深入 ProtoBuf - 序列化源码解析

抽奖

深入 ProtoBuf - 序列化源码解析

在上一篇 深入 ProtoBuf - 编码 中，我们详细解析了 ProtoBuf 的编码原理。

有了这个知识储备，我们就可以深入 ProtoBuf 序列化、反序列化的源码，从代码的层面理解 ProtoBuf 具体是如何实现对数据的编码（序列化）和解码（反序列化）的。

我们重新复习一下， ProtoBuf 的序列化使用过程：

- 定义 .proto 文件
- protoc 编译器编译 .proto 文件生成一系列接口代码
- 调用生成的接口实现对 .proto 定义的字段的读取以及 message 对象的序列化、反序列化方法

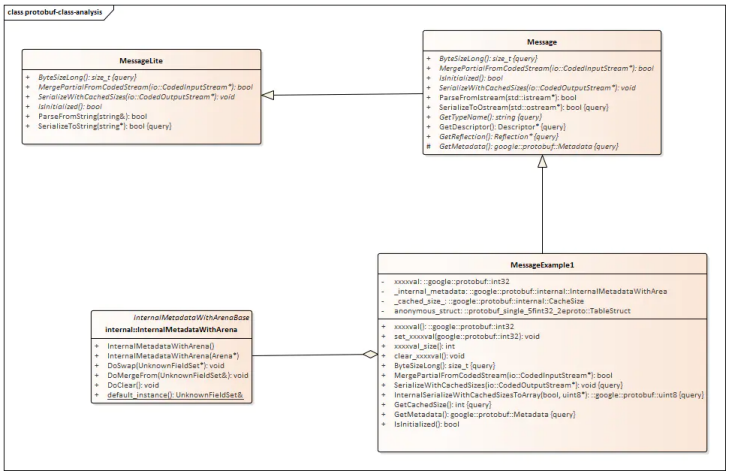
具体调用代码如下：

```
Example1 example1;
example1.set_int32val(val);
example1.set_stringval("hello,world");
example1.SerializeToString(&output);
```

调用 SerializeToString 函数将 example1 对象序列化（编码）成字符串。我们的目的就是了解 SerializeToString 函数里到底发生了什么，是怎么一步一步得到最终的序列化结果的。

> 注意：并非编码成字符串数据，string 只是作为编码结果的容器

我们在 .proto 文件中定义的 message 在最终生成的对应语言的代码中，例如在 C++（xxxx.pb.h、xxxx.pb.cpp）中每一个在 .proto 文件中定义的 message 字段都会在代码中构造成一个类，且这些 message 消息类继承于 ::google::protobuf::Message，而 ::google::protobuf::Message 继承于一个更为轻量的 MessageLite 类。其相关的类图如下所示：



protobuf-class-analysis.png
而我们经常调用的序列化函数 SerializeToString 并定义在基类 MessageLite 中。

编码

当某个 Message 调用 SerializeToString 时，经过一层层调用最终会调用底层的关键编码函数 WriteVarint32ToArray 或 WriteVarint64ToArray，整个过程如下图所示：

ProtoBuf 序列化_反序列化时序图.png
WriteVarint32ToArray 函数可在源码目录下的 google.protobuf.io 包下的 coded_stream.h 中找到。在上一篇 深入 ProtoBuf - 编码 中我们解析了 Varint 编码原理和详细过程，WriteVarint32ToArray（以及 WriteVarint64ToArray）便是 Varint 编码的核心。

可以对照上一篇指出的 Varints 编码的几个关键点来阅读以下代码，可以看出编码实现确实优雅，代码如下：

```
inline uint8* CodedOutputStream::WriteVarint32ToArray(uint32 value, uint8* targe
  // 0x80 -> 1000 0000
  // 大于 1000 0000 意味这进行 Varints 编码时至少需要两个字节
  // 如果 value < 0x80，则只需要一个字节，编码结果和原值一样，则没有循环直接返回
  // 如果至少需要两个字节
  while (value >= 0x80) {
    // 如果还有后续字节，则 value | 0x80 将 value 的最后字节的最高 bit 位设置为 1，并
    *target = static_cast<uint8>(value | 0x80);
    // 处理完七位，后移，继续处理下一个七位
    value >>= 7;
    // 指针加一，（数组后移一位）
    ++target;
  }
  // 跳出循环，则表示已无后续字节，但还有最后一个字节

  // 把最后一个字节放入数组
  *target = static_cast<uint8>(value);
```

```
  // 结束地址指向数组最后一个元素的末尾
  return target + 1;
}

// Varint64 同理
inline uint8* CodedOutputStream::WriteVarint64ToArray(uint64 value,
                                                      uint8* target) {
  while (value >= 0x80) {
    *target = static_cast<uint8>(value | 0x80);
    value >>= 7;
    ++target;
  }
  *target = static_cast<uint8>(value);
  return target + 1;
}
```

在上面已添加详细注释，这里再强调几个关键点。

- **value | 0x80**：xxx ... xxxx xxxx | 000 ... 1000 0000 的结果其实就是将最后一个字节的第一个 bit（最高位） 置 1，其他位不变，即 xxx ... 1xxx xxxx。注意 target 是 uint8 类型的指针，这意味它只会截断获取最后一个字节，即 1xxx xxxx，这里的 1 意味着什么？这个 1 就是所谓的 msb 了，意味着后续还有字节。之后就是右移 7 位（去掉最后 7 位），处理下一个 7位。
- 通过这里的代码应该可以体会到为什么 Varints 编码结果是低位排在前面了。

了解了最底层 IO 包中的编码函数，再结合上篇文章介绍的编码原理，对 ProtoBuf 的编码应该有了更深入的认识。

Varints 类型序列化实现

int32、*int64*、*uint32*、*uint64*

int32 类型编码函数对应为 WriteInt32ToArray，源码如下：

```
// WriteTagToArray 函数将 Tag 部分写入
// WriteInt32NoTagToArray 函数将 Value 部分写入
// WriteTagToArray 和 WriteInt32NoTagToArray 底层
// 均调用 coded_stream.h 中的 WriteVarint32ToArray
//因为 ProtoBuf 中的 Tag 均采用 Varint 编码
// int32 的 Value 部分也采用 Varint 编码
inline uint8* WireFormatLite::WriteInt32ToArray(int field_number, int32 value,
                                                uint8* target) {
  target = WriteTagToArray(field_number, WIRETYPE_VARINT, target);
  return WriteInt32NoTagToArray(value, target);
}
```

int64、uint32、uint64 类型与 int32 类型同理，只是处理位数有所不同。

uint32 和 uint64 也是采用 Varint 编码，所以底层编码实现与 int32、int64 一致。

sint32、sint64

这两种类型编码函数对应为 WriteSInt32ToArray 和 WriteSInt64ToArray 。

在上一篇文章 深入 ProtoBuf - 编码 中我们已经介绍过 Varint 编码在负数的情况下编码效率很低，固对于 sint32、sint64 类型我们会采用 ZigZag 编码将负数映射成正数然后再进行 Varint 编码，而这种映射并非采用存储的 Map，而是使用移位实现。sint32 的 ZigZag 源码实现如下：

```
inline uint32 WireFormatLite::ZigZagEncode32(int32 n) {
  // 右移为算数右移
  // 左移时需要先将 n 转成 uint32 类型，防止溢出
  // 当 n 为正数时 result = 2 * n
  // 当 n 为负数时 result = - (2 * n + 1)
  return (static_cast<uint32>(n) << 1) ^ static_cast<uint32>(n >> 31);
}
```

经过 ZigZagEncode32 编码之后，数字成为一个正数，之后等同于 int32 或 int64 进行完全相同的编码处理。

bool 与 enum

bool 和 enum 本质就是整型，编码处理与 int32、int64 相同。

32-bit、64-bit

fixed32/fixed64

fixed32 类型对应 WriteFixed32ToArray 函数，32-bit、64-bit类型的字段比起上述 Varint 类型则要简单的多，因为每个数字均是固定字节，源码如下：

```
inline uint8* WireFormatLite::WriteFixed32ToArray(int field_number,
                                                  uint32 value, uint8* target) {
  // WriteTagToArray：Tag 依然是 Varint 编码，与上一节 Varint 类型是一致的
  // WriteFixed32NoTagToArray：固定写四个字节即可
  target = WriteTagToArray(field_number, WIRETYPE_FIXED32, target);
  return WriteFixed32NoTagToArray(value, target);
}
```

fixed64 与 fixed32 同理，不再赘述。

sfixed32/sfixed64

sfixed32 类型对应 WriteSFixed32ToArray 函数，源码如下：

```
inline uint8* WireFormatLite::WriteSFixed32ToArray(int field_number,
                                                   int32 value, uint8* target) {
  target = WriteTagToArray(field_number, WIRETYPE_FIXED32, target);
```

```
  return WriteSFixed32NoTagToArray(value, target);
}
```

其中 WriteSFixed32NoTagToArray 源码如下：

```
inline uint8* WireFormatLite::WriteSFixed32NoTagToArray(int32 value,
                                                        uint8* target) {
  return io::CodedOutputStream::WriteLittleEndian32ToArray(
      static_cast<uint32>(value), target);
}
```

由此可知，对于位数固定的 sfixed32 是将其转成 uint32 类型，然后使用与 fixed32 相同的函数写入。

sfixed64 与 sfixed32 同理，不赘述。

Length delimited 字段序列化

因为其编码结构为 **Tag - Length - Value**，所以其字段完整的序列化会稍微多出一些过程，其中有一些需要我们进一步整理。现在以一个 string 类型字段的序列化为例，来看看其序列化的完整过程，画出其程序时序图（上文出现过）如下：

ProtoBuf 序列化_反序列化时序图.png
可对照上述时序图来阅读源码，其序列化实现的几个关键函数为：

- **ByteSizeLong**：计算对象序列化所需要的空间大小，在内存中开辟相应大小的空间
- **WriteTagToArray**：将 Tag 值写入到之前开辟的内存中
- **WriteStringWithSizeToArray**：将 Length + Value 值写入到之前开辟的内存中

其序列化代码的重点过程在上图的右下角，先是调用 **WriteTagToArray** 函数将 Tag 值写入到内存，返回指向下一个字节的指针以便继续写入。调用 **WriteStringWithSizeToArray** 函数，这个函数主要又执行了两个函数，先是执行 **WriteVarint32ToArray** 函数（注意 WriteTagToArray 内部调用的也是这个函数，因为 Tag 和 Length 都采用 Varints 编码），此函数的作用是将 Length 写入。执行的第二个函数为 **WriteStringToArray**，此函数的作用是将 Value(一个 UTF-8 string 值) 写入到内存，其中底层调用了 **memcpy()** 函数。

综上，对于 Varint 类型的字段自然采用 Varint 编码。

而对于 Length delimited 类型的字段，Tag-Length-Value 中的 Tag 和 Length 依然采用 Varint 编码，Value 若为 String 等类型，则直接进行 memcpy。

另外对于 embedded message 或 packed repeated ，则套用上述规则。底层编码实现实际便是遍历字段下所有内嵌字段，然后递归调用编码函数即可。