应用层协议设计ProtoBuf20220104

1 协议概述

2 判断消息的完整性

3 协议设计

- 3.1 协议设计范例
  - 3.1.1 范例1-IM即时通讯
  - 3.1.2 范例2-云平台节点服务器
  - 3.1.3 范例3-nginx
  - 3.1.4 范例4-HTTP协议
  - 3.1.5 范例5-redis协议
- 3.2 序列化方法
  - 3.2.1 常见序列化方法
  - 3.2.2序列化结果数据对比
    - xml
    - json
    - protobuf
  - 3.2.3 序列化、反序列化速度对比
- 3.3 协议安全
- 3.4 数据压缩
- 3.5 协议升级

4 protobuf的使用

- 4.1 ProtoBuf 协议的工作流程
- 4.2 protobuf的编译安装
  - 编译安装
  - protobuf option部分选项
- 4.3 标量数值类型
- 4.4 protobuf的使用实例-和范例1-即时通讯同代码
- 4.5 protobuf工程经验

5 protobuf的编码原理

5.1 Varints 编码(变长的类型才使用)

5.2 Zigzag 编码(针对负数的)

5.3 Protobuf 的数据组织

5.4 编码总结

6. protobuf协议消息升级

补充知识

原码、反码、补码，计算机中负数的表示

【翻译】Protobuf比JSON性能更好

Protobuf 的 proto3 与 proto2 的区别

零声学院 Darren QQ 326873713

C/C++Linux服务器开发/高级架构师

https://ke.qq.com/course/420945?tuin=137bb271

时间：2022年01月04日

通信协议设计核心

- 解析效率
- 可扩展可升级

协议设计细节

- 帧完整性判断
- 序列化和反序列化
- 协议升级
- 协议安全
- 数据压缩

设计目标

- 解析效率：互联网业务具有高并发的特点，解析效率决定了使用协议的CPU成本；编码长度：信息编码出来的长度，编码长度决定了使用协议的网络带宽及存储成本；
- 易于实现：互联网业务需要一个轻量级的协议，而不是大而全的；
- 可读性：编码后的数据的可读性决定了使用协议的调试及维护成本（不同的序列化协议是有不同的应用的场景）；

- 兼容性: 互联网的需求具有灵活多变的特点，协议会经常升级，使用协议的双方是否可以独立升级协议、增减协议中的字段是非常重要的；
- 跨平台跨语言：互联网的的业务涉及到不同的平台和语言，比如Windows用C++，Android用Java，Web用Js，IOS用object-c。
- 安全可靠：防止数据被破解

1 协议概述

什么协议：协议是一种约定，通过约定，不同的进程可以对一段数据产生相同的理解，从而可以相互协作，存在进程间通信的程序就一定需要协议。

为什么说进程间通信就需要协议？而不是说客户端和服务器端之前？

为什么需要自己设计协议？

比如不同表的插头，还需要进行各种转换，如果我们两端进行通信没有约定好协议，那彼此是不知道对方发送的数据是什么意义。

2 判断消息的完整性

为了能让对端知道如何给消息帧分界，目前一般有以下做法：

1. ~~以固定大小字节数目来分界，如每个消息100个字节，对端每收齐100个字节，就当成一个消息来解析；~~
2. 以特定符号来分界，如每个消息都以特定的字符来结尾（如\r\n），当在字节流中读取到该字符时，则表明上一个消息到此为止 `'$6\r\nfoobar\r\n'` ；
3. **固定消息头+消息体结构**，这种结构中一般消息头部分是一个固定字节长度的结构，并且消息头中会有一个特定的字段指定消息体的大小。收消息时，先接收固定字节数的头部，解出这个消息完整长度，按此长度接收消息体。这是目前各种网络应用用的最多的一种消息格式；header + body
4. 在序列化后的buffer前面增加**一个字符流的头部**，其中有个字段存储消息总长度，根据特殊字符（比如根据\n或者\0）判断头部的完整性。这样通常比3要麻烦一些，HTTP和REDIS采用的是这种方式。收消息的时候，先判断已收到的数据中是否包含结束符，收到结束符后解析消息头，解出这个消息完整长度，按此长度接收消息体。

3 协议设计

3.1 协议设计范例

3.1.1 范例1–IM即时通讯

即时通讯的协议设计

字段	类型	长度(字节)	说明
length	unsigned int	4	整个消息的长度包括　协议头 +　BODY
version	unsigned short	2	通信协议的版本号
appid	unsigned short	2	对外SDK提供服务时，用来识别不同的客户
service_id	unsigned short	2	对应命令的分组类比，比如login和msg是不同分组
command_id	unsigned short	2	分组里面的子命令，比如login和login response
seq_num	unsigned short	2	消息序号
reserve	unsigned short	2	预留字节
body	unsigned char[]	n	具体的协议数据

3.1.2 范例2–云平台节点服务器

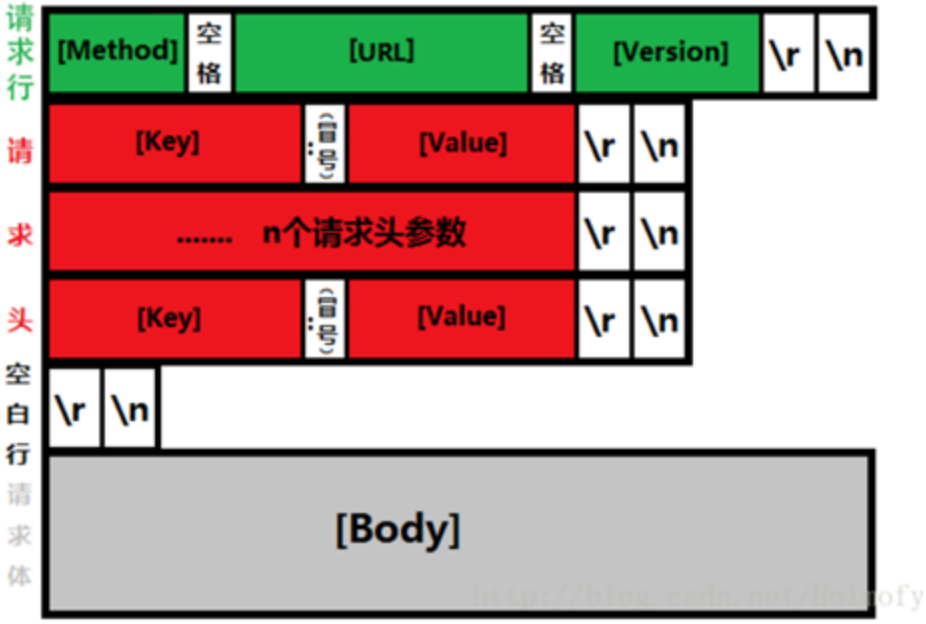
字段	类型	长度(字节)	说明
STAG	unsigned short	2	通信协议数据包的开始标志 0xff 0xfe h264 **0 0 0 1**
version	unsigned short	2	通信协议的版本号，目前为**0x01**
checksum	unsigned char	1	计算协议数据校验和，如果为加密数据，则计算密文校验和。校验和计算范围：协议头CheckSum字段后数据，协议体全部数据。
type	unsigned char	1	0表示协议体是json格式，其它值未定义。设备心跳消息类型的值为0xA0
seqno	unsigned int	**4**	通信数据报文的序列号，应答报文序列号必须与请求报文序列号相同
Length	unsigned short	4	报文内容长度，即从该字段后报文内容长度
reserve	unsigned int	4	预留字节，设备心跳消息类型的值为devid
body	unsigned char []	n	数据

具体参考《云平台节点服务器设计说明书_v0.4.10.pdf 》

3.1.3 范例3–nginx

```
typedef struct {
    ngx_char_t    magic[2];     // magic number
    ngx_short_t   version;      // protocol version
    ngx_short_t   type;         // protocol type: json, xml, binary, protobuf, flatbuffer and so on
    ngx_short_t   len;          // body length
    ngx_uint_t    seq;          // message number
    ngx_short_t   id;           // message id, HEARTBEAT, LOGIN ...
    ngx_char_t    reserve[2];   // reserve
}ngx_message_head_t;
```

3.1.4 范例4–HTTP协议



HTTP协议是我们最常见的协议，我们是否可以采用HTTP协议作为互联网后台的协议呢？这个一般是不适当的，主要是考虑到以下2个原因：

1）HTTP协议只是一个框架，没有指定包体的序列化方式，所以还需要配合其他序列化的方式使用才能传递业务逻辑数据。

2）HTTP协议解析效率低，而且比较复杂（不知道有没有人觉得HTTP协议简单，其实不是http协议简单，而是HTTP大家比较熟悉而已）

有些情况下是可以使用HTTP协议的：

1) 对公网用户api，HTTP协议的穿透性最好，所以最适合；

2) 效率要求没那么高的场景；

3) 希望提供更多人熟悉的接口，比如新浪微、腾讯博提供的开放接口；

比如图床项目：

```
调用接口
http://42.194.128.13/reg
参数
{
  "email": "472251823@qq.com",
  "firstPwd": "e10adc3949ba59abbe56e057f20f883e",
  "nickName": "lucky",
  "phone": "18612345678",
  "userName": "qingfu"
}

返回结果
{
  "code": 0
}

```

问题：http的body是二进制还是文本?

http content type: https://tool.oschina.net/commons

3.1.5 范例5–redis协议

REDIS协议：

基本原理是：先发送一个字符串表示参数个数，然后再逐个发送参数，每个参数发送的时候，先发送一个字符串表示参数的数据长度，再发送参数的内容。

具体参考：《 Redis协议规范–20220104.pdf 》

3.2 序列化方法

- TLV编码及其变体(TLV是tag, length和value的缩写)：比如Protobuf。
- 文本流编码：比如XML/JSON
- 固定结构编码： 基本原理是，协议约定了传输字段类型和字段含义，和TLV的方式类似，但是没有了tag和len，只有value，比如TCP/IP
- 内存dump：基本原理是，把内存中的数据直接输出，不做任何序列化操作。反序列化的时候，直接还原内存。

3.2.1 常见序列化方法

主流序列化协议：xml、json、protobuf

1. XML指可扩展标记语言（eXtensible Markup Language）。是一种通用和重量级的数据交换格式。以文本方式存储。
2. JSON(JavaScript ObjectNotation, JS 对象简谱) 是一种通用和轻量级的数据交换格式。以文本结构进行存储。
3. protocol buffer是Google的一种独立和轻量级的数据交换格式。以二进制结构进行存储。

类型	通用些	大小	格式
XML	通用	重量级	文本格式
JSON	通用	轻量级	文本格式（方便调试）
Protobuf（编译器，生成对应语言的代码）	独立	轻量级	二进制格式

3.2.2序列化结果数据对比

xml

```xml
<?xml version="1.0" encoding="UTF-8"?>
<name>darren</name>
<age>80</age>
<languages>
    <lang>C++</lang>
    <lang>Linux</lang>
</languages>
<phone>
    <number>18570368134</number>
    <type>home</type>
</phone
<books>
    <book>
        <name>Linux kernel development</name>
        <price>7.7</price>
    </book>
    <book>
        <name>Linux server development</name>
        <price>8.0</price>
    </book>
</books>
<vip>true</vip>
<address>yageguoji</address>
```

XML 文本

json

```
{
    "name": "darren",
    "age":  80,
    "languages":    ["C++", "Linux"],
    "phone":    {
        "number":   "18570368134",
        "type": "home"
    },
    "books":    [{
            "name": "Linux kernel development",
            "price":    7.7
        }, {
            "name": "Linux server development",
            "price":    8.0
        }],
    "vip":  true,
    "address":  "yageguoji"
}
```

json 文本

protobuf

protobuf 二进制

```
0a 06 64 61 72 72 65 6e 10 50 1a 03 43 2b 2b 1a sf
05 4c 69 6e 75 78 22 0f 0a 0b 31 38 35 37 30 33 sf
36 38 31 33 34 10 01 2a 1f 0a 18 4c 69 6e 75 78 sf
20 6b 65 72 6e 65 6c 20 64 65 76 65 6c 6f 70 6d sf
65 6e 74 15 66 66 f6 40 2a 1f 0a 18 4c 69 6e 75 sf
78 20 73 65 72 76 65 72 20 64 65 76 65 6c 6f 70 sf
6d 65 6e 74 15 00 00 00 41 30 01 3a 09 79 61 67 sf
65 67 75 6f 6a 69
```

3.2.3 序列化、反序列化速度对比

测试10万次序列化

库	默认	-O1	序列化后字节
cJSON(C语言)	488ms	452ms	297
jsoncpp(C++语言)	871ms	709ms	255
rapidjson(C++语言)	701ms	113ms	239
tinyxml2(XML)	1383ms	770ms	**474**
protobuf	241ms	83ms	117

测试10万次反序列化

库	默认	-O1
cJSON	284ms	251ms
jsoncpp	786ms	709ms
rapidjson	1288ms	128ms
tinyxml2	1781ms	953ms
protobuf	190ms	80ms

3.3 协议安全

1. xxtea 固定key （https://blog.csdn.net/gsls200808/article/details/48243019）
2. **AES** 固定key（https://www.yuque.com/docs/share/e1cc0245-5daf-466b-ad25-32f9be9dff06?# 《AES详解》）
3. **openssl**
4. Signal protocol 端到端的通讯加密协议（https://www.yuque.com/docs/share/d8305ed5-53ba-4533-b4ac-f1cbe52b7c58?# 《Signal protocol 开源协议理解》）

3.4 数据压缩

（《字符编码Unicode原理及编程实践》 该节课讲解）

1. deflate nginx
2. gzip
3. lzw

文本的可以考虑做压缩，带宽成问题的时候再去考虑

3.5 协议升级

协议升级= 增加字段 大版本

1. **通过版本号指明协议版本，即是通过版本号辨别不同类型的协议**
2. **支持协议头部可扩展，即是在设计协议头部的时候有一个字段用来指明头部的长度。**

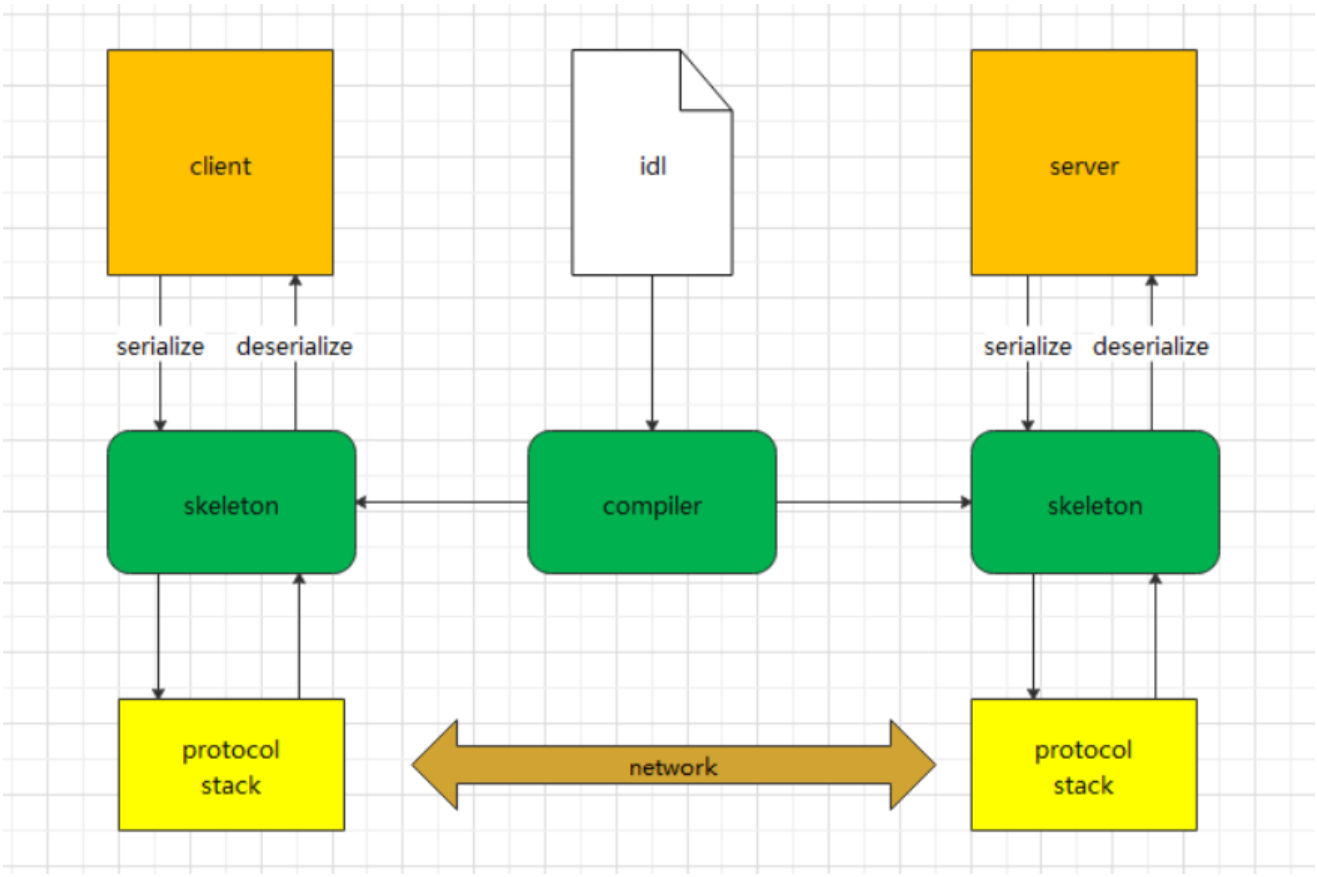
4 protobuf的使用

Protocol buffers 是一种语言中立，平台无关，可扩展的序列化数据的格式，可用于通信协议，数据存储等。

Protocol buffers 在序列化数据方面，它是灵活的，高效的。相比于 XML 来说，Protocol buffers 更加小巧，更加快速，更加简单。一旦定义了要处理的数据的数据结构之后，就可以利用 Protocol buffers 的代码生成工具生成相关的代码。甚至可以在无需重新部署程序的情况下更新数据结构。只需使用 Protobuf 对数据结构进行一次描述，即可利用各种不同语言或从各种不同数据流中对你的结构化数据轻松读写。

Protocol buffers 很适合做数据存储或 RPC 数据交换格式。可用于通讯协议、数据存储等领域的语言无关、平台无关、可扩展的序列化结构数据格式。tars brpc

4.1 ProtoBuf 协议的工作流程



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。

4.2 protobuf的编译安装

https://github.com/protocolbuffers/protobuf
谷歌开源的 协议标准+工具

编译安装

安装工具->根据编写的proto文件产生c++代码

1. 解压

tar zxvf protobuf-cpp-3.8.0.tar.gz

2. 编译（make的时间有点长）

cd protobuf-3.8.0/
./configure
make
sudo make install

sudo ldconfig

3. 显示版本信息

protoc --version

生成各种版本的代码

4. 编写proto文件

参考：《Protobuf3语法详解.pdf》

参考：4-addressbook代码

5. 将proto文件生成对应的.cc和.h文件

protoc -I=/路径1 --cpp_out=./路径2 /路径1/addressbook.proto

路径1为.proto所在的路径

路径2为.cc和.h生成的位置

将指定proto文件生成.pb.cc和.pb.h

protoc -I=./ --cpp_out=./ test.proto

将对应目录的所有proto文件生成.pb.cc和.pb.h

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

6. 编译范例

比如：

```
g++ -std=c++11 -o list_people list_people.cc addressbook.pb.cc -lprotobuf -lpthread
```

protobuf option部分选项

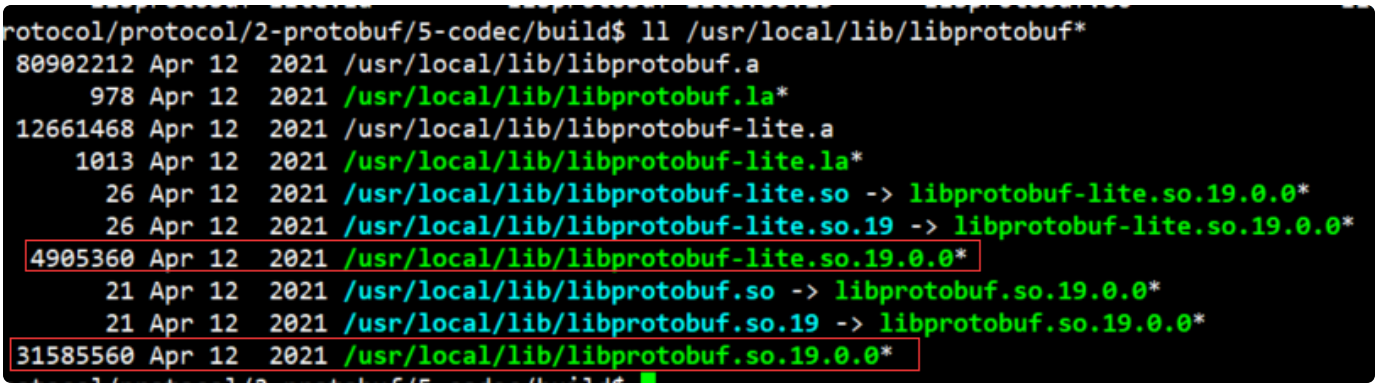
option optimize_for = LITE_RUNTIME;

optimize_for是文件级别的选项，Protocol Buffer定义三种优化级别SPEED/CODE_SIZE/LITE_RUNTIME。缺省情况下是SPEED。

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。
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Mobile。

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。这是以牺牲Protocol Buffer提供的反射功能为代价的。因此我们在C++中链接Protocol Buffer库时仅需链接libprotobuf-lite，而非libprotobuf。

比如下面显示的



protobuf的反射参考（实际开发中C++使用反射的场景比较少）：

1. https://blog.csdn.net/JMW1407/article/details/107223287
2. https://blog.csdn.net/chengangdzzd/article/details/50446367

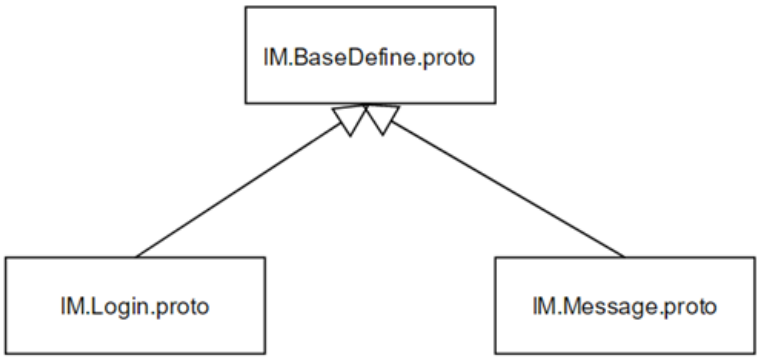
4.3 标量数值类型

一个标量消息字段可以含有一个如下的类型——该表格展示了定义于.proto文件中的类型，以及与之对应的、在自动生成的访问类中定义的类型：

.proto Type	Notes	C++ Type	Java Type	Go Type
double		double	double	float64
float		float	float	float32
int32	使用变长编码，对于负值的效率很低，如果你的域有可能有负值，请使用sint64替代	int32	int	int32
uint32	使用变长编码	uint32	int	uint32
uint64	使用变长编码	uint64	long	uint64
sint32	使用变长编码，这些编码在负值时比int32高效的多	int32	int	int32
sint64	使用变长编码，有符号的整型值。编码时比通常的int64高效。	int64	long	int64
fixed32	总是4个字节，如果数值总是比总是比2^28大的话，这个类型会比uint32高效。	uint32	int	uint32
fixed64	总是8个字节，如果数值总是比总是比2^56大的话，这个类型会比uint64高效。	uint64	long	uint64
sfixed32	总是4个字节	int32	int	int32
sfixed64	总是8个字节	int64	long	int64
bool		bool	boolean	bool
string	一个字符串必须是UTF-8编码或者7-bit ASCII编码的文本。	string	String	string
bytes	可能包含任意顺序的字节数据。	string	ByteString	[]byte

4.4 protobuf的使用实例-和范例1-即时通讯同代码

1. pb协议设计
2. 登录
3. 发送消息
4. 代码验证



4.5 protobuf工程经验

1. proto文件命名规则；
2. proto命名空间；
3. 引用文件；
4. 多个平台使用同一份proto文件

5 protobuf的编码原理

varints 和zigzag

protobuf的编码原理 参考：高效的数据压缩编码方式Protobuf

编码原理，只讲重点原理，而不是完全去讲这个编码原理的实现。

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)
4	End group	groups (deprecated)
5	32–bit	fixed32, sfixed32, float

5.1 Varints 编码(变长的类型才使用)

通常来说, 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, 这点可以引起人们的思考, 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, 这便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, 为了达到这一点, **Varints 编码使用每个字节的最高有效位作为标志位, 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, 之所以叫这个名字原因即是在这种方式中, **使用 7 bit 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**, 举例来看, 对于数字 1, 我们假设 int 类型占 4 个字节, 以标准的整型存储, 其二进制表示应为

(00000000 00000000 00000000 00000001)

可见, 只有最后一个字节存储了有效数值, 前 3 个字节都是 0, 若采用 Varints 编码, 其二进制形式为

(00000001)

因为其没有后续字节, 因此其最高有效位为 0, 其余的 7 位以补码形式存放 1, 再比如数字 666, 其以标准的整型存储, 其二进制表示为

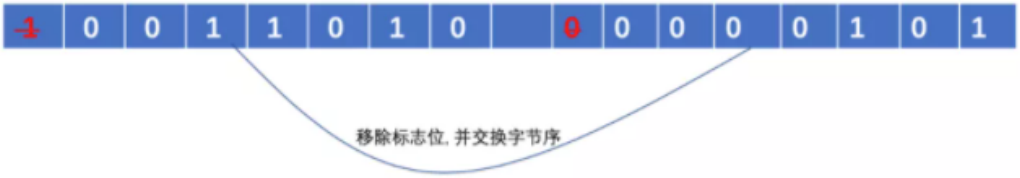
(00000000 00000000 00000010 10011010)

而采用 Varints 编码, 其二进制形式为

(10011010 00000101)

我们可以尝试来复原一下上面这个 Base128 Varints 编码的二进制串, 首先看最高有效位

接下来我们移除标识位, 由于 Base128 Varints 采用小端字节序, 因此数字的高位存放于低地址上,



移除标志位并交换字节序, 便得到原本的数值 1010011010, 即数字 666

从上面的编码解码过程可以看出, 可变长整型编码对于不同大小的数字, 其所占用的存储空间是不同的, 编码思想与 CPU 的间接寻址原理相似, 都是用一比特来标识是否走到末尾, 但采用这种方式存储数字, 也有一个相对不好的点便是, 无法对一个序列的数值进行随机查找, 因为每个数字所占用的存储空间不是等长的, 因此若要获得序列中的第 N 个数字, 无法像等长存储那样在查找之前直接计算出 Offset, **只能从头开始顺序查找**

5.2 Zigzag 编码(针对负数的)

Varints 编码的实质在于去掉数字开头的 0, 因此可缩短数字所占的存储字节数, 在上面的例子中, 我们只举例说明了正数的 Varints 编码, 但如果数字为负数, 则采用 Varints 编码会恒定占用 10 个字节, 原因在于负数的符号位为 1, 对于负数其从符号位开始的高位均为 1, 在 Protobuf 的具体实现中, 会将此视为一个很大的无符号数, 以C++语言的实现为例, 对于 int32 类型的 pb 字段, 对于如下定义的 proto

Bash 复制代码

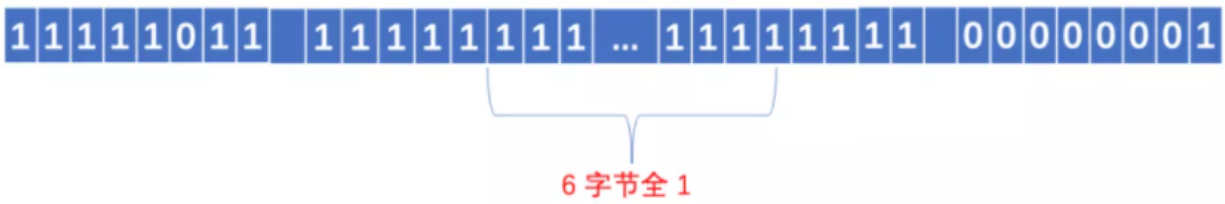
```
message Tint32{
    int32   n1      = 1;
}
```

Request 中包含类型为 int32 类型的字段, 当 a 为负数时, 其序列化之后将恒定占用 10 个字节, 我们可以使用如下的测试代码（见2-protobuf\5-codec代码的**Tint32()函数**）

```Basic
std::string strPb;
uint8_t *szData;
Codec::Tint32 int1;

int1.set_n1(-5);
int1Size = int1.ByteSize();        // 序列化后的大小, 占用11字节
std::cout << "-5 int1Size = " << int1Size << std::endl;
strPb.clear();
strPb.resize(int1Size);
szData = (uint8_t *)strPb.c_str();
int1.SerializeToArray(szData, int1Size);   // 拷贝序列化后的数据
printHex(szData, int1Size);    // 08 fb ff ff ff ff ff ff ff ff 01
```

对于 int32 类型的数字 -5, 其序列化之后的二进制为



究其原因在于 Protobuf 的内部将 int32 类型的负数转换为 **uint64** 来处理。

比如整数 -5。先取 5 的原码：00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000101，得反码： 11111111 11111111 11111111 11111111 11111111 11111111 11111111 11111010，对反码加1最后得补码：11111111 11111111 11111111 11111111 11111111 11111111 11111111 11111011，即-5在计算机里用二进制表示结果11111111 11111111 11111111 11111111 11111111 11111111 11111111 11111011。

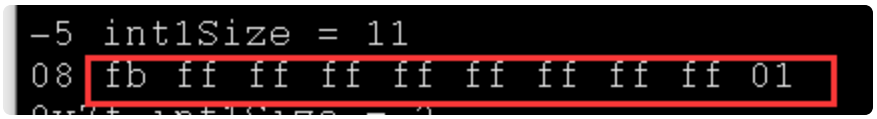
转成每7bit占用1个字节，

（高位）1 1111111 1111111 1111111 1111111 1111111 1111111 1111111 1111111 1111011（低位）

然后高地址存储到低地址，并且不是结束字节最高位为1，即是

（低位）11111011 11111111 11111111 11111111 11111111 11111111 11111111 11111111 11111111 00000001（高位）

转成16进制：fb ff ff ff ff ff ff ff ff 01 数据本身就占用了10字节。



转换后的 uint64 数值的高位全为 1，相当于是一个 8 字节的很大的无符号数，因此采用 Base128 Varints 编码后将恒定占用 10 个字节的空间，可见 Varints 编码对于表示负数毫无优势，甚至比普通的固定 32 位存储还要多占 4 个字节。Varints 编码的实质在于设法移除数字开头的 0 比特，而对于负数，由于其数字高位都是 1，因此 Varints 编码在此场景下失效，Zigzag 编码便是为了解决这个问题，Zigzag 编码的大致思想是首先对负数做一次变换，将其映射为一个正数，变换以后便可以使用 Varints 编码进行压缩，这里关键的一点在于变换的算法，首先算法必须是可逆的，即可以根据变换后的值计算出原始值，否则就无法解码，同时要求变换算法要尽可能简单，以避免影响 Protobuf 编码、解码的速度，我们假设 n 是一个 32 位类型的数字，则 Zigzag 编码的计算方式为

(n << 1) ^ (n >> 31)（正向）

(n>>1)^ -(n&1) （逆向）

要注意这里左边是逻辑移位，右边是算术移位，右边的含义实际是得到一个全 1（对于负数）或全 0（对于正数)的比特序列，因为对于任意一个位数为 η 的有符号数 n，其最高位为符号位，剩下的 η – 1 位为数字位，将其算术右移 η – 1 位，由于是算术移位，因此右移时左边产生的空位将由符号位来填充，进行 η – 1 次算术右移之后便得到 η 位与原先的符号位相等的序列，然后对两边按位异或便得到 Zigzag 编码，我们用一个图示来直观地说明 Zigzag 编码的设计思想，为了简化，我们假定数字是 16 位的，先来看负数的情形，假设数字为 –5，其在内存中的形式为

首先对其进行一次**逻辑左移**，移位后空出的比特位由 0 填充

然后对原数字进行 15 次**算术右移**，得到 16 位全为原符号位(即 1)的数字

然后对逻辑移位和算术移位的结果按位异或，便得到最终的 Zigzag 编码

（Zigzag目的是把高位的1变成0）

可以看到，对负数使用 Zigzag 编码以后，其高位的 1 全部变成了 0，这样以来我们便可以使用 Varints 编码进行进一步地压缩，再来看正数的情形，对于 16 位的正数 5，其在内存中的存储形式为

我们按照与负数相同的处理方法，可以得到其 Zigzag 编码为

从上面的结果来看，无论是正数还是负数，经过 Zigzag 编码以后，数字高位都是 0，这样以来，便可以进一步使用 Varints 编码进行数据压缩，即 Zigzag 编码在 Protobuf 中并不单独使用，而是配合 Varints 编码共同来进行数据压缩，Google 在 Protobuf 的官方文档中写道：

> If you use `int32` or `int64` as the type for a negative number, the resulting varint is *always ten bytes long* – it is, effectively, treated like a very large unsigned integer. If you use one of the signed types, the resulting varint uses ZigZag encoding, which is much more efficient。

逆向原理不做进一步引申了，ZigZag的逆函数 = `(n>>1)^ -(n&1)`，有兴趣参考：《小而巧的数字压缩算法：zigzag》
`https://blog.csdn.net/zgwangbo/article/details/51590186`

sint32 -> Zigzag 编码 +varints编码合起来

同样是表示-5，sint32只需要2个字节，int32需要11字节。

5.3 Protobuf 的数据组织

在上面的讨论中，我们了解了 Protobuf 所使用的 Varints 编码和 Zigzag 编码的编码原理，本节我们来讨论 Protobuf 的数据组织方式，首先来看一个例子，假设客户端和服务端使用 protobuf 作为数

据交换格式, proto 的具体定义为

Basic 复制代码

```
syntax = "proto3";
package pbTest;

message Request {
    int32 age = 1;
}
```

Request 中包含了一个名称为 name 的字段, 客户端和服务端双方都用同一份相同的 proto 文件是没有任何问题的, 假设客户端自己将 proto 文件做了修改, 修改后的 proto 文件如下

Basic 复制代码

```
syntax = "proto3";
package pbTest;

message Request {
    int32 age_test = 1;
}
```

在这种情形下, 服务端不修改应用程序仍能够正确地解码, 原因在于序列化后的 Protobuf 没有使用字段名称, 而仅仅采用了字段编号, 与 json xml 等相比, Protobuf 不是一种完全自描述的协议格式, 即接收端在没有 proto 文件定义的前提下是无法解码一个 protobuf 消息体的, 与此相对的, json xml 等协议格式是完全自描述的, 拿到了 json 消息体, 便可以知道这段消息体中有哪些字段, 每个字段的值分别是什么, 其实对于客户端和服务端通信双方来说, 约定好了消息格式之后完全没有必要在每一条消息中都携带字段名称, Protobuf 在通信数据中移除字段名称, 这可以大大降低消息的长度, 提高通信效率, Protobuf 进一步将通信线路上消息类型做了划分, 如下表所示

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

对于 int32, int64, uint32 等数据类型在序列化之后都会转为 Varints 编码, 除去两种已标记为 deprecated 的类型, 目前 Protobuf 在序列化之后的消息类型(wire-type) 总共有 4 种, Protobuf 除了存储字段的值之外, 还存储了字段的编号以及字段在通信线路上的格式类型(wire-type), 具体的存储方式为

field_num << 3 | wire type

即将字段标号逻辑左移 3 位, 然后与该字段的 wire type 的编号按位或, 在上表中可以看到, wire type 总共有 6 种类型, **因此可以用 3 位二进制来标识, 所以低 3 位实际上存储了其后所跟的数据的 wire type**, 接收端可以利用这些信息, 结合 proto 文件来解码消息结构体, 我们以上面 proto 为例来看一段 Protobuf 实际序列化之后的完整二进制数据, 假设 age 为 5, 由于 age 在 proto 文件中定义的是 int32 类型, 因此序列化之后它的 wire type 为 0, 其字段编号为 1, 因此按照上面的计算方式, 即 1 << 3 | 0, 所以其类型和字段编号的信息只占 1 个字节, 即 00001000, 后面跟上字段值 5 的 Varints 编码, 所以整个结构体序列化之后为

有了字段编号和 wire type, 其后所跟的数据的长度便是确定的, 因此 Protobuf 是一种非常紧密的数据组织格式, 其不需要特别地加入额外的分隔符来分割一个消息字段, 这可大大提升通信的效率, 规避冗余的数据传输。

1.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。

2.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。

3.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。

filed_num范围：

1. **1到15，仅使用1bytes**。每个byte包含两个部分，一个是field_number一个是tag，其中field-number就是protobuf中每个值后等号后的数字（在C++和Java中，如果不设置这个值，则它是随机的，如果在Python中，不设置，它则不被处理（这个在message binary format中的Field Order一节中有提到）。那么我们可以认为这个field_number是必须的。那么一个byte用来表达这个值就是0**00000000**，其中红色表示是否有后续字节，如果为0表示没有也就是这是一个字节，蓝色部分表示field-number，绿色部分则是wire_type部分，表示数据类型。也就是(field_number << 3) | wire_type。其中wire_type只有3位，表示数据类型。那么能够表示field_number的就是4位蓝色的数字，4位数字能够表达的最大范围就是1-15（其中0是无效的）。
2. **16到2047**，与上面的规则其实类似，下面以2bytes为例子，那么就有1**0000000** 0**0000000**，其中红色部分依然是符号位，因为每个byte的第一位都用来表示下一byte是否和自己有关，那么对于>1byte的数据，第一位一定是1，因为这里假设是2byte，那么第二个byte的第一位也是红色，刨除这两位，再扣掉3个wire_type位，剩下11位（2*8-2-3），能够表达的数字范围就是2047（2^{11}）。

当filed_num > 15时， 以16、64、65为例，value值为1

int32 n1_int32 = 16;

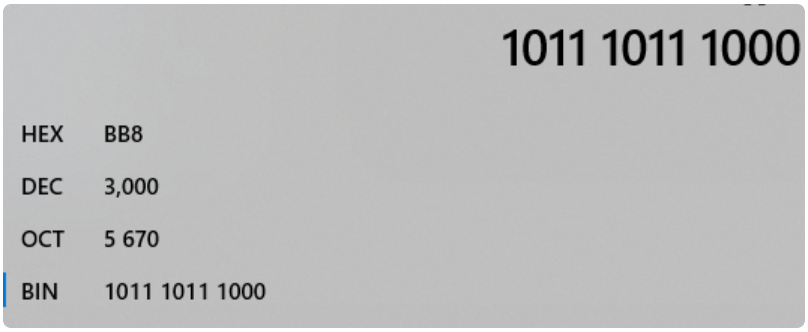
field_number 和 wire_type的关系为 有效位为红色，wire_type为绿色，field_number为蓝色：

16 -> 80 01 01 二进制 1**0000000** **0000001** 00000001 即是 **0000001** 0000 **-> 得出来16**

64 -> 80 04 01 二进制 1**0000000** **0000100** 00000001 即是 **0000100** 0000 **-> 得出来64**

65 -> 88 04 01 二进制 1**0001000** **0000100** 00000001 即是 **0000100** 0001 **-> 得出来65**

3000-> c0 bb 01 01 二进制 1**1000000** 1**0111011** **0000001** 00000001 即是：**0000001 0111011** 1000



5.4 编码总结

1. 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
2. Protobuf 不是完全自描述的信息描述格式，接收端需要有相应的解码器(即 proto 定义)才可解析数据格式，序列化后的 Protobuf 数据不携带字段名，只使用字段编号来标识一个字段，因此更改 proto 的字段名不会影响数据解析(但这显然不是一种好的行为)，字段编号会被编码进二进制的消息结构中，因此我们应尽可能地使用小字段编号
3. 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。

更详细的说明见：codec_test源码。

6. protobuf协议消息升级

如果后面发现之前定义 message 需要增加字段了，这个时候就体现出 Protocol Buffer 的优势了，不需要改动之前的代码。不过需要满足以下 10 条规则：

1. 不要改动原有字段的数据结构。

2. 如果您添加新字段，则任何由代码使用“旧”消息格式序列化的消息仍然可以通过新生成的代码进行分析。您应该记住这些元素的默认值，以便新代码可以正确地与旧代码生成的消息进行交互。同样，由新代码创建的消息可以由旧代码解析：旧的二进制文件在解析时会简单地忽略新字段。（具体原因见 未知字段 这一章节）
3. 只要字段号在更新的消息类型中不再使用，字段可以被删除。您可能需要重命名该字段，可能会添加前缀“OBSOLETE_”，或者标记成保留字段号 `reserved`，以便将来的 `.proto` 用户不会意外重复使用该号码。

Bash 复制代码

```
syntax "proto3";

message Stock {
    reserved 3, 4; //通过，隔开
    int32 id = 1;
    string symbol = 2;
}

message Info {
    reserved 2, 9 to 11, 15; // 可以通过to指定连续返回
    // ...
}
```

4. int32，uint32，int64，uint64 和 bool 全都兼容。这意味着您可以将字段从这些类型之一更改为另一个字段而不破坏向前或向后兼容性。如果一个数字从不适合相应类型的线路中解析出来，则会得到与在 C++ 中将该数字转换为该类型相同的效果（例如，如果将 64 位数字读为 int32，它将被截断为 32 位）。
5. sint32 和 sint64 相互兼容，但与其他整数类型不兼容。
6. 只要字节是有效的UTF-8，string 和 bytes 是兼容的。
7. 嵌入式 message 与 bytes 兼容，如果 bytes 包含 message 的 encoded version。
8. fixed32与sfixed32兼容，而fixed64与sfixed64兼容。
9. enum 就数组而言，是可以与 int32，uint32，int64 和 uint64 兼容（请注意，如果它们不适合，值将被截断）。但是请注意，当消息反序列化时，客户端代码可能会以不同的方式对待它们：例如，未识别的 proto3 枚举类型将保留在消息中，但消息反序列化时如何表示是与语言相关的。（这点和语言相关，上面提到过了）Int 域始终只保留它们的值。
10. 将单个**值**更改为新的成员是安全和二进制兼容的。如果您确定一次没有代码设置多个**字段**，则将多个字段移至新的字段可能是安全的。将任何**字段**移到现有字段中都是不安全的。（注意字段和值的区别，字段是 field，值是 value）

补充知识

原码、反码、补码，计算机中负数的表示

原码：将一个整数，转换成二进制，就是其原码。
如单字节的5的原码为：0000 0101；–5的原码为1000 0101。

反码：正数的反码就是其原码；负数的反码是将原码中，除符号位以外，每一位取反。
如单字节的5的反码为：0000 0101；–5的反码为1111 1010。

补码：正数的补码就是其原码；负数的反码+1就是补码。
如单字节的5的补码为：0000 0101；–5的原码为1111 1011。

在计算机中，正数是直接用原码表示的，如单字节5，在计算机中就表示为：0000 0101。
负数用补码表示，如单字节–5，在计算机中表示为1111 1011。

这儿就有一个问题，为什么在计算机中，负数用补码表示呢？为什么不直接用原码表示？如单字节–5：1000 0101。

拿单字节整数来说，无符号型，其表示范围是[0,255]，总共表示了256个数据。有符号型，其表示范围是[–128,127]。
（1）先看无符号，0表示为0000 0000，255表示为1111 1111，刚好满足了要求，可以表示256个数据。
（2）再看有符号的，若是用原码表示，0表示为0000 000。因为咱们有符号，所以应该也有个负0（虽然它还是0）：1000 0000。

那我们看看这样还能够满足我们的要求，表示256个数据么？
正数，没问题，127是0111 1111，1是0000 0001，当然其它的应该也没有问题。
负数呢，–1是1000 0001，那么把负号去掉，最大的数是111 1111，也就是127，所以负数中最小能表示的数据是–127。
这样似乎不太对劲，该如何去表示–128？貌似直接用原码无法表示，而我们却有两个0。
如果我们把其中的一个0指定为–128，不行么？这也是一个想法，不过有两个问题：一是它与–127的跨度过大；二是在用硬件进行运算时不方便。
所以，计算机中，负数是采用补码表示。
如 单字节–1，原码为1000 0001，反码为1111 1110，补码为1111 1111，计算机中的单字节–1就表示为1111 1111。

单字节–127，原码是1111 1111，反码1000 0000，补码是1000 0001，计算机中单字节–127表示为1000 0001。

单字节-128，原码貌似表示不出来，除了符号为，最大的数只能是127了，其在计算机中的表示为10000000。

【翻译】Protobuf比JSON性能更好

https://zhuanlan.zhihu.com/p/53339153

Protobuf 的 proto3 与 proto2 的区别

https://solicomo.com/network-dev/protobuf-proto3-vs-proto2.html