Google Protocol Buffer (PB)简明入门

蒙融信息技术有限公司
黄志丹 2013/6/04

地址： 上海市杨浦区安波路 533 号 3#楼 7 层 01-02 室 邮编： 200433 电话： 021-65330277

目　　录

写在最前：本文以在 Windows 下 C++中使用 Protocol Buffer （以下简称 PB）为例展示如何安装和使用 PB。其他语言和平台也可做为参考。

一. 什么是 PB

PB是 Google 开发的一种开源数据交换方式。 特别适合于在 RPC间交换对象及数据结构。与其相似的应用有 XML、JASON、THRIFT等。

二. 为什么要用 PB

相对于其主要同类应用 XML, PB 的主要优势在于更小的数据 size（比 XML 小 3-10 倍）和更快的解析速度（比 XML 快 20-100 倍），同时在使用上也更简单。 PB 的劣势主要是可读性比较差，由于其生成的是二进制数据，可读性要远低于 XML 的明文格式，同时编辑也要借助代码来完成（ XML 可以直接编辑）。

三. 安装

1. 下载 PB
   至 https://code.google.com/p/protobuf/downloads/list 下载最新版 PB。建议最好是下载源代码（也提供了 Binary 下载），然后自己来编译。以下主要以 windows 下的编译做示例， Linux 下的可以参考自行完成。（Windows 下的下载包 protobuf-2.5.0.zip ）
2. 编译
   在 vsprojects 文件夹下可以找到解决方案 protobuf.sln（该方案是用 VS2008 生成的，低版本的 VS可以使用目录下提供的 Linux 脚本 convert2008to2005.sh 降级），用 VS 打开后直接启动编译即可。编译完成后会生成一个主要的 exe 和三个 .lib 库，分别是 protoc.exe, libprotobuf.lib, libprotobuf-lite.lib, libprotoc.lib. 其中 protoc.exe 是用于编译 .proto 文件的，其他三个库是用于编译序列化 /反序列化代码时使用的。 （稍后会讲到）

四. 使用

1. 构建对象描述文件（ .proto 文件）
   要使用 PB, 首先你要构建一个对象描述文件。见下例：

   person.proto

   // 包名。编译后就是名空间名称

   package mrPerson;

```
// 对象名。在该例中，对象名称为 Person
message Person
{
    // 字段的属性、类型、名称、 ID
    optional string name = 1;
    optional int32 age = 2;
}
```

上面是一个简单的对象描述文件。有点类似于伪代码。这里解释一下字段的属性、类型、名称及 ID。字段的属性分为三种 : required, optional, repeated. 分别表示该字段是必须的，可选的及重复的。具体含义如下：

Each field must be annotated with one of the following modifiers:

? required : a value for the field must be provided, otherwise the message will be considered "uninitialized". If libprotobuf is compiled in debug mode, serializing an uninitialized message will cause an assertion failure. In optimized builds, the check is skipped and the message will be written anyway. However, parsing an uninitialized message will always fail (by returning false from the parse method). Other than this, a required field behaves exactly like an optional field.

? optional : the field may or may not be set. If an optional field value isn't set, a default value is used. For simple types, you can specify your own default value, as we've done for the phone number type in the example. Otherwise, a system default is used: zero for numeric types, the empty string for strings, false for bools. For embedded messages, the default value is always the "default instance" or "prototype" of the message, which has none of its fields set. Calling the accessor to get the value of an optional (or required) field which has not been explicitly set always returns that field's default value.

? repeated : the field may be repeated any number of times (including zero). The order of the repeated values will be preserved in the protocol buffer. Think of repeated fields as dynamically sized arrays.

概括一下， required 属性的字段是必须要给初值的，否则解析时会返回 false ；optional 如果没给初值， PB 会使用默认初值； repeated 代表的是数组。注意： Google PB 官方文档中特意指明，从 Google 内部目前使用情况来看，一个比较好的实践方式是只使用 optional 和 repeated ，而不使用 required 。这样可以达到最好的向前兼容性。

字段类型列表如下：（主要用到的是 double,int64,int32,string,byte 等）

.proto Type	Notes	C++ Type	Java Type	Python Type [2]
double		double	double	float
float		float	float	float
int32	Uses variable-length encoding. Inefficient for encoding negative numbers – if your field is likely to have negative values, use sint32 instead.	int32	int	int
int64	Uses variable-length encoding. Inefficient for encoding	int64	long	int/long [3]

	negative numbers – if your field is likely to have negative values, use sint64 instead.			
uint32	Uses variable-length encoding.	uint32	int [1]	int/long [3]
uint64	Uses variable-length encoding.	uint64	long [1]	int/long [3]
sint32	Uses variable-length encoding. Signed int value. These more efficiently encode negative numbers than regular int32s.	int32	int	int
sint64	Uses variable-length encoding. Signed int value. These more efficiently encode negative numbers than regular int64s.	int64	long	int/long [3]
fixed32	Always four bytes. More efficient than uint32 if values are often greater than 2^{28}.	uint32	int [1]	int
fixed64	Always eight bytes. More efficient than uint64 if values are often greater than 2^{56}.	uint64	long [1]	int/long [3]
sfixed32	Always four bytes.	int32	int	int
sfixed64	Always eight bytes.	int64	long	int/long [3]
bool		bool	boolean	boolean
string	A string must always contain UTF-8 encoded or 7-bit ASCII text.	string	String	str/unicode [4]
bytes	May contain any arbitrary sequence of bytes.	string	ByteString	str

字段 ID 的主要作用是做向前兼容。 .proto 文件扩充后，可以通过定义不重复的 id，实现向前兼容性。

2. 将对象描述文件编译成代码
   使用如下命令行“ protoc -I=$SRC_DIR--cpp_out=$DST_DIR $SRC_DIR/person.proto ”来编译 .proto 文件。注意：$SRC_DIR表示 .proto 文件夹路径； $DST_DIR表示输出的 .cc 和 .h 文件夹路径； ”号表示使用当前路径。编译完成后， 会在 $DST_DIR下生成两个文件“ person.pb.h ”和“ person.pb.cc ”。
3. 使用 PB 序列化和反序列化对象
   代码说话：

```
#include <iostream>
#include <fstream>
#include <string>
#include "person.pb.h"
using namespace std;
using namespace mrPerson;

#pragma comment(lib, "libprotobuf.lib")
#pragma comment(lib, "libprotoc.lib")
#pragma comment(lib, "libprotobuf-lite.lib")
```

```
// 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.
int main(int argc, char* argv[]) {
  // Verify that the version of the library that we linked against is
  // compatible with the version of the headers we compiled against.
    GOOGLE_PROTOBUF_VERIFY_VERSION;

  //     定义 Person 对象
  Person person;
  //     设置字段
  person.set_name("huangzhidan");
  person.set_age(17);

  // ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
  //     以下展示使用文件做为载体序列化和反序列化对象
  //     将对象写入文件
  fstream output("test_person.dat", ios::out | ios::trunc | ios::binary);
  if(!person.SerializeToOstream(&output))
  {
    cerr << "Failed to write person." << endl;
    return -1;
  }
  output.flush();

  //     将文件反序列化成对象
  Person person_out;
  fstream input("test_person.dat", ios::in | ios::binary);
  if(!person_out.ParseFromIstream(&input))
  {
    cerr << "Failed to read person." << endl;
    return -1;
  }

  //     验证
  string name = person_out.name();
  int age = person_out.age();

  // ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
  //     以下展示使用流数据做为载体序列化和反序列化对象
  string streamData;
  //     可以使用    SerializeToString 接口将对象序列化成流数据，然后通过        Socket
通信
```

```
        person.SerializeToString(&streamData);

        Person person_stream;
        person_stream.ParseFromString(streamData);

        //    验证
        name = person_out.name();
        age = person_out.age();

        // Optional:  Delete all global objects allocated by libprotobuf.
        google::protobuf::ShutdownProtobufLibrary();

        return 0;
    }
```

以上代码演示了在 PB 中使用文件和流数据为载体来序列化和反序列化对象（库和头文件路径请自行配置）。可以看到，使用 PB 承载对象还是比较容易的，代码编写也很简单。

下面是使用 PB来承载一个较复杂的嵌套对象过程：

(1) addressbook.proto

```
// See README.txt for information and build instructions.

package tutorial;

option java_package = "com.example.tutorial";
option java_outer_classname = "AddressBookProtos";

message Person {
  required string name = 1;
  required int32 id = 2;                // Unique ID number for this person.
  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;
```

这个定义在message类的内部

这个定义在message类的内部

```
}

// Our address book file is just one of these.
message AddressBook {
  repeated Person person = 1;
}
```

(2) 参考以上编译成 addressbook.pb.h 和 addressbook.pb.cc

(3) Writing A Message

```
#include <iostream>
#include <fstream>
#include <string>
#include "addressbook.pb.h"
using namespace std;

// This function fills in a Person message based on user input.
void PromptForAddress(tutorial::Person* person) {
  cout << "Enter person ID number: ";
  int id;
  cin >> id;
  person->set_id(id);
  cin.ignore(256, '\n');

  cout << "Enter name: ";
  getline(cin, *person->mutable_name());

  cout << "Enter email address (blank for none): ";
  string email;
  getline(cin, email);
  if (!email.empty()) {
    person->set_email(email);
  }

  while (true) {
    cout << "Enter a phone number (or leave blank to finish): ";
    string number;
    getline(cin, number);
    if (number.empty()) {
      break;
    }

    tutorial::Person::PhoneNumber* phone_number = person->add_phone();
    phone_number->set_number(number);
```

```
    cout << "Is this a mobile, home, or work phone? ";
    string type;
    getline(cin, type);
    if (type == "mobile") {
      phone_number->set_type(tutorial::Person::MOBILE);
    } else if (type == "home") {
      phone_number->set_type(tutorial::Person::HOME);
    } else if (type == "work") {
      phone_number->set_type(tutorial::Person::WORK);
    } else {
      cout << "Unknown phone type.          Using default." << endl;
    }
  }
}

// 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.
int main(int argc, char* argv[]) {
  // Verify that the version of the library that we linked against is
  // compatible with the version of the headers we compiled against.
  GOOGLE_PROTOBUF_VERIFY_VERSION;

  if (argc != 2) {
    cerr << "Usage:       " << argv[0] << " ADDRESS_BOOK_FILE" << endl;
    return -1;
  }

  tutorial::AddressBook address_book;

  {
    // Read the existing address book.
    fstream input(argv[1], ios::in | ios::binary);
    if (!input) {
      cout << argv[1] << ": File not found.          Creating a new file." << endl;
    } else if (!address_book.ParseFromIstream(&input)) {
      cerr << "Failed to parse address book." << endl;
      return -1;
    }
  }

  // Add an address.
```

```
PromptForAddress(address_book.add_person());

{
  // Write the new address book back to disk.
  fstream output(argv[1], ios::out | ios::trunc | ios::binary);
  if (!address_book.SerializeToOstream(&output)) {
    cerr << "Failed to write address book." << endl;
    return -1;
  }

  // Optional:  Delete all global objects allocated by libprotobuf.
  google::protobuf::ShutdownProtobufLibrary();

  return 0;
}
```

(4) Reading A Message

```
#include <iostream>
#include <fstream>
#include <string>
#include "addressbook.pb.h"
using namespace std;

// Iterates though all people in the AddressBook and prints info about them.
void ListPeople(const tutorial::AddressBook& address_book) {
  for (int i = 0; i < address_book.person_size(); i++) {
    const tutorial::Person& person = address_book.person(i);

    cout << "Person ID: " << person.id() << endl;
    cout << "  Name: " << person.name() << endl;
    if (person.has_email()) {
      cout << "  E-mail address: " << person.email() << endl;
        }

    for (int j = 0; j < person.phone_size(); j++) {
      const tutorial::Person::PhoneNumber& phone_number =
person.phone(j);

          switch (phone_number.type()) {
        case tutorial::Person::MOBILE:
          cout << "  Mobile phone #: ";
          break;
        case tutorial::Person::HOME:
```

```
        cout << "  Home phone #: ";
        break;
      case tutorial::Person::WORK:
        cout << "  Work phone #: ";
        break;
    }
    cout << phone_number.number() << endl;
  }
}

// Main function:  Reads the entire address book from a file and prints all
//   the information inside.
int main(int argc, char* argv[]) {
  // Verify that the version of the library that we linked against is
  // compatible with the version of the headers we compiled against.
  GOOGLE_PROTOBUF_VERIFY_VERSION;

  if (argc != 2) {
    cerr << "Usage:  " << argv[0] << " ADDRESS_BOOK_FILE" << endl;
    return -1;
  }

  tutorial::AddressBook address_book;

  {
    // Read the existing address book.
    fstream input(argv[1], ios::in | ios::binary);
    if (!address_book.ParseFromIstream(&input)) {
      cerr << "Failed to parse address book." << endl;
      return -1;
    }
  }

  ListPeople(address_book);

  // Optional:  Delete all global objects allocated by libprotobuf.
  google::protobuf::ShutdownProtobufLibrary();

  return 0;
}
```

注意以上用例中，枚举类型和嵌套结构的用法。

以上只是展示了　PB的基础用法，其实　PB还有许多高级用法：

Advanced Usage

Protocol buffers have uses that go beyond simple accessors and serialization. Be sure to explore the C++ API reference to see what else you can do with them.

One key feature provided by protocol message classes is reflection. You can iterate over the fields of a message and manipulate their values without writing your code against any specific message type. One very useful way to use reflection is for converting protocol messages to and from other encodings, such as XML or JSON. A more advanced use of reflection might be to find differences between two messages of the same type, or to develop a sort of "regular expressions for protocol messages" in which you can write expressions that match certain message contents.

If you use your imagination, it's possible to apply Protocol Buffers to a much wider range of problems than you might initially expect!

Reflection is provided by the Message::Reflection interface.

3. 如何在一个 .proto 中调用另外一个 .proto 中的结构体
示例如下：

book.proto:

```
package mybook;

message CBook
{
    optional string name = 1;
    optional double price = 2;
}
```

store.proto:

```
package mystore;

import "book.proto"; //　引用 book.proto

message CStore
{
    optional string name = 1;
    repeated mybook.CBook book = 2; //　使用 book.proto 中定义的结构体 : CBook. 注意加上名空间 mybook
}
```

五.　附录

(1) Pb 类型对照表

.proto Type	Notes	C++ Type	Java Type	Python Type [2]
double		double	double	float

float		float	float	float
int32	Uses variable-length encoding. Inefficient for encoding negative numbers – if your field is likely to have negative values, use sint32 instead.	int32	int	int
int64	Uses variable-length encoding. Inefficient for encoding negative numbers – if your field is likely to have negative values, use sint64 instead.	int64	long	int/long [3]
uint32	Uses variable-length encoding.	uint32	int [1]	int/long [3]
uint64	Uses variable-length encoding.	uint64	long [1]	int/long [3]
sint32	Uses variable-length encoding. Signed int value. These more efficiently encode negative numbers than regular int32s.	int32	int	int
sint64	Uses variable-length encoding. Signed int value. These more efficiently encode negative numbers than regular int64s.	int64	long	int/long [3]
fixed32	Always four bytes. More efficient than uint32 if values are often greater than 2^{28}.	uint32	int [1]	int
fixed64	Always eight bytes. More efficient than uint64 if values are often greater than 2^{56}.	uint64	long [1]	int/long [3]
sfixed32	Always four bytes.	int32	int	int
sfixed64	Always eight bytes.	int64	long	int/long [3]
bool		bool	boolean	boolean
string	A string must always contain UTF-8 encoded or 7-bit ASCII text.	string	String	str/unicode [4]
bytes	May contain any arbitrary sequence of bytes.	string	ByteString	str

六. Reference：

(2) Tutorials:
https://developers.google.com/protocol-buffers/docs/tutorials
(3) Protocol Buffer Basics: C++
https://developers.google.com/protocol-buffers/docs/cpptutorial?hl=zh-CN
(4) Language Guide:
https://developers.google.com/protocol-buffers/docs/proto