

Protocol Buffers 介绍

写在前面——有不对的地方，请大家批评指正，我会继续努力提高自己。如果转载请注明出处，谢谢大家支持——Forward。

基本概念

Protocol Buffers（以下简称 PB）是一种独立于语言、独立于开发平台、可扩展的序列化数据结构框架，它常常被用在通信、数据序列化保存等方面。

PB 是一种敏捷、高效、自动化的数据结构框架。同 XML 比较，比 XML 更小、更快、更简单。你一旦定义了期望的数据结构，就可以根据定义生成特定的源码，从而轻而易举地对你的数据进行读写操作，你甚至可以在不修改原来的程序源码的情况下，更新自己的之前定义的数据结构。

PB 与其他协议的比较

语言：

PB 与 Json 都是跨语言的，PB 支持 C++、Java、Python 等语言；

Json 支持的语言比 PB 更多，类似 Json 结构；

XML 结构支持的语言也是要比 PB 更多。

结构：

PB 发送与接受的数据属于一个完整的对象；

Json 发送与接受的是一个键值对的数据结构；

XML 结构从本质上讲也是一种键值对的数据结构。

数据保存：

PB 是二进制方式方式保存；

Json 属于文本保存；

XML 保存的是文本

（说明：文本的话更容易被破解，如果要在网络传输中使用，后面两种我们就得自己去实现加密算法来保证数据在网络中的安全性了，当然，并不是说 PB 的二进制就完全没法破解，只是较之其他两种结构来说安全一点）。

开发和扩展成本：

PB 只要维护一份 proto 文件就可以直接生成特定语言的类，保证了开发的高效，降低了维护成本；

Json 协议一般需要发送与接受方事先定义好结构，同时因为 Json 协议解析的时候是大小写敏感的，使得开发和维护成本较之 PB 略高；

XML 结构一般来说组织和解析都是需要开发人员自己去实现，开发成本和维护成本比

PB 和 Json 更高;

适用范围:

PB 和 Json 一般大都使用在客户端与服务器通信模块, 主要是因为数据的组织和解析较为简单;

而 XML 结构一般是作为配置文件来使用, 主要是因为方便属性的配置和修改。

通过下面的表格我们可以更清楚这三种结构之间的关系:

协议	语言	结构	数据保存	开发成本	数据大小	适用范围
PB	C++/Java/Python	对象	二进制文件	低	小	C/S 通信
Json	多种	键值对	文本	较低	一般	C/S 通信
XML	多种	键值对	文本	高	大	配置文件

表 1

PB 的使用过程

1、定义自己的数据结构

PB 允许我们定义自己期望的数据结构 Message, 它的定义是通过 .proto 文件来实现的。每个 Message 都是一个信息的逻辑记录块, 这个块包括了一系列键对值 (“键” 相当于生成对应类文件的属性名, “值” 表示这个属性的权限)

这里是作者自己写的一个 .proto 文件。

```
1 option java_package="com.ftd.message";
2 option java_outer_classname="DemoMessages";
3
4 message WorkerInfoList
5 {
6     repeated Worker workerInfo = 1;
7 }
8
9 message Worker
10 {
11     required string id = 1;
12     required string name = 2;
13     optional string email = 3;
14     optional string address = 4;
15 }
```

图 1

如图 1 前两行分别定义了 java 包名和类名, 在这个结构中, 作者定义了一个 WorkerInfoList, 这就相当于一个员工的信息列表, 每个员工信息定义在 Work 中, 其中包括了员工号 (id)、员工姓名 (name)、员工邮箱 (email)、员工地址 (address) 等信息。这样我们就完成了一个 PB 的模板 (.proto 文件)。

这里需要说明的是模板定义中使用的三种不同的属性 optional、required 以及 repeated。

Optional 属性说明该属性是可选的, 就是说在组织数据包时, 这个数据项可以是可有可

无的。

Required 属性说明该属性是必须有值的，不可为空。

Repeated 属性说明该属性是一个列表，我们可以把这种数据项理解为一个 **list** 来使用。

2、生成对应语言的源码；

有了模板这是第一步，接下来就要通过这个模板去生成对应语言下的类文件，google 为我们提供了生成工具（**protoc.exe**，很多地方都可以下载到），作者是从 <https://developers.google.com/protocol-buffers/> 这里下载了（作者在使用的时候，ProtocolBuf 最新版本是 2.4.1）。

下载完成打开之后我们可以看到如下图 2 的目录结构：

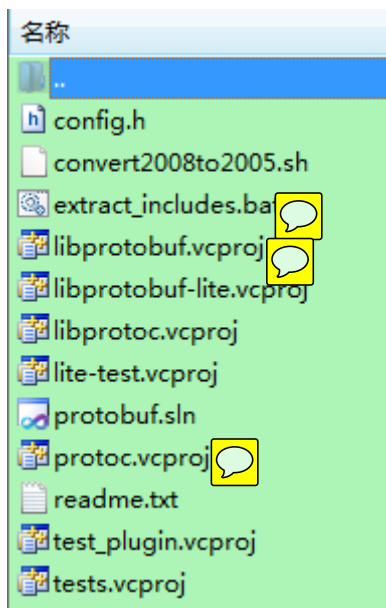


图 2

其中 protoc 工程编译之后就可以得到我们需要的 protoc.exe 工具。

在 cmd 中启动 protoc 并给定参数就可以生成我们需要的 C、Java、Python 的类文件。这里，作者简单对 protoc 生成对应文件的参数做一说明：

--proto_path 参数是指定我们的 proto 模板的目录（不要定义成这个模板文件，否则会报“Not find file”的错误）；

第二个参数取决于我们想生成的文件类型，如果想生成 C++ 的类参数名是“--cpp_out”；如果想生成 Java 的类参数名是“--java_out”；同理，如果想生成 Python 类，参数名就是“--python_out”。

接下来我们通过上面定义的模板来生成对应的类文件：

C++:

```
F:\PB>protoc --proto_path=F:\PB --cpp_out=F:\PB F:\PB\demo.proto
F:\PB>_
```

图 3

Java:

```
F:\PB>protoc --proto_path=F:\PB --java_out=F:\PB F:\PB\demo.proto
F:\PB>_
```

图 4

Python:

```
F:\PB>protoc --proto_path=F:\PB --python_out=F:\PB F:\PB\demo.proto
F:\PB>
```

图 5

生成结果如下所示:

C++类文件:



 demo.pb.cc	2013/1/27 15:52	C++ Source	23 KB
 demo.pb.h	2013/1/27 15:52	C/C++ Header	16 KB

图 6

Java 类文件:

F:\PB\com\ftd\message				
组织	包含到库中	共享	刻录	新建文件夹
★ 收藏夹	名称	修改日期	类型	大小
📁 下载	DemoMessages.java	2013/1/27 15:54	JAVA 文件	45 KB

图 7

Python 类文件:


 demo_pb2.py	2013/1/27 16:15	PY 文件
---	-----------------	-------

图 8

这里需要单独说明一下的是 Java 类文件生成的时候会出现多级目录, 这个前面已经说过, 每一层的目录是我们在模板中定义的。

3、使用

有了类文件, 我们就可以通过对应的 API 去组织和解析对应的数据了。这里简单说一下 C++中的类调用 (Java 和 Python 中的使用可以自己去看)。

```
// required string id = 1;
inline bool has_id() const;
inline void clear_id();
static const int kIdFieldNumber = 1;
inline const ::std::string& id() const;
inline void set_id(const ::std::string& value);
inline void set_id(const char* value);
inline void set_id(const char* value, size_t size);
inline ::std::string* mutable_id();
inline ::std::string* release_id();
```

图 9

如图 9 是上面的模板中 Worker 类的 id 调用接口。id 返回数据值, set_id 设置数据值这些接口基本都可以通过名字知道其含义。

```

// repeated .Worker workerInfo = 1;
inline int workerinfo_size() const;
inline void clear_workerinfo();
static const int kWorkerInfoFieldNumber = 1;
inline const ::Worker& workerinfo(int index) const;
inline ::Worker* mutable_workerinfo(int index);
inline ::Worker* add_workerinfo();
inline const ::google::protobuf::RepeatedPtrField< ::Worker >&
    workerinfo() const;
inline ::google::protobuf::RepeatedPtrField< ::Worker >*
    mutable_workerinfo();

```

图 10

图 10 中，是 WorkInfoList 的接口，因为这个类中包含了一个 Worker 的 repeated 数据项，前面说过我们可以把它当做一个 list 来使用，即在组织数据的时候通过 add_workerinfo 来添加一项员工数据，这个接口返回其指针，通过 mutable_workerinfo(int index)接口来获取下标为 index 的数据指针并进行操作（设置数据等）；在解析其参数的时候通过 workerinfo(int index)来获取下标为 index 的数据等等。具体的我们可以在真正使用的时候去进一步熟悉掌握。