深入 ProtoBuf - 编码

抽奖

在对 ProtoBuf 做了一些基本介绍之后，这篇开始进入正题，深入 ProtoBuf 的一些原理，让我们看看 ProtoBuf 是如何尽其所能的压榨编码性能和效率的。

编码结构

TLV 格式是我们比较熟悉的编码格式。

所谓的 TLV 即 Tag - Length - Value。Tag 作为该字段的唯一标识，Length 代表 Value 数据域的长度，最后的 Value 便是数据本身。

ProtoBuf 编码采用类似的结构，但是实际上又有较大区别，其编码结构可见下图：

ProtoBuf 编码结构图.png
我们来一步步解析上图所表达的编码结构。

首先，每一个 message 进行编码，其结果由一个个字段组成，每个字段可划分为 **Tag - [Length] - Value**，如下图所示：

ProtoBuf 编码结构图-1.png

> 特别注意这里的 [Length] 是可选的，含义是针对不同类型的数据编码结构可能会变成 **Tag - Value** 的形式，如果变成这样的形式，没有了 Length 我们该如何确定 Value 的边界？答案就是 **Varint 编码**，在后面将详细介绍。

继续深入 Tag ，Tag 由 **field_number** 和 **wire_type** 两个部分组成：

- **field_number**：message 定义字段时指定的字段编号
- **wire_type**：ProtoBuf 编码类型，根据这个类型选择不同的 Value 编码方案。

整个 Tag 采用 Varints 编码方案进行编码，**Varints 编码**会在后面详细介绍。

Tag 结构如下图所示：

ProtoBuf 编码结构图-2.png
3 bit 的 wire_type 最多可以表达 8 种编码类型，目前 ProtoBuf 已经定义了 6 种，如下图所示：

PB 编码数据类型.png
第一列即是对应的类型编号，第二列为面向最终编码的编码类型，第三列是面向开发者的 message 字段的类型。

> 注意其中的 **Start group** 和 **End group** 两种类型已被遗弃。

> 另外要特别注意一点，虽然 wire_type 代表编码类型，但是 Varint 这个编码类型里针对 sint32、sint64 又会有一些特别编码（**ZigTag 编码**）处理，相当于 Varint 这个编码类型里又存在两种不同编码。

重新来看完整的编码结构图：

ProtoBuf 编码结构图.png
现在我们可以理解一个 message 编码将由一个个的 field 组成，每个 field 根据类型将有如下两种格式：

- **Tag - Length - Value**：编码类型表中 Type = 2 即 Length-delimited 编码类型将使用这种结构，
- **Tag - Value**：编码类型表中 Varint、64-bit、32-bit 使用这种结构。

> 可以思考一下为什么这么设计编码方案？可以看完下面各种编码详细的介绍再来细细品味这个问题。

其中 Tag 由字段编号 field_number 和 编码类型 wire_type 组成， Tag 整体采用 Varints 编码。

现在来模拟一下，我们接收到了一串序列化的二进制数据，我们先读一个 Varints 编码块，进行 Varints 解码，读取最后 3 bit 得到 wire_type（由此可知是后面的 Value 采用的哪种编码），随后获取到 field_number （由此可知是哪一个字段）。依据 wire_type 来正确读取后面的 Value。接着继续读取下一个字段 field...

Varints 编码

上一节中多次提到 Varints 编码，现在我们来正式介绍这种编码方案。

总结的讲，Varints 编码的规则主要为以下三点：

1. 在每个字节开头的 bit 设置了 **msb(most significant bit)**，标识是否需要继续读取下一个字节
2. 存储数字对应的二进制补码
3. 补码的低位排在前面

> 为什么低位排在前面？这里主要是为编码实现（移位操作）做的一个小优化。可以尝试写个二进制移位进行编码解码的小例子来体会这一点。

先来看一个最为简单的例子：

```
int32 val =  1;  // 设置一个 int32 的字段的值 val = 1; 这时编码的结果如下
原码：0000 ... 0000 0001  // 1 的原码表示
补码：0000 ... 0000 0001  // 1 的补码表示
Varints 编码：0#000 0001 (0x01)   // 1 的 Varints 编码，其中第一个字节的 msb = 0
```

- **编码过程：**
  数字 1 对应补码 0000 ... 0000 0001（规则 2），从末端开始取每 7 位一组并且反转排序（规则 3），因为 0000 ... 0000 0001 除了第一个取出的 7 位组（即原数列的后 7 位），剩下的均为 0。所以只需取第一个 7 位组，无需再取下一个 7 bit，那么第一个 7 位组的 msb = 0。最终得到

  0 | 000 0001 (0x01)
- **解码过程：**
  我们再做一遍解码过程，加深理解。

  编码结果为 0#000 0001 (0x01)。首先，每个字节的第一个 bit 为 msb 位，msb = 1 表示需要再读一个字节（还未结束），msb = 0 表示无需再读字节（读取到此为止）。

  在上面的例子中，数字 1 的 Varints 编码中 msb = 0，所以只需要读完第一个字节无需再读。去掉 msb 之后，剩下的 **000 0001** 就是**补码的逆序**，但是这里

只有一个字节，所以无需反转，直接解释补码 000 0001，还原即为数字 1。

注意：这里编码数字 1，Varints 只使用了 1 个字节。而正常情况下 int32 将使用 4 个字节存储数字 1。

再看一个需要两个字节的数字 666 的编码：

```
int32 val = 666; // 设置一个 int32 的字段的值 val = 666; 这时编码的结果如下
原码：000 ... 101 0011010  // 666 的源码
补码：000 ... 101 0011010  // 666 的补码
Varints 编码：1#0011010  0#000 0101  (9a 05)   // 666 的 Varints 编码
```

- **编码过程：**
  666 的补码为 000 ... 101 0011010，从后依次向前取 7 位组并反转排序，则得到：

  0011010 | 0000101
  加上 msb，则

  1 0011010 | 0 0000101 (0x9a 0x05)
- **解码过程：**
  编码结果为 1#0011010 0#000 0101 (9a 05)，与第一个例子类似，但是这里的第一个字节 msb = 1，所以需要再读一个字节，第二个字节的 msb = 0，则读取两个字节后停止。读到两个字节后先去掉两个 msb，剩下：

  0011010 000 0101
  将这两个 7-bit 组反转得到补码：

  000 0101 0011010
  然后还原其原码为 666。

注意：这里编码数字 666，Varints 只使用了 2 个字节。而正常情况下 int32 将使用 4 个字节存储数字 666。

仔细品味上述的 Varints 编码，我们可以发现 Varints 的本质实际上是每个字节都**牺牲**一个 bit 位（msb），来表示是否已经结束（是否还需要读取下一个字节），msb 实际上就起到了 Length 的作用，正因为有了 msb（Length），所以我们可以摆脱原来那种无论数字大小都必须分配四个字节的窘境。**通过 Varints 我们可以让小的数字用更少的字节表示。**从而提高了空间利用和效率。

这里为什么强调**牺牲**？因为每个字节都拿出一个 bit 做 msb，而原先这个 bit 是可直接用来表示 Value 的，现在每个字节都少了一个 bit 位即只有 7 位能真正用来表达 Value。那就意味这 4 个字节能表达的最大数字为 2^{28}，而不再是 2^{32} 了。

这意味着什么？意味着当数字大于 2^{28} 时，采用 Varints 编码将导致分配 5 个字节，而原先明明只需要 4 个字节，此时 Varints 编码的效率不仅不是提高反而是下降。
但这并不影响 Varints 在实际应用时的高效，因为事实证明，在大多数情况下，小于 2^{28} 的数字比大于 2^{28} 的数字出现的更为频繁。

到目前为止，好像一切都很完美。但是当前的 Varints 编码却存在着明显缺陷。我们的例子好像只给出了正数，我们来看一下负数的 Varints 编码情况。

int32 val = -1
原码：1000 ... 0001 // 注意这里是 8 个字节
补码：1111 ... 1111 // 注意这里是 8 个字节
再次复习 Varints 编码：对补码取 7 bit 一组，低位放在前面。
上述补码 8 个字节共 64 bit，可分 9 组且这 9 组均为 1，这 9 组的 msb 均为 1（因为还有最后一组）
最后剩下一个 bit 的 1，用 0 补齐作为最后一组放在最后，最后得到 Varints 编码
Varints 编码：1#1111111 ... 0#000 0001 (FF FF FF FF FF FF FF FF FF 01)

注意，因为负数必须在最高位（符号位）置 1，这一点意味着无论如何，负数都必须占用所有字节，所以它的补码总是占满 8 个字节。你没法像正数那样去掉多余的高位（都是 0）。再加上 msb，最终 Varints 编码的结果将固定在 10 个字节。

> **为什么是十个字节？** int32 不应该是 4 个字节吗？这里是 ProtoBuf 基于兼容性的考虑（比如开发者将 int64 的字段改成 int32 后应当不影响旧程序），而将 int32 扩展成 int64 的八个字节。
> **为什么之前讲正数的时候没有这种扩展？**。请仔细品味 Varints 编码，正数的前提下 int32 和 int64 天然兼容！

所以目前的情况是我们定义了一个 int32 类型的变量，如果将变量值设置为负数，那么直接采用 Varints 编码的话，其编码结果将总是占用十个字节，这显然不是我们希望得到的结果。如何解决？

ZigZag 编码

在上一节中我们提到了 Varints 编码对负数编码效率低的问题。

为解决这个问题，ProtoBuf 为我们提供了 sint32、sint64 两种类型，当你在使用这两种类型定义字段时，ProtoBuf 将使用 ZigZag 编码，而 ZigZag 编码将解决负数编码效率低的问题。

ZigZag 的原理和概念比我们想象的简单易懂，一句话就可概括介绍 ZigZag 编码：

> **ZigZag 编码：有符号整数映射到无符号整数，然后再使用 Varints 编码**

如下图所示：

对于 ZigZag 编码的思维不难理解，既然负数的 Varints 编码效率很低，那么就将负数映射到正数，然后对映射后的正数进行 Varints 编码。解码时，解出正数之后再按映射关系映射回原来的负数。

例如我们设置 int32 val = -2。映射得到 3，那么对数字 3 进行 Varints 编码，将结果存储或发送出去。接收方接到数据后进行 Varints 解码，得到数字 3，再将 3 映射回 -2。

> 这里的“映射”是以移位实现的，并非存储映射表。

Varint 类型

介绍了 Varints 编码和 ZigZag 编码之后，我们就可以继续深入分析每个类型的编码。

在第一节中我们提到了 wire_type 目前已定义 6 种，其中两种已被遗弃（Start group 和 End group），只剩下四种类型：**Varint、64-bit、Length-delimited、32-bit。**

接下来我们就来一个个详细分析，彻底搞明白 ProtoBuf 针对每种类型的编码策略。

注意，我们在之前已经强调过，与其它三种类型不同，Varint 类型里不止一种编码策略。 除了 int32、int64 等类型的 Varints 编码，还有 sint32、sint64 类型的 ZigZag 编码。

int32、int64、uint32、uint64、bool、enum

当我们使用 int32、int64、uint32、uint64、bool、enum 声明字段类型时，其字段值将使用之前介绍的 Varints 编码。

其中 bool 的本质为 0 和 1，enum 本质为整数常量。

在结合本文开头介绍的编码结构： **Tag - [Length] - Value**，这里的 Value 采用 Varints 编码，因此不需要 Length，则编码结构为 **Tag - Value**，其中 Tag 和 Value 均采用 Vartins 编码。

int32、int64、uint32、uint64

来看一个最简单的 int32 的小例子：

```
syntax = "proto3";


// message 定义
message Example1 {
    int32 int32Val = 1;
}
```

在程序中设置字段值为 1，其编码结果为：

```
//  设置字段值 为 1
Example1 example1;
example1.set_int32val(1);
// 编码结果
tag-(Varints)0#0001 000 + value-(Varints)0#000 0001 = 0x08 0x01
```

在程序中设置字段值为 666，其编码结果为：

```
//  设置字段值 为 666
Example1 example1;
example1.set_int32val(666);
// 编码结果
tag-(Varints)00001 000 + value-(Varints)1#0011010  0#000 0101 = 0x08 0x9a 0x05
```

在程序中设置字段值为 -1，其编码结果为：

```
//  设置字段值 为 1
Example1 example1;
example1.set_int32val(-1);
// 编码结果
tag-(Varints)00001 000 + value-(Varints)1#1111111 ... 0#000 0001 = 0x08 0xFF 0xFF 0xFF 0xFF 0x
```

int64、uint32、uint64 与 int32 同理

bool、enum

bool 的例子：

```
syntax = "proto3";


// message 定义
message Example1 {
    bool boolVal = 1;
}
```

在程序中设置字段值为 true，其编码结果为：

```
//  设置字段值 为 true
Example1 example1;
example1.set_boolval(true);
```

```
// 编码结果
tag-(Varints)00001 000 + value-(Varints)0#000 0001 = 08 01
```

在程序中设置字段值为 false，其编码结果为：

```
//  设置字段值 为 false
Example1 example1;
example1.set_boolval(false);

// 编码结果
空
```

> 这里有个有意思的现象，当 boolVal = false 时，其编码结果为空，为什么？
> 这里是 ProtoBuf 为了提高效率做的又一个小技巧：规定一个默认值机制，当读出来的字段为空的时候就设置字段的值为默认值。而 bool 类型的默认值为 false。也就是说将 false 编码然后传递（消耗一个字节），不如直接不输出任何编码结果（空），终端解析时发现该字段为空，它会按照规定设置其值为默认值（也就是 false）。
> 如此，可进一步节省空间提高效率。

enum 的例子：

```
syntax = "proto3";

// message 定义
message Example1 {
    enum COLOR {
        YELLOW = 0;
        RED = 1;
        BLACK = 2;
        WHITE = 3;
        BLUE = 4;
    }
    // 枚举常量必须在 32 位整型值的范围
    // 使用 Varints 编码,对负数不够高效，因此不推荐在枚举中使用负数
    COLOR colorVal = 1;
}
```

在程序中设置字段值为 Example1_COLOR_BLUE，其编码结果为：

```
//  设置字段值 为 Example1_COLOR_BLUE
Example1 example1;
example1.set_colorval(Example1_COLOR_BLUE);

// 编码结果
tag-(Varints)00001 000 + value-(Varints)0#000 0100 = 08 04
```

sint32、sint64

sint32、sint64 将采用 ZigZag 编码。编码结构依然为 Tag - Value，只不过在编码和解码的过程中多出一个映射的过程，映射后依然采用 Varints 编码。
来看 sint32 的例子：

```
syntax = "proto3";

// message 定义
message Example1 {
    sint32 sint32Val = 1;
}
```

在程序中设置字段值为 -1，其编码结果为：

```
//  设置字段值 为 -1
Example1 example1;
example1.set_colorval(-1);

// 编码结果，1 映射回 -1
tag-(Varints)00001 000 + value-(Varints)0#000 0001 = 08 01
```

在程序中设置字段值为 -2，其编码结果为：

```
//  设置字段值 为 -2
Example1 example1;
example1.set_colorval(-2);

// 编码结果，3 映射回 -2
```

编码结果：tag-(Varints)00001 000 + value-(Varints)0#000 0011 = 08 03

sint64 与 sint32 同理。

> int、uint 和 sint：之所以同时出现了这三种类型，是因为历史和代码迭代的结果。ProtoBuf 最初只有 int 类型，由于 int 类型不适合负数（负数编码效率低），所以提供了 sint。因为 sint 的一部分正数其实是表达的负数，所以其正数范围有所减小，所以在一些全是正数场景下需要提供 uint 类型。

64-bit 和 32-bit 类型

64-bit 和 32-bit 比较简单，与 Varints 一样其编码结构为 **Tag-Value**，不同的是不管数字大小，64-bit 存储 8 字节，32-bit 存储 4 字节。读取时同理，64-bit 直接读取 8 字节，32-bit 直接读取 4 字节。

> 为什么需要 64-bit 和 32-bit？之前已经分析过了 Varints 编码在一定范围内是有高效的，超过某一个数字占用字节反而更多，效率更低。如果现在有场景是存在大量的大数字，那么使用 Varints 就不太合适了，此时使用 64-bit 和 32-bit 更为合适。具体的，如果数值比 2^{56} 大的话，64-bit 这个类型比 uint64 高效，如果数值比 2^{28} 大的话，32-bit 这个类型比 uint32 高效。

fixed64、sfixed64、double

来看例子：

```
// message 定义
syntax = "proto3";

message Example1 {
    fixed64 fixed64Val = 1;
    sfixed64 sfixed64Val = 2;
    double doubleVal = 3;
}
```

在程序中分别设置字段值 1、-1、1.2，其编码结果为：

```
//  设置字段值 为 -2
example1.set_fixed64val(1)
example1.set_sfixed64val(-1)
example1.set_doubleval(1.2)

// 编码结果，总是 8 个字节
09 # 01 00 00 00 00 00 00 00
11 # FF FF FF FF FF FF FF FF  (没有 ZigZag 编码)
19 # 33 33 33 33 33 33 F3 3F
```

fixed32、sfixed32、float

与 64-bit 同理。

Length-delimited 类型

string、bytes、EmbeddedMessage、repeated

终于遇到了体现编码结构图中 [Length] 意义的类型了。
Length-delimited 类型的编码结构为 **Tag - Length - Value**

这种编码方式很好理解，来看例子：

```
syntax = "proto3";

// message 定义
message Example1 {
    string stringVal = 1;
    bytes bytesVal = 2;
    message EmbeddedMessage {
        int32 int32Val = 1;
        string stringVal = 2;
    }
    EmbeddedMessage embeddedExample1 = 3;
    repeated int32 repeatedInt32Val = 4;
    repeated string repeatedStringVal = 5;
}
```

设置相应的值：

```
Example1 example1;
example1.set_stringval("hello,world");
example1.set_bytesval("are you ok?");

Example1_EmbeddedMessage *embeddedExample2 = new Example1_EmbeddedMessage();

embeddedExample2->set_int32val(1);
embeddedExample2->set_stringval("embeddedInfo");
example1.set_allocated_embeddedexample1(embeddedExample2);

example1.add_repeatedint32val(2);
example1.add_repeatedint32val(3);
example1.add_repeatedstringval("repeated1");
example1.add_repeatedstringval("repeated2");
```

最终编码的结果为：

```
0A 0B 68 65 6C 6C 6F 2C 77 6F 72 6C 64
12 0B 61 72 65 20 79 6F 75 20 6F 6B 3F
1A 10 08 01 12 0C 65 6D 62 65 64 64 65 64 49 6E 66 6F
22 02 02 03[ proto3 默认 packed = true]（编码结果打包处理，见下一小节的介绍）
2A 09 72 65 70 65 61 74 65 64 31 2A 09 72 65 70 65 61 74 65 64 32(repeated string 为啥不进行默认
```

读者可对照上面介绍过的编码来理解这段相对复杂的编码结果。
（为降低难度，已按字段分行，即第一个字段的编码结果对应第一行，第二个字段对应第二行...）

补充 packed 编码

在 proto2 中为我们提供了可选的设置 [packed = true]，而这一可选项在 proto3 中已成默认设置。

> packed 目前只能用于 primitive 类型。

packed = true 主要使让 ProtoBuf 为我们把 repeated primitive 的编码结果打包，从而进一步压缩空间，进一步提高效率、速度。这里打包的含义其实就是：原先的 repeated 字段的编码结构为 **Tag-Length-Value-Tag-Length-Value-Tag-**

Length-Value...，因为这些 Tag 都是相同的（同一字段），因此可以将这些字段的 Value 打包，即将编码结构变为 **Tag-Length-Value-Value-Value...**

上一节例子中 repeatedInt32Val 字段的编码结果为：

```
22 | 02 02 03
22 即 00100010 -> wire_type = 2(Length-delimited), field_number = 4(repeatedInt32Val 字段)，02 字节长度为 2，则读取两个字节，之后按照 Varints 解码出数字 2 和 3。
```

例子代码

本文所有例子的代码请访问 例子代码

下一篇

汪

汪