定义一个消息类型：

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;
}
```

分配标识号

正如上述文件格式中的1，2，3，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。

指定字段规则：

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；

optional：消息格式中该字段可以有0个或1个值（不超过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。表示该值可以重复，相当于java中的List。

一个.proto文件中可以有多个消息类型：

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;// 最终返回的页数
  optional int32 result_per_page = 3 [default = 10];//为optional的元素指定默认值
}

message SearchResponse {
```

```
 …
}
```

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 = UNIVERSAL];
}
```

导入定义（使用其他.proto文件中定义的消息类型）

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";
```

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。

嵌套类型：

你可以在其他消息类型中定义、使用消息类型，

```
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;
}
```

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。更新消息而不破坏已有代码必须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（**分配标识号**）
- 所添加的任何字段都必须是optional或repeated的。
- 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）。
- 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。
- 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，类似与强制转换。
- 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。

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。

```
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。

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

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。

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;
}
```

定义服务：

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;
}
```