Protobuf通信协议

一、python安装Protobuf包

```
sudo pip install protobuf==2.5.0
```

二、安装编译工具包

2.1 下载安装包

1、通过github克隆

```
wget
https://github.com/protocolbuffers/protobuf/releases/download/v2.5.0/protobuf-2.5.0.tar.gz
```

2、获取资源超时，直接从网站下载

https://download.csdn.net/download/qq_48974566/87400597

2.2 解压

```
tar -zxvf protobuf-2.5.0.tar.gz
```

进入解压后的文件

```
cd protobuf-2.5.0
```

2.3 安装

1、安装前的环境

```
sudo apt-get install autoconf automake libtool curl make
g++ unzip
```

2、安装子模块

```
git clone https://github.com/google/protobuf.git
```

3、正式安装

执行命令，如果没有克隆子模块，make check 会失败但是可以继续 makeinstall，但是使用某些功能时可能会出错

```
./configure
make
make check
sudo make install
```

4、配置 protobuf环境(根据自身情况配置)

```
sudo vim /etc/profile
```

在文件末尾添加两行代码

```
export PATH=/usr/local/protoc/bin:$PATH
export PATH=/usr/local/protoc/include:$PATH
```

更新(使/etc/profile 中的配置立即生效)

```
source /etc/profile
```

5、配置动态链接库

```
sudo vim /etc/ld.so.conf

新加一行:
/usr/local/protobuf/lib
执行如下命令:
ldconfig
```

2.4 安装成功

```
protoc --version
```

三、Protobuf通信协议的作用

用于封装数据的一个协议

传100个字节的数据，100个字节可能有很多种方式去传输，比如xml，json等传输格式

不使用封装数据的协议的话，你传输100个字节，就是100个字节。传大的数据（1000个字节），它还是1000个字节，不会进行压缩。

在网络中，你传的数据包很大的话，会很慢，而且比较耗流量。

想要优化的话，就需要使用通信协议。经过通信协议的序列号方法之后，会减少数据的大小

Protobuf是谷歌发明的一个通信协议

Protobuf是不依赖编译语言的，支持跨语言，在编译器直接调用接口即可完成对不同语言的封装

缺点就是 封装完成后是二进制流，可读性差

优点：

（1）同样的内容，大小比xml格式小3-10倍，传输速度快20-100倍

（2）向后兼容性好，传入消息添加了其他的字段，直接加就行，旧版本也不会读取新的字段

不用停服务器就能完成新旧版本的切换

四、基本使用（Protobuf 2）

4.1 数据格式

限定修饰符 数据类型 字段名字 =字段编码值[字段默认值]

4.2 注释

proto文件中的注释和 c，c++的注释风格相同，使用//和 /*...*/

4.3 限定修饰符

```
//字段规则
//required:必须有一个
//optional:0或1个
//repeated:任意数量(包括0)
```

4.4 数据类型

proto 消息类型	C++ 类型	Python 类型	说明

消息类型	C++ 类型	Python 类型	说明
double	double	double	双精度浮点型
float	float	float	单精度浮点型
int32	int32	int32	使用可变长编码方式，负数时不够高效，应该使用 sint32
int64	int64	int64	使用可变长编码方式，负数时不够高效，应该使用 sint32
uint32	uint32	uint32	使用可变长编码方式
uint64	uint64	uint64	使用可变长编码方式
sint32	int32	int32	使用可变长编码方式，有符号的整型值，负数编码时比通常的 int32 高效
sint64	int64	int64	使用可变长编码方式，有符号的整型值，负数编码时比通常的 int64
fixed32	uint32	uint32	总是 4 个字节，如果数值总是比 2^28 大的话，这个类型会比 uint32 高效
fixed64	uint64	uint64	总是 8 个字节，如果数值总是比 2^56 大的话，这个类型会比 uint64 高效
sfixed32	int32	int32	总是 4 个字节
sfixed64	int64	int64	总是 8 个字节
bool	bool	bool	布尔类型
string	string	string	一个字符串必须是 utf-8 编码或者 7-bit 的 ascii 编码的文本
bytes	string	string	可能包含任意顺序的字节数据

- 字段默认值

- 字段默认值

4.5 字段默认值

- 字段默认值

类型	默认值
string	空字符串
bytes	空 bytes
bool	false
数值类型	0
枚举	默认是第一个定义的枚举值，必须为 0

4.6 字段编码值

消息定义中每个字段都有一个唯一的编号。这些字段编号用于以二进制格式标识字段，一旦这个消息类型被使用就不应该更改。

4.7 编写proto文件

```
syntax = "proto2";
//默认位使用 proto2 语法，如果写 syntax = "proto3"
//则说明使用的使 proto3 的语法。这一行必须是文件中非空非注释行的第一行

//可以在同一个 proto 中定义多个
message MailDetail{
    required string title = 1;
    required string content = 2;
}

message Mail{
    required int32 userid = 1;
    required int32 mailid = 2;
}

//----------------------------------------

//可以嵌套定义
message Mail{
    message MailDetail{
        required string title = 1;
        required string content = 2;
    }
    required int32 userid = 1;
    required int32 mailid = 2;
    required MailDetail maildetail = 3;
}

//嵌套定义后，如果想在它的父消息类型外部重用这个消息类型
message MailMessage{
    required Mail.MailDetail maildetail = 1;
}

//----------------------------------------

//可以将其他消息类型作为字段的类型
message MailDetail{
    required string title = 1;
```

```
//可以将其他消息类型作为字段的类型
message MailDetail{
    required string title = 1;
    required string content = 2;
}

message Mail{
    required int32 userid = 1;
    required int32 mailid = 2;
    required MailDetail maildetail = 3;
}
```

同一个 proto 文件可以生成不同语言，比如生成 mail pb2.py 文件(python 使用)/mai.pb.cc和mail.pb.h(C++使用)

```
protoc -I./--python_out=./*.proto
protoc -I./--cpp_out=./*.proto
```

-I: *.proto文件所在路径
--python out=./ : 表示生成的.py 文件存放的位置
--cpp_out=./ : 表示生成的.pb.cc 和 .pb.h 文件存放的位置
最后为需要编译的 proto 文件

proto.sh

```
protoc -I/--python out=./*.proto
```

sh proto.sh

使用的三个步骤:

编写 proto 文件定义消息格式
使用 protobuf编译器编译 .proto 文件生成对应语言所需要的文件
使用 C++/Python 等对应的 protobuf调用编译后文件提供的 API即可

五、编码原理

5.1 编码类型

Type	Meaning	Used For
0	Varint	int32, int64, uint32, uint64, sint32, sint64, bool, enum
1	64-bit	fixed64, sfixed64, double
2	Length- delimited (长度分割)	string, bytes, embedded messages, packed repeated fields
3	Start group	groups (deprecated 荒废，官方不再推荐)
4	End group	groups (deprecated 荒废，官方不再推荐)
5	32-bit	fixed32, sfixed32, float

4	End group	groups (deprecated 荒废，官方不再推荐)
5	32-bit	fixed32, sfixed32, float

总结：

（1）变长编码类型 Varints
（2）固定 32bits 类型
（3）固定 64bits 类型
（4）有长度标记类型

1、Varints编码（变长的类型才使用）

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。

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，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一个字节。

举例：

```
uint32_t = 1;
// 原码: 0000 0000 0000 0000 0000 0000 0000 0001
// 补码: 0000 0000 0000 0000 0000 0000 0000 0001
// varints 编码: 0|000 0001(0x01)

// 整数 1 的 variant 解码
// 读取到 0x01, msb 位为 0, 说明读取结束, 该数就只占一个字节.
// 读到 7bits 值为 000 0001 的补码, 显然其原码是 000 0001,也就是数值 1
// 如此, 便将 variant 编码的整数 1 解码出来了。而 0x01 只用到了一个字节


uint32_t = 665;
// 原码: 0000 0000 0000 0000 0000 0010 1001 1001
// 补码: 0000 0000 0000 0000 0000 0010 1001 1001
// 按照 7bit 一组, 0000 ... 0000101 0011001
// 从低位往高位依次取 7bit,重新编码,注意是反转排序, 全 0 的 bit 舍弃
// 0011001 0000101
// 再加上 msb, 每 7bit 组 组成一个字节
// varints 编码即为:
// 1|0011001 0|0000101
// 0x99       0x05
// 如此, 即将整数 665 编码成 0x99 0x05,将原来的 4 个字节压缩成 2 个字节

// 再来看看解码过程
// 编码为 0x99 0x05, 第一个字节的 msb 位为 1, 表示继续读取下一个字节; 第二个字节的 msb 为 0, 读取结束。
// 读出两个字节的内容后, 去掉 msb, 还原成 2 组 7bits 组, 即 0011001 0000101
// 再将两组 7bits 反转, 即 0000101 0011001 (补码)
// 由于是整数, 所以得到原码 0010 1001 1001, 即 665
```

注意:由于每个字节都拿出了1个比特作为 msb 位，那么 4个字节最大能表示的数就为 2^28，而不是 2^32 了，在实际情况下，大于 2^28的数很少出现，所以在实际工程中并不会影响高效性。

```
int32_t val = -1 // 4 字节
// 原码: 1000 0000 0000 0000 0000 0000 0000 0001
// 补码: 1111 1111 1111 1111 1111 1111 1111 1111
// 按照 variant 的编码规则, 7bit 一组, 低位在前
// 补码的 4 个字节, 共 32bit, 按 7bit 分组, 可分为 5 组, 这 5 组的 msb 均为 1; 除
```

```
// 补码: 1111 1111 1111 1111 1111 1111 1111 1111
// 按照 variant 的编码规则，7bit 一组，低位在前
// 补码的 4 个字节，共 32bit，按 7bit 分组，可分为 4 组，这 4 组的 msb 均为 1；余下的 4bit 在第 5 组，msb 为 0，其余位补 0
// 1111111 1111111 1111111 1111111 1111
// 1|1111111 1|1111111 1|1111111 1|1111111 0|0001111
//    0xFF      0xFF      0xFF      0xFF       0x0F
// 所以，-1 的 varint 编码为 0xff 0xff 0xff 0xff 0xf 共 5 字节
```

缺陷:从上面这个例子来看，原本占4个字节的-1，经过 Variant 编码后，占了5个字节，并没有压缩，所以这就是 Variant 编码的缺陷:Variant 编码**对负数编码效率低**。通常使用 zigzag 编码将负数转为正数然后再使用 Variant 编码。

2、Zigzag编码（针对负数）

对于正整数，可以把无意义的 0 去掉，只存储从1开始的“有效”数据，就可以压缩数据了。但是负数的符号位为1，阻碍了对于无意义0的压缩。

编码方法:

```
对于 n = -1, (11111111 11111111 11111111 11111111)补
n << 1: a=(11111111 11111111 11111111 11111110)补，数据位整体左移 1 位
n >> 31: b=(11111111 11111111 11111111 11111111)补，符号位移到最低位
c = a ^ b, c=(00000000 00000000 00000000 00000001)补，将 -1 编码为 1
注意：因为 a 中数据位冗余的前导 1 刚好与 b 中数据位冗余的前导 1 相对应，那么进行异或操作时，就能将这些冗余的前导 1 消除掉，数据位完成了取反动作，这样就能压缩数据了

对于 32 位整数，(n << 1) ^ (n >> 31)，即能实现 zigzag 编码
```

解码方法:

```
对于 zigzag 编码的值位 2, (00000000 00000000 00000000 00000010)补
n >> 1: a=(00000000 00000000 00000000 00000001)补，整体右移还原数据位
-(n&1): b=-(2&1)=-(0)=(00000000 00000000 00000000 00000000)补，最低位按位与，取符号，还原符号位
c = a ^ b, c=(00000000 00000000 00000000 00000001)补，将 zigzag 编码值 2 解码还原成了 1

对于 zigzag 编码的值位 3, (00000000 00000000 00000000 00000011)补
n >> 1: a=(00000000 00000000 00000000 00000001)补，整体右移还原数据位
-(n&1): b=-(3&1)=-(1)=(11111111 11111111 11111111 11111111)补，最低位按位与，取符号，还原符号位
c = a ^ b, c=(11111111 11111111 11111111 11111110)补，将 zigzag 编码值 3 解码还原成了 -2

对于 32 位整数，(n >> 1) ^ -(n & 1)，即能实现 zigzag 解码
```