GO

BOOTCAMP

Clase en vivo

//Go Bases

IT BOARDING

BOOTCAMP

Objetivos de esta clase

- Aprender a manejar errores.
- Conocer el paquete errors.
- Aprender a comparar errores.
- Entender la importancia de validar errores y su utilidad en el diseño de funciones.

Índice

IT BOARDING
BOOTCAMP

1 | Repaso

IT BOARDING

BOOTCAMP

Errores

// ¿Por qué es importante manejar errores?

Normalmente, el programa que desarrollamos puede arrojar uno o más errores. Manejar adecuadamente esos errores nos brinda al menos dos ventajas:

- **Ahorrar tiempo:** al poder encontrar más fácilmente qué es lo que falla y dónde.
- **Evitar que la ejecución finalice de modo o en tiempo no deseado:** al incorporar funciones que capturen los errores y permitan al programa continuar su ejecución.

Es conveniente (o necesario) que nuestro código siempre posea un adecuado manejo de errores.

Ejemplo implementando errors.New() #1

Veamos un ejemplo paso a paso de cómo implementar la función **New()** del paquete **errors**:
1.- Primero: definimos nuestro package **main** e importamos los packages **fmt** y **errors**.

```
{}
package main

import (
    “fmt”
    “errors”
)
```

Ejemplo implementando errors.New() #2

2.- Declaramos nuestra función **main().**

Definimos una variable llamada **statusCode**" con un valor de tipo **int**. Luego realizamos una validación para comprobar si **statusCode** es válido. En cuyo caso utilizamos **validateStatusCode()** para generar la validación.

{}

```
func main() {
    statusCode := 404
    if err := validateStatusCode(statusCode); err != nil {
        fmt.Printf("http request failed: %v", err)
        return
    }
    fmt.Println("the program ended successfully")
}
```

Ejemplo implementando errors.New() #3

3.- Definimos nuestra función **validateStatusCode().**

En este caso, la función devolverá un error, cuando el ***code*** recibido sea mayor a 399, caso contrario no será un error.

{}

```
func validateStatusCode(code int) error {
    if code > 399 {
        return errors.New("unexpected http status code")
    }
    return nil
}
```

2 | Paquete errors

IT BOARDING

BOOTCAMP

// ¿Para qué sirve el paquete errors?

El package errors implementa funciones para manipular errores: New(), Is(), As() y Unwrap().

Aclaración:

La función **New()** crea errores que sólo contienen un texto de mensaje tipo string. De modo que vamos a concentrarnos en las restantes funciones del package **errors**: **Is(), As()** y **Unwrap().**

Método As()

As()

La función **As()** comprueba si un error es de un tipo específico. Si encuentra coincidencia devuelve "***true***", caso contrario devuelve "***false***".

La función se comporta como una aserción o afirmación de tipo.

```
{}  func As(err error, target interface{}) bool
```

Un error coincide con el objetivo, o target, si el valor concreto del error es asignable al valor señalado por el objetivo.

Ejemplo implementando As(): #1

Veamos un ejemplo paso a paso de cómo implementar la función **As():**
1.- Primero lo primero: definir nuestro package **main** e importar los packages **fmt** y **errors**.

```
{}
package main

import (
    "fmt"
    "errors"
)
```

Ejemplo implementando As(): #2

2.- Luego definimos un **struct** (al que nombraremos **myError**) y hacemos que implemente el método **Error()** de la interface **error**.

{}

```
// we define a type struct
type myError struct {
   msg string
   x string
}
//we make our type struct implement the Error() method
func (e *myError) Error() string {
   return fmt.Sprintf("An error occurred: %s, %s", e.msg, e.x)
}
```

Ejemplo implementando As(): #3

4.- Finalmente generamos nuestra función **main().**
Dentro definiremos una variable llamada “e” que será de tipo **myError** y asignaremos un valor a sus atributos de tipo **string**.
Y declararemos otra variable que apuntaremos a nuestro tipo **myError**.

{}

```
func main() {
   e := &myError{"new error", "404"}

   var err *myError

   isMyError := errors.As(e, &err)// compares errors

   fmt.Println(isMyError)//prints true because the errors match
}
```

Método Is()

errors.Is()

La función **Is()** compara un error con un valor. Si encuentra coincidencia devuelve "*true*", caso contrario, devuelve "*false*".

Se comporta como una comparación con un error centinela.

```
{}  func Is(err, target error) bool
```

Un error coincide con un objetivo si es igual a ese objetivo.

Ejemplo implementando errors.Is() #1

Veamos un ejemplo paso a paso de cómo implementar la función **Is()** del paquete **errors**:
1.- Primero lo primero: definimos nuestro package **main** e importamos los packages **fmt** y **errors**.

```
{}
package main

import (
    "fmt"
    "errors"
)
```

Ejemplo implementando errors.Is() #2

2.- Creamos un nuevo error con la función **New()** y lo asignamos a una variable que definimos como **err1**. Luego declaramos una función "*x*" que retorna "*err1*".

```
var err1 = errors.New("error number 1")

func x() error {
    return fmt.Errorf("extra error information: %w", err1)
}
```

Ejemplo implementando errors.Is() #3

3.- Declaramos nuestra función **main()**.
Definimos una variable llamada “e”, a la que le asignamos el retorno de nuestra función “*x*”.
Implementamos la función **Is()** del paquete **errors** para comparar “e” con “*err1*” y asignamos su retorno a la variable **coincidence**.
Finalmente imprimimos el valor de **coincidence**.

```
{}
func main() {
    e := x()
    coincidence := errors.Is(e, err1)
    fmt.Println(coincidence) //print true
}
```

Método Unwrap()

errors.Unwrap()

Un error que contiene a otro puede implementar **Unwrap()**. Esta función devuelve el error subyacente. Si *"e1.Unwrap()"* regresa "*e2*", entonces, decimos que "*e1*" envuelve a "*e2*" y que se puede desenvolver "*e1*" para obtener "*e2*".

```
{}  func Unwrap(err error) error
```

Si tenemos un error llamado "e1" y otro llamado "e2", y "e1" envuelve a "e2", al ejecutar la función **errors.Unwrap(e1)** devolverá "e2", caso contrario devolverá "nil".

Ejemplo implementando errors.Unwrap() #1

Veamos un ejemplo paso a paso de cómo implementar la función "*Unwrap()*" del paquete "*errors*":
1.- Primero: definimos nuestro package *main* e importamos los paquetes "*fmt*" y "*errors*".

```
{}
package main

import (
    "fmt"
    "errors"
)
```

Ejemplo implementando errors.Unwrap() #2

2.- Creamos una estructura “*errorTwo*” que tiene un método “*Error()*”, por lo que implementa la interfaz de “*error*”.

```
type errorTwo struct{}

func (e errorTwo) Error() string {
    return "error two happened"
}
```

Ejemplo implementando errors.Unwrap() #3

3.- Luego declaramos nuestra función “*main*” y dentro creamos una instancia de la estructura “*errorTwo*” llamada “*e2*”.
Envolvemos esa instancia “*e2*” en otro error “*e1*”.
Finalmente, utilizando “*errors.Unwrap()*”, desenvolvemos “*e1*” para obtener “*e2*”.
Pero si intentamos desenvolver “*e2*” obtendremos “*nil*”.
Implementar Unwrap es útil para poder comparar correctamente los errores con los errors.As() y errors.Is()

{}

```
func main() {
   e2 := errorTwo{}
   e1 := fmt.Errorf("e2: %w", e2)
   fmt.Println(errors.Unwrap(e1))//print: error two happened
   fmt.Println(errors.Unwrap(e2))//print: <nil>
}
```

A codear...

3 | Retorno de errores

IT BOARDING

BOOTCAMP

// ¿Cúal es el rol de las funciones multi-retorno en el manejo de errores?

GO permite que las funciones devuelvan más de un valor. Entre los valores retornados, puede haber uno de tipo error.

IT BOARDING
BOOTCAMP

Para tener en cuenta:

Las funciones multi-retorno de GO nos permiten retornar, entre otros valores, un error.
Este error puede ser uno que nosotros mismos hayamos creado, o bien, alguno de los errores que dichas funciones retornan por defecto.

A diferencia de otros lenguajes que usan excepciones para el manejo de muchos errores, en Go es idiomático usar valores de retorno que indican errores siempre que sea posible.

Buenas prácticas:

En la práctica, usualmente, no creamos un error para utilizarlo de forma directa. Por lo general, lo creamos para utilizarlo como retorno de una función.

Por convención, cuando utilicemos funciones que retornan más de un valor, el error debe ser devuelto en última posición.

Si el error devuelto es != nil, los demás parámetros devueltos NUNCA deben usarse, ya que no son de confianza.

Sintaxis

Sintaxis

Para crear una función que devuelva más de un valor, incluyendo el retorno de un error, enumeramos los tipos de cada valor devuelto dentro de paréntesis en la firma de la función. Por ejemplo:

```
func FuncName() (string, error) {
   //return string, error
}
```

Recuerda que el “error” debe ser el último tipo retornado por nuestra función.

Ejemplo #1

Veamos un ejemplo paso a paso de cómo utilizar una función que retorna un tipo error junto con otro valor:
1.- Primero: definimos nuestro package “*main*” e importamos los packages “*fmt*” y “*errors*”.

```
{}
package main

import (
    "fmt"
    "errors"
)
```

Ejemplo #2

2.- Luego declaramos nuestra función “*SayHello*” que recibirá como argumento un “*string*” y retornará dos valores: un “*string*” y un “*error*”.

{}

```
func SayHello(name string) (string, error) {
     if name == "" {
          return "", errors.New("no name provided")
     }
     return fmt.Sprintf("Hi %s", name), nil
}
```

Ejemplo #3

3.- Finalmente generamos nuestra función “*main()*”. Dentro llamamos a nuestra función “*SayHello*”:

{}

```
func main() {
   name := "Meli"
   greeting, err := SayHello(name)
   if err != nil {
      fmt.Println("error: ", err)
}
   fmt.Println(greeting)
}
```

Para tener en cuenta...

Al devolver un error de una función con varios valores de retorno, el código idiomático GO también establecerá un “valor cero” para cada valor “non-error”. Los valores cero son, por ejemplo, una cadena vacía para *strings*, “0” para enteros, una estructura vacía para tipos de estructura y “*nil*” para tipos de punteros e interfaces, entre otros.

Manejo del error retornado

Manejando el retorno “*error*”

Cuando una función devuelve muchos valores, GO requiere que asignemos una variable a cada uno de ellos. Como vimos en nuestro ejemplo anterior, lo hacemos al proporcionar nombres para los dos valores que devuelve la función “*SayHello*”.

```
{}  greeting, err := SayHello(name)
```

El primer valor devuelto se asignará a la variable “*greeting*”, y el segundo valor (el error) se asignará a la variable “*err*”.

A veces, solo nos interesa el valor de error. En tal caso, podemos descartar cualquier valor no deseado que devuelva una función utilizando el identificador blank “_”.

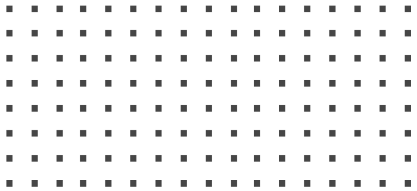
```
{}  _, err := SayHello(name)
```

Validaciones

Validando el retorno error #1

Al retornar un tipo *error* junto con otros valores en una función, es posible que dicho error efectivamente ocurra y, por ende, contenga algo. O bien, puede que no ocurra ningún problema y nuestro error se retorne como "*nil*".

En cualquier caso, es necesario realizar una validación de lo ocurrido, antes de continuar con nuestro código.



Validando el retorno error #2

Retomando el ejemplo anterior, dentro de la función "*main()*", utilizamos un condicional "*if*" para verificar si nuestro error ocurrió o retornó con valor cero ("*nil*").

{}

```
func main() {
    name := "Meli"
    greeting, err := SayHello(name)
    //validate if there was an error
    if err != nil {
        fmt.Println("cannot greet", err)
    }
    //if there was no error, we continue with our execution
    fmt.Println(greeting)
}
```

A codear...

Conclusiones

Hoy aprendimos cómo utilizar los errores en GO.

Conocimos el uso del paquete **errors** con sus métodos para poder manejar los errores.

También vimos cómo personalizar nuestros errores.

Actividad

Gracias.

IT BOARDING

BOOTCAMP