From Bytes to Go and Back

Encoding and Decoding deep-dive

Egon Elbre
Golang Estonia
2023-01-11-T19:00:00
egonelbre.com

From Bytes to Go and Back

Most network and disk IO requires converting from `[]byte` to some structs and back to `[]byte`. Let's take a deep dive into different ways of writing encoding and decoding libraries.

Code can be found at github.com/golang-estonia/structs-to-bytes (https://github.com/golang-estonia/structs-to-bytes).

Non-Topics

- What should be encoded?
- Forwards/Backwards compatibility concerns
- High-Performance details
- Compression
- Integer encoding formats
- Is JSON or Protobuf the best format?
- Protocols

Topics

1. Standard Library
2. Call-Based Marshaling
3. Composed Marshaling
4. Reflection Based Marshaling
5. Schema Based Approaches
6. Unsafe

Standard Library

encoding/binary

Easiest way to encode an integer to `[]byte`.

- `binary.BigEndian`
- `binary.LittleEndian`

```go
func Example() {
    data := make([]byte, 8)
    binary.LittleEndian.PutUint64(data, 123456789)
    fmt.Println(data)
    value := binary.LittleEndian.Uint64(data)
    fmt.Println(value)

    // Output:
    // [21 205 91 7 0 0 0 0]
    // 123456789
}
```

encoding

There are quite a few standard packages:

- **encoding/json**
- encoding/xml
- encoding/gob
- encoding/asn1

encoding/json: types

There are quite a few different options depending on the specific encoding package. We won't cover them as these aren't that interesting.

```
type Person struct {
    Name     string `json:"name"`
    Email    string `json:"email"`
    Address  string `json:"address,omitempty"`
    Password string `json:"-"`
}
```

encoding/json: Encode

You've probably used this already, but for completeness:

```
func ExampleEncode() {
    data, err := json.Marshal(Person{
        Name:     "John",
        Email:    "john@email.test",
        Password: "hunter2",
    })
    if err != nil {
        panic(err)
    }
    fmt.Println(string(data))

    // Output: {"name":"John","email":"john@email.test"}
}
```

encoding/json: Decode

Ditto:

```
func ExampleDecode() {
    var person Person
    data := []byte(`{"name":"John","email":"john@email.test"}`)
    err := json.Unmarshal(data, &person)
    if err != nil {
        panic(err)
    }
    fmt.Printf("%#v\n", person)

    // Output: main.Person{Name:"John", Email:"john@email.test", Address:"", Password:""}
}
```

encoding/json: inline Decode

Less common, but can be useful for API requests:

```
func ExampleInlineDecode() {
    var person struct {
        Name  string
        Email string
    }
    data := []byte(`{"name":"John","email":"john@email.test"}`)
    err := json.Unmarshal(data, &person)
    if err != nil {
        panic(err)
    }
    fmt.Printf("%+v\n", person)

    // Output: {Name:John Email:john@email.test}
}
```

encoding/json: Encoder

The packages also support streaming, reading from `io.Reader` and writing to `io.Writer`.

```go
func ExampleEncoder() {
    var buf bytes.Buffer
    enc := json.NewEncoder(&buf)
    _ = enc.Encode(Person{Name: "Alice"})
    _ = enc.Encode(Person{Name: "Bob"})
    _ = enc.Encode(Person{Name: "Charlie"})
    fmt.Println(buf.String())

    // Output:
    // {"name":"Alice","email":""}
    // {"name":"Bob","email":""}
    // {"name":"Charlie","email":""}
}
```

TextMarshaler and BinaryMarshaler

encoding package contains definitions:

```
// Default text encoding for different encoding packages.
type TextMarshaler interface {
    MarshalText() (text []byte, err error)
}
type TextUnmarshaler interface {
    UnmarshalText(text []byte) error
}

// Default binary encoding for different encoding packages.
type BinaryMarshaler interface {
    MarshalBinary() (data []byte, err error)
}
type BinaryUnmarshaler interface {
    UnmarshalBinary(data []byte) error
}
```

TextMarshaler: implementation

We can write a custom point serialization:

```
type Point struct {
    X, Y int32
}

// Custom encoding for `encoding/json`, `encoding/xml` and `encoding/gob`.
func (p Point) MarshalText() ([]byte, error) {
    return []byte(fmt.Sprintf("x%d y%d", p.X, p.Y)), nil
}

// Custom encoding for `encoding/gob`.
func (p Point) MarshalBinary() ([]byte, error) {
    var data [8]byte
    binary.BigEndian.PutUint32(data[0:4], uint32(p.X))
    binary.BigEndian.PutUint32(data[4:8], uint32(p.Y))
    return data[:], nil
}
```

TextMarshaler: usage

Use as a regular field and the appropriate `MarshalText` will be called and also escaped as necessary.

```
func ExamplePoint() {
    type Drone struct {
        Name     string
        Location Point
    }

    drone := Drone{
        Name:     "Johnny",
        Location: Point{X: 100, Y: 100},
    }

    data, err := json.Marshal(drone)
    if err != nil {
        panic(err)
    }

    fmt.Println(string(data))
    // Output: {"Name":"Johnny","Location":"x100 y100"}
}
```

Unmarshaling: implementation

We'll skip the usage, as it's staight-forward

```
func (p *Point) UnmarshalBinary(data []byte) error {
    if len(data) != 8 {
        return fmt.Errorf("expected length 8, but got %d", len(data))
    }
    data = data[0:8]
    p.X = int32(binary.BigEndian.Uint32(data[0:4]))
    p.Y = int32(binary.BigEndian.Uint32(data[4:8]))
    return nil
}
```

json.Marshaler and json.Unmarshaler

There are also encoding specific interfaces:

```
type Marshaler interface {
    MarshalJSON() ([]byte, error)
}
type Unmarshaler interface {
    UnmarshalJSON([]byte) error
}
```

json.Marshaler: example

Remember the `MarshalJSON` must output valid JSON:

```
type QuotedPerson struct {
    Name string
}

func (p QuotedPerson) MarshalJSON() ([]byte, error) {
    qname, err := json.Marshal("!" + p.Name + "!")
    if err != nil {
        return nil, err
    }

    return []byte(`{"quoted": ` + string(qname) + `}`), nil
}

func ExampleQuotedPerson() {
    data, err := json.Marshal(QuotedPerson{Name: "Hello"})
    if err != nil {
        panic(err)
    }
    fmt.Println(string(data))
    // Output: {"quoted":"!Hello!"}
}
```

json.Marshaler: temporary

It's also possible to use a different type when you need to do data-munging:

```go
type TempPerson struct {
    Name string
}

func (p TempPerson) MarshalJSON() ([]byte, error) {
    var temp struct {
        Quoted string `json:"quoted"`
    }
    temp.Quoted = "!" + p.Name + "!"
    return json.Marshal(temp)
}

func ExampleTempPerson() {
    data, err := json.Marshal(TempPerson{Name: "Hello"})
    if err != nil {
        panic(err)
    }
    fmt.Println(string(data))
    // Output: {"quoted":"!Hello!"}
}
```

json.Marshaler

Also look at the examples in encoding/json (https://pkg.go.dev/encoding/json) .

Call Based

Call Based: Notes

This is pretty much `json.Marshaler` and `json.Unmarshaler` wired together manually.

Notes:

- The most straightforward code
- Easy to make typos
- Potential binary bloat
- More difficult to optimize after-wards.

Composed Marshaling

Composed Marshaling: Notes

Usually a custom solution for a given problem.

Notes:

- Useful when you need to support multiple encodings
- Helps to reduce boiler-plate and is declarative
- Not as annoying to write as reflection code
- Potential binary bloat

Reflection Based

Reflection Based: Notes

`encoding/json` and et al. are good examples of this approach.

Notes:

- Decent middle-of-the-road approach
- Just having field tags is usually sufficient
- Reflection based code is annoying
- Usually minimal binary bloat

Schema Based

Schema Based: Notes

Well known example is protobuf.

Notes:

- Useful when you need to support multiple languages
- Useful when you have lots of message types in an specification (e.g. DNS)
- Allows to avoid writing boiler-plate
- Great flexibility in capabilities
- Less convenient to use
- Binary bloat is a concern

Unsafe

Unsafe: Notes

Use only when absolutely there's no other way to achieve what you need.

Notes:

- Can be very performant
- Doesn't play nice with pointers (that includes slices and strings)
- Older ARM processors don't support "byte" addressing
- Take note of computer endianess
- Take note of `int` size
- See all the warnings in https://pkg.go.dev/unsafe
- Mistakes can lead to huge problems

General recommendation, don't use it.

Unsafe, but friendlier

It's possible to write the `unsafe` usage in a friendlier manner.

But you still shouldn't use it.

Thank you

Egon Elbre
Golang Estonia
2023-01-11-T19:00:00
egonelbre.com