Protocol Buffers in Go

...

Submitted by ajanicij on Thu, 06/07/2012 - 10:34

This is a tutorial for using Protocol Buffers in Go. I use the same .proto file as the one in Google's original tutorials for using Protocol Buffers in C++, Java and Python. This was the way for me to convince myself that 1) I understood Protocol Buffers correctly and 2) the implementation of Protocol Buffers for Go is correct (i.e. compatible with the implementations for C++, Java and Python).
You can find the original tutorials here. They all use the same .proto file, but first let's describe how to install the necessary Go package, goprotobuf. As its home page says, run

```
go get code.google.com/p/goprotobuf/{proto,protoc-gen-go}
```

and it will install the thing for you.
Next, here's the addressbook.proto file:

Recent posts

- Event-driven TLS Programming, Part 4
- Event-driven TLS Programming, Part 3
- Event-driven TLS Programming, Part 2
- Event-driven TLS Programming, Part 1
- OpenSSL Sample Code for using BIO

More

Recent comments

No comments available.

```
package tutorial;

option java_package = "com.example.tutorial";
option java_outer_classname = "AddressBookProtos";

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

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

 message PhoneNumber {
    required string number = 1;
    optional PhoneType type = 2 [default = HOME];
 }

 repeated PhoneNumber phone = 4;
}

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

Assuming that we are in the directory where addressbook.proto is and that we want the compiler output in the same directory, we compile this file to Go stub source using the following command:

```
protoc --go_out=. addressbook.proto
```

Project goprotobuf has added option --go_out to the standard protoc

compiler from Protocol Buffers project. That command has generated file addressbook.pb.go.
This file is central for our use of Protocol Buffers from Go, so let's discuss it.
First, since addressbook.proto contained the directive

```
package tutorial;
```

the generated file addressbook.pb.go has the package declaration with the same name:

```
package tutorial
```

We have to decide the path of the package tutorial. By the rules of Go, all source files for packages on a system should be under $GOPATH/src, and all compiled packages should be under $GOPATH/pkg, where GOPATH is the environment variable that was set in the installation procedure of Go.
The path of a package should be unique, and the best way to ensure that is to include a domain name that we own in the path. For example, all packages control by Google are under code.google.com/p: goprotobuf's package proto has the path code.google.com/p/goprotobuf/proto.
For this tutorial, we will use my (currently dormant) domain vogonsoft.com, and the path to the package is vogonsoft.com/experiment/protobuf.
Because my GOPATH is ~/mygo, this means addressbook.pb.go should be in ~/mygo/src/vogonsoft.com/experiment/protobuf/.
We can either copy addressbook.pb.go to that directory manually (or via Makefile or whichever build system we use), or we could specify the directory in the protoc command line, like this:

```
protoc --
go_out=$GOPATH/src/vogonsoft.com/experiment/protobuf/
addressbook.proto
```

Before we can use this package, we have to build it. We do it running the following command from any directory:

go install vogonsoft.com/experiment/protobuf

This produces a compiled package file (on Linux; I don't know what the file name would be on Windows):

~/mygo/pkg/linux_386/vogonsoft.com/experiment/protobuf/tutorial.a

Note that the file name (tutorial.a) is unrelated to the source file name (a package can have many source files anyway), but is equal to the package name.

When we import the package tutorial in our Go applications, we use the import statement:

```
import "vogonsoft.com/experiment/protobuf/tutorial"
```

and all exported names from the package tutorial should be prefixed by tutorial.
Now we discuss the contents of addressbook.pb.go. If you have studied the tutorials for Protocol Buffers in C++, Java or Python, you will notice that the Go version is smaller and simpler. Here we will compare with the C++ version, described in

https://developers.google.com/protocol-buffers/docs/cpptutorial

Whereas the C++ version (declared in addressbook.pb.h and implemented in addressbook.pb.cc) declares class Person, the Go version declared struct Person, which is pretty bare-bones:

```
type Person struct {
   Name             *string
`protobuf:"bytes,1,req,name=name"
json:"name,omitempty"`
   Id               *int32
`protobuf:"varint,2,req,name=id" json:"id,omitempty"`
```

```
   Email            *string
`protobuf:"bytes,3,opt,name=email"
json:"email,omitempty"`
   Phone            []*Person_PhoneNumber
`protobuf:"bytes,4,rep,name=phone"
json:"phone,omitempty"`
   XXX_unrecognized []byte                `json:"-"`
}

func (this *Person) Reset()         { *this = Person{}
}
func (this *Person) String() string { return
proto.CompactTextString(this) }
```

All fields defined in addressbook.proto exist here, but with capitalized first letter in order to be exported. For example, in addressbook.proto we have

```
required string name = 1;
```

and in addressbook.pb.go:

```
Name             *string
```

Name is defined as pointer to string. C++ stub has accessor has_name(), but in Go you have to test for nil. In C++ stub you clear the field by calling clear_name(), whereas in Go you set Name to nil. There are no accessors in Go stub - we deal with the Name field directly.
A repeated field compiles to a slice. In addressbook.proto, message Person has

```
repeated PhoneNumber phone = 4;
```

and that compiles to

```
Phone            []*Person_PhoneNumber
```

Again, no accessors: we deal with the field directly. We get the size of the slice by calling the built-in function len and add new phone number by calling the built-in function append. We will see how to do these later when we show the programs to read and write address book files.

Enums and Nested Classes

In addressbook.proto we have enum PhoneType nested in the message Person:

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

In the generated addressbook.pb.go, this is translated to type Person_PhoneType and the set of constants:

```
type Person_PhoneType int32

const (
   Person_MOBILE Person_PhoneType = 0
   Person_HOME   Person_PhoneType = 1
   Person_WORK   Person_PhoneType = 2
)

var Person_PhoneType_name = map[int32]string{
   0: "MOBILE",
   1: "HOME",
   2: "WORK",
}
var Person_PhoneType_value = map[string]int32{
```

```
    "MOBILE": 0,
    "HOME":   1,
    "WORK":   2,
}
```

Standard Message Methods

The structure Person in addressbook.pb.go has only two methods, Reset and String:

```
func (this *Person) Reset()         { *this = Person{}
}
func (this *Person) String() string { return
proto.CompactTextString(this) }
```

Parsing and Serialization

Unlike the C++ stub, where parsing and serialization are implemented by member functions of class Person, in the Go version there are no method for that. We achieve these operations by using Marshal and Unmarshal functions of the proto package, as described in

http://code.google.com/p/goprotobuf/source/browse/READ

Writing a Message

Just like in the C++ tutorial, we write a program which reads an AddressBook from a file, adds one new Person to it based on user input, and writes the new AddressBook back out to the file again.

```
package main

import (
   "fmt"
   "os"
   "io"
   "bufio"
   "code.google.com/p/goprotobuf/proto"
   "vogonsoft.com/experiment/protobuf/tutorial"
)

func PromptForAddress() *tutorial.Person {
   person := &tutorial.Person{}
   fmt.Print("Enter person ID number: ")
   var id int32
   fmt.Scan(&id)
   person.Id = &id

   fmt.Print("Enter name: ")
   var name string
   name = readLine(os.Stdin)
   person.Name = &name

   fmt.Print("Enter email address (blank for none): ")
   email := readLine(os.Stdin)
   if email != "" {
       person.Email = &email
   }

   for {
```

```
        fmt.Println("Enter a phone number (or leave
blank to finish): ")
        number := readLine(os.Stdin)
        if number == "" {
            break
        }
        phone_number :=
new(tutorial.Person_PhoneNumber)
        phone_number.Number = &number
        fmt.Print("Is this a mobile, home, or work
phone? ")
        _type := readLine(os.Stdin)
        var phone_type tutorial.Person_PhoneType
        switch {
            case _type == "mobile":
                phone_type = tutorial.Person_MOBILE
            case _type == "home":
                phone_type = tutorial.Person_HOME
            case _type == "work":
                phone_type = tutorial.Person_WORK
            default:
                fmt.Println("Unknown phone type. Using
default.")
        }
        phone_number.Type = &phone_type
        person.Phone = append(person.Phone,
phone_number)
    }
    return person
}

func main() {
    if len(os.Args) != 2 {
        fmt.Printf("Usage: %s ADDRESS_BOOK_FILE\n",
os.Args[0])
        os.Exit(-1)
    }
```

```
    address_book := &tutorial.AddressBook{}

    file, err := os.Open(os.Args[1])
    if err != nil {
        fmt.Println(os.Args[1], ": File not found.
 Creating a new file.")
    } else {
        fi, err := file.Stat()
        CheckError(err)

        buffer := make([]byte, fi.Size())
        _, err = io.ReadFull(file, buffer)
        file.Close()

        err = proto.Unmarshal(buffer, address_book)
        CheckError(err)
    }

    person := PromptForAddress()
    address_book.Person = append(address_book.Person,
person)

    file, err = os.OpenFile(os.Args[1], os.O_CREATE |
os.O_WRONLY, 0644)
    CheckError(err)
    buffer2, err := proto.Marshal(address_book)
    CheckError(err)
    _, err = file.Write(buffer2)
    CheckError(err)
    file.Close()
}

func CheckError(err error) {
    if err != nil {
        fmt.Println(err.Error())
        os.Exit(-1)
```

```
    }
}

func readLine(rd io.Reader) string {
    reader := bufio.NewReader(rd)
    line := ""
    for {
        buffer, isPrefix, err := reader.ReadLine()
        if err != nil {
            break
        }
        line = line + string(buffer)
        if !isPrefix {
            break
        }
    }
    return line
}
```

I will not describe all details of this program, but note how we parse a message file by first reading all of it to memory and then calling proto.Unmarshal. Likewise, we serialize by marshaling an object to a memory buffer (calling proto.Marhsal) and saving the buffer to a file.

Reading a Message

This is a program to read an address book file:

```
package main

import (
    "fmt"
    "os"
```

```
    "io"
    "code.google.com/p/goprotobuf/proto"
    "vogonsoft.com/experiment/protobuf/tutorial"
)

func ListPeople(address_book *tutorial.AddressBook) {
    for _, person := range address_book.Person {
        fmt.Println("Person ID:", *person.Id)
        fmt.Println("  Name:", *person.Name)
        if person.Email != nil {
            fmt.Println("  E-mail address:",
*person.Email)
        }
        for _, phone_number := range person.Phone {
            if *phone_number.Type ==
tutorial.Person_MOBILE {
                fmt.Print("  Mobile phone #: ")
            } else if *phone_number.Type ==
tutorial.Person_HOME {
                fmt.Print("  Home phone #: ")
            } else if *phone_number.Type ==
tutorial.Person_WORK {
                fmt.Print("  Work phone #: ")
            }
            fmt.Println(*phone_number.Number)
        }
    }
}

func main() {
    if len(os.Args) != 2 {
        fmt.Printf("Usage: %s ADDRESS_BOOK_FILE\n",
os.Args[0])
        os.Exit(-1)
    }

    // Read file
```

```
    file, err := os.Open(os.Args[1])
    CheckError(err)

    fi, err := file.Stat()
    CheckError(err)

    buffer := make([]byte, fi.Size())
    _, err = io.ReadFull(file, buffer)
    file.Close()

    address_book := &tutorial.AddressBook{}
    err = proto.Unmarshal(buffer, address_book)
    CheckError(err)

    ListPeople(address_book)
}

func CheckError(err error) {
    if err != nil {
        fmt.Println(err.Error())
        os.Exit(-1)
    }
}
```

And that's it. The C++ tutorial ends with a discussing how extend a protocol buffer, optimization tips, and notes on advanced usage, which I won't reproduce in this basic tutorial.

Tags:
Protocol Buffers in Go Go language

Post new comment

Your name *

E-mail *

The content of this field is kept private and w ill not be show n publicly.

Homepage

Subject

Comment *

WYSIWYG HTML

Filtered HTML

More information about text formats

- Web page addresses and e-mail addresses turn into links automatically.
- Allowed HTML tags: <a>   <cite> <blockquote> <code>   <dl> <dt> <dd> <p>
- Lines and paragraphs break automatically.

Plain text

- No HTML tags allowed.
- Web page addresses and e-mail addresses turn into links automatically.
- Lines and paragraphs break automatically.

By submitting this form, you accept the Mollom privacy policy.

Save Preview

Powered by Drupal Gardens
