# 语言指南

Translated by Jeff

王晖 译

（译者联系方式为：lonelystellar@gmail.com）



注：关于本文档的翻译错误（包括语法错误，错别字）或中文技术交流，可以发送邮件至译者邮箱：lonelystellar@gmail.com，我们共同研究，共同进步，不胜感激。

本指南描述了怎么使用 Protocol Buffers 语言来构造 Protocol Buffers 数据,包括.proto 文件的语法和怎么使用.proto 文件来生产数据访问类.
这也是一个引用指南 –提供了一个使用了在本文中描述的许多特性的示例, 可以根据选择不同的编程语言来阅读 指南.

## 定义一个消息类型

首先来看一个非常简单的示例.假如定义了一个搜索请求消息的格式, 并且每个搜索请求都有一个查询字符串, 那么就能得到那个你想要的结果页面和每页的页数.下面是一个定义了消息类型的 .proto 文件.

```
message SearchRequest {
 required string query = 1;
 optional int32 page_number = 2;
 optional int32 result_per_page = 3;
}
```

SearchRequest 消息定义了三个字段(键值对),每一个数据都包含在消息这种类型中, 每个字段都有一个名字和类型.

**指定字段类型**

在上面的示例中,所有的字段都是 标量类型(scalar types):2 个整型(page_number 和 result_per_page) 以及一个字符串类型(query). 但是,还能为字段指定组合类型,如枚举 和其他消息类型.

**分配标签**

就像你所看到的,在消息中定义的每个字段都有一个**唯一的编号标签**. 这些标签被用来标明在消息二进制格式中的字段, 并且在消息类型在使用时不能被改变.注意标签的取值从 1 到 15 之间编码时需要占用一个字节. 标签取值从 16 到 2047 需要占用 2 个字节.因此在非常频繁的消息元素中保留 1 到 15 的标签号. 同时别忘了给在将来将会添加进来的频繁出现的元素留点空间.

能指定的最小的标签编号是 1,最大的是 $2^{29}$ - 1,也就是 536,870,911.还有不能使用 19000 到 19999 (FieldDescriptor::kFirstReservedNumber 到 FieldDescriptor::kLastReservedNumber), 因为他们是给 Protocol Buffers 实现保留的- 如果在.proto 文件中使用了保留编号 Protocol Buffers 编译器将会报错.

**指定字段规则**

只能指定如下的其中一种给消息字段:

- required: 消息(well-formed message)必须含有此字段.
- optional: 消息中此字段只能有一个或不填.
- repeated:消息中 此字段可能填重复多次(包括 0),对重复值的顺序将被保留.

由于历史的原因,基本数字类型的 repeated 字段不能有效的编码.新代码可以使用选项[packed=true] 来得到更加有效的编码,例如:

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

**必填字段是永久性的** 当设置字段为 required 时应该非常小心.如果在一些地方不需要写或发送一个必填字段,那么它在转换它为可选字段时将会出问题-以前的读取者将

会认为这些消息是没有这些字段的,是不完整的,并且还有可能丢弃或拒绝它.因此在写应用时需要为 Protocol Buffers 指定自己的验证方式.在 Google 的一些工程师总结出一个结论,那就是不用 required 字段比用更好;他们更喜欢只用 optional 字段和 repeated 字段,但是,这并不代表所有的观点.

**加入一个消息类型**

能在单个.proto 文件中定义多种消息类型.这在定义多种相关联的消息时非常有有–例如,如果需要定义对应于 SearchResponse 消息类型的答复消息格式, 可以在同一个.proto 文件中加入如下代码:

```
message SearchRequest {
 required string query = 1;
 optional int32 page_number = 2;
 optional int32 result_per_page = 3;
}

message SearchResponse {
 ...
}
```

**加入注释**

用 C/C++的 // 语法在 .proto 文件中加入注释.

```
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.
}
```

**从.proto 中会生成什么?**

当在.proto 文件上运行 Protocol Buffer 编译器 , 编译器会生成你所选择的编程语言的能工与文件中描述的消息类型一同工作的代码 , 包括获得(getting)和设置(setting)字段的值, 序列号消息到输出流, 以及解析从输入流中解析消息.

对于 **C++**, 编译器会针对每一个.proto 文件生成 .h 和.cc 文件, 里面包含了文件中定义的每个消息类型对应的类.

对于 **Java**, 编译器生成一个包含每个消息类型的类的.java 文件,并且还有特定的创建消息类实例的 Builder 类.

**Python** 就稍微有点不同了– Python 编译器根据.proto 文件为每一个消息类型生成一个含有静态描述器的模块, 可被用来与 *metaclass* 一起在运行时创建必要的 Python 数据访问类.

在如下不同的语言的指南中能找到更多关于使用 API 的信息. 如需了解更详细的可阅读 API 引用.

## 标量值类型

一个标量消息字段可以指定如下类型中的任意一个– 下表展示了在.proto 文件中的类型, 以及对应的自动生成类的类型:

| .proto 类型 | 说明 | C++ 类型 | Java 类型 |
|---|---|---|---|
| double | | double | double |
| float | | float | float |
| int32 | 使用可变长度编码.编码负数时效率不高-如果字段中有负数用 sint32 来替代. | int32 | int |
| int64 | 使用可变长度编码.编码负数时效率不高-如果字段中有负数用 sint64 来替代. | int64 | long |

| | | | |
|---|---|---|---|
| uint32 | 使用可变长度编码. | uint32 | int[1] |
| uint64 | 使用可变长度编码. | uint64 | long[1] |
| sint32 | 使用可变长度编码.有符号整数.它在编码负数时比 int32 更高效. | int32 | int |
| sint64 | 使用可变长度编码. 有符号整数. 它在编码负数时比 int64s 更高效. | int64 | long |
| fixed32 | 总是占用 4 个字节. 在值大于 $2^{28}$ 时比 uint32 更高效. | uint32 | int[1] |
| fixed64 | 总是占用 8 个字节. 在值大于 $2^{56}$ 时比 uint64 更高效. | uint64 | long[1] |
| sfixed32 | 总是占用 4 个字节. | int32 | int |
| sfixed64 | 总是占用 8 个字节. | int64 | long |
| bool | | bool | boolean |
| string | 必须是 UTF-8 编码的或这 7-bit 的 ASCII 文本. | string | String |
| bytes | 可以包含任何任意的字节序列. | string | ByteString |

关于在序列化消息时这些类型是怎么编码的可以阅读 Protocol Buffer 编码.

[1] 在 Java 中, 无符号 32-bit 和 64-bit 整数如果使用它们的有符号部分时, 最高位只会简单的存储在有符号位.

**可选字段和默认字段**

如上所述, 在消息描述中的元素可以被标记为 optional. 一个格式良好的消息可能或可能不包含可选元素. 当消息被解析时, 如果它不包含可选元素, 在被解析的对象中的相应的字段将会被设置为它的默认值. 默认值可以被指定作为消息的描述的一部分. 例如, 如果需要把默认值 10 赋给 SearchRequest 的 result_per_page 值, 如下:

```
optional int32 result_per_page = 3 [default = 10];
```

如果可选元素没有指定默认值, 那么将会使用一个类型指定的默认值来替代它: 对于字符串, 默认值就是空字符串. 对于布尔型, 默认值是 false. 对于数字类型, 默认值是 0. 对于枚举类型, 默认值是枚举类型定义中所列的第一个值.

## 枚举

当定义一个消息类型时,可能需要它的字段中的其中一个, 但是只有预先定义的列表中的其中一个值.you might want one of its fields to only have one of a pre-defined list of values. 例如,假如需要添加一个 corpus 字段给每个 SearchRequest , corpus 的值可以是 UNIVERSAL, WEB, IMAGES, LOCAL, NEWS, PRODUCTS 或 VIDEO. 你就可以非常简单的加入一个 enum 到消息定义中 – 一个 enum 类型的字段只能拥有指定的常量值来作为它的值 (如果想要指定一个不同的值, 解析器会认为它是一个未知字段). 在如下的实例中, 添加了一个包含所有可能值的名为 Corpus 的 enum . Corpus 类型的字段如下:

```
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];
}
```

枚举常量必须是一个 32-bit 范围的整型. 由于 enum 值使用可变整型来编码, 因此不推荐使用负数值因为它的效率很低. 可以在消息定义中定义 enum, 就像上面的示例一样, 或者是外部– 这些 enum 可以在.proto 文件中的任意消息定义中被重复使用. 还可以在消息中使用不同消息的字段的类型中声明时使用 enum 类型, 使用语法 *MessageType.EnumType*.

当在一个使用了 enum 的.proto 文件中运行 Protocol Buffer 编译器,将会生成与 Java 或 C++相对应的 enum 代码,或生成一个 Python 用来创建一组是整数值象征常量的特定 EnumDescriptor 类,在运行时生成的类.

想要了解消息中 enum 是如何工作的更多信息,请阅读对应语言的 生成代码指南 e.

## 使用其他消息类型

还能使用其他消息类型作为字段类型. 例如, 如果想在每个 SearchResponse 消息中包含 Result 消息,那么就可以在同一个.proto 文件中定义一个 Result 消息类型并且在 SearchResponse 中指定一个 Result 类型的字段:

```
message SearchResponse {
  repeated Result result = 1;
}

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

### 引入定义

在上面的例子中, Result 消息类型被定义在了同一个文件中名字为 SearchResponse – 那么如果要定义的字段类型已经在其他的.proto 文件中被定义了怎么办呢?
那么可以在其他.proto 文件中被定义的类型通过引入来实现. 如下使用了一条在文件顶部的引入语来引入来自其他 .proto 文件的定义:

```
import "myproject/other_protos.proto";
```

Protocol Buffers 编译器将会在使用 Protocol Buffers 编译器命令行中的-I/--import_path 指定的一组目录中搜寻要引入的文件. 如果没有给出, 那么它将会搜素编译器在被调用时所在目录.

### 内嵌类型

可以在其他消息类型的内部定义和使用消息类型, 在如下的示例中 - Result 消息就被定义在了 SearchResponse 消息中:

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

如果需要在他外部的父消息类型中使用这些消息类型, 可以使用 *Parent.Type*:

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

还能嵌入更深层次的消息:

```
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;
   }
 }
}
```

**组(Groups)**

**注意本特性被废弃了,并且当创建新消息类型时不能被使用-请使用嵌入消息类型代替**

组是另外一种内嵌在消息定义中的方式. 例如, 另一种在 SearchResponse 中指定含有一些 Result 的方式如下:

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

一个组就简单的在单个生命中综合了一个内嵌的消息类型和一个字段. 在代码里, 这些消息可以被认为是一个名叫 result 的拥有 Result 类型字段的消息 (为了不与前者的名字相冲突后着的名字被转换成小写的了). 因此, 本示例和上面的 SearchResponse 是一样的, 除了这个消息有个不同的包装格式.

## 更新一个消息类型

如果一个已有的消息类型不能满足你的要求时– 例如, 你希望消息格式拥有一个额外的字段– 但是你还想用来的格式来创建代码, 不用担心! 根据以下的跪着, 就能非常简单的在不用破坏已有代码的情况下更新消息类型:

- 不要改变任何已有字段数字标记.
- 添加任何新字段是最好是 optional 或者 repeated. 这意味着使用”老的”消息格式的序列化消息能被新生成的代码所解析, 并且不会丢失任何 required 元素. 为了使新代码能够与老代码生成的消息交互, 最好为这些元素设置一个敏感的默认值. 同样的, 通过新代码生成的消息能被老代码所解析: 老大类库在解析时会忽略新字段. 但是, 未知字段则不会被丢弃, 并且当消息是后来被序列化的, 未知字段将与其一起被序列化– 因此如果消息被传递到新代码中, 新字段仍然是可用的. 注意保留的未知字段目前在 Python 中是不可用的.
- 非必填字段不能被移除, 只要标签编号不在已更新的消息类型中被再次使用就可以了(这可能比重命名字段要好, 或者添加一个前缀"OBSOLETE_",以便将来.proto 文件的使用者不会意外的重复使用这些编码).
- 非必填字段与扩展 之间可以相互转换, 只要类型和编号一样就行.
- int32, uint32, int64, uint64, 和 bool 都是可兼容的– 这意味着修改字段中的这些类型其种到另一个种而不会破坏向前或向后的兼容性. 如果从包装中解析的编号不能匹配到相应的类型, 你会得到与在 C++中强制转换这个编号为类型的效果是一样的(如:如果一个 64-bit 数字被读成 int32, 那么它会被截断为 32-bit 的).
- sint32 和 sint64 是相互兼容的, 但是*不能兼容*其他整型类型.
- string 和 bytes 是可兼容的, 只要 bytes 是有效的 UTF-8 编码.
- 嵌入的消息与 bytes 兼容, 如果 bytes 包含一个已编码的消息版本.
- fixed32 与 sfixed32 兼容, 以及 fixed64 字段 sfixed64 兼容.

- 修改一个默认值在理论上是允许的，只要记住默认值永远不要发送到包装器上. 因此如果程序接收到一条消息而其中的特定字段没有设置值，那么程序将会认为它的值是程序的协议版本定义的默认值. 在发送端中定义的默认值接收端是看不到的.

## 扩展

扩展可以为第三方扩展在消息中声明一个字段编号的范围. 其他人可以在他们自己的.proto 文件中用这些数字的标签声明与你的消息类型对应的新字段.看下面的示例:

```
message Foo {
 // ...
 extensions 100 to 199;
}
```

上面所说的字段的数字范围[100,199]将作为 Foo 的扩展被保留.现在其他用户可以在自己的.proto 文件中引入你的.proto 文件来添加新字段到 Foo 中，在你指定的范围内使用标签–例如:

```
extend Foo {
 optional int32 bar = 126;
}
```

上面所描述的是 Foo 含有一个可选的 int32 字段叫做 bar.
当用户的 Foo 消息被编码时，包装格式和用户在 Foo 里定义的新字段的一样. 但是，在应用程序中访问扩展字段和访问普通字段有点不同 – 生成的数据访问类代码拥有特殊的访问器来访问扩展. 例如，以下在 C++中如何设置值的:

```
Foo foo;
foo.SetExtension(bar, 15);
```

同样的，Foo 类也定义了模板访问器
HasExtension(), ClearExtension(), GetExtension(), MutableExtension(), 和 AddExtension(). 对一个普

通字段所有语义都匹配对应的生成的访问器. 关于使用更多扩展的信息,请阅读对应编程语言的生成代码的引用文档.
注意扩展可以是任何字段类型,包括消息类型.

**内嵌扩展**

以下是在其他类型的范围内声明扩展:

```
message Baz {
 extend Foo {
   optional int32 bar = 126;
 }
 ...
}
```

这种情况下, 访问本扩展的 C++代码如下::

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

换句话说, `bar` 只在定义它的 `Baz` 的范围内有效.

得到一个结论: 声明一个 `extend` 块内嵌在一个消息类型内***并不意味着***外部类型和扩展类型有关联. 特别说明, 上面的例子中***不是说*** `Foo` 是 `Baz` 的子类. 他的意思是标记 `bar` 被声明在 `Baz` 的范围内; 它只是一个静态成员.

一个通用的模式就是在扩展的字段类型内部定义扩展– 例如, 下面是一个在 `Foo` 中的类型为 `Baz` 的扩展, 并且扩展被定义为 `Baz` 的一部分:

```
message Baz {
 extend Foo {
   optional Baz foo_ext = 127;
 }
 ...
```

```
}
```

但是, 消息类型的一个扩展定义在那个类型的内部是没有什么特别的需求的. 如下:

```
message Baz {
 ...
}


// This can even be in a different file.
extend Foo {
 optional Baz foo_baz_ext = 127;
}
```

事实上, 这个语法能避免产生混淆. 如上所说, 内嵌的语法在用户还不熟悉扩展的时候通常会和内部类的搞混.

**选择扩展编号**

确保两个用户不能使用同一个数字标签添加扩展到同一个消息类型中是非常重要的 – 如果数据错误可能会导致扩展意外的解析成一个错误的类型. 因此你需要为你的项目定义一个扩展编号约定来防止这些事的发生.

如果编号约定可能涉及非常大编号的扩展来作为标记, 那么你可以同时用 max 关键字来指定你的扩展的范围扩大到最大可能的编号:

```
message Foo {
 extensions 1000 to max;
}
```

max 为 $2^{29}$ - 1, or 536,870,911.

由于可以选择一般的标记编号, 编号公约还需要避免字段编号为 19000 到 19999 的数字 (FieldDescriptor::kFirstReservedNumber 到 FieldDescriptor::kLastReservedNumber), 他们是被 Protocol Buffers 实现所保留的.你还是可以定义一个使用这些范围内的数字的扩展, 但是 Protocol Buffers 编译器将不会你使用这些数字定义扩展的.

## 包

可以在.proto 文件中添加一个可选的 package 说明,来防止协议消息类型之间的名字冲突.

```
package foo.bar;
message Open { ... }
```

然后可以在定义消息类型的时候使用这些包说明, 如下:

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

一个包的说明影响着你选择编程语言所生成的代码:

- 在 **C++** 中,生成的类被包含在 C++的命名空间中. 例如, Open 将在命名空间 foo::bar 中.
- 在 **Java** 中, 包就是 Java 的包, 除非你显示的在.proto 文件中提供一个 option java_package 属性.
- 在 **Python** 中, 包被直接忽略, 因为 Python 模块是根据它们所在的位置在文件系统中自动被管理的.

**包和名称解析**

在 protocol buffer 中可进行类型名称解析的语言,如 C++的工作方式是: 首先最里层的范围被搜索,然后是最里层的外面一层,依次类推, 每个包都被认为是它父包"内部"的.通过在前面加一个'.' (例如, .foo.bar.Baz).

Protocol Buffer 编译器通过解析引入的.proto 文件来解析所有类型名称. 每种语言的代码生成器都知道怎么对应到它的语言的类型, 即使它有不同范围规则.

## 定义服务

如果想在 RPC 系统中使用消息类型, 可以定义在.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语言生成服务接口代码和实现. 例如,定义一个方法使用 SearchRequest 作为参数并返回 SearchResponse 的 RPC 服务, 可在.proto 文件中按照如下代码来定义:

```
service SearchService {

 rpc Search (SearchRequest) returns (SearchResponse);

}
```

Protocol Buffer 编译器会生成一个名字为 SearchService 的抽象接口和相对应的"stub"实现. 这个实现转发所有的调用到一个 RpcChannel, 它是一个, which in turn is an abstract interface that you must define yourself in terms of your own RPC system. 例如, 可以实现 RpcChannel 接口来序列化消息并通过 HTTP 协议来把它发送到服务器. 换句话说,生成的实现为基于 Protocol Buffer 的调用提供了一个类型安全的接口, 而不用去锁定任何 RPC 实现, 以下是 C++的代码示例:

```
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;
}
```

所有的服务类也都实现了 Service 接口, 它提供了一种调用特定方法的方式而不需要在编译时知道方法名字或者它的输入和输出类型. 在服务器端, 这能被用来实现能注册服务的 RPC 服务器.

```
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;
}
```

Protocol Buffers 有很多正在进行中的第三方项目来开发 RPC 实现. 想要了解这些有关项目的列表, 请参见 第三方插件 WIKI 页.

## 选项

.proto 文件中单独的声明可以被注解成一组*选项*. 选项不会改变声明的含义, 但是可能会影响在特定上下文环境下的处理方式. 可用的选项的完整列表被定义在 google/protobuf/descriptor.proto.

一些选项是文件级别的选项, 意思是应该在在最高级别范围可写, 而不是在任何消息, 枚举或者服务定义的内部. 一些选项是消息级别的选项, 意思是在消息定义内部可写.

选项也能在字段, 枚举类型, 枚举值, 服务类型, 和服务方法可写; 但是对于任何这些选项目前没有有用的存在.
下面是一些最常用的选项:

- java_package (文件选项): 生成 Java 类的包名.如果在.proto 文件中没有明确的指定 java_package 选项值, 那么默认将使用原始的包名(在.proto 文件中"package"所指定的值). 但是, 原始包不适合使用在 Java 上,因为 Java 包需要用反写的字段名作为包名.如果没有生成 Java 代码, 本选项将不会起作用.

```
option java_package = "com.example.foo";
```

- java_outer_classname (文件选项): 生成的最外层的 Java 类的类名(也是文件名). 如果没有在.proto 文件中明确的指定 java_outer_classname 的值, 那么类名将会根据.proto 文件名并按照驼峰命名法来构造类名(如: foo_bar.proto 就会变成 FooBar.java). 如果不生成 Java 代码,此选项不会生效.

```
option java_outer_classname = "Ponycopter";
```

- optimize_for (文件选项): 可以设置为 SPEED, CODE_SIZE, 或 LITE_RUNTIME. 此选项在如下几种情况下会影响 C++ 和 Java 的代码生成器(以及第三方生成器):
    - SPEED (默认值): protocol buffer 编译器将会生成代码在消息类型上进行序列化, 解析, 和执行其他常用操作. 这些代码都进行最大优化了.
    - CODE_SIZE: protocol buffer 编译器将会生成生成很少的代码 ,并且依赖共享, 反射来实现序列化, 解析, 和其他各种操作. 生成的代码比用 SPEED 属性生成的代码小得多, 而且操作也慢些. 类实现的公共接口跟在 SPEED 模式下还是一样的. 此模式适用在那些应用程序中含有很多.proto 文件但不需要他们所有的文件都瞎快.
    - LITE_RUNTIME: protocol buffer 编译器将会生成类,仅仅依靠"轻量级"运行时类库(libprotobuf-lite 替代 libprotobuf).轻量级运行时比全类库小的多(小一个数量级)但是忽略了一些诸如描述器和反射的特性. 这在一些受限制的平台如手机上比较有用.编译器仍将会生成所有方法的快速实现就像在

SPEED 模式中一样. 生成的类根据不同的语言仅实现 MessageLite 接口, 该接口只是提供了 Message 接口的方法子集.

```
option optimize_for = CODE_SIZE;
```

- cc_generic_services, java_generic_services, py_generic_services (文件选项): 不管怎样 protocol buffer 编译器都会分别生成基于在 C++,Java 和 Python 语言中的服务定义抽象的服务代码. 由于历史的原因, 这些都默认设置为 true. 但是在版本 2.3.0(2010 年 1 月)中,它考虑到为 RPC 实现提供代码生成器插件来生成代码到每个系统比依赖"抽象"的服务更合适.

```
// 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 (消息选项): 如果设置为 true, 消息会使用一个不同的二进制格式来与 Google 内部称为 MessageSet 的格式兼容. Google 以外的用户可能永远都不会用到这个选项. 此消息必须按照下面的方式准确的声明:

```
message Foo {
 option message_set_wire_format = true;
 extensions 4 to max;
}
```

- packed (字段选项): 如果在一个基于基本整型类型的可重复字段上设置为 true , 将会使用一种更加缩减的编码方式. 使用此选项是没有缺点的. 但是注意在 2.3.0 之前的版本,解析器如果接收到的不是打包过的数据将会忽略它. 因此它不会把一个已经存在的字段修改为一个打包的格式,而不破坏包装兼容性. 在 2.3.0 中和以后的版本这个改变是安全的, 解析器会接受两种格式的打包的字段, 但是注意需要解决在旧的程序中如果使用了 protobuf 版本的问题.

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

- deprecated (字段选项):如果设置为 true, 表示字段是废弃的并不会在新代码中被使用. 目前此选项没有任何实际的影响, 但是在将来的语言指定的代码生成器可能会在字段的访问器上生成一个废弃的注释, 它会在尝试使用该字段的代码被编译时候释放一个警告.

```
optional int32 old_field = 6 [deprecated=true];
```

**自定义选项**

Protocol Buffers 还允许用户定义自己的选项. 注意这是一个大多数的人都不会用到的**高级特性**.因此在 google/protobuf/descriptor.proto (如 FileOptions 或 FieldOptions)定义的消息中,定义选项就是简单的扩展这些消息, 例如:

```
import "google/protobuf/descriptor.proto";


extend google.protobuf.MessageOptions {
 optional string my_option = 51234;
}


message MyMessage {
 option (my_option) = "Hello world!";
}
```

在这里通过扩展 MessageOptions 定义了一个新的消息级别选项. 当使用这个选项时, 选项的名字必须包含在一个括号中来表示它是一个扩展. 在 C++中可以通过如下方式读取 my_option 的值:

```
string value = MyMessage::descriptor()->options().GetExtension(my_option);
```

在这里,MyMessage::descriptor()->options() 返回 MessageOptions 协议消息给 MyMessage. 从中读取自定义选项的值就跟读取其他扩展一样. 类似的, 在 Java 中可以这样写:

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

当写此文的时候(版本 2.0.3), 自定义选项在 Python 语言中是不可访问的.
自定义选项可以以任何一种形式被定义在 Protocol Buffers 语言中. 如下是几个例子:

```
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";
 }
}
```

注意, 如果在包中使用自定义选项而不是使用本来就定义好的选项, 则必须使用选项的前缀来作为包名, 就像作为类型名称一样. 例如::

```
// 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: 因为自定义选项都是扩展, 他们必须像其他字段或扩展一样指定字段的编号. 在上面的例子中, 使用的字段编号是 50000-99999 之间的. 这个范围被单独的组织作为内部使用了, 因此你可以在非公开的应用程序中自由的使用这个范围的数字. 如果想要在公共的应用程序中使用自定义选项, 那么就必须保证你的字段编号是全球唯一的.如果想要获取全球唯一的字段编号,可以给 kenton@google.com 发邮件.简单的告诉他需要多少个字段编号;而不需要解释你想做什么用, 然后他会给你发回一个可以使用的字段编号范围.

## 生成类

为了生成 Java, Python,或者 C++代码, 就必须用在.proto 文件中定义的消息类型, 还得在.proto 上运行 Protocol Buffer 的编译器 protoc. 如果没有安装编译器, 下载安装包并按照 README 中的指示安装.

Protocol Buffer 编译器按照如下方式调用:

```
protoc --proto_path=IMPORT_PATH --cpp_out=DST_DIR --java_out=DST_DIR --python_out=DST_DIR
path/to/file.proto
```

- IMPORT_PATH 在解析 import 指令时,指定一个搜寻.proto 文件的目录. 如果没有指定那么将使用当前运行目录. 多个引入包可以通过使用多个--proto_path 选项来指定目录; 编译器将会按照指定的顺序来搜索. -I=*IMPORT_PATH* 的缩写格式是--proto_path.
- 还能提供一个或多个*输出目录*:
  - --cpp_out 在 DST_DIR 中生成 C++代码, 更多请参见 C++ 代码生成引用.
  - --java_out 在 DST_DIR 中生成 Java 代码, 更多请参见 Java 代码生成引用.
  - --python_out 在 DST_DIR 中生成 Python 代码. 更多请参见 Python 代码生成引用.

  为了方便, 如果 DST_DIR 是以.zip 或 .jar 结尾的, 编译器将会按照给定的名字输出为一个单个的 ZIP 格式的归档文件. .jar 也会输出根据 JAR 规范需要的 manifest 文件. 注意如果输出的归档已经存在, 它将会被覆盖; 编译器不是聪明到添加文件到一个已经存在的归档中去.
- 必须提供一个或多个 .proto 文件作为输入. 多个.proto 文件可以一次被指定.虽然文件都是按照与当前目录关联关系来命名的, 但是每个文件都必须在 IMPORT_PATHS 中的一个来指定,以便于编译器能够检测出它的规范的名字.