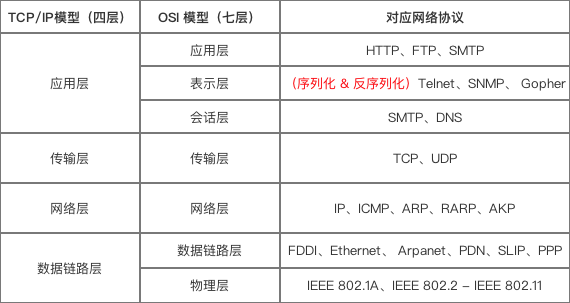
**Protocol Buffer 序列化原理**

# 1. 知识基础

### 1.1 网络通信协议

* 序列化 & 反序列化 属于通讯协议的一部分
* 通讯协议采用分层模型：TCP/IP模型（四层） & OSI 模型 （七层）



* **序列化 / 反序列化 属于 TCP/IP模型 应用层 和 OSI`模型 展示层的主要功能：**
  1. **（序列化）把 应用层的对象 转换成 二进制串**
  2. **（反序列化）把 二进制串 转换成 应用层的对象**
* 所以， Protocol Buffer属于 TCP/IP模型的应用层 & OSI模型的展示层

### 1.2 数据结构、对象与二进制串

不同的计算机语言中，数据结构，对象以及二进制串的表示方式并不相同。

#### a. 对于数据结构和对象

* 对于面向对象的语言（如Java）：对象 = Object = 类的实例化；在Java中最接近数据结构 即 POJO（Plain Old Java Object），或Javabean（只有 setter/getter 方法的类）
* 对于半面向对象的语言（如C++），对象 = class，数据结构 = struct

#### b. 二进制串

* 对于C++，因为具有内存操作符，所以 二进制串 容易理解：C++的字符串可以直接被传输层使用，因为其本质上就是以 '\0' 结尾的存储在内存中的二进制串

#### 1.3 T - L - V 的数据存储方式

* 定义  
  即 Tag - Length - Value，标识 - 长度 - 字段值 存储方式
* 作用  
  以 标识 - 长度 - 字段值 表示单个数据，最终将所有数据拼接成一个 字节流，从而 实现 数据存储 的功能

其中 Length可选存储，如 储存Varint编码数据就不需要存储Length

* 示意图



# 2. 序列化原理解析

请记住Protocol Buffer的 三个关于数据存储 的重要结论：

* 结论1： Protocol Buffer将 消息里的每个字段 进行编码后，再利用T - L - V 存储方式 进行数据的存储，最终得到的是一个 二进制字节流

序列化 = 对数据进行编码 + 存储

* 结论2：Protocol Buffer对于不同数据类型 采用不同的 序列化方式（编码方式 & 数据存储方式），如下图：  
  

从上表可以看出：

1. 对于存储Varint编码数据，就不需要存储字节长度 Length，所以实际上Protocol Buffer的存储方式是 T - V；
2. 若Protocol Buffer采用其他编码方式（如LENGTH\_DELIMITED）则采用T - L - V

* 结论3：因为 Protocol Buffer对于数据字段值的 独特编码方式 & T - L - V数据存储方式，使得 Protocol Buffer序列化后数据量体积如此小

【注释】 编码过程中字段问题：

TLV 2 的21次方-1 4个字节

2 的 21 地方 5个字节

大于2的21次方的采用fixed方式

### 2.1 Wire Type = 0时的编码 & 数据存储方式



#### 2.1.1 编码方式： Varint & Zigzag

#### A. Varint编码方式介绍

#### i. 简介

* 定义：一种变长的编码方式
* 原理：用字节 表示 数字：值越小的数字，使用越少的字节数表示
* 作用：通过减少 表示数字 的字节数 从而进行数据压缩

如：

* 对于 int32 类型的数字，一般需要 4个字节 表示；

1. 若采用 Varint编码，对于很小的 int32 类型 数字，则可以用 1个字节 来表示
2. 虽然大的数字会需要 5 个 字节 来表示，但大多数情况下，消息都不会有很大的数字，所以采用 Varint方法总是可以用更少的字节数来表示数字

#### ii. 原理介绍

* 源码分析

private void writeVarint32(int n) {

int idx = 0;

while (true) {

if ((n & ~0x7F) == 0) {

i32buf[idx++] = (byte)n;

break;

} else {

i32buf[idx++] = (byte)((n & 0x7F) | 0x80);

// 步骤1：取出字节串末7位

// 对于上述取出的7位：在最高位添加1构成一个字节

// 如果是最后一次取出，则在最高位添加0构成1个字节

n >>>= 7;

// 步骤2：通过将字节串整体往右移7位，继续从字节串的末尾选取7位，直到取完为止。

}

}

trans\_.write(i32buf, 0, idx);

// 步骤3： 将上述形成的每个字节 按序拼接 成一个字节串

// 即该字节串就是经过Varint编码后的字节

}

从上面可看出：Varint 中每个 字节 的最高位 都有特殊含义：

* 如果是 1，表示后续的 字节 也是该数字的一部分
* 如果是 0，表示这是最后一个字节，且剩余 7位 都用来表示数字

所以，当使用Varint解码时时，只要读取到最高位为0的字节时，就表示已经是Varint的最后一个字节

因此：

* 小于 128 的数字 都可以用 1个字节 表示；
* 大于 128 的数字，比如 300，会用两个字节来表示：10101100 00000010

下面，两个个例子来说明Varint编码方式的使用

* 目的：对 数据类型为Int32 的 字段值为296 和字段值为104 进行Varint编码
* 以下是编码过程



从上面可以看出：

* 对于 int32 类型的数字，一般需要 4 个字节 来表示；
* 但采用 Varint 方法，对于很小的 Int32 类型 数字（小于256），则可以用 1个字节 来表示；

以此类推，比如300也只需要2个字节

* 虽然大的数字会需要 5 个字节 来表示，但大多数情况下，消息都不会有很大的数字
* 所以采用 Varint 方法总是可以用更少的字节数来表示数字，从而更好地实现数据压缩

## Varint 编码方式的不足

* 背景：在计算机内，负数一般会被表示为很大的整数

因为计算机定义负数的符号位为数字的最高位

* 问题：如果采用 Varint编码方式 表示一个负数，那么一定需要 5 个 byte（因为负数的最高位是1，会被当做很大的整数去处理）
* 解决方案： Protocol Buffer 定义了 sint32 / sint64 类型表示负数，通过先采用 Zigzag 编码（将 有符号数 转换成 无符号数），再采用 Varint编码，从而用于减少编码后的字节数  
  

1. 对于int32 / int64 类型的字段值（正数），Protocol Buffer直接采用 Varint编码
2. 对于sint32 / sint64 类型的字段值（负数），Protocol Buffer会先采用 Zigzag 编码，再采用 Varint编码

* 总结：为了更好地减少 表示负数时 的字节数，Protocol Buffer在 Varint编码上又增加了Zigzag 编码方式进行编码
* 下面将详细介绍 Zigzag编码方式

#### B. Zigzag编码方式详解

#### i. 简介

* 定义：一种变长的编码方式
* 原理：使用 无符号数 来表示 有符号数字；
* 作用：使得绝对值小的数字都可以采用较少 字节 来表示；

特别是对 表示负数的数据 能更好地进行数据压缩

#### b. 原理

* 源码分析

public int int\_to\_zigzag(int n)

// 传入的参数n = 传入字段值的二进制表示（此处以负数为例）

// 负数的二进制 = 符号位为1，剩余的位数为 该数绝对值的原码按位取反；然后整个二进制数+1

{

return (n <<1) ^ (n >>31);

// 对于sint 32 数据类型，使用Zigzag编码过程如下：

// 1. 将二进制表示数 左移1位（左移 = 整个二进制左移，低位补0）

// 2. 将二进制表示数 右移31位

// 对于右移：

// 首位是1的二进制（有符号数），是算数右移，即右移后左边补1

// 首位是0的二进制（无符号数），是逻辑左移，即右移后左边补0

// 3. 将上述二者进行异或

// 对于sint 64 数据类型 则为： return (n << 1> ^ (n >> 63) ；

}

// 附：将Zigzag值解码为整型值

public int zigzag\_to\_int(int n)

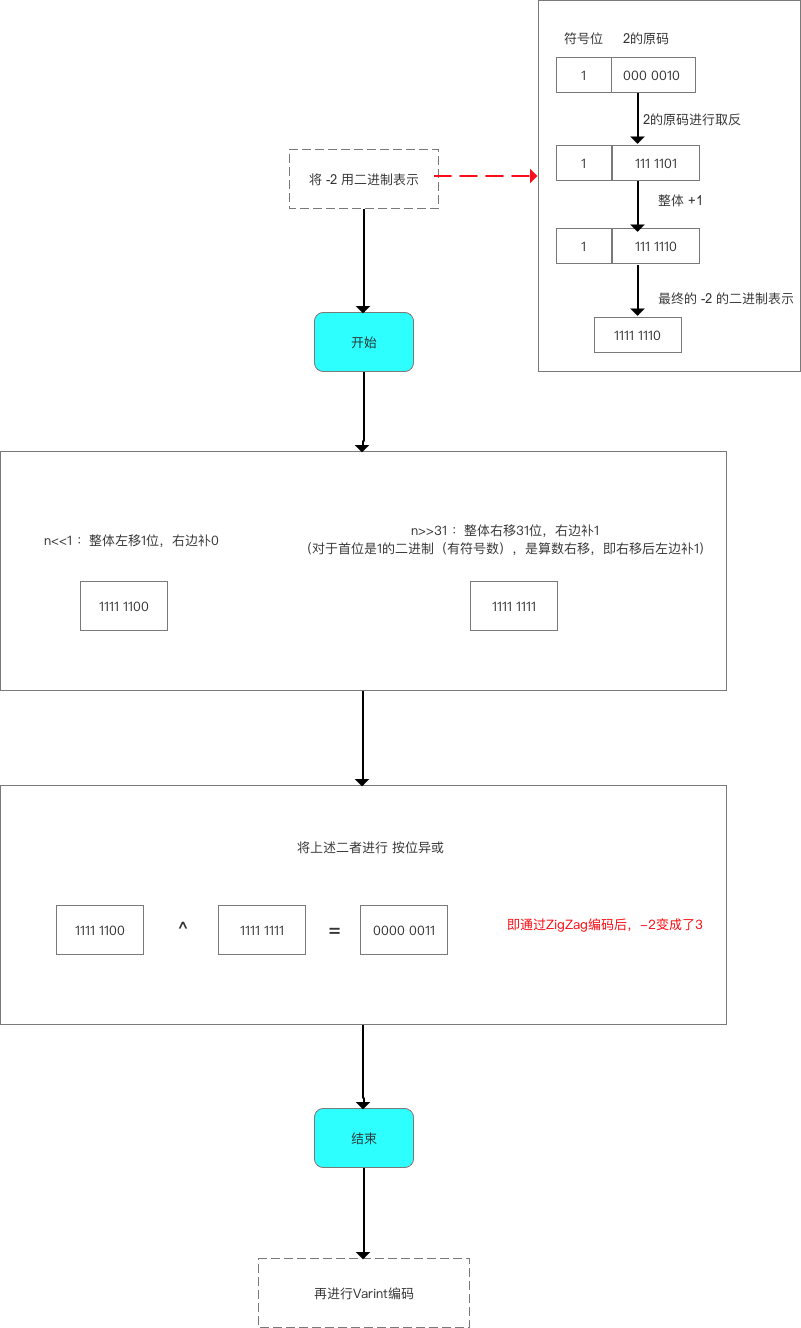
{

return(n >>> 1) ^ -(n & 1);

// 右移时，需要用不带符号的移动，否则如果第一位数据位是1的话，就会补1

}

* 实例说明：将 -2进行 Zigzag编码：



* Zigzag 编码 是补充 Varint编码在 表示负数 的不足，从而更好的帮助 Protocol Buffer进行数据的压缩
* 所以，如果提前预知字段值是可能取负数的时候，记得采用sint32 / sint64 数据类型

#### 总结

Protocol Buffer 通过Varint和Zigzag编码后大大减少了字段值占用字节数。

#### 2.1.2 存储方式:T - V

* 消息字段的标识号、数据类型 & 字段值经过 Protocol Buffer采用 Varint & Zigzag 编码后，以 T - V 方式进行数据存储

对于 Varint & Zigzag 编码，省略了T - L - V中的字节长度Length



下面将详细介绍T - V 存储方式中的存储细节：Tag & Value

#### 1. Tag

* 定义：经过 Protocol Buffer采用Varint & Zigzag编码后 的消息字段 标识号 & 数据类型 的值
* 作用：标识 字段

1. 存储了字段的标识号（field\_number）和 数据类型（wire\_type），即Tag = 字段数据类型（wire\_type） + 标识号（field\_number）
2. 占用 一个字节 的长度（如果标识号超过了16，则占用多一个字节的位置）
3. 解包时，Protocol Buffer根据 Tag 将 Value 对应于消息中的 字段

* 具体使用

// Tag 的具体表达式如下

Tag = (field\_number << 3) | wire\_type

// 参数说明：

// field\_number：对应于 .proto文件中消息字段的标识号，表示这是消息里的第几个字段

// field\_number << 3：表示 field\_number = 将 Tag的二进制表示 右移三位 后的值

// field\_num左移3位不会导致数据丢失，因为表示范围还是足够大地去表示消息里的字段数目

// wire\_type：表示 字段 的数据类型

// wire\_type = Tag的二进制表示 的最低三位值

// wire\_type的取值

enum WireType {

WIRETYPE\_VARINT = 0,

WIRETYPE\_FIXED64 = 1,

WIRETYPE\_LENGTH\_DELIMITED = 2,

WIRETYPE\_START\_GROUP = 3,

WIRETYPE\_END\_GROUP = 4,

WIRETYPE\_FIXED32 = 5

};

// 从上面可以看出，`wire\_type`最多占用 3位 的内存空间（因为 3位 足以表示 0-5 的二进制）

// 以下是 wire\_type 对应的 数据类型 表



* 实例说明

// 消息对象

message person

{

required int32 id = 1;

// wire type = 0，field\_number =1

required string name = 2;

// wire type = 2，field\_number =2

}

// 如果一个Tag的二进制 = 0001 0010

标识号 = field\_number = field\_number << 3 =右移3位 = 0000 0010 = 2

数据类型 = wire\_type = 最低三位表示 = 010 = 2

### 2. Value

经过 Protocol Buffer采用Varint & Zigzag编码后 的消息字段的值

#### 2.1.3 实例说明

下面通过一个实例进行整个编码过程的说明：

* 消息说明

message Test

{

required int32 id1 = 1；

required int32 id2 = 2；

}

// 在代码中给id1 附上1个字段值：296

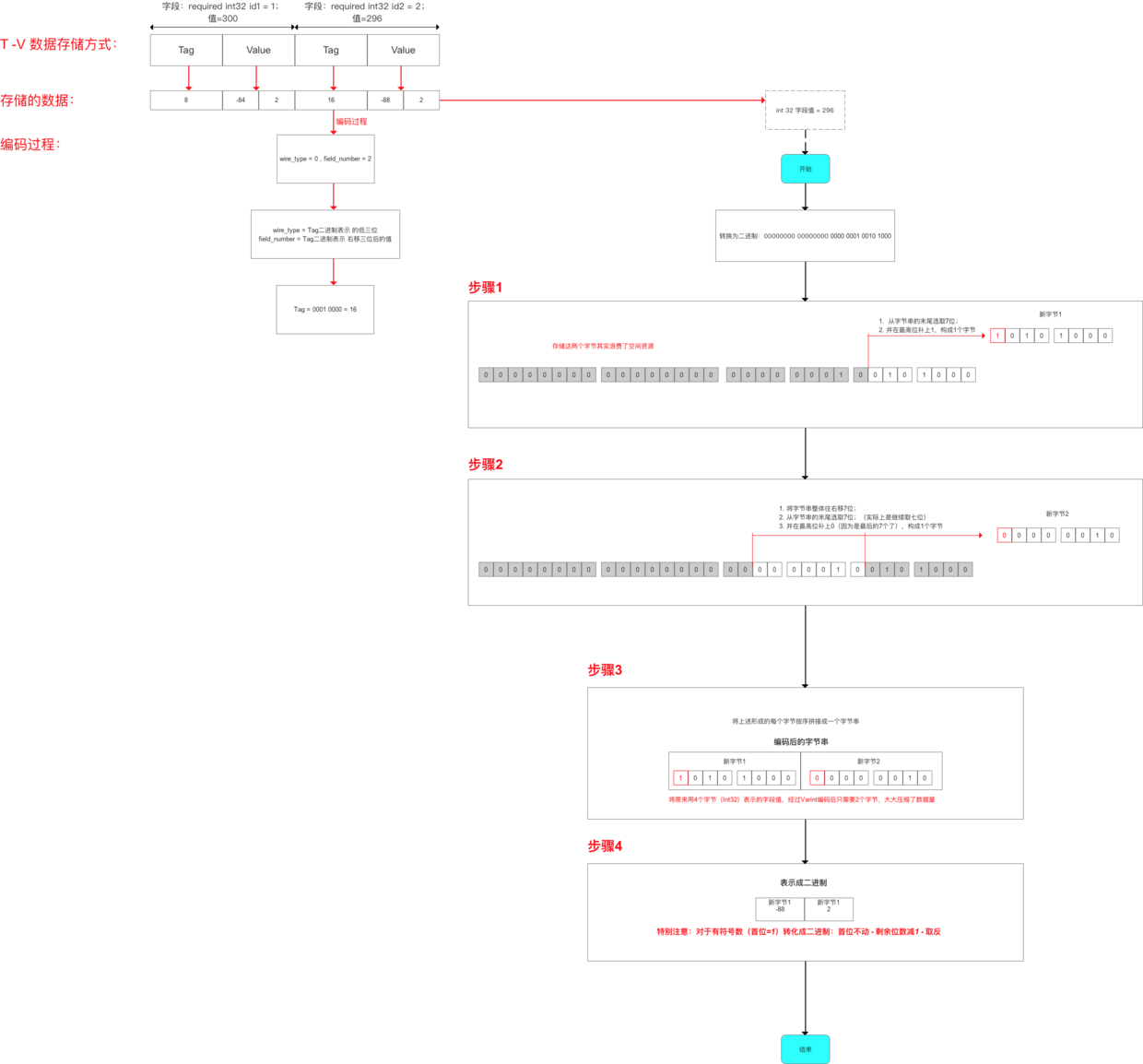
// 在代码中给id2 附上1个字段值：296

Test.setId1（300）；

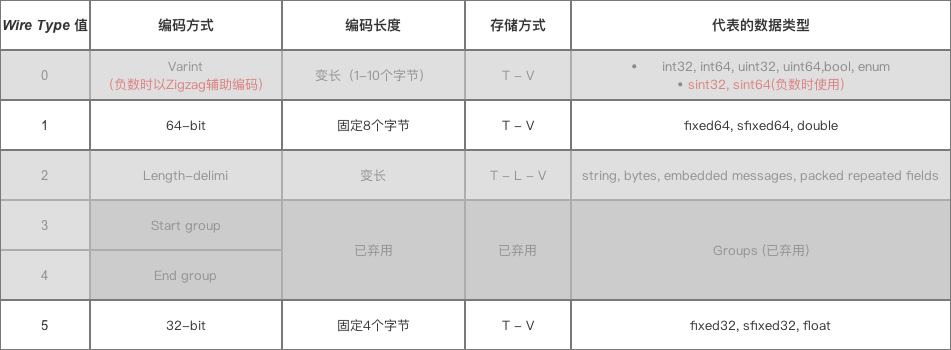
Test.setId2（296）；

// 编码结果为：二进制字节流 = [8，-84，2，16, -88, 2]

* 整个编码过程如下



### 2.2 Wire Type = 1& 5时的编码&数据存储方式



* 64（32）-bit编码方式较简单：编码后的数据具备固定大小 = 64位（8字节） / 32位（4字节）

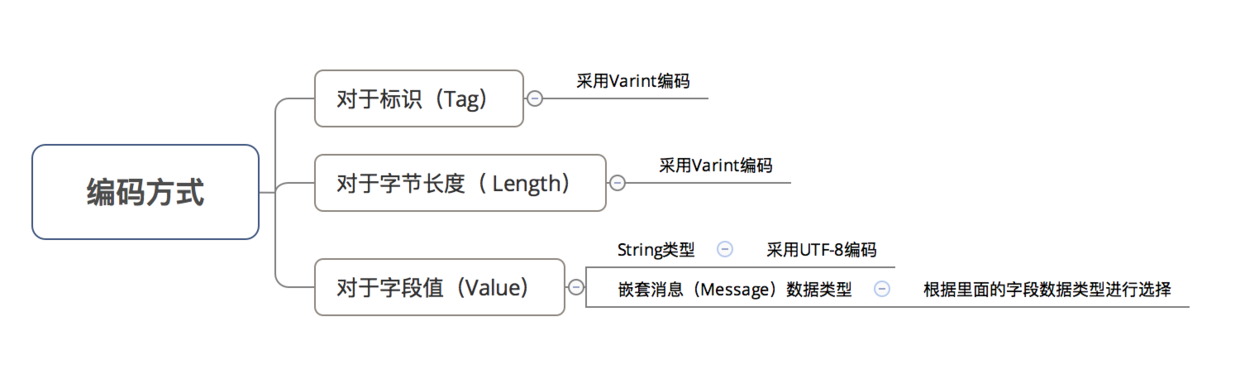
两种情况下，都是高位在后，低位在前

* 采用T - V方式进行数据存储，同上。

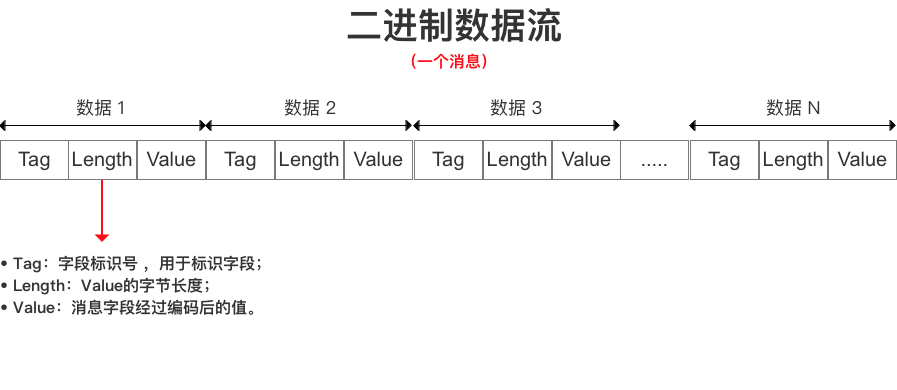
### 2.3 Wire Type = 2时的 编码 & 数据存储方式



* 对于编码方式：



* 数据存储方式： T - L - V



此处主要讲解三种数据类型：

* String类型
* 嵌套消息类型（Message）
* 通过packed修饰的 repeat 字段（即packed repeated fields）

### 1. String类型

字段值（即V） 采用UTF-8编码



* 例子：

message Test2

{

required string str = 2;

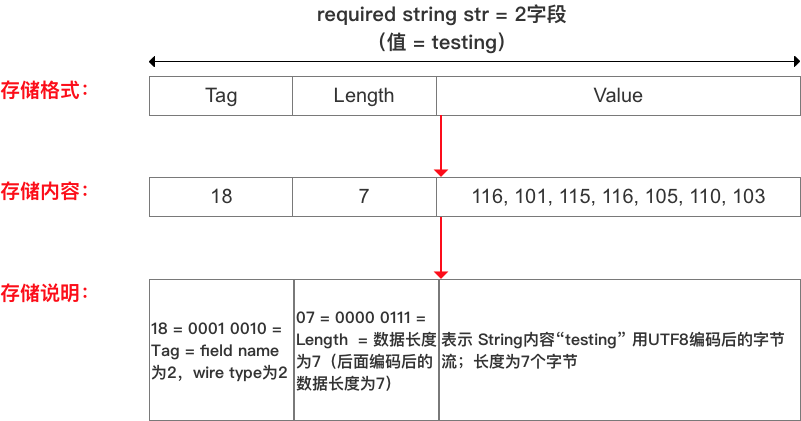
}

// 将str设置为：testing

Test2.setStr（“testing”）

// 经过protobuf编码序列化后的数据以二进制的方式输出

// 输出为：18, 7, 116, 101, 115, 116, 105, 110, 103



### 2. 嵌套消息类型（Message）

* 存储方式：T - L - V

1. 外部消息的 V即为 嵌套消息的字段
2. 在 T - L -V 里嵌套了一系列的 T - L -V

* 编码方式：字段值（即V） 根据字段的数据类型采用不同编码方式



* 实例  
  定义如下嵌套消息：

message Test2

{

required string str = 1;

required int32 id1 = 2；

}

message Test3 {

required Test2 c = 1;

}

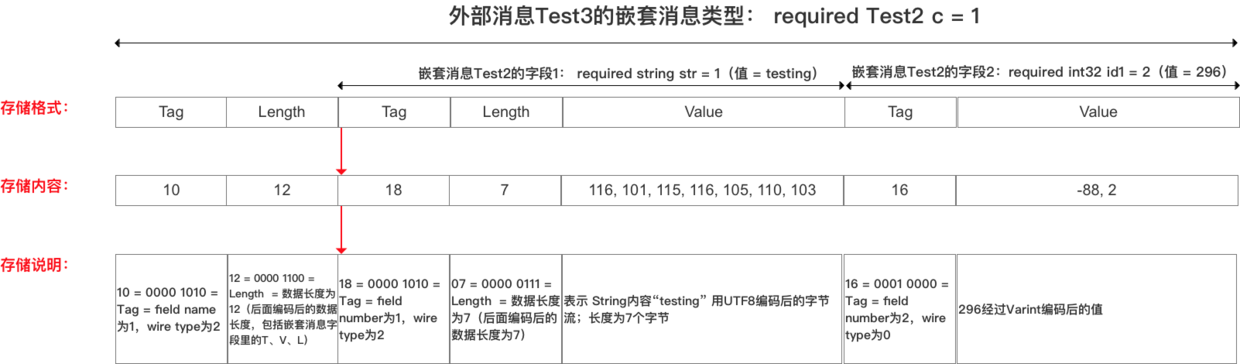
// 将Test2中的字段str设置为：testing

// 将Test2中的字段id1设置为：296

// 编码后的字节为：10 ，12 ，18，7，116, 101, 115, 116, 105, 110, 103，16，-88，2

1

编码 & 存储方式如下



### 3. 通过packed修饰的 repeat 字段

repeated 修饰的字段有两种表达方式：

message Test

{

repeated int32 Car = 4 ;

// 表达方式1：不带packed=true

repeated int32 Car = 4 [packed=true];

// 表达方式2：带packed=true

// proto 2.1 开始可使用

// 区别在于：是否连续存储repeated类型数据

}

// 在代码中给`repeated int32 Car`附上3个字段值：3、270、86942

Test.setCar（3）；

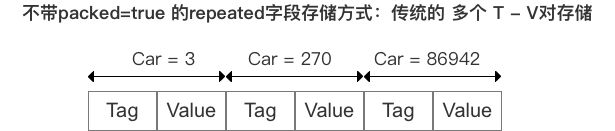
Test.setCar（270）；

Test.setCar（86942）；

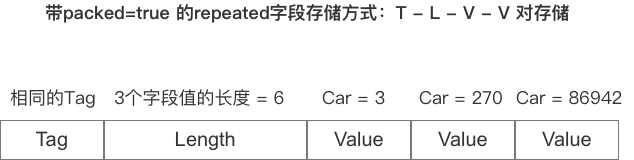
* 背景：对于同一个 repeated字段、多个字段值来说，他们的Tag都是相同的，即数据类型 & 标识号都相同

repeated类型可以看成是数组

* 问题：若以传统的多个 T - V对存储（不带packed=true），则会导致Tag的冗余，即相同的Tag存储多次；



* 解决方案：采用带packed=true 的 repeated 字段存储方式，即将相同的 Tag 只存储一次、添加 repeated 字段下所有字段值的长度Length、连续存储 repeated 字段值，组成一个大的Tag - Length - Value -Value -Value对，即T - L - V - V - V对。



通过采用带packed=true 的 repeated 字段存储方式，从而更好地压缩序列化后的数据长度。

#### 特别注意

* Protocol Buffer 的 packed修饰只用于repeated字段 或 基本类型的repeated字段
* 用在其他字段，编译 .proto 文件时会报错