Protocol Buffer 소개 및 사용법

출처: http://ourcstory.tistory.com/47

구글 프로토콜 버퍼 (Google Protocol Buffer) 란?

Google에서 개발한 protocol buffer의 특징은 아래와 같습니다.

- language-neutral
- platform-neutral
- extensible mechanism for serializing structed data

쉽게 말하면 XML, Json등 과 같이 데이터를 저장하는 하나의 포맷이라고 할 수 있습니다. 하지만 가볍고, 빠르고, 그리고 사용하기에 쉽습니다. 사용법은 최초에 우리가 사용하고자 하는 데이터를 구조화하고, 사용하는 언어의 코드로 컴파일링을 하면 자동으로 코드가 생산됩니다. 자동으로 생성된 코드는 파일을 쓰고/읽는데 사용하면 됩니다. 구글 프로토콜 버퍼는 Java, Python, 그리고 C++을 지원하고 있습니다. proto3부터는 Go, JavaNono, Ruby, 그리고 C#까지 지원이 가능하다고 합니다.

Proto 다운받고, 설치하기

설치하기 위해서는 https://github.com/google/protobuf 에 접속하면 다운로드가 가능합니다. protocol compiler는 C++로 작성되어 있기 때문에 직접 설치가 가능합니다. 하지만 C++유져가 아니라면, pre-built binary를 통해서 설치가 가능합니다. pre-built binary 다운받기

Proto 포맷 정의하기

처음에는 사용하고자 하는 데이터들을 구조화 하는 작업이 필요합니다. 하나의 틀 이라고 생각하시면 됩니다. 이 정의된 틀을 통해 proto는 각 언어에 맞는 코드를 자동으로 생성을 해줍니다. 이렇게 생성된 코드는 쓰기/읽기에 사용이 가능합니다. 문법은 약간 C++, Java와 비슷한 형태를 띄우고 있습니다.

예를들어서 addressbook.proto를 생성합니다.

```
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;
}
```

코드에 대해서 간단하게 설명을 드리면, .proto의 시작은 package로 정의 해야 합니다. 그 이유는 다른 프로젝트들간에 충돌을 막기 위해서 입니다. 위 예제는 java이기 때문에 java_package, java_outer_classname을 정의해 줍니다. 결과적으로 proto로 컴파일링을 하게 되면 /PROJECT_ROOT/com/example/tutorial/AddressBookProtos.java의 파일이 생성됩니다.

변수 옆에 붙는 숫자는 "tag"로 binary encoding할때 사용되는 필드입니다. (중복이 되면 안됩니다.) 변수 앞에 붙는 필드 required, optional, repeated

- required는 항상 값이 요구되는 값으로, 없으면 에러가 발생합니다.
- optional은 값이 있어도 되고, 없어도 되는 값을 말합니다. 값을 넣지 않으면 default값으로 해당 filed의 default값이 들어가게 됩니다.
- repeated는 배열이라고 생각하시면 됩니다.

위 예제를 풀어 설명하면 AddressBook에는 여러명의 Person이 저장될 수 있고, Person에는 여러개의 PhoneNumber가 들어갈 수 있는 데이터 구조입니다. 또한 PhoneNumber에는 number라는 string이 항상 필요하고, PythoneType의 값은 설정을 안해도 됩니다. 설정을 안하면 default로 HOME이라는 1이 들어갑니다.

Proto 컴파일링

이렇게 만들어진 .proto는 설치된 protoc를 통해서 컴파일링을 하고 결과적으로 com/example/tutorial/의 경로에 AddressBookProtos.java의 파일이 생성이 됩니다.

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

만약 여기서 python의 코드로 생성을 하려고 한다면 아래와 같이 컴파일링하고 결과적으로 .py의 파일이 생성이 됩니다.

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

생성된 코드를 사용해 쓰기/읽기 하는 방법 (자동으로 생성된 코드는 수정을 하지 않습니다.) 생성된 AddressBookProtos.java의 파일을 보면 AddressBookProtos라는 class가 생성이 되어있습니다. 각 class마다 자신의 Builder의 클래스를 갖습니다. 이 클래스는 instance를 생성하는데 사용하게 됩니다. message의 경우에는 오직 getters만 갖고 있고, builders의 경우에는 getters와 setters를 갖고 있습니다.

Person

```
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);
```

Person.Builder

```
// 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();
```

PhoneType은 Person의 nested로 자동 생성 됩니다.

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

Builder vs Message

message는 protocol buffer의 compiler에 의해 immutable하게 생성되는 클래스 입니다. 즉, 한번 객체가 생성이 되면 수정이 불가능 합니다. (java의 String처럼) message를 구성하기 위해서는 첫번째로 builder를 만들어야 합니다. 그 다음 값들을 set, add를 한뒤에 build()의 함수를 통해 만들 수 있습니다.

```
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();
```

Message의 Methods

- isInitialized() : required 필드가 모두 세팅이 되었는지, (만약에 required 필드가 하나라도 누락되면 에러를 발생시킵니다.)
- toString() : debugging의 사용에 유용합니다. print문으로 값을 확인하고자 할때 사람이 일을 수 있는 표현의 메세지를 리턴합니다.
- mergeFrom(Message other) : builder에게 있는 메소드로, 다른 message와 통합하는 것을 말합니다. (singular filed는 overwriting 되고, repeats는 concatenating됩니다)
- clear() : builder에게만 있고 모든 필드르 empty state로 합니다.

Parsing and Serialization

rotocol buffer 클래스는 message를 binary format으로 읽고 쓰는 메소드를 갖고 있습니다.

- byte[] toByteArray(): message를 serialize하고, byte array에 포함되어 있는 raw bytes를 리턴하는 메소드
- static Person parseFrom(byte[] data) : 주어진 byte array로 부터 message를 parses하는 메소드
- void writeTo(OutputStream output): message를 serialize하고, OutputStream으로 쓰는 메소드
- static Person parseFrom(InputStream input): InputStream으로 부터 메시지를 read, parse를 하는 메소드

Writing A Message

```
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();
  }
}
```

Reading A Message

```
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);
  }
}
```

[참고 사이트]
https://developers.google.com/protocol-buffers/

언어별 튜토리얼
https://developers.google.com/protocol-buffers/docs/tutorials

개발자 가이드 문서
https://developers.google.com/protocol-buffers/docs/overview

언어별 API 문서
https://developers.google.com/protocol-buffers/docs/reference/overview

예제 코드
https://github.com/google/protobuf/tree/master/examples

Protocol Buffer 로 C++ 과 C# 에서 데이터 읽기

출처 http://parkpd.egloos.com/4011040

프로토콜 버퍼에 대한 기본 설명은 자바워크님의 블로그를 참고하자.

Google Protocol Buffers 기본 사용법 by 자바워크
Google Protocol Buffers에서 Reflection 사용법 by 자바워크
Protocol Buffers를 패킷으로 활용해 보자 by 자바워크

cpp 에서 프로토콜 버퍼를 사용하려면 다음과 같이 한다.

protobuf 프로젝트를 디버그, 릴리즈별로 따로 빌드한다.
addressbook.proto 와 protoc.exe 파일이 들어있는 폴더에서 ProtocolGen.bat 을 실행해 addressbook.pb.h, addressbook.pb.cc 파일을 만든 뒤 프로젝트에 추가한다. addressbook.proto 파일 내용은 다음과 같다.

```
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;
}
```

ProtocolGen.bat 파일 내용은 다음과 같다.

```
set DST_DIR=D:\work\ProtocolBuffer\Test\PbTest4
set SRC_DIR=D:\work\ProtocolBuffer\Test\PbTest4
protoc -I=%SRC_DIR% --cpp_out=%DST_DIR% %SRC_DIR%/addressbook.proto
```

protobuf-2.5.0\protobuf-2.5.0\src 폴더를 include 에 추가하고, protobuf 를 빌드해서 나온 .lib 파일들(libprotobuf.lib)을 모아놓은 폴더를 lib 에 추가하고 해당 lib 를 Add 해 준다.(디버그 용을 위해 libprotobufd.lib 도 만들었다. link warning 을 막으려면 libprotobufd.pdb 도 생성해 준다.)

C++ 용 코드는 아래와 같다.

관련내용 : Google Protocol Buffers 기본 사용법 by 자바워크

```
#include "stdafx.h"
#include <iostream>
#include <fstream>
#include <string>
#include "addressbook.pb.h"
#include <google/protobuf/text_format.h>
#include <google/protobuf/io/zero_copy_stream_impl_lite.h>

using namespace std;
using namespace google;

int _tmain(int argc, _TCHAR* argv[])
{
    // Message 객체에 값 세팅
    tutorial::Person src_person;
    src_person.set_id(41);
    src_person.set_name("alice");
    src_person.set_email("alice@anydomain.com");
    tutorial::Person::PhoneNumber* phone0 = src_person.add_phone();
    phone0->set_number("123-0101");
    phone0->set_type(tutorial::Person_PhoneType_MOBILE);
    tutorial::Person::PhoneNumber* phone1 = src_person.add_phone();
    phone1->set_number("456-0202");
    phone1->set_type(tutorial::Person_PhoneType_HOME);

    // 미리 생성해야 하는 버퍼의 길이를 알아내어 버퍼할당
    int bufSize = src_person.ByteSize();
    char* outputBuf = new char[bufSize];
```

```
    // 버퍼에 직렬화
    protobuf::io::ArrayOutputStream os(outputBuf, bufSize);
    src_person.SerializeToZeroCopyStream(&os);

    // 버퍼에서 역직렬화
    protobuf::io::ArrayInputStream is(outputBuf, bufSize);
    tutorial::Person dst_person0;
    dst_person0.ParseFromZeroCopyStream(&is);

    // Message 객체에서 값 가져오기
    string name = dst_person0.name();
    int id = dst_person0.id();
    for (int i=0;i<dst_person0.phone_size();++i)
    {
        const tutorial::Person_PhoneNumber& phone = dst_person0.phone(i);
        tutorial::Person_PhoneType phone_type = phone.type();
        string phone_number = phone.number();
    }

    // 파일에 직렬화
    const char* test_filename = "person_in_cpp.txt";
    fstream ofs(test_filename, ios::out | ios::trunc | ios::binary);
    src_person.SerializeToOstream(&ofs);
    ofs.close();

    // 파일에서 역직렬화
    fstream ifs(test_filename, ios::in | ios::binary);
    tutorial::Person dst_person1;
    dst_person1.ParseFromIstream(&ifs);
    ifs.close();

    string textFormatStr;
    protobuf::TextFormat::PrintToString(src_person, &textFormatStr);
    printf("%s\n", textFormatStr.c_str());

    tutorial::Person dst_person2;
    protobuf::TextFormat::ParseFromString(textFormatStr, &dst_person2);

    // 메모리 해제
    delete [] outputBuf;
    outputBuf = NULL;
    protobuf::ShutdownProtobufLibrary();

    return 0;
}
```

C# 에서는 아래와 같은 방식으로 데이터를 읽는다.

먼저 Google's Protocol Buffers project, ported to C# 에서 필요한 파일을 다운로드 받는다.

ProtocolGen.bat 에는 `protogen addressbook.proto --proto_path=.` 와 같이 입력한다.

같은 폴더에 protoc.exe, ProtoGen.exe, Google.ProtocolBuffers.dll, Google.ProtocolBuffers.Serialization.dll 가 있어야 한다.

정상적으로 실행되었다면 Addressbook.cs 파일이 만들어졌을 것이다.

CSharp CRT 프로젝트를 만든 뒤 Addressbook.cs 파일을 추가하고, Google.ProtocolBuffers.dll, Google.ProtocolBuffers.Serialization.dll 을 references 에 추가한다. 코드는 다음과 같다.

```
using System;
using System.IO;
using tutorial;

namespace PbTest3
{
    class Program
    {
        static void Main(string[] args)
        {
            Person.Builder newPersonBuilder = Person.CreateBuilder();
            newPersonBuilder.SetId(41)
                .SetName("alice")
                .SetEmail("alice@anydomain.com");
            newPersonBuilder.AddPhone(
                Person.Types.PhoneNumber.CreateBuilder()
                    .SetNumber("123-0101")
                    .SetType(Person.Types.PhoneType.MOBILE));
            newPersonBuilder.AddPhone(
                Person.Types.PhoneNumber.CreateBuilder()
                    .SetNumber("456-0202")
                    .SetType(Person.Types.PhoneType.HOME));

            Person person = newPersonBuilder.Build();

            // 버퍼에 직렬화
            byte[] personByte = person.ToByteArray();

            // 버퍼에서 역직렬화
            Person person1 = Person.ParseFrom(personByte);

            // Message 객체에서 값 가져오기
            string name = person.Name;
            int id = person.Id;
            foreach (Person.Types.PhoneNumber pn in person.PhoneList)
            {
                Console.WriteLine(pn.Number);
            }

            // 파일에 직렬화
            using (FileStream fs = new FileStream("person_in_csharp.txt",
FileMode.OpenOrCreate))
            {
                person.WriteTo(fs);
            }

            // 파일에서 역직렬화
            Person person2;
            using (FileStream fs = new FileStream("person_in_cpp.txt",
FileMode.Open))
            {
```

```
                person2 = Person.ParseFrom(fs);
            }

            // 텍스트 형식으로 출력
            string textFormatStr = person2.ToString();
            Console.WriteLine(textFormatStr);

            // 문자열에서 parsing
            //Person person3 = Person.ParseFromString(textFormatStr);       // 없
는건가?
        }
    }
}
```

직렬화 방법, 구글의 Protocol Buffers

출처: http://aploit.egloos.com/5233561

직렬화란 메모리안의 구조적인 데이타를 IO를 통해 타 프로세스로 보내거나 저장하기 위해 연속된 bit로 만드는 과정이다. c의 구조체나 클래스의 인스턴스와 같이 구조적인 혹은 계층적인 데이타를 iO를 통해 내보내려면 한줄로 쭉 정렬해야 합니다. 이런 한 줄로 쭉 세우는 과정이 직렬화이고, 반대의 작업이 역직렬화이다. 외부의 프로세스와 통신을 하거나 데이타를 파일로 저장을 한다고 하려면 어떤 방식이건 간에 직렬화를 해야 한다. 소켓 통신에서 메시지의 전문 포멧을 정하는 것이나, xml로 변환하거나, java의 Serializable이나, c의 structure 메모리 매핑을 그대로 보내는 것들 전부가 직렬화 방법들이다. 각 방법마다 장단점이 있다. 속도, 다른 OS간 통신, 다른 언어간 통신, 직렬화된 데이타의 사이즈, 개발 생산성, 개발 편의성, 유지보수성 정도가 고려되는 항목이다. xml의 경우 언어에 관계없고, 프로세스가 동작하는 OS에도 관계없지만, 직렬화/역직렬화의 속도가 떨어지고 직렬화된 데이터 사이즈가 크다는 단점이 있다. 데이타 전문의 경우 속도는 빠르지만 직렬화/역직렬화 구현을 전부 손으로 해야하기 때문에 개발편의성이 좋지 않고, 유지보수가 쉽지 않고, 버그를 찾기가 어려운 단점이 있다. c structure의 메모리 매핑은 메모리의 내용을 그대로 사용하는 것인데, OS만 달라도 사용이 불가하고, 물론 타 언어간의 통신은 불가하다. 대신 별도의 직렬화작업이 없는 이유로 속도는 가장 빠르다.

Protocol Buffers는 구글에서 사용하는 직렬화 방법이다. 구글 내에 많은 시스템이 있고, 당연히 다양한 OS들이 있고, 다양한 언어가 사용되는 환경에서 서로 통신하기 위한 직렬화 방법으로 개발한 것을 공개한 것이다. 라이센스는 new BSD이다. c structure 정의와 비슷한 형태로 데이타 구조를 정의하고, 컴파일러를 사용하여 c++과 java의 소스코드를 자동생성한다. 어플이케이션에서는 이 코드를 사용하여 값을 설정하고 직렬화하고 조회한다. 다음과 같은 장점을 가지고 있다.

- 속도 빠르고(xml에 비해 10~100배)
    - 데이타 사이즈 작고(xml에 비해 3~10배)
        - 코드 더 간단하고
            - 개발 쉽고

Subtype.proto 파일에 다음과 같이 메시지를 정의하고

```
package Subtype;
message Root {
    optional sint64 id = 1;
    optional string name = 2;
    optional Some some = 3;
}
message Some {
    optional sint64 someId = 1;
    optional string someName = 2;
    optional Some some = 3;
}
```

콘솔에서 다음과 같이 컴파일 한다.

```
protoc -cpp_out=. Subtype.proto    // cpp 코드 생성
protoc -java_out=. Subtype.proto    // java 코드 생성
```

컴파일 하면 Subtype.pb.cc, Subtype.pb.h, Subtype.java 파일이 생성된다.

다음은 c++ 어플리케이션에서 생성된 코드를 사용하는 코드이다.

```
Subtype::Root* root = new Subtype::Root();    // 자동 생성된 코드에 정의된 클래스.

root->set_id(1);
root->set_name("Tom");

Subtype::Some* sub1 = root->mutable_some();
sub1->set_someid(2);
sub1->set_somename((xc8*)"Jerry");

Subtype::Some* sub2 = sub1->mutable_some();
sub2->set_someid(3);
sub2->set_somename((xc8*)"Brute");

// 인코딩된 메시지 사이즈를 얻는다. (인코딩은 하기 전인데도.)
int encodedSize = root->ByteSize();

// 인코딩된 데이터가 담길 버퍼
char* encoded = (xuc8*)malloc(encodedSize);

// 인코딩 하고
int rtn = root->SerializeToArray(encoded, encodedSize);
if(!rtn) { print(“encoding failed\n”); }
delete message; message = NULL;

// 디코딩해서 담을 빈 메시지를 하나 만들고
Subtype::Root* newRoot = new Subtype::Root();

// 디코딩
rtn = newRoot ->ParseFromArray(encoded, encodedSize);
if(!rtn) { printf(“decoding failed\n”); }

free(encoded);
```

```
printf("Root.id = %d\n", newRoot ->id());
printf("Root.name = %s\n", newRoot ->name());

printf("Root.Some.someId = %d\n", newRoot ->some().someid());
printf("Root.Some.someName = %s\n", newRoot ->some().somename());

printf("Root.Some.Some.someId = %d\n", newRoot ->some().some().someid());
printf("Root.Some.Some.someName = %s\n", newRoot ->some().some().somename());

delete newRoot ; newRoot = NULL;
```

다음은 java 어플리케이션에서의 사용 코드이다.

```
Subtype.Root.Builder root = Subtype.Root.newBuilder();    // 자동 생성된 코드에 정
의된 클래스

root.setId(1);
root.setName("Tom");

Some.Builder sub1 = Some.newBuilder();
sub1.setSomeId(2);
sub1.setSomeName("Jerry");

Some.Builder sub2 = Some.newBuilder();
sub2.setSomeId(3);
sub2.setSomeName("Brute");

sub1.setSome(sub2);

root.setSome(sub1);

// 인코딩된 데이터가 담길 버퍼
byte[] encoded = null;


// 인코딩 하고
try {
     encoded = root.build().toByteArray();
} catch(UninitializedMessageException e) {
    System.out.println(“encoding failed. e=“+e);
}


// 디코딩해서 담을 빈 메시지를 하나 만들고
Subtype.Root newRoot = null;

// 디코딩
try {
    newRoot = sample.Subtype.Message.Root.parseFrom(encoded);
} catch(InvalidProtocolBufferException e) {
    System.out.println("decoding failed. e="+e);
}


println("Root.id = "+newRoot .getId());
println("Root.name = "+newRoot .getName());
```

```
println("Root.Some.someId = "+newRoot .getSome().getSomeId());
println("Root.Some.someName = "+newRoot .getSome().getSomeName());

println("Root.Some.Some.someId = "+newRoot .getSome().getSome().getSomeId());
println("Root.Some.Some.someName = "+newRoot
.getSome().getSome().getSomeName());
```

기타 사항은 다음과 같다.

- 기본적으로 c++, java, python을 지원한다.
    - AddOn에 의해 다음언어가 가능하다. Action Script, c, c#, Java ME, Javascript, Objective C, Perl, Php, Ruby, Visual Basic, Clojure, Common Lisp, D, Erlang, Haskell, Mercury, R
        - 메시지 말고 RPC 인터페이스를 proto 파일에 정의하고 stub 코드가 자동생성된다.
            - AddOn에 의해 RPC 구현체의 자동생성이 가능하다.
                - 패킷캡쳐 프로그램인 wireshark의 플러그인 존재
                    - NetBeans IDE의 플러그인 존재
                        - 컴파일러의 소스로 제공. linux, Unix는 빌드필요. windows의 경우 실행파일 protoc.exe만 따로 제공.
                            - java의 경우 필요한 jar파일을 maven repository에서 다운받을 수 있다.
                                - 메시지의 필드는 태그값(optional sint64 id = 1; 정의의 1)에 의해 관리된다.
                                    - 필드의 값 설정의 횟수의 타입은 required, optional, repeated 세가지가 있다. required는 반드시 한번, optional은 0 또는 1회, repeated는 횟수 제한없다.
                                        - 필드 타입은 double, float, bool, string, bytes와 정수형 관련 int32, int64, uint32, uint64, sint32, sint64, fixed32, fixed64, sfixed32, sfixed64들이 있다.
                                            - java의 경우 생성될 코드의 패키지와 클래스 이름의 개별 지정이 가능하다.
                                                - c++의 경우 네임스페이스의 정의가 가능하다.
                                                    - nested 메시지 가능.
                                                        - 타 파일에 정의한 메시지를 import 가능.
                                                            - 컴파일 시 성능 최적화, 크기 최적화가 가능하고(메시지별) 일부 기능을 제외한 lite 버전이 가능하다.
                                                                - 필드의 메타데이타를 사용하기 위한 reflection,

descriptor 지원.

- 정의한 메시지가 수정된되더라도 하위 호환이 보장된다. 즉 모든 시스템을 업그레이드하지 않아도 된다. 이쪽 시스템에서 새로운 메시지로 보냈을 때 저쪽에서는 기존의 필드는 처리하고, 모르는 필드는 무시한다.
    - java에서는 각 메시지마다 read only의 Message와 read & write

의 Builder 두 개가 있다. immutable인 String과 같이 성능상의 이유인 듯.

성능

테스트 환경은 Linux 2.6.18 Ubuntu, Intel Xeon X5460 @ 3.16GHz, memory 8G, java 1.5였다.
싱글 thread로 테스트 하였으며, java 컴파일 시 비디버깅모드로 하였다.
Protocol Buffers의 버번은 2.3.0.

측정된 값은 인코딩된 데이타 사이즈, 인코딩 시간, 디코딩 시간이었으며, 인코딩 시간은 자동생성된 코드에 정의된 객체를 생성하고 직렬활를 하는 시간이며, 디코딩 시간은 역직렬활를 하여 객체를 받기 까지의 시간이다.

두가지 데이타가 사용되었으며, 이 중 짧은 데이타는 다음과 같다.(Json으로 표현)

```
{
    "result":       100,
    "vcID": 9,
    "srcCSAID":     1,
    "targetCSAID":  2,
    "css":  [{
        "srcCSID":      1,
        "newCSID":      2
     }],
    "legs": [{
        "srcCSID":      1,
        "newCSID":      2,
        "srcLegID":     1,
        "newLegID":     3
     }, {
        "srcCSID":      1,
```

```
        "newCSID":      2,
        "srcLegID":     2,
        "newLegID":     4
    }]
}
```

인코딩된 메시지 사이즈는 짧은 메시지가 34byte, 긴 메시지가 1234 byte였다.
c++은 초당 130만회, 11만회 직렬화와 역직렬화를 하였다.
java는 초당 87만회, 6만회 직렬화와 역직렬화를 하였다.

같은 조건에서 cJSON, Pxml2를 사용한 결과를 비교하면 Protocol Buffers가 100배, 10배 빠르다. 메지시 사이즈는 2배, 1.6배 짧다.

Reference

Protocol Buffers home
http://code.google.com/intl/ko-KR/apis/protocolbuffers/
Third-Party Add-ons for Protocol Buffers
http://code.google.com/p/protobuf/wiki/ThirdPartyAddOns
Protocol Buffers c++ API
http://code.google.com/intl/ko-KR/apis/protocolbuffers/docs/reference/cpp/index.html
Protocol Buffers Java API
http://code.google.com/intl/ko-KR/apis/protocolbuffers/docs/reference/java/index.html

---

Google Protocol Buffers에서 Reflection 사용법

출처 http://javawork.egloos.com/2720973

Protocol Buffers(이하 PB)에 대한 설명은 이전 포스트를 참고해주세요.

http://javawork.egloos.com/2720889

PB에서 Reflection을 사용하면 정의된 필드/값에 하나씩 접근할수 있습니다. 이런 방식으로 PB의 소스를 수정하지 않고 여러 기능을 추가할수 있습니다. 예를 들면 제가 하려고 하는, 직렬화된 PB 버퍼를 JSON 형식으로 변환하기, 같은 기능이죠.

아래 코드는 PB에서 제공하는 text format과 같은 형식의 문자열을 출력하는 예제입니다. .proto 파일은

이전 포스트

에서 사용한 데이터를 그대로 사용했습니다.

```
#include <fstream>
#include <string>
#include "addressbook.pb.h"
#include <google/protobuf/descriptor.h>


using namespace std;
using namespace google;
```

```
void PrintGroupField(const protobuf::Message& message, const
protobuf::FieldDescriptor* descriptor);
void PrintField(const protobuf::Message& message, const
protobuf::FieldDescriptor* descriptor);
void IteratingWithDescriptor(const protobuf::Message& root_message);


void PrintGroupField(const protobuf::Message& message, const
protobuf::FieldDescriptor* descriptor)
{
    const protobuf::Reflection* reflection = message.GetReflection();
    int repeated_fieldCount = reflection->FieldSize(message, descriptor);
    const protobuf::Descriptor* child_descriptor = descriptor->message_type();
    for(int j=0;j<repeated_fieldCount;++j)
    {
        printf("%s {\n", descriptor->name().c_str());
        const protobuf::Message& child_message = reflection-
>GetRepeatedMessage(message, descriptor, j);
        const protobuf::Reflection* child_reflection =
child_message.GetReflection();
        int child_fieldCount = child_descriptor->field_count();
        for(int k=0;k<child_fieldCount;++k)
        {
            const protobuf::FieldDescriptor* curChildFd = child_descriptor-
>field(k);
            printf("  ");
            PrintField(child_message, curChildFd);
        }
        printf("}\n");
    }
}


void PrintField(const protobuf::Message& message, const
protobuf::FieldDescriptor* descriptor)
{
    const protobuf::Reflection* reflection = message.GetReflection();
    switch(descriptor->type())
    {
    case protobuf::FieldDescriptor::TYPE_INT32:
        {
            printf("%s : %d", descriptor->name().c_str(), reflection-
>GetInt32(message, descriptor));
            break;
        }
    case protobuf::FieldDescriptor::TYPE_STRING:
        {
            printf("%s : %s", descriptor->name().c_str(), reflection-
>GetString(message, descriptor).c_str());
            break;
        }
    case protobuf::FieldDescriptor::TYPE_ENUM:
        {
            const protobuf::EnumValueDescriptor* enumValue = reflection-
>GetEnum(message, descriptor);
            printf("%s : %s", descriptor->name().c_str(), enumValue-
>name().c_str());
            break;
```

```
        }
    case protobuf::FieldDescriptor::TYPE_MESSAGE:
        {
            PrintGroupField(message, descriptor);
            break;
        }
    }
    printf("\n");
}


void IteratingWithDescriptor(const protobuf::Message& root_message)
{
    const protobuf::Descriptor* root_descriptor = root_message.GetDescriptor();
    int root_fieldCount = root_descriptor->field_count();
    for(int i=0;i<root_fieldCount;++i)
    {
        const protobuf::FieldDescriptor* curFd = root_descriptor->field(i);
        PrintField(root_message, curFd);
    }
}


int main(int argc, char* argv[])
{
    const char* test_filename = "person.txt";
    fstream ifs(test_filename, ios::in | ios::binary);
    tutorial::Person person_message;
    person_message.ParseFromIstream(&ifs);
    ifs.close();
    IteratingWithDescriptor(person_message);
    return 0;
}
```

예외처리도 안되어 있고, 소스 자체가 person 데이터 형식의 출력에 치우쳐있어서 다른 형식으로 직렬화된 버퍼는 바르게 출력하지 못할수 있습니다. Reflection의 사용법을 알려드리기 위한 예제이니 감안하고 봐주시기 바랍니다.