编码

Translated by Jeff

王晖 译

（译者联系方式为：lonelystellar@gmail.com）

注：关于本文档的翻译错误（包括语法错误，错别字）或中文技术交流，可以发送邮件至译者邮箱：lonelystellar@gmail.com，我们共同研究，共同进步，不胜感激。

本文描述了 Protocol Buffer 的二进制编码格式. 在应用程序中使用 Protocol Buffer 是不需要了解它的, 但是通过它来了解不同格式的编码对消息大小的影响是很有用的.

一个简单的消息

例如,如下是一个简单的消息定义:

```
message Test1 {
 required int32 a = 1;
}
```

在应用程序中, 创建一个 `Test1` 消息并设置 `a` 的值为 150. 然后序列化该消息到一个输出流. 如果能够看到编码后的消息,将会得到如下的字节:

```
08 96 01
```

到目前为止, 他还是很小的并且还是数字 – 但是他是什么意思呢? 请继续阅读...

Base 128 Varints

为了理解简单 Protocol Buffer 的编码, 首先需要理解*可变整型*.可变整型是一种使用一个或多个字节来序列化整型的方法. 小的数字用小的字节.

可变整型中的每一个字节, 除了最后一个字节外, 都是*最高有效位*集合–这些表示还有更多的字节.每个字节的低 7 位都被用来存储 7 位组中数字的两个补充说明,**最低有效位的那组在最前面**.

例如, 这里有数字 1 – 是一个单字节, 因此不需要设置最高有效位:

```
0000 0001
```

下面是 300 – 这个位比较复杂一点:

```
1010 1100 0000 0010
```

那么怎么说它就是 300 呢? 首先从每个字节放一个最高有效位, 这只是告诉我们打到了数字的末尾(可以看到, 它在第一个字节中设置了比在 varint 中更多的字节):

```
1010 1100 0000 0010

→ 010 1100  000 0010
```

反转 7 位的两组, 因为 varint 先存储最低有效位的数字. 那么就可以串联它们来得到最终的值:

```
000 0010  010 1100

→ 000 0010 ++ 010 1100

→ 100101100

→ 256 + 32 + 8 + 4 = 300
```

消息结构

现在知道 一个 protocol buffer 消息就是一组键值对. 二进制的消息只用字段的编号作为键– 每个字段的名称和声明的类型仅能够被用来按照消息类型的定义来解码(例如.proto 文件).

当消息被编码时, 键和值被串联成一个字节流. 当消息被解码时, 解析器需要能够跳过那些不能识别的字段. 这种情况下, 添加新字段到消息里的时候就 不需要破坏那些已有的不知道该新字段的程序.为此, 在有线格式消息中每对"键"其实有 2 个值– .proto 文件中的字段编号, 加上一个提供足够信息来查找如下值的长度的*有线类型*. 可用的类型如下:

类型	说明	用途
0	Varint	int32, int64, uint32, uint64, sint32, sint64, bool, enum
1	64-bit	fixed64, sfixed64, double
2	Length-delimited	string, bytes, embedded messages, packed repeated fields
3	Start group	groups (deprecated)
4	End group	groups (deprecated)
5	32-bit	fixed32, sfixed32, float

在已流化的消息中的每个键都是一个值为(field_number << 3) | wire_type 的 varint – 换句话说, 数字的最后 3 位存储 wire type.

现在再看看那个简单的例子. 现在已经了解了流中的第一个数字总是一个 varint 键, 这里它是 08, 或者 (分解的最高有效位):

```
000 1000
```

去掉最后的 3 位来获得 wire type(0)然后右移 3 位来获得字段的编号(1).因此现在知道了那个标签是 1 并且接下来的值就是一个 varint. 使用前面所学到的 varint 解码知识, 就能看到接下来的 2 个字节存储的值是 150.

```
96 01 = 1001 0110  0000 0001

    → 000 0001  ++  001 0110 (drop the msb and reverse the groups of 7 bits)

    → 10010110

    → 2 + 4 + 16 + 128 = 150
```

更多值类型

有符号整型

在前面的章节中, 所有的 wire type 为 0 的 protocol buffer 类型都字段被编码成 varint. 但是在有符号的整型类型(sint32 和 sint64)和"标准的"整型类型(int32 和 int64)之间还有一个重要的差别就是当它编码一个负数时. 如果使用 int32 或 int64 作为负数的类型,那么编码 varint 的长度永远是 10 个字节长– 它会有效把它认为是一个非常大的无符号整型. 如果使用有符号类型中的一种, 那么编码 varint 将使用一种更高效的 ZigZag 方式编码.

ZigZag 编码使用有符号整型映射为无符号整型, 以便于一个小的绝对值的数字(例如, -1)同样拥有一个小的 varint 编码值. 它用这种"zig-zag"的方式来回的穿梭于正整数和负整数之间, 因此 -1 被编码成 1， 1 被编码成 2， -2 被编码成 3， 以此类推, 如下表:

Signed Original	Encoded As
0	0
-1	1
1	2
-2	3
2147483647	4294967294
-2147483648	4294967295

换句话说, 每个值 n 都是用

```
(n << 1) ^ (n >> 31)
```

编码成 sint32s 的, 或者用

```
(n << 1) ^ (n >> 63)
```

编码成 64-bit 的.

注意第二个移位– (n >> 31) – 是一个算术移位. 因此, 也就是移位的结果不是数字都是 0 位(如果 n 是整数)要么就都是 1 位(如果 n 是负数).
当 sint32 或 sint64 被解析时, 他的值被解码回原来的值, 有符号的版本.

非 varint 数字

非 varint 数字类型都是简单的 – double 和 fixed64 的 wire type 为 1, 它告诉解析器需要一个固定 64 位的块数据; 同样的 float 和 fixed32 的 wire type 为 5, 它告诉解析器需要一个 32 位的块. 两种情况的值都存储成一个小的字节序.

字符串

wire type 为 2 (长度分隔)意味着这个值是一个 varint 编码长度的指定数字的字节数据.

```
message Test2 {
 required string b = 2;
}
```

设置 b 的值为"testing":

```
12 07 74 65 73 74 69 6e 67
```

红色字节是 UTF 编码的"testing". 这里键是 0x12 → tag = 2, type = 2. 值中 varint 的长度是 7 并且很显而易见. 接下来的 7 个字节就是要得到的字符串.

嵌套消息

如下是一个含有 Test1 的嵌套的消息的定义:

```
message Test3 {
 required Test1 c = 3;
}
```

下面是编码后版本:

```
1a 03 08 96 01
```

可以看到, 组后的 3 个字节就是在第一个例子中看到的(08 96 01), 并且它们前面的数字是 3–嵌套消息就跟字符串(wire type =2) 是一摸一样的.

可选的和可重复的元素

如果消息定义中含有 repeated 元素 (没有 [packed=true] 选项), 编码后的消息拥有 0 个或多个相同标签编号的键值对. 这些重复值不必连续的出现; 它们可能与其他字段交错开来. 在解析时元素的顺序被保留, 与其他字段的顺序将被丢失.

如果任何一个元素是 optional, 编码后的消息可能会或可能不会含有一个有编号的键值对.

通常, 一个已经编码的消息将永远不会拥有超过 1 个 optional 或 required 的字段. 但是, 解析器都能处理它. 对于数字类型和字符串, 如果同一个值出现很多次, 解析器取*最后*一个可见的值. 对于嵌套消息字段, 解析器合并多个同名字字段的实例, 就像 Message::MergeFrom 方法– 就是在靠后的所有奇数标字段实例替换靠前的实例, 奇数嵌套消息都被合并了, 重复字段都被串联起来. 这些规则的效果是解析两个串联起来并编码过的消息来得出的结果与分开解析两个消息并合并的结果是一样的. 如下:

```
MyMessage message;
message.ParseFromString(str1 + str2);
```

等于:

```
MyMessage message, message2;
message.ParseFromString(str1);
message2.ParseFromString(str2);
message.MergeFrom(message2);
```

这个属性某些时候很有用, 它可以合并两个你并不知道他们类型的消息.

盒装重复字段

版本 2.1.0 介绍了盒装重复字段, 它跟重复字段的声明差不多只是多了一个特别的选项 [packed=true] . 它们的功能就像重复字段, 但是编码的方式不同. 一个盒装重复字段包含 0 个不出现在已编码消息中的元素. 否则, 所有字段的元素都被包装到一个 wire 类型为 2(长度分割)的单键值对中. 通常每个元素都是按照同样的方式被编码的. 除非没有在前面加标签.
例如, 如果有如下的消息类型:

```
message Test4 {
 repeated int32 d = 4;
}
```

构造一个 Test4, 给重复字段 d 提供值:3, 270, 和 86942. 那么编码后的格式就是:

```
22      // tag (field number 4, wire type 2)
06      // payload size (6 bytes)
03      // first element (varint 3)
8E 02    // second element (varint 270)
9E A7 05  // third element (varint 86942)
```

只有原生数字类型的重复字段(使用 varint, 32-bit 或 64-bit wire 类型的类型)才能被声明成"盒装的".
注意,虽然对一个盒装重复字段的键值对进行编码没有什么理由,但是编码器必须准备接受多个键值对. 在这种情况下, 有效载荷将会被串联起来, 每个键值对必须包含一整个元素的数量.

字段顺序

在.proto 文件中可以使用任何顺序的字段编号, 在 C++,Java 和 Python 序列化代码中, 当消息被序列化时就会按照字段编号的顺序来写. 解析器代码就能依赖已经顺序好字段编号进行编码优化. 但是, protocol buffer 解析器能解析够任何顺序的字段, 就像所有的消息的创建都是通过简单的序列化一个对象来实现的–例如, 有时候合并两个消息只需要简单的串联它们就可以了.

如果一个消息含有未知字段, 目前的 Java 和 C++实现在已排序的已知字段后面用一个很随意的顺序来编写它们. 目前的 Python 实现不会识别未知字段.