GO

BOOTCAMP

Clase en vivo

//Go Bases

IT BOARDING

BOOTCAMP

Objetivos de esta clase

- Profundizar el manejo de errores.
- Conocer qué es Panic en Golang.
- Conocer casos frecuentes que generan un Panic.
- Aprender cómo manejar un Panic.

Índice

IT BOARDING BOOTCAMP

1 | Repaso

IT BOARDING

BOOTCAMP

Revisión Práctica Errores

Actividad

2 | Panic: Bases

IT BOARDING

BOOTCAMP

Panic

Sintaxis de un panic()

Función

Nos permite declarar un panic, podemos personalizarlo dependiendo de las necesidades.

```
{}    panic("cause of panic")
```

Argumento

El argumento de un panic es del tipo interface, podemos utilizarlo para pasar la información que nos ayude a comprender el panic cuando se produzca.

Creando nuestros panic.

Un uso común de la función **panic()**, es abortar si una función devuelve un valor de error que, por algún motivo, no vamos a manejar (o porque aún no sabemos cómo hacerlo, o porque no tenemos interés en hacerlo, etc.).

Veamos un ejemplo, para el cual debemos, previamente, haber definido nuestro package **main** e importado los package **fmt** y **os**:

{}

```
func main() {
    fmt.Println("Starting... ")
    _, err := os.Open("no-file.txt")
    if err != nil {
        panic(err)
    }
    fmt.Println("End")
}
```

A codear...

3 | Panic: Casos

IT BOARDING

BOOTCAMP

Index Out of Bounds Panics #1/3

Este caso se da cuando intentamos acceder a un índice más allá de la longitud de un slice o la capacidad de un array.

En este caso, GO producirá un **panic** en tiempo de ejecución. Definamos nuestro package **main** e importemos el package **fmt** y veamos un ejemplo:

{}

```
func main() {
    animals := []string{
        "cow",
        "dog",
        "hawk",
    }
    fmt.Println("only flies on: ", animals[len(animals)])
}
```

Index Out of Bounds Panics #2/3

Al ejecutar nuestro programa obtendremos por consola una salida similar a esta:

```
panic: runtime error: index out of range [3] with length 3

goroutine 1 [running]:
main.main()
   /tmp/sandbox009542944/prog.go:13 +0x1b

Program exited: status 2.
```

Index Out of Bounds Panics #3/3

Al chequear la salida por consola vemos que:

1. Se produjo un **panic** al intentar acceder a un índice que excede la longitud del slice creado.
2. GO finalizó en tiempo de ejecución.
3. Las instrucciones posteriores al **panic** no se ejecutaron. Esto dado que el **panic**, al producirse, llama a las funciones diferidas por la función en pánico y aborta nuestro programa.

Receptores nulos #1/3

GO da la posibilidad de trabajar con punteros para referenciar a una instancia específica de algún tipo existente en la memoria del equipo en tiempo de ejecución. Los punteros pueden asumir el valor **nil** para indicar que no apuntan a nada. Cuando intentamos invocar métodos en un puntero que tenga el valor **nil**, se producirá **panic**. Veamos un ejemplo:

{}

```
type Dog struct {
   Name string
}
func (s *Dog) WoofWoof() {
   fmt.Println(s.Name, "Goes woof woof")
}
func main() {
   s := &Dog{"Sammy"}
   s = nil
   s.WoofWoof()
}
```

Receptores nulos #2/3

Al ejecutar nuestro programa obtendremos por consola una salida similar a esta:

{}

```
panic: runtime error: invalid memory address or nil pointer dereference
[signal SIGSEGV: segmentation violation code=0x1 addr=0x0 pc=0x497783]

goroutine 1 [running]:
main.(*Dog).WoofWoof(...)
   /tmp/sandbox602314289/prog.go:12
main.main()
   /tmp/sandbox602314289/prog.go:18 +0x23

Program exited: status 2.
```

Receptores nulos #3/3

Al chequear la salida por consola vemos que:

1. Se produjo un **panic** al intentar acceder a una dirección no válida de memoria o apuntar a un receptor nulo.
2. El **panic** se produce en tiempo de ejecución y aborta la ejecución del programa con **status 2**.

A codear...

4 | Panic: Cómo manejarlos

IT BOARDING

BOOTCAMP

// ¿Cómo manejar un panic?

Con las sentencias incorporadas “*defer*” y “*recover*” podemos controlar los efectos de un *panic* y evitar que nuestro programa finalice de modo no deseado.

Recuerda:

defer y **recover** son funciones incorporadas al lenguaje, específicamente diseñadas para evitar o controlar la naturaleza destructiva de un **panic**.

Si bien se presentan como funciones independientes, se requiere un uso complementario entre ambas para lograr resultados de mejor performance.

Defer

// ¿Qué es “*defer*”?

Es una sentencia incorporada en GO, que nos permite diferir la ejecución de ciertas funciones y “asegurar” que sean ejecutadas antes de la finalización de la ejecución de un programa.

Puede decirse que es el similar a **ensure** o **finally** utilizado en otros lenguajes de programación.

// ¿Para qué es útil y cómo se utiliza?

Es de gran utilidad para asegurarnos de limpiar recursos durante la ejecución de nuestro programa, incluso ante la ocurrencia de un **panic**.

Se utiliza como mecanismo de seguridad para brindar protección contra los cortes de ejecución y salidas abruptas que generan los **panics**.

Las funciones se difieren invocándolas de la forma habitual y añadiendo luego un prefijo a toda la instrucción con la palabra clave **defer**.

Defer. Ejemplo #1

Definamos nuestro package **main**, importemos el package **fmt** y probemos un ejemplo de función diferida usando **defer**:

{}

```
func main() {
    //apply “defer” to the invocation of an anonymous function
    defer func() {
        fmt.Println("This function is executed despite a panic occurring")
    }()
    //create a panic with a message that it occurred
    panic("panic occurred!!!")
}
```

Defer. Ejemplo #2

Al ejecutar nuestro programa obtendremos por consola una salida similar a esta:

```
This function is executed despite a panic occurring
panic: panic occurred!!!

goroutine 1 [running]:
main.main()
   /tmp/sandbox501206329/prog.go:13 +0x5b

Program exited: status 2.
```

Defer. Ejemplo #3

Al chequear la salida por consola vemos que:

1. La función anónima diferida se ejecuta e imprime su mensaje.
2. Se encuentra el **panic** que se generó en nuestra función **main**.
3. Se produjo la salida del programa con **status 2**.

Tener en cuenta:

Si bien las funciones diferidas se ejecutan, incluso, ante la producción de **panics**, NO se ejecutan ante la ejecución de la función **log.Fatal().**

En caso de haberse diferido varias funciones, al ser llamadas, se ejecutarán en orden, iniciando desde la última función diferida hacia la primera de ellas.

A codear...

Recover

// ¿Qué es “*recover*” y para qué es útil?

Es una función incorporada que permite interceptar un **panic** y evitar que este termine con la ejecución del programa en forma inesperada o no deseada.

Al ser parte del paquete incluido en GO, puede invocarse sin importar paquetes adicionales.

// ¿Cómo se utiliza?

Lo correcto es utilizar la función incorporada **recover** dentro de una declaración **defer**. De este modo, al producirse un **panic**, esa función diferida recuperará el control de la rutina en pánico, y el valor establecido en **panic**, y evitaremos que nuestro programa termine de forma no deseada.

Si utilizáramos **recover** fuera de una declaración **defer**, la producción de un panic terminaría con la ejecución del programa antes de que **recover** puede recuperar el valor de **panic**. Es decir, en este caso, **recover** retornaría **nil** y no evitaría la finalización abrupta de la ejecución.

Recover. Ejemplo #1

Definamos nuestro package **main**, importemos el package **fmt** y probemos un ejemplo de **recover**.

Para esto vamos a declarar una función llamada **isPair()** que recibirá como argumento un número entero y analizará si es par o no.

Caso de ser impar, producirá **panic** y llamará a la función anónima diferida que contiene la función **recover**.

El *panic* será controlado y su valor recuperado por **recover** y asignado a la variable **err**.

Al ser **err** distinto de **nil**, se imprimirá por consola el valor recuperado del **panic** producido. La función diferida finalizará su ejecución y el programa continuará la suya.

Recover. Ejemplo #2

{}

```
func isPair(num int) {
    defer func() {
        err := recover()
        if err != nil {
            fmt.Println(err)
        }
    }()
    if (num % 2) != 0 {
        panic("Not an even number")
    }
    fmt.Println(num, "is an even number!")
}
```

Recover. Ejemplo #3

En nuestra función **main(),** llamaremos a la función **isPair()** y le pasaremos como argumento un número impar para que genere **panic**.

Veremos que el **panic** generado en la función **isPair()** es controlado por **recover**, y no se aborta la ejecución de nuestra **main()**.

{}

```
func main() {
    num := 3
    isPair(num)
    fmt.Println("Execution completed!")
}
```

Recover. Ejemplo #4

Al ejecutar nuestro programa obtendremos por consola una salida similar a esta:

```
Not an even number
Execution completed!

Program exited.
```

Observa como el valor de **panic** fue recuperado por **recover** y el programa completó su ejecución hasta el final.

A codear...

Conclusiones

En esta clase pudimos cerrar los conceptos de manejos de errores y aprender sobre los Panics.

Repasamos los usos y manipulación al momento de interactuar con Panics.

Gracias.

IT BOARDING

BOOTCAMP