



链滴

# proptobuf 笔记

作者: [whitespur](#)

原文链接: <https://ld246.com/article/1544434175187>

来源网站: [链滴](#)

许可协议: [署名-相同方式共享 4.0 国际 \(CC BY-SA 4.0\)](#)

# protobuf协议介绍

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

## What are protocol buffers?

Protocol buffers are Google's language-neutral, platform-neutral, extensible 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.

是google推出的一种轻便高效的结构化数据存储格式，可以用于结构化数据串行化，或者说序列化。它很适合做数据存储或 RPC 数据交换格式。可用于通讯协议、数据存储等领域的语言无关、平台无关、可扩展的序列化结构数据格式。

我们把变量从内存中变成可存储或传输的过程称之为序列化，序列化之后，就可以把序列化后的内容写入磁盘，或者通过网络传输到别的机器上。

反过来，把变量内容从序列化的对象重新读到内存里称之为反序列化。

## 例子

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

- Define message formats in a .proto file.
- Use the protocol buffer compiler.
- Use the Java protocol buffer API to write and read messages.

```
syntax = "proto2";
```

```
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 phones = 4;
}

message AddressBook {
repeated Person people = 1;
}
```

## 语法

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

```
message SearchRequest {
required string query = 1;
optional int32 page_number = 2;
optional int32 result_per_page = 3;
}
```

Assigning Field Numbers

## 指定域值

As you can see, each field in the message definition has a unique number. These numbers 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 field numbers in the range 1 through 15 take one byte to encode, including the field number and the field's type (you can find out more about this in Protocol Buffer Encoding). Field numbers in the range 16 through 2047 take two bytes. So you should reserve the field numbers 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.

Specifying Field Rules

## 特定域规则

You specify that message fields are one of the following:

- required: a well-formed message must have exactly one of this field.
- optional: a well-formed message can have zero or one of this field (but not more than one).
- 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.

## Scalar Value Types

类型映射表

## Optional Fields And Default Values

可选参数的默认值

```
optional int32 result_per_page = 3 [default = 10];
```

## Enumerations

枚举

```
enum Corpus {  
UNIVERSAL = 0;  
WEB = 1;  
IMAGES = 2;  
}
```

## Nested Types

嵌套

```
message Outer { // Level 0  
  message MiddleAA { // Level 1  
    message Inner { // Level 2  
      required int64 ival = 1;  
      optional bool  booly = 2;  
    }  
  }  
  message MiddleBB { // Level 1  
    message Inner { // Level 2  
      required int32 ival = 1;  
      optional bool  booly = 2;  
    }  
  }  
}
```

## Maps

映射

If you want to create an associative map as part of your data definition, protocol buffers provides a handy shortcut syntax:

```
map<key_type, value_type> map_field = N;
```

where the key\_type can be any integral or string type (so, any scalar type except for floating

oint types and bytes). Note that enum is not a valid key\_type. The value\_type can be any type except another map.

```
map<string, Project> projects = 3;
```

Packages

包名

You can add an optional package specifier to a .proto file to prevent name clashes between protocol message types.

```
package foo.bar;  
message Open { ... }
```

## Compile编译

Compiling Your Protocol Buffers

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

## API介绍

The Protocol Buffer API

Each class has its own Builder class that you use to create instances of that class.

messages have only getters while builders have both getters and setters.

Once a message object is constructed, it cannot be modified, just like a Java String. To construct a message, you must first construct a builder, set any fields you want to set to your chosen values, then call the builder's build() method.

Person john =

```
Person.newBuilder()  
  .setId(1234)  
  .setName("John Doe")  
  .setEmail("jdoe@example.com")  
  .addPhones(  
    Person.PhoneNumber.newBuilder()  
      .setNumber("555-4321")  
      .setType(Person.PhoneType.HOME))  
  .build();
```

Parsing and Serialization

解析和序列化

## 使用API

## Writing A Message

# Read A Message

## 原理，为什么这么快

<https://www.ibm.com/developerworks/cn/linux/l-cn-gpb/index.html>

对比其他序列化方法java内置, json, Protobuff, xml

## Protobuf 的主要优点就是：简单，快。具体性能测试结果见

<https://github.com/eishay/jvm-serializers/wiki>

同 XML 相比，Protobuf 的主要优点在于性能高。它以高效的二进制方式存储，比 XML 小 3 到 10，快 20 到 100 倍。

有两项技术保证了采用 Protobuf 的程序能获得相对于 XML 极大的性能提高。

## 体积小

第一点，我们可以考察 Protobuf 序列化后的信息内容。您可以看到 Protocol Buffer 信息的表示非紧凑，这意味着消息的体积减少，自然需要更少的资源。比如网络上传输的字节数更少，需要的 IO 少等，从而提高性能。

## 编码方式Varint

第二点我们需要理解 Protobuf 封解包的大致过程，从而理解为什么会比 XML 快很多。

序列化后所生成的二进制消息非常紧凑，这得益于 Protobuf 采用的非常巧妙的 Encoding 方法。

比如对于 int32 类型的数字，一般需要 4 个 byte 来表示。但是采用 Varint，对于很小的 int32 类型数字，则可以用 1 个 byte 来表示。当然凡事都有好的也有不好的一面，采用 Varint 表示法，大的数则需要 5 个 byte 来表示。从统计的角度来说，一般不会所有的消息中的数字都是大数，因此大多数情况下，采用 Varint 后，可以用更少的字节数来表示数字信息。

Varint 中的每个 byte 的最高位 bit 有特殊的含义，如果该位为 1，表示后续的 byte 也是该数字的一部分，如果该位为 0，则结束。其他的 7 个 bit 都用来表示数字。因此小于 128 的数字都可以用一个 byte 表示。大于 128 的数字，比如 300，会用两个字节来表示：1010 1100 0000 0010。

在计算机内，一个负数一般会被表示为一个很大的整数，因为计算机定义负数的符号位为数字的最高。如果采用 Varint 表示一个负数，那么一定需要 5 个 byte。为此 Google Protocol Buffer 定义了 sint32 这种类型，采用 zigzag 编码。

Zigzag 编码用无符号数来表示有符号数字，正数和负数交错，这就是 zigzag 这个词的含义了。

使用 zigzag 编码，绝对值小的数字，无论正负都可以采用较少的 byte 来表示，充分利用了 Varint 种技术。

## 封解包的速度

首先我们来了解一下 XML 的封解包过程。XML 需要从文件中读取字符串，再转换为 XML 文档对结构模型。之后，再从 XML 文档对象结构模型中读取指定节点的字符串，最后再将这个字符串转换指定类型的变量。这个过程非常复杂，其中将 XML 文件转换为文档对象结构模型的过程通常需要完词法语法分析等大量消耗 CPU 的复杂计算。

反观 Protobuf，它只需要简单地将一个二进制序列，按照指定的格式读取到 C++ 对应的结构类型中就可以了。从上一节的描述可以看到消息的 decoding 过程也可以通过几个位移操作组成的表达式计算可完成。速度非常快。