Type system in Go: overview

Session 06

Golang course by Exadel

20 Oct 2022

Sergio Kovtunenko
Lead backend developer, Exadel

Agenda

- ▶️ Addition to the session#05: "Error handling, defer, panic, recovery"
- ▶️ Type system introduction
- ▶️ Types in Go
- ▶️ Some built-in types in Go
- ▶️ What is left?
- ▶️ Homework
- ▶️ Next time...

Addition to the session#05: "Error handling, defer, panic, recovery"

Defer trick with anonymous function

- It’s possible to use `defer` in this situation by using an inline `func` *to wrap a set of operations instead:*

```go
func (s *Service) SaveUser(id string) {
    // do stuff first...
    func() {
        s.mu.Lock()
        defer s.mu.Unlock()
        // safely modify the map
        id, present := s.users[id]
        if present {
              s.UpdateUser(id)
              return
        }
        user := s.AddUser(id)
        s.users[id] = user
    }() // call a function!
    // do more stuff...
}
```

Used anonymous function execution!

▶️ Refer to the code example `code/anonymous_fn_with_defer_test.go`

Avoid error-check repetition when possible

▶️ Having this type definition:

```go
type binWriter struct {
    w    io.Writer
    size int64
    err  error
}
```

▶️ For almost each method we may have a precondition check:

```go
func (w *binWriter) Write(v interface{}) {
    if w.err != nil { // <-- check this!
        return
    }
    if w.err = binary.Write(w.w, binary.LittleEndian, v); w.err == nil {
        w.size += int64(binary.Size(v))
    }
}
```

Source: "Twelve Go Best Practices" by Francesc Campoy Flores (https://go.dev/talks/2013/bestpractices.slide#1)

More information to read

▶️ "Working with Errors in Go 1.13" by Damien Neil and Jonathan Amsterdam (https://go.dev/blog/go1.13-errors)

▶️ "Don't just check errors, handle them gracefully" by Dave Cheney (https://dave.cheney.net/2016/04/27/dont-just-check-errors-handle-them-gracefully)

▶️ "More about Deferred Function Calls" by Tapir (https://go101.org/article/defer-more.html)

▶️ "Deferred Function Calls" by Tapir (https://go101.org/article/control-flows-more.html#defer)

Type system introduction

Go type system foreword

▶️ Clarity is critical.

▶️ When reading code, it should be clear what the program will do.

▶️ When writing code, it should be clear how to make the program do what you want.

▶️ Sometimes this means writing out a loop instead of invoking an obscure function.

▶️ For more background on design:

- "Less is exponentially more (Pike, 2012)" (http://commandcenter.blogspot.com/2012/06/less-is-exponentially-more.html)
- "Go at Google: Language Design in the Service of Software Engineering (Pike, 2012)" (http://go.dev/talks/2012/splash.article)

Source: "Go for Java Programmers" by Sameer Ajmani (https://go.dev/talks/2015/go-for-java-programmers.slide#16)

Go is about composition

▶️ Go is Object-Oriented, but not in the usual way.

- no **classes** (methods may be declared on any type)
- no subtype **inheritance**
- **interfaces** are satisfied implicitly (structural typing)

▶️ The result: simple pieces connected by small interfaces.

▶️ **You can build your own types based on (using) available built-in types!**

Source: "Go: code that grows with grace" by Andrew Gerrand (https://go.dev/talks/2012/chat.slide#5)

Types in Go

Types in Go

- ✓ Types can be primitive (basic) and composite:
  - **Basic types**:
    - Built-in string type:
      - string
    - Built-in boolean type:
      - bool
    - Built-in numeric types:
      - int8, uint8 (byte), int16, uint16, int32 (rune), uint32, int64, uint64, int, uint, uinptr.
      - float32, float64.
      - complex64, complex128.
  - **Composite types**:
    - pointer types
    - struct types
    - function types - functions are first-class types in Go.
    - container types:
      - array types - fixed-length container types.
      - slice type - dynamic-length and dynamic-capacity container types.
      - map types - maps are associative arrays (or dictionaries).
    - channel types - channels are used to synchronize data among goroutines.
    - interface types - interfaces play a key role in reflection and polymorphism.

Underlying type

- A **type** determines a set of values together with operations and methods specific to those values.
    - Each type T has an **underlying type**:
        - If T is one of the predeclared *boolean*, *numeric*, or *string* types, or a *type literal*, the corresponding underlying type is T itself.
        - Otherwise, T's underlying type is the underlying type of the type to which T refers in its type declaration.

```go
type (
	A1 = string   // underlying type is `string`
	A2 = A1       // underlying type is `string`
)

type (
	B1 string     // underlying type is `string`
	B2 B1         // underlying type is `string`
	B3 []B1       // underlying type is `[]B1`
	B4 B3         // underlying type is `[]B1`
)
```

Type declarations

- ✓ Type declarations come in *two forms*:
  - **alias declarations**
  - **type definitions**
- ✓ Types may be *named* or *unnamed*

Type definitions

- ✓ **type definitions:**
  - A type definition creates a new, distinct type with the same underlying type and operations as the given type, and binds an identifier to it.
  - New defined type is different from any other type, including the type it is created from.
  - A defined type may have methods associated with it.
  - Operations defined for the existing type *are also defined* for the new type.
  - Types can be defined *within function bodies*.

▶️ Example code: `code/type_definitions_test.go`

Type definitions example

```
type (
    Point struct{ x, y float64 }  // Point and struct{ x, y float64 } are different types
    polar Point                   // polar and Point denote different types
)

// A Mutex is a data type with two methods, Lock and Unlock.
type Mutex struct         { /* Mutex fields */ }
func (m *Mutex) Lock()    { /* Lock implementation */ }
func (m *Mutex) Unlock()  { /* Unlock implementation */ }

// NewMutex has the same composition as Mutex but its method set is empty.
type NewMutex Mutex

// The method set of PtrMutex's underlying type *Mutex remains unchanged,
// but the method set of PtrMutex is empty.
type PtrMutex *Mutex

// The method set of *PrintableMutex contains the methods
// Lock and Unlock bound to its embedded field Mutex.
type PrintableMutex struct {
    Mutex
}

type Block interface {
    BlockSize() int
    Encrypt(src, dst []byte)
    Decrypt(src, dst []byte)
}
// MyBlock is an interface type that has the same method set as Block.
type MyBlock Block
```

Type Alias

- ✓ **alias declarations:**
  - within the scope of the identifier, it serves as an alias for the type:

```
type (
    nodeList = []*Node  // nodeList and []*Node are identical types
    Polar    = polar    // Polar and polar denote identical types
)
```

- Like type definitions, type aliases can also be declared within function bodies.

Named and Unnamed types in Go

- ✓ Types may be *named or unnamed*.
  - **Named types** are specified by a (possibly qualified, like **math.Sin**) *type name*.
    - All basic types are named types.
  - **Unnamed types** are specified using a *type literal*, which *composes a new type from existing types*.

- ✓ Type definitions may be used to define different boolean, numeric, or string types and associate methods with them:

```
type TimeZone int

const (
    EST TimeZone = -(5 + iota)
    CST
    MST
    PST
)

func (tz TimeZone) String() string {
    return fmt.Sprintf("GMT%+dh", tz)
}
```

Some built-in types in Go

All primitive types in Go

✓ List of all available primitive types:

```
bool        // values are: 'true' or 'false'
string

// Numeric types:

uint        // either 32 or 64 bits depends on host platform
int         // same size as uint
uintptr     // an unsigned integer large enough to store the uninterpreted bits of a pointer value
uint8       // the set of all unsigned  8-bit integers (0 to 255)
uint16      // the set of all unsigned 16-bit integers (0 to 65535)
uint32      // the set of all unsigned 32-bit integers (0 to 4294967295)
uint64      // the set of all unsigned 64-bit integers (0 to 18446744073709551615)

int8        // the set of all signed  8-bit integers (-128 to 127)
int16       // the set of all signed 16-bit integers (-32768 to 32767)
int32       // the set of all signed 32-bit integers (-2147483648 to 2147483647)
int64       // the set of all signed 64-bit integers (-9223372036854775808 to 9223372036854775807)

float32     // the set of all IEEE-754 32-bit floating-point numbers
float64     // the set of all IEEE-754 64-bit floating-point numbers

complex64   // the set of all complex numbers with float32 real and imaginary parts
complex128  // the set of all complex numbers with float64 real and imaginary parts

byte        // alias for uint8
rune        // alias for int32 (represents a Unicode code point)
```

String type

- ✓ A string is in effect a **read-only sequence of bytes**.
- ✓ The **default value** for string variable is *empty string* “”. Thus, a string cannot be **nil**.
  - Only use a string pointer ***string** if you need **nil**. Otherwise, use a normal string:

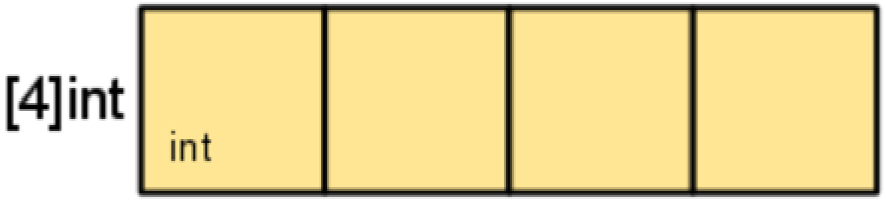
```
func UseString(s *string) error {
    if s == nil {
        temp := "" // *string cannot be initialized
        s = &temp  // in one statement
    }
    value := *s    // safe to dereference the *string
}
```

- ✓ The **length** of a string s (its size in bytes) can be discovered using the built-in function **len()**.
- ✓ A **string's bytes** can be accessed by *integer indices* **0** through **len(s)-1**.
- ✓ It is not required to hold Unicode text, *UTF-8 text*, or any other predefined format.
  - But source code in Go is defined to be *UTF-8 text*; no other representation is allowed. So the source code for the string literal (**normal and raw string literals**) is *UTF-8 text*.
- ✓ *No guarantee* is made in Go that characters in strings are normalized.
- ✓ A string might not even hold characters.

Arrays (1/2)

- ✓ An array is a *numbered sequence of elements* of a single type, called the element type.
  - The number of elements is called the **length** and is *never negative*.
  - Arrays cannot be resized.

- ✓ The in-memory representation of `[4]int` :



- ✓ An **array type definition** specifies a length and an element type.
  - An **array variable** denotes the entire array - it is not a pointer to the first array element (as would be the case in C).

- ✓ The elements can be addressed by integer indices `0` through `len(a)-1`.

Arrays (2/2)

- ✓ The **length** of array a can be discovered using the built-in function `len()`.
- ✓ Arrays are **values** in Go language!
- ✓ One way to think about arrays is as a sort of struct but with indexed rather than named fields: a fixed-size composite value.
- ✓ The length is part of the array's type
  - it must evaluate at compile-time to a non-negative constant representable by a value of type **int**.
  - That means `[4]int` and `[5]int` are distinct, incompatible types.
- ✓ Array types are always one-dimensional but may be composed to form *multi-dimensional types*.
- ✓ Example of arrays:

```
[32]byte
[2*N] struct { x, y int32 }
[1000]*float64
[3][5]int
[2][2][2]float64  // same as [2]([2]([2]float64))
```

Pointers (1/2)

- ✓ *Pointers are values* which point to other values.
- ✓ Pointers default values is **nil**.
- ✓ For *each type*, there exists a *distinct pointer type*, accessible via operators:
  - the address of, &
  - dereference, *
- ✓ There is **no pointer arithmetic** in Golang:

```
x := 1000
y := &x
y += 4          // nope
```

Pointers (2/2)

- ✓ When are function parameters passed by value?
  - As in all languages in the C family, everything in Go is passed by value.
  - **Map** and **slice** values behave like pointers:
    - *they are descriptors* that contain pointers to the underlying map or slice data.
  - **Copying a map or slice** value doesn't copy the data it points to.
  - **Copying an interface value** makes a copy of the thing stored in the interface value.
  - If the interface value holds a struct, **copying the interface** value makes a copy of the struct.
  - If the interface value holds a pointer, **copying the interface** value makes a copy of the pointer, but again ~~not the data it points to~~.

- ✓ When should I use a **pointer to an interface**?

  - *Almost never*.
  - **Pointers to interface** values arise only in rare, tricky situations involving disguising an interface value's type for delayed evaluation.
  - A pointer to a concrete type *can satisfy an interface*, with one exception a pointer to an interface can never satisfy an interface.
  - The one exception is that any value, even a pointer to an interface, can be assigned to a variable of empty interface type (**interface{}**).

What is left?

What is left?

- Slices
- Maps
- Functions (+methods)
- Struct types
- Interfaces
- Channels

Homework

- Read: "Go Type System Overview" by Tapir (https://go101.org/article/type-system-overview.html)
- Read: "Strings, bytes, runes and characters in Go" by Rob Pike (https://go.dev/blog/strings)
- Read: "Go Slices: usage and internals" by Andrew Gerrand (https://go.dev/blog/slices-intro)

Be familiar and understand motivation and arguments:

- "Less is exponentially more (Pike, 2012)" (http://commandcenter.blogspot.com/2012/06/less-is-exponentially-more.html)
- "Go at Google: Language Design in the Service of Software Engineering (Pike, 2012)" (http://go.dev/talks/2012/splash.article)

Next time...

▶️ Session07: **An in-depth look at Slices and Maps**

- Slices
    - Slice creation
    - Slice internals
    - Slicing slices pitfalls
    - Appending to and copying slices
    - Common pitfalls
- Maps
    - Map literals
    - Mutating maps
    - Concurrent access to maps

Thank you

Golang course by Exadel

20 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)