# 一种自动反射消息类型的 Google Protobuf 网络传输方案

这篇文章要解决的问题是：在接收到 protobuf 数据之后，如何自动创建具体的 Protobuf Message 对象，再做的反序列化。“自动”的意思是：当程序中新增一个 protobuf Message 类型时，这部分代码不需要修改，不需要自己去注册消息类型。其实，Google Protobuf 本身具有很强的反射(reflection)功能，可以根据 type name 创建具体类型的 Message 对象，我们直接利用即可。

本文假定读者了解 Google Protocol Buffers 是什么，这不是一篇 protobuf 入门教程。

本文以 C++ 语言举例，其他语言估计有类似的解法，欢迎补充。

本文的示例代码在： <https://github.com/chenshuo/recipes/tree/master/protobuf>

# 网络编程中使用 protobuf 的两个问题

[Google Protocol Buffers](http://code.google.com/p/protobuf/) (Protobuf) 是一款非常优秀的库，它定义了一种紧凑的可扩展二进制消息格式，特别适合网络数据传输。它为多种语言提供 binding，大大方便了分布式程序的开发，让系统不再局限于用某一种语言来编写。

在网络编程中使用 protobuf 需要解决两个问题：

* **长度**，protobuf 打包的数据没有自带长度信息或终结符，需要由应用程序自己在发生和接收的时候做正确的切分；
* **类型**，protobuf 打包的数据没有自带**类型信息**，需要由发送方把类型信息传给给接收方，接收方创建具体的 Protobuf Message 对象，再做的反序列化。

第一个很好解决，通常的做法是在每个消息前面加个固定长度的 length header，例如我在 《[Muduo 网络编程示例之二： Boost.Asio 的聊天服务器](http://blog.csdn.net/Solstice/archive/2011/02/04/6172391.aspx)》 中实现的 LengthHeaderCodec，代码见 <http://code.google.com/p/muduo/source/browse/trunk/examples/asio/chat/codec.h>

第二个问题其实也很好解决，Protobuf 对此有内建的支持。但是奇怪的是，从网上简单搜索的情况看，我发现了很多山寨的做法。

# 山寨做法

以下均为在 protobuf data 之前加上 header，header 中包含 int length 和**类型信息**。类型信息的山寨做法主要有两种：

* 在 header 中放 int typeId，接收方用 switch-case 来选择对应的消息类型和处理函数；
* 在 header 中放 string typeName，接收方用 look-up table 来选择对应的消息类型和处理函数。

这两种做法都有问题。

第一种做法要求保持 typeId 的唯一性，它和 protobuf message type 一一对应。如果 protobuf message 的使用范围不广，比如接收方和发送方都是自己维护的程序，那么 typeId 的唯一性不难保证，用版本管理工具即可。如果 protobuf message 的使用范围很大，比如全公司都在用，而且不同部门开发的分布式程序可能相互通信，那么就需要一个公司内部的全局机构来分配 typeId，每次增加新 message type 都要去注册一下，比较麻烦。

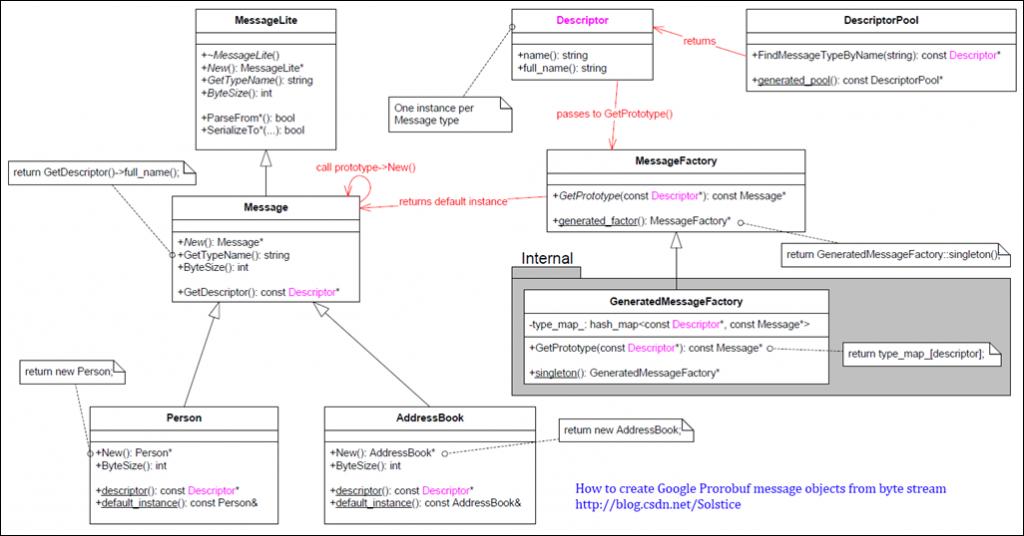
第二种做法稍好一点。typeName 的唯一性比较好办，因为可以加上 package name（也就是用 message 的 fully qualified type name），各个部门事先分好 namespace，不会冲突与重复。但是每次新增消息类型的时候都要去手工修改 look-up table 的初始化代码，比较麻烦。

其实，不需要自己重新发明轮子，protobuf 本身已经自带了解决方案。

# 根据 type name 反射自动创建 Message 对象

Google Protobuf 本身具有很强的反射(reflection)功能，可以根据 type name 创建具体类型的 Message 对象。但是奇怪的是，其官方教程里没有明确提及这个用法，我估计还有很多人不知道这个用法，所以觉得值得写这篇 blog 谈一谈。

以下是陈硕绘制的 Protobuf  class diagram，[点击查看原图](http://www.cppblog.com/images/cppblog_com/Solstice/16438/o_protobuf_classdiagram.png)。



我估计大家通常关心和使用的是图的左半部分：MessageLite、Message、Generated Message Types (Person, AddressBook) 等，而较少注意到图的右半部分：Descriptor, DescriptorPool, MessageFactory。

上图中，其关键作用的是 Descriptor class，每个具体 Message Type 对应一个 Descriptor 对象。尽管我们没有直接调用它的函数，但是Descriptor在“根据 type name 创建具体类型的 Message 对象”中扮演了重要的角色，起了桥梁作用。上图的红色箭头描述了根据 type name 创建具体 Message 对象的过程，后文会详细介绍。

### 原理简述

Protobuf Message class 采用了 [prototype pattern](http://en.wikipedia.org/wiki/Prototype_pattern)，Message class 定义了 New() 虚函数，用以返回本对象的一份新实例，类型与本对象的真实类型相同。也就是说，拿到 Message\* 指针，不用知道它的具体类型，就能创建和它类型一样的具体 Message Type 的对象。

每个具体 Message Type 都有一个 default instance，可以通过 ConcreteMessage::default\_instance() 获得，也可以通过 MessageFactory::GetPrototype(const Descriptor\*) 来获得。所以，现在问题转变为 1. 如何拿到 MessageFactory；2. 如何拿到 Descriptor\*。

当然，ConcreteMessage::descriptor() 返回了我们想要的 Descriptor\*，但是，在不知道 ConcreteMessage 的时候，如何调用它的静态成员函数呢？这似乎是个鸡与蛋的问题。

我们的英雄是 DescriptorPool，它可以根据 type name 查到 Descriptor\*，只要找到合适的 DescriptorPool，再调用 DescriptorPool::FindMessageTypeByName(const string& type\_name) 即可。眼前一亮？

在最终解决问题之前，先简单[**测试**](http://lib.csdn.net/base/softwaretest)一下，看看我上面说的对不对。

### 简单测试

本文用于举例的 proto 文件：query.proto，见 <https://github.com/chenshuo/recipes/blob/master/protobuf/query.proto>

package muduo;

message Query {

required int64 id = 1;

required string questioner = 2;

repeated string question = 3;

}

message Answer {

required int64 id = 1;

required string questioner = 2;

required string answerer = 3;

repeated string solution = 4;

}

message Empty {

optional int32 id = 1;

}

其中的 Query.questioner 和 Answer.answerer 是我在前一篇文章这提到的《[分布式系统中的进程标识](http://blog.csdn.net/Solstice/archive/2011/03/29/6285216.aspx)》。

以下代码验证 ConcreteMessage::default\_instance()、ConcreteMessage::descriptor()、 MessageFactory::GetPrototype()、DescriptorPool::FindMessageTypeByName() 之间的不变式 (invariant)：

<https://github.com/chenshuo/recipes/blob/master/protobuf/descriptor_test.cc#L15>

typedef muduo::Query T;

std::string type\_name = T::descriptor()->full\_name();

cout << type\_name << endl;

const Descriptor\* descriptor = DescriptorPool::generated\_pool()->FindMessageTypeByName(type\_name);

assert(descriptor == T::descriptor());

cout << "FindMessageTypeByName() = " << descriptor << endl;

cout << "T::descriptor() = " << T::descriptor() << endl;

cout << endl;

const Message\* prototype = MessageFactory::generated\_factory()->GetPrototype(descriptor);

assert(prototype == &T::default\_instance());

cout << "GetPrototype() = " << prototype << endl;

cout << "T::default\_instance() = " << &T::default\_instance() << endl;

cout << endl;

T\* new\_obj = dynamic\_cast(prototype->New());

assert(new\_obj != NULL);

assert(new\_obj != prototype);

assert(typeid(\*new\_obj) == typeid(T::default\_instance()));

cout << "prototype->New() = " << new\_obj << endl;

cout << endl;

delete new\_obj;

### 根据 type name 自动创建 Message 的关键代码

好了，万事具备，开始行动：

1. 用 DescriptorPool::generated\_pool() 找到一个 DescriptorPool 对象，它包含了**程序编译的时候所链接的全部 protobuf Message types**。
2. 用 DescriptorPool::FindMessageTypeByName() 根据 type name 查找 Descriptor。
3. 再用 MessageFactory::generated\_factory() 找到 MessageFactory 对象，它能创建程序编译的时候所链接的全部 protobuf Message types。
4. 然后，用 MessageFactory::GetPrototype() 找到具体 Message Type 的 default instance。
5. 最后，用 prototype->New() 创建对象。

示例代码见 <https://github.com/chenshuo/recipes/blob/master/protobuf/codec.h#L69>

Message\* createMessage(const std::string& typeName)

{

Message\* message = NULL;

const Descriptor\* descriptor = DescriptorPool::generated\_pool()->FindMessageTypeByName(typeName);

if (descriptor)

{

const Message\* prototype = MessageFactory::generated\_factory()->GetPrototype(descriptor);

if (prototype)

{

message = prototype->New();

}

}

return message;

}

调用方式：<https://github.com/chenshuo/recipes/blob/master/protobuf/descriptor_test.cc#L49>

Message\* newQuery = createMessage("muduo.Query");

assert(newQuery != NULL);

assert(typeid(\*newQuery) == typeid(muduo::Query::default\_instance()));

cout << "createMessage(/"muduo.Query/") = " << newQuery << endl;

古之人不余欺也 :-)

注意，createMessage() 返回的是动态创建的对象的指针，调用方有责任释放它，不然就会内存泄露。在 muduo 里，我用 shared\_ptr 来自动管理 Message 对象的生命期。

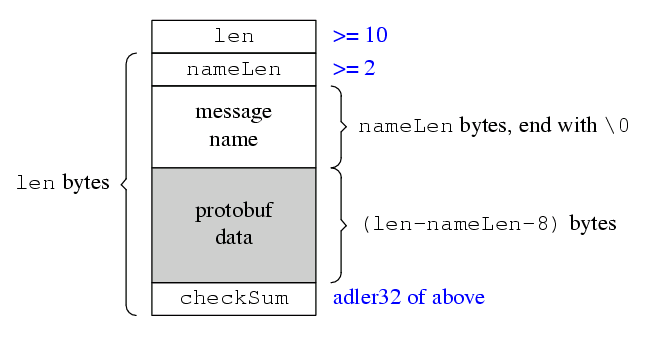
### 线程安全性

Google 的文档说，我们用到的那几个 MessageFactory 和 DescriptorPool 都是线程安全的，Message::New() 也是线程安全的。并且它们都是 const member function。

关键问题解决了，那么剩下工作就是**设计一种包含长度和消息类型的 protobuf 传输格式**。

# Protobuf 传输格式

陈硕设计了一个简单的格式，包含 protobuf data 和它对应的长度与类型信息，消息的末尾还有一个 check sum。格式如下图，图中方块的宽度是 32-bit。



用 C struct 伪代码描述：

struct ProtobufTransportFormat \_\_attribute\_\_ ((\_\_packed\_\_))

{

int32\_t len;

int32\_t nameLen;

char typeName[nameLen];

char protobufData[len-nameLen-8];

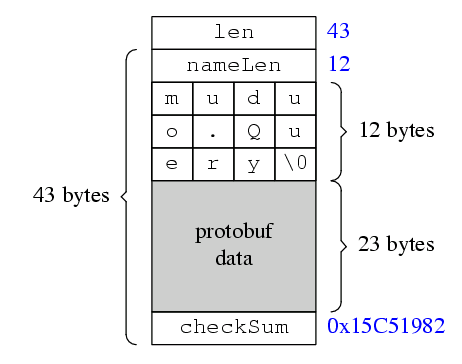
int32\_t checkSum; // adler32 of nameLen, typeName and protobufData

};

注意，这个格式不要求 32-bit 对齐，我们的 decoder 会自动处理非对齐的消息。

### 例子

用这个格式打包一个 muduo.Query 对象的结果是：



### 设计决策

以下是我在设计这个传输格式时的考虑：

* **signed int**。消息中的长度字段只使用了 signed 32-bit int，而没有使用 unsigned int，这是为了移植性，因为 Java 语言没有 unsigned 类型。另外 Protobuf 一般用于打包小于 1M 的数据，unsigned int 也没用。
* **check sum**。虽然 TCP 是可靠传输协议，虽然 Ethernet 有 CRC-32 校验，但是网络传输必须要考虑数据损坏的情况，对于关键的网络应用，check sum 是必不可少的。对于 protobuf 这种紧凑的二进制格式而言，肉眼看不出数据有没有问题，需要用 check sum。
* **adler32 算法**。我没有选用常见的 CRC-32，而是选用 adler32，因为它计算量小、速度比较快，强度和 CRC-32差不多。另外，zlib 和 java.unit.zip 都直接支持这个算法，不用我们自己实现。
* **type name 以 '/0' 结束**。这是为了方便 troubleshooting，比如通过 tcpdump 抓下来的包可以用肉眼很容易看出 type name，而不用根据 nameLen 去一个个数字节。同时，为了方便接收方处理，加入了 nameLen，节省 strlen()，空间换时间。
* **没有版本号**。Protobuf Message 的一个突出优点是用 optional fields 来避免协议的版本号（凡是在 protobuf Message 里放版本号的人都没有理解 protobuf 的设计），让通信双方的程序能各自升级，便于系统演化。如果我设计的这个传输格式又把版本号加进去，那就画蛇添足了。具体请见本人《[分布式系统的工程化开发方法](http://blog.csdn.net/Solstice/archive/2010/10/19/5950190.aspx)》第 57 页：消息格式的选择。

# 示例代码

为了简单起见，采用 std::string 来作为打包的产物，仅为示例。

打包 encode 的代码：<https://github.com/chenshuo/recipes/blob/master/protobuf/codec.h#L35>

解包 decode 的代码：<https://github.com/chenshuo/recipes/blob/master/protobuf/codec.h#L99>

测试代码： <https://github.com/chenshuo/recipes/blob/master/protobuf/codec_test.cc>

如果以上代码编译通过，但是在运行时出现“cannot open shared object file”错误，一般可以用 sudo ldconfig 解决，前提是 libprotobuf.so 位于 /usr/local/lib，且 /etc/ld.so.conf 列出了这个目录。

$ make all # 如果你安装了 boost，可以 make whole

$ ./codec\_test   
./codec\_test: error while loading shared libraries: libprotobuf.so.6: cannot open shared object file: No such file or directory

$ sudo ldconfig

**参考资料**：http://blog.csdn.net/solstice/article/details/6300108

# 自动识别Protocolbuf消息类型

Self-describing Messages

Protocol Buffers do not contain descriptions of their own types. Thus, given only a raw message without the corresponding .proto file defining its type, it is difficult to extract any useful data.

However, note that the contents of a .proto file can itself be represented using protocol buffers. The file src/google/protobuf/descriptor.proto in the source code package defines the message types involved. protoccan output a FileDescriptorSet – which represents a set of .proto files – using the --descriptor\_set\_out option. With this, you could define a self-describing protocol message like so:

message SelfDescribingMessage {

// Set of .proto files which define the type.

required FileDescriptorSet proto\_files = 1;

// Name of the message type. Must be defined by one of the files in

// proto\_files.

required string type\_name = 2;

// The message data.

required bytes message\_data = 3;

}

By using classes like DynamicMessage (available in C++ and Java), you can then write tools which can manipulate SelfDescribingMessages.

All that said, the reason that this functionality is not included in the Protocol Buffer library is because we have never had a use for it inside Google.

This technique requires support for proto2 syntax (as this is the syntax descriptor.proto uses) as well as support for dynamic messages using descriptors. Please check that the platforms you need support these features before using self-describing messages.

**参考资料**：https://developers.google.com/protocol-buffers/docs/techniques