Go Generated Code

Compiler Invocation (#invocation)

Packages (#package)

Messages (#message)

Fields (#fields)

- Singular Scalar Fields (proto2) (#singular-scalar-proto2)
- Singular Scalar Fields (proto3) (#singular-scalar-proto3)
- Singular Message Fields (#singular-message)
- Repeated Fields (#repeated)
- Map fields (#map)
- Oneof Fields (#oneof)

Enumerations (#enum)

Services (#service)

This page describes exactly what Go code the protocol buffer compiler generates for any given protocol definition. Any differences between proto2 and proto3 generated code are highlighted - note that these differences are in the generated code as described in this document, not the base API, which are the same in both versions. You should read the proto2 language guide (https://developers.google.com/protocol-buffers/docs/proto) and/or the proto3 language guide (https://developers.google.com/protocol-buffers/docs/proto3) before reading this document.

Compiler Invocation

The protocol buffer compiler requires a plugin to generate Go (//github.com/golang/protobuf) code. Installing it with

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

provides a `protoc-gen-go` binary which `protoc` uses when invoked with the `--go_out` command-line flag. The `--go_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 making two changes:

- The extension (`.proto`) is replaced with `.pb.go`. For example, a file called `player_record.proto` results in an output file called `player_record.pb.go`.
- The proto path (specified with the `--proto_path` or `-I` command-line flag) is replaced with the output path (specified with the `--go_out` flag).

When you run the proto compiler like this:

```
protoc --proto_path=src --go_out=build/gen src/foo.proto src/bar/baz.proto
```

the compiler will read the files `src/foo.proto` and `src/bar/baz.proto`. It produces two output files: `build/gen/foo.pb.go` and `build/gen/bar/baz.pb.go`.

The compiler automatically creates the directory `build/gen/bar` if necessary, but it will *not* create `build` or `build/gen`; they must already exist.

Packages

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

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

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

generates a file with the Go package name `hs`.

Otherwise, if a `.proto` file does not contain a package declaration, the generated code uses the file name (minus the extension) as its Go package name, converting `.` characters into `_` first. For example, a proto package named `high.score.proto` without a package declaration will result in a file `high.score.pb.go` with package `high_score`.

Messages

Given a simple message declaration:

```
message Foo {}
```

the protocol buffer compiler generates a struct called `Foo`. A `*Foo` implements the Message (https://godoc.org/github.com/golang/protobuf/proto#Message) interface. See the inline comments for more information.

```
type Foo struct {
}
```

```
// Reset sets the proto's state to default values.
func (m *Foo) Reset()         { *m = Foo{} }

// String returns a string representation of the proto.
func (m *Foo) String() string { return proto.CompactTextString(m) }

// ProtoMessage acts as a tag to make sure no one accidentally implements the
// proto.Message interface.
func (*Foo) ProtoMessage()    {}
```

Note that all of these members are always present; the **`optimize_for`** option does not affect the output of the Go code generator.

Nested Types

A message can be declared inside another message. For example:

```
message Foo {
  message Bar {
  }
}
```

In this case, the compiler generates two structs: **`Foo`** and **`Foo_Bar`**.

Well-known types

Protobufs come with a set of predefined messages, called well-known types (WKTs) (https://developers.google.com/protocol-buffers/docs/reference/google.protobuf). These types can be useful either for interoperability with other

services, or simply because they succinctly represent common, useful patterns. For example, the `Struct` (https://developers.google.com/protocol-buffers/docs/reference/google.protobuf#google.protobuf.Struct) message represents the format of an arbitrary C-style struct.

Pre-generated Go code for the WKTs is distributed as part of the Go protobuf library (//github.com/golang/protobuf), and this code is referenced by the generated Go code of your messages if they use a WKT. For example, given a message such as:

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

message NamedStruct {
  string name = 1;
  google.protobuf.Struct definition = 2;
  google.protobuf.Timestamp last_modified = 3;
}
```

the generated Go code will look something like the following:

```
import google_protobuf "github.com/golang/protobuf/ptypes/struct"
import google_protobuf1 "github.com/golang/protobuf/ptypes/timestamp"

...

type NamedStruct struct {
   Name         string
   Definition   *google_protobuf.Struct
   LastModified *google_protobuf1.Timestamp
}
```

Generally speaking, you shouldn't need to import these types directly into your code. However, if you need to reference one of these types directly, simply import the `github.com/golang/protobuf/ptypes/[TYPE]` package, and use the type normally.

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.

Note that 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 (https://developers.google.com/protocol-buffers/docs/style)). The case-conversion works as follows:

1. The first letter is capitalized for export. If the first character is an underscore, it is removed and a capital X is prepended.
2. If an interior underscore is followed by a lower-case letter, the underscore is removed, and the following letter is capitalized.

Thus, the proto field `foo_bar_baz` becomes `FooBarBaz` in Go, and `_my_field_name_2` becomes `XMyFieldName_2`.

Singular Scalar Fields (proto2)

For either of these field definitions:

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

the compiler generates a struct with an `*int32` field named `Foo` and an accessor method `GetFoo()` which returns the `int32` value in `Foo` or the default value if the field is unset. If the default is not explicitly set, the zero value (https://golang.org/ref/spec#The_zero_value) of that type is used instead (`0` for numbers, the empty string for strings).

For other scalar field types (including **`bool`**, **`bytes`**, and **`string`**), **`*int32`** is replaced with the corresponding Go type according to the scalar value types table (https://developers.google.com/protocol-buffers/docs/proto.html#scalar).

Singular Scalar Fields (proto3)

For this field definition:

```
int32 foo = 1;
```

The compiler will generate a struct with an **`int32`** field named **`Foo`**. No helper methods are generated.

For other scalar field types (including **`bool`**, **`bytes`**, and **`string`**), **`int32`** is replaced with the corresponding Go type according to the scalar value types table (https://developers.google.com/protocol-buffers/docs/proto3.html#scalar). Unset values in the proto will be represented as the zero value (https://golang.org/ref/spec#The_zero_value) of that type (**`0`** for numbers, the empty string for strings).

Singular Message Fields

Given the message type:

```
message Bar {}
```

For a message with a **`Bar`** field:

```
// proto2
message Baz {
  optional Bar foo = 1;
  // The generated code is the same result if required instead of optional.
```

```
}

// proto3
message Baz {
  Bar foo = 1;
}
```

The compiler will generate a Go struct

```
type Baz struct {
        Foo *Bar
}
```

Message fields can be set to `nil`, which means that the field is unset, effectively clearing the field. This is not equivalent to setting the value to an "empty" instance of the message struct.

The compiler also generates a `func (m *Baz) GetFoo() *Bar` helper function. This makes it possible to chain get calls without intermediate `nil` checks.

Repeated Fields

Each repeated field generates a slice of `T` field in the struct in Go, where `T` is the field's element type. For this message with a repeated field:

```
message Baz {
  repeated Bar foo = 1;
}
```

the compiler generates the Go struct:

```
type Baz struct {
        Foo  []*Bar
}
```

Likewise, for the field definition `repeated bytes foo = 1;` the compiler will generate a Go struct with a `[][]byte` field named `Foo`. For a repeated enumeration (#enum) `repeated MyEnum bar = 2;`, the compiler generates a struct with a `[]MyEnum` field called `Bar`.

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. For this message with a map field:

```
message Bar {}

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

the compiler generates the Go struct:

```
type Baz struct {
        Foo map[string]*Bar
}
```

Oneof Fields

For a oneof field, the protobuf compiler generates a single field with an interface type **`isMessageName_MyField`**. It also generates a struct for each of the singular fields (#singular-scalar) within the oneof. These all implement this **`isMessageName_MyField`** interface.

For this message with a oneof field:

```
package account;
message Profile {
  oneof avatar {
    string image_url = 1;
    bytes image_data = 2;
  }
}
```

the compiler generates the structs:

```
type Profile struct {
        // Types that are valid to be assigned to Avatar:
        //      *Profile_ImageUrl
        //      *Profile_ImageData
        Avatar isProfile_Avatar `protobuf_oneof:"avatar"`
}

type Profile_ImageUrl struct {
        ImageUrl string
}
type Profile_ImageData struct {
        ImageData []byte
}
```

Both `*Profile_ImageUrl` and `*Profile_ImageData` implement `isProfile_Avatar` by providing an empty `isProfile_Avatar()` method. This means that you can use a type switch on the value to handle the different message types.

```
switch x := m.Avatar.(type) {
case *account.Profile_ImageUrl:
        // Load profile image based on URL
        // using x.ImageUrl
case *account.Profile_ImageData:
        // Load profile image based on bytes
        // using x.ImageData
case nil:
        // The field is not set.
default:
        return fmt.Errorf("Profile.Avatar has unexpected type %T", x)
}
```

The compiler also generates get methods `func (m *Profile) GetImageUrl() string` and `func (m *Profile) GetImageData() []byte`. Each get function returns the value for that field or the zero value if it is not set.

Enumerations

Given an enumeration like:

```
message SearchRequest {
  enum Corpus {
    UNIVERSAL = 0;
    WEB = 1;
    IMAGES = 2;
    LOCAL = 3;
```

```
    NEWS = 4;
    PRODUCTS = 5;
    VIDEO = 6;
  }
  Corpus corpus = 1;
  ...
}
```

the protocol buffer compiler generates a type and a series of constants with that type.

For enums within a message (like the one above), the type name begins with the message name:

```
type SearchRequest_Corpus int32
```

For a package-level enum:

```
enum Foo {
  DEFAULT_BAR = 0;
  BAR_BELLS = 1;
  BAR_B_CUE = 2;
}
```

the Go type name is unmodified from the proto enum name:

```
type Foo int32
```

This type has a `String()` method that returns the name of a given value.

The protocol buffer compiler generates a constant for each value in the enum. For enums within a message, the constants begin with the enclosing message's name:

```
const (
        SearchRequest_UNIVERSAL SearchRequest_Corpus = 0
        SearchRequest_WEB       SearchRequest_Corpus = 1
        SearchRequest_IMAGES    SearchRequest_Corpus = 2
        SearchRequest_LOCAL     SearchRequest_Corpus = 3
        SearchRequest_NEWS      SearchRequest_Corpus = 4
        SearchRequest_PRODUCTS  SearchRequest_Corpus = 5
        SearchRequest_VIDEO     SearchRequest_Corpus = 6
)
```

For a package-level enum, the constants begin with the enum name instead:

```
const (
        Foo_DEFAULT_BAR Foo = 0
        Foo_BAR_BELLS   Foo = 1
        Foo_BAR_B_CUE   Foo = 2
)
```

The protobuf compiler also generates a map from integer values to the string names and a map from the names to the values:

```
var Foo_name = map[int32]string{
        0: "DEFAULT_BAR",
        1: "BAR_BELLS",
        2: "BAR_B_CUE",
}
var Foo_value = map[string]int32{
        "DEFAULT_BAR": 0,
        "BAR_BELLS":   1,
        "BAR_B_CUE":   2,
}
```

Note that the `.proto` language allows multiple enum symbols to have the same numeric value. Symbols with the same numeric value are synonyms. These are represented in Go in exactly the same way, with multiple names corresponding to the same numeric value. The reverse mapping contains a single entry for the numeric value to the name which appears first in the .proto file.

Services

The Go code generator does not produce output for services by default. If you enable the gRPC (//www.grpc.io/) plugin (see the gRPC Go Quickstart guide (//github.com/grpc/grpc-go/tree/master/examples)) then code will be generated to support gRPC.

上次更新日期：二月 1, 2017

Downloads
Protocol buffers downloads and instructions

GitHub
The latest protocol buffers code and releases