rewriting-caffe recode

按照一步一步的来实现caffe工程

1. 编写CMakeLists.txt

第一步编写CMakeLists.txt文档，有两个目的：

- 为了将整个工程结构有个大概了解，
- 在编写每个模块时，能跑通整个工程

2. 文件功能说明

```
#include "caffe/proto/caffe.pb.h"
```

包含caffe空间定义
如果不添加该文件的include，则会产生caffe没定义; 因为它是第一个生成的.h文件，且含有namespace caffe {}的caffe命名空间定义

3. caffe::string 的来源

是有namespace caffe{
std:string
}

产生的; 这个申明在common.hpp中，
因此开始先写common.hpp

4. lint是C/C++ 强大的测试工具

lint检查C程序中潜在的错误，包括（但不限于）可疑的类型组合、未使用的变量、不可达的代码以及不可移植的代码。lint会产生一系列程序员有必要从头到尾仔细阅读的诊断信息。使用lint的好处是：1.它可以检查出被编译器漏掉的错误; 2.可以关联很多文件进行错误的检查和代码分析,具有较强大灵活性.lint可以检查的错误类型大体如下:

- 可能的空指针
- 在释放内存后使用了指向该内存的指针
- 赋值次序问题
- 拼写错误
- 被0除
- 失败的case语句(遗漏了break语句)
- 不可移植的代码(依赖了特定的机器实现)
- 宏参数没有使用圆括号
- 符号的丢失
- 异常的表达式
- 变量没有初始化
- 可疑的判断语句(例如,if(x=0))
- printf/scanf的格式检查

因此，代码中注释带有//NOLINT时， 表示That's a comment. In this case, it's a comment designed to be read by a static analysis tool to tell it to shut up about this line. 让静态分析工具不要在意这一行.

5. CPU_ONLY 是怎么从cmake传给C++内部的

在CaffeConfig.cmake.in中传入的, **具体怎么转化为#define CPU_ONLY 的，需要考察**
cmake/Templates/CaffeConfig.cmake.in:53:set(Caffe_CPU_ONLY @CPU_ONLY@)

ConfigGen.cmake 产生CaffeTargets.cmake, 这个文件中有：

```
set_target_properties(caffe PROPERTIES
  INTERFACE_COMPILE_DEFINITIONS "USE_LMDB;USE_LEVELDB;CPU_ONLY;USE_OPENCV"
  INTERFACE_INCLUDE_DIRECTORIES "/usr/include;/usr/local/include;/usr/local/include;/usr/include;/home/
)
```

CPU_ONLY 在这传进去的
INTERFACE_COMPILE_DEFINITIONS： 官方说明：

List of public compile definitions requirements for a library.

Targets may populate this property to publish the compile definitions required to compile against the headers for the target. The target_compile_definitions() command populates this property with values given to the PUBLIC and INTERFACE keywords. Projects may also get and set the property directly.

When target dependencies are specified using target_link_libraries(), CMake will read this property from all target dependencies to determine the build properties of the consumer.

Contents of INTERFACE_COMPILE_DEFINITIONS may use “generator expressions” with the syntax $<...>. See the [cmake-generator-expressions(7)] manual for available expressions. See the [cmake-buildsystem(7)] -manual for more on defining buildsystem properties.

[cmake-generator-expressions(7)]---: 也就是说这个INTERFACE_COMPILE_DEFINITIONS或者target_compile_definition的设置，会在编译的时候控制一些用宏定义为判断条件的代码是否编译编译，也就是控制条件性的定义
Generator expressions are evaluated during build system generation to produce information specific to each build configuration.

Generator expressions are allowed in the context of many target properties, such as LINK_LIBRARIES, INCLUDE_DIRECTORIES, COMPILE_DEFINITIONS and others. They may also be used when using commands to populate those properties, such as target_link_libraries(), target_include_directories(), target_compile_definitions() and others.

They enable conditional linking, conditional definitions used when compiling, conditional include directories, and more. The conditions may be based on the build configuration, target properties, platform information or any other queryable information.

Generator expressions have the form $<...>. To avoid confusion, this page deviates from most of the CMake documentation in that it omits angular brackets <...> around placeholders like condition, string, target, among others.

Generator expressions can be nested, as shown in most of the examples below.

6. device_alternate.hpp

配置 cuda的错误检查宏，以及配置block数目和threads数目

7. common.hpp 解析

这里有点绕，特别是Caffe类里面有个RNG，RNG这个类里面还有个Generator类
在RNG里面会用到Caffe里面的Get()函数来获取一个新的Caffe类的实例（如果不存在的话）。
然后RNG里面用到了Generator。Generator是实际产生随机数的。
（1）Generator类
该类有两个构造函数：

```
Generator()//用系统的熵池或者时间来初始化随机数
explicit Generator(unsigned int seed)// 用给定的种子初始化
```

（2）RNG类
RNG类内部有generator_，generator_是Generator类的实例
该类有三个构造函数：

```
RNG(); //利用系统的熵池或者时间来初始化RNG内部的generator_
explicit RNG(unsigned int seed); // 利用给定的seed来初始化内部的generator_
explicit RNG(const RNG&);// 用其他的RNG内部的generator_设置为当前的generator_
```

（3）Caffe类
1)含有一个Get函数，该函数利用Boost的局部线程存储功能实现

```
// Make sure each thread can have different values.
// boost::thread_specific_ptr是线程局部存储机制
// 一开始的值是NULL
static boost::thread_specific_ptr<Caffe> thread_instance_;

Caffe& Caffe::Get() {
  if (!thread_instance_.get()) {// 如果当前线程没有caffe实例
    thread_instance_.reset(new Caffe());// 则新建一个caffe的实例并返回
  }
  return *(thread_instance_.get());
```

2)此外该类还有

```
SetDevice
DeviceQuery
mode
set_mode
set_random_seed
solver_count
set_solver_count
root_solver
set_root_solver
```

等成员函数
3)内部还有一些比较技巧性的东西比如：

```
// CUDA: various checks for different function calls.
//  防止重定义cudaError_t，这个以前在linux代码里面看过
// 实际上就是利用变量的局部声明
#define CUDA_CHECK(condition) \
  /* Code block avoids redefinition of cudaError_t error */ \
  do { \
    cudaError_t error = condition; \
    CHECK_EQ(error, cudaSuccess) << " " << cudaGetErrorString(error); \
  } while (0)
```

6.1 boost::thread_specific_ptr是线程局部存储机制

大多数函数都不是可重入的。这也就是说在某一个线程已经调用了一个函数时，如果你再调用同一个函数，那么这样是不安全的。一个不可重入的函数通过连续的调用来保存静态变量或者是返回一个指向静态数据的指针。 举例来说，std::strtok就是不可重入的，因为它使用静态变量来保存要被分割成符号的字符串。

有两种方法可以让不可重用的函数变成可重用的函数。第一种方法就是改变接口，用指针或引用代替原先使用静态数据的地方。比方说，POSIX定义了strok_r，std::strtok中的一个可重入的变量，它用一个额外的char**参数来代替静态数据。这种方法很简单，而且提供了可能的最佳效果。但是这样必须改变公共接口，也就意味着必须改代码。另一种方法不用改变公有接口，而是用本地存储线程（thread local storage）来代替静态数据（有时也被成为特殊线程存储，thread-specific storage）。

Boost线程库提供了智能指针boost::thread_specific_ptr来访问本地存储线程。每一个线程第一次使用这个智能指针的实例时，它的初值是NULL，所以必须要先检查这个它的只是否为空，并且为它赋值。Boost线程库保证本地存储线程中保存的数据会在线程结束后被清除。

List5是一个使用boost::thread_specific_ptr的简单例子。其中创建了两个线程来初始化本地存储线程，并有10次循环，每一次都会增加智能指针指向的值，并将其输出到std::cout上（由于std::cout是一个共享资源，所以通过互斥体进行同步）。main线程等待这两个线程结束后就退出。从这个例子输出可以明白的看出每个线程都处理属于自己的数据实例，尽管它们都是使用同一个boost::thread_specific_ptr。

----------另一个解释

对于线程局部存储的概念，正如字面意思，每个变量在每个线程中都有一份独立的拷贝。通过使用线程局部存储技术，可以避免线程间的同步问题，并且不同的线程可以使用不同的日志设置。通过 Boost 库的智能指针 boost::thread_specific_ptr 来存取线程局部存储，每个线程在第一次试图获取这个智能指针的实例时需要对它进行初始化，并且线程局部存储的数据在线程退出时由 Boost 库来释放。

使用线程局部变量的多线程日志的优势：

使用 static 的线程局部变量很容易能实现线程级别的单例日志系统；
通过智能指针 boost::thread_specific_ptr 来存取线程局部存储的生成和清除工作简单方便；
使用线程局部变量很容易实现对一个已有的单例日志系统进行多线程支持的改造，并且不用改动任何原来的日志接口；

错误记录

1. proto编译错误

```
[ 14%] Building CXX object src/caffe/CMakeFiles/caffeproto.dir/__/__/include/caffe/proto
c++: fatal error: no input files
compilation terminated.
/bin/sh: 1: -fPIC: not found
/bin/sh: 1: -Wall: not found
src/caffe/CMakeFiles/caffeproto.dir/build.make:74: recipe for target 'src/caffe/CMakeFil
make[2]: *** [src/caffe/CMakeFiles/caffeproto.dir/__/__/include/caffe/proto/caffe.pb.cc.
CMakeFiles/Makefile2:183: recipe for target 'src/caffe/CMakeFiles/caffeproto.dir/all' fa
make[1]: *** [src/caffe/CMakeFiles/caffeproto.dir/all] Error 2
Makefile:127: recipe for target 'all' failed
make: *** [all] Error 2

查看  prtoc --version  2.6.1,   不是版本的问题
```

根据提示可以发现：

-fPIC not found

-Wall not found

查看错误处代码：

```
src/caffe/CMakeFiles/caffeproto.dir/__/__/include/caffe/proto/caffe.pb.cc.o: include/caf
        @$(CMAKE_COMMAND) -E cmake_echo_color --switch=$(COLOR) --green --progress-dir=/
        cd /home/lshm/Projects/rewrite_caffe/build/src/caffe && /usr/bin/c++   $(CXX_DEF

1) 去掉-fPIC, -Wall  编译成功
2）c++ 换成g++ 也不行

通过打印发现，${CXX_FLAGS}的内容是 ";-fPIC;-Wall"，而一般这两个编译选项是不带";"分号的，
因此赋值时出错；  而CXX_FLAGS是make的系统变量，它是由CMAKE_CXX_FLAGS的值生成的. 而根目录下的CMake
  set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS}" -fPIC -Wall)
对照原来的caffe的CMakeLists.txt是
  set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -fPIC -Wall")
所以修改后错误消失
```