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.
- remove gray areas from the language bindings. Provides for declared schemas, sane data evolution, and more compact encoding.
- maintain 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 <-> field names/type.
- MIT licensed. https://github.com/glycerine/zebrapack

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"`
  Sibs     int         `zid:"3"`
  GPA      float64     `zid:"4" msg:",deprecated"` // a deprecated field.
  Friend   bool        `zid:"5"`
}
```

zebrapack: the main idea 2

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

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 simple and short spec.
- msgpack2 is dynamic-language friendly because it is largely self-describing.
- most significantly: the existing library github.com/tinylib/msgp is extremely well tuned, and generates Go bindings by reading your Go source.

Problems with msgpack2

- poorly defined language binding (signed/unsigned/bitwidth of integer?)
- a.k.a. insufficiently strong typing.
- weak support for data evolution. i.e. no conflict detection, no omitempty support from the prior libraries => they crash on unexpected fields.

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.
- simply parse from the Go source. No separate IDL, your Go code is your one source of truth.
- For efficiency and data evolution: adopt a new convention about how to encode the field names of structs. Use small integer fields.

Addressing the problems II

- Structs are encoded in msgpack2 using maps, as usual.
- maps that represent structs are now keyed by integers.
- Rather than strings as keys
- these integers are associated with a field name and type in a (separable) schema.
- The schema is also defined and encoded in msgpack2.

Result

- resulting binary encoding is very similar in style to protobufs/Thrift/Capn'Proto.
- However it is much more friendly to dynamic languages; e.g. R, python, zygo
- Also it is screaming fast.

Benchmarking Reads

benchmark	iter	time/iter	bytes alloc	alloc
BenchmarkZebraPackUnmarshal-4	10000000	227 ns/op	0 B/op	0 a
BenchmarkGencodeUnmarshal-4	10000000	229 ns/op	112 B/op	3 a
BenchmarkFlatBuffersUnmarshal-4	10000000	232 ns/op	32 B/op	2 a
BenchmarkGogoprotobufUnmarshal-4	10000000	232 ns/op	96 B/op	3 a
BenchmarkCapNProtoUnmarshal-4	10000000	258 ns/op	0 B/op	0 a
BenchmarkMsgpUnmarshal-4	5000000	296 ns/op	32 B/op	2 a
BenchmarkGoprotobufUnmarshal-4	2000000	688 ns/op	432 B/op	9 a
BenchmarkProtobufUnmarshal-4	2000000	707 ns/op	192 B/op	10 a
BenchmarkGobUnmarshal-4	2000000	886 ns/op	112 B/op	3 a
BenchmarkHproseUnmarshal-4	1000000	1045 ns/op	320 B/op	10 a
BenchmarkCapNProto2Unmarshal-4	1000000	1359 ns/op	608 B/op	12 a
BenchmarkXdrUnmarshal-4	1000000	1659 ns/op	239 B/op	11 a
BenchmarkBinaryUnmarshal-4	1000000	1907 ns/op	336 B/op	22 a
BenchmarkVmihailencoMsgpackUnmarshal-4	1000000	2085 ns/op	384 B/op	13 a
BenchmarkUgorjiCodecMsgpackUnmarshal-4	500000	2620 ns/op	3008 B/op	6 a
BenchmarkUgorjiCodecBincUnmarshal-4	500000	2795 ns/op	3168 B/op	9 a
BenchmarkSerealUnmarshal-4	500000	3271 ns/op	1008 B/op	34 a
BenchmarkJsonUnmarshal-4	200000	5576 ns/op	495 B/op	8 a

Benchmarking Writes

benchmark	iter	time/iter	bytes alloc	alloc
BenchmarkZebraPackMarshal-4	10000000	115 ns/op	0 B/op	0 a
BenchmarkGogoprotobufMarshal-4	10000000	148 ns/op	64 B/op	1 a
BenchmarkMsgpMarshal-4	10000000	161 ns/op	128 B/op	1 a
BenchmarkGencodeMarshal-4	10000000	176 ns/op	80 B/op	2 a
BenchmarkFlatBufferMarshal-4	5000000	347 ns/op	0 B/op	0 a
BenchmarkCapNProtoMarshal-4	3000000	506 ns/op	56 B/op	2 a
BenchmarkGoprotobufMarshal-4	3000000	617 ns/op	312 B/op	4 a
BenchmarkGobMarshal-4	2000000	887 ns/op	48 B/op	2 a
BenchmarkProtobufMarshal-4	2000000	912 ns/op	200 B/op	7 a
BenchmarkHproseMarshal-4	1000000	1052 ns/op	473 B/op	8 a
BenchmarkCapNProto2Marshal-4	1000000	1214 ns/op	436 B/op	7 a
BenchmarkBinaryMarshal-4	1000000	1427 ns/op	256 B/op	16 a
BenchmarkVmihailencoMsgpackMarshal-4	1000000	1772 ns/op	368 B/op	6 a
BenchmarkXdrMarshal-4	1000000	1802 ns/op	455 B/op	20 a
BenchmarkJsonMarshal-4	1000000	2500 ns/op	536 B/op	6 a
BenchmarkUgorjiCodecBincMarshal-4	500000	2514 ns/op	2784 B/op	8 a
BenchmarkSerealMarshal-4	500000	2729 ns/op	912 B/op	21 a
BenchmarkUgorjiCodecMsgpackMarshal-4	500000	3274 ns/op	2752 B/op	8 a

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

```
(contiguous integers from 0..N-1).
```

- 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"`
  Sibs     int         `zid:"3"`
  GPA      float64     `zid:"4" msg:",deprecated"` // a deprecated field.
  Friend   bool        `zid:"5"`
}
```

deprecating fields II

```
type A struct {
  Name     string      `zid:"0"`
  Bday     time.Time   `zid:"1"`
  Phone    string      `zid:"2"`
  Sibs     int         `zid:"3"`
  GPA      struct{}    `zid:"4" msg:",deprecated"` // a deprecated field should have its type changed to
  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
- handle cycles in an object graph, by detecting

```
(large) repeated references and encoding pointers as object IDs.
```

- your idea here.
- (One idea from meetup: optional bitmap to designate set/unset field, as in flatbuffers).

Thank you

Jason E. Aten, Ph.D.
Computer Scientist/Gopher
j.e.aten@gmail.com (mailto:j.e.aten@gmail.com)