Seminario de Lenguajes opción Go

Raúl Champredonde

Seminario de Lenguajes opción Go

- Manejo de Errores
- Estrategias de Manejo de Errores
- Package errors
- Function Values
- Anonymous Functions
- Variadic Functions
- Deferred Function Calls

Manejo de Errores

- Los errores son comportamientos esperables/inevitables.
- Las funciones para las cuales un error es uno de los posibles comportamientos, usualmente retornan un valor resultado adicional.

- Si el error tiene una única causa posible ...

```
value, ok := cache.Lookup(key)
if !ok {
    // ...cache[key] does not exist...
}
```

Manejo de Errores

- Si el error puede tener múltiples causas, el resultado adicional es de tipo `error`

```
type error interface {
    Error() string
}
```

```
i, err := strconv.Atoi("42")
if err != nil {
    fmt.Printf("Error: %v\n", err)
    return
}
fmt.Println("Entero convertido: ", i)
// qué pasaría con ...
i, err := strconv.Atoi("42a")
// qué pasaría con ...
i, err := strconv.Atoi("99999999999999999")
```

Estrategias de Manejo de Errores

- Cuando una función produce un error, es responsabilidad del "llamador" chequearlo y tomar la acción apropiada.

Estrategias de Manejo de Errores

- Propagación del error:

```
func AddStr(s1, s2 string) (string, error) {
    i1, err := strconv.Atoi(s1)
    if err != nil {
        return "", err
    }
    i2, err := strconv.Atoi(s2)
    if err != nil {
        return "", err
    }
    return strconv.Itoa(i1 + i2), nil
}
```

```
func main() {
    s, err := AddStr("42", "28a")
    if err != nil {
        fmt.Printf("Error: %v\n", err)
        return
    }
    fmt.Println("Suma: ", s)
}
```

```
Error: strconv.Atoi: parsing "28a": invalid syntax
```

Estrategias de Manejo de Errores

- Reemplazo del error:

```
func AddStr(s1, s2 string) (string, error) {
    i1, err := strconv.Atoi(s1)
    if err != nil {
        return "", fmt.Errorf("convirtiendo %s", s1)
    }
    i2, err := strconv.Atoi(s2)
    if err != nil {
        return "", fmt.Errorf("convirtiendo %s", s2)
    }
    return strconv.Itoa(i1 + i2), nil
}
```

```
func main() {
    s, err := AddStr("42", "28a")
    if err != nil {
        fmt.Printf("Error: %v\n", err)
        return
    }
    fmt.Println("Suma: ", s)
}
```

```
Error: convirtiendo 28a
```

`fmt.Errorf` formatea el mensaje de error usando `fmt.Sprintf` y devuelve un nuevo valor `error`.

Estrategias de Manejo de Errores

- Reintentar la operación que generó el error:

```
func WaitForServer(url string) error {
    const timeout = 1 * time.Minute
    deadline := time.Now().Add(timeout)
    for tries := 0; time.Now().Before(deadline); tries++ {
        _, err := http.Head(url)
        if err == nil {
            return nil // success
        }
        log.Printf("server not responding (%s); retrying...", err)
        time.Sleep(time.Second << uint(tries)) // exponential back-off
    }
    return fmt.Errorf("server %s failed to respond after %s", url, timeout)
}
```

Estrategias de Manejo de Errores

- Terminación controlada (si es imposible evitar o recuperarse de un error):

```
// En main ...
if err := WaitForServer(url); err != nil {
   fmt.Fprintf(os.Stderr, "Site is down: %v\n", err)
   os.Exit(1)
}



// or ...
if err := WaitForServer(url); err != nil {
   log.Fatalf("Site is down: %v\n", err)
}
```

Estrategias de Manejo de Errores

- Registrar el error y continuar (tal vez condicionando alguna funcionalidad):

```
if err := Ping(); err != nil {
   log.Printf("ping failed: %v; networking disabled", err)
}


// or ...
if err := Ping(); err != nil {
   fmt.Fprintf(os.Stderr, "ping failed: %v; networking disabled\n", err)
}
```

Estrategias de Manejo de Errores

- Ignorar el error:

```
content, err := os.ReadFile("data.json")

if err != nil {
   content = []byte("Datos a usar en caso de haber error al leer el archivo")
}

fmt.Println(string(content))
```

Package errors

```
package errors

func New(text string) error { return &errorString{text} }
type errorString struct { text string }
func (e *errorString) Error() string { return e.text }
```

```
package fmt

import "errors"

func Errorf(format string, args ...interface{}) error {
   return errors.New(Sprintf(format, args...))
}
```

Function Values

- Los valores "function" tienen un tipo (como cualquier otro valor) y pueden ser asignados, pasados como parámetros o retornados por una función

```
func square(n int) int { return n * n }
func negative(n int) int { return -n }
func product(m, n int) int { return m * n }

f := square
fmt.Println(f(3)) // "9"

f = negative
fmt.Println(f(3)) // "-3"
fmt.Printf("%T\n", f) // "func(int) int"

f = product // compile error: can't assign f(int, int) int to f(int) int
```

Function Values

- El valor por defecto de un tipo “function” es `nil`

```
var f func(int) int
f(3) // runtime error: call of nil function


var f func(int) int
if f != nil {
   f(3)
}
```

Function Values

- Ejemplo

```
func forEachNode(n *Node, pre, post func(n *Node)) {
    if pre != nil {
        pre(n)
    }
    for c := n.FirstChild; c != nil; c = c.NextSibling {
        forEachNode(c, pre, post)
    }
    if post != nil {
        post(n)
    }
}
```

Anonymous Functions

- Ejemplo

```
s := strings.Map(func(r rune) rune { return r + 1 }, "HAL-9000")

// or ...
rot13 := func(r rune) rune {
        switch {
        case r >= 'A' && r <= 'Z':
           return 'A' + (r-'A'+13)%26
        case r >= 'a' && r <= 'z':
           return 'a' + (r-'a'+13)%26
        }
        return r
     }
s := strings.Map(rot13, "HAL-9000")
```

Anonymous Functions

- Ejemplo

```
func squares() func() int {
    var x int
    return func() int {
        x++
        return x * x
    }
}
func main() {
    f := squares()
    fmt.Println(f()) // "1"
    fmt.Println(f()) // "4"
    fmt.Println(f()) // "9"
    fmt.Println(f()) // "16"
}
```

Variadic Functions

- Puede ser invocada con una cantidad variable de parámetros

```
func sum(vals ...int) int {
   total := 0
   for _, val := range vals {
      total += val
   }
   return total
}

fmt.Println(sum())           // "0"
fmt.Println(sum(3))          // "3"
fmt.Println(sum(1, 2, 3, 4)) // "10"

values := []int{1, 2, 3, 4}
fmt.Println(sum(values...))  // "10"
```

Deferred Function Calls

- La invocación se ejecuta después de que termina la función invocadora

```
// function ReadFile del package "os"

func ReadFile(name string) ([]byte, error) {
   f, err := Open(name)
   if err != nil {
      return nil, err
   }
   defer f.Close()
   // calcular el tamaño
   ...
   // leer el contenido del archivo
   ...
}
```

Panic

- Cuando Go detecta un error en tiempo de ejecución, entra en pánico.
- La ejecución normal se detiene,
  se ejecutan las invocaciones a funciones diferidas que hubiera dentro de la función que generó el error y
  el programa falla mostrando un mensaje de error que incluye el valor del panic.

Panic

```
package main

import "fmt"

func main() {
   f(3)
}

func f(x int) {
   fmt.Printf("f(%d)\n", 300/x)
   defer fmt.Printf("defer %d\n", x)
   f(x - 1)
}
```

```
f(100)
f(150)
f(300)
defer 1
defer 2
defer 3
panic: runtime error: integer divide by zero

main.f
        C:/source/pan/pan.go:9
main.f(0x1)
        C:/source/pan/pan.go:11
main.f(0x2)
        C:/source/pan/pan.go:11
main.f(0x3)
        C:/source/pan/pan.go:11
main.main()
        C:/source/pan/pan.go:6
exit status 2
```

Panic

- El panic también puede ser generado por el programa

```
func Reset(x *Buffer) {
    if x == nil {
        panic("x is nil") // Innecesario! Salvo mejor mensaje
    }
    x.elements = nil
}
```

- `panic` es una función incluida en el lenguaje, que recibe un parámetro de cualquier tipo.
- Para este tipo de errores "esperables" es mejor usar los `error`. Y usar los panic para errores "no esperables".

Recover

- Ante la ocurrencia de un `panic` es posible recuperar la ejecución, o al menos dejar todo prolijo.

```
func Parse(input string) (s *Syntax, err error) {
    defer func() {
        if p := recover(); p != nil {
            err = fmt.Errorf("internal error: %v", p)
        }
    }()
    // ...parser...
}
```

- Si `recover` es invocada dentro de una función diferida, cuando entra en pánico la función que incluye la sentencia diferida, `recover` finaliza el estado de pánico y retorna el valor `panic`.
- La función que entró en pánico no continúa y termina "normalmente".
- Si `recover` es invocada en cualquier otro momento no tiene ningún efecto y retorna `nil`.