OVERVIEW

"Đầu tư vào tri thức mang lợi nhuận cao nhất"

Benjamin Franklin

I. Hạn chế của các phương thức truyền thống:

Từ khi máy tính xuất hiện, nhu cầu gửi-nhận thông tin qua lại giữa chúng cũng phát sinh. Tuy nhiên, máy tính thì chỉ hiểu được mã nhị phân (byte/bit) chứ không thể hiểu các kiểu dữ liệu mà con người có thể đọc được (human-readable). Do đó, bản thân người gửi và người nhận phải tự thỏa thuận với nhau về format của dãy nhị phân để serialize và deserialize cho chính xác.

Thời gian đầu, máy tính quá đắt đỏ, chỉ một vài cơ quan cao cấp mới đủ khả năng sử dụng nhưng sau này, các doanh nghiệp cũng bắt đầu vào cuộc, khái niệm mạng máy tính cũng ra đời. Kiểu format binary tự định nghĩa nhanh chóng trở nên lạc hậu. Lúc này nhu cầu về một chuẩn format chung, thuận tiện cho nhiều người dễ dàng trao đổi, hiểu được, bắt đầu xuất hiện. Đó là lúc XML ra đời.

Với việc chia sẻ chung một tài liệu XML, tất cả các bên có thể dễ dàng hiểu được và tự xây dựng các thư viện serialize và deserialize của riêng mình mà không phải phụ thuộc lẫn nhau như trước. Mặc dù vậy, bản thân XML lại thừa quá nhiều ký tự (nhất là các ký tự kết thúc) nên JSON với kiến trúc gọn hơn dần được dùng để thay thế đến tận nay. Ví dụ dưới đây cho thấy cùng một lượng thông tin, nhưng JSON chỉ tốn 29 ký tự thay vì 53 như XML.

```
<config>
  <key1>value1</key1>
  <key2>1000</key2>
</config>

{
 "key1":"value1",
 "key2":1000
}
```

Hình 1: So sánh kích thước JSON và XML

Mặc dù vậy, bản thân JSON vẫn có những nhược điểm cố hữu mà XML chưa khắc phục được. Trước hết, đó là chỉ gửi được text mà không gửi được binary. Để giải quyết vấn đề đó, thì ta có thể encode sang dạng khác (ví dụ: base64) hoặc dùng BSON để thay thế. Nhưng như vậy thì phải tốn thêm chi phí cho quá trình xử lý. Thứ hai, cũng vì là text, nên khi deserialize buộc phải duyệt trên từng ký tự, tốc độ xử lý chắc chắn sẽ rất hạn chế. Vấn đề thứ ba nữa đó là sự không thống nhất giữa các thư viện serialize/deserialize của bên gửi/bên nhận dẫn đến việc sai lệch dữ liệu. Điều này xảy ra với các hệ thống có nhiều thành phần tham gia (nhiều server, nhiều app client, nhiều doanh nghiệp hợp tác cùng thực hiện, ...).

Vấn đề nghiêm trọng nhất là tương thích ngược. Giả sử, lúc đầu thông tin chỉ gồm 3 field, nhưng do nhu cầu phát sinh (nâng cấp version, yêu cầu thuật toán mới, ...) mà phải tăng lên 4 field. Để giải quyết vấn đề này, cách duy nhất trước giờ luôn là phải có field version, bên nhận sẽ đọc version trước, sau đó rẽ nhánh để xử lý (một API không có field version sẽ bị xem là API không an toàn). Hệ quả gây ra là tình trạng Code Tệ (Bad Code) khi phải if else liên tục. Đồng thời, bản thân khối dữ liệu truyền tải luôn phải tốn thêm dung lượng cho field version, trong khi field này thực chất không phải là thông tin cần cho quá trình xử lý thuật toán. Điều này trở thành gánh nặng cho chi phí duy trì hệ thống mạng.

Hơn 10 năm trước, Google hiểu rằng để phát triển các hệ thống toàn cầu họ phải giải quyết bằng được các vấn đề trên. Và câu trả lời chính là Protobuf.

II. **Ưu điểm của Protobuf**:

Do bản thân vốn đã là bibary nên dữ liệu chứa protobuf có thể là bất cứ gì không bị giới hạn.

Protobuf được Google hỗ trợ trên gần hết các ngôn ngữ lập trình phổ biến hiện nay nên lập trình viên dễ dàng ứng dụng cho tất cả ứng dụng của mình mà không cần lo lắng về vấn đề serialize/deserialize.

Protobuf tổ chức dữ liệu theo hướng tối ưu tốc độ xử lý.

Bằng việc thiết kế các filed ở dạng optional, protobuf cho phép phát triển các ứng dụng nhanh chóng mà không vướng vấn đề tương thích ngược (sẽ trình bày ở sau).

Khác với JSON, dữ liệu của Protobuf được nén để giảm kích thước giúp giảm chi phí vận hành hệ thống.

III. **Tổng quan cách dùng protobuf**:

Tương tự XML, protobuf cần có một file quy định các thành phần tạo nên các message gọi là proto (cách hiện thực file proto sẽ đề cập đến ở phần sau).

Protobuf cung cấp một tool là protoc (có được khi build opensource của protobuf). Từ file .proto, protoc sẽ tạo ra các class tương ứng với từng ngôn ngữ.

- Các message sẽ trở thành các class.
- Các field của message sẽ trở thành các biến của class.
- Mỗi field sẽ có Data Type riêng, tùy vào ngôn ngữ mà sẽ được map vào kiểu dữ liệu cho biến tương ứng. (Bảng 2)
- Các enum sẽ trở thành các define.

Các class này cùng với các library (.lib, .dll. .a, .jar, ...) sẽ được import và sử dụng trong project.

COMPILE PROTOBUF

"Đôi khi câu hỏi phức tạp lại có câu trả lời đơn giản"

Theodor Seuss Geisel

"Phương pháp càng đơn giản, thực hiện càng ít sai"

Huỳnh Trọng Tiến

Để sử dụng protobuf trên java ta cần chuẩn bị trước hai thành phần sau: công cụ protoc để tạo ra các class protobuf từ file .proto và các lib .jar

I. **Nguồn download**:

Để có protoc ta nên build trực tiếp từ opensource. Ngoài ra, ta cũng có thể download từ internet nhưng như vậy sẽ không an toàn. Bởi lẽ, đó có thể là protoc của phiên bản trước, không còn phù hợp với phiên bản hiện tại.

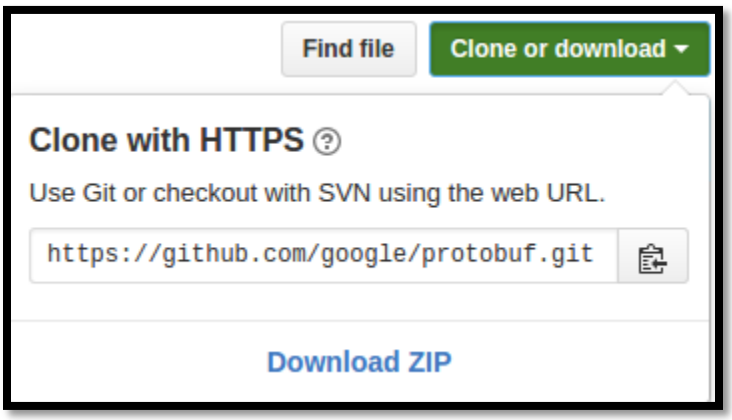
Cũng như các opensource khác ta nên vào đi từ link có trong trang hướng dẫn chính thức của Google, không nên đi từ các link khác bởi có thể đó là của các nhánh đang phát triển, không ổn định, trong trường hợp này là:

https://developers.google.com/protocol-buffers/docs/downloads

Hiện tại, sau khi vào đây, ta sẽ được dẫn đến github:

https://github.com/google/protobuf

Click chọn button **Clone or Download** để download file zip của source về



Hình 2: Download source code protobuf

Tương tự cho các file .jar, trên trang chính của Google sẽ hướng ta đến trang của maven:

https://mvnrepository.com/artifact/com.google.protobuf/protobuf-java

Lưu ý: để tránh việc version của protoc khác với các file .jar, ta nên build source protobuf trước, sau đó kiểm tra version của protoc rồi mới chọn version của .jar cho thích hợp

Command	**protoc –version**
Output	**libprotoc 2.6.1**

Như trên, version của protoc là **2.6.1** nên ta sẽ download **protobuf-java-2.6.1.jar**.

II. **Các bước build source protobuf**:

Bước 1: Unzip file source code vào thư mục nào đó. Ví dụ: **~/Dev/protobuf-master**

Bước 2: lần lượt thực hiện các lệnh sau và chờ đợi. Hoặc chạy file **build-protobuf.sh** (đính kèm). Thời gian build có thể tùy thuộc vào cấu hình máy.

sudo apt-get install autoconf automake libtool curl make g++ unzip
cd ~/Dev/protobuf-master/
./autogen.sh
./configure
make
make check
sudo make install
sudo ldconfig

Bước 3: Kiểm tra kết quả build bằng lệnh **protoc –version**

Sau khi đã build thành công opensource và download đúng .jar thì cơ bản ta đã xây dựng xong môi trường làm việc với protobuf trên Java.

FILE PROTO

"Hãy dùng chân vì chân mạnh hơn tay, hãy dùng đầu vì đầu mạnh hơn chân."

Tatsuya Hiruta

Lưu ý:

- Thông tin về phương thức tạo file proto rất nhiều, trong phạm vi bài viết không thể trình bày hết. Ở đây chỉ là những kiến thức người viết cho rằng cốt lõi, thường dùng nhất, đúc kết từ kinh nghiệm làm việc. Người đọc có thể tự tìm hiểu thêm thông tin đầy đủ qua: https://developers.google.com/protocol-buffers/docs/proto
- Không được phép sửa file output (.java) của protobuf. Việc chỉnh sửa chỉ được phép làm với file .proto. Mọi chỉnh sửa trên file .java output sẽ khiến việc build bị lỗi.

```
package protobuf;
option java_package = "com.vng.zalopay.protobuf";
option java_outer_classname = "ZPMsgProtos";

enum OSType {
        IOS = 1;
        ANDROID = 2;
        WINDOW_PHONE = 3;
}

message AuthMessage {
        required uint64 userid = 1;
        required string accesstoken = 2;
        optional uint32 ostype = 3;
}
```

Hình 3: Ví dụ file proto đơn giản

I. **Kiến trúc chung file proto**:

Trong mỗi file proto sẽ thường có 3 thành phần cơ bản: **header**, các **message** và các **enum**

1. Header:

Header thường dùng để chỉ định các thông tin chung như class name, version, package name. Trong đó quan trọng nhất là **java_package** và **java_outer_classname.**

java_package: chính là package mà ta sẽ import các file .java của protobuf vào project.

java_outer_classname: chính là Outer Class của tất cả các message có trong file proto.

2. Message:

Mỗi message (class), sẽ gồm có tên và các field (các biến của class). Mỗi field sẽ gồm 4 thành phần:

a. Rule: có 3 loại rule khác nhau

Rule	Giải thích
required	Bắt buộc phải có nếu không khi deserialize sẽ thất bại
optional	Có thể có hoặc không. Có thể gán giá trị default nếu cần
repeated	Tương tự mảng

Bảng 1: Danh sách các rule của field

b. Type: có thể là kiểu dữ liệu nguyên tố do protobuf quy định, cũng có thể là một message khác.

.proto Type	C++ Type	Java Type
int32	int32	int
uint32	uint32	int
int64	int64	long
uint64	uint64	long
float	float	float
double	double	double
string	string	String
bytes	string	ByteString
bool	bool	boolean

Bảng 2: Bảng map kiểu dữ liệu nguyên tố trong protobuf

c. Name: nếu message name là tên class thì field name chính là tên của các biến.

d. Tag: không có ý nghĩa về thuật toán nhưng rất quan trọng khi serialize dữ liệu. Sau khi đã chọn tag, các bản nâng cấp sau này của .proto không được thay đổi tag.

3. Enum: tương tự enum, define trong C/Java

II. **Các lưu ý khi tạo một file proto**:

Hạn chế tối đa các field required (trong protobuf 3 thì required đã được bỏ hẳn)

Khi bên gửi và bên nhận có những giá trị cần phải thông nhất với nhau, chẳng hạn như OS type, Status, ... Ta nên dùng enum để tránh sai lệch.

Mặc dù protobuf cho cho phép sử dụng default value cho các field optional nhưng chỉ nên dùng khi chắc chắn rằng sẽ không còn sự thay đổi nào nữa. Nhưng thực tế, do hiện trạng business tại VNG thay đổi liên tục, nên thuật toán cũng thường xuyên thay đổi, do đó không nên set default, để tránh việc update file proto quá nhiều gây phiền hà cho các bên liên quan.

Mặc dù ta có thể sử dụng hoàn toàn optional và repeated mà không cần reqired, nhưng nên có ít nhất một0filed nào đó bắt buộc phải có giá trị, điều này không có ý nghĩa thuật toán mà có ý nghĩa tránh sai sót cho người nhận. Bởi không có field nào hết thì protobuf sẽ rỗng, không có byte nào cả, lúc này người nhận sẽ bị lúng túng. Bởi không rõ việc nhận dữ liệu rỗng là do nhận bị lỗi, hay thực sự là không có field nào được gán giá trị.

III. **Cách nâng cấp file .proto để đảm bảo tính tương thích ngược**:

Bộ nguyên tắc tương thích ngược có rất nhiều. Người đọc có thể tham khảo chi tiết tại: http://www.slideshare.net/fabricioepa/protocol-buffers-44187777 . (slide 20 và 21). Ở đây người viết dựa vào kinh nghiệm thực tế để cho ra các nguyên tắc sau, đây không hẳn là bộ nguyên tắc tối ưu nhưng đảm bảo tối thiểu sai sót:

+ Tuyệt đối không được thay đổi tag của các field

+ Không được thay đổi các field có rule required

+ Không được xóa/thay đổi các field có sẵn từ trước.

+ Nếu có thêm thì chỉ được thêm dạng optional. Không cần lo lắng việc thừa field optional. Vì nếu không được gán giá trị, field optional sẽ không có bất kỳ ảnh hưởng nào cho protobuf.

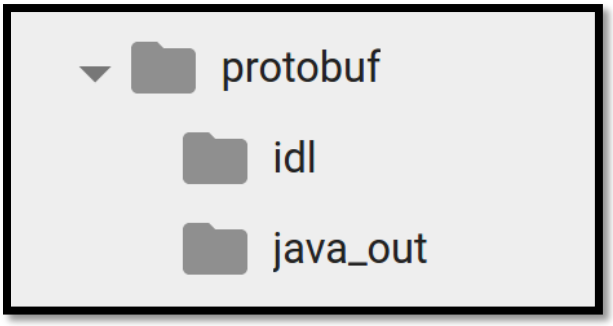
BUILD CÁC CLASS PROTOBUF

"Đừng xấu hổ vì không biết, hãy xấu hổ vì không học"

Khuyết danh

I. **Tổ chức thư mục build**:

Tại thư mục bất kỳ (giả sử ngay trong Home), tạo thư mục có tên protobuf, trong đó tạo hai thư mục con lần lượt có tên là **idl** và **java_out**. File proto **zalopay.proto** được đặt trong thư mục **idl**



Hình 4: Tổ chức thư mục build proto

Lưu ý: idl là từ khóa, không nên thay đổi. Ngược lại, java_out chỉ là thư mục để chứa các class output, ta có thể đặt các tên khác tùy ý, miễn là phải truyền tên vào câu lệnh ở phần sau.

II. **Build file .proto**:

Bước 1: chuyển đường dẫn thư mục làm việc vào prptobuf

cd ~/protobuf

Bước 2: dùng protoc để build các file .java

protoc -I=idl --java_out=java_out idl/zalopay.proto

SỬ DỤNG PROTOBUF TRONG MỘT PROJECT JAVA

"Sai lầm lớn nhất của của đầu bếp mới vào nghề là nghĩ rằng mình có thể tạo ra món mới ngay trong tháng đầu"

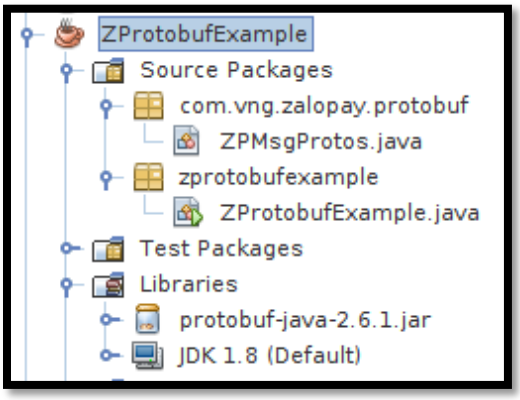
Huỳnh Trọng Tiến

Bước 1: Trước khi thực hiện ta cần chuẩn bị trước:

- protobuf-java-2.6.1.jar: download từ maven
- Các file .java: build từ proto

Bước 2: Tạo mới một project có package có tên trùng **java_package** trong header của file proto.

- Copy các file .java đã build được vào package được tạo ra
- Import protobuf-java-2.6.1.jar vào project



Hình 5: Tổ chức project sử dụng protobuf

Bước 3: Serialize và deserialize protobuf

```
try
{
    //Serialize
    AuthMessage msgEncode = AuthMessage.newBuilder()
                                       .setUserid(9999)
                                       .setAccesstoken("abcdefghk")
                                       .setOstype(OSType.IOS_VALUE)
                                       .build();

    //Deserialize
    AuthMessage msgDecode = AuthMessage.parseFrom(msgEncode.toByteArray());

    System.out.println("User ID: " + String.valueOf(msgDecode.getUserid()));
    System.out.println("AccessToken: " + msgDecode.getAccesstoken());
    if(msgDecode.hasOstype())
        System.out.println("OSType: " + String.valueOf(msgDecode.getOstype()));
    else
        System.out.println("OSType: not exist");
}
catch(Exception ex)
{
    System.err.println("Process protobuf failed. Cause: " + ex.getMessage());
    return;
}
```

Hình 6: Ví dụ Serialize/Deserialize protobuf

Để Serialize protobuf, ta dùng **Builder** của chính protobuf đó. Ngược lại, để Deserialize ta dùng phương thức static **parseFrom**.

Lưu ý ở đây có phương thức **toByteArray**, phương thức này trả về byte[] cũng chính là khối binary mà ta sẽ gửi nhận qua mạng.

Như vậy, ta sẽ dễ dàng hình dung được quy trình xử lý của protobuf như sau:

- Bên gửi dùng Builder để tạo ra object (msgEncode) của class AuthMessage.
- Sử dụng phương thức toByteArray của msgEncode để Serialize data và có được khối binary (**X**).
- X được gửi qua mạng cho Bên nhận.
- Bên nhận sau khi nhận được X, thì dùng phương thức parseFrom với X là param đầu vào để deserialize trở lại object AuthMessage mới (msgDecode).
- Lưu ý:
- + Luôn try/catch khi làm việc với protobuf
- + Trước khi sử dụng các field optional thì luôn phải dùng hàm **has_** trước.

"**Cuộc sống vốn đơn giản, đừng cố làm nó trở nên phức tạp**"