ProtoBuf简介

# 环境

## 版本

Protobuf 2.5.0

Visual Studio 2008

## 下载：

<https://github.com/google/protobuf>

默认为最新版本，历史版本可在tags里面找到。选择2.5.0后，点download zip下载源代码。

<https://github.com/google/protobuf/tree/v2.5.0>

## 编译

### VS2008

使用VS2008打开protobuf-2.5.0\vsprojects\protobuf.sln

如果提示gtest和gtest\_main项目不存在，并且编译时test\_plugin和tests项目会因为丢失文件无法编译，忽略即可。

libprotobuf/libprotobuf-lite是protobuf的lib库，默认用于静态链接。

libprotoc是protoc的lib库，默认为静态链接库，可用于自定义的.proto文件解析，如编写其他语言的.proto编译器。

Protoc是.proto文件编译工具，用于根据pb定义文件(.proto)生成相应语言的源代码（官方支持c++, java, python, 可在社区找到其他语言支持）。

默认情况下，生成的库时静态链接的。由于proto buf更新及动态链接时可能导致的不一致，所以建议使用静态链接。如需更改为动态链接，可以参考protobuf-2.5.0\vsprojects\readme.txt中DLLs vs. static linking部分。

### VS2013

类似VS2008，可以从vs2008项目迁移而来。

#### 编译错误：找不到符号min/max

WIN7 SP1 X64，VS2013 w/ update 4下编译会提示错误：

zero\_copy\_stream\_impl\_lite.cc找不到符号min/max。

解决方法：

在zero\_copy\_stream\_impl\_lite.cc文件开头部分加入如下代码即可：

#ifndef max

#define max(a,b) (((a) > (b)) ? (a) : (b))

#endif

#ifndef min

#define min(a,b) (((a) < (b)) ? (a) : (b))

#endif

#### warning MSB8012: TargetPath not match the OutputFile

原因：VS2008升级到VS2013时，使用MSBuild替换自定义的VCBuild系统，此时会添加两个新属性并采用默认值：

目标文件名（TargetName）默认值为项目名)

目标文件扩展名（TargetExt）通用属性-常规

如果VS2008修改了生成文件的默认输出，则可能导致二者不一致。该问题可能导致调试时使用错误的名字启动调试。

解决方法：

修改 输出文件名(OutputFile) 或 目标文件名(TargetName) 使二者一致。

#### warning LNK4075: 忽略“/EDITANDCONTINUE”(由于“/SAFESEH”规范)

原因：如果某些特性和EditAndContinue特性不兼容，则会禁用EditAndContinue。升级到VS2013时新增特性，默认为SAFESEH，生成映像的安全异常处理程序，和EditAndContinue不兼容。

解决：建议debug版关闭SAFESEH并开启EditAndContinue，release版则反之。

属性 - 配置属性 – 连接器 – 高级 – 影响具有安全异常处理程序。

### VS2008和VS2013共存

有时为了方便，可能同时需要编译08或13的库，由于二者项目文件不同扩展名，因此只要将解决方案(.sln)文件使用不同名字即可。

1. protobuf.sln改名为protobuf-08.sln，并使用vs2008打开，设置好想关参数，确认编译通过后关闭。
2. protobuf-08.sln复制为protobuf-13.sln，并使用vs2013打开，迁移项目。完成后，选择重新编译解决方案。一般不会有错误，可能会由于项目迁移导致一些选项更改产生wanning，修正即可。
3. vs2008和vs2013的项目文件现在可以共存了，需要相应库时只要打开相应sln文件并重新编译项目即可

### MT/MTd/MD/MDd

默认情况下，protobuf采用MD(Release)或MDd(Debug)，如果使用protobuf的项目采用了其他的线程库，则需要修改为和项目相同，否则会提示连接错误。

## 参考资料

项目首页：

<https://developers.google.com/protocol-buffers/>

# protoc

用于解析PROTO文件并根据参数生成相应输出

## 参数及说明

**-IPATH, --proto\_path=PATH**

Specify the directory in which to search for imports. May be specified multiple times; directories will be searched in order. If not given, the current working directory is used.

指定导入消息路径。可以多次指定，按顺序搜索。未指定默认为当前路径。

--**version**

Show version info and exit.

显示版本号。

**-h, --help**

Show this text and exit.

显示帮助

**--encode=MESSAGE\_TYPE**

Read a text-format message of the given type from standard input and write it in binary to standard output. The message type must be defined in PROTO\_FILES or their imports.

编码消息。从标准输入读取指定类型的文本格式消息并且将二进制形式写入到标准输出。消息类型必须在PROTO文件火导入消息中定义。

**--decode=MESSAGE\_TYPE**

Read a binary message of the given type from standard input and write it in text format to standard output. The message type must be defined in PROTO\_FILES or their imports.

解码消息。对指定类型的消息，标准输入二进制->标准输出文本形式，消息类型必须在PROTO文件火导入消息中定义。

**--decode\_raw**

Read an arbitrary protocol message from standard input and write the raw tag/value pairs in text format to standard output. No PROTO\_FILES should be given when using this flag.

直接数据解码。从标准输入读入任意消息，以标记/值对以文本形式直接写入到标准输出。不需要PROTO文件

**-oFILE, --descriptor\_set\_out=FILE**

Writes a FileDescriptorSet (a protocol buffer, defined in descriptor.proto) containing all of the input files to FILE.

**--include\_imports**

When using --descriptor\_set\_out, also include all dependencies of the input files in the set, so that the set is self-contained.

**--include\_source\_info**

When using --descriptor\_set\_out, do not strip SourceCodeInfo from the FileDescriptorProto. This results in vastly larger descriptors that include information about the original location of each decl in the source file as well as surrounding comments.

**--error\_format=FORMAT**

Set the format in which to print errors. FORMAT may be 'gcc' (the default) or 'msvs' (Microsoft Visual Studio format).

**--plugin=EXECUTABLE**

Specifies a plugin executable to use. Normally, protoc searches the PATH for plugins, but you may specify additional executables not in the path using this flag. Additionally, EXECUTABLE may be of the form NAME=PATH, in which case the given plugin name is mapped to the given executable even if the executable's own name differs.

**--cpp\_out=OUT\_DIR**

Generate C++ header and source.

生成C++头和源码文件。

**--java\_out=OUT\_DIR**

Generate Java source file.

生成JAVA源码文件。

**--python\_out=OUT\_DIR**

Generate Python source file.

生成Python源码文件。

## 示例

将message.proto编译为c++代码，并输出到当前目录下

protoc message.proto –cpp\_out=.

# PROTO文件

## PROTO示例

package tutorial;

message Person {

required string name = 1;

required int32 id = 2;

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;

}

message AddressBook {

repeated Person person = 1;

}

## 关键字

package: 包声明，用于解决名称冲突。相当于C++中的namespace。可选。

message: 消息定义，是一组字段（成员）的集合。可以使用简单类型及消息。

字段序号：字段后面的”=1”, “=2”是二进制编码中元素的唯一tag。tag序号1-15比更高的序号少使用一个字节进行编码，将其用于常用元素可以优化代码。Repeated字段的每一个重复元素都需要tag序号重新编码，因此该优化对常用repeated字段更加有效。

成员限定符：

required：必须提供该字段的值，否则该消息为未初始化。debug模式下，序列化未初始化的消息会导致libprotobuf产生断言错误。优化的编译下，该检查会取消，消息序可强制序列化。但是，解析为序列化的消息时会失败(parse函数返回false)。除此之外，required字段和optional字段行为完全一致。

optional: 该字段值得设置为可选的。未设置时采用默认值。对于简单类型，可直接提供默认值；否则，系统会自动设置默认值：数值类型设为0，string类型为空字符串，bool类型为false。对于内嵌的消息，默认值为消息的“默认实例”或”原型“，即**所有字段都未设置**。获取未显示设置的optional或required字段的值将会返回该字段的默认值。

repeated: 该字段可以重复任意次数（包含0），其次序将被保留。可以将其看做动态长度的数组。

注意：required是永久性的。required限定的使用需要非常小心。如果某一时刻需要停止写入或发送required字段时，将其变为optional将会产生问题：旧版本的消息解析器会认为该消息为初始化并拒绝或丢弃该消息。可以考虑针对每一个程序使用定制的消息验证程序。Google的一些工程师已经得出结论，使用required弊大于利，他们更倾向于仅使用optional和repeated限定符。但这种看法并不一定通用。

## 字段类型

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| .proto  Type | Notes | C++  Type | Java  Type | Python  Type[2] |
| double |  | double | double | float |
| float |  | float | float | float |
| int32 | Uses variable-length encoding. Inefficient for encoding negative numbers – if your field is likely to have negative values, use sint32 instead. | int32 | int | int |
| int64 | Uses variable-length encoding. Inefficient for encoding negative numbers – if your field is likely to have negative values, use sint64 instead. | int64 | long | int/long[3] |
| uint32 | Uses variable-length encoding. | uint32 | int[1] | int/long[3] |
| uint64 | Uses variable-length encoding. | uint64 | long[1] | int/long[3] |
| sint32 | Uses variable-length encoding. Signed int value. These more efficiently encode negative numbers than regular int32s. | int32 | int | int |
| sint64 | Uses variable-length encoding. Signed int value. These more efficiently encode negative numbers than regular int64s. | int64 | long | int/long[3] |
| fixed32 | Always four bytes. More efficient than uint32 if values are often greater than 228. | uint32 | int[1] | int |
| fixed64 | Always eight bytes. More efficient than uint64 if values are often greater than 256. | uint64 | long[1] | int/long[3] |
| sfixed32 | Always four bytes. | int32 | int | int |
| sfixed64 | Always eight bytes. | int64 | long | int/long[3] |
| bool |  | bool | boolean | bool |
| string | A string must always contain UTF-8 encoded or 7-bit ASCII text. | string | String | str/unicode[4] |
| bytes | May contain any arbitrary sequence of bytes. | string | ByteString | str |

[1] In Java, unsigned 32-bit and 64-bit integers are represented using their signed counterparts, with the top bit simply being stored in the sign bit.

[2] In all cases, setting values to a field will perform type checking to make sure it is valid.

[3] 64-bit or unsigned 32-bit integers are always represented as long when decoded, but can be an int if an int is given when setting the field. In all cases, the value must fit in the type represented when set. See [2].

[4] Python strings are represented as unicode on decode but can be str if an ASCII string is given (this is subject to change).

## protobuf API

将示例文件编译为cpp文件：

protoc -I=$SRC\_DIR --cpp\_out=$DST\_DIR $SRC\_DIR/addressbook.proto

可以得到.h和.cc源码，含有消息中的相应类的定义及实现。

字段存取函数包括：

set\_ 设置值

set\_allocated\_ 设置值并获取入参指针所有权（用于类成员）。

has\_ 测试是否赋值(required或optional字段)

clear\_ 清除赋值

mutable\_ 获取对象或字符串指针（message或string类型）

对于repeated字段，会有以下函数：

\_size: 字段元素数量（类似数组长度）

add\_: 添加一个元素

index函数：根据索引获取元素的常量引用（取值）或指针（修改）

以Person类为例，编译器为每一个字段生成了相应的存取函数：

// name

inline bool has\_name() const; // has\_是否已赋值

inline void clear\_name(); // clear\_清除值

inline const ::std::string& name() const; // &常量引用，取值

inline void set\_name(const ::std::string& value); // set\_设置值

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

inline ::std::string\* mutable\_name(); // 字符串指针(对string类型)

inline void set\_allocated\_name(::std::string\* name); // 使用指针name设置值，name指针对象由Person类维护并释放。因此入参name指针的内存必须放在堆上，并且不能在外部释放。

// 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; // \_size获取元素个数

inline void clear\_phone();

inline const ::google::protobuf::RepeatedPtrField< ::tutorial::Person\_PhoneNumber >& phone() const;

inline ::google::protobuf::RepeatedPtrField< ::tutorial::Person\_PhoneNumber >\* mutable\_phone(); // 首元素指针

inline const ::tutorial::Person\_PhoneNumber& phone(int index) const; // 常量索引引用取值

inline ::tutorial::Person\_PhoneNumber\* mutable\_phone(int index); // 索引指针，用于赋值

inline ::tutorial::Person\_PhoneNumber\* add\_phone(); // 添加元素

## 枚举量

消息内定义的枚举量为类内的类。使用时需要加上类域。如Person::MOBILE

## 标准消息方法

bool IsInitialized() const

checks if all the required fields have been set.

检查是否所有required字段都已赋值

string DebugString() const

returns a human-readable representation of the message, particularly useful for debugging.

返回可阅读的消息表示，用于调试。

void CopyFrom(const Person& from)

overwrites the message with the given message's values.

用指定消息的值覆盖该消息

void Clear()

clears all the elements back to the empty state.

清空所有元素

## 解析和序列化

ByteSize

消息序列化后的字节数。

SerializeToOstream

SerializeToString

SerializeToArray

将消息序列化到ostream, string或内存。写入错误时返回false，也就是说如果required字段未设置仍然返回true，但debug下会抛出异常。所以，序列化前最好检查IsInitialized。

SerializePartialToXXX

将消息序列化，即使部分required字段未设置。

ParseFromXXX

解析序列化的消息。如果required字段未设置，返回false，并输出错误信息。

ParsePartialFromXXX

解析序列化的消息，即使部分required字段未设置。

## 释放库内存

google::protobuf::ShutdownProtobufLibrary();

结束后调用，释放库所使用的全局内存。对于程序可不释放，因为操作系统会自动回收。但是如果是可能被重复加载和卸载的lib库，则必须释放，否则可能导致内存泄露。

# 项目示例

## 环境

VS2008 professional sp1

Protobuf 2.5.0

## 编译protobuf

参考1.3编译protobuf项目：libprotobuf, libprotobuf-lite, libprotoc, protoc.

将相关文件收集并存放到新建的目录下（如D:\dev\Protobuf2.5），用于protobuf消息文件的编译及链接，包括：

lib库文件（protobuf-2.5.0\vsprojects\Release下）

libprotobuf-lite.lib

libprotobuf.lib

libprotoc.lib

protobuf源代码（在protobuf-2.5.0\src下，需复制整个google文件夹）

## 编译proto

文本编辑器生成person.proto文件，内容为：

message Person {

required string name = 1;

required int32 id = 2;

optional string email = 3;

repeated bytes binarray = 4;

}

命令行使用protoc编译该文件：

protoc Person.proto --cpp\_out=.

得到编译后的文件：

Person.pb.cc

Person.pb.h

## 新建pb测试项目

VS2008新建win32控制台程序，pbtest，空项目。

更改项目属性，VC++目录，包含目录，添加 D:\dev\Protobuf2.5

添加Person.pb.cc, Person.pb.h到项目。

main.cpp:

#include <iostream>

using namespace std;

#ifdef \_DEBUG

#pragma comment(lib, "libprotobufD.lib")

#else

#pragma comment(lib, "libprotobuf.lib")

#endif // \_DEBUG

#include "Person.pb.h"

int main(void)

{

char bin1[5] = {0, 1, 2, 3, 4};

char bin2[5] = {-10, -11, -12, -13, -14};

char bin3[5] = {5, 6, 7, 8, 9};

string\* ps = NULL;

const char\* pc = NULL;

bool bret = false;

Person person;

person.set\_name("john");

person.set\_id(123);

person.add\_binarray(bin1, 5);

// mutable: to modify member

ps = person.mutable\_binarray(0);

pc = ps->c\_str();

person.set\_binarray(0, bin3, 5);

pc = ps->c\_str();

\*ps = "cover this";

pc = ps->c\_str();

person.add\_binarray(bin2, 5);

pc = person.mutable\_binarray(1)->c\_str();

const string debugstring = person.DebugString();

cout << debugstring << endl;

int nlen = person.ByteSize();

char\* pbuff = new char[nlen];

bret = person.IsInitialized();

person.SerializeToArray(pbuff, nlen);

Person person2;

person2.ParseFromArray(pbuff, nlen);

cout << person2.DebugString() << endl;

// unInitialized test

Person person0;

person0.set\_name("part");

cout << person0.DebugString() << endl;

int nlen2 = person0.ByteSize();

char\* pbuff2 = new char[nlen2];

bret = person0.IsInitialized(); // false

// bret = person0.SerializeToArray(pbuff2, nlen2); // true. exception in debug.

bret = person0.SerializePartialToArray(pbuff2, nlen2); // true

Person person3;

bret = person3.ParseFromArray(pbuff2, nlen2); // false, error message & cannot DebugString()

bret = person3.has\_name(); // true

bret = person3.has\_id(); // false

Person person4;

bret = person4.ParsePartialFromArray(pbuff2, nlen2); // true

cout << person4.DebugString() << endl; // ok

bret = person4.has\_id(); // false

person4.set\_id(4);

bret = person4.has\_name(); // true

bret = person4.has\_id(); // true

cout << person4.DebugString() << endl;

return 0;

}