Funcitons and Methods in Go. Introduction to Interfaces.

Session 09

Golang course by Exadel

31 Oct 2022

Sergio Kovtunenko
Lead backend developer, Exadel

Agenda

- ▶️ Functions in Go
- ▶️ Methods in Go
- ▶️ Interfaces in Go - Part 1

Functions in Go

Generic info about functions

✓ *Regular* ***functions*** looks like:

```
func Sin(x float64) float64
func AddScale(x, y int, f float64) int
```

✓ **Functions** with *multiple return values*:

```
func Write(data []byte) (written int, err error) // named return parameters
func WriteTo(data []byte) (int, error)
```

✓ **Functions** with *variadic parameter lists* looks like:

```
func Printf(format string, args ...interface{})
```

✓ **Functions** are first-class values:

- A **function type** denotes the set of all functions *with the same parameter and result types*.
- The value of an *uninitialized variable* of **function type** is **nil.**

Function declarations

- ✓ A *function* declaration binds an identifier (the function name) to a function.
- ✓ If the **function's signature** declares result parameters, the function body's statement list *must end in a terminating statement*:

```
func IndexRune(s string, r rune) int {
    for i, c := range s {
        if c == r {
            return i
        }
    }
    // invalid: missing return statement
}
```

- ✓ A **function declaration** may omit the body.
  - Such a declaration provides the signature for a function implemented outside Go, such as an *assembly* routine.

```
func flushICache(begin, end uintptr)  // implemented externally
```

Function declarations: function literals and varargs

- ✓ A **function literal** represents an anonymous function.
  - A **function literal** can be assigned to a variable or invoked directly.
  - There is no distinct method type and there are no method literals.
  - Example:

```
f := func(x, y int) int { return x + y }
func(ch chan int) { ch <- ACK }(replyChan)
```

- ✓ The **final incoming parameter** in a function signature may have a type prefixed with ...
  - A function with such a parameter is called **variadic** and may be invoked with zero or more arguments for that parameter.

```
func variadicFunction(a int, b float32, more ...interface{}) bool {
    return true
}
```

Function declarations: named return variables

- ✓ Function declaration can have named return variables.
  - They are treated as variables defined at the top of the function.
  - These names should be used to document the meaning of the return values.
  - **WARNING:** another benefit of named return values is the use in closures (i.e. **defer** statements).
    - Thus one may access the named return value in a function that is called as the result of a **defer** statement and act accordingly.
    - Without using named return variables the next method will always return **nil** as result (or **default values** for other variables):

```go
func functionWithNamedReturnVariables() (err error) {
    // If there is a panic we need to recover in a deferred func:
    defer func() {
        if r := recover(); r != nil {
            // the named return value will be modified:
            err = errors.New("return value will be modified successfully because of named return value")
        }
    }()

    legacyFunctionThatWillPanic() // always panic inside

    fmt.Println("This is unreachable call!")
    return err // because of panic, this return statement won't be executed
}
func legacyFunctionThatWillPanic() {
    panic("PANIC!")
}
```

Function declarations: named return variables - Example

▶️ Code:

Closures

- ✓ A **closure** is a function value that references variables from outside its body.
- ✓ The function **may access and assign** to the referenced variables; in this sense the function is "bound" to the variables.
- ✓ Variable scope inside closure:
  - **Closures** in Go capture variables by reference. That means the inner function holds a reference to the `i` variable in the outer scope, and each call of it accesses this same variable.
  - This is one of the many benefits of a closure - we can persist data between function calls while also isolating the data from other code.

```
func makeEvenGenerator() func() uint {
    i := uint(0) // this variable captured by reference!!!
    return func() (ret uint) {
        ret = i
        i += 2
        return
    }
}

func main() {
    nextEven := makeEvenGenerator()
    fmt.Println(nextEven()) // Out: 0
    fmt.Println(nextEven()) // Out: 2
    fmt.Println(nextEven()) // Out: 4
}
```

Closures - Example

▶️ Code:

Closures pitfall: a program with a bug

✓ Example of incorrect program:

```go
package main
import "fmt"

func main() {
    done := make(chan bool)

    values := []string{"a", "b", "c"}
    for _, v := range values {
        go func() {
            fmt.Println(v)
            done <- true
        }()
    }

    // wait for all goroutines to complete before exiting
    for _ = range values {
        <-done
    }
}
// OUTPUT:
// c
// c
// c
```

Closures pitfall: fixed solution

✓ Fix #1: pass the variable as an argument to the closure:

```
for _, v := range values {
    go func(u string) {
        fmt.Println(u)
        done <- true
    }(v)
}
```

✓ Fix #2: recreate local copy of outer variable:

```
for _, v := range values {
    v := v // create a new 'v'.
    go func() {
        fmt.Println(v)
        done <- true
    }()
}
```

Closures pitfall - Example

▶️ Code:

Methods in Go

Methods in Go: general information

✓ **Methods** are functions with a receiver parameter:

```
func (p Point) String() string {
    return fmt.Sprintf("(%d, %d)", p.x, p.y)
}
```

✓ The **receiver** binds the method to its base type (`Point` in this example):

```
type Point struct {
    x, y int
}
```

✓ **Methods** are invoked via the usual dot notation:

```
func main() {
    p := Point{2, 3}
    fmt.Println(p.String())
    fmt.Println(Point{3, 5}.String())
}
```

Methods in Go: methods on type from another package

- ✓ You can't declare a method with a receiver whose type is defined in another package (which includes the built-in types such as `int`).
  - To workaround this, we need to define an alias for the type we want to extend.
  - Methods can be defined for any user-defined type!

```
type MyFloat float64

func (f MyFloat) Abs() float64 {
    if f < 0 {
        return float64(-f)
    }
    return float64(f)
}
```

Methods in Go: method set

- ✓ **Key concept** - Method sets:
  - Types **T** and ***T** have *different* method sets.
  - The **method set** of any other named type **T** consists of all methods with receiver type **T**.
  - The **method set** of the corresponding pointer type ***T** is the set of all methods with receiver ***T** or **T**
    - that is, it also contains the method set of **T**.

```
type MyStruct struct{}
func (s *MyStruct) pointerMethod() { } // method on pointer
func (s MyStruct)  valueMethod()   { } // method on value
```

- ✓ There are **no method overloading.**

- ✓ **Methods** can be dispatched:
  - statically:
    - methods on a struct or any other concrete type are always resolved statically.
  - dynamically:
    - the only way to have dynamically dispatched methods is through an interface.

Methods in Go: method set - Example

▶️ Code:

Methods in Go: non-pointer method receivers

- ✓ We can define methods using **both** *pointer* and *non-pointer* method receivers.
- ✓ We can *treat the receiver* as if it was an *argument being passed to the method*. All the same reasons why you might want to pass by value or pass by reference apply.
- ✓ When to use **value method receiver**?
  - If the receiver is a **map**, **func** or **chan**.
  - If the receiver is a **small array** or **struct** that is naturally a **value type** (*for instance, something like the `time.Time` type*), with no mutable fields and no pointers, or is just a simple basic type such as `int` or `string`.
    - A **value receiver** can *reduce the amount of garbage* that can be generated;
      - if a value is passed to a value method, an on-stack copy can be used instead of allocating on the heap.

Methods in Go: pointer method receivers

✓ **When to use pointer method receiver?**

- You want to actually **modify** the receiver ("read/write" as opposed to just "read");
- For structs that contain a `sync.Mutex` or similar synchronizing field (they musn't be copied).
- The **struct is very large** and a deep copy is expensive;
- **Consistency**:
  - if some of the methods on the struct have pointer receivers, the rest should too => this allows predictability of behaviour because *the method set is consistent regardless of how the type is used*
- If the receiver is a **struct**, **array** or **slice** and any of its elements is a pointer to something that might be mutating, prefer a pointer receiver, as it will make the intention more clear to the reader.

Methods in Go: methods are just functions - Example

▶️ Code:

Interfaces in Go - Part 1

Interfaces in Go: general information

- ✓ Why do we use interfaces?
  - Writing generic algorithms;
  - Hiding implementation details:
    - decouple implementation from API;
    - easily switch between implementations / or provide multiple ones;
  - Providing interception points.

- ✓ Main ideas behind **interfaces**:
  - Strict: interfaces for behaviour, static types for data.
  - *The broader interface, the weaker abstraction.*
  - Interface, in fact, should be created by consumer.
    - Define interfaces where you use them.
    - If you don't want to provide multiple implementations of the same high-level behavior, you don't introduce interfaces.
  - "A great rule of thumb for Go is ***accept interfaces, return structs***." (Jack Lindamood)
    - Another advice: be generic when describing what a function needs, and be explicit when describing what a package provides.

- ✓ Having declared variable of `interface` type we know that:
  - There is *nothing real* about this variable.
  - There is *nothing concrete* about this variable.
  - This variable is **valueless**.

Interfaces in Go: rules to satisfy them

- ✓ Internally interfaces are **two words wide**:
  - schematically they look like: **(type, value)**
    - a pointer to a **method table** (holding type and method implementations)
    - a pointer to a **concrete value** (the type defined by the method table)
  - so values of interface types are *prone to race-conditions*. Because of 2-word nature.
- ✓ **Rule about implementing interfaces is simple**: "are the function's names and signatures exactly those of the interface?".
- ✓ An interface can contain the name of one or more other interface(s), which is equivalent to *explicitly enumerating the methods of the embedded interface in the containing interface.*

▶️ Code:

Homework

- "Using named return variables to capture panics in Go" by Jon Calhoun (https://www.calhoun.io/using-named-return-variables-to-capture-panics-in-go/)
- "What is a Closure?" by Jon Calhoun (https://www.calhoun.io/what-is-a-closure/#closuresprovidedataisolation)
- "Receiver Type" by Golang official Wiki (https://github.com/golang/go/wiki/CodeReviewComments#receiver-type)
- "understanding the interface" by Francesc Campoy (https://speakerdeck.com/campoy/understanding-the-interface)
- "Interface Values Are Valueless" by William Kennedy (https://www.ardanlabs.com/blog/2018/03/interface-values-are-valueless.html)
- Read carefully: "Go Data Structures: Interfaces" by Russ Cox (https://research.swtch.com/interfaces)
- What “accept interfaces, return structs” means in Go (https://medium.com/@cep21/what-accept-interfaces-return-structs-means-in-go-2fe879e25ee8)
- Play with code: "https://github.com/Evertras/go-interface-examples" (https://github.com/Evertras/go-interface-examples)

Next time...

▶️ Session10:

Interfaces - part 2. Testing in Go

- Basic Tests
- Benchmarks
- Table-driven tests
- Main test
- Mocking
- Testify assertion library

Thank you

Golang course by Exadel

31 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)