

ZebraPack: Fast, friendly serialization

GolangDFW Meetup, 2017 February 16

Jason E. Aten, Ph.D.

Computer Scientist/Gopher

ZebraPack

- a data description language and serialization format. Like Gobs version 2.0.
- ZebraPack is a data definition language and serialization format. It removes gray areas from msgpack2 serialized data, and provides for declared schemas, sane data evolution, and more compact encoding.
- It does all this while maintaining the possibility of easy compatibility with all the dynamic languages that already have msgpack2 support.
- a day's work to adapt an existing language binding to read zebrapack: the schema are in msgpack2, and then one simply keeps a hashmap to translate between small integer \leftrightarrow field names/type.

motivation Why start with [msgpack2] (<http://msgpack.org>)?

- msgpack2 is simple, fast, and extremely portable.
- It has an implementation in every language you've heard of, and some you haven't (some 50 libraries are available).
- It has a well defined and short spec.
- msgpack2 is dynamic-language friendly because it is largely self-describing.

Problems with msgpack2

- poorly defined language binding
- weak support for data evolution
- insufficiently strong typing.

Problem example

- the widely emulated C-encoder for msgpack chooses to encode signed positive integers as unsigned integers.
- This causes crashes in readers who were expected a signed integer
- which they may have originated themselves in the original struct.
- the existing practice for msgpack2 language bindings allows the data types to change as they are read and re-serialized.
- Simple copying of a serialized struct can change the types of data from signed to unsigned.
- This is horrible.

Addressing the problems

- for language binding: strongly define the types of fields
- for efficiency and data evolution: adopt a new convention about how to encode the field names of structs.

Addressing the problems II

- for language binding: strongly define the types of fields
- for efficiency and data evolution: adopt a new convention about how to encode the field names of structs.
- Structs are encoded in msgpack2 using maps, as usual.
- maps that represent structs are now keyed by integers.
- Rather than have string keys
- these integers are associated with a field name and type in a (separable) schema.
- The schema is also defined and encoded in msgpack2.

zebrapack: the main idea

```
//given this definition, defined in Go:
type A struct {
    Name      string      `zid:"0"`
    Bday      time.Time   `zid:"1"`
    Phone     string      `zid:"2"`
    Sibbs     int         `zid:"3"`
    GPA       float64     `zid:"4" msg:",deprecated" // a deprecated
    Friend    bool        `zid:"5"`
}
```


zebrapack: the main idea 2

original(msgpack2) ->	schema(msgpack2)	+	each instance
-----	-----		-----
a := A{	zebra.StructT{		map{
"Name": "Atlanta",	0: {"Name", String},		0: "Atlan
"Bday": tm("1990-12-20"),	1: {"Bday", Timestamp},		1: "1990-
"Phone": "650-555-1212",	2: {"Phone", String},		2: "650-5
"Sibs": 3,	3: {"Sibs", Int64},		3: 3,
"GPA" : 3.95,	4: {"GPA", Float64},		4: 3.95,
"Friend":true,	5: {"Friend", Bool},		5: true,
}	}		}

Result

- resulting binary encoding is very similar in style to protobufs/Thrift/Capn'Proto.
- However it is much more friendly to other (dynamic) languages.
- Also it is screaming fast.

Benchmarking Reads

benchmark	iter	time
-----	----	-----
BenchmarkZebraPackUnmarshal-4	10000000	227
BenchmarkGencodeUnmarshal-4	10000000	229
BenchmarkFlatBuffersUnmarshal-4	10000000	232
BenchmarkGogoprotobufUnmarshal-4	10000000	232
BenchmarkCapNProtoUnmarshal-4	10000000	258
BenchmarkMsgpackUnmarshal-4	5000000	296
BenchmarkGoprotobufUnmarshal-4	2000000	688
BenchmarkProtobufUnmarshal-4	2000000	707
BenchmarkGobUnmarshal-4	2000000	886
BenchmarkHproseUnmarshal-4	1000000	1045
BenchmarkCapNProto2Unmarshal-4	1000000	1359
BenchmarkXdrUnmarshal-4	1000000	1659
BenchmarkBinaryUnmarshal-4	1000000	1907
BenchmarkVmhailencoMsgpackUnmarshal-4	1000000	2085
BenchmarkUgorjiCodecMsgpackUnmarshal-4	500000	2620
BenchmarkUgorjiCodecBincUnmarshal-4	500000	2795
BenchmarkSerealUnmarshal-4	500000	3271
BenchmarkJsonUnmarshal-4	200000	5576

Benchmarking Writes

benchmark	iter	time
-----	----	-----
BenchmarkZebraPackMarshal-4	10000000	115
BenchmarkGogoprotobufMarshal-4	10000000	148
BenchmarkMsgpackMarshal-4	10000000	161
BenchmarkGencodeMarshal-4	10000000	176
BenchmarkFlatBufferMarshal-4	5000000	347
BenchmarkCapNProtoMarshal-4	3000000	506
BenchmarkGoprotobufMarshal-4	3000000	617
BenchmarkGobMarshal-4	2000000	887
BenchmarkProtobufMarshal-4	2000000	912
BenchmarkHproseMarshal-4	1000000	1052
BenchmarkCapNProto2Marshal-4	1000000	1214
BenchmarkBinaryMarshal-4	1000000	1427
BenchmarkVmhailencoMsgpackMarshal-4	1000000	1772
BenchmarkXdrMarshal-4	1000000	1802
BenchmarkJsonMarshal-4	1000000	2500
BenchmarkUgorjiCodecBincMarshal-4	500000	2514
BenchmarkSerealMarshal-4	500000	2729
BenchmarkUgorjiCodecMsgpackMarshal-4	500000	3274

Advantages and advances: pulling the best ideas from other formats

- Once we have a schema, we can be very strongly typed, and be very efficient.
- We borrow the idea of field deprecation from FlatBuffers
- For conflicting update detection, we use CapnProto's field numbering discipline
- support for the `omitempty` tag
- in ZebraPack, all fields are `omitempty`
- If they are empty they won't be serialized on the wire. Like FlatBuffers and Protobufs, this enables one to define a very large schema of possibilities, and then only transmit a very small (efficient) portion that is currently relevant over the wire.

Credit to Philip Hofer

Full credit: the ZebraPack code descends from the fantastic msgpack2 code generator <https://github.com/tinylib/msgp> by Philip Hofer.

deprecating fields

```
type A struct {  
    Name      string      `zid:"0"`  
    Bday      time.Time   `zid:"1"`  
    Phone     string      `zid:"2"`  
    Sibbs     int          `zid:"3"`  
    GPA       float64     `zid:"4" msg:",deprecated" // a deprecated  
    Friend    bool        `zid:"5"`  
}
```

deprecating fields II

```
type A struct {  
    Name      string      `zid:"0"`  
    Bday      time.Time   `zid:"1"`  
    Phone     string      `zid:"2"`  
    Sibbs     int           `zid:"3"`  
    GPA       struct{}    `zid:"4" msg:",deprecated" // a deprecated`  
    Friend    bool         `zid:"5"`  
}
```


Safety rules during data evolution

- Rules for safe data changes: To preserve forwards/backwards compatible changes, you must **never remove a field** from a struct, once that field has been defined and used.
- In the example above, the `zid: "4"` tag must stay in place, to prevent someone else from ever using 4 again.
- This allows sane data forward evolution, without tears, fears, or crashing of servers.
- The fact that `struct{}` fields take up no space also means that there is no need to worry about loss of performance when deprecating.
- We retain all fields ever used for their zebra ids, and the compiled Go code wastes no extra space for the deprecated fields.

schema details

- Precisely defined format
- see the repo for examples and details.
- <https://github.com/glycerine/zebrapack>

`zebrapack -msgp` as a msgpack2 code-generator

``msg:"",omitempty`` tags on struct fields

If you're using ``zebrapack -msgp`` to generate msgpack2 serialization code, then you can use the `omitempty` tag on your struct fields.

In the following example,

```
type Hedgehog struct {  
  Furriness string msg: "",omitempty"  
}
```

If Furriness is the empty string, the field will not be serialized, thus saving the space of the field name on the wire.

It is safe to re-use structs even with ``omitempty``

``addzid`` utility

The `addzid` utility (in the `cmd/addzid` subdir) can help you get started. Running ``addzid mysource.go`` on a `.go` source file

will add the `zid:"0"...` fields automatically. This makes adding ZebraPack serialization to existing Go projects easy.

See <https://github.com/glycerine/zebrapack/blob/master/cmd/addzid/README.md> for more detail.

What's next. New ideas.

- microschema
- declare how many follow-on objects a schema is good for

Thank you

Jason E. Aten, Ph.D.

Computer Scientist/Gopher

j.e.aten@gmail.com (<mailto:j.e.aten@gmail.com>)