[google protocol buffer -2-.proto 定义规则](http://blog.csdn.net/gd2008/article/details/8269118)

message为主要关键字，类似于java中的class。  
定义简单message类型

SearchRequest.proto定义了每个查询请求的消息格式，每个请求都会有查询关键词query，查询结果的页数，每页的结果数量这三个属性。于是

message SearchRequest{  
        required string query = 1;  
        optional int32 page\_number = 2;  
        optional int32 result\_per\_page =3;  
        repeated int32 samples = 4 [packed=true];      
}

message定义了三个field，每个field由名字和类型来组成。

* 指定field类型

在这个例子中，SearchRequest的field都是基本类型，两个integer（page\_number和result\_per\_page）和一个Stirng(query)，也可以指定复杂的类型属性，包括枚举和其它类型。

* 分配标签

每个field都是唯一数字的标记，这是用来标记这个field在message二进制格式中的位置的，一旦使用就不能再修改顺序了。

注:标记从1-15只有一个字节编码，包括自增长属性（更多的见Protocol Buffer Encoding）

标记从16-2047占用两个字节。因此尽量频繁使用1-15，记住为未来的扩展留下一些位置。

最小的tag你可以定义为1，最大2的29次方-1  536870922.你同样不能使用19000-19999（这个位置已经被GPB自己实现），

* 指定field规则

message SearchRequest{  
        required string query = 1;  
        optional int32 page\_number = 2;  
        optional int32 result\_per\_page =3;  
        repeated int32 samples = 4 **[**packed=true**]**;      
}

由于历史原因，repeated字段如果是基本数字类型的话，不能有效地编码。现在代码可以使用特殊选项[packed=true]来得到更有效率的编码。

注： 由于required是永远的，应该非常慎重地给message某个字段设置为required。如果未来你希望停止写入或者输出某个required字段，那就会成为问题；因为旧的reader将以为没有这个字段无法初始化message，会丢掉这部分信息。一些来自google的工程师们指出使用required弊大于利，尽量使用optional和repeated。

这个观点并不是通用的。

* 更多message类型

多个message类型能被定义在一个简单的.proto文件中，通常是创建具有关联关系的message时候这么作。

message SearchRequest{  
        required string query = 1;  
        optional int32 page\_number = 2;  
        optional int32 result\_per\_page =3;  
        repeated int32 samples = 4 **[**packed=true**]**;      
}

* **添加注释**

使用c/C++ style

message SearchRequest{  
        required string query = 1;  //  
        optional int32 page\_number = 2;  // which page number do we want?  
        optional int32 result\_per\_page =3; // Number of results to return per page?  
        repeated int32 samples = 4 [packed=true];      
}

* **.proto文件自动生成代码**

protocol buffer编译一个proto文件，生成对应语言的代码。

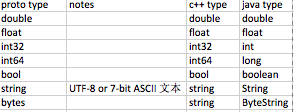
大概包括各个字段的get和set方法，序列化message到输出流的方法，从输入流转成message的方法。

C++，为每个proto生成一个.h和.cc文件

Java，为每个proto生成一个.java文件

Python，有点不同，生成一个module

* **基本属性**



* **optional字段和默认值**

当含有optional字段的message从流转换成对象的时候，如果没有包含optional字段的数据，那么对象的optional字段会设置成默认值。

默认值可以作为message的描述出现。举个例子：

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

如

pasting

  果没有指定默认值的话，string 默认为空串，bool 默认为false，数字类型默认0，枚举类型，默认为类型定义中的第一个值，

* **Enumerations**

如果字段的属性值是固定的几个值，可以使用枚举

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 = WEB];  
}

* **自定义消息类型**

可以使用message类型做字段的属性，看例子：

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

* **import 定义**

上面的例子SearchResponse 与Result在一个.proto文件中。其实也可以使用另一个.proto文件来定义字段类型。

你可以通过import来定义。

import "myproject/other\_protos.proto";

protocol编译器查找引入文件是通过编译器的命令参数 -I/--proto\_path

如果没有指定，就在protoc执行目录下寻找。

The protocol compiler searches for imported files in a set of directories specified on the protocol compiler command line using the -I/--proto\_path flag.

If no flag was given, it looks in the directory in which the compiler was invoked.

In general you should set the --proto\_path flag to the root of your project and use fully qualified names for all imports.

* **内部类**

你可以定义和使用内部message类。

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

如果要引用内部类，则通过parent.type方式来调用

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

还可以很深、很深的内部类

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;  
    }  
  }  
}

* **Groups**

 废弃的属性，了解即可，采用内部类代替。

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

**Extentions**

extensions 声明一个消息中的一定范围的field的顺序数字用于进行扩展。其它人可以在自己的.proto文件中重新定义这些消息field，而不需要去修改原始的.proto文件

message Foo {  
  // http://www.blogjava.net/Images/dot.gif  
  extensions 100 to 199;  
}

这些说明100-199的field是保留的。其它用户可以用这些field在他们自己的.proto文件中添加新的fields给Foo。举例：

extend Foo {  
  optional int32 bar = 126;  
}

说明 Foo有一个optional的int32类型的名称为bar的field

当Foo的message编码后，数据格式就跟用户在Foo中定义一个新的field完全一样。但是你在程序中访问extension field的方式与访问正常的属性略微有点不同。生成的extensions的访问代码是不同的。举例：c++中如何set属性bar的值：

Foo foo;  
foo.SetExtension(bar,15);

同样，Foo 定义了模板访问器  HasExtendsion(),ClearExtension(),GetExtension(),MutableExtension(),AddExtension().

所有 访问         
注： extensions能使用任何field类型，包括自定义消息类型。

* **内嵌的extensions**

能声明extensions在另一个message中

message Baz {  
  extend Foo {  
    optional int32 bar = 126;  
  }  
  http://www.blogjava.net/Images/dot.gif  
}

在这个例子中, the C++ 代码访问访问这个属性:

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

换句话说，这么做唯一影响是bar定义在了Baz的范围之内。

注意：容易混淆的地方 声明一个消息内部的继承类并不意味着外部类和extended类有任何关系。特别 以上的例子并不意味着Baz是任何Foo的子类。这些只是意味着符号bar是声明在Baz的范围之内的，它看起来更像是个静态成员。

一个通用的模式是在extensions的field范围内来定义extensions，举例说明，这里有一个Foo的extension作为Baz的一部分的属性类型是Baz

message Baz {  
  extend Foo {  
    optional Baz foo\_ext = 127;  
  }  
  http://www.blogjava.net/Images/dot.gif  
}

没有必要非得在message内部定义一个extension的类型。你也可以这么做：

message Baz {  
  http://www.blogjava.net/Images/dot.gif  
}  
// This can even be in a different file.  
extend Foo {  
  optional Baz foo\_baz\_ext = 127;  
}

事实上，上面的这个语法更加有效地避免混淆。正如上文所述，内部的那种语法语法对于不是熟悉extensions的人来说，经常会错认为子类。

* **选择Extension 顺序数字**

非常重要的一点是双方不能使用同样数字添加一样的message类型，这样extension会被解释为错误类型。

可能需要有一个关于field的数字顺序的约定来保证你的project不会发生这样的重复的问题。

如果你的field数字比较大的话，可以使用max来指定你的textension范围上升到最大的范围

message Foo {  
  extensions 1000 to max;  
}  
max is 229 - 1, or 536,870,911.

19000-19999是protocol buffers的使用的字段，所以这个范围内的数字需要区别开来。

Packages

可以给一个.protol文件增加一个optional的package描述，来保证message尽量不会出现名字相同的重名。

package foo.bar;  
message Open {   
http://www.blogjava.net/Images/dot.gif   
}

也可以在指定field类型的时候使用

message Foo {  
  http://www.blogjava.net/Images/dot.gif  
  required foo.bar.Open open = 1;  
  http://www.blogjava.net/Images/dot.gif  
}

package会根据选择的语言来生成不同的代码：

C++      生成的classes是用C++的namespace来区分的。举例：Open would be in the namespace foo::bar。  
Java      package用于Java的package，除非你单独的指定一个option java\_package 在.proto文件中。  
Python   package是被忽略的，因为Python的modules是通过它们的文件位置来组织的。

* **Packages和name**

在protocol buffer中package名称的方案看起来像C++，首先，最里面的范围被搜索，然后搜索次一级的范围，

每个package被认为在他的父package内。一个. （.foo.bar.Baz）意味着从最外层开始.

options

在一个proto文件中，还可以存在一些options。Options不能改变一个声明的整体的意义，但是可以影响一定的上下文。

可用的options的完整list定义在 Google/protobuf/descriptor.proto

一些options是第一级的，意味着它们应该被写在顶级范围，而不是在任何message,enum，sercie的定义中。

一些options是message级别的，意味着它们应该被写入message的描述中，

一些options是field-level级别的，意味着它们应该被写入field的描述中，

options也可以被写入enum类型中，enum的值，service类型 和service方法；

列举了常用的options：

java\_package(file option)

定义生成的java class的package。如果在proto文件中没有明确的java\_package选项，那么默认会使用package关键字指定的package名。

但是proto package通常不会好于Java packages，因为proto packages通常不会以domain名称开始。

如果不生成java代码，此选项没有任何影响。

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

java\_outer\_classname:(file option)

指定想要生成的class名称，如果此参数没有指定的话，那么默认使用.proto文件名来做为类名，并且采用驼峰表示（比如：foo\_bar.proto 为 FooBar.java）

如果不生成java代码，此选项没有影响。

option java\_outer\_classname = "Ponycopter";

optimize\_for (file option)

可以设置为speed、code\_size或者lite\_runtime.

SPEED:默认。protocol编译器会生成classes代码，提供了message类的序列化、转换和其它通用操作。这个代码是被高度优化过的。

CODE\_SIZE: protocol编译器会生成最小的classes，并且依赖共享、基于反射的代码实现序列化、转换和其它通用操作。生成的classes代码小于speed，但是操作会慢一点。classes会实现跟SPEED模式一样的公共API。这个模式通常用在一个应用程序包含了大量的proto文件，但是并不需要所有的代码都执行得很快

LITE\_RUNTIME: protocol编译器会生成仅仅依赖 lite 运行库（libprotobuf-lite代替libprotobuf）。lite运行时比全量库小很多，省略了某种特性（如： descriptors and reflection）这个选项对于运行在像移动手机这种有约束平台上的应用更有效。 编译器仍然会对所有方法生成非常快的代码实现，就像SPEED模式一样。protocol编译器会用各种语言来实现MessageList接口，但是这个接口仅仅提供了其它模式实现的Message接口的一部分方法子集。

例子

option optimize\_for = CODE\_SIZE;

cc\_generic\_services, java\_generic\_services, py\_generic\_services (file options)

无论如何，protoc编译器会生成基于C++,Java,Python的抽象service代码，这些默认都是true。截至到2.3.0版本，RPC实现提供了代码生成插件去生成代码，不再使用抽象类。

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

message\_set\_wire\_format (message option)

如果设置为true，消息使用不同的二进制格式来兼容谷歌内部使用的称为MessageSet的旧格式。用户在google以外使用，将不再需要使用这个option。

消息必须按照以下声明

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

packed (field option)

如果设置为true， 一个repeated的基本integer类型的field，会使用一种更加紧凑的压缩编码。请注意，在2.3.0版之前，protocol生成的解析逻辑收到未预期的压缩的数据将会忽略掉。因此，改变一个已经存在的field，一定会破坏其线性兼容性。在2.3.0以后，这种改变就是安全的，解析逻辑可以识别压缩和不压缩的格式，但是，一定要小心那些使用原先旧版本的protocol的程序。

repeated int32 samples = 4 **[**packed=true**]**;

deprecated (field option):

如果设置为true，表示这个field被废弃，应该使用新代码。大多数语言中，这个没有任何影响。在java中，会生成@Deprecated的注释。未来，其它语言代码在field的访问方法上也会生成相应的注释。

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

* **自定义options**

protocol buffer还允许你自定义options。这是个高级特性，大多数人并不需要。options其实都定义在 google/protobuf/descriptor.proto文件中。

自定义的options是简单的，继承这些messages

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

这里我们定义了一个message级别的消息选项，当使用这个options的时候，选项的名称必须用括号括起来，以表明它是一个extension。

我们在C++中读取my\_option的值就像下面这样：

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

这里，MyMessage::descriptor()->options()返回的MessageOptions protocol类型 message。

读取自定义就如同读取继承属性一样。

在Java中

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

自定义options可以对任何message的组成元素进行定义

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";  
  }  
}

如果想使用在package里面的自定义的option，必须要option前使用包名，如下

// 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：既然自定义的options是extensions，他们必须指定field number就像其它field或者extension一样。如果你要在公共应用中使用自定义的options，那么一定要确认你的field numbers是全局唯一的

你能通过多选项带有一个extension 把它们放入一个子message中

message FooOptions {  
  optional int32 opt1 = 1;  
  optional string opt2 = 2;  
}  
extend google.protobuf.FieldOptions {  
  optional FooOptions foo\_options = 1234;  
}  
// usage:  
message Bar {  
  optional int32 a = 1 **[**(foo\_options.opt1) = 123, (foo\_options.opt2) = "baz"**]**;  
  // alternative aggregate syntax (uses TextFormat):  
  optional int32 b = 2 **[**(foo\_options) = { opt1: 123 opt2: "baz" }**]**;  
}

生成class代码

为了生成java、python、C++代码，你需要运行protoc编译器 protoc 编译.proto文件。

编译器运行命令：

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

import\_path 查找proto文件的目录，如果省略的话，就是当前目录。存在多个引入目录的话，可以使用--proto\_path参数来多次指定，

-I=IMPORT\_PATH就是--proto\_path的缩写

输出目录

--cpp\_out       生成C++代码在DST\_DIR目录

--java\_out      生成Java代码在DST\_DIR目录

--python\_out    生成Python代码在DST\_DIR目录

有个额外的好处，如果DST是.zip或者.jar结尾，那么编译器将会按照给定名字输入到一个zip压缩格式的文件中。

输出到.jar会有一个jar指定的manifest文件。注意 如果输出文件已经存在，它将会被覆盖；编译器的智能不足以自动添加文件到一个存在的压缩文件中。

你必须提供一个或者多个.proto文件用作输入。虽然文件命名关联到当前路径，每个文件必须在import\_path路径中一边编译器能规定它的规范名称  
  
  
更新message

如果一个message 不再满足所有需要，需要对字段进行调整.(举例:对message增加一个额外的字段，但是仍然有支持旧格式message的代码在运行)

要注意以下几点:  
1、不要修改已经存在字段的数字顺序标示

2、可以增加optional或者repeated的新字段。这么做以后，所有通过旧格式message序列化的数据都可以通过新代码来生成对应的对象，正如他们不会丢失任何required元素。

你应该为这些元素添加合理的默认值，以便新代码可以与旧代码生成的消息交互。 新代码创建的消息中旧代码不存在的字段，在解析的时候，旧代码会忽略掉新增的字段。

无论如何，未知的field不会被丢弃，如果message晚点序列化，为。

注意 未知field对于Python来说当前不可用。

3、非required字段都可以转为extension ，反之亦然，只要type和number保持不变。

4、int32, uint32, int64, uint64, and bool 是全兼容的。这意味着你能改变一个field从这些类型中的一个改变为另一个，而不用考虑会打破向前、向后兼容性。

如果一个数字是通过网络传输而来的相应类型转换，你将会遇到type在C++中遇到的问题（e.g. if a 64-bit number is read as an int32, it will be truncated to 32 bits）

5、sint32 and sint64 彼此兼容,但是不能兼容其它integer类型.

6、string and bytes 在UTF-8编码下是兼容的.

7、如果bytes包含一个message的编码,内嵌message与bytes兼容.

8、fixed32 兼容 sfixed32,  fixed64 兼容 sfixed64.

9、optional 兼容 repeated. 用一个repeat字段的编码结果作为输入,认为这个字段是可选择的客户端会这样处理,如果是原始类型的话,获得最后的输入作为相应的option值;如果是message 类型,合并所有输入元素.

10、更改默认值通常是OK的.要记得默认值并不会通过网络发送,如果一个程序接受一个特定字段没有设置值的消息,应用将会使用自己的版本协议定义的默认值,不会看见发送者的默认值.

# [基于netty的项目中使用protobuf，巧妙定义proto完成不同消息的编码和解码处理](http://blog.csdn.net/gd2008/article/details/8233841)

分类： [Netty](http://blog.csdn.net/gd2008/article/category/1271341)2012-11-28 12:53 1593人阅读 [评论](http://blog.csdn.net/gd2008/article/details/8233841#comments)(2) [收藏](javascript:void(0);) [举报](http://blog.csdn.net/gd2008/article/details/8233841#report)

在基于netty的项目中使用protobuf，需要处理不同的消息，因此需要不同的编码和解码方式（如下）

p.addLast("protobufDecoder", new ProtobufDecoder(Communication.TRequest.getDefaultInstance()));

p.addLast("protobufDecoder", new ProtobufDecoder(Communication.TResponse.getDefaultInstance()));

但netty中每个管道仅能注册一个解码和编码的方式，经过研究，想到把这些不同的消息封装成一个消息组，在不同的处理逻辑中再get相应的消息即可，而管道注册那边只需要注册一个消息格式：

p.addLast("protobufDecoder", new ProtobufDecoder(Communication.ProtocolMessage.getDefaultInstance()));

proto文件：

//封装Request和Response消息，这样netty可以统一编码和解码

message ProtocolMessage

{

optional TRequest tRequest=1;

optional TResponse tResponse=2;

}

终于成功解决了问题！