Google Protobuf编码技术

May 05, 2015

Google Protocol Buffer是Google於2008年開源的一款非常優秀的**序列化反序列化工具**，它最突出的特點是輕便簡介，而且有很多語言的接口（官方的支持C++，Java，Python，C，以及第三方的Erlang, Perl等）。

关于序列化的定义，摘自WikiPedia：

> *In computer science, in the context of data storage, serialization is the process of translating data structures or object state into a format that can be stored (for example, in a file or memory buffer, or transmitted across a network connection link) and reconstructed later in the same or another computer environment. When the resulting series of bits is reread according to the serialization format, it can be used to create a semantically identical clone of the original object.*

个人理解：序列化是把以某种数据结构储存的信息转换成基于某种便于传输的格式组织的数据，目的是为了便于传输。反序列化是按照某种格式读取数据，再将其转换为原来的数据结构格式。既然是用于传输目的的格式，Protocol必然会对其进行数据压缩。

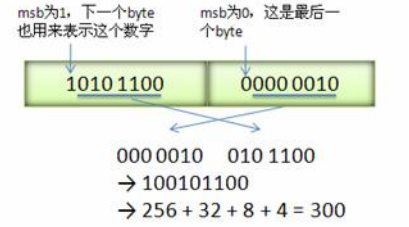
本文以int32和string为例子,探讨了Protobuf将特定结构体序列化，变为二进制数据的技术。

Varint

Varint是一种紧凑的数字表示方法，它可以用一个或多个字节表示一个数字。数字的值越小，表示需要的字节也就越少。

Varint中每个byte的最高位bit有特殊含义，0代表此字节是数据最后byte，1代表后面的byte仍是数据的一部分。每个byte剩余的7bits用于表示数据的大小。

以一个例子为例：300，以varint表示的话是两个字节： 1010 1100 0000 0010



整个过程如下：

1. 首先将300以二进制表示出来：0001 0010 1100。
2. 从后往前，每7位加上标识是否最后字节的最高bit(以红色标出)组成新的序列：0000 0010 1010 1100。标识位的值可能感到与直觉相反，但请考虑到数据的存储是依据**小端规则(little endian)**进行的，即最低字节存储在低地址单元。从数据存储的角度来看，最低字节反而是数据开始的首字节。

由此，整形300的话我们以两个字节就能够表示，无需用到四个字节或八个字节来存储。

从这里我们也可以看出，**压缩的原理是用标识位来取代了用于高字节占位的若干个0**，数据的值越小，表示需要的字节也就越少,因为值越小，用于占位的0也就越多，小于128的整数只需一个字节就能表示。很容易想到，这种压缩技术应用在值非常大的整形或者负数会有反效果，反而增加了需要表示的字节数，**因为省去的高位占位的0数目小于增加的标志位数目**,在这种情况下Protobuf有别的处理方法，这里不详述。

Protobuf正是用这种编码技术对整型数据进行压缩，附上Protocal Buffer对于int32的Varint示例代码和简单分析。

```
inline uint8* CodedOutputStream::WriteVarint32FallbackToArrayInline(
    uint32 value, uint8* target) {
  target[0] = static_cast<uint8>(value | 0x80);
  if (value >= (1 << 7)) {
    target[1] = static_cast<uint8>((value >>  7) | 0x80);
    if (value >= (1 << 14)) {
      target[2] = static_cast<uint8>((value >> 14) | 0x80);
      if (value >= (1 << 21)) {
        target[3] = static_cast<uint8>((value >> 21) | 0x80);
        if (value >= (1 << 28)) {
          target[4] = static_cast<uint8>(value >> 28);
          return target + 5;
        } else {
          target[3] &= 0x7F;
          return target + 4;
        }
      } else {
        target[2] &= 0x7F;
        return target + 3;
      }
    } else {
      target[1] &= 0x7F;
      return target + 2;
    }
  } else {
    target[0] &= 0x7F;
    return target + 1;
  }
}
```

具体的实现是以一个 `uint8` 的数组 `target` 存储每一个字节，每次使适当右移后(使特定的某7位成为低7位)的值的低七位与1000 0000按位或(最高位1表示默认不是最后一个字节)，存储结果到 `target` ，最后使最后一个字节与0111 1111按位与，置最高位为0表结束，最后返回指向最后一个字节的指针 `target` 。

Protocal Buffer Message

在PB中，有一种proto类型的文件，要求用户自定义信息逻辑单元*Message*，即一个结构体。下面举例说明一个简单的Hello.proto文件，。

```
message Hello {
    required int32 id = 1;
    required float flt = 2;
    required string str = 3;
    optional int32 opt = 4;
}
```

首先每个结构体需要用关键字*Message*描述，其中每个字段的修饰符有 `required` ， `repeated` ， `optional` 三种，required表示該字段是必須的，repeated表示該字段可以重复出現，它描述的字段可以看做C語言中的數組，optional表示該字段可有可無。

同时，必须人为地赋予每一个字段一个标号**field_number**，如图中的1,2,3,4所示。

Protocol Buffer的数据类型

Protocol Buffer需要用户自定义自己的数据体，而且结构体的定义要符合google制定的规则。结构体中每个字段都需要一个数据类型，Protocol Buffer支持的数据类型在wire_format_lite.h中定义：

```
 enum WireType {
    WIRETYPE_VARINT           = 0,
    WIRETYPE_FIXED64          = 1,
    WIRETYPE_LENGTH_DELIMITED = 2,
    WIRETYPE_START_GROUP      = 3,
    WIRETYPE_END_GROUP        = 4,
    WIRETYPE_FIXED32          = 5,
  };
```

其中：

VARINT类数据表示要用variant编码对所传入的数据做压缩存储，variant编码细节见上文。

FIXED32和FIXED64类数据不对用户传入的数据做variant压缩存储，只存储原始数据。

LENGTH_DELIMITED类数据主要针对string类型、repeated类型和嵌套类型，对这些类型编码时需要存储他们的长度信息。

START_GROUP是一个组（该组可以是嵌套类型，也可以是repeated类型）的开始标志。

END_GROUP是一个组（该组可以是嵌套类型，也可以是repeated类型）的结束标志。

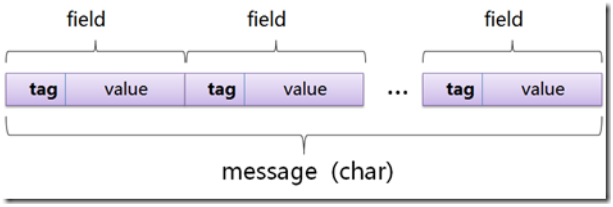
每类数据包含的具体数据类型如下表所示：

WireType	表示类型
VARINT	int32,int64,uint32,uint64,sint32,sint64,bool,enum
FIXED64	fixed64,sfixed64,double
LENGTH_DELIMITED	string,bytes,embedded messages, packed repeadted field
START_GROUP	group的开始标志
END_GROUP	group的结束标志
FIXED32	fixed32,sfixed32,float

Protocal Buffer的编码

有了上述预备知识后，我们开始正式研究PB的编码。**Protobuf的编码尽其所能地将字段的元信息和字段的值压缩存储，并且字段的元信息中含有对这个字段描述的所有信息。**

整個结构体序列化后，如下图：



可以看到，整個消息是以二进制流的方式存储，在這個二进制流中，逐個字段以定义的顺序紧紧相邻。每個字段中由标签tag和字段的值value組成。

其中tag是這样编码的：

1. field_number << 3 | wire_type(wire_type共有6种，因此需要低三位来表示，tag描述了字段的元信息)
2. 对上面得到的无符号类型整數做varint编码

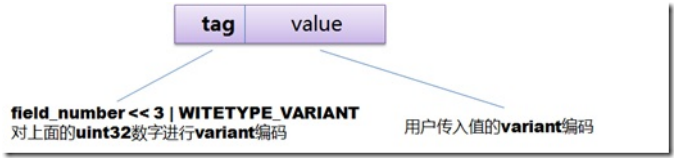
其中field_number是定义*Message*时，赋予每个字段的标号，wire_type是字段的数据类型。

下面以int32和string为例，看看PB压缩编码的过程。

---

int32类型的编码

1. 计算字段长度 1 + Int32Size(value)。
2. 调用WriteString()写入标签与压缩后的值。



计算字段长度 `1 + Int32Size(value)`;

以下代码片截取自 `Hello.pb.cc`

```
  // required int32 id = 1;
   if (has_id()) {
     total_size += 1 +
       ::google::protobuf::internal::WireFormatLite::Int32Size(
         this->id());
   }
```

调用了 `Int32Size()`,定义如下：

```
inline int WireFormatLite::Int32Size(int32 value) {
  return io::CodedOutputStream::VarintSize32SignExtended(value);
}
```

里面嵌套调用了 `VarintSize32SignExtended(value)` ，声明如下：

```
  // If negative, 10 bytes.  Otherwise, same as VarintSize32().
  static int VarintSize32SignExtended(int32 value);
```

`VarintSize32()`声明如下：

```
  // Returns the number of bytes needed to encode the given value as a varint.
  static int VarintSize32(uint32 value);
```

函数功能已很清晰，返回varint压缩value后的字节数(实现代码已在上文贴出)。

调用 `WriteInt32()`将该字段的标签和字段值写入新空间中：

```
// @@protoc_insertion_point(serialize_to_array_start:Hello)
  // required int32 id = 1;
  if (has_id()) {
    target = ::google::protobuf::internal::WireFormatLite::WriteInt32ToArray(1, this->id(), target);
  }
```

`WriteInt32ToArray()`定义如下

```
inline uint8* WireFormatLite::WriteInt32ToArray(int field_number,
                                                int32 value,
                                                uint8* target) {
  target = WriteTagToArray(field_number, WIRETYPE_VARINT, target);
  return WriteInt32NoTagToArray(value, target);
}
```

```
inline uint8* WireFormatLite::WriteTagToArray(int field_number,
                                              WireType type,
                                              uint8* target) {
  return io::CodedOutputStream::WriteTagToArray(MakeTag(field_number, type),
                                                target);
}
```

里面`MakeTag`实际调用了一个Macro，相关代码如下

```
#define GOOGLE_PROTOBUF_WIRE_FORMAT_MAKE_TAG(FIELD_NUMBER, TYPE)                \
  static_cast<uint32>(                                                          \
    ((FIELD_NUMBER) << ::google::protobuf::internal::WireFormatLite::kTagTypeBits) \
      | (TYPE))
```

```
  // Number of bits in a tag which identify the wire type.
  static const int kTagTypeBits = 3;
```

然后我们得到了`field\_number \<\< 3 \| wire\_type`这个标签，接着调用`WriteTagToArray()`把标签写入`target`指针指向的空间中。

```
inline uint8* CodedOutputStream::WriteTagToArray(
    uint32 value, uint8* target) {
  if (value < (1 << 7)) {
    target[0] = value;
    return target + 1;
  } else if (value < (1 << 14)) {
    target[0] = static_cast<uint8>(value | 0x80);
    target[1] = static_cast<uint8>(value >> 7);
    return target + 2;
  } else {
    return WriteVarint32FallbackToArray(value, target);
  }
}
```

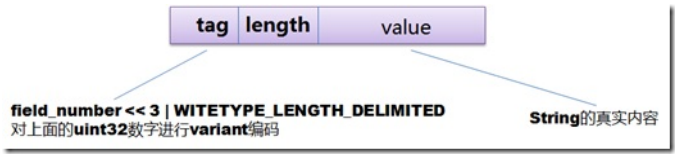
最后调用`WriteInt32NoTagToArray()`把varint(value)也写入`target`指向的空间中，整个编码工作就完结了。

```
void CodedOutputStream::WriteVarint32(uint32 value) {
  if (buffer_size_ >= kMaxVarint32Bytes) {
    // Fast path:  We have enough bytes left in the buffer to guarantee that
    // this write won't cross the end, so we can skip the checks.
    uint8* target = buffer_;
    uint8* end = WriteVarint32FallbackToArrayInline(value, target);
    int size = end - target;
    Advance(size);
  } else {
    // Slow path:  This write might cross the end of the buffer, so we
    // compose the bytes first then use WriteRaw().
    uint8 bytes[kMaxVarint32Bytes];
    int size = 0;
    while (value > 0x7F) {
      bytes[size++] = (static_cast<uint8>(value) & 0x7F) | 0x80;
      value >>= 7;
    }
    bytes[size++] = static_cast<uint8>(value) & 0x7F;
    WriteRaw(bytes, size);
  }
}
```

---

string类型

1. 计算长度 1 + varint(stringLength) + stringLength。
2. 调用WriteInt32()将标签、长度和原串写入内存。



与int32一样，先计算长度，`1 + varint(stringLength) + stringLength`从这里可以看出，PB对String的值不作压缩，因此最终字符串长度比原串要更长。

```
  // required string str = 3;
  if (has_str()) {
    total_size += 1 +
      ::google::protobuf::internal::WireFormatLite::StringSize(
        this->str());
  }
```

计算Varint(字符串长度)与字符串长度。与Int32相比，代码简单不少。

```
inline int WireFormatLite::StringSize(const string& value) {
  return io::CodedOutputStream::VarintSize32(value.size()) +
         value.size();
}
```

```
  // Returns the number of bytes needed to encode the given value as a varint.
  static int VarintSize32(uint32 value);
```

调用`WriteString()`将该字段的标签、长度和值写入内存中，完成string的编码。

```
void WireFormatLite::WriteString(int field_number, const string& value,
                                 io::CodedOutputStream* output) {
  // String is for UTF-8 text only
  WriteTag(field_number, WIRETYPE_LENGTH_DELIMITED, output);
  GOOGLE_CHECK(value.size() <= kint32max);
  output->WriteVarint32(value.size());
  output->WriteString(value);
}
```

这三个函数调用过于冗长，这里不做展开，原理与Int32相近。

```
inline void WireFormatLite::WriteTag(int field_number, WireType type,
                                     io::CodedOutputStream* output) {
  output->WriteTag(MakeTag(field_number, type));
}
```

```
void CodedOutputStream::WriteVarint32(uint32 value) {
  if (buffer_size_ >= kMaxVarint32Bytes) {
    // Fast path:  We have enough bytes left in the buffer to guarantee that
    // this write won't cross the end, so we can skip the checks.
    uint8* target = buffer_;
    uint8* end = WriteVarint32FallbackToArrayInline(value, target);
    int size = end - target;
    Advance(size);
  } else {
    // Slow path:  This write might cross the end of the buffer, so we
    // compose the bytes first then use WriteRaw().
    uint8 bytes[kMaxVarint32Bytes];
    int size = 0;
    while (value > 0x7F) {
      bytes[size++] = (static_cast<uint8>(value) & 0x7F) | 0x80;
      value >>= 7;
    }
    bytes[size++] = static_cast<uint8>(value) & 0x7F;
    WriteRaw(bytes, size);
  }
}
```

```
inline void CodedOutputStream::WriteString(const string& str) {
  WriteRaw(str.data(), static_cast<int>(str.size()));
}
```

Reference

1. https://developers.google.com/protocol-buffers/?hl=zh-cn—Protobuf首页
2. http://www.cnblogs.com/cobbliu/archive/2013/03/02/2940074.html —博客園CobbLiu