Protocol Buffers

What Are Protocol Buffers

Protocol buffers are Google's language-neutral, platform-neutral, extensible mechanism for serializing structured data – think XML, but smaller, faster, and simpler. You define how you want your data to be structured once, then you can use special generated source code to easily write and read your structured data to and from a variety of data streams and using a variety of languages.

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

message Person {
  string name = 1;
  int32 id = 2;
  string email = 3;
}
```

Pick Your Favorite Language

Protocol buffers currently supports generated code in Java, Python, and C++. With plugins, you can also work with Go, JavaNano, Ruby, and C#, with more languages to come.

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

Getting The Protobuff Compiler

Download Protocol Buffer Compiler

https://github.com/google/protobuf/releases

Compiler Invocation

The protocol buffer compiler requires a plugin to generate Go code.

```
$ go get -u github.com/gogo/protobuf/proto
$ go get -u github.com/gogo/protobuf/protoc-gen-gogo
$ go get -u github.com/gogo/protobuf/gogoproto
```

This will install a `protoc-gen-gogo` binary which `protoc` uses when invoked with the `--gogo_out` command-line flag.

Compiler Invocation

The `--gogo_out` flag tells the compiler where to write the Go source files.

The compiler creates a single source file for each `.proto` file input.

The names of the output files are computed by taking the name of the `.proto` file and replacing the extension (`.proto`) with `.pb.go`.

Example Invocation

A file called `person.proto` results in an output file called `person.pb.go`.

When you run the proto compiler like this:

```
$ protoc --gogo_out=tutorial person.proto
```

The compiler will produce a new file, `tutorial/person.pb.go`.

The compiler automatically creates any needed sub-directories if necessary, but it will not create any top level directories, in this case the `tutorial` directory.

Using Go Generate

If you want to automatically generate your proto files with `go generate`, you can add this directive in one of the go files in the directory that contains the proto files:

```
//go:generate protoc --gogo_out=. your_proto_file.proto
```

To make the definition private by using the `internal` package name:

```
//go:generate protoc --gogo_out=. internal/your_proto_file.proto
```

It's common to put this directive in an empty file called `generate.go`

Troubleshooting

If generating the protobuf code is failing for you, check each of the following:

- Ensure the protobuf library can be found. Make sure that `LD_LIBRRARY_PATH` includes the directory in which the library `libprotoc.so` has been installed.
- Ensure the command `protoc-gen-gogo`, found in `GOPATH/bin`, is in your `$PATH`. This can be done by adding `$GOPATH/bin` to `$PATH`.

Your First Proto Project

Create a file called `node.proto` and add the following contents:

```
syntax = "proto3";
package services;

message Node {
  string uuid = 1; // Universally Unique ID for this node
  string uri = 2; // URI to access this resource
  int64 updatedAt = 3;
}

message service {
  enum ServiceType {
    MasterPrimary = 0;
    MasterSecondary = 1;
  }
  string name = 1; // The name of the service
  ServiceType type = 2;
  repeated Node = 3;
}
```

Breaking It Apart

Syntax

```
syntax = "proto3";
```

The first line of the file specifies that you're using `proto3` syntax: if you don't do this the protocol buffer compiler will assume you are using `proto2`.

This must be the first non-empty, non-comment line of the file.

Packages

```
package services;
```

If a `.proto` file contains a package declaration, the generated code uses the proto's package as its Go package name, converting `.` characters into `_` first.

A proto package name of `example.high_score` results in a Go package name of `example_high_score`.

Packages

You can override the default generated package for a particular `.proto` using the option `go_package` in your `.proto` file.

```
package example.high_score;
option go_package = "hs";
```

https://developers.google.com/protocol-buffers/docs/reference/go-generated

Message

A `message` in protobuf is analogous to a `struct` in Go. It's a representation of the message that will be passed.

```
message Node {
  string uuid = 1; // Universally Unique ID for this node
  string uri = 2; // URI to access this resource
  int64 updatedAt = 3;
}
```

Well-Known Types

Protobufs comes with a set of "predefined types".

You can import these types using the `import` directive.

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

message NamedStruct {
    string name = 1;
    google.protobuf.Timestamp last_modified = 2;
}
```

```
$ go get -u -v github.com/golang/protobuf/...
```

https://developers.google.com/protocol-buffers/docs/reference/google.protobuf

Why Use Well-Known Types

These well-known, predefined, types allow us to ensure compatiblity across different languages.

For example if you talking between Go and Java the `google.protobuf.Timestamp` will make sure that "time" will get translated correctly across the languages.

Fields

The protocol buffer compiler generates a `struct` field for each field defined within a message. The exact nature of this field depends on its type and whether it is a `singular`, `repeated`, `map`, or `oneof` field.

```
// <type> <name> = <tag>;
int64 updatedAt = 3;
```

Field Tags (Numbers)

Each field in the message definition has a **unique** numbered tag. These tags are used to identify your fields in the message binary format, and should **NOT** be changed once your message type is in use.

```
message Node {
  string uuid = 1; // Universally Unique ID for this node
  string uri = 2; // URI to access this resource
  int64 updatedAt = 3;
}
```

Field Tag Optimizations

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.

Specifying Field Rules

Message fields can be one of the following:

- `singular`: a well-formed message can have zero or one of this field (but not more than one).
- `repeated`: this field can be repeated any number of times (including zero) in a well-formed message. The order of the repeated values will be preserved.

Generated Field Names

The generated Go field names always use camel-case naming, even if the field name in the `.proto` file uses lower-case with underscores (as it should).

- The first letter is capitalized for export.
- If the first character is an underscore, it is removed and a capital `X` is prepended.
- If an interior underscore is followed by a lower-case letter, the underscore is removed, and the following letter is capitalized.
    - `foo_bar_baz -> FooBarBaz`
    - `_my_field_name_2 -> XMyFieldName_2`

Map Fields

Each map field generates a field in the struct of type `map[TKey]TValue` where `TKey` is the field's key type and `TValue` is the field's value type.

```
// baz.proto
message Bar {}

message Baz {
  map<string, Bar> foo = 1;
}
```

```
// baz.pb.go
type Baz struct {
  Foo map[string]*Bar
}
```

Require Vs. Optional

Prior to `proto3`, you could use the `optional` and `required` syntax to define whether a field was required or not.

```
optional int32 foo = 1;
required int32 foo = 1;
```

Initially the `optional` and `required` directives seemed to make sense, but as projects evolved, it began to create cruft for fields that were originally defined as `required`… and then one day they weren't.

Services

The Go code generator does not produce output for services by default. If you enable the gRPC plugin (see the gRPC Go Quickstart guide) then code will be generated to support gRPC.

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).

Exercise

Write a protobuf message that contains the following fields:

- `name` which is a `string`
- `age` which is an `int`
- `preferences` which is a `map[string]string{}`

Generate the Go code using `protoc`.

Bonus:

- Add a `repeated` field and regenerate.
- Try using to identical number tags. What happens?
- What happens if you specify a type that doesn't exist?
- Use a well-known type like `google.protobuf.Timestamp` and regenerate.
- Use `go generate` with a `generate.go` file to generate the code.

Solution

```
syntax = "proto3";

import "google/protobuf/timestamp.proto";

package tutorial;

message Person {
  string name = 1;
  int32 age = 2;
  map<string, string> preferences = 3;
  repeated string kids = 4;
  google.protobuf.Timestamp birthdate = 5;
}
```

```
$ protoc --gogo_out=tutorial/ person.proto
```

```
//go:generate protoc --gogo_out=. person.proto
package tutorial
```

References

A number of these slides contain content copied directly from the following resources:

1. https://developers.google.com/protocol-buffers/docs/reference/go-generated
2. https://github.com/google/protobuf
3. https://developers.google.com/protocol-buffers/docs/gotutorial
4. https://godoc.org/github.com/golang/protobuf/proto
5. https://developers.google.com/protocol-buffers/