Protobuf in Elixir

@tony612

Overview

- Intro to Proto Buffers
- How I implement it in Elixir
- What I learned by writing protobuf-elixir

"Protocol buffers are a language-neutral, platform-neutral extensible mechanism for serializing structured data."

— by Google

Protobuf

- A data format(like JSON)
- Structured data with schema
- Encoded as binary
- Written in proto and generated in any language

```proto
syntax = "proto3";

package demo;

message Location {
  string street = 0;
  int32 number = 1;
}

message Person {
  string name = 0;
}

message Meetup {
  string theme = 0;
  double time = 1;
  Location location = 2;
  bool free = 3;
  repeated Person attendee = 4;
}
```

Protobuf

```json
{
  "meetup": {
    "theme": "",
    "time": 1506146400,
    "location": {
      "street": "",
      "number": 970
    },
    "free": true,
    "attendees": [
      {
        "name": "Tony"
      },
      {
        "name": "Vangie"
      }
    ]
  }
}
```

JSON

Encoding

```
message Test1 {
   required int32 a = 1;
}
```

%Test1{a: 150}

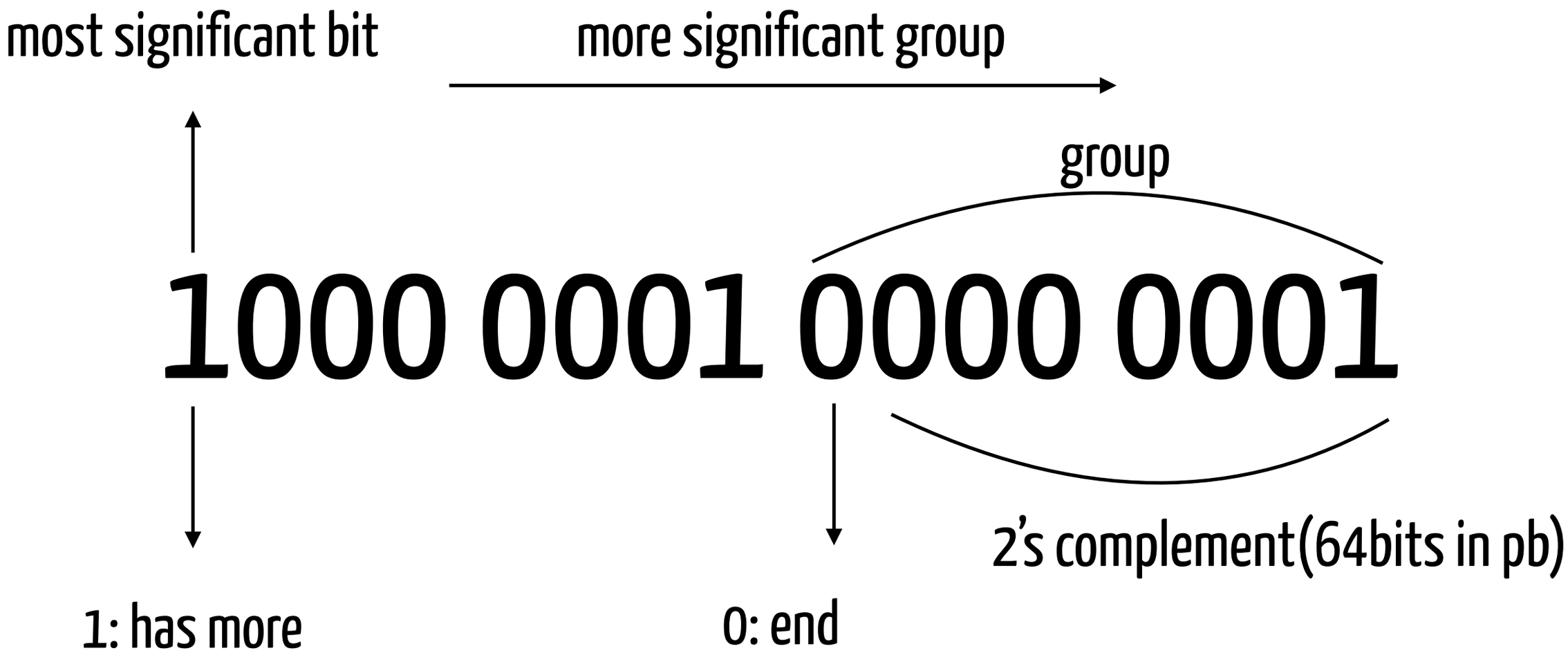
08 150 01

Protobuf 3 bytes

```
{"a":150}
```

JSON 9 bytes

Base 128 Varints



variable int: store an arbitrarily large integer in a small number of bytes

Base 128 Varints

Base128 varint	Decimal	Calculate
0000 0001	1	1
0111 1111	127	111 1111 = 127
1000 0000 0000 0001	128	000 0001 000 0000
1001 0110 0000 0001(150 01)	150	000 0001 001 0110
1111 1111 …(total 9) 0000 0001	-1	1111 1111 111 1111… (64 1)

Wire type

Type	Meaning	Used For
0	Varint	int32, int64, uint32, uint64, sint32, sint64, bool, enum
1	64-bit	fixed64, sfixed64, double
2	Length-delimited	string, bytes, embedded messages, packed repeated fields
3	Start group	groups (deprecated)
4	End group	groups (deprecated)
5	32-bit	fixed32, sfixed32, float

Encoding

```
message Test1 {
   required int32 a = 1;
}
```

08 150 01

varint of key

value

(field_number << 3) | wire_type

(1 << 3) | 0

1x8+0

Decode logic

<<varint_key, varint_val, varint_key, length_delimited_val, varint_key, 64bits_val, ...>>

1. Decode varint to get field number and wire type
2. Get the bit string(value) based on wire type(varint, Length-delimited)
3. Decode the bit string to get right value based on metadata

Protobuf VS JSON

Protobuf	JSON
Binary(smaller sometimes)	Text
pre-defined schema	Free schema
No schema in data	Schema included in data
Better backward compatibility (but with proper usage)	not easy to break things
Typed	-
Computer readable	Human readable

JSON can be smaller

```
message Test1 {
   required int32 a = 1;
}
```

%Test1{a: -1}

08 255 …(9) 1

{“a” : -1}

Protobuf 11 bytes

JSON 8 bytes

Must-know for Protobuf

- Only add new fields
- Don’t change old fields(only if they’re not used anywhere or types are compatible, like int32 and int64). refer: updating
- There’s no way to distinguish zero-value or not setting(Protobuf 3)
- Always set 0 of Enum to a unused value(like UNKNOWN)

Generate code

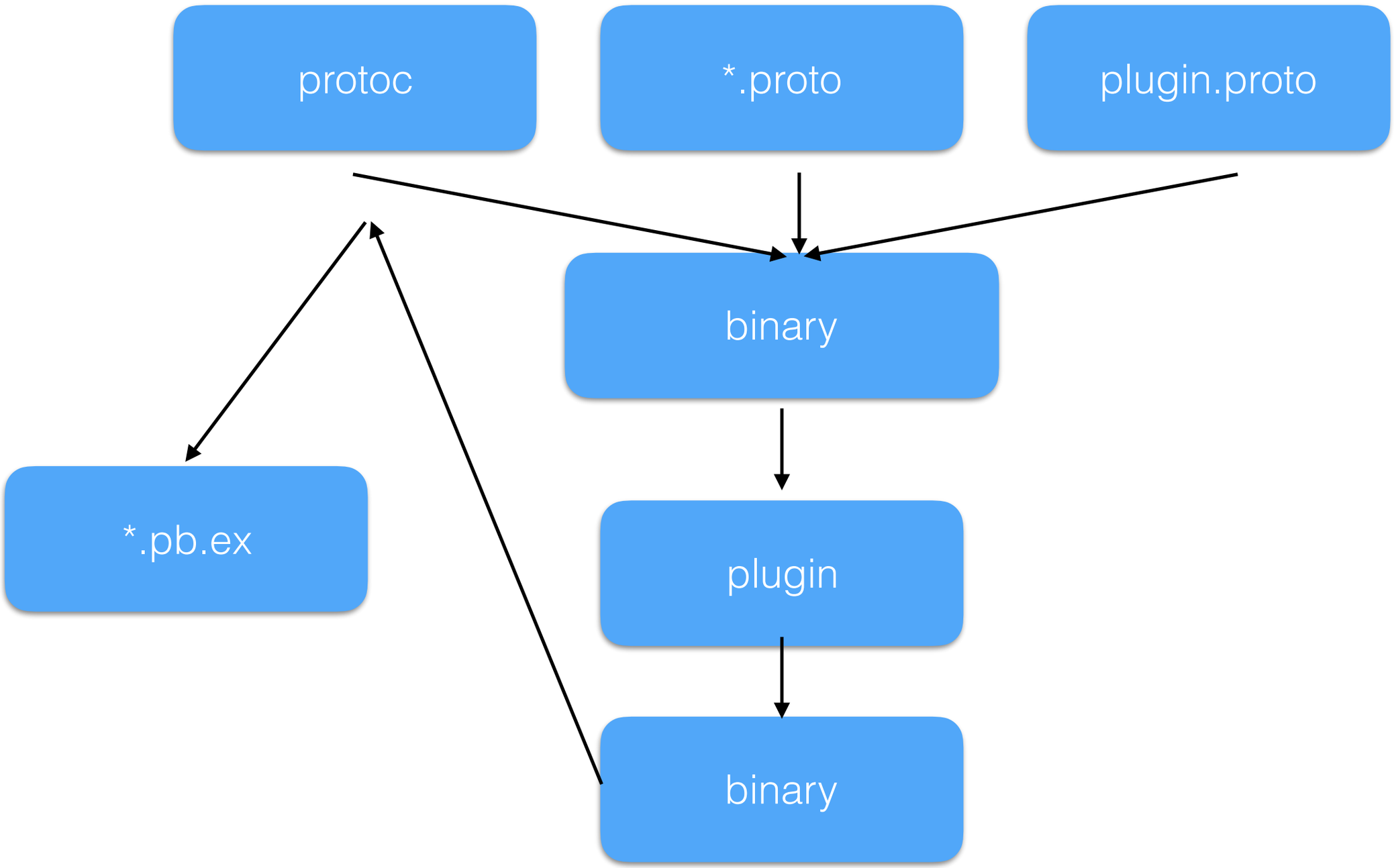
```
$ protoc -I=$SRC_DIR
--somelang_out=$DST_DIR
--plugin=./protoc-gen-somelang
$SRC_DIR/demo.proto
```

plugin can be inferred from -somelang_out to find protoc-gen-somelang in $PATH

Generate code

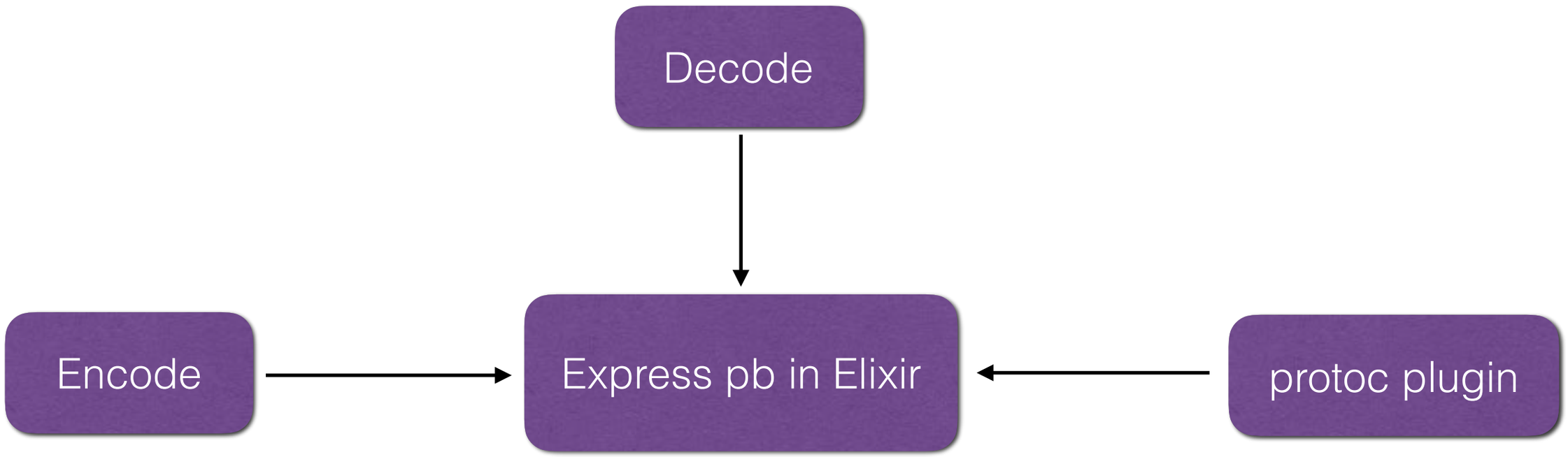
1. protoc(c++) parse your protobuf files, then generate encoded binary using plugin.proto(plugin.proto is defined by protoc)
2. protoc runs your executable plugin and send the encoded binary to your plugin via STDOUT
3. Your plugin generate the code, encode it using plugin.proto and write the binary to STDOUT

Generate code



How I implement Protobuf in Elixir

Components



DSL for a message

```elixir
defmodule Foo do
  defstruct [:a, :b, :c, :d, :e, :f, :g, :h, :i, :j, :k, :l, :m]

  field :a, 1, type: :int32
  field :b, 2, type: :fixed64
  field :c, 3, type: :string
  # 4 is skipped for testing
  field :d, 5, type: :float
  field :e, 6, type: Foo.Bar
  field :f, 7, type: :int32
  field :g, 8, repeated: true, type: :int32, packed: false
  field :h, 9, repeated: true, type: Foo.Bar
  field :i, 10, repeated: true, type: :int32
  field :j, 11, type: EnumFoo, enum: true
  field :k, 12, type: :bool
  field :l, 13, repeated: true, type: MapFoo, map: true
  field :m, 14, type: EnumFoo, enum: true
end
```

Where's field?

Where the magic happens

```elixir
defmodule Foo do
  use Protobuf, syntax: :proto3

  defstruct [:a, :b, :c, :d, :e, :f, :g, :h, :i, :j, :k, :l, :m]

  field :a, 1, type: :int32
  field :b, 2, type: :fixed64
  field :c, 3, type: :string
  # 4 is skipped for testing
  field :d, 5, type: :float
  field :e, 6, type: Foo.Bar
  field :f, 7, type: :int32
  field :g, 8, repeated: true, type: :int32, packed: false
  field :h, 9, repeated: true, type: Foo.Bar
  field :i, 10, repeated: true, type: :int32
  field :j, 11, type: EnumFoo, enum: true
  field :k, 12, type: :bool
  field :l, 13, repeated: true, type: MapFoo, map: true
  field :m, 14, type: EnumFoo, enum: true
end
```

import DSL

```elixir
defmodule Protobuf do
  defmacro __using__(opts) do
    quote do
      import Protobuf.DSL, only: [field: 3]
      Module.register_attribute(__MODULE__, :fields, accumulate: true)

      @options unquote(opts)
      @before_compile Protobuf.DSL
    end
  end
```

define __message_props__ function

```elixir
defmodule Protobuf.DSL do
  defmacro field(name, fnum, options) do
    quote do
      @fields {unquote(name), unquote(fnum), unquote(options)}
    end
  end

  defmacro __before_compile__(env) do
    fields = Module.get_attribute(env.module, :fields)
    msg_props = generate_msg_props(fields, oneofs, options)
    quote do
      def __message_props__ do
        unquote(Macro.escape(msg_props))
      end
```

Express pb in Elixir

```elixir
defmodule Protobuf.MessageProps do
  defstruct [
    ordered_tags: [],
    tags_map: %{},
    field_props: %{},
    repeated_fields: [],
    syntax: :proto2,
    oneof: [],

    enum?: false,
    oneof?: false,
    extendable?: false,
    map?: false,
  ]
end
```

```elixir
defmodule Protobuf.FieldProps do
  defstruct [
    fnum: nil,
    name: nil,
    name_atom: nil,
    wire_type: nil,
    type: nil,
    enum_type: nil,
    default: nil,
    oneof: nil,

    required?: false,
    optional?: false,
    repeated?: false,
    enum?: false,
    embedded?: false,
    packed?: false,
    map?: false,
  ]
end
```

By now, we can store protobuf info in a function of a module,
which we can use to decode, encode pb

Decoding logic

```elixir
@spec decode(binary, atom) :: any
def decode(data, module) when is_atom(module) do
  do_decode(data, module.__message_props__(), module.new)
end

@spec do_decode(binary, MessageProps.t, struct) :: any
defp do_decode(bin, props, msg) when is_binary(bin) and byte_size(bin) > 0 do
  {key, rest} = decode_varint(bin)
  tag = bsr(key, 3)
  wire_type = band(key, 7)
  case find_field(props, tag) do
    {:field_num, prop} ->
      case class_field(prop, wire_type) do
        type when type in [:normal, :embedded, :packed] ->
          {val, rest} = decode_type(type_to_decode(type, prop.type), wire_type, rest)
          new_msg = put_field(type, msg, prop, prop.name_atom, val)
          do_decode(rest, props, new_msg)
        end
      end
```

escript for building plugin

```elixir
defmodule Protobuf.Mixfile do
  use Mix.Project

  def project do
    [app: :protobuf,
     escript: escript(),
    ]
  end

  defp escript do
    [main_module: Protobuf.Protoc.CLI,
     name: "protoc-gen-elixir",
     app: nil]
  end
```

protoc plugin

```elixir
defmodule Protobuf.Protoc.CLI do
  def main(_) do
    # https://groups.google.com/forum/#!topic/elixir-lang-talk/T5enez_BBTI
    :io.setopts(:standard_io, encoding: :latin1)
    bin = IO.binread(:all)
    request = Protobuf.Decoder.decode(bin, Google_Protobuf_Compiler.CodeGeneratorRequest)
    ctx = %Protobuf.Protoc.Context{}
    ctx = parse_params(ctx, request.parameter)
    files = request.proto_file
      |> Enum.map(fn(desc) -> Protobuf.Protoc.Generator.generate(ctx, desc) end)
    response = Google_Protobuf_Compiler.CodeGeneratorResponse.new(file: files)
    IO.binwrite(Protobuf.Encoder.encode(response))
  end
```

A trick for generator

plugin.pb.ex for plugin.proto is needed for decoding STDOUT and encoding to STDOUT when generating Elixir code

But how to generate Elixir code for plugin.proto?

A trick for generator

Write plugin.pb.ex by hand at first😅

What I learned

- Macro of Elixir is powerful. Elixir is powerful
- Binary handling in Elixir is easy
- Keep macro simple
- Creating DSL is hard 🙈
- Encapsulate your structured data in struct(like MessageProps, FieldProps)
- Use functions and modules to keep your logic clear