Programming in Go

Matt Holiday
Christmas 2020

Reflection

Type assertion

"`interface{}` says nothing" since it has no methods

It's a "generic" thing, but sometimes we need its "real" type

We can extract a specific type with a *type assertion* (a/k/a "downcasting")

This has the form `value.(T)` for some type `T`

```
var w io.Writer = os.Stdout

f := w.(*os.File)       // success: f == os.Stdout

c := w.(*bytes.Buffer)  // panic: interface holds *os.File, not *bytes.Buffer
```

Type assertion

If we use the two-result version, we can avoid `panic`

```
var w io.Writer = os.Stdout

f, ok := w.(*os.File)       // success:  ok, f == os.Stdout

b, ok := w.(*bytes.Buffer)  // failure: !ok, b == nil
```

Switching on type

We can also use type assertion in a `switch` statement (matching a *type* not a *value*)

```
func Println(args ...interface{}) {
    buf := make([]byte, 0, 80)

    for arg := range args {
        switch a := arg.(type) {
            case string:                              // concrete type
                buf = append(buf, a...)
            case Stringer:                            // interface
                buf = append(buf, a.String()...)
            . . .
        }
```

Here the switch variable a has a specific type if the case has a *single* type

Deep equality

We can use the `reflect` package in UTs to check equality

```
want := struct{
    a: "a string",
    b: []int{1, 2, 3}     // not comparable with ==
}

got := gotGetIt( . . . )

if !reflect.DeepEqual(got, want) {
    t.Errorf("bad response: got=%#v, want=%#v", got, want)
}
```

You can use `github.com/kylelemon/godebug/pretty` to show a deep diff

Hard JSON

Not all JSON messages are well-behaved

What if some keys depend on others in the message?

```
{
  "item": "album",
  "album": {"title": "Dark Side of the Moon"}
}
```

```
{
  "item": "song",
  "song": {"title": "Bella Donna", "artist": "Stevie Nicks"}
}
```

Custom JSON decoding

We'll make a wrapper and a custom decoder

```
type response struct {
    Item   string `json:"item"`
    Album  string
    Title  string
    Artist string
}

type respWrapper struct {
    response
}
```

We need `respWrapper` because it must have a *separate* unmarshal method from the `response` type (see below)

Custom JSON decoding

```
func (r *respWrapper) UnmarshalJSON(b []byte) (err error) {
    var raw map[string]interface{}

    err = json.Unmarshal(b, &r.response)  // ignore error handling
    err = json.Unmarshal(b, &raw)

    switch r.Item {
    case "album":
        inner, ok := raw["album"].(map[string]interface{})

        if ok {
            if album, ok := inner["title"].(string); ok {
                r.Album = album
            }
        }

        . . .
```

Custom JSON decoding

```
    case "song":
        inner, ok := raw["song"].(map[string]interface{})

        if ok {
            if title, ok := inner["title"].(string); ok {
                r.Title = title
            }

            if artist, ok := inner["artist"].(string); ok {
                r.Artist = artist
            }
        }
    }

    return err
}
```

Custom JSON decoding

```
func main() {
    var resp1, resp2 respWrapper
    var err error

    if err = json.Unmarshal([]byte(j1), &resp1); err != nil {
        log.Fatal(err)
    }

    fmt.Printf("%#v\n", resp1.response)

    if err = json.Unmarshal([]byte(j2), &resp2); err != nil {
        log.Fatal(err)
    }

    fmt.Printf("%#v\n", resp2.response)
}
```

Custom JSON decoding

```
var j1 = `{
  "item": "album",
  "album": {"title": "Dark Side of the Moon"}
}`

var j2 = `{
  "item": "song",
  "song": {"title": "Bella Donna", "artist": "Stevie Nicks"}
}`



// main.response{Item:"album", Album:"Dark Side of the Moon",
//               Title:"", Artist:""}
//
// main.response{Item:"song", Album:"", Title:"Bella Donna",
//               Artist:"Stevie Nicks"}
```

Testing JSON

We want to know if a known fragment of JSON is contained in a larger unknown piece

```
{"id": "Z"} in? {"id": "Z", "part": "fizgig", "qty": 2}
```

All done with reflection from a generic map

```
func matchNum(key string, exp float64, data map[string]interface{}) bool {
    if v, ok := data[key]; ok {
        if val, ok := v.(float64); ok && val == exp {
            return true
        }
    }

    return false
}
```

Testing JSON

```
func matchString(key, exp string, data map[string]interface{}) bool {
    // is it in the map?

    if v, ok := data[key]; ok {
        // is it a string, and does it match?

        if val, ok := v.(string); ok && strings.EqualFold(val, exp) {
            return true
        }
    }

    return false
}
```

Testing JSON

```
func contains(exp, data map[string]interface{}) error {
    for k, v := range exp {
        switch x := v.(type) {

        case float64:
            if !matchNum(k, x, data) {
                return fmt.Errorf("%s unmatched (%d)", k, int(x))
            }

        case string:
            if !matchString(k, x, data) {
                return fmt.Errorf("%s unmatched (%s)", k, x)
            }

        . . .
```

Testing JSON

```
        . . .

        case map[string]interface{}:
            if val, ok := data[k]; !ok {
                return fmt.Errorf("%s missing in data", k)
            } else if unk, ok := val.(map[string]interface{}); ok {
                if err := contains(x, unk); err != nil {
                    return fmt.Errorf("%s unmatched (%+v): %s", k, x, err)
                }
            } else {
                return fmt.Errorf("%s wrong in data (%#v)", k, val)
            }
        }
    }

    return nil
}
```

Testing JSON

```
func CheckData(want, got []byte) error {
    var w, g map[string]interface{}

    if err := json.Unmarshal(want, &w); err != nil {
        return err
    }

    if err := json.Unmarshal(got, &g); err != nil {
        return err
    }

    return contains(w, g)
}
```

Testing JSON

Run the tests and analyze the code coverage

```
// go test -v
// go test ./... -cover
// go test ./... -coverprofile=c.out -covermode=count
// go tool cover -html=c.out

var unknown = `{
        "id": 1,
        "name": "bob",
        "addr": {
            "street": "Lazy Lane",
            "city": "Exit",
            "zip": "99999"
        },
        "extra": 21.1
    }`
```

Testing JSON

```
func TestContains(t *testing.T) {
    var known = []string{
        `{"id": 1}`,
        `{"extra": 21.1}`,
        `{"name": "bob"}`,
        `{"addr": {"street": "Lazy Lane", "city": "Exit"}}`,
    }

    for _, k := range known {
        if err := CheckData(k, []byte(unknown)); err != nil {
            t.Errorf("invalid: %s (%s)\n", k, err)
        }
    }
}
```

Testing JSON

```
func TestNotContains(t *testing.T) {
    var known = []string{
        `{"id": 2}`,
        `{"pid": 2}`,
        `{"name": "bobby"}`,
        `{"first": "bob"}`,
        `{"addr": {"street": "Lazy Lane", "city": "Alpha"}}`,
    }

    for _, k := range known {
        if err := CheckData(k, []byte(unknown)); err == nil {
            t.Errorf("false positive: %s\n", k)
        } else {
            t.Log(err)
        }
    }
}
```

Testing JSON

`go test` has options to help visualize code coverage