caffe框架中的回调机制以及工厂模式

1、device_query()、train()、test()和time()函数的回调机制

注册表结构

(tools/caffe.cpp)

```
// A simple registry for caffe commands.
typedef int (*BrewFunction)();
typedef std::map<caffe::string, BrewFunction> BrewMap;
BrewMap g_brew_map;
```

首先在这里使用typedef定义了一种函数指针类型，该类型的名字叫做BrewFunction，其所对应的是返回值为int，传入值为void的函数。那我们这张注册表的结构是一个map数据结构，其key为string类型，value的值为BrewFunction类型。

注册机

(tools/caffe.cpp)

```
#define RegisterBrewFunction(func) \
namespace { \
class __Registerer_##func { \
 public: /* NOLINT */ \
  __Registerer_##func() { \
    g_brew_map[#func] = &func; \
  } \
}; \
__Registerer_##func g_registerer_##func; \
}
```

首先说明一点，在caffe中大量使用这种宏定义函数，使用这种宏定义函数可以节省一定的代码，但是会给caffe的初学者造成一定的困惑，其实宏函数的使用非常简单，就是挨个将相应的值给替换成我们输入的值就可以了，但同时也要注意宏函数的一些规定。

两个特殊符号：在这个宏函数中可以看到#和##这两个符号，对于单个#的作用是取改func的字符串的意思，而两个##是将两边的符号合并成一个符号，那我们看下面一个例子，比如我们调用下面这条语句，然后将其扩展开来。

```
RegisterBrewFunction(train)
```

那改语句扩展开来可以得到：

```
namespace{
    class __Registerer_train{
        public:
        __Registerer_train(){
            g_brew_map["train"] = &train;
        }
    };
    __Registerer_train g_registerer_train;
}
```

那我们具体看一下该宏函数做了什么事情。

第一步：定义一个__Registerer_train的类，而且该类的构造函数的作用是向前面的定义的注册表g_brew_map中注册train函数。

第二步：使用__Registerer_train类实例化一个对象，只有通过实例化对象才可以调用类中的构造函数，从而将train函数给注册到注册表中。（记住这一点，因为这在后面的Solver类注册动作和Layer类的注册动作中都有用到这一原理）

注册动作

整个注册动作都是在device_query()、train()、test()、test()函数定义之后进行的，在函数后面调用下面的宏函数完成注册的动作。

```
RegisterBrewFunction(device_query);
RegisterBrewFunction(train);
RegisterBrewFunction(test);
RegisterBrewFunction(time);
```

main函数中的回调机制

main函数进行回调的动作如下面的代码片段所示，main函数通过GetBrewFunction函数去进行回调动作。

(tools/caffe.cpp)

```
if (argc == 2) {
#ifdef WITH_PYTHON_LAYER
    try {
#endif
        return GetBrewFunction(caffe::string(argv[1]))(); //调用回调函数
#ifdef WITH_PYTHON_LAYER
    } catch (bp::error_already_set) {
        PyErr_Print();
        return 1;
    }
#endif
```

下面是GetBrewFunction函数的定义，在这里它真正的调用注册在注册表g_brew_map的函数，从而完成整个main函数的回调机制。

(tools/caffe.cpp)

```
static BrewFunction GetBrewFunction(const caffe::string& name) {
    if (g_brew_map.count(name)) {
        return g_brew_map[name];        //真正的从注册表中调用回调函数
    } else {
        LOG(ERROR) << "Available caffe actions:";
        for (BrewMap::iterator it = g_brew_map.begin();
                it != g_brew_map.end(); ++it) {
            LOG(ERROR) << "\t" << it->first;
        }
        LOG(FATAL) << "Unknown action: " << name;
        return NULL;  // not reachable, just to suppress old compiler warnings.
    }
}
```

2、Solver类和Layer类的工厂模式及其中的回调机制

接下来我们来看一下Solver类和Layer类的工厂模式以及其回调机制，由于Solver类的工厂模式以及回调机制和Layer类很像，可以说几乎一模一样。那在这里我主要是针对Solver类的工厂函数及回调进行分析。

首先我们在train()函数中看到了这一句代码：

(tools/caffe.cpp)

```
shared_ptr<caffe::Solver<float> >
        solver(caffe::SolverRegistry<float>::CreateSolver(solver_param));
```

它的意思是通过调用SolverRegistry类的CreateSolver方法来获得指向Solver类的指针，那很明显我们就要去了解SolverRegistry类以及CreateSolver方法具体做了什么事情。

注册表结构

(include/caffe/solver_factory.hpp)

```
template <typename Dtype>
class SolverRegistry {
 public:
  typedef Solver<Dtype>* (*Creator)(const SolverParameter&);   //typedef定义函数指针类型
  typedef std::map<string, Creator> CreatorRegistry;          //注册表结构
```

首先定义了函数指针类型Creator，该类型所对应的函数的传入参数为const SolverParameter&，返回值为Solver*的指针。我们的注册表同样是采用map数据结构来进行维护的。其key为string，其实就是各个Solver子类的名字，Creator我们可以暂且称其为类的创建函数，主要负责创建子类对象。

```
 static CreatorRegistry& Registry() {                          //注册表读取函数
    static CreatorRegistry* g_registry_ = new CreatorRegistry();  //调用new生成注册表
    return *g_registry_;
  }
```

Registry()函数的作用是注册表读取函数，通过调用该函数我们可以将整张注册表给取出来，那下面我们开始尝试将这张注册表取出来，当然在这里我们要自己写一些测试代码，关于caffe中如何添加自己的代码模块可以参考《caffe工程构建》。

```
  // Adds a creator.
  static void AddCreator(const string& type, Creator creator) {   //注册表内容添加函数
    CreatorRegistry& registry = Registry();
    CHECK_EQ(registry.count(type), 0)
        << "Solver type " << type << " already registered.";
    registry[type] = creator;
  }

  // Get a solver using a SolverParameter.
  static Solver<Dtype>* CreateSolver(const SolverParameter& param) {  //回调函数
    const string& type = param.type();
    CreatorRegistry& registry = Registry();
    CHECK_EQ(registry.count(type), 1) << "Unknown solver type: " << type
        << " (known types: " << SolverTypeListString() << ")";
    return registry[type](param);
  }

  static vector<string> SolverTypeList() {         //注册表信息获取接口
    CreatorRegistry& registry = Registry();
    vector<string> solver_types;
    for (typename CreatorRegistry::iterator iter = registry.begin();
         iter != registry.end(); ++iter) {
      solver_types.push_back(iter->first);
    }
    return solver_types;
  }

 private:
  // Solver registry should never be instantiated - everything is done with its
  // static variables.
  SolverRegistry() {}     //将构造函数放在private中，说明这个类Singleton模式或者不能被实例化
  static string SolverTypeListString() {   //将注册表的中的内容转换为string类型
    vector<string> solver_types = SolverTypeList();
    string solver_types_str;
    for (vector<string>::iterator iter = solver_types.begin();
         iter != solver_types.end(); ++iter) {
      if (iter != solver_types.begin()) {
        solver_types_str += ", ";
      }
      solver_types_str += *iter;
    }
    return solver_types_str;
  }
};
```