Caffemodel解析

阅读：6863　时间：2015-04-19 17:07

因为工作需要最近一直在琢磨Caffe，纯粹新手，写博客供以后查阅方便，请大神们批评指正！

Caffe中，数据的读取、运算、存储都是采用Google Protocol Buffer来进行的，所以首先来较为详细的介绍下Protocol Buffer(PB)。

PB是一种**轻便、高效的结构化数据存储格式**，可以用于结构化数据串行化，很适合做数据存储或 RPC 数据交换格式。它可用于通讯协议、数据存储等领域的**语言无关、平台无关、可扩展的序列化结构数据格式**。是一种效率和兼容性都很优秀的**二进制数据传输格式**，目前提供了 C++、Java、Python 三种语言的 API。Caffe采用的是C++和Python的API。

接下来，我用一个简单的例子来说明一下。

使用PB和 C++ 编写一个十分简单的例子程序。该程序由两部分组成。第一部分被称为Writer，第二部分叫做Reader。Writer 负责将一些结构化的数据写入一个磁盘文件，Reader则负责从该磁盘文件中读取结构化数据并打印到屏幕上。准备用于演示的结构化数据是HelloWorld，它包含两个基本数据：

ID，为一个整数类型的数据；

Str，这是一个字符串。

首先我们需要编写一个proto文件，定义我们程序中需要处理的结构化数据，**Caffe是定义在caffe.proto文件中**。在PB的术语中，结构化数据被称为 Message。proto文件非常类似java或C语言的数据定义。代码清单 1 显示了例子应用中的proto文件内容。

清单 1. proto 文件

```
1 package lm;
2
3 message helloworld
4
5  {
6
7     required int32     id = 1;   // ID
8
9     required string    str = 2;  // str
10
11    optional int32     opt = 3;  // optional field
12
13  }
```

View Code

一个比较好的习惯是认真对待proto文件的文件名。比如将命名规则定于如下：packageName.MessageName.proto

在上例中，package名字叫做 lm，定义了一个消息helloworld，该消息有三个成员，类型为int32的id，另一个为类型为string的成员str。optional是一个可选的成员，即消息中可以不包含该成员，required表明是必须包含该成员。一般在定义中会出现如下三个字段属性：

对于required的字段而言，初值是必须要提供的，否则字段的便是未初始化的。 在Debug模式的buffer库下编译的话，序列化话的时候可能会失败，而且在反序列化的时候对于该字段的解析会总是失败的。所以，对于修饰符为required的字段，请在序列化的时候务必给予初始化。

对于optional的字段而言，如果未进行初始化，那么一个默认值将赋予该字段，当然也可以指定默认值。

对于repeated的字段而言，该字段可以重复多个，谷歌提供的这个 addressbook例子便有个很好的该修饰符的应用场景，即每个人可能有多个电话号码。在高级语言里面，我们可以通过数组来实现，而在proto定义文件中可以使用repeated来修饰，从而达到相同目的。当然，出现0次也是包含在内的。

写好proto文件之后就可以用PB编译器(protoc)将该文件编译成目标语言了。本例中我们将使用C++。假设proto文件存放在 $SRC_DIR 下面，您也想把生成的文件放在同一个目录下，则可以使用如下命令：

```
1  protoc -I=$SRC_DIR --cpp_out=$DST_DIR $SRC_DIR/addressbook.proto
```

View Code

命令将生成两个文件：

lm.helloworld.pb.h， 定义了C++ 类的头文件；

lm.helloworld.pb.cc，C++类的实现文件。

在生成的头文件中，定义了一个 C++ 类 helloworld，后面的 Writer 和 Reader 将使用这个类来对消息进行操作。诸如对消息的成员进行赋值，将消息序列化等等都有相应的方法。

如前所述，Writer将把一个结构化数据写入磁盘，以便其他人来读取。假如我们不使用PB，其实也有许多的选择。一个可能的方法是将数据转换为字符串，然后将字符串写入磁盘。转换为字符串的方法可以使用 sprintf()，这非常简单。数字 123 可以变成字符串”123”。这样做似乎没有什么不妥，但是仔细考虑一下就会发现，这样的做法对写Reader的那个人的要求比较高，Reader 的作者必须了解Writer 的细节。比如”123”可以是单个数字 123，但也可以是三个数字 1、2 和 3等等。这么说来，我们还必须让Writer定义一种分隔符一样的字符，以便Reader可以正确读取。但分隔符也许还会引起其他的什么问题。最后我们发现一个简单的Helloworld 也需要写许多处理消息格式的代码。

如果使用 PB，那么这些细节就可以不需要应用程序来考虑了。使用PB，Writer 的工作很简单，需要处理的结构化数据由 .proto 文件描述，经过上一节中的编译过程后，该数据化结构对应了一个 C++ 的类，并定义在 lm.helloworld.pb.h 中。对于本例，类名为lm::helloworld。

Writer 需要include该头文件，然后便可以使用这个类了。现在，在Writer代码中，**将要存入磁盘的结构化数据由一个lm::helloworld类的对象表示**，它提供了一系列的 get/set 函数用来修改和读取结构化数据中的数据成员，或者叫field。

当我们需要将该结构化数据保存到磁盘上时，类 lm::helloworld 已经提供相应的方法来把一个复杂的数据变成一个字节序列，我们可以将这个字节序列写入磁盘。

对于想要读取这个数据的程序来说，也只需要使用类 lm::helloworld 的相应反序列化方法来将这个字节序列重新转换会结构化数据。这同我们开始时那个“123”的想法类似，不过PB想的远远比我们那个粗糙的字符串转换要全面，因此，我们可以放心将这类事情交给PB吧。程序清单 2 演示了 Writer 的主要代码。

清单 2. Writer 的主要代码

```
#include "lm.helloworld.pb.h"

…

int main(void)

{

 lm::helloworld msg1;

 msg1.set_id(101);          //设置id

 msg1.set_str(“hello”);   //设置str

 // 向磁盘中写入数据流fstream

 fstream output("./log", ios::out | ios::trunc | ios::binary);

 if (!msg1.SerializeToOstream(&output)) {

     cerr << "Failed to write msg." << endl;

     return -1;

 }

 return 0;

}
```

View Code

Msg1 是一个helloworld类的对象，set_id()用来设置id的值。SerializeToOstream将对象序列化后写入一个fstream流。我们可以写出Reader代码，程序清单3列出了 reader 的主要代码。

清单 3. Reader的主要代码

```
#include "lm.helloworld.pb.h"

…

 void ListMsg(const lm::helloworld & msg) {

  cout << msg.id() << endl;

  cout << msg.str() << endl;

 }

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

  lm::helloworld msg1;

  {

    fstream input("./log", ios::in | ios::binary);

    if (!msg1.ParseFromIstream(&input)) {

      cerr << "Failed to parse address book." << endl;

      return -1;

    }

  }

  ListMsg(msg1);

  …

 }
```

View Code

同样，Reader 声明类helloworld的对象msg1，然后利用ParseFromIstream从一个fstream流中读取信息并反序列化。此后，ListMsg中采用get方法读取消息的内部信息，并进行打印输出操作。

运行Writer和Reader的结果如下：

```
>writer
```

```
>reader
```

```
101
```

```
Hello
```

Reader 读取文件 log 中的序列化信息并打印到屏幕上。这个例子本身并无意义，但只要稍加修改就可以将它变成更加有用的程序。比如将磁盘替换为网络 socket，那么就可以实现基于网络的数据交换任务。而存储和交换正是PB最有效的应用领域。

到这里为止，我们只给出了一个简单的没有任何用处的例子。在实际应用中，人们往往需要定义更加复杂的 Message。我们用“复杂”这个词，不仅仅是指从个数上说有更多的 fields 或者更多类型的 fields，而是指更加复杂的数据结构：**嵌套 Message**，Caffe.proto文件中定义了大量的嵌套Message。使得Message的表达能力增强很多。代码清单 4 给出一个嵌套 Message 的例子。

清单 4. 嵌套 Message 的例子

```
message Person {
 required string name = 1;
 required int32 id = 2;        // Unique ID number for this person.
 optional string email = 3;
 enum PhoneType {
   MOBILE = 0;
   HOME = 1;
   WORK = 2;
 }

 message PhoneNumber {
   required string number = 1;
   optional PhoneType type = 2 [default = HOME];
 }
 repeated PhoneNumber phone = 4;
}
```

View Code

在 Message Person 中，定义了嵌套消息 PhoneNumber，并用来定义 Person 消息中的 phone 域。这使得人们可以定义更加复杂的数据结构。

以上部分参考网址：http://www.ibm.com/developerworks/cn/linux/l-cn-gpb/ (http://www.ibm.com/developerworks/cn/linux/l-cn-gpb/)

在Caffe中也是类似于上例中的Writer和Reader去读写PB数据的。接下来，具体说明下Caffe中是如何存储Caffemodel的。在Caffe主目录下的**solver.cpp**文件中的一段代码展示了Caffe是如何存储Caffemodel的，代码清单5如下：

清单 5. Caffemodel存储代码

```
template <typename Dtype>

void Solver<Dtype>::Snapshot() {

  NetParameter net_param;    // NetParameter为网络参数类

  // 为了中间结果，也会写入梯度值

  net_->ToProto(&net_param, param_.snapshot_diff());

  string filename(param_.snapshot_prefix());

  string model_filename, snapshot_filename;

  const int kBufferSize = 20;

  char iter_str_buffer[kBufferSize];

  // 每训练完1次，iter_就加1

snprintf(iter_str_buffer, kBufferSize, "_iter_%d", iter_ + 1);

  filename += iter_str_buffer;

  model_filename = filename + ".caffemodel"; //XX_iter_YY.caffemodel

  LOG(INFO) << "Snapshotting to " << model_filename;

  // 向磁盘写入网络参数

  WriteProtoToBinaryFile(net_param, model_filename.c_str());

  SolverState state;

  SnapshotSolverState(&state);

  state.set_iter(iter_ + 1);    //set

  state.set_learned_net(model_filename);

  state.set_current_step(current_step_);

  snapshot_filename = filename + ".solverstate";

  LOG(INFO) << "Snapshotting solver state to " << snapshot_filename;

  // 向磁盘写入网络state

  WriteProtoToBinaryFile(state, snapshot_filename.c_str());

}
```

View Code

在清单5代码中，我们可以看到，其实Caffemodel存储的数据也就是网络参数net_param的PB，Caffe可以保存每一次训练完成后的网络参数，我们可以通过**XX.prototxt**文件来进行参数设置。在这里的 WriteProtoToBinaryFile函数与之前HelloWorld例子中的Writer函数类似，在这就不在贴出。那么我们只要弄清楚NetParameter类的组成，也就明白了Caffemodel的具体数据构成。在caffe.proto这个文件中定义了NetParameter类，如代码清单6所示。

清单6. Caffemodel存储代码

```
message NetParameter {

  optional string name = 1;   // 网络名称

  repeated string input = 3;  // 网络输入input blobs

  repeated BlobShape input_shape = 8; // The shape of the input blobs

  // 输入维度blobs，4维(num, channels, height and width)

  repeated int32 input_dim = 4;

  // 网络是否强制每层进行反馈操作开关

// 如果设置为False，则会根据网络结构和学习率自动确定是否进行反馈操作

  optional bool force_backward = 5 [default = false];

// 网络的state，部分网络层依赖，部分不依赖，需要看具体网络

  optional NetState state = 6;

  // 是否打印debug log

  optional bool debug_info = 7 [default = false];

  // 网络层参数，Field Number 为100，所以网络层参数在最后

  repeated LayerParameter layer = 100;

  // 弃用: 用 'layer' 代替

  repeated V1LayerParameter layers = 2;

}

// Specifies the shape (dimensions) of a Blob.

message BlobShape {

  repeated int64 dim = 1 [packed = true];
```

```
42
43 }
44
45 message BlobProto {
46
47   optional BlobShape shape = 7;
48
49   repeated float data = 5 [packed = true];
50
51   repeated float diff = 6 [packed = true];
52
53   optional int32 num = 1 [default = 0];
54
55   optional int32 channels = 2 [default = 0];
56
57   optional int32 height = 3 [default = 0];
58
59   optional int32 width = 4 [default = 0];
60
61 }
62
63
64
65 // The BlobProtoVector is simply a way to pass multiple blobproto instances
66
67 around.
68
69 message BlobProtoVector {
70
71   repeated BlobProto blobs = 1;
72
73 }
74
75 message NetState {
76
77   optional Phase phase = 1 [default = TEST];
78
79   optional int32 level = 2 [default = 0];
80
81   repeated string stage = 3;
82
83 }
84
85 message LayerParameter {
86
87   optional string name = 1;   // the layer name
88
89   optional string type = 2;   // the layer type
90
91   repeated string bottom = 3; // the name of each bottom blob
92
93   repeated string top = 4;    // the name of each top blob
94
95   // The train/test phase for computation.
```

```
 96
 97   optional Phase phase = 10;
 98
 99   // Loss weight值：float
100
101   // 每一层为每一个top blob都分配了一个默认值，通常是0或1
102
103   repeated float loss_weight = 5;
104
105   // 指定的学习参数
106
107   repeated ParamSpec param = 6;
108
109   // The blobs containing the numeric parameters of the layer.
110
111   repeated BlobProto blobs = 7;
112
113   // included/excluded.
114
115   repeated NetStateRule include = 8;
116
117   repeated NetStateRule exclude = 9;
118
119   // Parameters for data pre-processing.
120
121   optional TransformationParameter transform_param = 100;
122
123   // Parameters shared by loss layers.
124
125   optional LossParameter loss_param = 101;
126
127   // 各种类型层参数
128
129   optional AccuracyParameter accuracy_param = 102;
130
131   optional ArgMaxParameter argmax_param = 103;
132
133   optional ConcatParameter concat_param = 104;
134
135   optional ContrastiveLossParameter contrastive_loss_param = 105;
136
137   optional ConvolutionParameter convolution_param = 106;
138
139   optional DataParameter data_param = 107;
140
141   optional DropoutParameter dropout_param = 108;
142
143   optional DummyDataParameter dummy_data_param = 109;
144
145   optional EltwiseParameter eltwise_param = 110;
146
147   optional ExpParameter exp_param = 111;
148
149   optional HDF5DataParameter hdf5_data_param = 112;
```

```
150
151   optional HDF5OutputParameter hdf5_output_param = 113;
152
153   optional HingeLossParameter hinge_loss_param = 114;
154
155   optional ImageDataParameter image_data_param = 115;
156
157   optional InfogainLossParameter infogain_loss_param = 116;
158
159   optional InnerProductParameter inner_product_param = 117;
160
161   optional LRNParameter lrn_param = 118;
162
163   optional MemoryDataParameter memory_data_param = 119;
164
165   optional MVNParameter mvn_param = 120;
166
167   optional PoolingParameter pooling_param = 121;
168
169   optional PowerParameter power_param = 122;
170
171   optional PythonParameter python_param = 130;
172
173   optional ReLUParameter relu_param = 123;
174
175   optional SigmoidParameter sigmoid_param = 124;
176
177   optional SoftmaxParameter softmax_param = 125;
178
179   optional SliceParameter slice_param = 126;
180
181   optional TanHParameter tanh_param = 127;
182
183   optional ThresholdParameter threshold_param = 128;
184
185   optional WindowDataParameter window_data_param = 129;
186
187 }
```

View Code

那么接下来的一段代码来演示如何解析Caffemodel，我解析用的model为MNIST手写库训练后的model，Lenet_iter_10000.caffemodel。

清单7. Caffemodel解析代码

```
#include <stdio.h>
#include <string.h>
#include <fstream>
#include <iostream>
#include "proto/caffe.pb.h"

using namespace std;
using namespace caffe;

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

 caffe::NetParameter msg;

 fstream input("lenet_iter_10000.caffemodel", ios::in | ios::binary);
 if (!msg.ParseFromIstream(&input))
 {
   cerr << "Failed to parse address book." << endl;
   return -1;
 }
 printf("length = %d\n", length);
 printf("Repeated Size = %d\n", msg.layer_size());

 ::google::protobuf::RepeatedPtrField< LayerParameter >* layer = msg.mutable_l
ayer();
 ::google::protobuf::RepeatedPtrField< LayerParameter >::iterator it = layer->
begin();
 for (; it != layer->end(); ++it)
 {
   cout << it->name() << endl;
   cout << it->type() << endl;
   cout << it->convolution_param().weight_filler().max() << endl;
 }

 return 0;
}
```

View Code

参考网址：http://www.cnblogs.com/stephen-liu74/archive/2013/01/04/2842533.html (http://www.cnblogs.com/stephen-liu74/archive/2013/01/04/2842533.html)

相关阅读：

- 用c#开发微信 (22) 微信商城
- Python与硬件学习笔记：蓝牙(二)
- Oracle回收站操作
- Quartz任务调度器
- Windows服务器MySQL中文乱码的解决方法
- hdu 3605 Escape 二分图的多重匹配（匈牙利算

文章评论

提交评论