**ProtoBuf开发者指南**

**译者:**

gashero

目录

* **1   概览** 
  + **1.1   什么是protocol buffer**
  + **1.2   他们如何工作**
  + **1.3   为什么不用XML?**
  + **1.4   听起来像是为我的解决方案，如何开始?**
  + **1.5   一点历史**
* **2   语言指导** 
  + **2.1   定义一个消息类型**
  + **2.2   值类型**
  + **2.3   可选字段与缺省值**
  + **2.4   枚举**
  + **2.5   使用其他消息类型**
  + **2.6   嵌套类型**
  + **2.7   更新一个数据类型**
  + **2.8   扩展**
  + **2.9   包**
  + **2.10   定义服务**
  + **2.11   选项**
  + **2.12   生成你的类**
* **3   代码风格指导** 
  + **3.1   消息与字段名**
  + **3.2   枚举**
  + **3.3   服务**
* **4   编码** 
  + **4.1   一个简单的消息**
  + **4.2   基于128的Varints**
  + **4.3   消息结构**
  + **4.4   更多的值类型**
  + **4.5   内嵌消息**
  + **4.6   可选的和重复的元素**
  + **4.7   字段顺序**
* **5   ProtocolBuffer基础：C++**
* **6   ProtocolBuffer基础：Java**
* **7   ProtocolBuffer基础：Python** 
  + **7.1   为什么使用ProtocolBuffer?**
  + **7.2   哪里可以找到例子代码**
  + **7.3   定义你的协议格式**
  + **7.4   编译你的ProtocolBuffer**
  + **7.5   ProtocolBuffer API** 
    - **7.5.1 枚举**
    - **7.5.2 标准消息方法**
    - **7.5.3 解析与串行化**
  + **7.6   写消息**
  + **7.7   读消息**
  + **7.8   扩展ProtocolBuffer**
  + **7.9   高级使用**
* **8   参考概览**
* **9   C++代码生成**
* **10   C++ API**
* **11   Java代码生成**
* **12   Java API**
* **13   Python代码生成** 
  + **13.1   编译器的使用**
  + **13.2   包**
  + **13.3   消息**
  + **13.4   字段** 
    - **13.4.1 简单字段**
    - **13.4.2 简单消息字段**
    - **13.4.3 重复字段**
    - **13.4.4 重复消息字段**
    - **13.4.5 枚举类型**
    - **13.4.6 扩展**
  + **13.5   服务** 
    - **13.5.1 接口**
    - **13.5.2 存根(Stub)**
* **14   Python API**
* **15   其他语言**

**1   概览**

欢迎来到protocol buffer的开发者指南文档，一种语言无关、平台无关、扩展性好的用于通信协议、数据存储的结构化数据串行化方法。

本文档面向希望使用protocol buffer的Java、C++或Python开发者。这个概览介绍了protocol buffer，并告诉你如何开始，你随后可以跟随编程指导( http://code.google.com/apis/protocolbuffers/docs/tutorials.html )深入了解protocol buffer编码方式( http://code.google.com/apis/protocolbuffers/docs/encoding.html )。API参考文档( http://code.google.com/apis/protocolbuffers/docs/reference/overview.html )同样也是提供了这三种编程语言的版本，不够协议语言( http://code.google.com/apis/protocolbuffers/docs/proto.html )和样式( http://code.google.com/apis/protocolbuffers/docs/style.html )指导都是编写 .proto 文件。

**1.1   什么是protocol buffer**

ProtocolBuffer是用于结构化数据串行化的灵活、高效、自动的方法，有如XML，不过它更小、更快、也更简单。你可以定义自己的数据结构，然后使用代码生成器生成的代码来读写这个数据结构。你甚至可以在无需重新部署程序的情况下更新数据结构。

**1.2   他们如何工作**

你首先需要在一个 .proto 文件中定义你需要做串行化的数据结构信息。每个ProtocolBuffer信息是一小段逻辑记录，包含一系列的键值对。这里有个非常简单的 .proto 文件定义了个人信息:

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;  
}

有如你所见，消息格式很简单，每个消息类型拥有一个或多个特定的数字字段，每个字段拥有一个名字和一个值类型。值类型可以是数字(整数或浮点)、布尔型、字符串、原始字节或者其他ProtocolBuffer类型，还允许数据结构的分级。你可以指定可选字段，必选字段和重复字段。你可以在( http://code.google.com/apis/protocolbuffers/docs/proto.html )找到更多关于如何编写 .proto 文件的信息。

一旦你定义了自己的报文格式(message)，你就可以运行ProtocolBuffer编译器，将你的 .proto 文件编译成特定语言的类。这些类提供了简单的方法访问每个字段(像是 query() 和 set\_query() )，像是访问类的方法一样将结构串行化或反串行化。例如你可以选择C++语言，运行编译如上的协议文件生成类叫做 Person 。随后你就可以在应用中使用这个类来串行化的读取报文信息。你可以这么写代码:

Person person;  
person.set\_name("John Doe");  
person.set\_id(1234);  
person.set\_email("jdoe@example.com");  
fstream.output("myfile",ios::out | ios::binary);  
person.SerializeToOstream(&output);

然后，你可以读取报文中的数据:

fstream input("myfile",ios::in | ios:binary);  
Person person;  
person.ParseFromIstream(&input);  
cout << "Name: " << person.name() << endl;  
cout << "E-mail: " << person.email() << endl;

你可以在不影响向后兼容的情况下随意给数据结构增加字段，旧有的数据会忽略新的字段。所以如果使用ProtocolBuffer作为通信协议，你可以无须担心破坏现有代码的情况下扩展协议。

你可以在API参考( http://code.google.com/apis/protocolbuffers/docs/reference/overview.html )中找到完整的参考，而关于ProtocolBuffer的报文格式编码则可以在( http://code.google.com/apis/protocolbuffers/docs/encoding.html )中找到。

**1.3   为什么不用XML?**

ProtocolBuffer拥有多项比XML更高级的串行化结构数据的特性，ProtocolBuffer：

· 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ProtocolBuffer报文则如下:

#ProtocolBuffer的文本表示  
#这不是正常时使用的二进制数据  
person {  
name: "John Doe"  
email: "jdoe@example.com"  
}

当这个报文编码到ProtocolBuffer的二进制格式( http://code.google.com/apis/protocolbuffers/docs/encoding.html )时(上面的文本仅用于调试和编辑)，它只需要28字节和100-200ns的解析时间。而XML的版本需要69字节(除去空白)和5000-10000ns的解析时间。

当然，操作ProtocolBuffer也很简单:

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

而XML的你需要:

cout << "Name: "  
<< person.getElementsByTagName("name")->item(0)->innerText()  
<< endl;  
cout << "E-mail: "  
<< person.getElementsByTagName("email")->item(0)->innerText()  
<< end;

当然，ProtocolBuffer并不是在任何时候都比XML更合适，例如ProtocolBuffer无法对一个基于标记文本的文档建模，因为你根本没法方便的在文本中插入结构。另外，XML是便于人类阅读和编辑的，而ProtocolBuffer则不是。还有XML是自解释的，而 ProtocolBuffer仅在你拥有报文格式定义的 .proto 文件时才有意义。

**1.4   听起来像是为我的解决方案，如何开始?**

下载包( http://code.google.com/p/protobuf/downloads/ )，包含了Java、Python、C++的ProtocolBuffer编译器，用于生成你需要的IO类。构建和安装你的编译器，跟随README的指令就可以做到。

一旦你安装好了，就可以跟着编程指导( http://code.google.com/apis/protocolbuffers/docs/tutorials.html )来选择语言-随后就是使用ProtocolBuffer创建一个简单的应用了。

**1.5   一点历史**

ProtocolBuffer最初是在Google开发的，用以解决索引服务器的请求、响应协议。在使用ProtocolBuffer之前，有一种格式用以处理请求和响应数据的编码和解码，并且支持多种版本的协议。而这最终导致了丑陋的代码，有如:

if (version==3) {  
...  
}else if (version>4) {  
if (version==5) {  
...  
}  
...  
}

通信协议因此变得越来越复杂，因为开发者必须确保，发出请求的人和接受请求的人必须同时兼容，并且在一方开始使用新协议时，另外一方也要可以接受。

ProtocolBuffer设计用于解决这一类问题：

· 很方便引入新字段，而中间服务器可以忽略这些字段，直接传递过去而无需理解所有的字段。

· 格式可以自描述，并且可以在多种语言中使用(C++、Java等)

然而用户仍然需要手写解析代码。

随着系统的演化，他需要一些其他的功能：

· 自动生成编码和解码代码，而无需自己编写解析器。

· 除了用于简短的RPC(Remote Procedure Call)请求，人们使用ProtocolBuffer来做数据存储格式(例如BitTable)。

· RPC服务器接口可以作为 .proto 文件来描述，而通过ProtocolBuffer的编译器生成存根(stub)类供用户实现服务器接口。

ProtocolBuffer现在已经是Google的混合语言数据标准了，现在已经正在使用的有超过48,162种报文格式定义和超过12,183个 .proto 文件。他们用于RPC系统和持续数据存储系统。

**2   语言指导**

本指导描述了如何使用ProtocolBuffer语言来定义结构化数据类型，包括 .proto 文件的语法和如何生成存取类。

这是一份指导手册，一步步的例子使用文档中的多种功能，查看入门指导( http://code.google.com/apis/protocolbuffers/docs/tutorials.html )选择你的语言。

**2.1   定义一个消息类型**

@waiting …

**2.2   值类型**

@waiting …

**2.3   可选字段与缺省值**

@waiting …

**2.4   枚举**

@waiting …

**2.5   使用其他消息类型**

@waiting …

**2.6   嵌套类型**

@waiting …

**2.7   更新一个数据类型**

@waiting …

**2.8   扩展**

@waiting …

**2.9   包**

@waiting …

**2.10   定义服务**

@waiting …

**2.11   选项**

@waiting …

**2.12   生成你的类**

@waiting …

**3   代码风格指导**

本文档提供了 .proto 文件的代码风格指导。按照惯例，你将会，你将会生成一些便于阅读和一致的ProtocolBuffer定义文件。

**3.1   消息与字段名**

使用骆驼风格的大小写命名，即单词首字母大写，来做消息名。使用GNU的全部小写，使用下划线分隔的方式定义字段名:

message SongServerRequest {  
required string song\_name=1;  
}

使用这种命名方式得到的名字如下:

C++:  
const string& song\_name() {...}  
void set\_song\_name(const string& x) {...}  
Java:  
public String getSongName() {...}  
public Builder setSongName(String v) {...}

**3.2   枚举**

使用骆驼风格做枚举名，而用全部大写做值的名字:

enum Foo {  
FIRST\_VALUE=1;  
SECOND\_VALUE=2;  
}

每个枚举值最后以分号结尾，而不是逗号。

**3.3   服务**

如果你的 .proto 文件定义了RPC服务，你可以使用骆驼风格:

service FooService {  
rpc GetSomething(FooRequest) returns (FooResponse);  
}

**4   编码**

本文档描述了ProtocolBuffer的串行化二进制数据格式定义。你如果仅仅是在应用中使用ProtocolBuffer，并不需要知道这些，但是这些会对你定义高效的格式有所帮助。

**4.1   一个简单的消息**

@waiting …

**4.2   基于128的Varints**

@waiting …

**4.3   消息结构**

@waiting …

**4.4   更多的值类型**

@waiting …

**4.5   内嵌消息**

@waiting …

**4.6   可选的和重复的元素**

@waiting …

**4.7   字段顺序**

@waiting …

**5   ProtocolBuffer基础：C++**

@waiting …

**6   ProtocolBuffer基础：Java**

@waiting …

**7   ProtocolBuffer基础：Python**

本指南给Python程序员一个快速使用的ProtocolBuffer的指导。通过一些简单的例子来在应用中使用ProtocolBuffer，它向你展示了如何：

· 定义 .proto 消息格式文件

· 使用ProtocolBuffer编译器

· 使用Python的ProtocolBuffer编程接口来读写消息

这并不是一个在Python中使用ProtocolBuffer的完整指导。更多细节请参考手册信息，查看语言指导( http://code.google.com/apis/protocolbuffers/docs/proto.html )，Python API( http://code.google.com/apis/protocolbuffers/docs/reference/python/index.html )，和编码手册( http://code.google.com/apis/protocolbuffers/docs/encoding.html )。

**7.1   为什么使用ProtocolBuffer?**

下面的例子”地址本”应用用于读写人的联系信息。每个人有name、ID、email，和联系人电话号码。

如何串行化和读取结构化数据呢？有如下几种问题：

· 使用Python的pickle，这是语言内置的缺省方法，不过没法演化，也无法让其他语言支持。

· 你可以发明一种数据编码方法，例如4个整数”12:3-23:67″，这是简单而灵活的方法，不过你需要自己写解析器代码，且只适用于简单的数据。

· 串行化数据到XML。这种方法因为可读性和多种语言的兼容函数库而显得比较吸引人，不过这也不是最好的方法，因为XML浪费空间是臭名昭著的，编码解码也很浪费时间。而XML DOM树也是很复杂的。

ProtocolBuffer提供了灵活、高效、自动化的方法来解决这些问题。通过ProtocolBuffer，只需要写一个 .proto 数据结构描述文件，就可以编译到几种语言的自动编码解码类。生成的类提供了setter和getter方法来控制读写细节。最重要的是 ProtocolBuffer支持后期扩展协议，而又确保旧格式可以兼容。

**7.2   哪里可以找到例子代码**

源码发行包中已经包含了，在”example”文件夹。

**7.3   定义你的协议格式**

想要创建你的地址本应用，需要开始于一个 .proto 文件。定义一个 .proto 文件很简单：添加一个消息到数据结构，然后指定一个和一个类型到每一个字段，如下是本次例子使用的 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;  
}

有如你所见的，语法类似于C++或Java。让我们分块理解他们。

@waiting …

**7.4   编译你的ProtocolBuffer**

现在已经拥有了 .proto 文件，下一步就是编译生成相关的访问类。运行编译器 **protoc** 编译你的 .proto 文件。

1. 如果还没安装编译器则下载并按照README的安装。

2. 运行编译器，指定源目录和目标目录，定位你的 .proto 文件到源目录，然后执行:

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

因为需要使用Python类，所以 –python\_out 选项指定了特定的输出语言。

这个步骤会生成 addressbook\_pb2.py 到目标目录。

**7.5   ProtocolBuffer API**

不像生成的C++和Java代码，Python生成的类并不会直接为你生成存取数据的代码。而是(有如你在 addressbook\_pb2.py 中见到的)生成消息描述、枚举、和字段，还有一些神秘的空类，每个对应一个消息类型:

class Person(message.Message):  
\_\_metaclass\_\_=reflection.GeneratedProtocolMessageType  
class PhoneNumber(message.Message):  
\_\_metaclass\_\_=reflection.GeneratedProtocolMessageType  
DESCRIPTION=\_PERSON\_PHONENUMBER  
DESCRIPTOR=\_PERSON  
class AddressBook(message.Message):  
\_\_metaclass\_\_=reflection.GeneratedProtocolMessageType  
DESCRIPTOR=\_ADDRESSBOOK

这里每个类最重要的一行是 \_\_metaclass\_\_=reflection.GeneratedProtocolMessageType 。通过Python的元类机制工作，你可以把他们看做是生成类的模板。在载入时， GeneratedProtocolMessageType 元类使用特定的描述符创建Python方法。随后你就可以使用完整的功能了。

最后就是你可以使用 Person 类来操作相关字段了。例如你可以写:

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

需要注意的是这些赋值属性并不是简单的增加新字段到Python对象，如果你尝试给一个 .proto 文件中没有定义的字段赋值，就会抛出 AttributeError 异常，如果赋值类型错误会抛出 TypeError 。在给一个字段赋值之前读取会返回缺省值:

person.no\_such\_field=1 #raise AttributeError  
person.id="1234" #raise TypeError

更多相关信息参考( http://code.google.com/apis/protocolbuffers/docs/reference/python-generated.html )。

**7.5.1 枚举**

枚举在元类中定义为一些符号常量对应的数字。例如常量 addressbook\_pb2.Person.WORK 拥有值2。

**7.5.2 标准消息方法**

每个消息类包含一些其他方法允许你检查和控制整个消息，包括：

· IsInitialized() ：检查是否所有必须(required)字段都已经被赋值了。

· \_\_str\_\_() ：返回人类可读的消息表示，便于调试。

· CopyFrom(other\_msg) ：使用另外一个消息的值来覆盖本消息。

· Clear() ：清除所有元素的值，回到初识状态。

这些方法是通过接口 Message 实现的，更多消息参考( http://code.google.com/apis/protocolbuffers/docs/reference/python/google.protobuf.message.Message-class.html )。

**7.5.3 解析与串行化**

最后，每个ProtocolBuffer类有些方法用于读写消息的二进制数据( http://code.google.com/apis/protocolbuffers/docs/encoding.html )。包括：

· SerializeToString() ：串行化，并返回字符串。注意是二进制格式而非文本。

· ParseFromString(data) ：解析数据。

他们是成对使用的，提供二进制数据的串行化和解析。另外参考消息API参考( http://code.google.com/apis/protocolbuffers/docs/reference/python/google.protobuf.message.Message-class.html )了解更多信息。

Note

ProtocolBuffer与面向对象设计

ProtocolBuffer类只是用于存取数据的，类似于C++中的结构体，他们并没有在面向对象方面做很好的设计。如果你想要给这些类添加更多的行为，最好的方法是包装(wrap)。包装同样适合于复用别人写好的 .proto 文件。这种情况下，你可以把ProtocolBuffer生成类包装的很适合于你的应用，并隐藏一些数据和方法，暴露有用的函数等等。 **你不可以通过继承来给自动生成的类添加行为。** 这会破坏他们的内部工作机制。

**7.6   写消息**

现在开始尝试使用ProtocolBuffer的类。第一件事是让地址本应用可以记录联系人的细节信息。想要做这些需要先创建联系人实例，然后写入到输出流。

这里的程序从文件读取地址本，添加新的联系人信息，然后写回新的地址本到文件。

#! /usr/bin/python  
import addressbook\_pb2  
import sys  
#这个函数使用用户输入填充联系人信息  
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."  
#主函数，从文件读取地址本，添加新的联系人，然后写回到文件  
if len(sys.argv)!=2:  
print "Usage:",sys.argv[0],"ADDRESS\_BOOK\_FILE"  
sys.exit(-1)  
address\_book=addressbook\_pb2.AddressBook()  
#读取已经存在的地址本  
try:  
f=open(sys.argv[1],"fb")  
address\_book.ParseFromString(f.read())  
f.close()  
except OSError:  
print sys.argv[1]+": Count open file. Creating a new one."  
#添加地址  
PromptFromAddress(address\_book.person.add())  
#写入到文件  
f=open(sys.argv[1],"wb")  
f.write(address\_book.SerializeToString())  
f.close()

**7.7   读消息**

当然，一个无法读取的地址本是没什么用处的，这个例子读取刚才创建的文件并打印所有信息:

#! /usr/bin/python  
import addressbook\_pb2  
import sys  
#遍历地址本中所有的人并打印出来  
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:",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  
#主函数，从文件读取地址本  
if len(sys.argv)!=2:  
print "Usage:",sys.argv[0],"ADDRESS\_BOOK\_FILE"  
sys.exit(-1)  
address\_book=addressbook\_pb2.AddressBook()  
#读取整个地址本文件  
f=open(sys.argv[1],"rb")  
address\_book.ParseFromString(f.read())  
f.close()  
ListPeople(address\_book)

**7.8   扩展ProtocolBuffer**

在你发不了代码以后，可能会想要改进ProtocolBuffer的定义。如果你想新的数据结构向后兼容，而你的旧数据可以向前兼容，那么你就找对了东西了，不过有些规则需要遵守。在新版本的ProtocolBuffer中：

· 必须不可以改变已经存在的标签的数字。

· 必须不可以增加或删除必须(required)字段。

· 可以删除可选(optional)或重复(repeated)字段。

· 可以添加新的可选或重复字段，但是必须使用新的标签数字，必须是之前的字段所没有用过的。

这些规则也有例外( http://code.google.com/apis/protocolbuffers/docs/proto.html#updating )，不过很少使用。

如果你遵从这些规则，旧代码会很容易的读取新的消息，并简单的忽略新的字段。而对旧的被删除的可选字段也会简单的使用他们的缺省值，被删除的重复字段会自动为空。新的代码也会透明的读取旧的消息。然而，需要注意的是新的可选消息不会在旧的消息中显示，所以你需要使用 has\_ 严格的检查他们是否存在，或者在 .proto 文件中提供一个缺省值。如果没有缺省值，就会有一个类型相关的默认缺省值：对于字符串就是空字符串；对于布尔型则是false；对于数字类型默认为0。同时要注意的是如果你添加了新的重复字段，你的新代码不会告诉你这个字段为空(新代码)也不会，也不会(旧代码)包含 has\_ 标志。

**7.9   高级使用**

ProtocolBuffer不仅仅提供了数据结构的存取和串行化。查看Python API参考( http://code.google.com/apis/protocolbuffers/docs/reference/python/index.html )了解更多功能。

一个核心功能是通过消息类的映射(reflection)提供的。你可以通过它遍历消息的所有字段，和管理他们的值。关于映射的一个很有用的地方是转换到其他编码，如XML或JSON。一个使用映射的更高级的功能是寻找同类型两个消息的差异，或者开发出排序、正则表达式等功能。使用你的创造力，还可以用ProtocolBuffer实现比你以前想象的更多的问题。

映射是通过消息接口提供的。

**8   参考概览**

@waiting …

**9   C++代码生成**

@waiting …

**10   C++ API**

@waiting …

**11   Java代码生成**

@waiting …

**12   Java API**

@waiting …

**13   Python代码生成**

本页提供了Python生成类的相关细节。你可以在阅读本文档之前查看语言指导。

Python的ProtocolBuffer实现与C++和Java的略有不同，编译器只输出构建代码的描述符来生成类，而由Python的元类来执行工作。本文档描述了元类开始生效以后的东西。

**13.1   编译器的使用**

ProtocolBuffer通过编译器的 –python\_out= 选项来生成Python的相关类。这个参数实际上是指定输出的Python类放在哪个目录下。编译器会为每个 .proto 文件生成一个对应的 .py 文件。输出文件名与输入文件名相关，不过有两处修改：

· 扩展名 .proto 改为 .py 。

· 路径名的修改。

如果你按照如下调用编译器:

protoc --proto\_path=src --python\_out=build/gen src/foo.proto src/bar/baz.proto

编译器会自动读取两个 .proto 文件然后产生两个输出文件。在需要时编译器会自动创建目录，不过 –python\_out 指定的目录不会自动创建。

需要注意的是，如果 .proto 文件名或路径包含有无法在Python中使用的模块名(如连字符)，就会被自动转换为下划线。所以文件 foo-bar.proto 会变成 foo\_bar\_pb2.py 。

Note

在每个文件后缀的 \_pb2.py 中的2代表ProtocolBuffer版本2。版本1仅在Google内部使用，但是你仍然可以在以前发布的一些代码中找到它。自动版本2开始，ProtocolBuffer开始使用完全不同的接口了，从此Python也没有编译时类型检查了，我们加上这个版本号来标志Python文件名。

**13.2   包**

Python代码生成根本不在乎包的名字。因为Python使用目录名来做包名。

**13.3   消息**

先看看一个简单的消息声明:

message Foo {}

ProtocolBuffer编译器会生成类Foo，它是 google.protobuf.Message 的子类。这个实体类，不含有虚拟方法。不像C++和Java，Python生成类对优化选项不感冒；实际上Python的生成代码已经为代码大小做了优化。

你不能继承Foo的子类。生成类被设计不可以被继承，否则会被打破一些设计。另外，继承本类也是不好的设计。

Python的消息类没有特定的公共成员，而是定义接口，极其嵌套的字段、消息和枚举类型。

一个消息可以在另外一个消息中声明，例如 message Foo { message Bar {}} 。在这种情况下，Bar类定义为Foo的一个静态成员，所以你可以通过 Foo.Bar 来引用。

**13.4   字段**

对于消息类型中的每一个字段，都有对应的同名成员。

**13.4.1 简单字段**

如果你有一个简单字段(包括可选的和重复的)，也就是非消息字段，你可以通过简单字段的方式来管理，例如foo字段的类型是int32，你可以:

message.foo=123  
print message.foo

注意设置foo的值，如果类型错误会抛出TypeError。

如果foo在赋值之前就读取，就会使用缺省值。想要检查是否已经赋值，可以用 HasField() ，而清除该字段的值用 ClearField() 。例如:

assert not message.HasField("foo")  
message.foo=123  
assert message.HasField("foo")  
message.ClearField("foo")  
assert not message.HasField("foo")

**13.4.2 简单消息字段**

消息类型工作方式略有不同。你无法为一个嵌入消息字段赋值。而是直接操作这个消息的成员。因为实例化上层消息时，其包含的子消息同时也实例化了，例如定义:

message Foo {  
optional Bar bar=1;  
}  
message bar {  
optional int32 i=1;  
}

你不可以这么做，因为不能做消息类型字段的赋值:

foo=Foo()  
foo.bar=Bar() #WRONG!

而是可以直接对消息类型字段的成员赋值:

foo=Foo()  
assert not foo.HasField("bar")  
foo.bar.i=1  
assert foo.HasField("bar")

注意简单的读取消息类型字段的未赋值成员只不过是打印其缺省值:

foo=Foo()  
assert not foo.HasField("bar")  
print foo.bar.i #打印i的缺省值  
assert not foo.HasField("bar")

**13.4.3 重复字段**

重复字段表现的像是Python的序列类型。如果是嵌入的消息，你无法为字段直接赋值，但是你可以管理。例如给定的定义:

message Foo {  
repeated int32 nums=1;  
}

你就可以这么做:

foo=Foo()  
foo.nums.append(15)  
foo.nums.append(32)  
assert len(foo.nums)==2  
assert foo.nums[0]==15  
assert foo.nums[1]==32  
for i in foo.nums:  
print i  
foo.nums[1]=56  
assert foo.nums[1]==56

作为一种简单字段，清除该字段必须使用 ClearField() 。

**13.4.4 重复消息字段**

重复消息字段工作方式与重复字段很像，除了 add() 方法用于返回新的对象以外。例如如下定义:

message Foo {  
repeated Bar bar=1;  
}  
message Bar {  
optional int32 i=1;  
}

你可以这么做:

foo=Foo()  
bar=foo.bars.add()  
bar.i=15  
bar=foo.bars.add()  
bar.i=32  
assert len(foo.bars)==2  
assert foo.bars[0].i==15  
assert foo.bars[1].i==32  
for bar in foo.bars:  
print bar.i  
foo.bars[1].i=56  
assert foo.bars[1].i==56

**13.4.5 枚举类型**

@waiting …

**13.4.6 扩展**

@waiting …

**13.5   服务**

**13.5.1 接口**

一个简单的接口定义:

service Foo {  
rpc Bar(FooRequest) returns(FooResponse);  
}

ProtocolBuffer的编译器会生成类 Foo 来展示这个服务。 Foo 将会拥有每个服务定义的方法。在这种情况下 Bar 方法的定义是:

def Bar(self,rpc\_controller,request,done)

参数等效于 Service.CallMethod() ，除了隐含的 method\_descriptor 参数。

这些生成的方法被定义为可以被子类重载。缺省实现只是简单的调用 controller.SetFailed() 而抛出错误信息告之尚未实现。然后调用done回调。在实现你自己的服务时，你必须继承生成类，然后重载各个接口方法。

Foo继承了 Service 接口。ProtocolBuffer编译器会自动声响相关的实现方法：

· GetDescriptor ：返回服务的 ServiceDescriptor 。

· CallMethod ：检测需要调用哪个方法，并且直接调用。

· GetRequestClass 和 GetResponseClass ：返回指定方法的请求和响应类。

**13.5.2 存根(Stub)**

ProtocolBuffer编译器也会为每个服务接口提供一个存根实现，用于客户端发送请求到服务器。对于Foo服务，存根实现是 Foo\_Stub 。

Foo\_Stub 是Foo的子类，他的构造器是一个 RpcChannel 。存根会实现调用每个服务方法的 CallMethod() 。

ProtocolBuffer哭并不包含RPC实现。然而，它包含了你构造服务类的所有工具，不过选择RPC实现则随你喜欢。你只需要提供 RpcChannel 和 RpcController 的实现即可