**protobuf使用**

屈雄烈 2015-10

# 目录

[目录 1](#_Toc431384603)

[1. protocol-buffers介绍 2](#_Toc431384604)

[1.1. protocol-buffers是什么? 2](#_Toc431384605)

[1.2. protocol-buffers如何工作? 2](#_Toc431384606)

[1.3. 为什么不使用xml? 2](#_Toc431384607)

[2. protobuf2 3](#_Toc431384608)

[3. protobuf3 (Language Guide) 4](#_Toc431384609)

[3.1. Defining A Message Type 定义消息类型 4](#_Toc431384610)

[3.1.1. Specifying Field Types 5](#_Toc431384611)

[3.1.2. Assigning Tags 5](#_Toc431384612)

[3.1.3. Specifying Field Rules 6](#_Toc431384613)

[3.1.4. Adding More Message Types 6](#_Toc431384614)

[3.1.5. Adding Comments 7](#_Toc431384615)

[3.1.6. Reserved Fields 保留字段 7](#_Toc431384616)

[3.1.7. What's Generated From Your .proto? 8](#_Toc431384617)

[3.2. Scalar Value Types 9](#_Toc431384618)

[3.3. Default Values 默认值 9](#_Toc431384619)

[3.4. Enumerations 枚举 9](#_Toc431384620)

[3.5. Using Other Message Types 9](#_Toc431384621)

[3.6. Nested Types 内嵌类型 9](#_Toc431384622)

[3.7. Updating A Message Type 10](#_Toc431384623)

[3.8. Any 10](#_Toc431384624)

[3.9. One of 10](#_Toc431384625)

[3.10. Maps 10](#_Toc431384626)

[3.11. Packages 10](#_Toc431384627)

[3.12. Defining Services 定义服务 10](#_Toc431384628)

[3.13. JSON Mapping, JSON映射 10](#_Toc431384629)

[3.14. Options 10](#_Toc431384630)

[3.15. Generating Your Classes 10](#_Toc431384631)

[4. protobuf2/protobuf3差别 10](#_Toc431384632)

[5. encoding 编码 11](#_Toc431384633)

[6. grpc 11](#_Toc431384634)

[7. protobuf vs thrift 11](#_Toc431384635)

[8. 参考资料 11](#_Toc431384636)

# protocol-buffers介绍

## protocol-buffers是什么?

<https://developers.google.com/protocol-buffers/docs/overview>

Protocol buffers are a flexible, efficient, automated mechanism for serializing structured data – think XML, but smaller, faster, and simpler. You define how you want your data to be structured once, then you can use special generated source code to easily write and read your structured data to and from a variety of data streams and using a variety of languages. You can even update your data structure without breaking deployed programs that are compiled against the "old" format.

简单来说：

1. protobuf是一种将结构化数据进行序列化和反序列化的方法。
2. protobuf在功能上类似XML，但是序列化后的数据更小，解析更快，使用上更简单。
3. 只要按照proto语法在.proto文件中定义好数据的结构，就可以使用专门的工具(protoc)自动生成处理数据的代码，这样就能使用不同的语言从不同的数据流中读写结构化数据。
4. protobuf还提供了很好的向后兼容，更新数据的结构后，不需在发布新的程序就可以兼容旧的格式。

## protocol-buffers如何工作?

## 为什么不使用xml?

<https://developers.google.com/protocol-buffers/docs/overview>

Protocol buffers have many advantages over XML for serializing structured data. Protocol buffers:

1. are simpler
2. are 3 to 10 times smaller
3. are 20 to 100 times faster
4. are less ambiguous
5. generate data access classes that are easier to use programmatically

protobuf相对xml有很多优势。如:

1. 比xml简单
2. 比xml小3~10倍
3. 比xml快20~100倍
4. 歧义更少
5. 生成的数据访问类更容易程序化。(QU: why ?)

例子：

xml数据：

<person>

<name>John Doe</name>

<email>jdoe@example.com</email>

</person>

protobuf文本格式数据:

# Textual representation of a protocol buffer.

# This is \*not\* the binary format used on the wire.

person {

name: "John Doe"

email: "jdoe@example.com"

}

但编码成protobuf二进制数据时，仅有28bytes大小，解析时间仅有100~200纳秒。但xml去掉空白字符后至少有69bytes大小，解析时间大概有5000~10000纳秒。

操作上protobuf更简单：

protobuf

cout << "Name: " << person.name() << endl;

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

xml:

cout << "Name: "

<< person.getElementsByTagName("name")->item(0)->innerText()

<< endl;

cout << "E-mail: "

<< person.getElementsByTagName("email")->item(0)->innerText()

<< endl;

# protobuf2

# protobuf3 (Language Guide)

<https://developers.google.com/protocol-buffers/docs/proto3>

## Defining A Message Type 定义消息类型

搜索请求消息 .proto 文件：

syntax = "proto3";

message SearchRequest {

string query = 1;

int32 page\_number = 2;

int32 result\_per\_page = 3;

}

1. The first line of the file specifies that you're using proto3 syntax: if you don't do this the protocol buffer compiler will assume you are using proto2. This must be the first non-empty, non-comment line of the file.

第一行声明你是使用的proto3语法，如果没有语法syntax声明，默认使用的是proto2语法。语法声明在文件中必须是没有空行，没的注释的第一行(This must be the first non-empty, non-comment line of the file.)。

QU: --- start ---

经测试第一行是空行或注释行也是可以的,生成的代码都是一样的。

# cat SearchRequestProto3.proto

// sdawwda dawda awdawd

syntax = "proto3";

option java\_package = "com.quxionglie.rpc.protobuf3.proto";

option java\_outer\_classname = "SearchRequestProto3";

message SearchRequest {

string query = 1;

int32 page\_number = 2;

int32 result\_per\_page = 3;

}

但是如果没有syntax = "proto3";这一行，生成代码时会出错：

# protoc --java\_out=./ SearchRequestProto3.proto

[libprotobuf WARNING google/protobuf/compiler/parser.cc:491] No syntax specified for the proto file. Please use 'syntax = "proto2";' or 'syntax = "proto3";' to specify a syntax version. (Defaulted to proto2 syntax.)

SearchRequestProto3.proto:6:3: Expected "required", "optional", or "repeated".

SearchRequestProto3.proto:7:3: Expected "required", "optional", or "repeated".

SearchRequestProto3.proto:8:3: Expected "required", "optional", or "repeated".

QU: --- end ---

1. The SearchRequest message definition specifies three fields (name/value pairs), one for each piece of data that you want to include in this type of message. Each field has a name and a type.

### Specifying Field Types

In the above example, all the fields are [scalar types](https://developers.google.com/protocol-buffers/docs/proto3#scalar): two integers (page\_number andresult\_per\_page) and a string (query). However, you can also specify composite types for your fields, including [enumerations](https://developers.google.com/protocol-buffers/docs/proto3#enum) and other message types.

在上面的例子中，所有的field都是使用了scalar 类型：两个整型(integer)和字符串(string)。你同样也可能定义组合类型：枚举或其它消息类型。

### Assigning Tags

As you can see, each field in the message definition has a unique numbered tag. These tags are used to identify your fields in the message binary format, and should not be changed once your message type is in use. Note that tags with values in the range 1 through 15 take one byte to encode, including the identifying number and the field's type (you can find out more about this in Protocol Buffer Encoding). Tags in the range 16 through 2047 take two bytes. So you should reserve the tags 1 through 15 for very frequently occurring message elements. Remember to leave some room for frequently occurring elements that might be added in the future.

The smallest tag number you can specify is 1, and the largest is 2^29 - 1, or 536,870,911. You also cannot use the numbers 19000 though 19999 (FieldDescriptor::kFirstReservedNumberthrough FieldDescriptor::kLastReservedNumber), as they are reserved for the Protocol Buffers implementation - the protocol buffer compiler will complain if you use one of these reserved numbers in your .proto. Similarly, you cannot use any previously reserved tags.

几个要点：

1. 消息定义中每个field都有唯一的数字标签。数字标签一旦使用了就不能改变。
2. 数字标签1~15使用1个字节去编码，16~2047使用2个字节去编码。
3. 数字标签范围1到2^29-1，且19000~19999是protobuf预留的, 不能使用。

QU: --- start ---

➜ proto git:(master) ✗ cat testProto3.proto

syntax = "proto3";

option java\_package = "com.quxionglie.rpc.protobuf3.proto";

option java\_outer\_classname = "TestMsgProto3";

message TestMsg {

string query = 19000;

int32 page\_number = 2;

int32 result\_per\_page = 3;

} ➜ proto git:(master) ✗ protoc --java\_out=./ testProto3.proto

testProto3.proto:7:18: Field numbers 19000 through 19999 are reserved for the protocol buffer library implementation.

QU: --- end ---

### Specifying Field Rules

Message fields can be one of the following:

1. singular: a well-formed message can have zero or one of this field (but not more than one).
2. repeated: this field can be repeated any number of times (including zero) in a well-formed message. The order of the repeated values will be preserved.

For historical reasons, repeated fields of scalar numeric types aren't encoded as efficiently as they could be. New code should use the special option [packed=true] to get a more efficient encoding. For example:

repeated int32 samples = 4 [packed=true];

You can find out more about packed encoding in [Protocol Buffer Encoding](https://developers.google.com/protocol-buffers/docs/encoding.html#packed).

### Adding More Message Types

Multiple message types can be defined in a single .proto file. This is useful if you are defining multiple related messages – so, for example, if you wanted to define the reply message format that corresponds to your SearchResponse message type, you could add it to the same .proto:

message SearchRequest {

string query = 1;

int32 page\_number = 2;

int32 result\_per\_page = 3;

}

message SearchResponse {

...

}

一个.proto文件可以定义多个消息(message)。

### Adding Comments

To add comments to your .proto files, use C/C++-style // syntax.

message SearchRequest {

string query = 1;

int32 page\_number = 2; // Which page number do we want?

int32 result\_per\_page = 3; // Number of results to return per page.

}

支持C/C++风格的注释

### Reserved Fields 保留字段

If you update a message type by entirely removing a field, or commenting it out, future users can reuse the tag number when making their own updates to the type. This can cause severe issues if they later load old versions of the same .proto, including data corruption, privacy bugs, and so on. One way to make sure this doesn't happen is to specify that the field tags (and/or names, which can also cause issues for JSON serialization) of your deleted fields are reserved. The protocol buffer compiler will complain if any future users try to use these field identifiers.

message Foo {

reserved 2, 15, 9 to 11;

reserved "foo", "bar";

}

Note that you can't mix field names and tag numbers in the same reserved statement.

QU: 如果移除或注释字段时，未来的用户可能会重新启用数字标签去定义新的字段。这样会版本问题导致问题产生。为了确保已经删除的字段不能使用，可以使用reserved标签，这样编译proto文件时，编译器会产生警告信息。

QU: --- start ---

➜ proto git:(master) ✗ cat testProto3.proto

syntax = "proto3";

option java\_package = "com.quxionglie.rpc.protobuf3.proto";

option java\_outer\_classname = "TestMsgProto3";

message TestMsg {

string query = 1;

reserved 2;

int32 page\_number = 2;

int32 result\_per\_page = 3;

} ➜ proto git:(master) ✗ protoc --java\_out=./ testProto3.proto

testProto3.proto: Field "page\_number" uses reserved number 2.

➜ proto git:(master) ✗ cat testProto3.proto

syntax = "proto3";

option java\_package = "com.quxionglie.rpc.protobuf3.proto";

option java\_outer\_classname = "TestMsgProto3";

message TestMsg {

string query = 1;

reserved "page\_number";

int32 page\_number = 2;

int32 result\_per\_page = 3;

}% ➜ proto git:(master) ✗ protoc --java\_out=./ testProto3.proto

testProto3.proto:9:9: Field name "page\_number" is reserved.

QU: --- end ---

### What's Generated From Your .proto?

When you run the protocol buffer compiler on a .proto, the compiler generates the code in your chosen language you'll need to work with the message types you've described in the file, including getting and setting field values, serializing your messages to an output stream, and parsing your messages from an input stream.

• For C++, the compiler generates a .h and .cc file from each .proto, with a class for each message type described in your file.

• For Java, the compiler generates a .java file with a class for each message type, as well as a special Builder classes for creating message class instances.

• Python is a little different – the Python compiler generates a module with a static descriptor of each message type in your .proto, which is then used with a metaclass to create the necessary Python data access class at runtime.

• For Go, the compiler generates a .pb.go file with a type for each message type in your file.

• For Ruby, the compiler generates a .rb file with a Ruby module containing your message types.

• For JavaNano, the compiler output is similar to Java but there are no Builder classes.

• For Objective-C, the compiler generates a pbobjc.h and pbobjc.m file from each .proto, with a class for each message type described in your file.

• For C#, the compiler generates a .cs file from each .proto, with a class for each message type described in your file.

## Scalar Value Types

A scalar message field can have one of the following types – the table shows the type specified in the .proto file, and the corresponding type in the automatically generated class:

## Default Values 默认值

## Enumerations 枚举

## Using Other Message Types

## Nested Types 内嵌类型

## Updating A Message Type

## Any

## One of

## Maps

## Packages

## Defining Services 定义服务

## JSON Mapping, JSON映射

## Options

## Generating Your Classes

# protobuf2/protobuf3差别

# encoding 编码

# grpc

# protobuf vs thrift

# 参考资料