

# Protocol Buffer

## Protocol Buffer

Protocol buffers是google使用的一种结构化数据序列化编码解码方式，采用简单的二进制格式，他比XML、JSON格式体积更小，编码解码效率更高 下面是项目官方网站与XML对比的描述：**# are 3 to 10 times smaller # are 20 to 100 times faster** 这里有一个.NET环境下的对比测试：[Results of Northwind database rows serialization benchmarks](#)，用的是.NET下面的实现ProtoBuf.net [protobuf项目\(C++\)](#)，.NET下的实现有：[protobuf-net](#)、[protobuf-csharp-port](#)。另外一个.NET的项目是[Proto#](#)，不过作者似乎没有维护了

## 使用方式简介

首先定义消息类型：`message Person { required string name = 1; required int32 id = 2; optional string email = 3;`

`enum PhoneType { MOBILE = 0; HOME = 1; WORK = 2; }`

`message PhoneNumber { required string number = 1; optional PhoneType type = 2 [default = HOME]; }`

`repeated PhoneNumber phone = 4; }` Field Rules: 属性规则，required: 必须的属性；optional: 可选属性；

repeated: 可重复多个的属性 Field Type: 属性数据类型，标量值类型（scalar value types）支持double, float, int32, int64, uint32, uint64, sint32, sint64, fixed32, fixed64, bool, string, bytes等，另外支持枚举、嵌套/引用的消息类型等

Field Tags: 属性标签（例如name=1中的1），使用正整数表示，在序列化的二进制中使用这个标签来标记属性，比使用属性名称体积更小 详细的语法参考官方网站：[Language Guide](#)

消息类型定义在.proto文件中，使用protoc.exe根据.proto文件生成C++、Java、Python等类文件，这些类文件中定义了表示消息的对象，以及用于编码、解码的方法

体积方面，首先从上面消息类型的定义中可以看出，使用属性标签代替属性名称可以减小体积，另外在编码协议上对各种数据类型的处理，也尽量采用了压缩的表示方式以减小体积。速度方面，二进制协议比基于文本的解析更有优势

## 编码协议简介 - 2.3.0

详细的编码协议参考官方网站的[Encoding Base 128 Varints](#) 32位整数使用4字节存储，32位的整数值1同样要使用4个字节，比较浪费空间。Varint采用变长字节的方式存储整数，将高位为0的字节去掉，节约空间 高位为0的字节去掉以后，用来存储整数的每一个字节，其最高有效位（most significant bit）用作标识位，0表示这是整数的最后一个字节，1表示不是最后一个字节；其他7位用于存储整数的数值。字节序采用little-endian 示例：整数1，Varint的二进制值为0000 0001。因为1个字节就足够，所以最高有效位为0，后7位则为1的原码形式 整数300，Varint需要2字节表示，二进制值为1010 1100 0000 0010。第一个字节最高有效位设为1，最后一个字节最高有效位设为0。解码过程如下：a). 首先每个字节去掉最高有效位，得到：010 1100 000 0010 b). 按照little-endian方式处理字节序，得到：000 0010 010 1100 c). 二进制值100101100即为300

**ZigZag**编码 Varint对于无符号整数有效，对负数无法进行压缩，protocol buffer对有符号整数采用ZigZag编码后，再以varint形式存储 对32位有符号数，ZigZag编码算法为  $(n < 1) \wedge (n > 31)$ ，对64位有符号数的算法为  $(n < 1) \wedge (n > 63)$  注意：32位有符号数右移31位后，对于正数所有位为0，对于负数所有位为1 编码后的效果是0=>0, -1=>1, 1=>2, -2=>3, 2=>4.....，即将无符号数编码为有符号数表示，这样就能有效发挥varint的优势了

Protocol buffer用32位表示float和fixed32，用64位表示double和fixed64 String, bytes, 嵌入式消息等数据均采用定长数据类型（length-delimited）表示，这类数据在开始位置使用一个varint表示数据的字节长度，后面接着是数据值 消息结构 消息的所有属性都序列化为key-value pair（键-值对）的字节流形式，字节流中不包含属性的名称和声明的类型，这些信息必须从定义的消息类型中获取 key里面包含2个东西，一个是在消息类型里面为该属性指定的field tag，另一个是protocol buffer协议的封装类型（wire type）。这2个部分都是正整数，使用  $(field\_tag < 3) | wire\_type$  方式生成一个正整数，然后使用base 128 varint方式表示。key后面跟着是属性的值 wire type: `<!--[endif]->`

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

示例：消息类型如下 `message Test1 { required int32 attr = 1; }` 创建一个Test1的对象，将其属性attr的值设置为150，则对该对象编码过程如下 属性数据类型为int32，其wire type为0，所以key值为  $(1 \ll 3) | 0 \Rightarrow 0000\ 1000$  属性值150采用Varint编码  $150 \Rightarrow 10010110$  //二进制  $\Rightarrow 000\ 0001\ 001\ 0110$  //7位一组分开  $\Rightarrow 001\ 0110\ 000\ 0001$  //little-endian字节序  $\Rightarrow 1001\ 0110\ 0000\ 0001$  //设置最高标识位  $\Rightarrow 96\ 01$  //16进制 所以这个Test1对象编码后的16进制值为：08 96 01

如果有嵌入式消息类型定义如下 `message Test3 { required Test1 c = 3; }` 编码后的16进制值形如：1A 03 08 96 01，其中08 96 01就是上面示例的Test1对象，在Test3的属性中他与字符串的处理方式一样，前面的03就是表示其长度的varint

## protobuf-csharp-port的使用方式

protobuf-csharp-port跟protobuf的使用方式一样，即在开发过程中使用protoc.exe、ProtoGen.exe生成用于序列化、反序列化时的消息对象，在运行时通过这些对象进行编码解码 从[GitHub](#)下载项目源代码（目前还没有发布包），项目中带有示例AddressBook 生成消息通讯用的C#类分2个步骤 步骤1：使用lib目录下的protoc.exe生成二进制表示 `protoc --descriptor_set_out=addressbook.protobin --proto_path=..\protos --include_imports ..\protos\tutorial\addressbook.proto` 步骤2：使用编译生成的ProtoGen.exe从二进制表示生成C#类 ProtoGen.exe addressbook.protobin 会生成几个.cs文件，其中包括AddressBookProtos.cs，这个就是在addressbook.proto中定义的消息类型 运行时的项目需要引用编译生成的Google.ProtocolBuffers.dll，使用AddressBookProtos.cs完成编码解码操作，详细用法查看示例项目AddressBook 运行AddressBook.exe如下图：

```

C:\protobuf>AddressBook
Options:
  L: List contents
  A: Add new person
  Q: Quit
Action? a
Enter person ID: 8900021
Enter name: Richie
Enter email address (blank for none): xxxx@gmail.com
Enter a phone number (or leave blank to finish): 13899999999
Is this a mobile, home, or work phone? mobile
Enter a phone number (or leave blank to finish): 02188888888
Is this a mobile, home, or work phone? home
Enter a phone number (or leave blank to finish):

Options:
  L: List contents
  A: Add new person
  Q: Quit

```

输入的对象序列化为二进制后，默认

保存在addressbook.data文件中，可以使用ProtoDump.exe读取这个二进制文件：

```

C:\protobuf>ProtoDump.exe "Google.ProtocolBuffers.Examples.AddressBook.AddressBo
ok, AddressBook" addressbook.data
person {
  name: "Richie"
  id: 8900061
  email: "xxxxx@sina.com"
  phone {
    number: "13888888888"
    type: MOBILE
  }
  phone {
    number: "02199999999"
  }
  phone {
    number: "home"
    type: HOME
  }
}
person {
  name: "RicCC"
  id: 6000192
  email: "riccc.cn@gmail.com"
  phone {
    number: "13666666666"
    type: MOBILE
  }
}
}

```

## protobuf-net的使用方式 - r282

protobuf-net的使用与Google的protobuf完全不一样，他采用.NET的编程方式，可以非常方便的在.NET的序列化场景下使用，支持WCF的DataContract，WCF程序几乎不需要什么修改就能使用protobuf-net 下载protobuf-net，项目引用protobuf-net.dll，测试对象定义如下：

```

1  [ProtoContract]
2  public class TestObject
3  {
4      [ProtoMember(1)]

```

```

5         public string StringAttr1 {
6         get; set; }
7         [ProtoMember(2)]
8         public string StringAttr2 {
9         get; set; }
10        [ProtoMember(3)]
11        public int IntAttr { get; set;
12        }
13        [ProtoMember(4)]
14        public long LongAttr { get;
15        set; }
16        [ProtoMember(5)]
17        public decimal DecimalAttr {
18        get; set; }
19        [ProtoMember(6)]
20        public float FloatAttr { get;
21        set; }
22        [ProtoMember(7)]
23        public int[] ArrayAttr { get;
24        set; }
25        [ProtoMember(8)]
26        public IList<string> ListAttr
27        { get; set; }
28        [ProtoMember(9)]
29        public InnerObject
30        EmbeddedAttr { get; set; }
31        public override string ToString()
32        {
33            {
34                StringBuilder sb =
35                new StringBuilder()
36                .Append("TestObject
37                {\r\n")
38                .Append("
39                StringAttr1:
40                \").Append(this.StringAttr1).Append(
41                "\",\r\n")
42                .Append("
43                StringAttr2:
44                \").Append(this.StringAttr2).Append(
45                "\",\r\n")
46                .Append("    IntAttr:
47                ").Append(this.IntAttr).Append(",\r\n")
48                .Append("    LongAttr:
49                ").Append(this.LongAttr).Append(",

```

```

51     \r\n")
52         .Append("
53     DecimalAttr:
54 ").Append(this.DecimalAttr).Append
55     (" ,\r\n")
56         .Append("    FloatAttr:
57 ").Append(this.FloatAttr).Append("
58     ,\r\n");
59     if (this.ArrayAttr !=
60     null)
61     {
62         sb.Append("
63     ArrayAttr: [ ");
64         foreach (int i
65     in this.ArrayAttr)
66     sb.Append(i).Append(" , ");
67         sb.Remove(sb.Length -
68     2, 2);
69         sb.Append(" ],\r\n");
70     }
71     if (this.ListAttr != null)
72     {
73         sb.Append("
74     ListAttr: [ ");
75         foreach (string s
76     in this.ListAttr)
77     sb.Append(' ').Append(s).Append(" \
78     ", " ");
79         sb.Remove(sb.Length -
80     2, 2);
81         sb.Append(" ],\r\n");
82     }
83     if (this.EmbeddedAttr !=
84     null)
85     sb.Append("
86     EmbeddedAttr:
87 ").Append(this.EmbeddedAttr.ToStri
88     ng()).Append("\r\n");
89     return sb.Append("}").ToSt
90     ring();
91     }
92 }
93 [ProtoContract]
94 public class InnerObject
95 {
96     [ProtoMember(1)]

```

```

        public string Attr1 { get;
set; }
        [ProtoMember(2)]
        public DateTime Attr2 { get;
set; }
        [ProtoMember(3)]
        public bool Attr3 { get; set;
}
        [ProtoMember(6)]
        public byte Attr4 { get; set;
}
        [ProtoMember(9)]
        public sbyte Attr5 { get; set;
}
        public override string ToString()
        {
            return new StringBuilder()
                .Append("{\r\n")
                .Append("        Attr1:
") .Append(this.Attr1).Append("\r\n")
                .Append("        Attr2:
") .Append(this.Attr2.ToString("y
yy-MM-dd")).Append("\r\n")
                .Append("        Attr3:
") .Append(this.Attr3).Append("\r\n")
                .Append("        Attr4:
") .Append(this.Attr4).Append("\r\n")
                .Append("        Attr5:
") .Append(this.Attr5).Append("\r\n")
                .Append("
    }).ToString();
        }
    }
}

```

测试代码如下：

```

1  using (MemoryStream ms =
2  new MemoryStream())
3  {
4      TestObject obj =
5  new TestObject()

```

```
6      {
7          StringAttr1 = "string 1",
8          StringAttr2 = "string 2",
9          IntAttr = 300,
10         LongAttr = 1,
11         DecimalAttr = 34.10091M,
12         FloatAttr = 12.3f,
13         ArrayAttr = new int[] {
14 600, -9, 0 },
15         ListAttr =
16 new List<string> { "string 3",
17 "string 5" },
18         EmbeddedAttr =
19 new InnerObject()
20     {
21         Attr1 = "string 6",
22         Attr2 =
23 new DateTime(2010, 2, 1),
24         Attr3 = false,
25         Attr4 = 8,
26         Attr5 = -63
27     }
28 };
    Serializer.Serialize<TestObject>(ms, obj);
    ms.Flush();
    ms.Position = 0;
    TestObject obj2 =
    Serializer.Deserialize<TestObject>(ms);
    Console.WriteLine(obj2);
    Console.ReadKey();
}
```

```

TestObject {
  StringAttr1: "string 1",
  StringAttr2: "string 2",
  IntAttr: 300,
  LongAttr: 1,
  DecimalAttr: 34.10091,
  FloatAttr: 12.3,
  ArrayAttr: [ 600, -9, 0 1,
  ListAttr: [ "string 3", "string 5" ],
  EmbeddedAttr: {
    Attr1: "string 6",
    Attr2: "2010-02-01",
    Attr3: False,
    Attr4: 8,
    Attr5: -63
  }
}

```

运行结果:

## 附录

原码、反码、补码 对有符号数，最高位是符号位。正数的原码反码和补码都是一样的，就是本身。负数的反码是原码求反，补码是反码加1。例如-1的原码是1000 0001，反码是1111 1110，补码是1111 1111。负数都是用补码表示，从正数的原码推负数的二进制表示（补码）时，只须将正数各个位（包括符号位）取反加1 补码有2种，即one's complement (1's complement, 1的补码) 和 two's complement (2's complement, 2的补码)。按照定义，one's complement就是对各个位取反，two's complement是对各个位取反后加1。例如在8位处理器情况下，9的二进制是0000 1001，one's complement是1111 0110，two's complement是1111 0111 采用one's complement表示负数时存在正0 (0x00)和负0 (0xff)，并且有符号数相加必须采用end-around carry（循环进位）处理，例如

$$\begin{array}{r}
 \begin{array}{r}
 \text{0001 0001} \quad (17) \\
 + \quad \text{1111 0111} \quad (-8) \\
 \hline
 \text{1 0000 1000} \\
 \hookrightarrow \quad + 1 \\
 \hline
 \text{0000 1001} = (9)
 \end{array}
 \end{array}$$

相加之后发生溢出，则必须将溢出位加到最低位上，这样导致有符号数相加和无符号数相加算法不一致，而采用two's complement表示时不存在这些问题 关于2的补码表示可以参考阮一峰的[关于2的补码](#)一文，更专业的说明可以参考wikipedia上的[Method of complements](#): 二进制的基数补码（radix complement）叫做2的补码，二进制的基数减一补码（diminished radix complement）叫做1的补码；十进制的基数补码叫做10的补码，基数减一补码叫做9的补码

转自:<http://www.cnblogs.com/RicCC/archive/2010/03/10/protocol-buffers.html>

分类: [.net](#)

标签: [Protocol Buffer](#)