ProtoBuf（Google Protocol Buffers）

> 编码结构简介（Varint 、ZigZag）

protobuf(Google Protocol Buffers)是Google提供一个具有高效的协议数据交换格式工具库(类似Json)。

- 时间开销小；
- 空间开销小；
- 支持多种编程语言（C++、java、python）
- 二进制格式导致可读性差

背景

- 序列化： 将数据结构或对象转换成能够被存储和传输（例如网络传输）的格式，同时应当要保证这个序列化结果在之后（可能是另一个计算环境中）能够被重建回原来的数据结构或对象。

- Xml、Json是目前常用的数据交换格式，它们直接使用**字段名称**维护序列化后类实例中字段与数据之间的映射关系，一般用**字符串的形式**保存在序列化后的字节流中。**消息和消息的定义相对独立，可读性较好。**但序列化后的数据**字节很大，序列化和反序列化的时间较长**，数据传输效率不高。

- Protobuf和Xml、Json序列化的方式不同，采用了**二进制字节的序列化方式**，用**字段索引**和**字段类型**通过算法计算得到字段之前的**关系映射**，从而达到更高的时间效率和空间效率，特别适合对数据大小和传输速率比较敏感的场合使用。

以一个数字的序列化为例：

> JSON:{"id":42}，9 bytes
>
> xml:42，11 bytes 。一般还需要外层包裹实现。
>
> protobuf:0x08 0x2A，2 bytes 0x08 = field 1, type ： Variant 0x2A = 42 (raw) or 21 (zigzag)

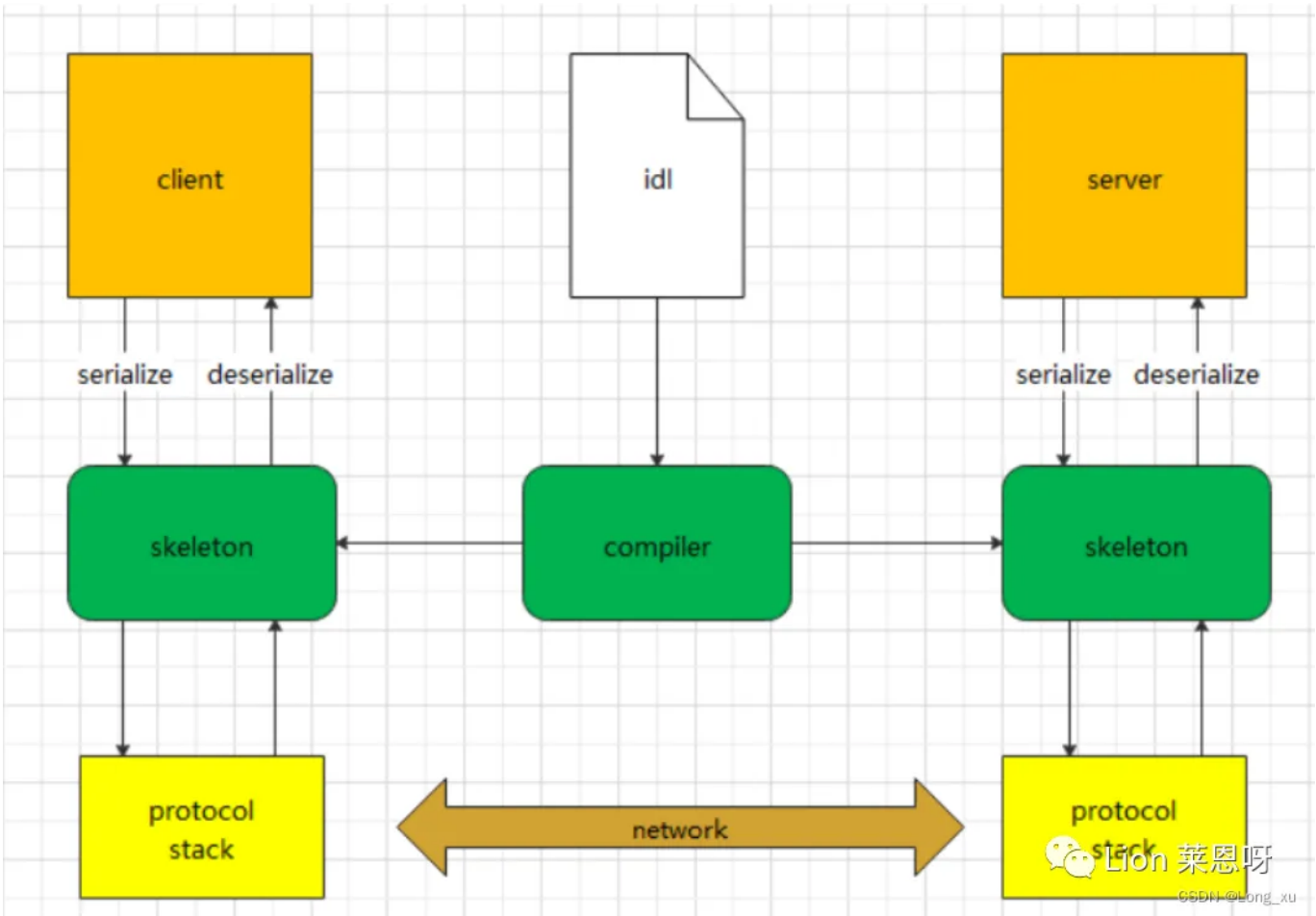
结构和使用

要使用protobuf序列化方式，要先编写proto文件。

```
syntax="proto3";                    // 版本，proto2和proto3
package IM.Login;                   // 类似CPP的命名空间
import "IM.BaseDefine.proto";       // 引用其他的proto文件
option optimize_for = LITE_RUNTIME;    // 编译优化

// 一个类
message IMLoginReq{
    // 各种字段
    string user_name=1;
    string password=2;
    IM.BaseDefine.UserStatType online_status=3;
    IM.BaseDefine.ClientType client_type=4;
    string client_version=5;
}
```

proto文件在发送端和接收端是公用的，及发送端和接收端使用的是同样的proto文件。

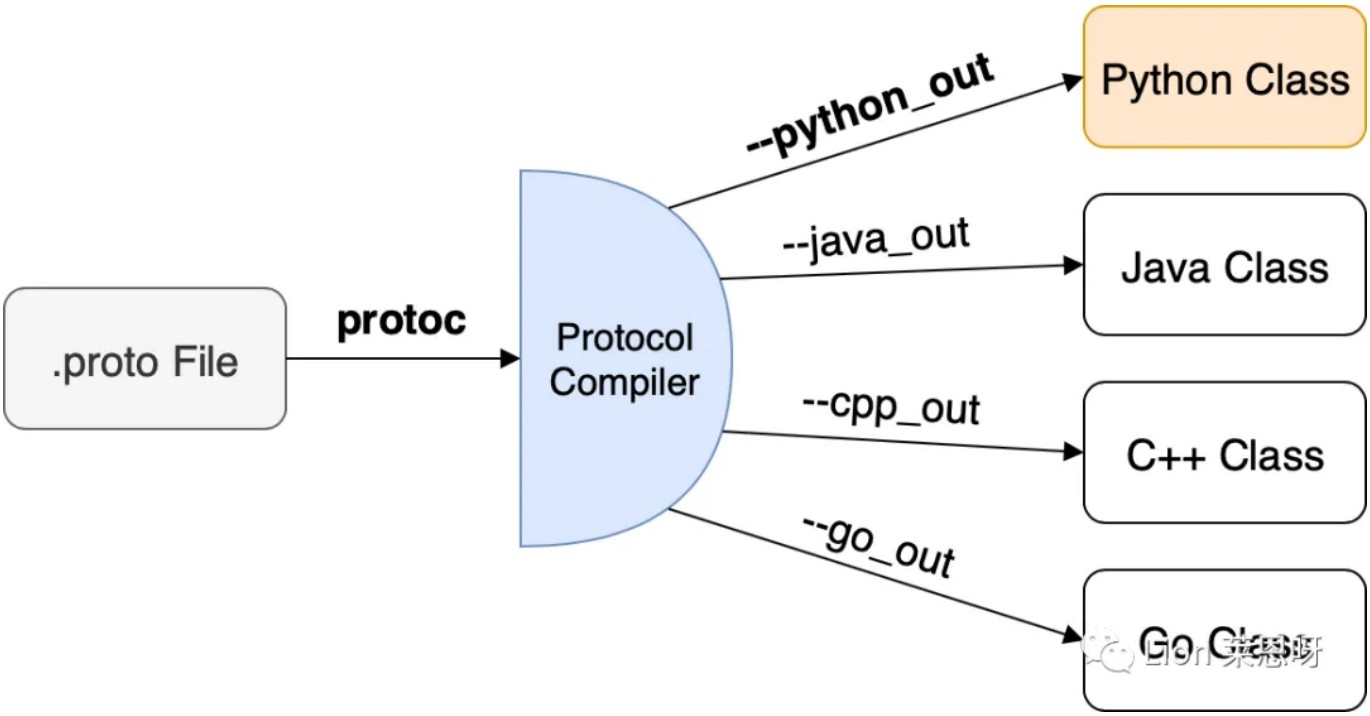


IDL是Interface description language的缩写，指接口描述语言。

可以看到，对于序列化协议来说，使用方只需要关注业务对象本身，即IDL定义（.proto），序列化和反序 列化的代码只需要通过工具生成即可。

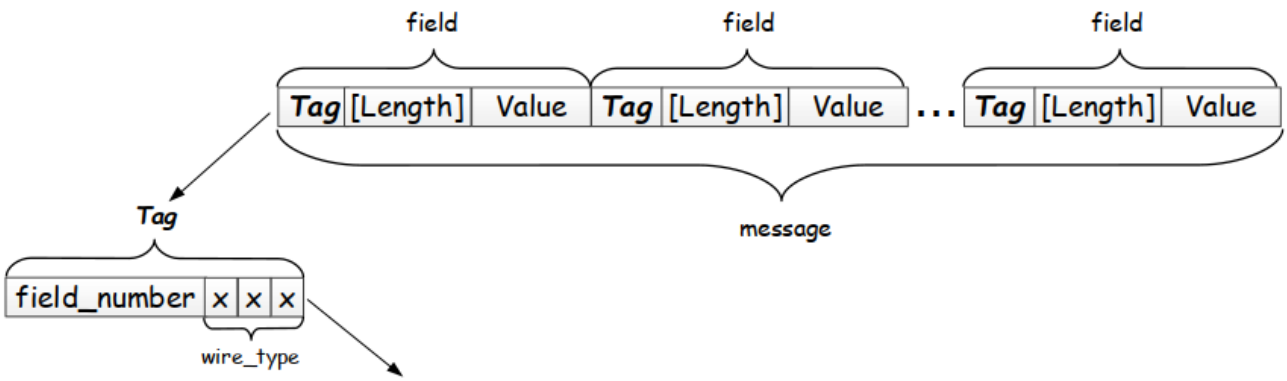
protobuf不能完全替代json，比如对外注册，json只需要把格式提供给对方，而protobuf还需要一些复杂的流程，会降低可读性。

同一个proto文件可以生成不同的语言。



Protobuf 提供了C++、java、python语言的支持，提供了**windows(proto.exe)** 和linux平台动态编译生成 **proto文件** 对应的源文件。

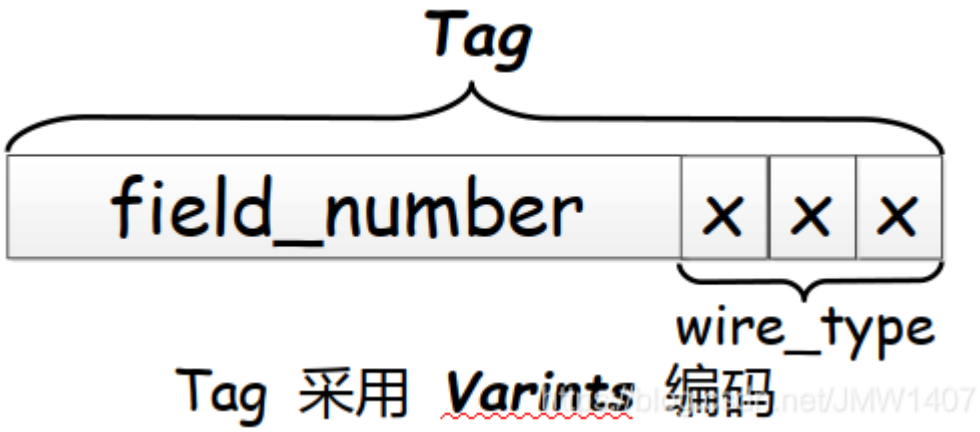
proto文件定义了协议数据中的**实体结构** `(message ,field)`



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 messages, packed repeated fields
3	Start group	groups (deprecated)
4	End group	groups (deprecated)
5	32-bit	fixed32, sfixed32, float

1、关键字message: 代表了实体结构，由多个消息字段**field**组成。通过 **key-value**对来表示，其实它是把 message 转成一系列的 key-value，

key 就是字段号，value 就是字段值，具体保存的时候实际保存的不是 key 而是 **tag**，key 字段号需要根据一个公式计算出 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（见下图），根据这个类型选择不同的 Value 编码方案。
    - wire_type由三位bit构成，故能表示8种类型。
    - 当wire_type等于0的时候整个二进制结构为：Tag-Value value的编码也采用Varints编码方式，故不需要额外的位来表示整个value的长度。因为Varint的msb位标识下一个字节是否是有效的就起到了指示长度的作用。
    - 当wire_type等于1、5的时候整个二进制结构也为：Tag-Value 因为都是取固定32位或者64位，因此也不需要额外的位来表示整个value的长度。
    - 当wire_type等于2的时候整个二进制结构为：Tag-[Length]-Value 因为表示的是可变长度的值，需要有额外的位来指示长度。

2、消息字段(field): 包括数据类型、字段名、字段规则、字段唯一标识、默认值。

> message使用数字标签作为key，Key 用来标识具体的 field，在解包的时候，Protocol Buffer 根据 Key 就可以知道相应的 Value 应该对应于消息中的哪一个 field。

3、数据类型（type）：常见的原子类型都支持 **4、字段规则：**

- required：必须初始化字段，如果没有赋值，在数据序列化时会抛出异常

- optional：可选字段，可以不必初始化。
- repeated：数据可以重复(相当于java 中的Array或List)
- 字段唯一标识：序列化和反序列化将会使用到。

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 messages, packed repeated fields
3	Start group	groups (deprecated)
4	End group	groups (deprecated)
5	32-bit	fixed32, sfixed32, float

数据类型type的对应关系如下：

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（wire_type ）， 第三列是面向开发者的 message 字段的类型。 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。 fixed32、fixed64：总是4字节和8字节 sfixed32、sfixed64：总是4字节和8字节 int32、int64：如果负数指定这两个类型编码效率较低。负数应该使用下面的类型。 sint32、sint64：对负数会进行ZigZag编码提高编码效率。 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` 。

总结下来就是： （1）变长编码类型Varints。 （2）固定32 bits类型。 （3）固定64 bits类型。 （4）有长度标记类型。

protobuf的编码原理

主要说明varints和zigzag。

protobuf的高效表现在： （1）解析高效。 （2）字节数占用少。

1. 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）；
2. 随后获取到 field_number （由此可知是哪一个字段）；
3. 依据 wire_type 来正确读取后面的 Value；
4. 接着继续读取下一个字段 field...

Varints 编码(变长的类型才使用)

为什么设计变长编码： 普通的int数据类型，无论其值的大小，所占用的存储空间都是相等的。比如 不管是0x12345678 还是0x12都占用4字节，那能否让0x12在表示的时候只占用1个字节呢？是否可以根据数值的大小来动态地占用存储空间, 使得值比较小的数字占用较少的字节数, 值相对比 较大的数字占用较多的字节数, 这即是变长整型编码的基本思想。

采用变长整型编码的数字, 其占用的字节数不是完全一致的, **Varints 编码使用每个字节的最高有效 位作为标志位msb(most significant bit), 而剩余的 7 位以二进制补码的形式来存储数字值本身**, 当最高有效位为 1 时, 代表其 后还跟有字节, 当最高有效位为 0 时, 代表已经是该数字的最后的一个字节。

在 Protobuf 中, 使用的是 Base128 Varints 编码, 在这种方式中, 使用 7 bit （即7的2次方为128） 来存储数字, 在 Protobuf 中, Base128 Varints 采用的是小端序（即数字的低位存放在高地址）。

> 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，字节旋转操作
>
> 00000101 | 00011010 经字节旋转，得到 00011010 | 00000101

Varints 编码和解码举例

实例1:

> 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，这里前面为什么这么多0，int32数据类型，对应32位，4个字节 补码：0000 ... 0000 0001 // 1 的补码表示，正数的补码和源码一样 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:

1、数字 1 对应补码 0000 ... 0000 0001（规则 2），

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。 3、那么第一个 7 位组的 msb = 0（规则1）

```
0 | 000 0001    对应16进制表达（0x01） ，注意最高位的0即为msb位
```

解码过程：

```
0#000 0001（0x01）
```

1、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）。 这里数字 1 的 Varints 编码中 msb = 0，所以只需要读完第一个字节无需再读 2、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， 3、还原即为数字 1。

> 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。

实例2：

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：

- 1、规则2，666 的补码为 `000 ... 101 0011010`
- 2、规则3，从后依次向前取 7 位组并反转排序

```
0011010 | 0000101
```

- 3、规则1，每个字节对应加上 msb，翻转后的高位字节的高位为1，低位字节的高位为0

```
1 0011010 | 0 0000101    对应16进制表达（0x9a 0x05）
```

解码过程：

- 1、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 组翻转得到补码：

```
000 0101 0011010    还原其原码为 666
```

> 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。

Varints 编码特征和问题

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。

负数的编码

```
int32 val = -1

原码：10000000 00000000 00000000 00000001
补码：11111111 11111111 11111111 11111111
再次复习 Varints 编码：对补码取 7 bit 一组，低位放在前面。

1.以7位1组单位逆序：
111 1111 111 1111 111 1111 111 1111 1111

2.7位1组，第一位高位为msb表示是否需要下一个字节。
1111 1111 1111 1111 1111 1111 1111 1111 0111 1000
故数字-1的varint编码为：
1111 1111 1111 1111 1111 1111 1111 1111 0111 1000
十六进制表示为：
0xFFFFFFFF78
```

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。

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**

ZigZag 编码

Google 又新增加了一种数据类型，叫 `sint`，专门用来处理这些负数，其实现原理是采用 `ZigZag` 编码。

有符号整数映射到无符号整数，编码结构依然为 Tag - Value，然后再使用 Varints 编码，ZigZag 编码的映射函数为

```
Zigzag(n) = (n << 1) ^ (n >> 31),  n为sint32时
Zigzag(n) = (n << 1) ^ (n >> 63),  n为sint64时
```

```
uint32 a = -1;
-1的二进制编码：
1111 1111 1111 1111 1111 1111 1111 1111
n << 1后为：
1111 1111 1111 1111 1111 1111 1111 1110
n >> 31后为：
1111 1111 1111 1111 1111 1111 1111 1111
故(n << 1) ^ (n >> 31)后为：
1111 1111 1111 1111 1111 1111 1111 1110
1111 1111 1111 1111 1111 1111 1111 1111----两行执行不进位的半加操作
0000 0000 0000 0000 0000 0000 0000 0001
故:Zigzag(-1) = 1;
```

> 目的是把多个1转成多个0表示。

最终的效果就是把所有的整数映射为正整数，比如 `0->0, -1->1, 1->2, -2->3`

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

解码时，解出正数之后再按映射关系映射回原来的负数。
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。

sint32 = Zigzag 编码 + varints编码合起来。 sint32 序列化：负数 -> Zigzag 编码 -> varints编码。 sint32 反序列化：varints解码 -> Zigzag 解码 -> 负数 。

总结

protobuf 是一种紧密的消息结构, 编码后字段之间没有间隔, 每个字段头由两部分组成: 字段编号和 wire type, 字段头可确定数据段的长度, 因此其字段之前无需加入间隔, 也无需引入特定的数据来标记字段末尾, 因此 Protobuf 的编码长度短, 传输效率高。

Protobuf 采用 Varints 编码和 Zigzag 编码来编码数据, 其中 Varints 编码的思想是移除数字高位的 0, 用变长的二进制位来描述一个数字, 对于小数字, 其编码长度短, 可提高数据传输效率, 但 由于它在每个字节的最高位额外采用了一个标志位来标记其后是否还跟有有效字节, 因此对于大 的正数, 它会比使用普通的定长格式占用更多的空间, 另外对于负数, 直接采用 Varints 编码将恒 定占用 10 个字节。

Zigzag 编码可将负数映射为无符号的正数, 然后采用 Varints 编码进行数据 压缩, 在各种语言的 Protobuf 实现中, 对于 int32 类型的数据, Protobuf 都会转为 uint64 而后 使用 Varints 编码来处理, 因此当字段可能为负数时, 我们应使用 sint32 或 sint64, 这样 Protobuf 会按照 Zigzag 编码将数据变换后再采用 Varints 编码进行压缩, 从而缩短数据的二进 制位数

问题1：为什么 Protobuf 能“知道”用 ZigZag?

1. **静态类型绑定**：
    - 开发者在 `.proto` 文件中显式指定 `sint32`，**编译器生成代码时会硬编码 ZigZag 逻辑。**
2. **编码/解码对称性**：

- 序列化时，`sint32` 字段的编码逻辑强制包含 ZigZag；
- 反序列化时，解析器根据 `.proto` 定义的类型自动调用 ZigZag 解码。

3. **无运行时类型检查**：
   - Protobuf 的编解码完全依赖预生成的代码，而非运行时反射，因此效率极高。

编码时：

- `sint32` → 先 ZigZag 转换，再按 Varint 编码
- `int32` → 直接按 Varint 编码（不转换）

解码时：

- 根据 `.proto` 定义的类型，决定是否执行 ZigZag 解码。

问题2：设置msb位算法

```
import java.io.ByteArrayOutputStream;

public class ProtobufVarintHelper {

    public static byte[] encodeVarint(long value) {
        ByteArrayOutputStream out = new ByteArrayOutputStream();
        while (true) {
            if ((value & ~0x7F) == 0) { // 检查高位是否全为0
                out.write((int) value); // 写入最后一字节（MSB=0）
                break;
            } else {
                out.write((int) (value & 0x7F) | 0x80); // 写入非最后一字节（MSB=1）
                value >>>= 7; // 无符号右移7位
            }
        }
        return out.toByteArray();
    }

    public static void main(String[] args) {
        long number = 300;
        byte[] encodedBytes = encodeVarint(number);
        System.out.print("Varint bytes for " + number + ": ");
        for (byte b : encodedBytes) {
            System.out.printf("0x%02X ", b);
        }
    }
}
```

输出结果：`0xAC 0x02`

关键步骤说明：

代码逻辑分解：

1. **循环条件**：持续处理直到所有高位为 `0`。
2. **掩码操作**：
   - `value & 0x7F`：提取低7位（`0x7F` 对应二进制 `01111111`）。
   - `| 0x80`：将第8位（MSB）设为 `1`（`0x80` 对应二进制 `10000000`）。
3. **位移操作**：`value >>>= 7` 将数值无符号右移7位，处理下一组。

数值：`300`
二进制表示：`100101100` → 补齐到 14 位 → `0000010 0101100`（分割为两组）

步骤分解：

1. **分割为 7 位组**：
   - **第一组（低位）**：`0101100` → 十进制 `44`（二进制 `0b0101100`）
   - **第二组（高位）**：`0000010` → 十进制 `2`（二进制 `0b0000010`）
2. **设置 MSB 并生成字节**：
   - **第一组**（非最后一组）：
     - 原始值：`44` → 二进制 `00101100`（8位）
     - 设置 MSB=1 → `10101100` → 十六进制 `0xAC`
   - **第二组**（最后一组）：
     - 原始值：`2` → 二进制 `00000010`（8位）
     - 设置 MSB=0 → `00000010` → 十六进制 `0x02`
3. **最终字节流**：`0xAC 0x02`