精通protobuf原理系列之（二）

编码原理剖析

精通protobuf原理之二：编码原理剖析

津的技术专栏
后端架构师

关注他

12 人赞同了该文章

1 说在前面

目录

因为网上都是一些零零散散的 protobuf 相关介绍，加上笔者最近因为项目的原因深入剖析了 protobuf，所以想做一个系统的《精通 protobuf 原理》系列的分享：

- 「精通 protobuf 原理之一：为什么要使用它以及如何使用」；
- 「精通 protobuf 原理之二：编码原理剖析」；
- 「精通 protobuf 原理之三：反射原理剖析」；
- 「精通 protobuf 原理之四：RPC 原理剖析」；
- 「精通 protobuf 原理之五：Arena 分配器原理剖析」。
- 后续的待定……

本文是系列文章的第二篇，本文适合 protobuf 入门、进阶的开发者阅读，是一篇讲原理的文章，主要是介绍了如何正确使用protobuf 的特性，以比较大地发挥它的优势。阅读本文之后，开发者能够对protobuf编码原理有深入的理解，在日常开发中能够熟练运用。

本文基于protobuf 的 `3.17.3` 版本进行分析、 `proto3` 的语法、编码示例使用 `C++` 语言实现。

阅读本文大概需要十分钟左右。建议读者先阅读目录，先大概了解有哪些内容，然后在选择全部阅读还是选择性阅读，以提高阅读效率。

2 初识protobuf语法

protobuf 官方实现了一门语言，专门用来自定义数据结构。 `protoc` 是这门语言的编译工具，可编译生成指定编程语言（如 `C++` 、 `Java` 、 `Golang` 、 `Python` 、 `C#` 等）的源代码，然后开发者可以轻松在这些语言中使用该源代码进行编程。

先从以下 `serialize.proto` 开始。

```
//协议版本
syntax = "proto3";

//命名空间
package mytest;

//依赖的其他 proto 源文件，
//在依赖的数据类型在其他 proto 源文件中定义的情况下，
```

```
//message 是消息体，它就是一个结构体/类
message SubTest {
  int32              i32      =   1;
}

message Test {
//[数据类型]   [字段]       [field-number]
  int32              i32      =   1;
  int64              i64      =   2;
  uint32             u32      =   3;
  uint64             u64      =   4;
  sint32             si32     =   5;
  sint64             si64     =   6;
  fixed32            fx32     =   7;
  fixed64            fx64     =   8;
  sfixed32           sfx32    =   9;
  sfixed64           sfx64    =   10;
  bool               bl       =   11;
  float              f32      =   12;
  double             d64      =   13;
  string             str      =   14;
  bytes              bs       =   15;
  repeated int32     vec      =   16;
  map<int32, int32>  mp       =   17;
  SubTest            test     =   18;
  oneof object {
    float            obj_f32  =   19;
    string           obj_str  =   20;
  }
  google.protobuf.Any any     =   21;
}
```

3 基于protobuf的编程示例

```
$ tree ./test_serialize
./test_serialize
├── Makefile
└── test_serialize.cpp
```

test_serialize.cpp 源文件：

```
#include <cstdio>
#include <iostream>
#include "test.pb.h"

//输出十六进制编码
static void dump_hexstring(const std::string& tag, const std::string& data) {
  printf("%s:\n", tag.c_str());
  for (size_t i = 0; i < data.size(); ++i) {
    printf("%02x ", (unsigned char)data[i]);
  }
  printf("\n\n");
}

static void test_1() {
  mytest::Test t1;
  t1.set_i32(300);
  std::string buf;
  t1.SerializeToString(&buf);
  dump_hexstring("==== test_1 ====", buf);
}
```

```
  return 0;
}
```

Makefile 源文件：

```
CC = g++
CXXFLAGS = -std=c++11
TARGET = test_serialize
SOURCE = $(wildcard *.cpp)
OBJS = $(patsubst %.cpp, %.o, $(SOURCE))
INCLUDE = -I./
LIBS = -lproto -lprotobuf
LIBPATH = -L../proto

$(TARGET): $(OBJS)
    $(CC) $(CXXFLAGS) -o $@ $^ $(LIBPATH) $(LIBS)

%.o: %.c
    protoc -I=./ --cpp_out=./ ./echo.proto
    $(CC) $(CXXFLAGS) $(INCLUDE) -o $@ -c $^

.PHONY:clean
clean:
    rm -f *.o $(TARGET)
```

编译和执行：

```
$ make
$ ./test_serialize
==== test_1 ====:
08 ac 02
```

4 Protobuf 的数据类型

以下是一个protobuf 数据类型和其他编程语言的数据类型的映射关系表。

Protobuf Type	说明	C++ Type	Java Type	Python Type[2]	Go Type
float	固定4个字节	float	float	float	float32
double	固定8个字节	double	double	float	float64
int32	varint编码	int32	int	int	int32
uint32	varint编码	uint32	int	int/long	uint32
uint64	varint编码	uint64	long	int/long	uint64
sint32	zigzag 和 varint编码	int32	int	int	int32
sint64	zigzag 和 varint编码	int64	long	int/long	int64
fixed32	固定4个字节	uint32	int	int	uint32
fixed64	固定8个字节	uint64	long	int/long	uint64
sfixed32	固定4个字节	int32	int	int	int32
sfixed64	固定8个字节	int64	long	int/long	int64
bool	固定一个字节	bool	boolean	bool	bool
string	Lenth-Delimited	uint64	String	str/unicode	string
	Lenth-				

bytes	Lenth-Delimited	string	ByteString	str	[]byte

这里读者先有个大概的印象，后面会详细介绍每个数据类型。

5 protobuf编码方式

在详细介绍protobuf 的数据类型之前，这里先了解一下protobuf的编码，在后面介绍每个数据类型的时候会用到这些知识点。

wire-type	名称	说明	类型
0	Varint	可变长整形	非ZigZag编码类型：int32, uint32, int64, uint64, bool, enum； ZigZag编码类型：sint32, sint64
1	64-bits	固定8个字节大小	fixed64, sfixed64, double
2	Length-delimited	Length + Body方式	string, bytes, embedding message, packed repeated fields
5	32-bits	固定4个字节大小	fixed32, sfixed32, float

> 注： `wire_type` 为 `3` 、 `4` 的编码类型官方已经弃用，所以这里也不在介绍。

5.1 Varint

5.1.1 Varint 是什么

Varint 编码是一种可变长的编码方式，值越小的数字，使用越少的字节数表示。它的原理是通过减少表示数字的字节数从而实现数据体积压缩。

先理解几个概念，因为后面需要用到这些概念：

- **field-number**：指的是 message 中的最后一列的数字；

```
protobuf int32 i32 = 1; //其中 1 就是 field-number。
```

- **wire-type**：编码类型，比如 Varint 的编码类型为 0「见以上“5 protobuf编码方式”的表格」；
- **msb**：全称 `most significant bit` ，指的是每个字节的最高位 （例如：0x80 的 二进制是 10000000，其最高位是 1，即msb 为 1）。

Varint 是怎样编码的？先了解一下 Tag 信息 和 Data 信息：

- **Tag 信息**：主要存储 field-number 和 wire-type；
- **Data 信息**：编码后的序列。

> 注：Varint 编码序列 = Tag信息 + Data信息。

5.1.2 Tag 信息

使用一个字节来表示 Tag 信息，高 `5` 位表示 `field-number` ，低 `3` 位表示 `wire-type` 。

```
[7] [6] [5] [4] [3] [2]  [1]  [0]
|<----- field ----->|<-- wire -->|
```

> 注：这个使用一个字节并非编码后的一个字节，而是编码前的一个字节，编码后可能是两个字节。为什么呢？因为 计算好 `Tag` 之后，还要经过 Varint 编码才是最终的编码，即 `Tag = VarintEncode( field-number << 3 | wire_type)` 。

5.1.3 Data 信息

在 C++ 中，int 类型的编码是固定的，无论数值大小，都使用固定 4 个字节来存储。假如数值为 1，二进制为 00000000 00000000 00000000 00000001，其实有效值只有最后一个字节 00000001，前三个字节是**浪费**的。

如果使用 `Length + body` 的方式编码呢?

- length 能不能和 Tag 公用一个字节？整形最大 8 个字节，二进制 1000，所以需要占用 4 位，wire-type 不能再压缩了，field-number 压缩之后只剩下 1 bit （5 - 4 = 1），这限制了 message 中的字段数量，此方案不可行；
- 使用单独字段表示length，那么编码之后为 00000001 00000001，第一个00000001 为 length，第二个为值，加上 Tag 一个字节，总共 3 个字节。

但是，Varint 编码可以做到总共可以只用 2 个字节来表示值 1。***这是怎么做到的?***

> 注：Varint的每个字节只有低7位存储数据，最高位（即msb）作为标志位，0 代表后面没有再跟字节了，1 代表后面的字节还是属于当前字段的，可以继续读一个字节，以此类推 ……。

1 的 Varint 编码的 data 为 00000001，即0（msb） + 0000001（低 7 位）。

下面详细分析该编码。

在 「基于 protobuf 的编程示例」一节的示例中，int32 类型的 i32 字段（ `field-number` 为 1），其值为 300 的时候，编码结果为 `08 ac 02` ，怎么得来的?

```
比如这个int32类型字段的值为300，那么序列化之后：
08 ac 02
 |  |__|___ 值
 |_________ 元数据 (field-number << 3 | wire-type) = (1 << 3 | 0) = 0x08

ac 02 是怎么得来的?

i32的值为 300
|__ 0x012c             //十六进制
|__ 00000001 00101100  //二进制
|__ 0000000100101100   //合并
|__ 00 0000010 0101100 //重新按7位一组切割
|__ 0000010 0101100    //高位全0的组省略
|__ 0101100 0000010    //逆序，因为使用小端字节序，
|__ 10101100 00000010  //每一组加上msb，除了最后一组是msb是0，其他的都为1
|__ ac 02               //十六进制
```

> 注： 计算机硬件有两种储存数据的方式：大端字节序（big endian）和小端字节序（little endian）。 举例来说，数值0x2211使用两个字节储存：高位字节是0x22，低位字节是0x11。大端字节序：高位字节在前，低位字节在后，这是人类读写数值的方法，即以0x2211形式存储； 小端字节序：低位字节在前，高位字节在后，即以0x1122形式储存。

5.2 ZigZag 编码

5.2.1 ZigZag 编码解决了什么问题

先看下背景。假如 int32 类型的字段，其值为 -1 时，在内存中，因为使用了补码，所以存储为 ffffffff（4个字节），然后在 Varint 序列化的之前会强制转换成 int64 类型，这样其值会变为 ffffffff ffffffff。转换成 Varint 编码时，会加上 Tag ，以及 msb ，总共是 10 个字节。我们希望其绝对值越小，编码之后使用越少的字节数表示，显然这里编码之后得到的结果和我们期望的结果相

5.2.2 ZigZag如何编码

很简单，两个公式就搞定了，没有复杂的编码转换。

```
zigzag32(n) = (n << 1) ^ (n >> 31)  //对于 sint32
zigzag64(n) = (n << 1) ^ (n >> 63)  //对于 sint64
```

一般情况下我们认为，使用较多的是小整数（确切地说应该是绝对值小的整数），那么较小的整数应使用更少的字节数来编码，ZigZag 编码正是如此，如下表格：

n	十六进制	zigzag(n)	varint(zigzag(n))
0	00 00 00 00	00 00 00 00	00
-1	ff ff ff ff	00 00 00 01	01
1	00 00 00 01	00 00 00 02	02
-2	ff ff ff fe	00 00 00 03	03
2	00 00 00 02	00 00 00 04	04
...	...	...	...
2147483647	7f ff ff ff	ff ff ff fe	ff ff ff fe
-2147483648	80 00 00 00	ff ff ff ff	ff ff ff ff

5.3 Length-delimited

这种编码很容易理解，就是 `length + body` 的方式，使用一个 Varint 类型表示 length，然后 length 的后面接着 length 个字节的内容。

```
/* message {
 *   string str = 14;
 * } */
static void test_1() {
  mytest::Test t1;
  t1.set_str("string"); //field_number = 14, wire_type = 5 (varint)
  std::string buf;
  t1.SerializeToString(&buf);
  dump_hexstring("==== test_1 ====", buf);
}

int main() {
  test_1();
  return 0;
}
```

编译和执行结果：

```
==== test_1 ====:
72 06 73 74 72 69 6e 67
```

分析结果：

```
72 06 73 74 72 69 6e 67
|  |  s  t  r  i  n  g
|  |  |__|__|__|__|__|__ body 的 ASCII 码
|  |__ length = 6 = 0x06
|__ Tag (field-number << 3 | wire-type) = (14 << 3 | 5) = 114 = 0x72
```

6 如何选型

测试 - 1

```
static void test_1() {
  mytest::Test t1;
  t1.set_i32(1);  //field_number = 1, wire_type = 0 (varint)
  t1.set_i64(2);  //field_number = 2, wire_type = 0 (varint)
  t1.set_u32(1);  //field_number = 3, wire_type = 0 (varint)
  t1.set_u64(2);  //field_number = 4, wire_type = 0 (varint)
  t1.set_si32(1); //field_number = 5, wire_type = 0 (varint)
  t1.set_si64(2); //field_number = 6, wire_type = 0 (varint)
  std::string buf;
  t1.SerializeToString(&buf);
  dump_hexstring("==== test_1 ====", buf);
}

int main() {
  test_1();
  return 0;
}
```

编译和执行结果：

```
==== test_1 ====:
08 01 10 02 18 01 20 02 28 02 30 04
```

分析结果：

```
08 01  |__ i32 字段
10 02  |__ i64 字段
18 01  |__ u32 字段
20 02  |__ u64 字段
28 02  |__ si32 字段
30 04  |__ si64 字段
```

测试 - 2

```
static void test_1() {
  mytest::Test t1;
  t1.set_i32(-1);  //field_number = 1, wire_type = 0 (varint)
  t1.set_i64(-2);  //field_number = 2, wire_type = 0 (varint)
  t1.set_u32(-1);  //field_number = 3, wire_type = 0 (varint)
  t1.set_u64(-2);  //field_number = 4, wire_type = 0 (varint)
  t1.set_si32(-1); //field_number = 5, wire_type = 0 (varint)
  t1.set_si64(-2); //field_number = 6, wire_type = 0 (varint)
  std::string buf;
  t1.SerializeToString(&buf);
  dump_hexstring("==== test_1 ====", buf);
}

int main() {
  test_1();
  return 0;
}
```

编译和执行结果：

```
==== test_1 ====:
08 ff ff ff ff ff ff ff ff ff 01 10 fe ff ff ff ff ff ff ff ff 01 18 ff ff ff ff 0f 20
```

分析结果：

```
08 ff ff ff ff ff ff ff ff ff 01  |___ i32 字段
10 fe ff ff ff ff ff ff ff ff 01  |___ i64 字段
18 ff ff ff ff 0f                |___ u32 字段
20 fe ff ff ff ff ff ff ff ff 01  |___ u64 字段
28 01                            |___ si32 字段
30 03                            |___ si64 字段
```

如何选型?

从编码步骤来看：

- int32/uint32/int64/uint64：直接进行 Varint 编码；
- sint32/sint64：先进行 ZigZag 编号，然后再对前者结果进行 Varint 编码，多了一个步骤。

从编码结果字节数来看：

- int32/int64：横向比较 int32 和 int64 编码结果一样，但是 int64 能够表示更大的数；
- uint32/uint64：横向比较 uint32 和 uint64 编码结果一样，但是 uint64 能够表示更大的数；
- sint32/sint64：横向比较 sint32 和 sint64 编码结果一样，但是 sint64 能够表示更大的数；
- 纵向比较：正数的时候，编码都一样，反而 sint32 和 sint64 多了一个步骤（ZigZag编码），但是负数的情况 sint32 和 sint64 使用的字节数较少。

综上所述，这样选型：

- 如果确定是正数：
- 如果数值确定小于等于 UINT32_MAX，可以用 uint32；
- 如果数值可能大于 UINT32_MAX，则可以用 uint64；
- 虽然序列化后结果一样，但是考虑到前者可能在内存分配上会少一点，这里说“可能”，是因为还和内存对齐有关系）。
- 如果可能是负数，其ZigZag编码之后确定是正数：
- 如果ZigZag编码后的值确定小于等于 INT32_MAX 且大于等于 INT32_MIN，可以用 sint32；
- 如果ZigZag编码后的值确定可能大于 INT32_MAX 或者 小于 INT32_MIN，则用 sint64；
- 虽然序列化后结果一样，但是考虑到前者可能在内存分配上会少一点，这里说“可能”，是因为还和内存对齐有关系）。

> 注：到这里，如果是对 protobuf 比较了解的读者，可能已经发现，以上少考虑了一种情况。因为 Varint 编码后的每个字节只有低 7 位表示 数据（最高位是 msb），那样的话，4 个字节能够表示的最大数为 2^28 - 1（不考虑符号），8 个字节能够表示的最大数为 2^56 - 1（不考虑符号）。 C++ 中的 uint32_t 可以表示的最大数为 2^32-1，uint64_t 可以表示的最大数为 2^64 - 1，那岂不是说在值 大于 2^28 - 1 或者 2^56 - 1 的情况下，其 Protobuf 编码后字节数还比对应的 C++ 类型的字节数还多了？ 在 「6.2 fixed32/sfixed32/fixed64/sfixed64」中就提供了解决方案。

6.2 fixed家族：fixed32/sfixed32/fixed64/sfixed64

fixed32/fixed64 分别对应 C++ 类型的 uint32_t 和 uint64_t，sfixed32/sfixed64 分别对应 C++ 类型的 int32_t 和 int64_t。没有经过任何编码，分别使用固定的 4 个字节 和 8 个字节表示该值。

sfixed32/sfixed64 也是分别使用了固定 4 个字节 和 8 个字节表示该值，只不过先经过 ZigZag 编码然后再存储。

综上所述，可以这样选型：

- 如果确定是正数：
- 如果数值确定小于等于 UINT32_MAX 且大于 2^28-1，可以用 fixed32（比如：这是一个表示时间戳的字段）；
- 如果数值确定可能大于 2^56-1，则可以用 fixed64（比如纯数字订单号：2021083011405200001，20210830（日期）+114052（时间）+ 00001（5 位系列号））
- 如果可能是负数，其 ZigZag 编码后确定是正数：

6.3 浮点家族：float/double

float 是使用固定 4 个字节来表示浮点数，double 是使用固定 8 个字节来表示浮点数。

有时候我们可以灵活一点，不一定需要用浮点数的类型类表示浮点数，比如有这样一个需求：使用一个字段来表示分数，满分一分，有效值扩展到小数点后 2 位小数（如：99.98分），如果使用 float 编码结果为 5 个字节（Tag 1个字节 + 固定 4 个字节）。

我们换一种思路，把分数转换成整型（如：9998），选型为 int32，那么使用的是 Varint 编码，最后的结果只需 3 个字节。

测试-1

```
/* message Test {
 *   int32 i32 = 1;
 *   float fl  = 12;
 * }
 * */
static void test_1() {
  mytest::Test t1;
  t1.set_i32(9998);     //field_number = 1, wire_type = 0 (varint)
  t1.set_fl(99.98);     //field_number = 12, wire_type = 5 (float)
  std::string buf;
  t1.SerializeToString(&buf);
  dump_hexstring("==== test_1 ====", buf);
}

int main() {
  test_1();
  return 0;
}
```

编译和执行结果：

```
==== test_1 ====:
08 8e 4e 65 c3 f5 c7 42
```

分析结果：

```
08 8e 4e        |__ i32
65 c3 f5 c7 42  |__ fl
```

double 也是按同样的思路分析，这里就不再细抠图。

6.4 字符串家族：string/bytes

string 和 bytes 都是字符串，使用了 Length-delimited 的编码方式，见「5.3 Length-delimited」。但是string会做 字符串编码检查，仅支持UTF-8编码或者7-bit ASCII编码的文本，而 bytes 可以是任意字符串。

6.5 序列：repeated

repeated 顾名思义，是重复这个字段，其主要是补充数组功能这块的空白，类似于 C++ 语言中的 vector。

repeated 使用了 Length-delimited 的编码方式，见「5.3 Length-delimited」。先看一下它的序列化模型：

```
[Tag] [Length] [Data-1][Data-2][Data-3]...[Data-n]
```

```
/* message Test {
 *   repeated int32 vec = 16;
 * } */
static void test_1() {
  mytest::Test t1;
  t1.add_vec(1);  //field_number = 16, wire_type = 2 (Length-Delimited)
  t1.add_vec(2);  //field_number = 16, wire_type = 2 (Length-Delimited)
  std::string buf;
  t1.SerializeToString(&buf);
  dump_hexstring("==== test_1 ====", buf);
}

int main() {
  test_1();
  return 0;
}
```

编译和执行结果：

```
==== test_1 ====:
82 01 02 01 02
```

分析结果：

```
82 01  |__ Tag
02     |__ Length
01 02  |__ 值 1 和 2
```

测试 - 2

```
/* message Test {
 *   repeated SubTest vec = 16;
 * } */
static void test_1() {
  mytest::Test t1;
  t1.add_vec()->set_i32(1);//field_number = 16, wire_type = 2 (Length-Delimited)
  t1.add_vec()->set_i32(2);//field_number = 16, wire_type = 2 (Length-Delimited)
  std::string buf;
  t1.SerializeToString(&buf);
  dump_hexstring("==== test_1 ====", buf);
}

int main() {
  test_1();
  return 0;
}
```

编译和执行结果：

```
==== test_1 ====:
82 01 02 08 01 82 01 02 08 02
```

分析结果：

```
82 01 02 08 01 //vec[0]
82 01  |__ Tag
02     |__ Length
08 01  |__ Subtest值, 08-> Tag, 01 -> 值

82 01 02 08 02 //vec[1]
82 01  |__ Tag
```

```
02     |__ Length
08 02  |__ Subtest值, 08-> Tag, 02 -> 值
```

这里笔者觉得很怪，为什么每个 repeated item 都要重复 Tag 和 Length 呢，这不是会增加无畏的字节码？

> ***问题***：不应该是这种方式编码后体积更小吗？为什么不用这种方法呢？

```
82 01 02 08 01 08 02
82 01  |__ Tag
02     |__ Length
08 01  |__ vec[0] SubTest值, 08 -> Tag, 01 -> 值
08 02  |__ vec[1] SubTest值, 08 -> Tag, 02 -> 值
```

这是因为如果 repeated 类型是基础类型（比如 Varint） 时，会做 packed 优化（也就是压缩）。

综上所述：

如果不是很必要，repeated 不要使用复杂的类型，就使用 Varint 的类型就可以了。

比如有这样一个需求，需要存储一个列表，列表的 item 包含两个字段，一个是 appid，一个是整形的 score。那么不建议使用这种：

```
message Item {
  int64 appid;
  int64 score;
}
message Test {
  repeated Item vec = 1;
}
```

可以使用下面这种（这种序列化知乎包体会小很多，vec size 越大，小得越明显）：

```
message Test {
  repeated int64 vec_appid = 1;
  repeated int64 vec_score = 2;
}
```

6.6 嵌套：embedding message

embedding message，也就是 message 中某个字段的类型是一个 message 的类型，使用了 Length-delimited 编码方式。

测试 - 1

```
/* message SubTest {
 *   int32 i32 = 1;
 * }
 * message Test {
 *   SubTest test = 18; //embedding message
 * } */
static void test_1() {
  mytest::Test t1;
  //field_number = 18, wire_type = 2 (Length-delimited)
  t1.mutable_test()->set_i32(1);
  std::string buf;
  t1.SerializeToString(&buf);
  dump_hexstring("==== test_1 ====", buf);
}
```

```
  test_1();
  return 0;
}
```

编译和执行结果：

```
==== test_1 ====:
92 01 02 08 01
```

分析结果：

```
92 01   |__ Tag
02      |__ Length
08 01   |__ Data, SubTest值, 08 -> Tag, 01 -> 值
```

6.7 映射：map

map 的底层实现是哈希表。类似 C++ 语言中的 unordered_map。

map 使用了 Length-delimited 编码方式。

测试-1

```
/* message Test {
 *   map<int32, int32>  mp = 17;
 * } */
static void test_1() {
  mytest::Test t1;
  //field_number = 17, wire_type = 2 (Length-Delimited)
  t1.mutable_mp()->insert({1, 10});
  t1.mutable_mp()->insert({2, 11});
  t1.mutable_mp()->insert({3, 12});
  std::string buf;
  t1.SerializeToString(&buf);
  dump_hexstring("==== test_1 ====", buf);
}
```

编译和执行结果：

```
==== test_1 ====:
8a 01 04 08 01 10 0a 8a 01 04 08 02 10 0b 8a 01 04 08 03 10 0c
```

分析结果：

```
8a 01  04 08 01 10 0a  |__ Key-Value-Group[0]
 |__|   |  |__|  |__|_____ Value (10-> Tag ①, 0a -> 值)
    |   |     |____________ Key   (08-> Tag ②, 01 -> 值)
    |   |__________________ Length
    |______________________ Tag

    ①: 10 = 2 << 3 | 0 = 16 = 0x10
       (map的value field-number 固定为 2)
    ②: 08 = 1 << 3 | 0 = 8 = 0x08
       (map的Key field-number 固定为 1)

 8a 01  04 08 02 10 0b  |__ Key-Value-Group[1]
 8a 01  04 08 03 10 0c  |__ Key-Value-Group[2]
```

> 注：值得注意的是每一组 key-value 都会带上 `8a 01` 这个 Tag，以及 `04` 这个 Length。

> 源文件：include/google/protobuf/any.proto

```
syntax = "proto3";

package google.protobuf;

option csharp_namespace = "Google.Protobuf.WellKnownTypes";
option go_package = "google.golang.org/protobuf/types/known/anypb";
option java_package = "com.google.protobuf";
option java_outer_classname = "AnyProto";
option java_multiple_files = true;
option objc_class_prefix = "GPB";

message Any {
  string type_url = 1;
  bytes value = 2;
}
```

去掉注释之后，也就一个 message，type_url 用来存储类型描述信息，value 用来存储序列化（C++ 是SerializeToString 函数）之后的字符串。

```
/* message SubTest {
 *   int32 i32 = 1;
 * }
 * message Test {
 *   google.protobuf.Any any = 21;
 * } */
static void test_1() {
  mytest::SubTest st1;
  st1.set_i32(1);
  mytest::Test t1;
  t1.mutable_any()->PackFrom(st1);  //field_number = 21, wire_type = 2 (Lenth-Delimite
  //std::cout << t1.any().type_url() << std::endl;
  std::string buf;
  t1.SerializeToString(&buf);
  dump_hexstring("==== test_1 ====", buf);
}

int main() {
  test_1();
  return 0;
}
```

编译和执行结果：

```
==== test_1 ====:
aa 01 28 0a 22 74 79 70 65 2e 67 6f 6f 67 6c 65 61 70 69 73 2e 63 6f 6d 2f 6d 79 74 65
```

分析结果：

```
aa 01           |__ Tag
28              |__ Length (40个字符)
0a 22 74 79 70 65 2e 67 6f 6f 67 6c 65 61 70 69 73 2e 63 6f 6d 2f 6d 79 74 65 73 74 2e
12 02 08 01     |__ 第二个字段（bytes value = 2）

第一个字段（string type_url）：
0a              |__ Tag
22              |__ Length (34个字符)
74 79 70 65 2e 67 6f 6f 67 6c 65 61 70 69 73 2e 63 6f 6d 2f 6d 79 74 65 73 74 2e 53 75
t  y  p  e  .  g  o  o  g  l  e  a  p  i  s  .  c  o  m  /  m  y  t  e  s  t  .  S  u
```

```
12              |__ Tag
02              |__ Length
08 01           |__ mytest::SubTest的编码: 08 -> Tag, 01 -> 值
```

> 注：Any 使用到了reflecttion（反射）功能，C++ 编译链接时不能使用 -lprotobuf-lite，而要使用 -lprotobuf。相关介绍请见 「7 可选项 optimize_for」。

6.9 oneof

如果需求有一条包含许多字段的消息，并且最多同时设置一个字段，那么可以使用 oneof 特性来节省内存。

oneof 字段类似于常规字段，除了 oneof 共享内存的所有字段之外，最多可以同时设置一个字段。设置 oneof 的任何成员都会自动清除所有其他成员。可以使用 case() 或 WhichOneof()方法检查 oneof 中的哪个值被设置(如果有的话)，具体取决于您选择的语言。

oneof 不能使用 repeated 字段。

测试-1：

```
/* message Test {
 *   oneof object {
 *     float   one_fl  = 19;
 *     string  one_str = 20;
 *   }
 * } */
static void test_1() {
  mytest::SubTest st1;
  st1.set_i32(1);
  mytest::Test t1;
  t1.set_one_fl(0.1);
  std::cout << "one_str:" << t1.one_str() << ", one_fl:" << t1.one_fl() << std::endl;
  t1.set_one_str("string");
  std::cout << "one_str:" << t1.one_str() << ", one_fl:" << t1.one_fl() << std::endl;
  std::string buf;
  t1.SerializeToString(&buf);
  dump_hexstring("==== test_1 ====", buf);
}

int main() {
  test_1();
  return 0;
}
```

编译和执行结果：

```
one_str:, one_fl:0.1
one_str:string, one_fl:0
==== test_1 ====:
a2 01 06 73 74 72 69 6e 67
```

分析结果：

```
one_str:, one_fl:0.1
one_str:string, one_fl:0
| 设置 one_str 的时候，one_fl被清空了，
| 操作（触发内存分配如执行set_xxx或者mutable_xxxx函数）
| 任何一个成员的时候，会清空所有其他成员。
==== test_1 ====:
```

```
|  |_____________________________ Length
|_____________________________ Tag
```

7 可选项 optimize_for

```
syntax = "proto3";
option optimize_for = SPEED; //SPEED（默认）、CODE_SIZE、LITE_RUNTIME
```

7.1 SPEED

表示生成的代码运行效率高，但是由此生成的代码编译后会占用更多的空间。

7.2 CODE_SIZE

和SPEED恰恰相反，代码运行效率较低，但是由此生成的代码编译后会占用更少的空间，通常用于资源有限的平台，如移动设备。

7.3 LITE_RUNTIME

生成的代码执行效率高，同时生成代码编译后的所占用的空间也是非常少，但是它会缺少一些属性像 reflection（反射）功能。在 C++ 中依赖 libprotobuf-lite 库，而非 libprotobuf 库。

```
$ cd /usr/local/Cellar/protobuf/3.17.3/lib/
$ ls -lh *.a
-r--r--r--  1 baron  admin   835K Jun  8 22:15 libprotobuf-lite.a
-r--r--r--  1 baron  admin   4.1M Jun  8 22:15 libprotobuf.a
...
```

> 注：看 lite 库的体积大小仅仅 非lite库的 1/5 ~ 1/4 左右。

8 附录

8.1 补码编码

我们先来看一下三个概念：源码、反码、补码

- 原码：最高位为符号位，剩余位表示绝对值；
- 反码：除符号位外，对原码剩余位依次取反；
- 补码：正数补码为其自身，负数补码为除符号位外对原码剩余位依次取反然后加1。

如果计算机存储正数时，存储的是原码：

- 数字 0 的表示

```
正数0：  [0000 0000]原
负数0：  [1000 0000]原
```

- 原码中还存在加法错误的问题

```
1 + (-1) = [0000 0001]原 + [1000 0001]原 = [1000 0010]原 = -2
```

如果存储的是补码呢？

```
正数0 = 负数0 = [0000 0000]补
```

没错，计算机存储整数时采用的是补码。

此外，整数的补码有一些有趣的性质：

- 左移 1 位（n << 1），无论正数还是负数，相当于乘以 2；对于正数，若大于MAX_INT/2（1076741823），则会发生溢出，导致左移1位后为负数
- 右移 31 位（n >> 31），对于正数，则返回0x00000000；对于负数，则返回0xffffffff。

9 参考文献

Language Guide (proto3) | Protocol Buffers | Google Developer

测试相关源码位置：github.com/sullivan1205...

10 说在最后

以上就是系列文章第二篇的所有内容。通过本文，读者应该已经了解 protobuf 各个数据类型是如何编码的，从而可以推算出其编码效率，以及压缩率，为能够选择最优的数据类型提供可靠的参考。按计划下一篇将分享反射的原理。

感谢阅读，如果还想了解更多的内容，请在评论区留言。

欢迎学习交流，也欢迎指正。

发布于 2022-12-11 17:04 · IP 属地广东

内容所属专栏

protobuf