Interfaces en Go

Juan Ignacio Roldán Catalini

¿ Qué es una interfaz en Go ?

Un type de interface en Go es como una definición. Define y describe los métodos exactos que debe tener algún otro tipo.

Ejemplo de interface type de la librería standard fmt

```
type Stringer interface {

        String() string

}
```

https://pkg.go.dev/fmt#Stringer

Siguiendo el ejemplo, podemos decir que algo satisface a esta interfaz si posee un método con la definición exacta: `String() string`

Ejemplo, El siguiente type Book satisface la interface, ya que posee un método `String() string`

```go
type Book struct {
    Title  string
    Author string
}

func (b Book) String() string {
    return fmt.Sprintf("Book: %s - %s", b.Title, b.Author)
}
```

O, como otro ejemplo, el siguiente type Count también satisface la interfaz fmt.Stringer, nuevamente porque tiene un método String ().

```
type Count int

func (c Count) String() string {
    return strconv.Itoa(int(c))
}
```

También se puede pensar en esto al revés. Si sabes que un objeto satisface la interfaz fmt.Stringer, puedes confiar en que tiene un método String() al que puedes llamar.

Cuando veas una una declaración en Go (como una variable, parámetro de una función o struct field) que tenga un interface type, puedes usar un objeto de cualquier type siempre y cuando satisfaga la interface.

Por ejemplo, digamos que tenemos la siguiente función:

```
func WriteLog(s fmt.Stringer) {

    log.Println(s.String())

}
```

Debido a que esta función `WriteLog()` usa el tipo de interfaz `fmt.Stringer` en su declaración de parámetros, podemos pasar cualquier objeto que satisfaga la interfaz `fmt.Stringer`. Por ejemplo, podríamos pasar cualquiera de los tipos `Book` y `Count` al método WriteLog (), y el código funcionaria correctamente.

Además, debido a que el objeto pasado satisface la `fmt.Stringer` interface, ya sabemos que este tiene un método `String() string` que `WriteLog()` puede llamar con seguridad.

```go
// Declaramos un Book type que satisface la fmt.Stringer interface.
type Book struct {
    Title  string
    Author string
}
func (b Book) String() string {
    return fmt.Sprintf("Book: %s - %s", b.Title, b.Author)
}
// Declaramos Count type que satisface fmt.Stringer interface.
type Count int

func (c Count) String() string {
    return strconv.Itoa(int(c))
}
// Declaramos la funcion WriteLog() que toma un objeto que satisface
// la fmt.Stringer interface
func WriteLog(s fmt.Stringer) {
    log.Println(s.String())
}
func main() {
    // Initialize a Book object and pass it to WriteLog().
    book := Book{"Alice in Wonderland", "Lewis Carrol"}
    WriteLog(book)

    // Initialize a Count object and pass it to WriteLog().
    count := Count(3)
    WriteLog(count)
}
```

Puedes correr el snippet anterior aqui

El punto clave a remarcar aquí es que usando un interface type en la declaración de nuestra función `WriteLog()` , hemos hecho a la función agnóstica(o flexible) con respecto al type exacto de objeto que recibe. Todo lo que importa son los métodos que posee.

¿ Por qué son útiles las interfaces ?

Hay muchas razones por las que puedes usar interfaces. Entre ellas podemos nombrar las más comunes:

1. Nos ayudan a reducir código duplicado (boilerplate code)
2. Hacer más fácil el uso de mocks en lugar de objetos reales en unit tests
3. Una herramienta que útil para la arquitectura.

¿ Qué es la interfaz vacía ?

Al principio de la presentación dijimos que Un type de interface en Go es como una definición. Define y describe los métodos exactos que debe tener algún otro tipo.

La interfaz vacía describe esencialmente que la ausencia de métodos. No tiene reglas. Y por lo tanto, cualquier (y todo) objeto satisface la interfaz vacía.

En otras palabras, la interfaz vacía `interface{}` es como un tipo de comodín.

Veamos el siguiente código:

```go
package main

import "fmt"


func main() {
    person := make(map[string]interface{}, 0)

    person["name"] = "Alice"
    person["age"] = 21
    person["height"] = 167.64

    fmt.Printf("%+v", person)
}
```

En el snippet de código anterior se inicializa un `person` map, que usa el type string para las keys, y un empty interface type `interface{}` para los values. Se le asignaron diferentes types como values del map(`string`, `int` y `float32`). ***Esto es posible ya que objetos de cualquier type satisfacen la interfaz vacía.***

Puedes ver la salida del código aquí.

Hay algo importante que remarcar cuando necesitamos recuperar y usar un valor de este map.

Por ejemplo, digamos que queremos el valor “age” e incrementarlo en uno.. Si escribes algo como en el siguiente snippet, fallará al compilarse

interfaces - juan ignacio roldán

```go
package main

import "log"

func main() {
    person := make(map[string]interface{}, 0)

    person["name"] = "Alice"
    person["age"] = 21
    person["height"] = 167.64

    person["age"] = person["age"] + 1

    fmt.Printf("%+v", person)
}
```

Y obtendrás un mensaje de error como este:

```
invalid operation: person["age"] + 1 (mismatched types interface {} and int)
```

Esto sucede porque el value almacenado en el map toma el type `interface{}` y deja de tener su tipo de `int` original, subyacente. Debido a que ya no es un tipo int, no podemos agregarle 1.

Para solucionarlo, necesitas hacer un type assert del value a un `int` antes de usarlo, como se muestra en el siguiente snippet

interfaces empty - juan ignacio roldán

```go
package main

import "log"

func main() {
    person := make(map[string]interface{}, 0)

    person["name"] = "Alice"
    person["age"] = 21
    person["height"] = 167.64

    age, ok := person["age"].(int)
    if !ok {
        log.Fatal("could not assert value to int")
        return
    }

    person["age"] = age + 1

    log.Printf("%+v", person)
}
```

Puedes correr el código [aquí]

Entonces, cuando deberíamos usar empty interfaces en nuestro código ?

La respuesta es probablemente no muy seguido. Como regla general, es más claro, seguro y performante usar concrete types en su lugar. En el snippet anterior podríamos haber definido una Person struct con los campos relevantes

```
type Person struct {
    Name   string
    Age    int
    Height float32
}
```

Dicho esto, la interfaz vacía es útil en situaciones donde necesitas aceptar y trabajar con “ valores impredecibles ó user - defined types . Puedes encontrar esto en varios lugar de la librería standard. Ejemplos de ello: `gob.Encode`, `fmt.Print` and `template.Execute` functions

¡Gracias!