有何错误请发邮件到我邮箱 806846483@qq.com

Encoding

此文档描述 protocol buffer message 的二进制编码，使用 protocol buffer 并不需要理解它的二进制格式 ,但是理解不同的 protocol buffer 编码是怎么影响待编码的 Message 非常有用。

A Simple Message

我们定义一个简单的 Message

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

你创建一个 Test1 message 并且设置 a 的值是 150。然后序列化 message 到输出流。如果你查看输出流内的字节，你会看到如下三个字节。

```
08 96 01
```

为什么一个如此小的数字需要三个字节，它们表示什么意思？

Base 128 Varints

为了理解上面这个简单 protocol buffer 的编码，首先你需要理解 *varints*（可变长编码）编码。Varints 使用一个或多个字节来序列化 integers。小的数字使用

更少的字节。

在 varint 内 ,每个字节(除了最后一个字节)都有一个最高有效位(most significant bit 后面使用 msb 表示这个有效位), 最高有效位用来指示是否继续读取下一个字节。剩余的 7 个 bit 位用来存储数字的 bit 位。

比如，数字 1，它只要一个字节，所以 msb 不需要设置。它的编码如下：

```
0000 0001
```

数字 300 的编码如下，它需要多个字节（300 的二进制表示是 100101100）

```
1010 1100 0000 0010
```

那么 ,这个是怎么计算出来是 300？首先 ,你需要删除每个字节的最高有效位(最后一个字节不需要，注意，一个字节是 8 位)。

```
1010 1100 0000 0010

→ 010 1100  000 0010
```

因为是小端编码（第一个字节在最前面), 第二步，反转每组的 7 字节，然后连接每个 bit 位。结果如下：

```
000 0010  010 1100

→  000 0010 ++ 010 1100

→  100101100

→  256 + 32 + 8 + 4 = 300
```

Message Structure

就像你看到的那样，一个 protocol buffer message 是一些 k-v 对。message 的

二进制版本仅使用字段的数字做 key（proto 文件里面的编码，比如上面的 optional int32 a = 1 内的 1）,字段名字仅在解码的时候通过定义文件来确定(比如.proto 文件)。

当 message 被编码时，key 和 value 被连接到字节流内。当 message 被解码时，解析器需要跳过无法识别的内容。这样，新字段就能加到 message 内（方便做往下兼容）。为了实现这个，message 的每个 key 中需要写入 proto 文件内字段的值和字段的类型（用来识别出来字段的长度）。在大多数语言的实现中 key 通常指的标记。

译注：field 的值的范围是 1-2^29-1（19000-19999 保留，不能使用）

1-15 使用一个字节

16-2047 使用俩个字节

以下是可用的字段类型：

Type	Meaning	Used for
0	Varint	int32，int64，uint32，uint64，sint32，sint64，bool，enum
1	64-bit	fixed64，sfixed64，double
2	Length-delimited	string，bytes，embedded message，packed repeated fields
3	start group	groups（deprecated）
4	end group	groups（deprecated）
5	32-bit	fixed32，sfixed32，float

译注：Length-delimited 的用处是区分出长度，见下文

每个 message 流内的 key 都是可变的值，它的值等于 (field_number << 3) | wire_type，说白了，最后三个 bit 位表示字段类型。

让我们再次回到我们简单的样例。你知道第一个数字总是表示类型 key，它是 08（丢掉 msb 位）

```
000 1000
```

你会发现最后三个 bit 的值是 0（字段类型），然后右移三位得到字段值是 1，现在我们就知道字段值是 1 然后字段类型是变长类型（字段类型是 0）。现在使用变长编码解析上节的 bit 位。你会发现往后俩个字节保存值 150

```
96 01 = 1001 0110  0000 0001

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

→ 10010110

→ 128 + 16 + 4 + 2 = 150
```

译注：150 的 protocol buffer 编码是 08 96 01

More Value Types

Signed Integers

正如你在前面章节看到的那样，protocol buffer 中类型编码 0 都是变长编码。然而，使用 singed int type(sint32 and sint64)编码负数时和标准的 int(int32 int64) 有非常大的区别。使用 int32 或 int64 作为负数的类型，编码后的结果总是 10 字节长度（被解释成一个非常大的无符号 int）。如果你使用有符号编码，结果就是使用变长的 ZigZag 编码，这是一个非常高效的编码方式。

ZigZag 编码将有符号数字映射成无符号数字，所以映射后的数字绝对值非常小（比如-1），这样编码后的结果就非常小。zig-zags 编码交错正数和负数来编码数字，所以，-1 当成 1 来编码，1 当成 2 来编码，-2 当成 3 来编码，可以参考下表。

Singed Original	Encodeed As
0	0
-1	1
1	2
-2	3
2147483647	4294967294
-2147483648	4294967295

明确来说，值 n 采用如下编码：

有符号 32 位：

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

64 位版本

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

注意，上面的右移运算是算术右移，所以，右移后左边都是符号位。

当解析 sint32 和 sint64 时，值被重新解码成原始有符号值。

Non-varint Numbers

非变长数字编码非常简单，double 和 fixed64 的类型是 1，解析器固定读取 64 位。同样，float 和 fixed32 的类型是 5，解析器应该读取 32 位。保存的字节序都是小端。

Strings

类型 2（表示成都）表示的是一个长度变化的编码方式，长度字节后是值字节。

比如下面这个 Message。

```
message Test2 {

  optional string b = 2;

}
```

假设 b 的值是 "testing"，它的字节表示如下

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

红色字节表示“testing”的 UTF8 字节。编码后的 key 是 0x12---->字段值是=2,编码类型=2。可变长度的值是 7，表示后面的 7 个字节是字符串"testing"。

译注：0x12 的二进制是 0001 0010，字段是 00010（0x12 右移 3 位），类型编码是 010。

Embedded Messages

下面是我们定义的一个内置信息，里面包含另外一个信息 Test1：

```
message Test3 {

  optional Test1 c = 3;

}
```

我们再次把它进行编码，假设 Test1 的值再次设置成 150。我们得到如下的字节

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

你可以发现，最后三个字节和我们前面的样例一模一样（08 96 01），前面的数字是 3-内置信息表示形式和 string 相同（类型是 2）

译注：0x1a　0001 1010 字段是 00011(0x1a 右移 3 三位)，类型编码是 010。长度是 0x03。

Optional And Repeated Elements

在 proto2 里定义一个 **repeated** 元素（没有设置 packed=true），编码后的 message 有 0 个或者多个相同字段的 k-v 数据（field 是同样的值）。这些相同的值不一定连续出现，在它们之间会插入其他字段。解析时保留 repeat 元素的顺序，尽管 repeat 字段与其他字段的顺序已经丢失。在 proto3 中，repeated 字段使用 packed encoding，你可以使用下面介绍的方法读取它。

proto3 内非 repeated 的字段和 proto2 内的 optional 字段，编码后的信息内可能有 k-v 值，也有可能没有 k-v 值。

正常而言，待编码的信息不会有多个 non-repeated 字段，然而，解析器应该能

处理多个 no-repeated 字段的情形。对于数值和字符串类型 ,如果同一个字段(字段名一样) 出现多次 , 解析器应该使用**最后一次**出现的字段。对于内置的信息字段，解析器合并相同字段的多个实例，比如 Message::MergeFrom 方法-也就是说 , 后面实例内单个数量的字段替换前面的字段 , 合并单个内置的消息 , 然后连接重复的字段。这个规则的影响是 : 解析俩个连接的消息生成的结果和单独解析然后合并的结果一样。比如下面这个：

```
MyMessage message;

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

和下面这个等价

```
MyMessage message, message2;

message.ParseFromString(str1);

message2.ParseFromString(str2);

message.MergeFrom(message2);
```

有时候这个功能非常有用，它允许你合并俩个消息，虽然你不知道它们的类型。

Packed Repeated Fields

版本 2.1.0 里面介绍了 Packed Repeated Fields , proto2 内是 repeated+特殊的 [packed=true] 选项。在 proto3 内，单个数字类型的重复字段自动有 packed 属性。这个功能和 repeated 字段相似，但是编码方式不同。包含 0 个元素的 packed repeated 字段不会在内置信息中出现。要不然所有的字段压缩到一个 k-v 内，并且使用类型 2（长度限定）, 所有元素的编码方式都和正常情况一样，除了前面没有 key。

比如，下面这个 message 类型：

```
message Test4 {

  repeated int32 d = 4 [packed=true];

}
```

假设构建了一个 Test4 对象，提供的值是 3，270，86942。编码后的值如下。

```
22       // key (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)
```

只有重复的原始数字类型（varint,32-int.64-int）能被声明为"packed"。

尽管没有必要为 packed 的 repeated 字段使用多个 k-v 编码 ,但是编码器必须能够接受多个 k-v 编码。如果使用多个 k-v 编码 ,应该连接每个 k-v 形成 payload，每个 k-v 内应该包含这个数字的完整编码。

Protocol buffer 解析器必须能够解析未进行 packed 的 packed repeated 和进行 packed repeated 的字段，这就允许添加[packed=true]到已经存在的字段，并且是无条件向下，向后兼容。

Field Order

proto 文件内字段编号可以是任意的顺序。这个顺序不会影响 message 编码方式。

当 message 序列化后，不保证字段的写入顺序。序列化顺序由实现方式指定，并且将来可能会变化。因此，protocol buffer 解析器必须能够以任何顺序解析字段。

Implications

1. 不要假设序列化的字节不变的。在其他 protocol buffer 的序列化中也是如此。
2. 默认情况下，多次调用重复序列化方法可能不会返回同样的字节；默认情况下，序列化字节不确定。

   确定序列化仅保证对于相同的二进制文件输出相同的字节。在不同的版本的二进制文件可能输出不同的字节。
3. 以下检查 protocol buffer message 实例的操作可能会失败

   ```
   foo.SerializeAsString() == foo.SerializeAsString()
   Hash(foo.SerializeAsString()) ==
   Hash(foo.SerializeAsString())
   CRC(foo.SerializeAsString()) == CRC(foo.SerializeAsString())
   FingerPrint(foo.SerializeAsString()) ==
   FingerPrint(foo.SerializeAsString())
   ```
4. 以下是几个简单的样例，foo 和 bar 在逻辑上相等，但是序列化出不同的字节

   1. bar 由老的服务序列化，某些字段不能识别。

2. bar 由不同语言的程序序列化，字段序列化时顺序不同。
3. bar 的某些字段使用不确定的序列化方式。
4. bar 有一个字段包含序列化后的字节，这些字节的序列化方式不同。
5. bar 由一个新的服务序列化，它序列化顺序和以前的不同。
6. foo 和 bar 都是连接不同的 message，但是它们连接的顺序不同。

原文链接：https://developers.google.com/protocol-buffers/docs/encoding