Protocol Buffer Basics: Go

This tutorial provides a basic Go programmer's introduction to working with protocol buffers, using the proto3 (https://developers.google.com/protocol-buffers/docs/proto3) version of the protocol buffers language. By walking through creating a simple example application, it shows you how to

- Define message formats in a `.proto` file.
- Use the protocol buffer compiler.
- Use the Go protocol buffer API to write and read messages.

This isn't a comprehensive guide to using protocol buffers in Go. For more detailed reference information, see the Protocol Buffer Language Guide (https://developers.google.com/protocol-buffers/docs/proto3), the Go API Reference (https://godoc.org/github.com/golang/protobuf/proto), the Go Generated Code Guide (https://developers.google.com/protocol-buffers/docs/reference/go-generated), and the Encoding Reference (https://developers.google.com/protocol-buffers/docs/encoding).

Why use protocol buffers?

The example we're going to use is a very simple "address book" application that can read and write people's contact details to and from a file. Each person in the address book has a name, an ID, an email address, and a contact phone number.

How do you serialize and retrieve structured data like this? There are a few ways to solve this problem:

- Use gobs (//golang.org/pkg/encoding/gob/) to serialize Go data structures. This is a good solution in a Go-specific environment, but it doesn't work well if you need to share data with applications written for other platforms.

- You can invent an ad-hoc way to encode the data items into a single string – such as encoding 4 ints as "12:3:-23:67". This is a simple and flexible approach, although it does require writing one-off encoding and parsing code, and the parsing imposes a small run-time cost. This works best for encoding very simple data.
- Serialize the data to XML. This approach can be very attractive since XML is (sort of) human readable and there are binding libraries for lots of languages. This can be a good choice if you want to share data with other applications/projects. However, XML is notoriously space intensive, and encoding/decoding it can impose a huge performance penalty on applications. Also, navigating an XML DOM tree is considerably more complicated than navigating simple fields in a class normally would be.

Protocol buffers are the flexible, efficient, automated solution to solve exactly this problem. With protocol buffers, you write a `.proto` description of the data structure you wish to store. From that, the protocol buffer compiler creates a class that implements automatic encoding and parsing of the protocol buffer data with an efficient binary format. The generated class provides getters and setters for the fields that make up a protocol buffer and takes care of the details of reading and writing the protocol buffer as a unit. Importantly, the protocol buffer format supports the idea of extending the format over time in such a way that the code can still read data encoded with the old format.

Where to find the example code

Our example is a set of command-line applications for managing an address book data file, encoded using protocol buffers. The command `add_person_go` adds a new entry to the data file. The command `list_people_go` parses the data file and prints the data to the console.

You can find the complete example in the examples directory (https://github.com/google/protobuf/tree/master/examples) of the GitHub repository.

Defining your protocol format

To create your address book application, you'll need to start with a `.proto` file. The definitions in a `.proto` file are simple: you add a *message* for each data structure you want to serialize, then specify a name and a type for each field in the message. In our example, the `.proto` file that defines the messages is **`addressbook.proto`** (https://github.com/google/protobuf/blob/master/examples/addressbook.proto).

The `.proto` file starts with a package declaration, which helps to prevent naming conflicts between different projects.

```
syntax = "proto3";
package tutorial;
```

In Go, the `package` name is used as the Go package, unless you have specified a `go_package`. Even if you do provide a `go_package`, you should still define a normal `package` as well to avoid name collisions in the Protocol Buffers name space as well as in non-Go languages.

Next, you have your message definitions. A message is just an aggregate containing a set of typed fields. Many standard simple data types are available as field types, including `bool`, `int32`, `float`, `double`, and `string`. You can also add further structure to your messages by using other message types as field types.

```
message Person {
  string name = 1;
  int32 id = 2;  // Unique ID number for this person.
  string email = 3;

  enum PhoneType {
    MOBILE = 0;
    HOME = 1;
    WORK = 2;
  }

  message PhoneNumber {
    string number = 1;
    PhoneType type = 2;
  }

  repeated PhoneNumber phones = 4;
}

// Our address book file is just one of these.
message AddressBook {
  repeated Person people = 1;
}
```

In the above example, the `Person` message contains `PhoneNumber` messages, while the `AddressBook` message contains `Person` messages. You can even define message types nested inside other messages – as you can see, the `PhoneNumber` type is defined inside `Person`. You can also define `enum` types if you want one of your fields to have one of a predefined list of values – here you want to specify that a phone number can be one of `MOBILE`, `HOME`, or `WORK`.

The " = 1", " = 2" markers on each element identify the unique "tag" that field uses in the binary encoding. Tag numbers 1-15 require one less byte to encode than higher numbers, so as an optimization you can decide to use those tags for the commonly used or repeated elements, leaving tags 16 and higher for less-commonly used optional elements. Each element in a repeated field requires re-encoding the tag number, so repeated fields are particularly good candidates for this optimization.

If a field value isn't set, a default value (https://developers.google.com/protocol-buffers/docs/proto3#default) is used: zero for numeric types, the empty string for strings, false for bools. For embedded messages, the default value is always the "default instance" or "prototype" of the message, which has none of its fields set. Calling the accessor to get the value of a field which has not been explicitly set always returns that field's default value.

If a field is `repeated`, the field may be repeated any number of times (including zero). The order of the repeated values will be preserved in the protocol buffer. Think of repeated fields as dynamically sized arrays.

You'll find a complete guide to writing `.proto` files – including all the possible field types – in the Protocol Buffer Language Guide (https://developers.google.com/protocol-buffers/docs/proto3). Don't go looking for facilities similar to class inheritance, though – protocol buffers don't do that.

Compiling your protocol buffers

Now that you have a `.proto`, the next thing you need to do is generate the classes you'll need to read and write `AddressBook` (and hence `Person` and `PhoneNumber`) messages. To do this, you need to run the protocol buffer compiler `protoc` on your `.proto`:

1. If you haven't installed the compiler, download the package (https://developers.google.com/protocol-buffers/docs/downloads.html) and follow the instructions in the README.
2. Run the following command to install the Go protocol buffers plugin:

   ```
   go get -u github.com/golang/protobuf/protoc-gen-go
   ```

   The compiler plugin `protoc-gen-go` will be installed in `$GOBIN`, defaulting to `$GOPATH/bin`. It must be in your `$PATH` for the protocol compiler `protoc` to find it.
3. Now run the compiler, specifying the source directory (where your application's source code lives – the current directory is used if you don't provide a value), the destination directory (where you want the generated code to go; often the same as `$SRC_DIR`), and the path to your `.proto`. In this case, you...:

   ```
   protoc -I=$SRC_DIR --go_out=$DST_DIR $SRC_DIR/addressbook.proto
   ```

   Because you want Go classes, you use the `--go_out` option – similar options are provided for other supported languages.

This generates `addressbook.pb.go` in your specified destination directory.

The Protocol Buffer API

Generating `addressbook.pb.go` gives you the following useful types:

- An `AddressBook` structure with a `People` field.
- A `Person` structure with fields for `Name`, `Id`, `Email` and `Phones`.
- A `Person_PhoneNumber` structure, with fields for `Number` and `Type`.
- The type `Person_PhoneType` and a value defined for each value in the `Person.PhoneType` enum.

You can read more about the details of exactly what's generated in the Go Generated Code guide (https://developers.google.com/protocol-buffers/docs/reference/go-generated), but for the most part you can treat these as perfectly ordinary Go types.

Here's an example from the `list_people` command's unit tests (https://github.com/google/protobuf/blob/master/examples/list_people_test.go) of how you might create an instance of Person:

```
p := pb.Person{
        Id:    1234,
        Name:  "John Doe",
        Email: "jdoe@example.com",
        Phones: []*pb.Person_PhoneNumber{
                {Number: "555-4321", Type: pb.Person_HOME},
        },
}
```

Writing a Message

The whole purpose of using protocol buffers is to serialize your data so that it can be parsed elsewhere. In Go, you use the `proto` library's Marshal (https://godoc.org/github.com/golang/protobuf/proto#Marshal) function to serialize your protocol buffer data. A pointer to a protocol buffer message's `struct` implements the `proto.Message` interface. Calling `proto.Marshal` returns the protocol buffer, encoded in its wire format. For example, we use this function in the `add_person` command (https://github.com/google/protobuf/blob/master/examples/add_person.go):

```
book := &pb.AddressBook{}
// ...

// Write the new address book back to disk.
out, err := proto.Marshal(book)
if err != nil {
        log.Fatalln("Failed to encode address book:", err)
}
if err := ioutil.WriteFile(fname, out, 0644); err != nil {
        log.Fatalln("Failed to write address book:", err)
}
```

Reading a Message

To parse an encoded message, you use the `proto` library's Unmarshal (https://godoc.org/github.com/golang/protobuf/proto#Unmarshal) function. Calling this parses the data in `buf` as a protocol buffer and places the result in `pb`. So to parse the file in the `list_people` command (https://github.com/google/protobuf/blob/master/examples/list_people.go), we use:

```
// Read the existing address book.
in, err := ioutil.ReadFile(fname)
if err != nil {
        log.Fatalln("Error reading file:", err)
}
book := &pb.AddressBook{}
if err := proto.Unmarshal(in, book); err != nil {
        log.Fatalln("Failed to parse address book:", err)
}
```

Extending a Protocol Buffer

Sooner or later after you release the code that uses your protocol buffer, you will undoubtedly want to "improve" the protocol buffer's definition. If you want your new buffers to be backwards-compatible, and your old buffers to be forward-compatible – and you almost certainly do want this – then there are some rules you need to follow. In the new version of the protocol buffer:

- you *must not* change the tag numbers of any existing fields.
- you *may* delete fields.
- you *may* add new fields but you must use fresh tag numbers (i.e. tag numbers that were never used in this protocol buffer, not even by deleted fields).

(There are some exceptions (https://developers.google.com/protocol-buffers/docs/proto3.html#updating) to these rules, but they are rarely used.)

If you follow these rules, old code will happily read new messages and simply ignore any new fields. To the old code, singular fields that were deleted will simply have their default value, and deleted repeated fields will be empty. New code will also transparently read old messages.

However, keep in mind that new fields will not be present in old messages, so you will need to do something reasonable with the default value. A type-specific default value (https://developers.google.com/protocol-buffers/docs/proto3#default) is used: for strings, the default value is the empty string. For booleans, the default value is false. For numeric types, the default value is zero.

上次更新日期：五月 12, 2017

Downloads
Protocol buffers downloads and instructions

GitHub
The latest protocol buffers code and releases