Protobuf3 语言指南

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：

```
syntax = "proto3";



message SearchRequest {

  string query = 1;

  int32 page_number = 2;

  int32 result_per_page = 3;

}
```

- 1
- 2
- 3
- 4
- 5
- 6
- 7

- 文件的第一行指定了你正在使用 proto3 语法：如果你没有指定这个，编译器会使用 proto2。这个指定语法行必须是文件的非空非注释的第一个行。
- 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。

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）或其他消息类型。

分配标识号

正如你所见，在消息定义中，每个字段都有唯一的一个数字标识符。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 2^29 - 1, or 536,870,911。不可以使用其中的[19000－19999]（ (从 FieldDescriptor::kFirstReservedNumber 到 FieldDescriptor::kLastReservedNumber)）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。同样你也不能使用早期保留的标识号。

指定字段规则

所指定的消息字段修饰符必须是如下之一：

- singular：一个格式良好的消息应该有 0 个或者 1 个这种字段（但是不能超过 1 个）。
- 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。

  在 proto3 中，repeated 的标量域默认情况虾使用 packed。

  你可以了解更多的 pakced 属性在 Protocol Buffer 编码

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 {

  string query = 1;

  int32 page_number = 2;

  int32 result_per_page = 3;

}
```

```
message SearchResponse {
 ...
}
```

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 {
  string query = 1;
  int32 page_number = 2;  // Which page number do we want?
  int32 result_per_page = 3;  // Number of results to return per page.
}
```

保留标识符（Reserved）

如果你通过删除或者注释所有域，以后的用户可以重用标识号当你重新更新类型的时候。如果你使用旧版本加载相同的.proto 文件这会导致严重的问题，包括数据损坏、隐私错误等等。现在有一种确保不会发生这种情况的方法就是指定保留标识符（and/o

r names, which can also cause issues for JSON serialization 不明白什么意思），protocol buffer 的编译器会警告未来尝试使用这些域标识符的用户。

```
message Foo {

  reserved 2, 15, 9 to 11;

  reserved "foo", "bar";

}
```

注：不要在同一行 reserved 声明中同时声明域名字和标识号

从.proto 文件生成了什么？

当用 protocol buffer 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。

- 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。
- 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）。
- 对 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。
- 对 go 来说，编译器会位每个消息类型生成了一个.pd.go 文件。
- 对于 Ruby 来说，编译器会为每个消息类型生成了一个.rb 文件。
- javaNano 来说，编译器输出类似域 java 但是没有 Builder 类
- 对于 Objective-C 来说，编译器会为每个消息类型生成了一个 pbobjc.h 文件和 pbobjcm 文件，.proto 文件中的每一个消息有一个对应的类。
- 对于 C#来说，编译器会为每个消息类型生成了一个.cs 文件，.proto 文件中的每一个消息有一个对应的类。

你可以从如下的文档链接中获取每种语言更多 API(proto3 版本的内容很快就公布)。API Reference

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 Type	Notes	C++ Type	Java Type	Python Type[2]	Go Type	Ruby Type	C#
double		double	double	float	float64	Float	do
float		float	float	float	float32	Float	flo
int32	使用变长编码，对于负值的效率很低，如果你的域有可能有负值，请使用 sint64 替代	int32	int	int	int32	Fixnum 或者 Bignum（根据需要）	int
uint32	使用变长编码	uint32	int	int/long	uint32	Fixnum 或者 Bignum（根据需要）	ui
uint64	使用变长编码	uint64	long	int/long	uint64	Bignum	ul
sint32	使用变长编码，这些编码在负值时比 int32 高效的多	int32	int	int	int32	Fixnum 或者 Bignum（根据需要）	int
sint64	使用变长编码，有符号的整型值。编码时比通常的 int64 高效。	int64	long	int/long	int64	Bignum	lo
fixed32	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。	uint32	int	int	uint32	Fixnum 或者 Bignum（根据需要）	ui
fixed64	总是 8 个字节，如果数值总是比总是	uint64	long	int/long	uint64	Bignum	ul

.proto Type	Notes	C++ Type	Java Type	Python Type[2]	Go Type	Ruby Type	C#
	比 256 大的话，这个类型会比 uint64 高效。						
sfixed32	总是 4 个字节	int32	int	int	int32	Fixnum 或者 Bignum（根据需要）	int
sfixed64	总是 8 个字节	int64	long	int/long	int64	Bignum	lon
bool		bool	boolean	bool	bool	TrueClass/FalseClass	bo
string	一个字符串必须是 UTF-8 编码或者 7-bit ASCII 编码的文本。	string	String	str/unicode	string	String (UTF-8)	str
bytes	可能包含任意顺序的字节数据。	string	ByteString	str	[]byte	String (ASCII-8BIT)	By

你可以在文章 Protocol Buffer 编码中，找到更多“序列化消息时各种类型如何编码”的信息。

1. 在 java 中，无符号 32 位和 64 位整型被表示成他们的整型对应形似，最高位被储存在标志位中。
2. 对于所有的情况，设定值会执行类型检查以确保此值是有效。
3. 64 位或者无符号 32 位整型在解码时被表示成为 ilong，但是在设置时可以使用 int 型值设定，在所有的情况下，值必须符合其设置其类型的要求。
4. python 中 string 被表示成在解码时表示成 unicode。但是一个 ASCIIstring 可以被表示成 str 类型。
5. Integer 在 64 位的机器上使用，string 在 32 位机器上使用

默认值

当一个消息被解析的时候，如果被编码的信息不包含一个特定的 singular 元素，被解析的对象锁对应的域被设置位一个默认值，对于不同类型指定如下：

- 对于 strings，默认是一个空 string
- 对于 bytes，默认是一个空的 bytes

- 对于 bools，默认是 false
- 对于数值类型，默认是 0
- 对于枚举，默认是第一个定义的枚举值，必须为 0;
- 对于消息类型（message），域没有被设置，确切的消息是根据语言确定的，详见 generated code guide

  对于可重复域的默认值是空（通常情况下是对应语言中空列表）。

  注：对于标量消息域，一旦消息被解析，就无法判断域释放被设置为默认值（例如，例如 boolean 值是否被设置为 false）还是根本没有被设置。你应该在定义你的消息类型时非常注意。例如，比如你不应该定义 boolean 的默认值 false 作为任何行为的触发方式。也应该注意如果一个标量消息域被设置为标志位，这个值不应该被序列化传输。

  查看 generated code guide 选择你的语言的默认值的工作细节。

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）并且为每个可能的值定义一个常量就可以了。

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 {

  string query = 1;

  int32 page_number = 2;

  int32 result_per_page = 3;

  enum Corpus {

    UNIVERSAL = 0;

    WEB = 1;
```

```
    IMAGES = 2;

    LOCAL = 3;

    NEWS = 4;

    PRODUCTS = 5;

    VIDEO = 6;

  }

  Corpus corpus = 4;

}
```

如你所见，Corpus 枚举的第一个常量映射为 0：每个枚举类型必须将其第一个类型映射为 0，这是因为：

- 必须有有一个 0 值，我们可以用这个 0 值作为默认值。
- 这个零值必须为第一个元素，为了兼容 proto2 语义，枚举类的第一个值总是默认值。

  你可以通过将不同的枚举常量指定位相同的值。如果这样做你需要将 allow_alias 设定位 true，否则编译器会在别名的地方产生一个错误信息。

```
enum EnumAllowingAlias {
```

```
  option allow_alias = true;

  UNKNOWN = 0;

  STARTED = 1;

  RUNNING = 1;

}

enum EnumNotAllowingAlias {

  UNKNOWN = 0;

  STARTED = 1;

  // RUNNING = 1;  // Uncommenting this line will cause a compile error inside
 Google and a warning message outside.

}
```

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）。

在反序列化的过程中，无法识别的枚举值会被保存在消息中，虽然这种表示方式需要依据所使用语言而定。在那些支持开放枚举类型超出指定范围之外的语言中（例如 C++和 Go），为识别的值会被表示成所支持的整型。在使用封闭枚举类型的语言中（Java），使用枚举中的一个类型来表示未识别的值，并且可以使用所支持整型来访问。在其他情况下，如果解析的消息被序列号，未识别的值将保持原样。

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 generated code guide

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 results = 1;

}



message Result {

  string url = 1;

  string title = 2;

  repeated string snippets = 3;

}
```

```
5
6
7
8
9
```

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，如：

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

```
1
```

默认情况下你只能使用直接导入的.proto 文件中的定义. 然而， 有时候你需要移动一个.proto 文件到一个新的位置， 可以不直接移动.proto 文件， 只需放入一个伪 .proto 文件在老的位置， 然后使用 import public 转向新的位置。import public 依赖性会通过任意导入包含 import public 声明的 proto 文件传递。例如：

```
// 这是新的 proto

// All definitions are moved here
```

```
1
2
```

```
// 这是久的 proto

// 这是所有客户端正在导入的包

import public "new.proto";

import "other.proto";
```

```
1
2
3
4
```

```
// 客户端 proto

import "old.proto";
```

```
// 现在你可以使用新久两种包的 proto 定义了。
```

通过在编译器命令行参数中使用`-I/--proto_path`protocal 编译器会在指定目录搜索要导入的文件。如果没有给出标志，编译器会搜索编译命令被调用的目录。通常你只要指定 proto_path 标志为你的工程根目录就好。并且指定好导入的正确名称就好。

使用 proto2 消息类型

在你的 proto3 消息中导入 proto2 的消息类型也是可以的，反之亦然，然后 proto2 枚举不可以直接在 proto3 的标识符中使用（如果仅仅在 proto2 消息中使用是可以的）。

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 {

    string url = 1;

    string title = 2;

    repeated string snippets = 3;

  }

  repeated Result results = 1;

}
```

```
7
8
```

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 {

  SearchResponse.Result result = 1;

}
```

```
1
2
3
```

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

      int64 ival = 1;

      bool  booly = 2;

    }

  }

  message MiddleBB {  // Level 1

    message Inner {   // Level 2

      int32 ival = 1;

      bool  booly = 2;

    }

  }
```

```
}
```

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。

- 不要更改任何已有的字段的数值标识。
- 如果你增加新的字段，使用旧格式的字段仍然可以被你新产生的代码所解析。你应该记住这些元素的默认值这样你的新代码就可以以适当的方式和旧代码产生的数据交互。相似的，通过新代码产生的消息也可以被旧代码解析：只不过新的字段会被忽视掉。注意，未被识别的字段会在反序列化的过程中丢弃掉，所以如果消息再被传递给新的代码，新的字段依然是不可用的（这和 proto2 中的行为是不同的，在 proto2 中未定义的域依然会随着消息被序列化）
- 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）。
- 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）。
- sint32 和 sint64 是互相兼容的，但是它们与其他整数类型不兼容。
- string 和 bytes 是兼容的——只要 bytes 是有效的 UTF-8 编码。
- 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。
- fixed32 与 sfixed32 是兼容的，fixed64 与 sfixed64 是兼容的。
- 枚举类型与 int32，uint32，int64 和 uint64 相兼容（注意如果值不相兼容则会被截断），然而在客户端反序列化之后他们可能会有不同的处理方式，例如，未识别的 proto3 枚举类型会被保留在消息中，但是他的表示方式会依照语言而定。int 类型的字段总会保留他们的

Any

Any 类型消息允许你在没有指定他们的.proto 定义的情况下使用消息作为一个嵌套类型。一个 Any 类型包括一个可以被序列化 bytes 类型的任意消息，以及一个 URL 作为一个全局标识符和解析消息类型。为了使用 Any 类型，你需要导入 `import google/protobuf/any.proto`

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



message ErrorStatus {

  string message = 1;

  repeated google.protobuf.Any details = 2;

}
```

对于给定的消息类型的默认类型 URL 是 `type.googleapis.com/packagename.messagename`。

不同语言的实现会支持动态库以线程安全的方式去帮助封装或者解封装 Any 值。例如在 java 中，Any 类型会有特殊的 `pack()`和 `unpack()`访问器，在 C++中会有 `PackFrom()`和 `UnpackTo()`方法。

```
// Storing an arbitrary message type in Any.

NetworkErrorDetails details = ...;

ErrorStatus status;

status.add_details()->PackFrom(details);
```

```
// Reading an arbitrary message from Any.
ErrorStatus status = ...;
for (const Any& detail : status.details()) {
  if (detail.Is<NetworkErrorDetails>()) {
    NetworkErrorDetails network_error;
    detail.UnpackTo(&network_error);
    ... processing network_error ...
  }
}
```

目前，用于 Any 类型的动态库仍在开发之中

如果你已经很熟悉 proto2 语法，使用 Any 替换拓展

Oneof

如果你的消息中有很多可选字段， 并且同时至多一个字段会被设置， 你可以加强这个行为，使用 oneof 特性节省内存.

Oneof 字段就像可选字段， 除了它们会共享内存， 至多一个字段会被设置。 设置其中一个字段会清除其它字段。 你可以使用 `case()`或者 `WhichOneof()` 方法检查哪个 oneof 字段被设置， 看你使用什么语言了.

使用 Oneof

为了在.proto 定义 Oneof 字段， 你需要在名字前面加上 oneof 关键字, 比如下面例子的 test_oneof:

```
message SampleMessage {

  oneof test_oneof {

    string name = 4;

    SubMessage sub_message = 9;

  }

}
```

然后你可以增加 oneof 字段到 oneof 定义中. 你可以增加任意类型的字段, 但是不能使用 repeated 关键字.

在产生的代码中, oneof 字段拥有同样的 getters 和 setters， 就像正常的可选字段一样. 也有一个特殊的方法来检查到底那个字段被设置. 你可以在相应的语言 API 指南中找到 oneof API 介绍.

Oneof 特性

- 设置 oneof 会自动清楚其它 oneof 字段的值. 所以设置多次后，只有最后一次设置的字段有值.

```
SampleMessage message;

message.set_name("name");

CHECK(message.has_name());

message.mutable_sub_message();   // Will clear name field.

CHECK(!message.has_name());
```

- 1
- 2
- 3
- 4
- 5

- 如果解析器遇到同一个 oneof 中有多个成员，只有最会一个会被解析成消息。
- oneof 不支持 `repeated`.
- 反射 API 对 oneof 字段有效.
- 如果使用 C++,需确保代码不会导致内存泄漏. 下面的代码会崩溃， 因为 `sub_message` 已经通过 `set_name()`删除了

```
SampleMessage message;

SubMessage* sub_message = message.mutable_sub_message();

message.set_name("name");      // Will delete sub_message

sub_message->set_...            // Crashes here
```

- 1
- 2
- 3
- 4

- 在 C++中，如果你使用 `Swap()`两个 oneof 消息，每个消息，两个消息将拥有对方的值，例如在下面的例子中，`msg1` 会拥有 `sub_message` 并且 `msg2` 会有 `name`。

```
SampleMessage msg1;

msg1.set_name("name");
```

```
SampleMessage msg2;

msg2.mutable_sub_message();

msg1.swap(&msg2);

CHECK(msg1.has_sub_message());

CHECK(msg2.has_name());
```

向后兼容性问题

当增加或者删除 oneof 字段时一定要小心. 如果检查 oneof 的值返回 `None/NOT_SET`, 它意味着 oneof 字段没有被赋值或者在一个不同的版本中赋值了。 你不会知道是哪种情况，因为没有办法判断如果未识别的字段是一个 oneof 字段。

Tage 重用问题：

- 将字段移入或移除 oneof：在消息被序列号或者解析后，你也许会失去一些信息（有些字段也许会被清除）
- 删除一个字段或者加入一个字段：在消息被序列号或者解析后，这也许会清除你现在设置的 oneof 字段
- 分离或者融合 oneof：行为与移动常规字段相似。

Map（映射）

如果你希望创建一个关联映射，protocol buffer 提供了一种快捷的语法：

```
map<key_type, value_type> map_field = N;
```

其中 `key_type` 可以是任意 Integer 或者 string 类型（所以，除了 floating 和 bytes 的任意标量类型都是可以的）`value_type` 可以是任意类型。

例如，如果你希望创建一个 project 的映射，每个 `Projecct` 使用一个 string 作为 key，你可以像下面这样定义：

```
map<string, Project> projects = 3;
```

- Map 的字段可以是 repeated。
- 序列化后的顺序和 map 迭代器的顺序是不确定的，所以你不要期望以固定顺序处理 Map
- 当为.proto 文件产生生成文本格式的时候，map 会按照 key 的顺序排序，数值化的 key 会按照数值排序。
- 从序列化中解析或者融合时，如果有重复的 key 则后一个 key 不会被使用，当从文本格式中解析 map 时，如果存在重复的 key。

生成 map 的 API 现在对于所有 proto3 支持的语言都可用了，你可以从 API 指南找到更多信息。

向后兼容性问题

map 语法序列化后等同于如下内容，因此即使是不支持 map 语法的 protocol buffer 实现也是可以处理你的数据的：

```
message MapFieldEntry {
  key_type key = 1;
  value_type value = 2;
}

repeated MapFieldEntry map_field = N;
```

包

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。
- 对于 Go，包可以被用做 Go 包名称，除非你显式的提供一个 `option go_package` 在你的.proto 文件中。
- 对于 Ruby，生成的类可以被包装在内置的 Ruby 名称空间中，转换成 Ruby 所需的大小写样式（首字母大写；如果第一个符号不是一个字母，则使用 PB_前缀），例如 `Open` 会在 `Foo::Bar` 名称空间中。
- 对于 javaNano 包会使用 Java 包，除非你在你的文件中显式的提供一个 `option java_package`。

- 对于 C#包可以转换为 `PascalCase` 后作为名称空间，除非你在你的文件中显式的提供一个 `option csharp_namespace`，例如，`Open` 会在 `Foo.Bar` 名称空间中

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。

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);

}
```

最直观的使用 protocol buffer 的 RPC 系统是 gRPC 一个由谷歌开发的语言和平台中的开源的 PRC 系统，gRPC 在使用 protocl buffer 时非常有效，如果使用特殊的 protocol buffer 插件可以直接为您从.proto 文件中产生相关的 RPC 代码。

如果你不想使用 gRPC，也可以使用 protocol buffer 用于自己的 RPC 实现，你可以从 proto2 语言指南中找到更多信息

还有一些第三方开发的 PRC 实现使用 Protocol Buffer。参考第三方插件 wiki 查看这些实现的列表。

JSON 映射

Proto3 支持 JSON 的编码规范，使他更容易在不同系统之间共享数据，在下表中逐个描述类型。

如果 JSON 编码的数据丢失或者其本身就是 `null`，这个数据会在解析成 protocol buffer 的时候被表示成默认值。如果一个字段在 protocol buffer 中表示为默认值，体会在转化成 JSON 的时候编码的时候忽略掉以节省空间。具体实现可以提供在 JSON 编码中可选的默认值。

proto3	JSON	JSON 示例	注意
message	object	{“fBar”: v, “g”: null, …}	产生 JSON 对象，消息字段名可以被映射成 low 且成为 JSON 对象键，null 被接受并成为对应字
enum	string	“FOO_BAR”	枚举值的名字在 proto 文件中被指定
map	object	{“k”: v, …}	所有的键都被转换成 string
repeated V	array	[v, …]	null 被视为空列表
bool	true, false	true, false	
string	string	“Hello World!”	
bytes	base64 string	“YWJjMTIzIT8kKiYoKSctPUB+”	
int32, fixed32, uint32	number	1, -10, 0	JSON 值会是一个十进制数，数值型或者 string
int64, fixed64, uint64	string	“1”, “-10”	JSON 值会是一个十进制数，数值型或者 string
float, double	number	1.1, -10.0, 0, “NaN”, “Infinity”	JSON 值会是一个数字或者一个指定的字符串如 Infinity”，数值型或者字符串都是可接受的，指数

proto3	JSON	JSON 示例	注意
Any	object	{“@type”: “url”, “f”: v, … }	如果一个 Any 保留一个特上述的 JSON 映射，则式：`{"@type": xxx, "value": yyy}`否则，该 JSON 对象，`@type` 字段会被插入所指定的确定
Timestamp	string	“1972-01-01T10:00:20.021Z”	使用 RFC 339，其中生成的输出将始终是 Z-归一3，6 或者 9 位小数
Duration	string	“1.000340012s”, “1s”	生成的输出总是 0，3，6 或者 9 位小数，具体依受所有可以转换为纳秒级的精度
Struct	object	{ … }	任意的 JSON 对象，见 struct.proto
Wrapper types	various types	2, “2”, “foo”, true, “true”, null, 0, …	包装器在 JSON 中的表示方式类似于基本类型，转换的过程中保留 null
FieldMask	string	“f.fooBar,h”	见 fieldmask.proto
ListValue	array	[foo, bar, …]	
Value	value		任意 JSON 值
NullValue	null		JSON null

选项

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：

- `java_package` (文件选项) :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";
```

- java_outer_classname (文件选项): 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";
```

- optimize_for(文件选项): 可以被设置为 SPEED, CODE_SIZE,或者 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: protocol buffer 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 libpr otobuf-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 Messa ger 接口的一个子集。

```
option optimize_for = CODE_SIZE;
```

- cc_enable_arenas(文件选项):对于 C++产生的代码启用 arena allocation
- objc_class_prefix(文件选项):设置 Objective-C 类的前缀，添加到所有 Objective-C 从此.p roto 文件产生的类和枚举类型。没有默认值，所使用的前缀应该是苹果推荐的 3-5 个大写字符，注意 2 个字节的前缀是苹果所保留的。
- deprecated(字段选项):如果设置为 true 则表示该字段已经被废弃，并且不应该在新的代码中使用。在大多数语言中没有实际的意义。在 java 中，这回变成@Deprecated 注释，在未来，其他语言的代码生成器也许会在字标识符中产生废弃注释，废弃注释会在编译器尝试使用该字段时发出警告。如果字段没有被使用你也不希望有新用户使用它，尝试使用保留语句替换字段声明。

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

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。如果你的确希望创建自己的选项，请参看 Proto2 Language Guide。注意创建自定义选项使用了拓展，拓展只在 proto3 中可用。

生成访问类

可以通过定义好的.proto 文件来生成 Java,Python,C++, Ruby, JavaNano, Objective-C, 或者 C# 代码，需要基于.proto 文件运行 protocol buffer 编译器 protoc。如果你没有安装编译器，下载安装包并遵照 README 安装。对于 Go,你还需要安装一个特殊的代码生成器插件。你可以通过 GitHub 上的 protobuf 库找到安装过程

通过如下方式调用 protocol 编译器：

```
protoc --proto_path=IMPORT_PATH --cpp_out=DST_DIR --java_out=DST_DIR --python_
out=DST_DIR --go_out=DST_DIR --ruby_out=DST_DIR --javanano_out=DST_DIR --objc_
out=DST_DIR --csharp_out=DST_DIR path/to/file.proto
```

- 1

- `IMPORT_PATH` 声明了一个.proto 文件所在的解析 import 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多次调用`--proto_path`，它们将会顺序的被访问并执行导入。`-I=IMPORT_PATH` 是`--proto_path` 的简化形式。
- 当然也可以提供一个或多个输出路径：
    - `--cpp_out` 在目标目录 DST_DIR 中产生 C++代码，可以在 C++代码生成参考中查看更多。
    - `--java_out` 在目标目录 DST_DIR 中产生 Java 代码，可以在 Java 代码生成参考中查看更多。
    - `--python_out` 在目标目录 DST_DIR 中产生 Python 代码，可以在 Python 代码生成参考中查看更多。
    - `--go_out` 在目标目录 DST_DIR 中产生 Go 代码，可以在 GO 代码生成参考中查看更多。
    - `--ruby_out` 在目标目录 DST_DIR 中产生 Go 代码，参考正在制作中。
    - `--javanano_out` 在目标目录 DST_DIR 中生成 JavaNano，JavaNano 代码生成器有一系列的选项用于定制自定义生成器的输出：你可以通过生成器的 README 查找更多信息，JavaNano 参考正在制作中。
    - `--objc_out` 在目标目录 DST_DIR 中产生 Object 代码，可以在 Objective-C 代码生成参考中查看更多。
    - `--csharp_out` 在目标目录 DST_DIR 中产生 Object 代码，可以在 C#代码生成参考中查看更多。

- `--php_out` 在目标目录 DST_DIR 中产生 Object 代码，可以在 PHP 代码生成参考中查看更多。

作为一个方便的拓展，如果 DST_DIR 以.zip 或者.jar 结尾，编译器会将输出写到一个 ZIP 格式文件或者符合 JAR 标准的.jar 文件中。注意如果输出已经存在则会被覆盖，编译器还没有智能到可以追加文件。
- 你必须提议一个或多个.proto 文件作为输入，多个.proto 文件可以只指定一次。虽然文件路径是相对于当前目录的，每个文件必须位于其 IMPORT_PATH 下，以便每个文件可以确定其规范的名称。

-