开发指南

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。

迎使用 Protocol Buffers 开发文档 - 一个语言中立，平台中立的，可扩展方式的序列化结构数据,被用在通信协议数据存储等等.

本文档是针对 Java,C++或 Python 开发者们想要在他们的应用程序中使用 Protocol Buffers.本概述介绍了 Protocol Buffers，并告诉怎么开始 - 然后是看教程或需要更深了解-Protocol Buffers 编码(protocol buffer encoding). API 引用文档(reference documentation)也同样提供了 3 种语言的,并且还提供了编写.proto 文件的语言(language)和样式(style) .

什么是 Protocol Buffers?

Protocol Buffers 是一个灵活,高效,和自动化机制的序列化结构的数据-联想到 XML,但是更小,更快,并且更简单.由自己来定义怎么使你的数据只被结构化一次,然后你可以使用生成的源代码很容易的用不同语言编写和阅读输入和输出到各种数据流结构化数据.你还可以更新你的数据结构而不用中断已部署并使用"旧"的数据格式编译的程序.

怎么工作?

你可以通过在.proto 文件定义 Protocol Buffers 消息类型来指定要如何结构序列化信息.每一个 Protocol Buffers 消息都是一个小的信息逻辑记录,包含了一系列的键值对.下面是一个.proto 文件的基本示例,其中定义了一个消息包含了关于一个人的信息:

```
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;
}
```

你还能看到,消息格式是非常简单的- 每一个小时类型都有一个或多个唯一的成员字段,并且每个字段都有一个名字和一个值类型,值类型可以是数字(整形或浮点型),布尔型,字符串型,原生字节型,甚至是其他(上面示例中所示)Protocol Buffers 消息类型.你能分级的结构化你的数据.你还能指定可选字段,必选字段,和重负字段.在 Protocol Buffers 语言指南中(Protocol Buffer Language Guide)找到更多关于编写.proto 文件的信息.

一旦定义了消息,就可以运行 Protocol Buffers 编译器根据.proto 文件来生成你的应用所用语言的数据访问类.这些类为每个字段(如 query() 和 set_query())提供了简单的访问器,以及从/到原始字节来序列化/解析整体结构的方法,举例来说,如果选择 C++语言,运行编译器来编译以上示例代码将会生产一个类名叫 Person.然后可以使用这个类来填充,序列化和获取 Person 的 Protocol Buffers 消息.代码示例如下:

```
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);
```

接下来,你能阅读你的消息在:

```
fstream input("myfile", ios::in | ios::binary);
Person person;
person.ParseFromIstream(&input);
```

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

你能添加新字段到你的消息格式中而不需要中断向后兼容(backwards-compatibility);旧包在解析的时候只会简单忽略新字段.所以如果你的通信协议使用的 Protocol Buffers 的数据格式,那么你能扩展协议而不需要担心中断已存在的代码.

在 API 引用部分(API Reference section)能找到使用生成 Protocol Buffers 代码的完整的引用,在 Protocol Buffers 编码(Protocol Buffer Encoding)中找到更多关于怎么使用编码的 Protocol Buffers 消息.

为什么不就用 XML?

Protocol Buffers 比 XML 在序列化结构的数据上拥有很多优点,Protocol Buffers 包含:

- 更加简单
- 更小 3 到 10 倍
- 更快 20 到 100 倍
- 更加的明确
- 生成的数据访问类更易于使用在编程上

举例,如果你希望得到一个含有 name 和 email 的 person 的模型,用 XML 的方式,如下:

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

而相应的 Protocol Buffers 消息则是(Protocol Buffers 文本格式(text format)):

```
# Textual representation of a protocol buffer.
# This is *not* the binary format used on the wire.
person {
  name: "John Doe"
  email: "jdoe@example.com"
```

```
}
```

当这个消息被编码成 Protocol Buffers 的二进制格式(binary format)(上面的文本格式只是一个便于阅读的以便能够调试和编辑的展示),这大概是 28 字节长并需要大概 100-200 纳秒来解析.然而 XML 版本的则至少是 69 字节长如果去掉空格的话,并且还需要花费大概 5000-10000 纳秒来解析他.

还有,操作 Protocol Buffers 更加简单:

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

而用 XML 就必须像这样:

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

然而,Protocol Buffers 也不总是比 XML 好-例如,使用 Protocol Buffers 不适合来模型化基于文本的标记语言的文档(如:HTML),因为不能轻易的使用文字交错结构.而相应的 XML 是可阅读可编辑的;但 Protocol Buffers 不是,至少在他的原生格式文件不是.XML 在一定程度上也不是,比如自描述文件.Protocol Buffers 只在拥有消息定义(.proto 文件)的时候才是有意义的.

听起来确实一种解决方案!那么我怎么入门呢?

下载类包 – 这包含了完整的针对 Java,Python,C++的 Protocol Buffers 编译器的源代码,以及 I/O 类和测试类.按照 README 中的说明来编译和按照你的编译器.

一旦所有的都设置好了,可以试着按照教程选择你所用的语言-他将会指引你用 Protocol Buffers 来创建一个简单的应用程序.

一点点关于 Protocol Buffers 的历史

Protocol Buffers 在最开始是 Google 设计的用来解决索引服务器的请求/响应协议.在此之前的 Protocol Buffers,请求和响应都有一个格式,是用的手动编组/解组请求和响应,并支持了该协议的一个版本数.这解决了一些非常难看的代码,如下:

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

显示化格式的协议也使新协议版本的推出复杂化,因为开发者们必须确保所有的服务器在原始请求者和实际处理请求的服务器在切准备换到新协议之前能够认识新协议.

Protocol Buffers 就是被设计成能够解决这些问题的:

- 新的字段能很容易的被引用进来,并且中间服务器不需要检查数据,只需要简单的解析他并且传递这些数据而不需要知道所有的字段.
- 格式不只是自我描述,并且还能通过不同的语言来处理(C++, Java, 等.)

但是,用户仍只需要手写解析代码.

本系统升级时,还需要一些其他的特性:

- 自动生成序列化和反序列化代码而不需要手工解析.
- 除了被用做即时的 RPC(远程过程调用)请求,人们开始使用 Protocol Buffers 为数据持久存储来作为一个方便的自描述格式 (例如,在大表中(Bigtable)).
- 服务器端 RPC 接口开始被声明成协议文件的一部分, 用户可以重写服务器的接口的实现用来使编译器生成根类(stub classes).

Protocol Buffers 现在是 Google 的数据*通用语言*(*lingua franca)* – 在编写时,总共有 48.162 种不同的消息类型被定义在 Google 代码树的 12,183 个.proto 文件中.同时使用了 RPC 系统和用来持久存储数据的各种存储系统中.