pbtools Documentation

Release 0.47.0

Erik Moqvist

Jun 04, 2023

Contents

CHAPTER 1

About

Google Protocol Buffers tools in Python 3.6+.

- *C* source code generator.
- proto3 language parser.

Known limitations:

- Options, services (gRPC) and reserved fields are ignored.
- Public imports are not implemented.

Project homepage: https://github.com/eerimoq/pbtools

Documentation: https://pbtools.readthedocs.io

CHAPTER 2

Installation

```
pip install pbtools
```

CHAPTER 3

C source code design

The C source code is designed with the following in mind:

- Clean and easy to use API.
- No malloc/free. Uses a workspace/arena for memory allocations.
- Fast encoding and decoding.
- Small memory footprint.
- Thread safety.

Known limitations:

- `char` must be 8 bits.

ToDo:

- Make `map` easier to use. Only one allocation should be needed before encoding, not one per sub-message item.

3.1 Memory management

A workspace, or arena, is used to allocate memory when encoding and decoding messages. For simplicity, allocated memory can't be freed, which puts restrictions on how a message can be modified between encodings (if one want to do that). Scalar value type fields (ints, strings, bytes, etc.) can be modified, but the length of repeated fields can't.

3.2 Scalar Value Types

Protobuf scalar value types are mapped to C types as shown in the table below.

Protubuf Type	C Type
double	double
float	float
int32	int32_t
int64	int64_t
uint32	uint32_t
uint64	uint64_t
sint32	int32_t
sint64	int64_t
fixed32	int32_t
fixed64	int64_t
sfixed32	int32_t
sfixed64	int64_t
bool	bool
string	char *
bytes	struct { uint8_t *buf_p, size_t size }

3.3 Message

A message is a struct in C.

For example, let's create a protocol specification.

```
syntax = "proto3";

package foo;

message Bar {
    bool v1 = 1;
}

message Fie {
    int32 v2 = 1;
    Bar v3 = 2;
}
```

One struct is generated per message.

```
struct foo_bar_t {
    bool v1;
};

struct foo_fie_t {
    int32_t v2;
    struct foo_bar_t *v3_p;
};
```

The sub-message `v3` has to be allocated before encoding and checked if `NULL` after decoding.

```
struct foo_fie_t *fie_p;

/* Encode. */
fie_p = foo_fie_new(...);
```

(continues on next page)

(continued from previous page)

```
fie_p->v2 = 5;
foo_fie_v3_alloc(fie_p);
fie_p->v3_p->v1 = true;
foo_fie_encode(fie_p, ...);

/* Decode. */
fie_p = foo_fie_new(...);
foo_fie_decode(fie_p, ...);

printf("%d\n", fie_p->v2);

if (fie_p->v3_p != NULL) {
    printf("%d\n", fie_p->v3_p->v1);
}
```

3.4 Oneof

A oneof is an enum (the choice) and a union in C.

For example, let's create a protocol specification.

```
syntax = "proto3";

package foo;

message Bar {
    oneof fie {
        int32 v1 = 1;
        bool v2 = 2;
    };
}
```

One enum and one struct is generated per oneof.

```
enum foo_bar_fie_e {
    foo_bar_fie_none_e = 0,
    foo_bar_fie_v1_e = 1,
    foo_bar_fie_v2_e = 2
};

struct foo_bar_t {
    enum foo_bar_fie_choice_e fie;
    union {
        int32_t v1;
        bool v2;
    };
};
```

The generated code can encode and decode messages. Call `_<field>_init()` or `_<field>_alloc()` to select which oneof field to encode. Use the `enum` to check which oneof field was decoded (if any).

```
struct foo_bar_t *bar_p;

/* Encode with choice v1. */
```

(continues on next page)

(continued from previous page)

```
bar_p = foo_bar_new(...);
foo_bar_v1_init(bar_p);
bar_p->v1 = -2;
foo_bar_encode(bar_p, ...);

/* Decode. */
bar_p = foo_bar_new(...);
foo_bar_decode(bar_p, ...);

switch (bar_p->fie) {

case foo_bar_fie_none_e:
    printf("Not present.\n");
    break;

case foo_bar_fie_v1_e:
    printf("%d\n", bar_p->v1);
    break;

case foo_bar_fie_v2_e:
    printf("%d\n", bar_p->v2);
    break;

default:
    printf("Can not happen.\n");
    break;
}
```

3.5 Benchmark

See benchmark for a benchmark of a few C/C++ protobuf libraries.

CHAPTER 4

Example usage

4.1 C source code

In this example we use the simple proto-file hello_world.proto.

```
syntax = "proto3";

package hello_world;

message Foo {
    int32 bar = 1;
}
```

Generate C source code from the proto-file.

```
$ pbtools generate_c_source examples/hello_world/hello_world.proto
```

See hello_world.h and hello_world.c for the contents of the generated files.

We'll use the generated types and functions below.

```
struct hello_world_foo_t {
   struct pbtools_message_base_t base;
   int32_t bar;
};

struct hello_world_foo_t *hello_world_foo_new(
    void *workspace_p,
    size_t size);

int hello_world_foo_encode(
    struct hello_world_foo_t *self_p,
    void *encoded_p,
    size_t size);
```

(continues on next page)

(continued from previous page)

```
int hello_world_foo_decode(
    struct hello_world_foo_t *self_p,
    const uint8_t *encoded_p,
    size_t size);
```

Encode and decode the Foo-message in main.c.

```
#include <stdio.h>
#include "hello_world.h"

int main(int argc, const char *argv[])
{
    int size;
    uint8_t workspace[64];
    uint8_t encoded[16];
    struct hello_world_foo_t *foo_p;

    /* Encode. */
    foo_p = hello_world_foo_new(&workspace[0], sizeof(workspace));

    if (foo_p == NULL) {
        return (1);
    }

    foo_p->bar = 78;
    size = hello_world_foo_encode(foo_p, &encoded[0], sizeof(encoded));

    if (size < 0) {
        return (2);
    }

    printf("Successfully encoded Foo into %d bytes.\n", size);

    /* Decode. */
    foo_p = hello_world_foo_new(&workspace[0], sizeof(workspace));

    if (foo_p == NULL) {
        return (3);
    }

    size = hello_world_foo_decode(foo_p, &encoded[0], size);

    if (size < 0) {
        return (4);
    }

    printf("Successfully decoded %d bytes into Foo.\n", size);
    printf("Foo.bar: %d\n", foo_p->bar);

    return (0);
}
```

Build and run the program.

```
$ gcc -I lib/include main.c hello_world.c lib/src/pbtools.c -o main
$ ./main
```

(continues on next page)

(continued from previous page)

```
Successfully encoded Foo into 2 bytes.
Successfully decoded 2 bytes into Foo.
Foo.bar: 78
```

See examples/hello_world for all files used in this example.

4.2 Command line tool

4.2.1 The generate C source subcommand

Below is an example of how to generate C source code from a proto-file.

```
$ pbtools generate_c_source examples/address_book/address_book.proto
```

See address_book.h and address_book.c for the contents of the generated files.

CHAPTER 5

Functions and classes

pbtools.**parse_file**(*filename*, *import_paths=None*)
Parse given proto3-file *filename* and its imports. Returns a *Proto* object.

import_paths is a list of paths where to search for imported files.

class pbtools.parser.**Proto**(*tree*, *abspath*, *import_paths*)
A proto3-file. *parse_file()* returns an instance of this class.

package
Package name, or None if missing.

imports
A list of all imports.

options
A list of all options.

services
A list of all services.

messages
A list of all messages.

enums
A list of all enums.

class pbtools.parser.**Message**(*tokens*, *namespace*)
A message.

pbtools.c_source.**generate_files**(*infiles*, *import_paths=None*, *output_directory='.'*, *options=None*)
Generate C source code from proto-file(s).

Index