**[在Python中使用protocol buffers参考指南](http://blog.csdn.net/losophy/article/details/17006573)**

# Protocol Buffer Basics: Python

本教程提供了一个Python程序员使用protocol buffers的基本的入门教程。通过创建一个简单的示例应用程序,它向您展示了如何

\*在一个.proto文件中定义Message的格式。

\*使用protocol buffer compiler。

\*使用Python protocol buffer API去读写Message。

这不是一个在Python中使用protocol buffers的一个全面的指南。如果想了解更详细的参考信息，请阅读 [Protocol Buffer Language Guide](https://developers.google.com/protocol-buffers/docs/proto)，[Python API Reference](https://developers.google.com/protocol-buffers/docs/reference/python/index.html" \t "_blank)，[Python Generated Code Guide](https://developers.google.com/protocol-buffers/docs/reference/python-generated" \t "_blank)和[Encoding Reference](https://developers.google.com/protocol-buffers/docs/encoding" \t "_blank)。

## Why Use Protocol Buffers?

我们将使用的示例是一个非常简单的“地址簿”应用程序，可以从一个文件中读写人们的联系方式。地址簿中的每个人都有一个名字,一个ID、一个电子邮件地址,和联系电话号码。

你怎样用这样方式序列化和检索结构数据？这里有一些办法可以解决这个问题：

\*使用Python处理。这是默认的方法,因为这种方法是直接用到语言，但它不利于模式演变，还有，它不利于你共享数据给c++或Java写的应用。

\*你可以发明一种特别的方式将数据项编码为一个字符串，如将4个int编码为“12:3:23:67”。这是一个简单的和灵活的方法，尽管它一次性需要编写编码和解析的代码，并为解析加上一个小小的运行成本。这方法最适合为非常简单的数据编码。

\*用XML序列化数据。这种方法非常有吸引力，因为XML具有易读性，还有了许多库，用来支持各种语言。这是一个好选择，如果你想和其它应用/工程共享数据。但是，XML也是出了名的耗空间，还有，编码/解码会令应用程序性能产生巨大的损失。加上，操纵一个XML DOM树通常会比操纵类中的字段复杂。

Protocol buffers会灵活、高效、自动化解答来准确地解决这个问题。有了protocol buffers，你就可以编写一个.proto文件用来描述你想存储的数据结构。因此，protocol buffer编译器会创建一个类，实现自动编码和解析protocol buffer数据，通过一个高效的二进制格式。这个生成的类提供了getter和setter的字段组成一条protocol buffer，而且把读出和写入的细节当成protocol buffer的一个单元。更重要的，protocol buffer支持在日后里扩展格式这种想法，这样，代码仍然可以读取用旧的格式编码的数据。

## Where to Find the Example Code

在源代码目录中，文件夹“examples”下包含所有的例程。[Download it here.](http://code.google.com/p/protobuf/downloads/" \t "_blank)

## Defining Your Protocol Format

为了创建你的“地址簿”应用，你会用到一个.proto文件。这是一个很简单的.proto文件定义：你可以为你想序列化的数据结构添加一条Message，然后在Message中为每个字段指定一个名称和一个类型。以下是你想为你的Message定义的.proto文件，addressbook.proto。

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;

}

如你所见，在语法上很像C++和Java。那就让我们看看文件中的每个部分和看看它们究竟是干什么的。

这个.proto文件开头是包的声明，为了帮助防止在不同的工程中命名冲突。在Python中，包通常由目录结构决定的，所以这个由你的.proto文件定义的包，在你生成你代码中是没有效果的。但是，你应该坚持声明这条语句，为了在protocol Buffers的命名空间中防止名子的冲突，就像其它非Python的语言那样。

然后，就是你定义的Message。一个Message是一个包含一组类型字段的集合。有许多简单的标准的数据类型可以用在类型字段中，包括bool，int32，float，double和string。你也可以使用更加多的结构来定义你的Message，例如用其它Message类型当作类型字段-在上面的例子PersonMessage中就包含了PhoneNumberMessage，还有AddressBookMessage包含PersonMessage。你也可以定义Message嵌入其它的Message——就如你所见到的那样，PhoneNumber类型就是在Person类型中定义的。你也可以定义一个枚举类型，如果你想你其中一个字段有一个预设的类型列表——在这里，你可以将你的电话号码列举为MOBILE，HOME或者WORK。

那 个“＝1”，“＝2”标记每个元素的识别，作为二进制编码中字段的唯一的标签。标签要求数字1－15比更高的数字少一个字节编码，所以，作为最优化的方 案，你可以决定对常用的和要重复使用的元素使用这些标签，把16或最高的数字留给不常用和可选择的元素。每个重复的字段里的元素要求重新编码它的标签号 码，所以重复的字段特别适合使用这种优化。

每个字段一定要被以下的修饰语修饰：

\*required：一定要提供一个值给这个字段，否则这条Message会被认为“没有初始化”。序列化一列没有初始化的Message会出现异常。 解析一条没有初始化的Message会失败。除此而外，这个required字段的行为更类似于一个optional字段。

\*optional：这个字段可以设置也可以不设置 。如果一个可选字段没有设置值，会用缺省的值。简单来说，你可以指定自己的默认值，就像我们在例子中对phone number类型所做的。另外，系统会缺省这样做：0给整数类型，空串给字符串类型，false给布尔类型。对于嵌入的Message，缺省的值通常会是“默认实例”或“原型”，对那些没有设置字段的Message。调用存取器获得一个可选的(或要求)字段的值，那些通常什么明确给出值的字段总是返回该字段的默认值。

\*repeated：这个字段会重复几次一些号码（包括0）。重复的值给按顺序保存在protocol buffer中。重复的字段会被认为是动态的数组。

**Required Is Forever** 你应该非常小心地把字段标记为required。如果在某一时刻你希望停止写或发送一个必填字段,那就把不确定的字段更改为一个可选的字段——老的阅读器会认为没有这个字段Message是不完整的,而且可能会无意中拒绝或删除它们。你应该考虑为你的buffer编写特定于应用程序的自定义验证例程。一些来自Google有些结论：使用required弊大于利；他们更愿意只用optional和repeated。但是，这一观点并不普遍。

你会找到编写.proto文件的指南——包括所有可能的类型字段——在[Protocol Buffer Language Guide](https://developers.google.com/protocol-buffers/docs/proto" \t "_blank).不要去找类似于类继承的设备,虽然——protocol buffers不这样做。

## Compiling Your Protocol Buffers

现在你有了自己的.proto文件，下一件你需要去做的事就是生成你需要读写AddressBook（还带有Person和PhoneNumber）Message的类。为了完成这件工作，你需要运行protocol buffer 编译器protoc去编译你的.proto文件：

1.如果你没有安装编译器，[download the package](http://code.google.com/p/protobuf/downloads/" \t "_blank)，按照在README的说明去做。

2.现在运行编译器，指定源目录（你的应用程序源码目录——如果你不提供这个目录，默认就是当前目录），目标目录（你的应用程序编译后生成的代码的目录；通常用$SRC\_DIR），还有你.proto文件的目录路径。在这种情况下，你可以

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

因为你想生成Python的类，所以你要用--python\_out选项——也有类似的选项支持其它语言。

这样addressbook\_pb2.py就会生成在你指定的目标目录中。

## The Protocol Buffer API

不你你生成的Java或C++的protocol buffer代码，Python protocol buffer编译器不会直接生成你可以数据访问的代码。反而（就你看见的那样，如果你看了addressbook\_pd2.py）它会为你的Message，枚举，字段生成指定的描述符，还有一些难以理解的空类，其中一段Message类型：

class Person(message.Message):

\_\_metaclass\_\_ = reflection.GeneratedProtocolMessageType

class PhoneNumber(message.Message):

\_\_metaclass\_\_ = reflection.GeneratedProtocolMessageType

DESCRIPTOR = \_PERSON\_PHONENUMBER

DESCRIPTOR = \_PERSON

class AddressBook(message.Message):

\_\_metaclass\_\_ = reflection.GeneratedProtocolMessageType

DESCRIPTOR = \_ADDRESSBOOK

每个类中都有一些重要的语句：\_\_metaclass\_\_ = reflection.GeneratedProtocolMessageType.尽管Python中metaclasses是如何工作的详细信息超出了本教程的范围，你可以把它们看作是创建类的模板。在加载时，GeneratedProtocolMessageType  metaclass 会用指定的描述符创建所有你需要用到的Message类型的Python方法和添加和这些方法相关的类。然后你就可以在你的代码中使用这些类。

这一切的最终效果是，你可以使用Person类就像你定义的Message的基类，将它当作常规的字段。例如，你可以这样写：

import addressbook\_pb2

person = addressbook\_pb2.Person()

person.id = 1234

person.name = "John Doe"

person.email = "jdoe@example.com"

phone = person.phone.add()

phone.number = "555-4321"

phone.type = addressbook\_pb2.Person.HOME

注意,这些赋值不仅仅是一个在通用的Python对象中添加任意的新字段。如果你尝试赋值一些值到.proto文件没有定义的字段变量中，会发生AttributeError 异常。如果你赋给一个字段错误的类型的值，会发生TypeError 异常。当然，同时,在看一个字段的值前，这个字段就已经设置为返回默认值。

person.no\_such\_field = 1 # raises AttributeError

person.id = "1234" # raises TypeError

如果想知道更多关于protocol 编译器为特定定义字段生成的成员，请看[Python generated code reference](https://developers.google.com/protocol-buffers/docs/reference/python-generated" \t "_blank).

### Enums

Enums是由metaclass扩展成一组具有符号常量的整数值。那么，举个例子，不变量addressbook\_pb2.Person.WORK有一个值为2.

### Standard Message Methods

每个Message类都包含一些其它方法让你检查或操作 entire message，包括：

 IsInitialized(): 检查required字段是否都设置了值。

 \_\_str\_\_(): 返回一个可读的message,对于调试尤其有用。(通常调用str(message)或打印message。)

 CopyFrom(other\_msg): 用给出的message的值覆盖这个message

 Clear(): 清除所有元素，回到空状态。  
这些方法是操作Message的接口，想了解更多，可以看[complete API documentation for Message](https://developers.google.com/protocol-buffers/docs/reference/python/google.protobuf.message.Message-class" \t "_blank).

### Parsing and Serialization

最后，每个protocol buffer类都有读message或写message的方法给你选择通过用protocol buffer [binary format](https://developers.google.com/protocol-buffers/docs/encoding)。包括：

* SerializeToString(): 序列化这个message和以字符串的方式返回。 注意，这是二进行字节，不是一个文本； 我们只使用str类型作为一个方便的容器。
* ParseFromString(data): 从给出的字符串中解析一条message。

这里有只是一些使用解析和序列化的选项。此外，也会以看 [Message API reference](https://developers.google.com/protocol-buffers/docs/reference/python/google.protobuf.message.Message-class)，查看完整的列表。

**Protocol Buffers和O-O Design Protocol buffer 类基本上是dumb data holders（类似于C++的structs）；它们在对象模型虽不做好first class citizens。如果你想为已生成的类添加丰富的行为，更好的方法是把已生成的protocol buffer类封装在一个特定于应用程序的类。封装protocol buffers是一个好想法，如果你不会控制.proto file的设计（如果说，你征用其它工程的代码）。在这种情况下，您可以使用封装类去制作接口会更适合您的应用程序，在独特环境中：隐藏一些数据和方法， 暴露便利的函数，等等。你绝不应该添加行为通过继承已生成的类。这将打破内部机制和没有良好的面向对象的体验。**

## Writing A Message

现在可以尝试用你的protocol buffer 类了。首先你第一件事你想让你的地址簿应用程序能够做的事情就是把个人信息写到你的地址簿文件里。为此，你创建和填写你的protocol buffer类的实例,然后把它们写到输出流。  
以下是从AddressBook文件中读一个程序，按用户的输入添加一个新Person，和写一个新AddressBook再一次返回这个文件。部分直接调用或引用的从protocol compiler生成的代码给出了高亮显示。

#! /usr/bin/python

import addressbook\_pb2

import sys

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

def PromptForAddress(person):

person.id = int(raw\_input("Enter person ID number: "))

person.name = raw\_input("Enter name: ")

email = raw\_input("Enter email address (blank for none): ")

if email != "":

person.email = email

while True:

number = raw\_input("Enter a phone number (or leave blank to finish): ")

if number == "":

break

phone\_number = person.phone.add()

phone\_number.number = number

type = raw\_input("Is this a mobile, home, or work phone? ")

if type == "mobile":

phone\_number.type = addressbook\_pb2.Person.MOBILE

elif type == "home":

phone\_number.type = addressbook\_pb2.Person.HOME

elif type == "work":

phone\_number.type = addressbook\_pb2.Person.WORK

else:

print "Unknown phone type; leaving as default value."

# Main procedure: Reads the entire address book from a file,

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

# file.

if len(sys.argv) != 2:

print "Usage:", sys.argv[0], "ADDRESS\_BOOK\_FILE"

sys.exit(-1)

address\_book = addressbook\_pb2.AddressBook()

# Read the existing address book.

try:

f = open(sys.argv[1], "rb")

address\_book.ParseFromString(f.read())

f.close()

except IOError:

print sys.argv[1] + ": Could not open file. Creating a new one."

# Add an address.

PromptForAddress(address\_book.person.add())

# Write the new address book back to disk.

f = open(sys.argv[1], "wb")

f.write(address\_book.SerializeToString())

f.close()

## Reading A Message

当然，一个地址簿，如果你不能从中得到任何信息，那也不会有多大用处！这个例子为从上面的示例读取文件并打印其所有信息。

#! /usr/bin/python

import addressbook\_pb2

import sys

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

def ListPeople(address\_book):

for person in address\_book.person:

print "Person ID:", person.id

print " Name:", person.name

if person.HasField('email'):

print " E-mail address:", person.email

for phone\_number in person.phone:

if phone\_number.type == addressbook\_pb2.Person.MOBILE:

print " Mobile phone #: ",

elif phone\_number.type == addressbook\_pb2.Person.HOME:

print " Home phone #: ",

elif phone\_number.type == addressbook\_pb2.Person.WORK:

print " Work phone #: ",

print phone\_number.number

# Main procedure: Reads the entire address book from a file and prints all

# the information inside.

if len(sys.argv) != 2:

print "Usage:", sys.argv[0], "ADDRESS\_BOOK\_FILE"

sys.exit(-1)

address\_book = addressbook\_pb2.AddressBook()

# Read the existing address book.

f = open(sys.argv[1], "rb")

address\_book.ParseFromString(f.read())

f.close()

ListPeople(address\_book)

## Extending a Protocol Buffer

迟早你会发布使用protocol buffer的代码，毫无疑问你会想“改善” protocol buffer的定义。如果你想让你的新buffers 是向后兼容的，又或者你的旧buffers是向前兼容的－那你就肯定想这样做－那这里有些规则你需要遵守了。在新版本的protocol buffer中：

* 你不能改变任何现有的标签数据字段。
* 你不能添加或删除任何repeated的字段。
* 你可以删除optional 或repeated 字段。
* 你可以添加新的optional 或repeated 字段但你必须使用新的标签号码。（即在protocol buffer从来没有使用过的标记数字，甚至通过删除字段）

这些规则有一些[例外](https://developers.google.com/protocol-buffers/docs/proto.html#updating),但他们很少使用  
如果你遵 守这些规则，旧的代码会很高兴地读新的messages并简单地忽略所有新字段。对于旧代码来说，被删除的可选optional 字段,简单地拥有它们的默认值，被删除的repeated字段会变为空。新代码会明显地读到旧代码。但是，记住，新的optional字段不会显示旧的message，所以你需要明确地检查它们是否有has\_设置，又或者提供一个合理的缺省的值在[default = value]的 标签数字后，在你的.proto文件上。如果没有为一个optional无线设置缺省的值，一个字符串的值会替代缺省的值，这个缺省的值则为一个空串。对 于布尔类型来说，缺省为false。对于整型类型来说，缺省为0。还有记住，如果你添加一个新repeated字段，你的新代码将无法判断它是空的(通过 新代码)或从未设置(旧代码)由于没有设置has\_ flag给它。  
  
本文翻译自：<https://developers.google.com/protocol-buffers/docs/pythontutorial>