途方面：前者一般用作目标，后者一般用作删除

#### =,与：=，+=等的区别





#### PHONY介绍



### Makefile中重要变量

#### all命令执行的依赖

需要的依赖如下：

在源代码里面如下：

## Makefile执行Caffe的方法

### 参考资料Makefile中使用宏定义进行条件编译

在源代码里面如下：  
#ifdef   MACRONAME   
//可选代码   
#endif

那在makefile里面   
gcc   -D   MACRONAME=MACRODEF   
或者   
gcc   -D   MACRONAME

这样就定义了预处理宏，编译的时候可选代码就会被编译进去了。

对于GCC编译器，有如下选项：   
        -D macro=string，等价于在头文件中定义：#define   macro   string。例如：-D TRUE=true，等价于：#define   TRUE   true  
        -D macro，等价于在头文件中定义：#define   macro   1，实际上也达到了定义：#define   macro的目的。例如：-D LINUX，等价于：#define   LINUX   1（与#define   LINUX作用类似）。  
        --define-macro   macro=string与-D macro=string作用相同。

注意：文件解析的时候，是-DXXX的方式

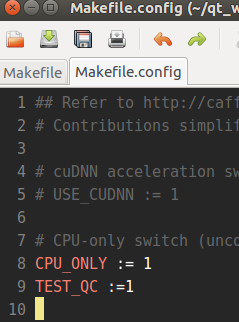
### Caffe中Makefile添加参数宏的方法

模板标题2正文

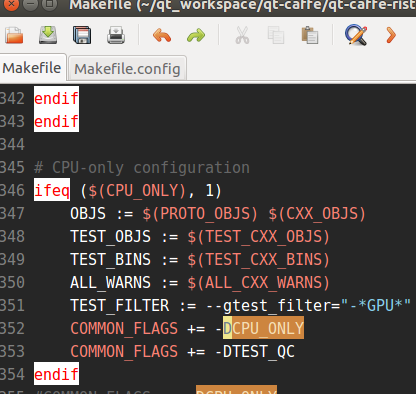
#### 如何添加参数宏

##### 在Makefile.config文件中添加如下

添加新的宏TEST\_QC :=1

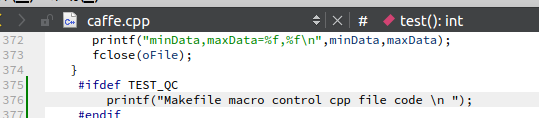


##### 在Makefile中添加宏名称



添加-DTEST\_QC ,其中-D表示添加宏的意思，后面TEST\_QC是真正的宏名称

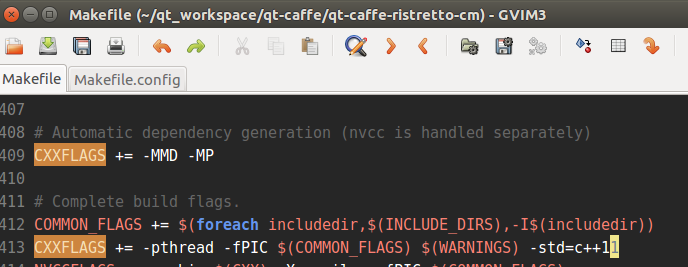
##### 在Makefile中添加宏名称



### Caffe中Makefile设置采用C++11编译的方法

特别注意：目前看，这个方法不太好用

方法如下：



只需要在原来的Makefile中CXXFLAGS，413行，后面添加-std=c++11，

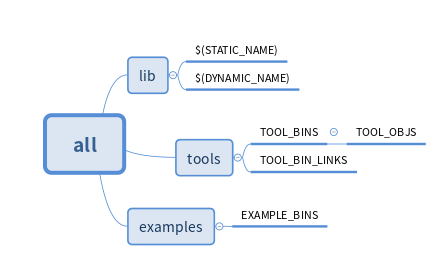
然后别忘记，在重新make clean, make all，一下，都在终端运行，试过在qt上跑不行，不知道为啥。

### Caffe中Makefile中make all执行过程

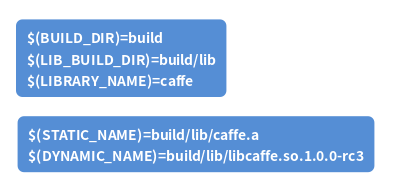
分析如下

#### all命令执行的依赖

需要的依赖如下：



#### lib



#### tools

##### TOOL\_BINS

分析如下：

第1：



.o=.bin，表示.bin的值都替换成.o

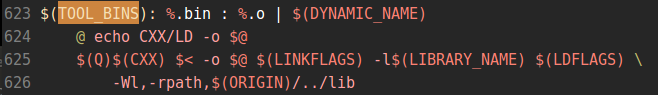




表示在build/tools/下面，找到所有的.cpp文件，然后再将.cpp文件替换成.o文件。

最后在131行，表示把所有的.bin文件变成.o文件

第2,



###### $(Q)分析

经过打印，$(Q)就是@的意思，在Makefile中@表示执行命令时，不输出命令本身，只输出结果。

###### $(CXX)分析

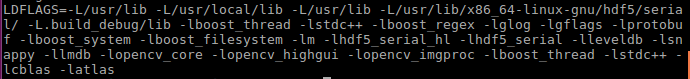
经过打印，$(CXX)就是g++的意思。

%.bin表示依赖于所有的.bin文件，%.o依赖于所有的.o文件。





LDFLAGS=如下



ORIGIN表示



##### TOOL\_BIN\_LINKS

分析如下：



##### EXAMPLE\_BINS文件

分析如下，思路和tools一致的。

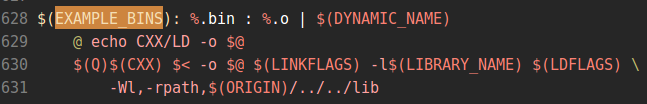
第1：







第2：



## TWN三值化网络找准16个正确源方法

注意，本次的实验，是以ResNet20网络为例，数据集是cifar10,

取消了BatchNormal和Scale模块，目的是为了找到16张图片，能准确的预测到结果，并且这16张图片，都能够执行caffe Test，随意获取任意层的数值，方便做FPGA TWN加速使用。

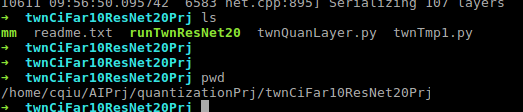
Why 16张，因为FPGA的批处理就是16,这样做能达到性能利用最大化。

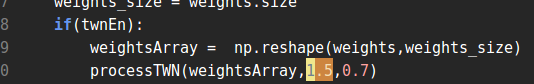
### 具体思路

首先进行量化工作，设置阈值。

#### 量化工作找到准确率高的阈值，生成caffemodel文件

首先进行量化工作，设置阈值，经过实验发现，这个值最高。



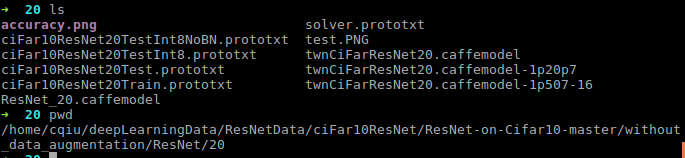


取1.5,0.7

然后执行：



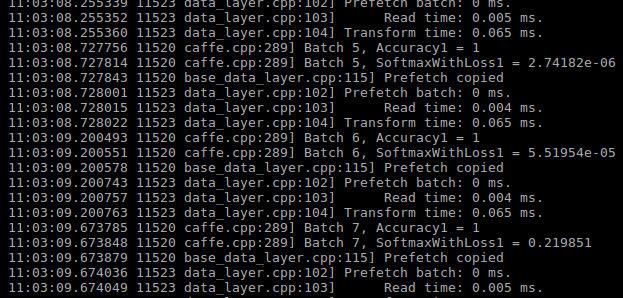
生成阈值结果。



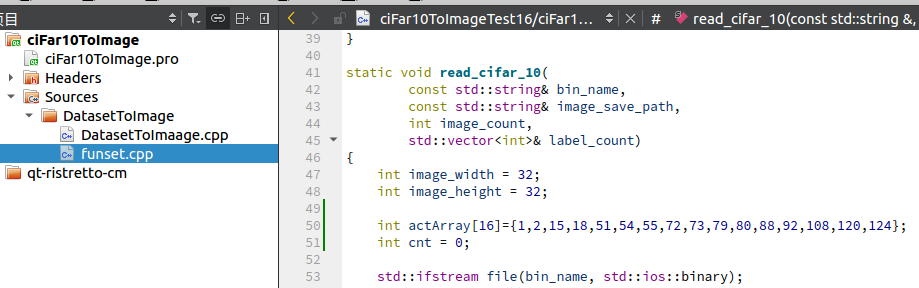
在这个文件下。



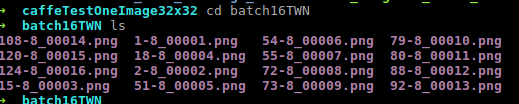
#### 找到正确预测时的图片标号



#### 执行ciFar10ToImage工程，生成16张图片



生成结果：

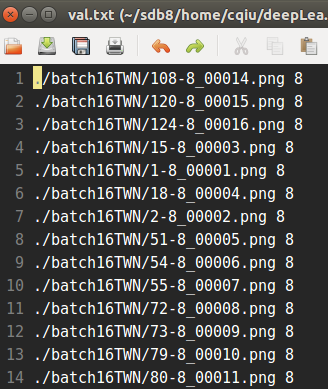


整理玩后，放入到batch16TWN下面。

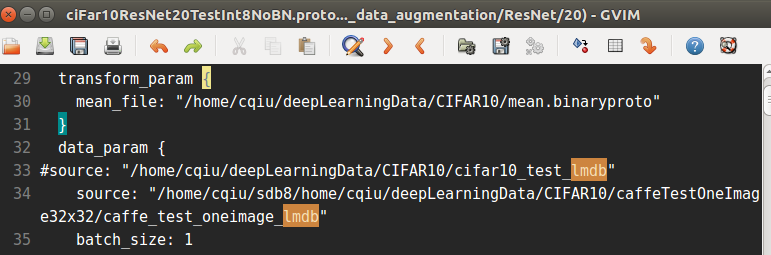
#### 执行create\_imagenet\_small\_val.sh，生成对应的lmdb文件



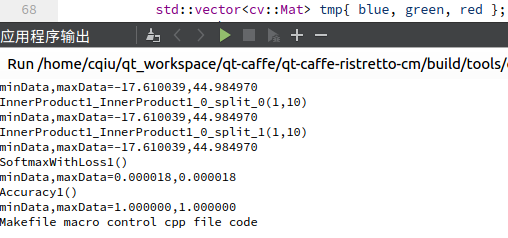
注意要改好val.txt文件，按照如下格式



#### 修改prototxt文件，改成希望的lmdb文件souce



#### 执行caffe\_test，随心所欲进行测试



或者：



完毕。

## 模板标题1

模板标题1正文

### 模板标题2

模板标题2正文

#### 模板标题3

模板标题3正文

##### 模板标题4

模板标题4正文

###### 模板标题5

模板标题5正文

模板标题6

模板标题6正文

模板标题6

阿斯顿发放阿斯蒂芬

阿斯蒂芬

撒旦法

模板标题7

模板标题7正文