Programming in Go

Matt Holiday
Christmas 2020

Structs

Structs

We've already seen a couple of aggregate types:

- slices & arrays group a sequence of the same type
- maps use one type to index a collection of another type

A `struct` is an aggregate of possibly disparate types

```
type Employee struct {
    Name   string
    Number int
    Boss   *Employee
    Hired  time.Time
}
```

Notice that we can have a pointer to the type we're defining

Structs

In other languages it's a "record" (using database terminology)

It's parts are called "fields" and each must have a unique name

Access to the fields is with "dot" notation

```
employees := make(map[string]*Employee)

var matt = Employee{
    Name:   "Matt",
    Number: 72,
    Boss:   employees["Lamine"],
    Hired:  time.Date(2017, 12, 9, 16, 30, 0, 0, time.UTC),
}

employees[matt.Name] = &matt
```

Struct initialization

With names, selected fields can be initialized

The others default to “zero”

```
employees := make(map[string]*Employee)

var matt = Employee{
    Name:   "Matt",
    Number: 72,
    Hired:  time.Date(2017, 12, 9, 16, 30, 0, 0, time.UTC),
}

employees[matt.Name] = &matt
```

Here `matt.Boss` is left to the default value `nil`

Anonymous Structs

Anonymous structs are possible:

```
var album = struct {
    title  string
    artist string
    year   int,
    copies int,
}{
    "The White Album",
    "The Beatles",
    1968,
    10000000,
}
```

Initialization can be done without names by setting all fields in the correct order

Anonymous Structs

But assignment can be very inconvenient:

```
var s1 struct {
    A int
    B string
}

func main() {
    s1 = struct{A int; B string}{1, "a"}

    fmt.Println(s1)
}
```

Struct compatibility

But assignment can be very inconvenient:

```
var s1 struct {
    A int
    B string
}

var s2 struct {
    A int
    B string
}

func main() {
    s1 = struct{A int; B string}{1, "a"}
    s2 = s1
    fmt.Println(s1, s2)
}
```

Struct compatibility

Two `struct` types are the compatible if

- the fields have the same types and names
- in the same order
- and the same tags (*)

A `struct` may be copied or passed as a parameter in its entirety

A `struct` is comparable if all its fields are comparable

The zero value for a `struct` is “zero” for each field in turn

Struct compatibility

Anonymous structs are compatible if they follow the rules:

```
func main() {
    v1 := struct {
        X int `json:"foo"`
    }{1}

    v2 := struct {
        X int `json:"foo"`
    }{2}

    v1 = v2
    fmt.Println(v1)     // prints {2}
}
```

Struct compatibility

Types with different *user-declared* names are never compatible:

```
type T1 struct {
    X int `json:"foo"`
}

type T2 struct {
    X int `json:"foo"`
}

func main() {
    v1 := T1{1}
    v2 := T2{2}

    v1 = v2              // TYPE MISMATCH
    fmt.Println(v1)
}
```

Struct compatibility

Named struct types are *convertible* if they are compatible:

```
type T1 struct {
    X int `json:"foo"`
}

type T2 struct {
    X int `json:"foo"`
}

func main() {
    v1 := T1{1}
    v2 := T2{2}

    v1 = T1(v2)          // type conversion
    fmt.Println(v1)      // prints {2}
}
```

Struct compatibility

From Go 1.8 tag differences don't prevent type conversions:

```
type T1 struct {
    X int `json:"foo"`
}

type T2 struct {
    X int `json:"bar"`  // NOTE difference
}

func main() {
    v1 := T1{1}
    v2 := T2{2}

    v1 = T1(v2)          // type conversion
    fmt.Println(v1)      // prints {2}
}
```

Passing structs

Structs are passed by value unless a pointer is used

```
var white album

func soldAnother(a album) {
    // oops
    a.copies++
}

func soldAnother(a *album) {
    // what you intended
    a.copies++
}
```

Note that “dot” notation works on pointers too, equivalent to `(*a).copies`

Struct gotcha

Here's an annoying little issue in Go:

```
employees := make(map[string]Employee)  // NOTE: not *Employee

var matt Employee{
    Name:   "Matt",
    Number: 72,
    Boss:   &lamine,
    Hired:  time.Date( . . . ),
}

employees[matt.Name] = matt

employees["Matt"].Number++     // can't do this
```

A map entry is not addressable; issue #3117

Make the zero value useful

"It is usually desirable that the zero value be a natural or sensible default.

For example, in `bytes.Buffer`, the initial value of the struct is a ready-to-use empty buffer." (*GOPL* §4.4)

```
type Buffer struct {
    buf      []byte // contents are the bytes buf[off : len(buf)]
    off      int    // read at &buf[off], write at &buf[len(buf)]
    lastRead readOp // last read operation, so that Unread* can work correctly.
}
```

which has a `nil` slice we can append to, and `off` starts as 0; the "zero" value for `readOp` is `opInvalid`.

Empty structs

A `struct` with no fields is useful; it takes up no space

```
// a set type (instead of bool)

var isPresent map[int]struct{}


// a very cheap channel type

done := make(chan struct{})
```

Struct Tags and JSON

Struct tags

Tags are a part of a `struct` definition captured by the compiler

They are available to code that uses *reflection*

```
type Response struct {
    Page  int      `json:"page"`
    Words []string `json:"words,omitempty"`
}
```

Sometimes multiple tags are appropriate, separated by a space

```
type Response struct {
    Page  int      `json:"page" db:"page"`
    Words []string `json:"words,omitempty" db:"words"`
}
```

Reflection in action: JSON support

The JSON package in Go uses reflection on Go objects

```
r := &Response{Page: 1, Words: []string{"up", "in", "out"}}

j, _ := json.Marshal(r)      // ignoring errs
fmt.Println(string(j))

// {"page":1,"words":["up","lo","an"]}

var r2 Response

json.Unmarshal(j, &r2)       // ignoring errs
fmt.Printf("%#v\n", r2)

// main.Response{Page:1, Words:[]string{"up", "in", "out"}}
```

Reflection in action: JSON support

"omitempty" causes a nil object to be ignored by `Marshal`

```
r := &Response{Page: 1, Words: []string{}}

j, _ := json.Marshal(r)      // ignoring errs
fmt.Println(string(j))

// {"page":1}

var r2 Response

json.Unmarshal(j, &r2)       // ignoring errs
fmt.Printf("%#v\n", r2)

// main.Response{Page:1, Words:[]string(nil)}
```

Struct tags have many uses

Tags can also be used in conjunction with SQL queries

```
import "github.com/jmoiron/sqlx"

type item struct {
    Name string `db:"name"`
    When string `db:"created"`
}

func PutStats(db *sqlx.DB, item *item) error {
    stmt := `INSERT INTO items (name, created)
             VALUES (:name, :created);`
    _, err := db.NamedExec(stmt, item)

    return err
}
```

Struct tag gotcha

Only **exported** (capitalized) field names are convertible

```
import "github.com/jmoiron/sqlx"

type item struct {
    Name string `db:"name"`
    when string `db:"created"`  // oops
}

func PutItem(db *sqlx.DB, item *item) error {
    stmt := `INSERT INTO items (name, created)
             VALUES (:name, :created);`

    _, err := db.NamedExec(stmt, item)   // FAILS, missing when

    return err
}
```