1. [l  定义一个消息类型](http://blog.csdn.net/yzzky/article/details/44650825" \l "t0)
   1. [指定字段类型](http://blog.csdn.net/yzzky/article/details/44650825#t1)
   2. [分配标识号](http://blog.csdn.net/yzzky/article/details/44650825#t2)
   3. [指定字段规则](http://blog.csdn.net/yzzky/article/details/44650825#t3)
   4. [添加更多消息类型](http://blog.csdn.net/yzzky/article/details/44650825#t4)
   5. [添加注释](http://blog.csdn.net/yzzky/article/details/44650825#t5)
   6. [从proto文件生成了什么](http://blog.csdn.net/yzzky/article/details/44650825#t6)
2. [l  标量数值类型](http://blog.csdn.net/yzzky/article/details/44650825#t7)
3. [l  Optional的字段和默认值](http://blog.csdn.net/yzzky/article/details/44650825#t8)
4. [l 枚举](http://blog.csdn.net/yzzky/article/details/44650825#t9)
5. [l  使用其他消息类型](http://blog.csdn.net/yzzky/article/details/44650825#t10)
   1. [导入定义](http://blog.csdn.net/yzzky/article/details/44650825#t11)
6. [l  嵌套类型](http://blog.csdn.net/yzzky/article/details/44650825#t12)
   1. [组](http://blog.csdn.net/yzzky/article/details/44650825#t13)
7. [l  更新一个消息类型](http://blog.csdn.net/yzzky/article/details/44650825" \l "t14)
8. [l  扩展](http://blog.csdn.net/yzzky/article/details/44650825#t15)
9. [l  嵌套的扩展](http://blog.csdn.net/yzzky/article/details/44650825#t16)
   1. [选择可扩展的标符号](http://blog.csdn.net/yzzky/article/details/44650825#t17)
10. [l  包Package](http://blog.csdn.net/yzzky/article/details/44650825#t18)
    1. [包及名称的解析](http://blog.csdn.net/yzzky/article/details/44650825#t19)
11. [l  定义服务Service](http://blog.csdn.net/yzzky/article/details/44650825#t20)
12. [l  选项Options](http://blog.csdn.net/yzzky/article/details/44650825#t21)
13. [l  生成访问类](http://blog.csdn.net/yzzky/article/details/44650825#t22)
14. [一种自动反射消息类型的Google Protobuf 网络传输方案](http://blog.csdn.net/yzzky/article/details/44650825#t23)
    1. [网络编程中使用 protobuf 的两个问题](http://blog.csdn.net/yzzky/article/details/44650825#t24)
    2. [山寨做法](http://blog.csdn.net/yzzky/article/details/44650825#t25)
    3. [根据 type name 反射自动创建 Message 对象](http://blog.csdn.net/yzzky/article/details/44650825#t26)
       1. [原理简述](http://blog.csdn.net/yzzky/article/details/44650825#t27)
       2. [简单测试](http://blog.csdn.net/yzzky/article/details/44650825#t28)
       3. [根据 type name 自动创建 Message 的关键代码](http://blog.csdn.net/yzzky/article/details/44650825#t29)
       4. [线程安全性](http://blog.csdn.net/yzzky/article/details/44650825#t30)
    4. [Protobuf 传输格式](http://blog.csdn.net/yzzky/article/details/44650825#t31)
       1. [例子](http://blog.csdn.net/yzzky/article/details/44650825#t32)
       2. [设计决策](http://blog.csdn.net/yzzky/article/details/44650825#t33)
    5. [示例代码](http://blog.csdn.net/yzzky/article/details/44650825#t34)
    6. [与 muduo 集成](http://blog.csdn.net/yzzky/article/details/44650825#t35)
       1. [消息的分发 dispatching](http://blog.csdn.net/yzzky/article/details/44650825#t36)
       2. [基于 muduo 的 Protobuf RPC](http://blog.csdn.net/yzzky/article/details/44650825#t37)
    7. [ProtoBuf开发者指南 非官方不完整版](http://blog.csdn.net/yzzky/article/details/44650825#t38)
15. [ProtoBuf开发者指南](http://blog.csdn.net/yzzky/article/details/44650825#t39)
16. [概览](http://blog.csdn.net/yzzky/article/details/44650825#t40)
    1. [1   什么是protocol buffer](http://blog.csdn.net/yzzky/article/details/44650825#t41)
    2. [2   他们如何工作](http://blog.csdn.net/yzzky/article/details/44650825#t42)
    3. [3   为什么不用XML](http://blog.csdn.net/yzzky/article/details/44650825#t43)
    4. [4   听起来像是为我的解决方案如何开始](http://blog.csdn.net/yzzky/article/details/44650825#t44)
    5. [5   一点历史](http://blog.csdn.net/yzzky/article/details/44650825#t45)
17. [语言指导](http://blog.csdn.net/yzzky/article/details/44650825#t46)
    1. [1   定义一个消息类型](http://blog.csdn.net/yzzky/article/details/44650825#t47)
    2. [2   值类型](http://blog.csdn.net/yzzky/article/details/44650825#t48)
    3. [3   可选字段与缺省值](http://blog.csdn.net/yzzky/article/details/44650825#t49)
    4. [4   枚举](http://blog.csdn.net/yzzky/article/details/44650825#t50)
    5. [5   使用其他消息类型](http://blog.csdn.net/yzzky/article/details/44650825#t51)
    6. [6   嵌套类型](http://blog.csdn.net/yzzky/article/details/44650825#t52)
    7. [7   更新一个数据类型](http://blog.csdn.net/yzzky/article/details/44650825#t53)
    8. [8   扩展](http://blog.csdn.net/yzzky/article/details/44650825#t54)
    9. [9   包](http://blog.csdn.net/yzzky/article/details/44650825#t55)
    10. [10   定义服务](http://blog.csdn.net/yzzky/article/details/44650825#t56)
    11. [11   选项](http://blog.csdn.net/yzzky/article/details/44650825#t57)
    12. [12   生成你的类](http://blog.csdn.net/yzzky/article/details/44650825#t58)
18. [代码风格指导](http://blog.csdn.net/yzzky/article/details/44650825#t59)
    1. [1   消息与字段名](http://blog.csdn.net/yzzky/article/details/44650825#t60)
    2. [2   枚举](http://blog.csdn.net/yzzky/article/details/44650825#t61)
    3. [3   服务](http://blog.csdn.net/yzzky/article/details/44650825#t62)
19. [编码](http://blog.csdn.net/yzzky/article/details/44650825#t63)
    1. [1   一个简单的消息](http://blog.csdn.net/yzzky/article/details/44650825#t64)
    2. [2   基于128的Varints](http://blog.csdn.net/yzzky/article/details/44650825#t65)
    3. [3   消息结构](http://blog.csdn.net/yzzky/article/details/44650825#t66)
    4. [4   更多的值类型](http://blog.csdn.net/yzzky/article/details/44650825#t67)
    5. [5   内嵌消息](http://blog.csdn.net/yzzky/article/details/44650825#t68)
    6. [6   可选的和重复的元素](http://blog.csdn.net/yzzky/article/details/44650825#t69)
    7. [7   字段顺序](http://blog.csdn.net/yzzky/article/details/44650825#t70)
20. [ProtocolBuffer基础C](http://blog.csdn.net/yzzky/article/details/44650825#t71)
21. [ProtocolBuffer基础Java](http://blog.csdn.net/yzzky/article/details/44650825#t72)
22. [ProtocolBuffer基础Python](http://blog.csdn.net/yzzky/article/details/44650825#t73)
    1. [1   为什么使用ProtocolBuffer](http://blog.csdn.net/yzzky/article/details/44650825#t74)
    2. [2   哪里可以找到例子代码](http://blog.csdn.net/yzzky/article/details/44650825#t75)
    3. [3   定义你的协议格式](http://blog.csdn.net/yzzky/article/details/44650825#t76)
    4. [4   编译你的ProtocolBuffer](http://blog.csdn.net/yzzky/article/details/44650825#t77)
    5. [5   ProtocolBuffer API](http://blog.csdn.net/yzzky/article/details/44650825#t78)
       1. [51   枚举](http://blog.csdn.net/yzzky/article/details/44650825#t79)
       2. [52   标准消息方法](http://blog.csdn.net/yzzky/article/details/44650825#t80)
       3. [53   解析与串行化](http://blog.csdn.net/yzzky/article/details/44650825#t81)
    6. [6   写消息](http://blog.csdn.net/yzzky/article/details/44650825#t82)
    7. [7   读消息](http://blog.csdn.net/yzzky/article/details/44650825#t83)
    8. [8   扩展ProtocolBuffer](http://blog.csdn.net/yzzky/article/details/44650825#t84)
    9. [9   高级使用](http://blog.csdn.net/yzzky/article/details/44650825#t85)
23. [参考概览](http://blog.csdn.net/yzzky/article/details/44650825#t86)
24. [C代码生成](http://blog.csdn.net/yzzky/article/details/44650825#t87)
25. [CAPI](http://blog.csdn.net/yzzky/article/details/44650825#t88)
26. [Java代码生成](http://blog.csdn.net/yzzky/article/details/44650825#t89)
27. [JavaAPI](http://blog.csdn.net/yzzky/article/details/44650825#t90)
28. [Python代码生成](http://blog.csdn.net/yzzky/article/details/44650825#t91)
    1. [1   编译器的使用](http://blog.csdn.net/yzzky/article/details/44650825#t92)
    2. [2   包](http://blog.csdn.net/yzzky/article/details/44650825#t93)
    3. [3   消息](http://blog.csdn.net/yzzky/article/details/44650825#t94)
    4. [4   字段](http://blog.csdn.net/yzzky/article/details/44650825#t95)
       1. [41   简单字段](http://blog.csdn.net/yzzky/article/details/44650825#t96)
       2. [42   简单消息字段](http://blog.csdn.net/yzzky/article/details/44650825#t97)
       3. [43   重复字段](http://blog.csdn.net/yzzky/article/details/44650825#t98)
       4. [44   重复消息字段](http://blog.csdn.net/yzzky/article/details/44650825#t99)
       5. [45   枚举类型](http://blog.csdn.net/yzzky/article/details/44650825#t100)
       6. [46   扩展](http://blog.csdn.net/yzzky/article/details/44650825#t101)
    5. [5   服务](http://blog.csdn.net/yzzky/article/details/44650825#t102)
       1. [51   接口](http://blog.csdn.net/yzzky/article/details/44650825#t103)
       2. [52   存根Stub](http://blog.csdn.net/yzzky/article/details/44650825#t104)
29. [PythonAPI](http://blog.csdn.net/yzzky/article/details/44650825#t105)
30. [其他语言](http://blog.csdn.net/yzzky/article/details/44650825#t106)
31. [Protobuf动态解析那些事儿 - jacksutencent](http://blog.csdn.net/yzzky/article/details/44650825#t107)
    1. [需求背景](http://blog.csdn.net/yzzky/article/details/44650825#t108)
    2. [技术介绍](http://blog.csdn.net/yzzky/article/details/44650825#t109)
    3. [实现](http://blog.csdn.net/yzzky/article/details/44650825#t110)
       1. [把本地地址映射为虚拟地址](http://blog.csdn.net/yzzky/article/details/44650825#t111)
       2. [构造 DescriptorPool](http://blog.csdn.net/yzzky/article/details/44650825#t112)
       3. [获取 Descriptor](http://blog.csdn.net/yzzky/article/details/44650825#t113)
       4. [通过 Descriptor获取Message](http://blog.csdn.net/yzzky/article/details/44650825#t114)
       5. [根据类型信息使用 DynamicMessage new 出这个类型的一个空对象](http://blog.csdn.net/yzzky/article/details/44650825#t115)
       6. [通过 Message 的 reflection 操作 message 的各个字段](http://blog.csdn.net/yzzky/article/details/44650825#t116)
    4. [总结](http://blog.csdn.net/yzzky/article/details/44650825#t117)
    5. [参考文献](http://blog.csdn.net/yzzky/article/details/44650825#t118)
32. [protobuf在网络编程中的应用思考](http://blog.csdn.net/yzzky/article/details/44650825#t119)
    1. [protobuf简介](http://blog.csdn.net/yzzky/article/details/44650825#t120)
    2. [例子介绍](http://blog.csdn.net/yzzky/article/details/44650825#t121)
    3. [在TCP网络编程中的考虑](http://blog.csdn.net/yzzky/article/details/44650825#t122)

Protobuf在[**Java**](http://lib.csdn.net/base/javase)中的简单使用

1.      在Java中使用protobuf需要jar包，下载protobuf-Java-2.5.0.jar包文件，添加到项目中。另外需要protoc.exe来编译proto文件。

2.      新建一个msg.proto文件：

package com.test.learn;

option java\_package = "com.test.learn";

option java\_outer\_classname = "ProtoBufTest";

message msgInfo {

required int32 ID = 1;

required int64 GoodID = 2;

required string Url = 3;

required string Guid = 4;

required string Type = 5;

required int32 Order = 6;

}

3.      将msg.proto文件和protoc.exe拷贝到同一目录下，然后使用下面的命令将msg.proto文件编译成Java类文件，可在当前目录下看到生成的ProtoBufTest.java文件：

protoc --java\_out=./ msg.proto

4.      在项目中导入上一步生成的ProtoBufTest.java文件，并将protobuf-Java-2.5.0.jar库添加到项目中，编写[**测试**](http://lib.csdn.net/base/softwaretest)文件进行测试：



5.      测试代码：

package com.test;

import com.google.protobuf.InvalidProtocolBufferException;

import com.test.learn.ProtoBufTest;

public class MainFile {

public static voidmain(String[] args) {

           ProtoBufTest.msgInfo.Builderbuilder = ProtoBufTest.msgInfo.newBuilder();

           builder.setGoodID(100);

           builder.setGuid("11111-22222-33333-44444");

           builder.setOrder(0);

           builder.setType("item");

           builder.setID(10);

           builder.setUrl("http://www.baidu.com");

           ProtoBufTest.msgInfomsgInfo = builder.build();

           byte[] result =msgInfo.toByteArray();

           try {

                    ProtoBufTest.msgInfomsg = ProtoBufTest.msgInfo.parseFrom(result);

                    System.out.println(msg);

           } catch(InvalidProtocolBufferExceptione) {

                    System.out.println(e.getMessage());

           }

}

}

[**Protobuf语言指南**](http://www.cnblogs.com/dkblog/archive/2012/03/27/2419010.html)

**Protobuf语言指南**

l  **定义一个消息（message）类型**

l  **标量值类型**

l  **Optional 的字段及默认值**

l  **枚举**

l  **使用其他消息类型**

l  **嵌套类型**

l  **更新一个消息类型**

l  **扩展**

l  **包（package）**

l  **定义服务（service）**

l  **选项（option）**

l  **生成访问类**

本指南描述了怎样使用protocolbuffer语言来构造你的protocol buffer数据，包括.proto文件语法以及怎样生成.proto文件的数据访问类。

本文是一个参考指南——如果要查看如何使用本文中描述的多个特性的循序渐进的例子，请在[http://code.google.com/intl/zh-CN/apis/protocolbuffers/docs/tutorials.html](http://www.open-open.com/home/link.php?url=http://code.google.com%2Fintl%2Fzh-CN%2Fapis%2Fprotocolbuffers%2Fdocs%2Ftutorials.html)中查找需要的语言的教程。

## l  定义一个消息类型

先来看一个非常简单的例子。假设你想定义一个“搜索请求”的消息格式，每一个请求含有一个查询字符串、你感兴趣的查询结果所在的页数，以及每一页多少条查询结果。可以采用如下的方式来定义消息类型的.proto文件了：

|  |
| --- |
| message SearchRequest {    required string query = 1;    optional int32 page\_number = 2;    optional int32 result\_per\_page = 3;  } |

SearchRequest消息格式有3个字段，在消息中承载的数据分别对应于每一个字段。其中每个字段都有一个名字和一种类型。

### Ø  指定字段类型

在上面的例子中，所有字段都是标量类型：两个整型（page\_number和result\_per\_page），一个string类型（query）。当然，你也可以为字段指定其他的合成类型，包括枚举（[enumerations](http://www.open-open.com/home/link.php?url=http://code.google.com%2Fintl%2Fzh-CN%2Fapis%2Fprotocolbuffers%2Fdocs%2Fproto.html%23enum)）或其他消息类型。

### Ø   分配标识号

正如上述文件格式，在消息定义中，每个字段都有唯一的一个标识符。这些标识符是用来在消息的二进制格式中识别各个字段的，一旦开始使用就不能够再改变。注：[1,15]之内的标识号在编码的时候会占用一个字节。[16,2047]之内的标识号则占用2个字节。所以应该为那些频繁出现的消息元素保留 [1,15]之内的标识号。切记：要为将来有可能添加的、频繁出现的标识号预留一些标识号。最小的标识号可以从1开始，最大到229 -1, or 536,870,911。不可以使用其中的[19000－19999]的标识号， Protobuf协议实现中对这些进行了预留。如果非要在.proto文件中使用这些预留标识号，编译时就会报警。

### Ø  指定字段规则

所指定的消息字段修饰符必须是如下之一：

²  required：一个格式良好的消息一定要含有1个这种字段。表示该值是必须要设置的；

²  optional：消息格式中该字段可以有0个或1个值（不超过1个）。

²  repeated：在一个格式良好的消息中，这种字段可以重复任意多次（包括0次）。重复的值的顺序会被保留。表示该值可以重复，相当于java中的List。

由于一些历史原因，基本数值类型的repeated的字段并没有被尽可能地高效编码。在新的代码中，用户应该使用特殊选项[packed=true]来保证更高效的编码。如：

|  |
| --- |
| repeated int32 samples = 4 [packed=true]; |

required是永久性的：在将一个字段标识为required的时候，应该特别小心。如果在某些情况下不想写入或者发送一个required的字段，将原始该字段修饰符更改为optional可能会遇到问题——旧版本的使用者会认为不含该字段的消息是不完整的，从而可能会无目的的拒绝解析。在这种情况下，你应该考虑编写特别针对于应用程序的、自定义的消息校验函数。Google的一些工程师得出了一个结论：使用required弊多于利；他们更愿意使用optional和repeated而不是required。当然，这个观点并不具有普遍性。

### Ø   添加更多消息类型

在一个.proto文件中可以定义多个消息类型。在定义多个相关的消息的时候，这一点特别有用——例如，如果想定义与SearchResponse消息类型对应的回复消息格式的话，你可以将它添加到相同的.proto文件中，如：

|  |
| --- |
| message SearchRequest {    required string query = 1;    optional int32 page\_number = 2;    optional int32 result\_per\_page = 3;  }    message SearchResponse {   …  } |

### Ø  添加注释

向.proto文件添加注释，可以使用C/C++/java风格的双斜杠（//） 语法格式，如：

|  |
| --- |
| message SearchRequest {    required string query = 1;    optional int32 page\_number = 2;// 最终返回的页数    optional int32 result\_per\_page = 3;// 每页返回的结果数  } |

### Ø  从.proto文件生成了什么？

当用protocolbuffer编译器来运行.proto文件时，编译器将生成所选择语言的代码，这些代码可以操作在.proto文件中定义的消息类型，包括获取、设置字段值，将消息序列化到一个输出流中，以及从一个输入流中解析消息。

²  对C++来说，编译器会为每个.proto文件生成一个.h文件和一个.cc文件，.proto文件中的每一个消息有一个对应的类。

²  对Java来说，编译器为每一个消息类型生成了一个.java文件，以及一个特殊的Builder类（该类是用来创建消息类接口的）。

²  对[**Python**](http://lib.csdn.net/base/python)来说，有点不太一样——Python编译器为.proto文件中的每个消息类型生成一个含有静态描述符的模块，该模块与一个元类（metaclass）在运行时（runtime）被用来创建所需的Python数据访问类。

你可以从如下的文档链接中获取每种语言更多API。[http://code.google.com/intl/zh-CN/apis/protocolbuffers/docs/reference/overview.html](http://www.open-open.com/home/link.php?url=http://code.google.com%2Fintl%2Fzh-CN%2Fapis%2Fprotocolbuffers%2Fdocs%2Freference%2Foverview.html" \t "_blank)

## l  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类型 | Java 类型 | C++类型 | 备注 |
| double | double | double |  |
| float | float | float |  |
| int32 | int | int32 | 使用可变长编码方式。编码负数时不够高效——如果你的字段可能含有负数，那么请使用sint32。 |
| int64 | long | int64 | 使用可变长编码方式。编码负数时不够高效——如果你的字段可能含有负数，那么请使用sint64。 |
| uint32 | int | uint32 | Uses variable-length encoding. |
| uint64 | long | uint64 | Uses variable-length encoding. |
| sint32 | int | int32 | 使用可变长编码方式。有符号的整型值。编码时比通常的int32高效。 |
| sint64 | long | int64 | 使用可变长编码方式。有符号的整型值。编码时比通常的int64高效。 |
| fixed32 | int | uint32 | 总是4个字节。如果数值总是比总是比228大的话，这个类型会比uint32高效。 |
| fixed64 | long | uint64 | 总是8个字节。如果数值总是比总是比256大的话，这个类型会比uint64高效。 |
| sfixed32 | int | int32 | 总是4个字节。 |
| sfixed64 | long | int64 | 总是8个字节。 |
| bool | boolean | bool |  |
| string | String | string | 一个字符串必须是UTF-8编码或者7-bit ASCII编码的文本。 |
| bytes | ByteString | string | 可能包含任意顺序的字节数据。 |
| map<int32, int32> | Map |  | 键值对数据类型 |

你可以在文章[http://code.google.com/apis/protocolbuffers/docs/encoding.html](http://www.open-open.com/home/link.php?url=http://code.google.com%2Fapis%2Fprotocolbuffers%2Fdocs%2Fencoding.html) 中，找到更多“序列化消息时各种类型如何编码”的信息。

## l  Optional的字段和默认值

如上所述，消息描述中的一个元素可以被标记为“可选的”（optional）。一个格式良好的消息可以包含0个或一个optional的元素。当解析消息时，如果它不包含optional的元素值，那么解析出来的对象中的对应字段就被置为默认值。默认值可以在消息描述文件中指定。例如，要为SearchRequest消息的result\_per\_page字段指定默认值10（在使用hasField()方法判断时默认值返回还是false），在定义消息格式时如下所示：

|  |
| --- |
| optional int32 result\_per\_page = 3 [default = 10]; |

如果没有为optional的元素指定默认值，就会使用与特定类型相关的默认值：对string来说，默认值是空字符串。对bool来说，默认值是false。对数值类型来说，默认值是0。对枚举来说，默认值是枚举类型定义中的第一个值。

## l 枚举

当需要定义一个消息类型的时候，可能想为一个字段指定某“预定义值序列”中的一个值。例如，假设要为每一个SearchRequest消息添加一个corpus字段，而corpus的值可能是UNIVERSAL，WEB，IMAGES，LOCAL，NEWS，PRODUCTS或VIDEO中的一个。其实可以很容易地实现这一点：通过向消息定义中添加一个枚举（enum）就可以了。一个enum类型的字段只能用指定的常量集中的一个值作为其值（如果尝试指定不同的值，解析器就会把它当作一个未知的字段来对待）。在下面的例子中，在消息格式中添加了一个叫做Corpus的枚举类型——它含有所有可能的值 ——以及一个类型为Corpus的字段：

|  |
| --- |
| message SearchRequest {    required string query = 1;    optional int32 page\_number = 2;    optional int32 result\_per\_page = 3 [default = 10];    enum Corpus {      UNIVERSAL = 0;      WEB = 1;      IMAGES = 2;      LOCAL = 3;      NEWS = 4;      PRODUCTS = 5;      VIDEO = 6;    }    optional Corpus corpus = 4 [default = UNIVERSAL];  } |

枚举常量必须在32位整型值的范围内。因为enum值是使用可变编码方式的，对负数不够高效，因此不推荐在enum中使用负数。如上例所示，可以在一个消息定义的内部或外部定义枚举——这些枚举可以在.proto文件中的任何消息定义里重用。当然也可以在一个消息中声明一个枚举类型，而在另一个不同的消息中使用它——采用MessageType.EnumType的语法格式。当对一个使用了枚举的.proto文件运行protocol buffer编译器的时候，生成的代码中将有一个对应的enum（对Java或C++来说），或者一个特殊的EnumDescriptor类（对 Python来说），它被用来在运行时生成的类中创建一系列的整型值符号常量（symbolic constants）。

关于如何在你的应用程序的消息中使用枚举的更多信息，请查看所选择的语言http://code.google.com/intl/zh-CN/apis/protocolbuffers/docs/reference/overview.html。

## l  使用其他消息类型

你可以将其他消息类型用作字段类型。例如，假设在每一个SearchResponse消息中包含Result消息，此时可以在相同的.proto文件中定义一个Result消息类型，然后在SearchResponse消息中指定一个Result类型的字段，如：

|  |
| --- |
| message SearchResponse {    repeated Result result = 1;  }  message Result {    required string url = 1;    optional string title = 2;    repeated string snippets = 3;  } |

### Ø  导入定义

在上面的例子中，Result消息类型与SearchResponse是定义在同一文件中的。如果想要使用的消息类型已经在其他.proto文件中已经定义过了呢？

 你可以通过导入（importing）其他.proto文件中的定义来使用它们。要导入其他.proto文件的定义，你需要在你的文件中添加一个导入声明，到如的文件必须存在package，否则生成的时候需要手动删除导入文件中错误的前缀'.'，如：

|  |
| --- |
| import "myproject/other\_protos.proto"; |

protocol编译器就会在一系列目录中查找需要被导入的文件，这些目录通过protocol编译器的命令行参数-I/–import\_path指定。如果不提供参数，编译器就在其调用目录下查找。

## l  嵌套类型

你可以在其他消息类型中定义、使用消息类型，在下面的例子中，Result消息就定义在SearchResponse消息内，如：

|  |
| --- |
| message SearchResponse {    message Result {      required string url = 1;      optional string title = 2;      repeated string snippets = 3;    }    repeated Result result = 1;  } |

如果你想在它的父消息类型的外部重用这个消息类型，你需要以Parent.Type的形式使用它，如：

|  |
| --- |
| message SomeOtherMessage {    optional SearchResponse.Result result = 1;  } |

当然，你也可以将消息嵌套任意多层，如：

|  |
| --- |
| message Outer {                  // Level 0    message MiddleAA {  // Level 1      message Inner {   // Level 2        required int64 ival = 1;        optional bool  booly = 2;      }    }    message MiddleBB {  // Level 1      message Inner {   // Level 2        required int32 ival = 1;        optional bool  booly = 2;      }    }  } |

### Ø  组

**注：该特性已被弃用，在创建新的消息类型的时候，不应该再使用它——可以使用嵌套消息类型来代替它。**

“组”是指在消息定义中嵌套信息的另一种方法。比如，在SearchResponse中包含若干Result的另一种方法是 ：

|  |
| --- |
| message SearchResponse {    repeated group Result = 1 {      required string url = 2;      optional string title = 3;      repeated string snippets = 4;    }  } |

一个“组”只是简单地将一个嵌套消息类型和一个字段捆绑到一个单独的声明中。在代码中，可以把它看成是含有一个Result类型、名叫result的字段的消息（后面的名字被转换成了小写，所以它不会与前面的冲突）。

因此，除了数据传输格式不同之外，这个例子与上面的SearchResponse例子是完全等价的。

## l  更新一个消息类型

如果一个已有的消息格式已无法满足新的需求——如，要在消息中添加一个额外的字段——但是同时旧版本写的代码仍然可用。不用担心！更新消息而不破坏已有代码是非常简单的。在更新时只要记住以下的规则即可。

²  不要更改任何已有的字段的数值标识。

²  所添加的任何字段都必须是optional或repeated的。这就意味着任何使用“旧”的消息格式的代码序列化的消息可以被新的代码所解析，因为它们不会丢掉任何required的元素。应该为这些元素设置合理的默认值，这样新的代码就能够正确地与老代码生成的消息交互了。类似地，新的代码创建的消息也能被老的代码解析：老的二进制程序在解析的时候只是简单地将新字段忽略。然而，未知的字段是没有被抛弃的。此后，如果消息被序列化，未知的字段会随之一起被序列化——所以，如果消息传到了新代码那里，则新的字段仍然可用。注意：对Python来说，对未知字段的保留策略是无效的。

²  非required的字段可以移除——只要它们的标识号在新的消息类型中不再使用（更好的做法可能是重命名那个字段，例如在字段前添加“OBSOLETE\_”前缀，那样的话，使用的.proto文件的用户将来就不会无意中重新使用了那些不该使用的标识号）。

²  一个非required的字段可以转换为一个扩展，反之亦然——只要它的类型和标识号保持不变。

²  int32, uint32, int64, uint64,和bool是全部兼容的，这意味着可以将这些类型中的一个转换为另外一个，而不会破坏向前、向后的兼容性。如果解析出来的数字与对应的类型不相符，那么结果就像在C++中对它进行了强制类型转换一样（例如，如果把一个64位数字当作int32来读取，那么它就会被截断为32位的数字）。

²  sint32和sint64是互相兼容的，但是它们与其他整数类型不兼容。

²  string和bytes是兼容的——只要bytes是有效的UTF-8编码。

²  嵌套消息与bytes是兼容的——只要bytes包含该消息的一个编码过的版本。

²  fixed32与sfixed32是兼容的，fixed64与sfixed64是兼容的。

## l  扩展

通过扩展，可以将一个范围内的字段标识号声明为可被第三方扩展所用。然后，其他人就可以在他们自己的.proto文件中为该消息类型声明新的字段，而不必去编辑原始文件了。看个具体例子：

|  |
| --- |
| message Foo {    // …    extensions 100 to 199;  } |

这个例子表明：在消息Foo中，范围[100,199]之内的字段标识号被保留为扩展用。现在，其他人就可以在他们自己的.proto文件中添加新字段到Foo里了，但是添加的字段标识号要在指定的范围内——例如：

|  |
| --- |
| extend Foo {    optional int32 bar = 126;  } |

这个例子表明：消息Foo现在有一个名为bar的optional int32字段。

当用户的Foo消息被编码的时候，数据的传输格式与用户在Foo里定义新字段的效果是完全一样的。

然而，要在程序代码中访问扩展字段的方法与访问普通的字段稍有不同——生成的数据访问代码为扩展准备了特殊的访问函数来访问它。例如，下面是如何在C++中设置bar的值：

|  |
| --- |
| Foo foo; foo.SetExtension(bar, 15); |

类似地，Foo类也定义了模板函数 HasExtension()，ClearExtension()，GetExtension()，MutableExtension()，以及 AddExtension()。这些函数的语义都与对应的普通字段的访问函数相符。要查看更多使用扩展的信息，请参考相应语言的代码生成指南。注：扩展可以是任何字段类型，包括消息类型。

## l  嵌套的扩展

可以在另一个类型的范围内声明扩展，如：

|  |
| --- |
| message Baz {    extend Foo {      optional int32 bar = 126;    }    …  } |

在此例中，访问此扩展的C++代码如下：

|  |
| --- |
| Foo foo;  foo.SetExtension(Baz::bar, 15); |

一个通常的设计模式就是：在扩展的字段类型的范围内定义该扩展——例如，下面是一个Foo的扩展（该扩展是Baz类型的），其中，扩展被定义为了Baz的一部分：

|  |
| --- |
| message Baz {    extend Foo {      optional Baz foo\_ext = 127;    }    …  } |

然而，并没有强制要求一个消息类型的扩展一定要定义在那个消息中。也可以这样做：

|  |
| --- |
| message Baz {    …  }    extend Foo {    optional Baz foo\_baz\_ext = 127;  } |

事实上，这种语法格式更能防止引起混淆。正如上面所提到的，嵌套的语法通常被错误地认为有子类化的关系——尤其是对那些还不熟悉扩展的用户来说。

### Ø  选择可扩展的标符号

在同一个消息类型中一定要确保两个用户不会扩展新增相同的标识号，否则可能会导致数据的不一致。可以通过为新项目定义一个可扩展标识号规则来防止该情况的发生。

如果标识号需要很大的数量时，可以将该可扩展标符号的范围扩大至max，其中max是229 - 1, 或536,870,911。如下所示：

|  |
| --- |
| message Foo {    extensions 1000 to max;  } |

通常情况下在选择标符号时，标识号产生的规则中应该避开[19000－19999]之间的数字，因为这些已经被Protocol Buffers实现中预留了。

## l  包（Package）

当然可以为.proto文件新增一个可选的package声明符，用来防止不同的消息类型有命名冲突。如：

|  |
| --- |
| package foo.bar;  message Open { ... } |

在其他的消息格式定义中可以使用包名+消息名的方式来定义域的类型，如：

|  |
| --- |
| message Foo {    ...    required foo.bar.Open open = 1;    ...  } |

包的声明符会根据使用语言的不同影响生成的代码。对于C++，产生的类会被包装在C++的命名空间中，如上例中的Open会被封装在 foo::bar空间中；对于Java，包声明符会变为java的一个包，除非在.proto文件中提供了一个明确有java\_package；对于 Python，这个包声明符是被忽略的，因为Python模块是按照其在文件系统中的位置进行组织的。

### Ø  包及名称的解析

Protocol buffer语言中类型名称的解析与C++是一致的：首先从最内部开始查找，依次向外进行，每个包会被看作是其父类包的内部类。当然对于（foo.bar.Baz）这样以“.”分隔的意味着是从最外围开始的。ProtocolBuffer编译器会解析.proto文件中定义的所有类型名。对于不同语言的代码生成器会知道如何来指向每个具体的类型，即使它们使用了不同的规则。

## l  定义服务(Service)

如果想要将消息类型用在RPC(远程方法调用)系统中，可以在.proto文件中定义一个RPC服务接口，protocol buffer编译器将会根据所选择的不同语言生成服务接口代码及存根。如，想要定义一个RPC服务并具有一个方法，该方法能够接收 SearchRequest并返回一个SearchResponse，此时可以在.proto文件中进行如下定义：

|  |
| --- |
| service SearchService {    rpc Search (SearchRequest) returns (SearchResponse);  } |

protocol编译器将产生一个抽象接口SearchService以及一个相应的存根实现。存根将所有的调用指向RpcChannel，它是一个抽象接口，必须在RPC系统中对该接口进行实现。如，可以实现RpcChannel以完成序列化消息并通过HTTP方式来发送到一个服务器。换句话说，产生的存根提供了一个类型安全的接口用来完成基于protocolbuffer的RPC调用，而不是将你限定在一个特定的RPC的实现中。C++中的代码如下所示：

|  |
| --- |
| using google::protobuf;  protobuf::RpcChannel\* channel; protobuf::RpcController\* controller; SearchService\* service; SearchRequest request; SearchResponse response;  void DoSearch() {   // You provide classes MyRpcChannel and MyRpcController, which implement   // the abstract interfaces protobuf::RpcChannel and protobuf::RpcController.   channel = new MyRpcChannel("somehost.example.com:1234");   controller = new MyRpcController;  // The protocol compiler generates the SearchService class based on the   // definition given above.  service = new SearchService::Stub(channel);   // Set up the request.   request.set\_query("protocol buffers");    // Execute the RPC.   service->Search(controller, request, response, protobuf::NewCallback(&Done)); }  void Done() {   delete service;   delete channel;   delete controller; } |

所有service类都必须实现Service接口，它提供了一种用来调用具体方法的方式，即在编译期不需要知道方法名及它的输入、输出类型。在服务器端，通过服务注册它可以被用来实现一个RPC Server。

|  |
| --- |
| using google::protobuf;  class ExampleSearchService : public SearchService {  public:   void Search(protobuf::RpcController\* controller,               const SearchRequest\* request,               SearchResponse\* response,               protobuf::Closure\* done) {     if (request->query() == "google") {       response->add\_result()->set\_url("http://www.google.com");     } else if (request->query() == "protocol buffers") {       response->add\_result()->set\_url("http://protobuf.googlecode.com");     }     done->Run();   } };  int main() {   // You provide class MyRpcServer.  It does not have to implement any   // particular interface; this is just an example.   MyRpcServer server;    protobuf::Service\* service = new ExampleSearchService;   server.ExportOnPort(1234, service);   server.Run();    delete service;   return 0; } |

## l  选项（Options）

在定义.proto文件时能够标注一系列的options。Options并不改变整个文件声明的含义，但却能够影响特定环境下处理方式。完整的可用选项可以在google/protobuf/descriptor.proto找到。

一些选项是文件级别的，意味着它可以作用于最外范围，不包含在任何消息内部、enum或服务定义中。一些选项是消息级别的，意味着它可以用在消息定义的内部。当然有些选项可以作用在域、enum类型、enum值、服务类型及服务方法中。到目前为止，并没有一种有效的选项能作用于所有的类型。

如下就是一些常用的选择：

²  java\_package (file option): 这个选项表明生成java类所在的包。如果在.proto文件中没有明确的声明java\_package，就采用默认的包名。当然了，默认方式产生的 java包名并不是最好的方式，按照应用名称倒序方式进行排序的。如果不需要产生java代码，则该选项将不起任何作用。如：

|  |
| --- |
| option java\_package = "com.example.foo"; |

²  java\_outer\_classname (file option): 该选项表明想要生成Java类的名称。如果在.proto文件中没有明确的java\_outer\_classname定义，生成的class名称将会根据.proto文件的名称采用驼峰式的命名方式进行生成。如（foo\_bar.proto生成的java类名为FooBar.java）,如果不生成java代码，则该选项不起任何作用。如：

|  |
| --- |
| option java\_outer\_classname = "Ponycopter"; |

²  optimize\_for (fileoption): 可以被设置为 SPEED, CODE\_SIZE,or LITE\_RUNTIME。这些值将通过如下的方式影响C++及java代码的生成：

·        SPEED (default):protocol buffer编译器将通过在消息类型上执行序列化、语法分析及其他通用的操作。这种代码是最优的。

·        CODE\_SIZE: protocolbuffer编译器将会产生最少量的类，通过共享或基于反射的代码来实现序列化、语法分析及各种其它操作。采用该方式产生的代码将比SPEED要少得多，但是操作要相对慢些。当然实现的类及其对外的API与SPEED模式都是一样的。这种方式经常用在一些包含大量的.proto文件而且并不盲目追求速度的应用中。

·        LITE\_RUNTIME:protocol buffer编译器依赖于运行时核心类库来生成代码（即采用libprotobuf-lite 替代libprotobuf）。这种核心类库由于忽略了一些描述符及反射，要比全类库小得多。这种模式经常在移动手机平台应用多一些。编译器采用该模式产生的方法实现与SPEED模式不相上下，产生的类通过实现 MessageLite接口，但它仅仅是Messager接口的一个子集。

|  |
| --- |
| option optimize\_for = CODE\_SIZE; |

²  cc\_generic\_services, java\_generic\_services, py\_generic\_services (file options): 在C++、java、python中protocol buffer编译器是否应该基于服务定义产生抽象服务代码。由于历史遗留问题，该值默认是true。但是自2.3.0版本以来，它被认为通过提供代码生成器插件来对RPC实现更可取，而不是依赖于“抽象”服务。

|  |
| --- |
| // This file relies on plugins to generate service code.  option cc\_generic\_services = false;  option java\_generic\_services = false;  option py\_generic\_services = false; |

²  message\_set\_wire\_format (message option):如果该值被设置为true，该消息将使用一种不同的二进制格式来与Google内部的MessageSet的老格式相兼容。对于Google外部的用户来说，该选项将不会被用到。如下所示:

|  |
| --- |
| message Foo {    option message\_set\_wire\_format = true;    extensions 4 to max;  } |

²  packed (field option): 如果该选项在一个整型基本类型上被设置为真，则采用更紧凑的编码方式。当然使用该值并不会对数值造成任何损失。在2.3.0版本之前，解析器将会忽略那些非期望的包装值。因此，它不可能在不破坏现有框架的兼容性上而改变压缩格式。在2.3.0之后，这种改变将是安全的，解析器能够接受上述两种格式，但是在处理protobuf老版本程序时，还是要多留意一下。

|  |
| --- |
| repeated int32 samples = 4 [packed=true]; |

²  deprecated (field option): 如果该选项被设置为true，表明该字段已经被弃用了，在新代码中不建议使用。在多数语言中，这并没有实际的含义。在java中，它将会变成一个 @Deprecated注释。也许在将来，其它基于语言声明的代码在生成时也会如此使用，当使用该字段时，编译器将自动报警。如：

|  |
| --- |
| optional int32 old\_field = 6 [deprecated=true]; |

**Ø 自定义选项**

ProtocolBuffers允许自定义并使用选项。该功能应该属于一个高级特性，对于大部分人是用不到的。由于options是定在 google/protobuf/descriptor.proto中的，因此你可以在该文件中进行扩展，定义自己的选项。如：

|  |
| --- |
| import "google/protobuf/descriptor.proto";    extend google.protobuf.MessageOptions {    optional string my\_option = 51234;  }    message MyMessage {    option (my\_option) = "Hello world!";  } |

在上述代码中，通过对MessageOptions进行扩展定义了一个新的消息级别的选项。当使用该选项时，选项的名称需要使用（）包裹起来，以表明它是一个扩展。在C++代码中可以看出my\_option是以如下方式被读取的。

|  |
| --- |
| string value = MyMessage::descriptor()->options().GetExtension(my\_option); |

在Java代码中的读取方式如下：

|  |
| --- |
| String value = MyProtoFile.MyMessage.getDescriptor().getOptions().getExtension(MyProtoFile.myOption); |

正如上面的读取方式，定制选项对于Python并不支持。定制选项在protocol buffer语言中可用于任何结构。下面就是一些具体的例子：

|  |
| --- |
| import "google/protobuf/descriptor.proto";    extend google.protobuf.FileOptions {    optional string my\_file\_option = 50000;  }  extend google.protobuf.MessageOptions {    optional int32 my\_message\_option = 50001;  }  extend google.protobuf.FieldOptions {    optional float my\_field\_option = 50002;  }  extend google.protobuf.EnumOptions {    optional bool my\_enum\_option = 50003;  }  extend google.protobuf.EnumValueOptions {    optional uint32 my\_enum\_value\_option = 50004;  }  extend google.protobuf.ServiceOptions {    optional MyEnum my\_service\_option = 50005;  }  extend google.protobuf.MethodOptions {    optional MyMessage my\_method\_option = 50006;  }    option (my\_file\_option) = "Hello world!";    message MyMessage {    option (my\_message\_option) = 1234;      optional int32 foo = 1 [(my\_field\_option) = 4.5];    optional string bar = 2;  }    enum MyEnum {    option (my\_enum\_option) = true;      FOO = 1 [(my\_enum\_value\_option) = 321];    BAR = 2;  }    message RequestType {}  message ResponseType {}    service MyService {    option (my\_service\_option) = FOO;      rpc MyMethod(RequestType) returns(ResponseType) {      // Note:  my\_method\_option has type MyMessage.  We can set each field      //   within it using a separate "option" line.      option (my\_method\_option).foo = 567;      option (my\_method\_option).bar = "Some string";    }  } |

注：如果要在该选项定义之外使用一个自定义的选项，必须要由包名 + 选项名来定义该选项。如：

|  |
| --- |
| // foo.proto  import "google/protobuf/descriptor.proto";  package foo;  extend google.protobuf.MessageOptions {    optional string my\_option = 51234;  }  // bar.proto  import "foo.proto";  package bar;  message MyMessage {    option (foo.my\_option) = "Hello world!";  } |

最后一件事情需要注意：因为自定义选项是可扩展的，它必须象其它的域或扩展一样来定义标识号。正如上述示例，[50000－99999]已经被占用，该范围内的值已经被内部所使用，当然了你可以在内部应用中随意使用。如果你想在一些公共应用中进行自定义选项，你必须确保它是全局唯一的。可以通过[protobuf-global-extension-registry@google.com](mailto:protobuf-global-extension-registry@google.com)来获取全局唯一标识号。

## l  生成访问类

可以通过定义好的.proto文件来生成Java、Python、C++代码，需要基于.proto文件运行protocol buffer编译器protoc。运行的命令如下所示：

|  |
| --- |
| protoc --proto\_path=IMPORT\_PATH --cpp\_out=DST\_DIR --java\_out=DST\_DIR --python\_out=DST\_DIR path/to/file.proto |

·        IMPORT\_PATH声明了一个.proto文件所在的具体目录。如果忽略该值，则使用当前目录。如果有多个目录则可以 对--proto\_path 写多次，它们将会顺序的被访问并执行导入。-I=IMPORT\_PATH是它的简化形式。

·        当然也可以提供一个或多个输出路径：

o   --cpp\_out 在目标目录DST\_DIR中产生C++代码，可以在 http://code.google.com/intl/zh-CN/apis/protocolbuffers/docs/reference/cpp-generated.html中查看更多。

o   --java\_out 在目标目录DST\_DIR中产生Java代码，可以在 http://code.google.com/intl/zh-CN/apis/protocolbuffers/docs/reference/java-generated.html中查看更多。

o   --python\_out 在目标目录 DST\_DIR 中产生Python代码，可以在http://code.google.com/intl/zh-CN/apis/protocolbuffers/docs/reference/python-generated.html中查看更多。

     作为一种额外的使得，如果DST\_DIR 是以.zip或.jar结尾的，编译器将输出结果打包成一个zip格式的归档文件。.jar将会输出一个 [**Java**](http://lib.csdn.net/base/java)JAR声明必须的manifest文件。注：如果该输出归档文件已经存在，它将会被重写，编译器并没有做到足够的[**智能**](http://lib.csdn.net/base/aiplanning)来为已经存在的归档文件添加新的文件。

·        你必须提供一个或多个.proto文件作为输入。多个.proto文件能够一次全部声明。虽然这些文件是相对于当前目录来命名的，每个文件必须在一个IMPORT\_PATH中，只有如此编译器才可以决定它的标准名称。

from:<http://www.open-open.com/home/space.php?uid=37924&do=blog&id=5873>

=========================================================

ProtoBuf开发者指南：http://gashero.yeax.com/?p=108

官方：<http://code.google.com/p/protobuf/>

语言指南  
http://code.google.com/intl/zh-CN/apis/protocolbuffers/docs/proto.html  
风格  
http://code.google.com/intl/zh-CN/apis/protocolbuffers/docs/style.html

# [protobuf在网络编程中的应用思考](http://blog.csdn.net/ciml/article/details/5753367" \t "_blank)

分类： [基础框架库](http://blog.csdn.net/ciml/article/category/713253)2010-07-21 17:40 54306人阅读 [评论](http://blog.csdn.net/ciml/article/details/5753367#comments)(19) 收藏 [举报](http://blog.csdn.net/ciml/article/details/5753367#report)

[编程](http://www.csdn.net/tag/%e7%bc%96%e7%a8%8b)[网络](http://www.csdn.net/tag/%e7%bd%91%e7%bb%9c)[google](http://www.csdn.net/tag/google)[string](http://www.csdn.net/tag/string)[文档](http://www.csdn.net/tag/%e6%96%87%e6%a1%a3)[tcp](http://www.csdn.net/tag/tcp)

目录[(?)](http://blog.csdn.net/ciml/article/details/5753367)[[+]](http://blog.csdn.net/ciml/article/details/5753367" \t "_blank" \o "展开)

### protobuf简介

protobuf是google提供的一个开源序列化框架，类似于XML，JSON这样的数据表示语言，其最大的特点是基于二进制，因此比传统的XML表示高效短小得多。虽然是二进制数据格式，但并没有因此变得复杂，开发人员通过按照一定的语法定义结构化的消息格式，然后送给命令行工具，工具将自动生成相关的类，可以支持java、c++、python等语言环境。通过将这些类包含在项目中，可以很轻松的调用相关方法来完成业务消息的序列化与反序列化工作。

protobuf在google中是一个比较核心的基础库，作为分布式运算涉及到大量的不同业务消息的传递，如何高效简洁的表示、操作这些业务消息在google这样的大规模应用中是至关重要的。而protobuf这样的库正好是在效率、数据大小、易用性之间取得了很好的平衡。

更多信息可参考[官方文档](http://code.google.com/apis/protocolbuffers/docs/overview.html)

### 例子介绍

先[下载protobuf-2.3.0.zip](http://code.google.com/p/protobuf/downloads/list)源代码库，下载后解压，选择vsprojects目录下的protobuf.sln解决方案打开，编译整个方案顺利成功。其中有一些测试工程，库相关的工程是libprotobuf、libprotobuf-lite、libprotoc和protoc。其中protoc是命令行工具。在example目录下有一个地址薄消息的例子，业务消息的定义文件后缀为.proto，其中的addressbook.proto内容为：

package tutorial;

option java\_package = "com.example.tutorial";

option java\_outer\_classname = "AddressBookProtos";

message Person {

  required string name = 1;

  required int32 id = 2;        // Unique ID number for this person.

  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;

}

// Our address book file is just one of these.

message AddressBook {

  repeated Person person = 1;

}

该定义文件，定义了地址薄消息的结构，顶层消息为AddressBook，其中包含多个Person消息，Person消息中又包含多个PhoneNumber消息。里面还定义了一个PhoneType的枚举类型。

类型前面有required表示必须，optional表示可选，repeated表示重复，这些定义都是一目了然的，无须多说。关于消息定义的详细语法可参考官方文档。

现在用命令行工具来生成业务消息类，切换到protoc.exe所在的debug目录，在命令行敲入：

protoc.exe--proto\_path=../../examples --cpp\_out=../../examples ../../examples/addressbook.proto

该命令中--proto\_path参数表示.proto消息定义文件路径，--cpp\_out表示输出c++类的路径，后面接着是addressbook.proto消息定义文件。该命令会读取addressbook.proto文件并生成对应的c++类头文件和实现文件。执行完后在examples目录生存了addressbook.pb.h和addressbook.pb.cpp。

现在新建两个空控制台工程，第一个不妨叫AddPerson，然后把examples目录下的add\_person.cc、addressbook.pb.h和addressbook.pb.cpp加入到该工程，另一个工程不妨叫ListPerson，将examples目录下的list\_people.cc、addressbook.pb.h和addressbook.pb.cpp加入到该工程，在两个工程的项目属性中附加头文件路径../src。两个工程的项目依赖都选择libprotobuf工程（库）。

给AddPerson工程添加一个命令行参数比如叫addressbook.dat用于将地址薄信息序列化写入该文件，然后编译运行AddPerson工程，根据提示输入地址薄信息:

输入完成后，将序列化到addressbook.dat文件中。

在ListPerson工程的命令行参数中加读取文件参数../AddPerson/addressbook.dat，然后在运行ListPerson工程，可在list\_people.cc的最后设个断点，避免命令行窗口运行完后关闭看不到结果：

写入地址薄的操作，关键操作就是调用address\_book.SerializeToOstream进行序列化到文件流。

而读取操作中就是address\_book.ParseFromIstream从文件流反序列化，这都是框架自动生成的类中的方法。

其他操作都是业务消息的字段set/get之类的对象级操作，很明了。更详细的API参考官方文档有详细说明。

### 在TCP网络编程中的考虑

从上面的例子可以看出protobuf这样的库是很方便高效的，那么自然的想到在网络编程中用来做业务消息的序列化、反序列化支持。在基于UDP协议的网络应用中，由于UDP本身是有边界，那么用protobuf来处理业务消息就很方便。但在TCP应用中，由于TCP协议没有消息边界，这就需要有一种机制来确定业务消息边界。在TCP网络编程中这是必须面对的问题。

注意上面的address\_book.ParseFromIstream调用，如果流参数的内容多一个字节或者少一个字节，该方法都会返回失败（虽然某些字段可能正确得到结果了），也就是说送给反序列化的数据参数除了格式正确还必须有正确的大小。因此在tcp网络编程中，要反序列化业务消息，就要先知道业务数据的大小。而且在实际应用中可能在一个发送操作中，发送多个业务消息，而且每个业务消息的大小、类型都不一样。而且可能发送很大的数据流，比如文件。

显然消息边界的确认问题和protobuf库无关，还得自己搞定。在官方文档中也提到，protobuf并不太适合来作[**大数据**](http://lib.csdn.net/base/hadoop)的处理，当业务消息超过1M时，就应该考虑是否应该用另外的替代方案。当然对于大数据，你也可以分割为多个小块用protobuf做小块消息封装进行传递。但对很多应用这样的作法显得比较多余，比如发送一个大的文件，一般是在接收方从协议栈收到多少数据就写多少数据到磁盘，这是一种边接收边处理的流模式，这种模式基本上和每次收到的数据量没有关系。这种模式下再采用分割成小消息进行反序列化就显得多此一举了。

由于每个业务消息的大小和处理方式都可能不一样，那么就需要独立抽象出一个边界消息来区分不同的业务消息，而且这个边界消息的格式和大小必须固定。对于网络编程熟手，可能早已经想到了这样的消息，我们可以结合protobuf库来定义一个边界消息，不妨叫BoundMsg：

message BoundMsg

{

  required int32 msg\_type = 1;

  required int32 msg\_size = 2;

}

可以根据需要扩充一些字段，但最基本的这两个字段就够用了。我们只需要知道业务消息的类型和大小即可。这个消息大小是固定的8字节，专门用来确定数据流的边界。有了这样的边界消息，在接收端处理任何业务消息就很灵活方便了，下面是接收端处理的简单伪代码示例：

if(net\_read(buf,8))

{

  boundMsg.ParseFromIstream(buf);

  switch(boundMsg.msg\_type)

  {

    case BO\_1:

      if(net\_read(bo1Buf,boundMsg.msg\_size))

      {

        bo1.ParseFromIstream(bo1Buf);

        ....

      }

     break;

    case BO\_2:

      if(net\_read(bo2Buf,boundMsg.msg\_size))

      {

        bo2.ParseFromIstream(bo2Buf);

        ....

      }

     break;

    case FILE\_DATA:

      count = 0;

      while(count < boundMsg.msg\_size)

      {

        piece\_size = net\_read(fileBuf,1024);

        write\_file(filename,fileBuf,piece\_size);

        count = count + piece\_size;

      }

      break;

  }

}

注意上面如果FILE\_DATA消息后，还紧接其他业务消息的话，需要小心，即count累计出的值可能大于

boundMsg.msg\_size的值，那么多出来的实际上应该是下一个边界消息数据了。为了避免处理的复杂性，上面所有的循环网络读取操作（上面BO\_1，BO\_2都可能需要循环读取，为了简化没有写成循环）的缓冲区位置和大小参数应该动态调整，即每次读取时传递的都是还期望读取的数据大小，对于文件的话，可能特殊点，因为边读取边写入，就没有必要事先要分配一个文件大小的缓冲区来存放数据了。对于文件分配一个小缓冲区来读，注意确认下边界即可。

上面是我的一点考虑，不妥之处还请大家讨论交流。想想借助于ACE、MINA这样的网络编程框架，然后结合protobuf这样的序列化框架，网络编程中技术基础设施层面的东西就给我们解决得差不多了，我们可以真正只关注于业务的实现。