JSON and struct composition in Go

10 Sep 2014

Say you are decoding a JSON object into a Go struct. It comes from a service that is not under your control, so you cannot do much about the schema. However, you want to encode it differently.

You could go wild with `json.Marshaler`, but it has some drawbacks:

- **complexity**: adds lots of extra code for big structs
- **memory usage**: must be careful not to do needless allocations

To be fair, in most cases you can avoid allocations in your `MarshalJSON()`, but that may lead to even more complexity, which now sits in your code base (instead of `encoding/json`), so it's your job to unit test it. And that's even more boring code to write.

Here are a few tricks to be used with big structs.

Omitting fields

Let's say you have this struct:

```
type User struct {
    Email    string `json:"email"`
    Password string `json:"password"`
    // many more fields…
}
```

What you want is to encode `User`, but without the `password` field. A simple way to do that with struct composition would be to wrap it in another struct:

```
type omit *struct{}

type PublicUser struct {
    *User
    Password omit `json:"password,omitempty"`
}

// when you want to encode your user:
json.Marshal(PublicUser{
    User: user,
})
```

The trick here is that we never set the `Password` property of the `PublicUser`, and since it is a pointer type, it will default to `nil`, and it will be omitted (because of `omitempty`).

Note that there's no need to declare the `omit` type, we could have simply used `*struct{}` or even `bool` or `int`, but declaring the type makes it explicit that we're omitting that field from the output. Which built-in type we use does not matter as long as it has a zero value that is recognised by the `omitempty` tag.

We could have used only anonymous values:

```
json.Marshal(struct {
    *User
    Password bool `json:"password,omitempty"`
}{
    User: user,
})
```

Try it in the playground.

Also note that we only include a pointer to the original `User` struct in our wrapper struct. This indirection avoids having to allocate a new copy of `User`.

Adding extra fields

Adding fields is even simpler than omitting. To continue our previous example, let's hide the password but expose an additional `token` property:

```
type omit *struct{}
type PublicUser struct {
    *User
    Token    string `json:"token"`
    Password omit   `json:"password,omitempty"`
}

json.Marshal(PublicUser{
    User:  user,
    Token: token,
})
```

Try it in the playground.

Composing structs

This is handy when combining data coming from different services. For example, here's a `BlogPost` struct that also contains analytics data:

```
type BlogPost struct {
    URL   string `json:"url"`
    Title string `json:"title"`
}

type Analytics struct {
    Visitors  int `json:"visitors"`
    PageViews int `json:"page_views"`
}

json.Marshal(struct{
    *BlogPost
    *Analytics
}{post, analytics})
```

Try it in the playground.

Splitting objects

This is the opposite of composing structs. Just like when encoding a combined structs, we can decode into a combined struct and use the values separately:

```
json.Unmarshal([]byte(`{
  "url": "attila@attilaolah.eu",
  "title": "Attila's Blog",
  "visitors": 6,
  "page_views": 14
}`), &struct {
  *BlogPost
  *Analytics
}{&post, &analytics})
```

Try it in the playground.

Renaming fields

This one is a combination of removing fields and adding extra fields: we simply remove the field and add it with a different `json:` tag. This can be done with pointer indirection to avoid allocating memory, although for small data types the indirection overhead can cost the same amount of memory as it would cost to create a copy of the field, plus the runtime overhead.

Here is an example where we rename two struct fields, using indirection for the nested struct and copying the integer:

```
type CacheItem struct {
    Key    string `json:"key"`
    MaxAge int    `json:"cacheAge"`
    Value  Value  `json:"cacheValue"`
}

json.Marshal(struct{
    *CacheItem

    // Omit bad keys
    OmitMaxAge omit `json:"cacheAge,omitempty"`
    OmitValue  omit `json:"cacheValue,omitempty"`

    // Add nice keys
    MaxAge int    `json:"max_age"`
    Value  *Value `json:"value"`
}{
    CacheItem: item,

    // Set the int by value:
    MaxAge: item.MaxAge,

    // Set the nested struct by reference, avoid making a copy:
    Value: &item.Value,
})
```

Try it in the playground.

Note that this is only practical when you want to rename one or two fields in a big struct. When renaming all fields, it is often simpler (and cleaner) to just create a new object altogether (i.e. a *serialiser*) and avoid the struct composition.

Related posts:

- JSON decoding in Go