# Protocol Buffer 基础: Java

Translated by Jeff

王晖 译

（译者联系方式为：lonelystellar@gmail.com）



注：关于本文档的翻译错误（包括语法错误，错别字）或中文技术交流，可以发送邮件至译者邮箱：lonelystellar@gmail.com，我们共同研究，共同进步，不胜感激。

本教程提供了 Protocol Buffers 的基本 Java 编程介绍.通过创建一个简单的示例程序,来告诉你怎么:

- 在 .proto 文件中定义消息格式.
- 使用 Protocol Buffer 编译器.
- 使用 Protocol Buffer 的 Java API 来读写消息.

这不是一个在 Java 中使用 Protocol Buffers 的全面的指南.如需更多更详细的参考信,请阅读 Protocol Buffer 语言指南, Java API 引用, Java 生成代码指南,以及编码参考.

## 为什么使用 Protocol Buffers?

下面将使用一个非常简单的程序"通讯录", 它能够从一个文件中读和写某个人的练习信息. 每个在通讯录中的人都有一个名字, ID, email 地址, 和一个手机号.
那么怎么序列化和放序列化这样结构的数据呢? 下面是几种解决方法:

- 使用 Java 的序列化. 这是一个默认的实现因为他被植入到 Java 语言中了, 但是他还有一些已知的问题(参见 Effective Java Josh Bloch 著,213 页), 并且不能与 C++或 Python 共享数据..
- 自定义一个特定的方式来编码数据项到一个字符串中– 例如编码整数 4 为 "12:3:-23:67". 这个实现非常的简单并且非常的有灵活性, 虽然它不需要编写通断的编码和解码的代码, 但是解析规定需要一个非常小的 运行时消耗. 这种方式对于非常简单的数据很有效.
- 序列化数据为 XML. 这个实现自从 XML 变成一种易读的并且有许多语言来支持它自后就变得非常的受欢迎. 如果需要共享数据给其他程序/项目时是一个不错的选择. 但是 XML 需要占用很多的空间, 并且编码/解码的在程序性能上的消耗巨大. 还有定位一个 XML DOM 树比在类中定位一个简单的字段要复杂的多.

Protocol buffers 就非常的灵活, 高效, 自动就解决上面所出现的这些问题. 有了 protocol buffers, 编写一个描述了希望存储的数据结构的.proto 文件. protocol buffer 编译器利用它来创建一个拥有高效二进制格式的实现自动编码和解析 protocol buffer 数据的类. 生成的类中含有字段的 getter 和 setter 方法 ,使 protocol buffer 作

为一个单元来管理读写 protocol buffer. 重要的是, protocol buffer 格式支持扩展某一种格式, 并仍能够读取用老格式编码的数据.

## 到哪里去找示例代码

示例代码都包含在源码包中, 放在"examples"目录下. 从这下载.

## 定义 Protocol 格式

为了创建通讯录程序, 首先需要从.proto 文件开始. 在.proto 文件中的定义都非常的简单: 为每个需要序列化的数据结构添加一个*消息*, 然后在消息中的每个字段指定一个名字和类型. 以下是一个定义了消息的.proto 文件,addressbook.proto.

```
package tutorial;


option java_package = "com.example.tutorial";

option java_outer_classname = "AddressBookProtos";


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;
}
```

可以看到, 语法跟 C++或 Java 类似. 下面分解这个文件的每个部分并看它都在干些什么.

.proto 文件的开始时一个包声明, 这可以避免多个不同项目之间的命名冲突. 在 Java 中, 这个包名就是 Java 的包名, 除非用 java_package 选项显示的指定了, 在这里, 即使用 java_package 指定了一个包名, 还是需要定义个普通的包 为了避免在非 Java 语言的 Protocol Buffer 命名空间中的名字冲突.

在包声明之后, 可以看到 2 个 Java 特定的选项: java_package 和 java_outer_classname. java_package 指定在生产的类中使用的 Java 包名. 如果没有显示的指定, 它会使用 package 声明中的包名字, 但是这些名字不是合适的 Java 包名(因为它们不是使用域名开头的). java_outer_classname 选项定义包含在此文件中所有的类的类名. 如果没有显示的指定 java_outer_classname, 它会把文件名转换成驼峰方式的名字来作为类名. 例如, "my_proto.proto" 默认的外部类名字就是 "MyProto" .

接下来, 就是消息定义. 一个消息就是由一组设置了类型的字段的集合组成的. 许多标准的简单数据类型都可以用来做为字段类型, 包括 bool, int32, float, double, 和 string. 还可以添加更多使用其他消息类型作为字段类型的结构到消息中– 在上面的例子中 Person 消息包含 PhoneNumber 消息, AddressBook 消息包含 Person 消息. 甚至还可以定义消息类型嵌套在其他消息中– 例如, PhoneNumber 类型定义在 Person 内部. 如果一个字段有预先定义好的值列表, 那么还能在把它定义为 enum 类型– 在这里可以指定手机号位 MOBILE, HOME, 或 WORK 中的一种.

每个元素的" = 1", " = 2"标记标明在二进制编码中每个字段都有一个唯一的”标签”. 标记编码 1-15 需要至少一个字节来编码对于更高的数字来说, 所以作为在普通的使用的或者重复的元素上使用标签可当做是一种优化, 标签号大于等于 16 是为不经常使用的可选可选元素准备的. 在重复字段中的每个元素需要重新编码标签号, 因此重复字段在此种优化时是个不错的选择..
每个字段都必须加上下面修饰符中的某一种:

- required: 必须指定字段的值, 否则消息将会认为是”未初始化的”. 如果编译一个未初始化的消息将会抛出一个 RuntimeException. 解析一个未初始化的消息将会抛出一个 IOException. 除此之外必填字段和可选字段都一样.
- optional: 字段可设值也可不设值. 如果可选字段没有设值, 那么将会使用它的默认值. 对于简单类型, 可以指定自定义的默认值, 就像上面的例子中电话号码的 type 一样. 否则将会使用系统的默认值: 数字类型是 0， 字符串类型是空字符串, 布尔型是 false,. 对于嵌套消息, 默认值总是”默认实例(default instance)”或”原生类型(prototype)”,并且他们的字段都未设置值. 调用访问器来获取没有显示的设置值的可选(或必选)字段时总是返回该字段的默认值.
- repeated: 字段可被重复许多次(包括 0 次). 重复字段值的顺序在 protocol buffer 中是被保留的. 可以把重复字段看做是动态大小的数组.

**必选是永久的** 在标记字段为 required 时要特别小心. 如果在某个地方想要停止写和发送必选字段, 简单的把字段改成可选字段是会出问题的– 老的接受器将会认为消息没有这个字段而认为是不完整的消息, 并且可能会在无意中拒绝或者丢弃它们.可以考虑为 buffer 编写应用程序特定的自定义验证方法. 在 Google 的一些工程师们得出结论使用 required 字段的坏处远多于好处; 他们更喜欢只是用 optional 和 repeated. 但是,这并代表全世界都是这个观点.

Protocol Buffer 语言指南可以找到编写.proto – 包括所有可能的字段类型. 不要去找类似于类继承关系的东西 – protocol buffers 不做那些.

## 编译 Your Protocol Buffers

现在已经有了一个.proto 文件了, 接下来就需要生成可以读写 AddressBook (还有 Person 和 PhoneNumber)消息的类. 要这么做的话, 就需要运行 protocol buffer 编译 protoc 来编译.proto 文件:

1. 如果还没有安装编译器, 下载此包 并按照 README 中指示操作.
2. 现在运行编译器, 指定源目录(应用程序的源代码所在目录– 如果没有给定值那么默认将使用当前目录), 目标目录 (需要生成代码的输出目录; 通常与 $SRC_DIR 相同), 还有.proto 的路径. 如下:

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

   因为需要 Java 类, 所以使用--java_out 选项– 所支持的其他语言也提供类似的选项.

这会在指定的目标目录中生成 com/example/tutorial/AddressBookProtos.java.

## The Protocol Buffer API

现在来看一些已经生成好的代码, 看看编译器为你生成了哪些类和方法. 在 AddressBookProtos.java 中, 可以看到定义了一个名叫 AddressBookProtos 的类, 内嵌在类中就如在 addressbook.proto 的每个消息中的类一样. 每个类都有它自己的 Builder 类用来为类创建实例. 可以在下面的 Builders vs. Messages 章节中找到更多关于 builder 的内容.

消息和 builders 都会为每个消息中的字段自动生成访问器方法; 消息在 builder 含有 getter 和 setter 方法时仅有 getter 方法. 下面是一些 Person 类的访问器(为了简洁省略具体实现):

```
// required string name = 1;
public boolean hasName();
public String getName();

// required int32 id = 2;
public boolean hasId();
public int getId();

// optional string email = 3;
public boolean hasEmail();
public String getEmail();

// repeated .tutorial.Person.PhoneNumber phone = 4;
```

```
public List<PhoneNumber> getPhoneList();
public int getPhoneCount();
public PhoneNumber getPhone(int index);
```

同时, Person.Builder 也有同样的 getters 和 setters 方法:

```
// required string name = 1;
public boolean hasName();
public java.lang.String getName();
public Builder setName(String value);
public Builder clearName();

// required int32 id = 2;
public boolean hasId();
public int getId();
public Builder setId(int value);
public Builder clearId();

// optional string email = 3;
public boolean hasEmail();
public String getEmail();
public Builder setEmail(String value);
public Builder clearEmail();

// repeated .tutorial.Person.PhoneNumber phone = 4;
public List<PhoneNumber> getPhoneList();
public int getPhoneCount();
public PhoneNumber getPhone(int index);
public Builder setPhone(int index, PhoneNumber value);
public Builder addPhone(PhoneNumber value);
public Builder addAllPhone(Iterable<PhoneNumber> value);
public Builder clearPhone();
```

可以看到, 每个字段都有简单的 JavaBeans 风格的 getter 和 setter 方法. 对于每个单个的字段还有 has 开头的 getter 方法,如果字段没有设置值就返回 true. 最后, 每个字段都有一个 clear 开头的方法用来清理字段的值使它变为空值的状态.

重复字段还有一个额外的方法– 一个 Count 方法 (即一个获取列表大小的简便方法), getters 和 setters 方法表示通过索引从列表中获取和设置一个特定的元素, add 方法表示添加一个新的元素到列表中, addAll 方法表示添加一整串的元素到列表中.

注意这些访问器为什么会使用驼峰命名法, 即便是.proto 文件中的名字使用的是小写加下划线的形式. 这个转换在 protocol buffer 编译器中式自动完成的, 以便于生成的类能匹配 Java 风格的格式. 在.proto 文件中应该始终使用小写加下划线格式的字段名字; 这保证了在所有的生成的语言中都有好的命名习惯. 更多关于好的.proto 格式的命名参见格式指南.

更多关于在任意特定字段的定义中 protocol 编译器会生成哪些成员的信息, 请参见 Java 生成代码引用.

**枚举和内部类**

生成的代码还包括一个 Java5 的枚举类型 PhoneType, 和一个内部类 Person:

```
public static enum PhoneType {
 MOBILE(0, 0),
 HOME(1, 1),
 WORK(2, 2),
 ;
 ...
}
```

内嵌类型 Person.PhoneNumber 就如你所期望的那样, 生成了一个内部类 Person.

**Builders vs. Messages**

通过 Protocol buffer 编译器生成的消息类都是*不可变的*. 一旦消息对象被构建了, 它就不能被改变了, 就像一个 Java 的 String 一样. 为了构建一个消息, 首先必须的构建一个 builder, 给每个想要设置给定值的字段设值, 然后调用 builder 的 build() 方法.

也许你注意到了 builder 的每个方法修改了消息并返回另外一个 builder. 返回的对象其实就是你用来调用方法的那个 builder. 这个做法在一行代码中使用 setter 方法链时非常方便.

下面是如何创建一个 Person 实例的例子:

```
Person john =
 Person.newBuilder()
   .setId(1234)
   .setName("John Doe")
   .setEmail("jdoe@example.com")
   .addPhone(
    Person.PhoneNumber.newBuilder()
      .setNumber("555-4321")
      .setType(Person.PhoneType.HOME))
   .build();
```

**标准消息方法**

每个消息和 builder 类都含有一些其他方法用来检测和操作整个消息, 如下:

- isInitialized(): 检测必选字段是否已经被设置了.

- toString(): 返回一个可读的消息描述, 一般用于调式.
- mergeFrom(Message other): (仅限 builder) 合并 other 消息的内容到此消息中, 它会覆盖单个字段并串联起重复的字段.
- clear(): (仅限 builder) 清理所有的字段是其回到空值状态.

这些方法都实现自 Message 和 Message.Builder 接口, 并共享给所有的 Java 消息和 builders 使用. 了解更多, 参见完整的 Message API 文档.

**解析和序列化**

最后, 每个 protocol buffer 类都有读写消息中使用 protocol buffer 二进制格式 的类型. 这包括:

- byte[] toByteArray();: 序列化消息并返回一个字节数组, 其中包含了它的原生字节.
- static Person parseFrom(byte[] data);: 从给定的字节数组中解析一个消息.
- void writeTo(OutputStream output);: 序列化消息并把它写到一个 OutputStream 中.
- static Person parseFrom(InputStream input);: 从 InputStream 中读取和解析一个消息.

这些只是提供给解析和序列化的几个方法. 同样, 需要完整的列表可参见 Message API 引用.

**Protocol Buffers 和面向对象设计(O-O Design)** Protocol buffer 类都基本上都是愚蠢的数据持有者(就像 C++中的 structs 一样); 他们不会在对象模型中创造出一个好的类. 如果想要在生成的类中添加更多功能, 最好的方法就是在应用程序的特定类中包装生成好的 protocol buffer 类. 当在不完全了解.proto 文件的设计(例如, 从其他项目中拿过来的文件)时包装 protocol buffer 同样是一个很好选择. 此时, 就可以使用包装类来为应用程序手工编写一个更能适合那个环境的接口: 隐藏一些数据和方法, 暴露一些便利的功能出去, 等等. **注意永远不要通过继承生成的类来添加功能**. 这回破坏内部机制并且不是一个好的面向对象的实现.

## 写一个消息

现在可以尝试使用 protocol buffer 类了. 首先需要的是通讯录程序能够把一个人的详细信息写进通讯录的文件中. 那么就需要参见和填充 protocol buffer 的实例并且把它写到输出流中.

下面是一个程序用来从一个文件中读取一个 AddressBook, 并通过用户的输入添加一个新的 Person 进去, 然后把新的 AddressBook 在重新写回到文件中. 直接调用或引用 protocol 编译器生成的代码的部分都被高亮出来了.

```
import com.example.tutorial.AddressBookProtos.AddressBook;
import com.example.tutorial.AddressBookProtos.Person;
import java.io.BufferedReader;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.InputStreamReader;
import java.io.IOException;
import java.io.PrintStream;

class AddPerson {
  // This function fills in a Person message based on user input.
  static Person PromptForAddress(BufferedReader stdin,
                                 PrintStream stdout) throws IOException {
    Person.Builder person = Person.newBuilder();

    stdout.print("Enter person ID: ");
    person.setId(Integer.valueOf(stdin.readLine()));

    stdout.print("Enter name: ");
    person.setName(stdin.readLine());

    stdout.print("Enter email address (blank for none): ");
    String email = stdin.readLine();
    if (email.length() > 0) {
      person.setEmail(email);
    }

    while (true) {
      stdout.print("Enter a phone number (or leave blank to finish): ");
      String number = stdin.readLine();
      if (number.length() == 0) {
        break;
      }

      Person.PhoneNumber.Builder phoneNumber =
        Person.PhoneNumber.newBuilder().setNumber(number);

      stdout.print("Is this a mobile, home, or work phone? ");
      String type = stdin.readLine();
      if (type.equals("mobile")) {
        phoneNumber.setType(Person.PhoneType.MOBILE);
      } else if (type.equals("home")) {
        phoneNumber.setType(Person.PhoneType.HOME);
      } else if (type.equals("work")) {
        phoneNumber.setType(Person.PhoneType.WORK);
      } else {
        stdout.println("Unknown phone type.  Using default.");
      }
```

```
      person.addPhone(phoneNumber);
    }

    return person.build();
  }

  // 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.
  public static void main(String[] args) throws Exception {
    if (args.length != 1) {
      System.err.println("Usage:  AddPerson ADDRESS_BOOK_FILE");
      System.exit(-1);
    }

    AddressBook.Builder addressBook = AddressBook.newBuilder();

    // Read the existing address book.
    try {
      addressBook.mergeFrom(new FileInputStream(args[0]));
    } catch (FileNotFoundException e) {
      System.out.println(args[0] + ": File not found.  Creating a new file.");
    }

    // Add an address.
    addressBook.addPerson(
      PromptForAddress(new BufferedReader(new InputStreamReader(System.in)),
                       System.out));

    // Write the new address book back to disk.
    FileOutputStream output = new FileOutputStream(args[0]);
    addressBook.build().writeTo(output);
    output.close();
  }
}
```

## 读一个消息

当然, 如果不能从通讯录中获得任何信息那么这个程序就没什么用处! 下面这个例子读取通过上面的例子创建的文件并打印其中所有的信息.

```
import com.example.tutorial.AddressBookProtos.AddressBook;
import com.example.tutorial.AddressBookProtos.Person;
import java.io.FileInputStream;
import java.io.IOException;
import java.io.PrintStream;

class ListPeople {
  // Iterates though all people in the AddressBook and prints info about them.
  static void Print(AddressBook addressBook) {
    for (Person person: addressBook.getPersonList()) {
```

```
    System.out.println("Person ID: " + person.getId());
    System.out.println("  Name: " + person.getName());
    if (person.hasEmail()) {
      System.out.println("  E-mail address: " + person.getEmail());
    }

    for (Person.PhoneNumber phoneNumber : person.getPhoneList()) {
      switch (phoneNumber.getType()) {
        case MOBILE:
          System.out.print("  Mobile phone #: ");
          break;
        case HOME:
          System.out.print("  Home phone #: ");
          break;
        case WORK:
          System.out.print("  Work phone #: ");
          break;
      }
      System.out.println(phoneNumber.getNumber());
    }
  }
}

// Main function:  Reads the entire address book from a file and prints all
//   the information inside.
public static void main(String[] args) throws Exception {
  if (args.length != 1) {
    System.err.println("Usage:  ListPeople ADDRESS_BOOK_FILE");
    System.exit(-1);
  }

  // Read the existing address book.
  AddressBook addressBook =
    AddressBook.parseFrom(new FileInputStream(args[0]));

  Print(addressBook);
}
}
```

## 扩展一个 Protocol Buffer

在发布使用了 protocol buffer 的代码后, 无疑将会想要”改进”protocol buffer 的定义. 如果想要新的 buffers 能够向后兼容, 新的 buffers 能够向前兼容– 那么下面这些规则必须 遵守. 在 protocol buffer 的新版本中:

- *不能*改变任何已存在的标签编号..
- *不能*添加和删除任何必字段.
- *可以*删除可选的或重复的字段..

- *可以*添加新的可选的或重复的字段, 但是必须使用全新的标签编号(例如, 标签编号必须是从来在这个 protocol buffer 使用过的, 即使是曾经被已删除的字段使用过的编号).

(对于这些规则还有一些例外, 但是它们都很少使用.)

如果按照这些规则, 老的代码能很好的阅读新的消息并忽略那些新的字段. 对于老的代码, 被删除的可选字段将会使用它们的默认值, 被删除的可重复字段将会置空. 新代码对于阅读老消息是透明的. 但是, 记住新的可选字段不会在老的消息中出现, 因此如果它们有 has_或者在.proto 文件中在标签编号之后通过[default = value] 设置了默认值的就需要检查可见性. 如果可选元素的默认值没有被指定, 那么将会使用一个特定类型的默认值代替: 对于字符串, 默认值是空字符串. 对于布尔型, 默认值是 false, 对于数字类型, 默认值是 0, 注意如果添加了一个新的可重复字段, 那么将没有人告诉新的代码到底它是被设置成空的(由新代码)还是从来就没有设置过值(由老代码), 因为它没有 has_ 标记.

## 高级用途

Protocol buffers 使用非常的简单的访问器和序列化机制. 可以通过漫游 Java API 引用来看看在它上面还有什么可做的.

Protocol buffer 提供的一个关键的特性就是*反射*. 可以迭代消息中的字段并且在无需编码针对任意一个特定消息类型代码的情况下操作他们的的值. 一个非常有用的方法来使用反射就是在其他编码的消息(例如, XML 或 JSON)和 protocol buffer 的消息之间相互转换. 一个反射的更高级使用就是找出 2 个消息中同一类型的区别, 或者"protocol 消息的通用表达式"来是你能够编写匹配指定消息内容的表达式. 如果想象力够丰富, 可能比最初预期的应用 Protocol Buffers 要更广!

反射是作为 Message 和 Message.Builder 接口的一部分来提供的.