Google Protocol Buffer

# 概述

**Google Protocol Buffer**（简称Protobuf）是google公司内部的混合语言数据标准。protobuf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，或者说序列化。它很适合做数据存储或 RPC（Remote Procedure Call Protocol远程过程调用）数据交换格式。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。proto2提供了 C++、Java、Python 三种语言的 API，而proto3增加了对Go、Ruby、JavaNano 的支持。

与XML相比，protobuf有如下优点：

·更简单

·小3~10倍

·快20~100倍

·更少的歧义

·可以方便的生成数据存取类

例如，让我们看看如何在XML中建模Person的name和email字段:

<person>  
    <name>John Doe</name>  
    <email>jdoe@example.com</email>  
</person>

对应的protobuf报文则如下:

#protobuf的文本表示  
#这不是正常时使用的二进制数据，这里只是为了说明存储情况

person {

name: "John Doe"

email: jdoe@example.com

}

# 下载与安装

## 下载

在网站 <http://code.google.com/p/protobuf/downloads/list>上可以下载 Protobuf 的源代码（需翻墙）。

github上同样有它的源码，使用git clone <https://github.com/google/protobuf.git>即可下载最新版本的protobuf。

## 安装

运行：

1. ./autogen.sh
2. ./configure --prefix=/usr （这样可以不用手动添加环境变量）
3. make
4. make check
5. make install

具体的安装过程可以参考README.md

# .proto文件说明

## 消息（message）

消息（message）由至少一个字段组合而成，类似于C语言中的结构。每个字段都有一定的格式。proto2和proto3的限定修饰符不一致，下面介绍的是proto2的限定修饰符。

字段格式：限定修饰符① | 数据类型② | 字段名称③ | = | 字段编码值④ | [字段默认值⑤]

### 限定修饰符required\optional\repeated

required: 表示是一个必须字段，“必须”对应于发送方，在发送消息之前必须设置该字段的值；对于接收方，必须能够识别该字段的意思。发送之前没有设置required字段或者无法识别required字段都会引发编解码异常，导致消息被丢弃。在proto3中不再支持。proto2中也不建议使用。

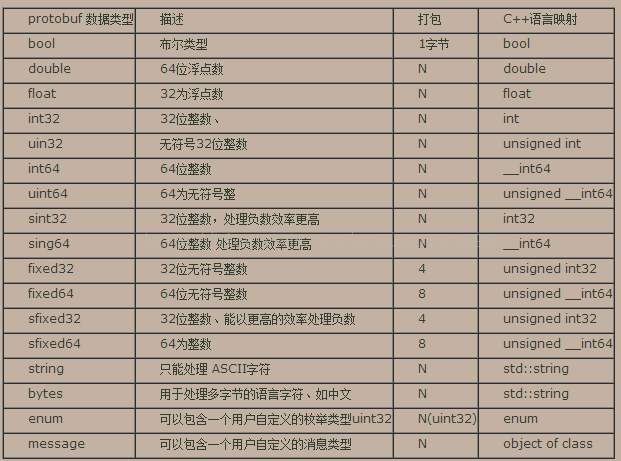
     optional：表示是一个可选字段，“可选”对于发送方，在发送消息时，可以有选择性的设置或者不设置该字段的值。对于接收方，如果能够识别可选字段就进行相应的处理，如果无法识别，则忽略该字段，消息中的其它字段正常处理。因为optional字段的特性，很多接口在升级版本中都把后来添加的字段都统一的设置为optional字段，这样老的版本无需升级程序也可以正常的与新的软件进行通信，只不过新的字段无法识别而已，因为并不是每个节点都需要新的功能，因此可以做到按需升级和平滑过渡。在proto3中不再支持，与之对应的是singular。

      repeated：表示该字段可以包含0~N个元素。其特性和optional一样，但是每一次可以包含多个值。可以看作是在传递一个数组的值。

建议：项目投入运营以后涉及到版本升级时的新增消息字段全部使用optional或者repeated，尽量不使用required。

### 数据类型

Protobuf定义了一套基本数据类型。几乎都可以映射到C++\Java等语言的基础数据类型.



N 表示打包的字节并不是固定。而是根据数据的大小或者长度。

例如int32，如果数值比较小，在0~127时，使用一个字节打包。

关于枚举的打包方式和uint32相同。

关于message，类似于C语言中的结构包含另外一个结构作为数据成员一样。

关于 fixed32 和int32的区别。fixed32的打包效率比int32的效率高，但是使用的空间一般比int32多。因此一个属于时间效率高，一个属于空间效率高。根据项目的实际情况，一般选择fixed32，如果遇到对传输数据量要求比较苛刻的环境，可以选择int32.

### 字段名称

字段名称的命名与C、C++、Java等语言的变量命名方式几乎是相同的。

protobuf建议字段的命名采用以下划线分割的驼峰式。例如 first\_name 而不是firstName.

### 字段编码值

有了该值，通信双方才能互相识别对方的字段。当然相同的编码值，其限定修饰符和数据类型必须相同。编码值的取值范围为 1~2^32（4294967296）。其中 1~15的编码时间和空间效率都是最高的，编码值越大，其编码的时间和空间效率就越低（相对于1-15），当然一般情况下相邻的2个值编码效率的是相同的，除非2个值恰好实在4字节，12字节，20字节等的临界区。比如15和16.1900~2000编码值为Google protobuf 系统内部保留值，建议不要在自己的项目中使用。

protobuf 还建议把经常要传递的值把其字段编码设置为1-15之间的值。消息中的字段的编码值无需连续，只要是合法的，并且不能在同一个消息中有字段包含相同的编码值。

### 默认值

当在传递数据时，对于required数据类型，如果用户没有设置值，则使用默认值传递到对端。当接受数据时，对于optional字段，如果没有接收到optional字段，则设置为默认值。

## 注意事项

### 关于import

protobuf 接口文件可以像C语言的h文件一个，分离为多个，在需要的时候通过 import导入需要对文件。其行为和C语言的#include或者java的import的行为大致相同。

### 关于package

避免名称冲突，可以给每个文件指定一个package名称，对于java解析为java中的包。对于C++则解析为名称空间。

### 关于message

支持嵌套消息，消息可以包含另一个消息作为其字段。也可以在消息内定义一个新的消息。

### 关于enum

枚举的定义和C++相同，但是有一些限制。

枚举值必须大于等于0的整数，**proto3中规定第一个枚举变量必须为0**。

使用分号(;)分隔枚举变量而不是C++语言中的逗号(,)

eg.

enum VoipProtocol

{

    H323 = 1;

    SIP  = 2;

    MGCP = 3;

    H248 = 4;

}

使用了required，需要全网统一升级，如果使用optional或者repeated可以平滑升级。

## 生成API

protoc --cpp\_out=. --java\_out=. --python\_out=. addressbook.proto

生成不同类型的API。

## caffe.pb.cc

大家应该可以知道其实caffe.pb.cc里面的东西都是从caffe.proto编译而来的，无非就是一些关于这些数据结构（类）的标准化操作，比如

void CopyFrom();

void MergeFrom();

void CopyFrom();

void MergeFrom;

void Clear();

bool IsInitialized() const;

int ByteSize() const;

bool MergePartialFromCodedStream();

void SerializeWithCachedSizes() const;

SerializeWithCachedSizesToArray() const;

int GetCachedSize()

void SharedCtor();

void SharedDtor();

void SetCachedSize() const;

# 实例分析

（example，删除部分注释、只讲解cpp相关代码）

使用“//”进行注释

## 编辑结构体描述文件.proto

syntax = "proto3";//指定使用的版本是proto3，若不指定，默认使用proto2

package tutorial;//包的名字为tutorial，对应于cpp的namespace

message Person {

string name = 1;

int32 id = 2; // Unique ID number for this person.

string email = 3;

enum PhoneType {

MOBILE = 0;

HOME = 1;

WORK = 2;

}

message PhoneNumber {

string number = 1;

PhoneType type = 2;

}

repeated PhoneNumber phones = 4;

}

// Our address book file is just one of these.

message AddressBook {

repeated Person people = 1;

}

使用命令：protoc --cpp\_out=. addressbook.proto

产生addressbook.pb.cc、addressbook.pb.h

生成的文件中有如下方法：

// name

  inline bool has\_name() const;

  inline void clear\_name();

  inline const ::std::string& name() const;

  inline void set\_name(const ::std::string& value);

  inline void set\_name(const char\* value);

  inline ::std::string\* mutable\_name();

  // id

  inline bool has\_id() const;

  inline void clear\_id();

  inline int32\_t id() const;

  inline void set\_id(int32\_t value);

  // email

  inline bool has\_email() const;

  inline void clear\_email();

  inline const ::std::string& email() const;

  inline void set\_email(const ::std::string& value);

  inline void set\_email(const char\* value);

  inline ::std::string\* mutable\_email();

  // phone

  inline int phone\_size() const;

  inline void clear\_phone();

  inline const ::google::protobuf::[RepeatedPtrField](https://developers.google.com/protocol-buffers/docs/reference/cpp/google.protobuf.repeated_field.html?hl=zh-CN#RepeatedPtrField)< ::tutorial::Person\_PhoneNumber >& phone() const;

  inline ::google::protobuf::[RepeatedPtrField](https://developers.google.com/protocol-buffers/docs/reference/cpp/google.protobuf.repeated_field.html?hl=zh-CN#RepeatedPtrField)< ::tutorial::Person\_PhoneNumber >\* mutable\_phone();

  inline const ::tutorial::Person\_PhoneNumber& phone(int index) const;

  inline ::tutorial::Person\_PhoneNumber\* mutable\_phone(int index);

  inline ::tutorial::Person\_PhoneNumber\* add\_phone();

枚举与嵌套类

 生成的代码包含一个PhoneType枚举。Person::PhoneType, Person:MOBILE, Person::HOME, Person:WORK.

编译器生成的嵌套类称为Person::PhoneNumber. 实际生成类为Person\_PhoneNumber.

**标准方法**

bool IsInitialized() const:               确认required字段是否被设置

string DebugString() const:                 返回消息的可读表示，用于调试

void CopyFrom(const Person& from):      使用给定消息值copy

void Clear():                               清除所有元素为空状态

**解析与序列化**

bool SerializeToString(string\* output) const:        序列化消息，将存储字节的以string方式输出。注意字节是二进制，而非文本；

bool ParseFromString(const string& data):           解析给定的string

bool SerializeToOstream(ostream\* output) const:  写消息给定的c++  ostream中

bool ParseFromIstream(istream\* input):             从给定的c++ istream中解析出消息

## add\_person.cc

#include <iostream>

#include <fstream>

#include <string>

#include "addressbook.pb.h"

using namespace std;

// This function fills in a Person message based on user input.

void PromptForAddress(tutorial::Person\* person) {

cout << "Enter person ID number: ";

int id;

cin >> id;

person->set\_id(id);

cin.ignore(256, '\n');

cout << "Enter name: ";

getline(cin, \*person->mutable\_name());

cout << "Enter email address (blank for none): ";

string email;

getline(cin, email);

if (!email.empty()) {

person->set\_email(email);

}

while (true) {

cout << "Enter a phone number (or leave blank to finish): ";

string number;

getline(cin, number);

if (number.empty()) {

break;

}

tutorial::Person::PhoneNumber\* phone\_number = person->add\_phones();

phone\_number->set\_number(number);

cout << "Is this a mobile, home, or work phone? ";

string type;

getline(cin, type);

if (type == "mobile") {

phone\_number->set\_type(tutorial::Person::MOBILE);

} else if (type == "home") {

phone\_number->set\_type(tutorial::Person::HOME);

} else if (type == "work") {

phone\_number->set\_type(tutorial::Person::WORK);

} else {

cout << "Unknown phone type. Using default." << endl;

}

}

}

// Main function: Reads the entire address book from a file,

// adds one person based on user input, then writes it back out to the same

// file.

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

// Verify that the version of the library that we linked against is

// compatible with the version of the headers we compiled against.

GOOGLE\_PROTOBUF\_VERIFY\_VERSION;

if (argc != 2) {

cerr << "Usage: " << argv[0] << " ADDRESS\_BOOK\_FILE" << endl;

return -1;

}

tutorial::AddressBook address\_book;

{

// Read the existing address book.

fstream input(argv[1], ios::in | ios::binary);

if (!input) {

cout << argv[1] << ": File not found. Creating a new file." << endl;

} else if (!address\_book.ParseFromIstream(&input)) {

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

return -1;

}

}

// Add an address.

PromptForAddress(address\_book.add\_people());

{

// Write the new address book back to disk.

fstream output(argv[1], ios::out | ios::trunc | ios::binary);

if (!address\_book.SerializeToOstream(&output)) {

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

return -1;

}

}

// Optional: Delete all global objects allocated by libprotobuf.

google::protobuf::ShutdownProtobufLibrary();

return 0;

}

## list\_person.cc

// See README.txt for information and build instructions.

#include <iostream>

#include <fstream>

#include <string>

#include "addressbook.pb.h"

using namespace std;

// Iterates though all people in the AddressBook and prints info about them.

void ListPeople(const tutorial::AddressBook& address\_book) {

for (int i = 0; i < address\_book.people\_size(); i++) {

const tutorial::Person& person = address\_book.people(i);

cout << "Person ID: " << person.id() << endl;

cout << " Name: " << person.name() << endl;

if (person.email() != "") {

cout << " E-mail address: " << person.email() << endl;

}

for (int j = 0; j < person.phones\_size(); j++) {

const tutorial::Person::PhoneNumber& phone\_number = person.phones(j);

switch (phone\_number.type()) {

case tutorial::Person::MOBILE:

cout << " Mobile phone #: ";

break;

case tutorial::Person::HOME:

cout << " Home phone #: ";

break;

case tutorial::Person::WORK:

cout << " Work phone #: ";

break;

}

cout << phone\_number.number() << endl;

}

}

}

// Main function: Reads the entire address book from a file and prints all

// the information inside.

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

// Verify that the version of the library that we linked against is

// compatible with the version of the headers we compiled against.

GOOGLE\_PROTOBUF\_VERIFY\_VERSION;

if (argc != 2) {

cerr << "Usage: " << argv[0] << " ADDRESS\_BOOK\_FILE" << endl;

return -1;

}

tutorial::AddressBook address\_book;

{

// Read the existing address book.

fstream input(argv[1], ios::in | ios::binary);

if (!address\_book.ParseFromIstream(&input)) {

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

return -1;

}

}

ListPeople(address\_book);

// Optional: Delete all global objects allocated by libprotobuf.

google::protobuf::ShutdownProtobufLibrary();

return 0;

}

# proto2和proto3的区别

总的来说，proto3 比 proto2 支持更多语言但 更简洁。去掉了一些复杂的语法和特性，更强调约定而弱化语法。如果是首次使用 Protobuf ，建议使用 proto3 。

1. 在第一行非空白非注释行，必须写：

syntax = "proto3";

若不加此语句，默认使用proto2。

1. 字段规则移除了 “required”，并把 “optional” 改名为 “singular”；

在 proto2 中 required 也是不推荐使用的。proto3 直接从语法层面上移除了 required 规则。

1. 语言增加 Go、Ruby、JavaNano 支持；
2. 移除了 default 选项；

在 proto2 中，可以使用 default 选项为某一字段指定默认值。在 proto3 中，字段的默认值只能根据字段类型由系统决定。也就是说，默认值全部是约定好的，而不再提供指定默认值的语法。

在字段被设置为默认值的时候，该字段不会被序列化。这样可以节省空间，提高效率。

但这样就无法区分某字段是根本没赋值，还是赋值了默认值。这在 proto3 中问题不大，但在 proto2 中会有问题。

比如，在更新协议的时候使用 default 选项为某个字段指定了一个与原来不同的默认值，旧代码获取到的该字段的值会与新代码不一样。

1. 另一个重约定而弱语法的例子是 Go 语言里的公共/私有对象。Go 语言约定，首字母大写的为公共对象，否则为私有对象。所以在 Go 语言中是没有 public、private 这样的语法的。
2. 枚举类型的第一个字段必须为 0 ；

这也是一个约定。

1. 移除了对分组的支持；

分组的功能完全可以用消息嵌套的方式来实现，并且更清晰。在 proto2 中已经把分组语法标注为『过期』了。这次也算清理垃圾了。

1. 旧代码在解析新增字段时，会把不认识的字段丢弃，再序列化后新增的字段就没了；

在 proto2 中，旧代码虽然会忽视不认识的新增字段，但并不会将其丢弃，再序列化的时候那些字段会被原样保留。

1. 移除了对扩展的支持，新增了 Any 类型；

Any 类型是用来替代 proto2 中的扩展的。目前还在开发中。

proto2 中的扩展特性很像 Swift 语言中的扩展。理解起来有点困难，使用起来更是会带来不少混乱。

相比之下，proto3 中新增的 Any 类型有点想 C/C++ 中的 void\* ，好理解，使用起来逻辑也更清晰。

1. 增加了 JSON 映射特性；