Slice Gotchas. Struct types in Go.

Session 08

Golang course by Exadel

27 Oct 2022

Sergio Kovtunenko
Lead backend developer, Exadel

Agenda

- ▶️ Slice Gotchas
- ▶️ User-defined type definitions recap
- ▶️ Struct types in Go
- ▶️ Homework
- ▶️ Next time...

Slice Gotchas

1. Passing slices to a function

✓ When you pass slices (not pointer to slices) to other functions or methods you should be aware that you may or may not have the elements in the original slice altered.

- You will never have to worry about the length or capacity changing without your knowledge, but the actual elements in the backing array may be altered.
- Slices passed in functions "*as-is*" can be altered inside the function (simplest case):

```go
func main() {
    var s []int // nil slice
    for i := 1; i <= 3; i++ {
        s = append(s, i) // append() works fine with nil slices
    }
    fmt.Println("before:", s, len(s), cap(s)) // before: [1 2 3] 3 4
    reverse(s)
    fmt.Println("after: ", s, len(s), cap(s)) // after:  [3 2 1] 3 4
}
// function to reverse the content of the slice
func reverse(s []int) {
    for i, j := 0, len(s) - 1; i < j; i++ {
        j = len(s) - (i + 1)
        s[i], s[j] = s[j], s[i]
    }
}
// OUTPUT:
// before: [1 2 3] 3 4
// after:  [3 2 1] 3 4
```

▶️ Example: code/slicegotchas/pass_slice_to_function_01_test.go

2. Append elements to a slice (if capacity not reached)

✓ Appending to the slice passed “as-is” inside the function in case if capacity not reached:

```go
func main() {
    var s []int
    for i := 1; i <= 3; i++ {
        s = append(s, i)
    }
    fmt.Println("before:", s, len(s), cap(s)) // before: [1 2 3] 3 4
    reverse(s)
    fmt.Println("after :", s, len(s), cap(s)) // after : [999 3 2] 3 4
}
func reverse(s []int) {
    // The new slice has new length attribute which isn't a pointer, but it still points to the same array.
    // Because the original slice passed “by value” its length wasn’t altered.
    s = append(s, 999)
    for i, j := 0, len(s) - 1; i < j; i++ {
        j = len(s) - (i + 1)
        s[i], s[j] = s[j], s[i]
    }
}
// OUTPUT:
// before: [1 2 3] 3 4
// after : [999 3 2] 3 4
```

▶️ Example: code/slicegotchas/capacity_is_not_reached_02_test.go

3. Append elements to a slice may allocate new array behind the scene

✓ Appending to the slice passed "as-is" inside the function may allocate new array behind the scene and modify the new array without affecting original one:

```go
func main() {
    var s []int
    for i := 1; i <= 3; i++ {
        s = append(s, i)
    }
    fmt.Println("before:", s, len(s), cap(s)) // before: [1 2 3] 3 4
    reverse(s)
    fmt.Println("after :", s, len(s), cap(s)) // after : [1 2 3] 3 4
}
func reverse(s []int) {
    s = append(s, 999, 1000, 1001) // new array will be allocated and then reversed
    for i, j := 0, len(s)-1; i < j; i++ {
        j = len(s) - (i + 1)
        s[i], s[j] = s[j], s[i]
    }
}
// OUTPUT:
// before: [1 2 3] 3 4
// after : [1 2 3] 3 4
```

▶️ Example: code/slicegotchas/append_with_new_allocated_array_03_test.go

4. Append to inner re-sliced slice

✓ Example (*append to inner resliced slice*):

```go
package main
import "fmt"

func main() {
    array := []string{"a", "b", "c", "d", "e", "f"}

    slice1 := array[:3] // Append operations on this 'slice1' is DANGEROUS!!!
    slice2 := array[3:]

    fmt.Printf("so far so good: slice1 %v slice2 %v\n", slice1, slice2)

    slice1 = append(slice1, "BANG")

    fmt.Printf("append to slice1: %v\n", slice1)
    fmt.Printf("slice2 is now corrupt: %v\n", slice2)

    fmt.Println("Base array:", array)
}
// OUTPUT:
// so far so good: slice1 [a b c] slice2 [d e f]
// append to slice1: [a b c BANG]
// slice2 is now corrupt: [BANG e f]
// Base array: [a b c BANG e f]
```

▶️ Example: code/slicegotchas/append_to_inner_resliced_slice_04_test.go

5. Re-slicing unexpected memory usage

✓ When you re-slice a slice, the new slice will reference the array of the original slice.

- If you forget about this behavior it can lead to unexpected memory usage if your application allocates large temporary slices creating new slices from them to refer to small sections of the original data:

```go
func brokenGet() []byte {
    raw := make([]byte, 10000) // obtain huge slice
    fmt.Println(len(raw), cap(raw), &raw[0]) // prints: 10000 10000 <byte_addr_x>
    return raw[:3] // re-slice to return just a small sub-slice
}

func main() {
    data := brokenGet()
    fmt.Println(len(data), cap(data), &data[0]) // prints: 3 10000 <byte_addr_x>
}
```

- To avoid this trap make sure to copy the data you need from the temporary slice (instead of reslicing it)

▶️ Example: code/slicegotchas/unexpected_memory_usage_05_test.go

User-defined type definitions recap

User-defined type definitions recap

▶️ Type definitions:

- Based on simple type: `type ID string`
- Based on collection type: `type IDs []ID` or `type IDSet map[ID]struct{}`
- Based on composite structure: `type Human struct{ Name string; Age int }`

▶️ Benefits:

- Compile-time type checks
- Ability to add new methods => new behavior

▶️ Drawbacks:

- Converting (https://go.dev/doc/faq#convert_slice_with_same_underlying_type) `[]T1` to `[]T2` if T1 and T2 have the **same underlying type**

▶️ Example: `code/userdefinedtype/sample_type_definitions_01_test.go`

Converting []T1 to []T2 if T1 and T2 have the same underlying type

Can I convert []T1 to []T2 if T1 and T2 have the same underlying type?

This last line of this code sample does not compile.

```
type T1 int
type T2 int
var t1 T1
var x = T2(t1) // OK
var st1 []T1
var sx = ([]T2)(st1) // NOT OK
```

In Go, types are closely tied to methods, in that every named type has a (possibly empty) method set. The general rule is that you can change the name of the type being converted (and thus possibly change its method set) but you can't change the name (and method set) of elements of a composite type. Go requires you to be explicit about type conversions.

- ▶️ Solution: write explicit `for` loops to copy values.
- ▶️ Example: `code/userdefinedtype/converting_slices_02_test.go`

Struct types in Go

Struct general info

- ✓ A struct is a sequence of named elements, called fields/properties, each of which has a **name** and a **type**.
  - **Field names** may be specified:
    - *explicitly* (for *identifier list*);
    - *implicitly* (for *embedded fields*).
  - Example:

```
// An empty struct -- nothing will be allocated.
struct {}

// A struct with 6 fields.
struct {
    x, y int
    u float32
    _ float32  // padding
    A *[]int
    F func()
}
```

Struct initialization

- ✓ Variables of struct type **initialization**:
  - A common way to "*initialize*" a variable containing struct or a reference to one, is to create a **struct literal.**
  - Another option is to create a "*constructor*".
    - Go has a philosophy to use zero-value to represent the default. Utilize zero-value *as much as possible* to provide the default behavior.
    - This is usually done when the zero value isn't good enough and you need to set some default field values for instance.

```
package main
import "fmt"

type Point struct {
    X, Y int
}

var (
    p = Point{1, 2}  // has type Point
    q = &Point{1, 2} // has type *Point
    r = Point{X: 1}  // Y:0 is implicit
    s = Point{}      // X:0 and Y:0
)

func main() {
    fmt.Println(p, q, r, s)
}
```

Struct embedding (1/3)

- ✓ **Embedded field** - a field declared *with a type* but *no explicit field name*.
- ✓ An embedded type **must** be specified as a type name `T` or as a pointer to a non-interface type name `*T`, and `T` itself *may not be a pointer type*.
  - By embedding a pointer to a non-interface type we can reuse same embedded object for many enclosing types.
- ✓ If couple embedded anonymous fields inside struct has same method, the compiler cannot decide which one should be the called.
  - To resolve this ambiguity we can add a `*conflicted*` method on the defined type itself:

```
type AnotherCompositeWriter struct { // both `anonymous fields` implemented Write() method!
    io.ReadWriter
    io.Writer
}

func (a *AnotherCompositeWriter) Write(p []byte) (n int, err error) {
    return a.Writer.Write(p) // To resolve issue we have to define our own Write() implementation
}
```

Struct embedding (2/3)

✓ The unqualified type name acts as the field name.

```
// A struct with four embedded fields of types T1, *T2, P.T3 and *P.T4
struct {
    T1        // field name is T1
    *T2       // field name is T2
    P.T3      // field name is T3
    *P.T4     // field name is T4
    x, y int  // field names are x and y
}
```

✓ The following declaration is illegal *because field names must be unique in a struct type*:

```
struct {
    T     // conflicts with embedded field *T and *P.T
    *T    // conflicts with embedded field T and *P.T
    *P.T  // conflicts with embedded field T and *T
}
```

Struct embedding (3/3)

- ✓ **Embedding is really not very different than having a regular field**
  - but allows you to embed the methods on the embedded type directly into the new type.
  - It is no different than providing `Lock()` and `Unlock()` methods from `File` and *make them operate on a `sync.Mutex` field*:

```
// Consider the following struct:
type File struct {
    sync.Mutex // struct embedding
    rw io.ReadWriter
}
func main() {
    // Then, File objects will directly have access to sync.Mutex methods:
    f := File{}
    f.Lock()
}
```

Embedding is not sub-classing (1/2)

- ✓ **Embedding is not sub-classing.**
  - **Embedding types** can not be assigned to what they are embedding:

```
type (
     Animal struct {}
     Dog struct {
          Animal // Type embedding
     }
)
func main() {
     var a Animal
     a = Dog{} // WILL NOT COMPILE!!!
}
```

Embedding is not sub-classing (2/2)

✓ The **embedded struct** has no access to the **embedding struct** (*The embedded struct is thing-in-itself*):

```
func main() {
    fb := Contractor{
        Human: Human{Name: "Ivan"},
        Name:  "Petr",
    }
    fb.Greet() // call will be promoted to `fb.Human` field, which doesn't know anything about
`Contractor.Name` field!
    fb.Human.Greet()
    // OUTPUT:
    // Hello, i'm Ivan
    // Hello, i'm Ivan
}

type Human struct {
    Name string
}

func (b Human) Greet() {
    fmt.Println("Hello, i'm", b.Name)
}

type Contractor struct {
    Human
    Name string // This field has same name as in `Human` struct!
}
```

Struct tags

- ✓ A field declaration may be followed by an **optional string literal tag.**
- ✓ **Tag** becomes an *attribute* for all the fields in the corresponding field declaration.
- ✓ An ***empty tag string*** is equivalent to an **absent tag.**
- ✓ The **tags** are made visible through a reflection interface.
  - They take part in type identity for structs but are otherwise ignored.

```
struct {
    x, y float64 ""  // an empty tag string is like an absent tag
    name string  "any string is permitted as a tag"
    _    [4]byte "ceci n'est pas un champ de structure"
}

// A struct corresponding to a TimeStamp protocol buffer.
// The tag strings define the protocol buffer field numbers;
// they follow the convention outlined by the reflect package.
struct {
    microsec  uint64 `protobuf:"1"`
    serverIP6 uint64 `protobuf:"2"`
}
```

▶️ Example: code/structexample/struct_tags_03_test.go

Pitfalls with struct types

- ✓ You should always be aware that structs with a pointer inside of them can be modified, even if the variables of struct type were passed “by value” into functions:

```
type A struct {
    Ptr1 *B
    Ptr2 *B
    Val B
}
```

- ✓ Layout of fields in struct
  - Organize fields in groups, with blank lines between them.
  - Put sync.Mutex in top of a block of fields that the mutex protects.

```
type Modifier struct {
    client *client.Client

    mu    sync.RWMutex
    pmod  *profile.Modifier
    cache map[string]time.Time
}
```

Interfaces in Go

Interfaces in Go

▶️ Will be covered next time :)

Homework

Homework

- Be familiar with the official F.A.Q. (https://go.dev/doc/faq)
- Read carefully: "Go Data Structures: Interfaces" by Russ Cox (https://research.swtch.com/interfaces)
- Read carefully: "Go Data Structures" by Russ Cox (https://research.swtch.com/godata)
- "Embedding in Go" (http://www.hydrogen18.com/blog/golang-embedding.html)
- "The Go type system for newcomers" by Rakyll (https://rakyll.org/typesystem/)
- What “accept interfaces, return structs” means in Go (https://medium.com/@cep21/what-accept-interfaces-return-structs-means-in-go-2fe879e25ee8)

Next time...

▶️ Session09:

Functions and Methods, method sets, closures + `Interfaces`

- Functions
  - Function declarations
  - Named return parameters
  - Anonymous functions and variables of type function
  - Closures and their use-cases
- Methods
  - Method declarations
  - Method values
  - Method expressions
  - Method sets

Thank you

Golang course by Exadel

27 Oct 2022

Sergio Kovtunenko
Lead backend developer, Exadel
skovtunenko@exadel.com (mailto:skovtunenko@exadel.com)
https://github.com/skovtunenko (https://github.com/skovtunenko)
@realSKovtunenko (http://twitter.com/realSKovtunenko)