EBook Gratis

APRENDIZAJE Go

Free unaffiliated eBook created from
Stack Overflow contributors.

#go

Tabla de contenido

Acerca de

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: go

It is an unofficial and free Go ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official Go.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Capítulo 1: Empezando con Go

Observaciones

Go es un lenguaje de código abierto, compilado y tipificado estáticamente según la tradición de Algol y C. Cuenta con características como recolección de basura, tipificación estructural limitada, funciones de seguridad de la memoria y programación concurrente de estilo CSP fácil de usar.

Versiones

La última versión de la versión principal está en negrita a continuación. Historial de lanzamiento completo se puede encontrar aquí .

Versión	Fecha de lanzamiento
1.8.3	2017-05-24
1.8.0	2017-02-16
1.7.0	2016-08-15
1.6.0	2016-02-17
1.5.0	2015-08-19
1.4.0	2014-12-04
1.3.0	2014-06-18
1.2.0	2013-12-01
1.1.0	2013-05-13
1.0.0	2012-03-28

Examples

¡Hola Mundo!

Coloque el siguiente código en un nombre de archivo `hello.go` :

```
package main

import "fmt"

func main() {
```

```
    fmt.Println("Hello, □□")
}
```

Patio de recreo

Cuando Go está instalado correctamente, este programa se puede compilar y ejecutar de la siguiente manera:

```
go run hello.go
```

Salida:

```
Hello, □□
```

Una vez que esté satisfecho con el código, se puede compilar en un ejecutable ejecutando:

```
go build hello.go
```

Esto creará un archivo ejecutable adecuado para su sistema operativo en el directorio actual, que luego puede ejecutar con el siguiente comando:

Linux, OSX y otros sistemas similares a Unix

```
./hello
```

Windows

```
hello.exe
```

***Nota** : los caracteres chinos son importantes porque demuestran que las cadenas Go se almacenan como segmentos de solo lectura de bytes.*

FizzBuzz

Otro ejemplo de los programas de estilo "Hello World" es FizzBuzz . Este es un ejemplo de una implementación de FizzBuzz. Muy idiomático entra en juego aquí.

```
package main

// Simple fizzbuzz implementation

import "fmt"

func main() {
    for i := 1; i <= 100; i++ {
        s := ""
        if i % 3 == 0 {
            s += "Fizz"
```

```
        }
        if i % 5 == 0 {
            s += "Buzz"
        }
        if s != "" {
            fmt.Println(s)
        } else {
            fmt.Println(i)
        }
    }
}
```

Patio de recreo

Listado de variables de entorno de Go

Las variables de entorno que afectan a la herramienta `go` pueden verse a través del comando `go env [var ...]` :

```
$ go env
GOARCH="amd64"
GOBIN="/home/yourname/bin"
GOEXE=""
GOHOSTARCH="amd64"
GOHOSTOS="linux"
GOOS="linux"
GOPATH="/home/yourname"
GORACE=""
GOROOT="/usr/lib/go"
GOTOOLDIR="/usr/lib/go/pkg/tool/linux_amd64"
CC="gcc"
GOGCCFLAGS="-fPIC -m64 -pthread -fmessage-length=0 -fdebug-prefix-map=/tmp/go-
build059426571=/tmp/go-build -gno-record-gcc-switches"
CXX="g++"
CGO_ENABLED="1"
```

Por defecto imprime la lista como un script de shell; sin embargo, si uno o más nombres de variables se dan como argumentos, imprime el valor de cada variable nombrada.

```
$go env GOOS GOPATH
linux
/home/yourname
```

Configurando el medio ambiente

Si Go no está preinstalado en su sistema, puede ir a https://golang.org/dl/ y elegir su plataforma para descargar e instalar Go.

Para configurar un entorno de desarrollo básico de Go, solo deben configurarse algunas de las muchas variables de entorno que afectan el comportamiento de la herramienta de `go` (consulte: Listado de variables de entorno de Go para obtener una lista completa) (generalmente en el `~/.profile` su shell archivo, o equivalente en sistemas operativos similares a Unix).

GOPATH

Al igual que la `PATH` entorno `PATH` del sistema, la ruta de Go es una `:` (`;` en Windows) lista delimitada de directorios donde Go buscará paquetes. La herramienta `go get` también descargará los paquetes al primer directorio de esta lista.

`GOPATH` es donde Go configurará las carpetas `bin` , `pkg` y `src` asociadas necesarias para el área de trabajo:

- `src` - ubicación de los archivos de origen: `.go` , `.c` , `.g` , `.s`
- `pkg` - ha compilado archivos `.a`
- `bin` - contiene archivos ejecutables construidos por Go

A partir de Go 1.8, la variable de entorno `GOPATH` tendrá un valor predeterminado si no está configurada. El valor predeterminado es $ HOME / go en Unix / Linux y% USERPROFILE% / go en Windows.

Algunas herramientas asumen que `GOPATH` contendrá un solo directorio.

GOBIN

El directorio bin donde `go install` y `go get` a colocar colocará los binarios después de construir los paquetes `main` . En general, esto se establece en algún lugar del sistema `PATH` para que los binarios instalados puedan ejecutarse y descubrirse fácilmente.

GOROOT

Esta es la ubicación de su instalación Go. Se utiliza para encontrar las bibliotecas estándar. Es muy raro tener que configurar esta variable cuando Go incrusta la ruta de compilación en la cadena de herramientas. La configuración de `GOROOT` es necesaria si el directorio de instalación difiere del directorio de compilación (o el valor establecido al compilar).

Acceso a la documentación fuera de línea

Para la documentación completa, ejecute el comando:

```
godoc -http=:<port-number>
```

Para un tour de Go (muy recomendable para principiantes en el idioma):

```
go tool tour
```

Los dos comandos anteriores iniciarán los servidores web con documentación similar a la que se encuentra en línea aquí y aquí, respectivamente.

Para una consulta de referencia rápida desde la línea de comandos, por ejemplo, para fmt.Print:

```
godoc cmd/fmt Print
```

```
# or
go doc fmt Print
```

La ayuda general también está disponible desde la línea de comandos:

```
go help [command]
```

Running Go en línea

El patio de juegos Go

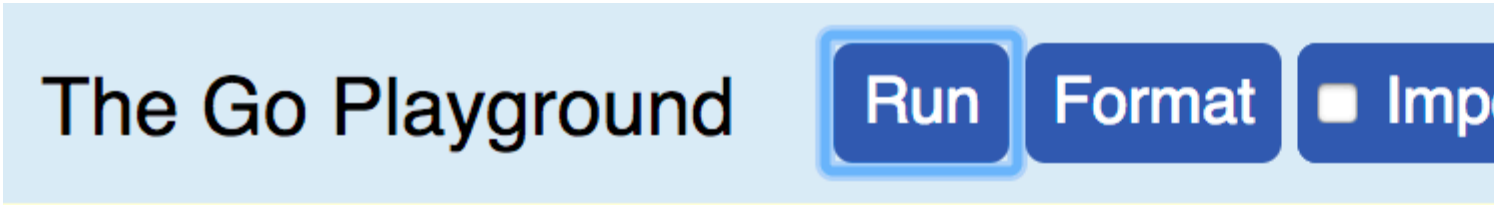
Una herramienta poco conocida de Go es **The Go Playground** . Si uno quiere experimentar con Go sin descargarlo, puede hacerlo fácilmente simplemente. . .

1. Visitando el Patio de Juegos en su navegador web.
2. Ingresando su código
3. Al hacer clic en "Ejecutar"

Compartiendo tu codigo

El Go Playground también tiene herramientas para compartir; Si un usuario presiona el botón "Compartir", se generará un enlace (como este) que se puede enviar a otras personas para probar y editar.

En acción




```
1  package main
2
3  import (
4          "fmt"
5  )
6
7  func main() {
8          fmt.Println("Hello, playgroun
9  }
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
```

Capítulo 2: Agrupación de memoria

Introducción

sync.Pool almacena un caché de elementos asignados pero no utilizados para uso futuro, evitando la pérdida de memoria para colecciones cambiadas con frecuencia y permitiendo la reutilización eficiente y segura de la memoria por subprocesos. Es útil administrar un grupo de elementos temporales compartidos entre clientes concurrentes de un paquete, por ejemplo, una lista de conexiones de base de datos o una lista de buffers de salida.

Examples

sync.Pool

Usando la estructura `sync.Pool` podemos agrupar objetos y reutilizarlos.

```
package main

import (
    "bytes"
    "fmt"
    "sync"
)

var pool = sync.Pool{
    // New creates an object when the pool has nothing available to return.
    // New must return an interface{} to make it flexible. You have to cast
    // your type after getting it.
    New: func() interface{} {
        // Pools often contain things like *bytes.Buffer, which are
        // temporary and re-usable.
        return &bytes.Buffer{}
    },
}

func main() {
    // When getting from a Pool, you need to cast
    s := pool.Get().(*bytes.Buffer)
    // We write to the object
    s.Write([]byte("dirty"))
    // Then put it back
    pool.Put(s)

    // Pools can return dirty results

    // Get 'another' buffer
    s = pool.Get().(*bytes.Buffer)
    // Write to it
    s.Write([]bytes("append"))
    // At this point, if GC ran, this buffer *might* exist already, in
    // which case it will contain the bytes of the string "dirtyappend"
    fmt.Println(s)
    // So use pools wisely, and clean up after yourself
```

```
    s.Reset()
    pool.Put(s)

    // When you clean up, your buffer should be empty
    s = pool.Get().(*bytes.Buffer)
    // Defer your Puts to make sure you don't leak!
    defer pool.Put(s)
    s.Write([]byte("reset!"))
    // This prints "reset!", and not "dirtyappendreset!"
    fmt.Println(s)
}
```

Lea Agrupación de memoria en línea: https://riptutorial.com/es/go/topic/4647/agrupacion-de-memoria

Capítulo 3: Análisis de archivos CSV

Sintaxis

- csvReader: = csv.NewReader (r)
- datos, err: = csvReader.Read ()

Examples

CSV simple de análisis

Considere estos datos CSV:

```
#id,title,text
1,hello world,"This is a ""blog""."
2,second time,"My
second
entry."
```

Estos datos se pueden leer con el siguiente código:

```
// r can be any io.Reader, including a file.
csvReader := csv.NewReader(r)
// Set comment character to '#'.
csvReader.Comment = '#'
for {
    post, err := csvReader.Read()
    if err != nil {
        log.Println(err)
        // Will break on EOF.
        break
    }
    fmt.Printf("post with id %s is titled %q: %q\n", post[0], post[1], post[2])
}
```

Y producir:

```
post with id 1 is titled "hello world": "This is a \"blog\"."
post with id 2 is titled "second time": "My\nsecond\nentry."
2009/11/10 23:00:00 EOF
```

Área de juegos: https://play.golang.org/p/d2E6-CGGIe .

Lea Análisis de archivos CSV en línea: https://riptutorial.com/es/go/topic/5818/analisis-de-archivos-csv

Capítulo 4: Análisis de argumentos de línea de comando y banderas

Examples

Argumentos de línea de comando

El análisis de los argumentos de la línea de comando es Go es muy similar a otros idiomas. En su código, solo tiene acceso a una porción de argumentos donde el primer argumento será el nombre del programa en sí.

Ejemplo rápido:

```
package main

import (
    "fmt"
    "os"
)

func main() {

    progName := os.Args[0]
    arguments := os.Args[1:]

    fmt.Printf("Here we have program '%s' launched with following flags: ", progName)

    for _, arg := range arguments {
        fmt.Printf("%s ", arg)
    }

    fmt.Println("")
}
```

Y la salida sería:

```
$ ./cmd test_arg1 test_arg2
Here we have program './cmd' launched with following flags: test_arg1 test_arg2
```

Cada argumento es sólo una cadena. En `os` paquete que parece: `var Args []string`

Banderas

Ir biblioteca estándar proporciona el paquete `flag` que ayuda con banderas de análisis pasados al programa.

Tenga en cuenta que el paquete de `flag` no proporciona las banderas habituales de estilo GNU. Eso significa que las banderas de varias letras deben iniciarse con un guión único como este: `-exampleflag` , no esto: `--exampleflag` . Las banderas de estilo GNU se pueden hacer con algún

paquete de terceros.

```
package main

import (
    "flag"
    "fmt"
)

func main() {

    // basic flag can be defined like this:
    stringFlag := flag.String("string.flag", "default value", "here comes usage")
    // after that stringFlag variable will become a pointer to flag value

    // if you need to store value in variable, not pointer, than you can
    // do it like:
    var intFlag int
    flag.IntVar(&intFlag, "int.flag", 1, "usage of intFlag")

    // after all flag definitions you must call
    flag.Parse()

    // then we can access our values
    fmt.Printf("Value of stringFlag is: %s\n", *stringFlag)
    fmt.Printf("Value of intFlag is: %d\n", intFlag)

}
```

`flag` ayuda con un mensaje:

```
$ ./flags -h
Usage of ./flags:
  -int.flag int
        usage of intFlag (default 1)
  -string.flag string
        here comes usage (default "default value")
```

Llamar con todas las banderas:

```
$ ./flags -string.flag test -int.flag 24
Value of stringFlag is: test
Value of intFlag is: 24
```

Llamada con banderas que faltan:

```
$ ./flags
Value of stringFlag is: default value
Value of intFlag is: 1
```

Lea Análisis de argumentos de línea de comando y banderas en línea:
https://riptutorial.com/es/go/topic/4023/analisis-de-argumentos-de-linea-de-comando-y-banderas

Capítulo 5: Aplazar

Introducción

Una declaración `defer` empuja una llamada de función a una lista. La lista de llamadas guardadas se ejecuta después de que vuelve la función que la rodea. El aplazamiento se usa comúnmente para simplificar las funciones que realizan varias acciones de limpieza.

Sintaxis

- diferir someFunc (args)
- aplazar func () {// el código va aquí} ()

Observaciones

La función de `defer` inyecta un nuevo marco de pila (la función llamada después de la palabra clave de `defer`) en la pila de llamadas debajo de la función que se está ejecutando actualmente. Esto significa que se garantiza que el aplazamiento se ejecute siempre que la pila se desenrolle (si su programa falla o obtiene un `SIGKILL` , el aplazamiento no se ejecutará).

Examples

Diferir lo básico

Una *declaración diferida* en Go es simplemente una llamada de función marcada para ejecutarse en un momento posterior. La declaración diferida es una llamada de función ordinaria prefijada por la palabra clave `defer` .

```
defer someFunction()
```

Una función diferida se ejecuta una vez la función que contiene los `defer` retorne el comando. La llamada real a la función diferida se produce cuando la función de cierre:

- ejecuta una declaración de retorno
- se cae del final
- pánico

Ejemplo:

```
func main() {
    fmt.Println("First main statement")
    defer logExit("main") // position of defer statement here does not matter
    fmt.Println("Last main statement")
}
```

```
func logExit(name string) {
    fmt.Printf("Function %s returned\n", name)
}
```

Salida:

```
First main statement
Last main statement
Function main returned
```

Si una función tiene varias declaraciones diferidas, forman una pila. El último `defer` es el primero que se ejecuta después de que se devuelve la función de cierre, seguido de las llamadas subsiguientes a los `defer` anteriores en orden (a continuación, el ejemplo devuelve causando un pánico):

```
func main() {
    defer logNum(1)
    fmt.Println("First main statement")
    defer logNum(2)
    defer logNum(3)
    panic("panic occurred")
    fmt.Println("Last main statement") // not printed
    defer logNum(3) // not deferred since execution flow never reaches this line
}

func logNum(i int) {
    fmt.Printf("Num %d\n", i)
}
```

Salida:

```
First main statement
Num 3
Num 2
Num 1
panic: panic occurred

goroutine 1 [running]:
....
```

Tenga en cuenta que las funciones diferidos han evaluado sus argumentos en el momento `defer` ejecuta:

```
func main() {
    i := 1
    defer logNum(i) // deferred function call: logNum(1)
    fmt.Println("First main statement")
    i++
    defer logNum(i) // deferred function call: logNum(2)
    defer logNum(i*i) // deferred function call: logNum(4)
    return // explicit return
}

func logNum(i int) {
    fmt.Printf("Num %d\n", i)
```

```
}
```

Salida:

```
First main statement
Num 4
Num 2
Num 1
```

Si una función ha nombrado valores de retorno, una función anónima diferida dentro de esa función puede acceder y actualizar el valor devuelto incluso después de que la función haya regresado:

```
func main() {
    fmt.Println(plusOne(1)) // 2
    return
}

func plusOne(i int) (result int) { // overkill! only for demonstration
    defer func() {result += 1}() // anonymous function must be called by adding ()

    // i is returned as result, which is updated by deferred function above
    // after execution of below return
    return i
}
```

Finalmente, una sentencia de `defer` generalmente se usa en operaciones que a menudo ocurren juntas Por ejemplo:

- abrir y cerrar un archivo
- conectar y desconectar
- bloquear y desbloquear un mutex
- marcar un grupo de espera como hecho (`defer wg.Done()`)

Este uso garantiza la liberación adecuada de los recursos del sistema independientemente del flujo de ejecución.

```
resp, err := http.Get(url)
if err != nil {
return err
}
defer resp.Body.Close() // Body will always get closed
```

Llamadas de función diferida

Las llamadas a funciones diferidas tienen un propósito similar a cosas como los bloques `finally` en lenguajes como Java: aseguran que alguna función se ejecutará cuando se devuelva la función externa, independientemente de si se produjo un error o qué declaración de devolución se golpeó en casos con múltiples devoluciones. Esto es útil para limpiar recursos que deben cerrarse como conexiones de red o punteros de archivos. La palabra clave `defer` indica una llamada a función diferida, de manera similar a la palabra clave `go` inicia una nueva goroutina. Al igual que

una llamada `go` , los argumentos de la función se evalúan inmediatamente, pero a diferencia de la llamada `go` , las funciones diferidas no se ejecutan simultáneamente.

```
func MyFunc() {
    conn := GetConnection()    // Some kind of connection that must be closed.
    defer conn.Close()        // Will be executed when MyFunc returns, regardless of how.
    // Do some things...
    if someCondition {
        return                // conn.Close() will be called
    }
    // Do more things
}// Implicit return - conn.Close() will still be called
```

Tenga en cuenta el uso de `conn.Close()` lugar de `conn.Close` : no solo está pasando una función, está aplazando una *llamada de* función completa, incluidos sus argumentos. Las múltiples llamadas de función se pueden diferir en la misma función externa, y cada una se ejecutará una vez en orden inverso. También puede aplazar los cierres, ¡no se olvide de los parens!

```
defer func(){
    // Do some cleanup
}()
```

Lea Aplazar en línea: https://riptutorial.com/es/go/topic/2795/aplazar

Capítulo 6: Archivo I / O

Sintaxis

- archivo, err: = os.Open (*nombre*) // Abre un archivo en modo de solo lectura. Se devuelve un error no nulo si no se pudo abrir el archivo.
- archivo, err: = os.Create (*nombre*) // Crea o abre un archivo si ya existe en modo de solo escritura. El archivo se sobrescribe si ya existe. Se devuelve un error no nulo si no se pudo abrir el archivo.
- file, err: = os.OpenFile (*name* , *flags* , *perm*) // Abre un archivo en el modo especificado por las banderas. Se devuelve un error no nulo si no se pudo abrir el archivo.
- data, err: = ioutil.ReadFile (*name*) // Lee todo el archivo y lo devuelve. Se devuelve un error no nulo si no se pudo leer el archivo completo.
- err: = ioutil.WriteFile (*name* , *data* , *perm*) // Crea o sobrescribe un archivo con los datos proporcionados y los bits de permiso de UNIX. Se devuelve un error no nulo si el archivo no se pudo escribir.
- err: = os.Remove (*name*) // Borra un archivo. Se devuelve un error no nulo si el archivo no se pudo eliminar.
- err: = os.RemoveAll (*name*) // Borra un archivo o toda la jerarquía de directorios. Se devuelve un error no nulo si no se pudo eliminar el archivo o directorio.
- err: = os.Rename (*oldName* , *newName*) // *Renombra* o mueve un archivo (puede ser a través de directorios). Se devuelve un error no nulo si el archivo no se pudo mover.

Parámetros

Parámetro	Detalles
nombre	Un nombre de archivo o ruta de acceso de tipo cadena. Por ejemplo: `"hello.txt"` .
errar	Un `error` Si no es `nil` , representa un error que ocurrió cuando se llamó a la función.
expediente	Un controlador de archivos de tipo `*os.File` devuelto por las funciones relacionadas con el archivo del paquete `os` . Implementa un `io.ReadWriter` , lo que significa que puede llamar a `Read(data)` y `Write(data)` en él. Tenga en cuenta que es posible que no se puedan llamar a estas funciones dependiendo de los indicadores abiertos del archivo.
datos	Un segmento de bytes (`[]byte`) que representa los datos sin procesar de un archivo.
permanente	Los bits de permiso de UNIX utilizados para abrir un archivo de tipo `os.FileMode` . Hay varias constantes disponibles para ayudar con el uso de bits de permiso.

Parámetro	Detalles
bandera	Los indicadores de apertura de archivos que determinan los métodos a los que se puede llamar en el controlador de archivos de tipo `int` . Varias constantes están disponibles para ayudar con el uso de banderas. Estos son: `os.O_RDONLY` , `os.O_WRONLY` , `os.O_RDWR` , `os.O_APPEND` , `os.O_CREATE` , `os.O_EXCL` , `os.O_SYNC` y `os.O_TRUNC` .

Examples

Leer y escribir en un archivo usando ioutil.

Un programa simple que escribe "¡Hola mundo!" to `test.txt` , lee los datos y los imprime. Demuestra operaciones simples de E / S de archivos.

```
package main

import (
    "fmt"
    "io/ioutil"
)

func main() {
    hello := []byte("Hello, world!")

    // Write `Hello, world!` to test.txt that can read/written by user and read by others
    err := ioutil.WriteFile("test.txt", hello, 0644)
    if err != nil {
        panic(err)
    }

    // Read test.txt
    data, err := ioutil.ReadFile("test.txt")
    if err != nil {
        panic(err)
    }

    // Should output: `The file contains: Hello, world!`
    fmt.Println("The file contains: " + string(data))
}
```

Listado de todos los archivos y carpetas en el directorio actual.

```
package main

import (
    "fmt"
    "io/ioutil"
)

func main() {
    files, err := ioutil.ReadDir(".")
    if err != nil {
        panic(err)
```

```
    }

    fmt.Println("Files and folders in the current directory:")

    for _, fileInfo := range files {
        fmt.Println(fileInfo.Name())
    }
}
```

Listado de todas las carpetas en el directorio actual

```
package main

import (
    "fmt"
    "io/ioutil"
)

func main() {
    files, err := ioutil.ReadDir(".")
    if err != nil {
        panic(err)
    }

    fmt.Println("Folders in the current directory:")

    for _, fileInfo := range files {
        if fileInfo.IsDir() {
            fmt.Println(fileInfo.Name())
        }
    }
}
```

Lea Archivo I / O en línea: https://riptutorial.com/es/go/topic/1033/archivo-i---o

Capítulo 7: Arrays

Introducción

Las matrices son tipos de datos específicos, que representan una colección ordenada de elementos de otro tipo.

En Go, los arreglos pueden ser simples (a veces llamados "listas") o multidimensionales (como, por ejemplo, un arreglo en 2 dimensiones representa un conjunto ordenado de arreglos, que contiene elementos)

Sintaxis

- var variableName [5] ArrayType // Declarar una matriz de tamaño 5.
- var variableName [2] [3] ArrayType = {{Value1, Value2, Value3}, {Value4, Value5, Value6}} // Declarar una matriz multidimensional
- variableName: = [...] ArrayType {Value1, Value2, Value3} // Declarar una matriz de tamaño 3 (El compilador contará los elementos de la matriz para definir el tamaño)
- arrayName [2] // Obteniendo el valor por índice.
- arrayName [5] = 0 // Estableciendo el valor en el índice.
- arrayName [0] // Primer valor de la matriz
- arrayName [len (arrayName) -1] // Último valor de la matriz

Examples

Creando matrices

Una matriz en go es una colección ordenada de elementos del mismo tipo.
La notación básica para representar arrays es usar `[]` con el nombre de la variable.

Crear una nueva matriz se parece a `var array = [size]Type` , reemplazando `size` por un número (por ejemplo, `42` para especificar que será una lista de 42 elementos), y reemplazando `Type` por el tipo de elementos que la matriz puede contener (para ejemplo `int` o `string`)

Justo debajo, hay un ejemplo de código que muestra la forma diferente de crear una matriz en Go.

```
// Creating arrays of 6 elements of type int,
// and put elements 1, 2, 3, 4, 5 and 6 inside it, in this exact order:
var array1 [6]int = [6]int {1, 2, 3, 4, 5, 6} // classical way
var array2 = [6]int {1, 2, 3, 4, 5, 6} // a less verbose way
var array3 = [...]int {1, 2, 3, 4, 5, 6} // the compiler will count the array elements by
itself

fmt.Println("array1:", array1) // > [1 2 3 4 5 6]
fmt.Println("array2:", array2) // > [1 2 3 4 5 6]
fmt.Println("array3:", array3) // > [1 2 3 4 5 6]
```

```
// Creating arrays with default values inside:
zeros := [8]int{}               // Create a list of 8 int filled with 0
ptrs := [8]*int{}               // a list of int pointers, filled with 8 nil references (
<nil> )
emptystr := [8]string{}         // a list of string filled with 8 times ""

fmt.Println("zeroes:", zeros)      // > [0 0 0 0 0 0 0 0]
fmt.Println("ptrs:", ptrs)         // > [<nil> <nil> <nil> <nil> <nil> <nil> <nil> <nil>]
fmt.Println("emptystr:", emptystr) // > [       ]
// values are empty strings, separated by spaces,
// so we can just see separating spaces


// Arrays are also working with a personalized type
type Data struct {
    Number int
    Text   string
}

// Creating an array with 8 'Data' elements
// All the 8 elements will be like {0, ""} (Number = 0, Text = "")
structs := [8]Data{}

fmt.Println("structs:", structs) // > [{0 } {0 } {0 } {0 } {0 } {0 } {0 } {0 }]
// prints {0 } because Number are 0 and Text are empty; separated by a space
```

jugar en el patio de recreo

Array Multidimensional

Las matrices multidimensionales son básicamente matrices que contienen otras matrices como elementos.
Se representa como el tipo `[sizeDim1][sizeDim2]..[sizeLastDim]type` , reemplazando `sizeDim` por números correspondientes a la longitud de la dimensión, y `type` por el tipo de datos en la matriz multidimensional.

Por ejemplo, `[2][3]int` representa una matriz compuesta de **2 subrays** de **3 elementos tipeados int** .
Básicamente puede ser la representación de una matriz de **2 líneas** y **3 columnas** .

Por lo tanto, podemos hacer una matriz numérica de grandes dimensiones como `var values := [2017][12][31][24][60]int` por ejemplo, si necesita almacenar un número por cada minuto desde el año 0.

Para acceder a este tipo de matriz, para el último ejemplo, buscando el valor de 2016-01-31 a las 19:42, tendrá acceso a los `values[2016][0][30][19][42]` (porque los **índices de matriz comienza a 0** y no a 1 como días y meses)

Algunos ejemplos siguientes:

```
// Defining a 2d Array to represent a matrix like
// 1 2 3     So with 2 lines and 3 columns;
```

```
// 4 5 6
var multiDimArray := [2/*lines*/][3/*columns*/]int{ [3]int{1, 2, 3}, [3]int{4, 5, 6} }

// That can be simplified like this:
var simplified := [2][3]int{{1, 2, 3}, {4, 5, 6}}

// What does it looks like ?
fmt.Println(multiDimArray)
// > [[1 2 3] [4 5 6]]

fmt.Println(multiDimArray[0])
// > [1 2 3]    (first line of the array)

fmt.Println(multiDimArray[0][1])
// > 2          (cell of line 0 (the first one), column 1 (the 2nd one))
```

```
// We can also define array with as much dimensions as we need
// here, initialized with all zeros
var multiDimArray := [2][4][3][2]string{}

fmt.Println(multiDimArray);
// Yeah, many dimensions stores many data
// > [[[["" ""] ["" ""]] [["" ""] ["" ""]] [["" ""] ["" ""]]]
//    [[["" ""] ["" ""]] [["" ""] ["" ""]] [["" ""] ["" ""]]]
//    [[["" ""] ["" ""]] [["" ""] ["" ""]] [["" ""] ["" ""]]]
//    [[["" ""] ["" ""]] [["" ""] ["" ""]] [["" ""] ["" ""]]]]
//   [[[["" ""] ["" ""]] [["" ""] ["" ""]] [["" ""] ["" ""]]]
//    [[["" ""] ["" ""]] [["" ""] ["" ""]] [["" ""] ["" ""]]]
//    [[["" ""] ["" ""]] [["" ""] ["" ""]] [["" ""] ["" ""]]]
//    [[["" ""] ["" ""]] [["" ""] ["" ""]] [["" ""] ["" ""]]]]
```

```
// We can set some values in the array's cells
multiDimArray[0][0][0][0] := "All zero indexes"  // Setting the first value
multiDimArray[1][3][2][1] := "All indexes to max"  // Setting the value at extreme location

fmt.Println(multiDimArray);
// If we could see in 4 dimensions, maybe we could see the result as a simple format

// > [[[["All zero indexes" ""] ["" ""]] [["" ""] ["" ""]] [["" ""] ["" ""]]]
//    [[["" ""] ["" ""]] [["" ""] ["" ""]] [["" ""] ["" ""]]]
//    [[["" ""] ["" ""]] [["" ""] ["" ""]] [["" ""] ["" ""]]]
//    [[["" ""] ["" ""]] [["" ""] ["" ""]] [["" ""] ["" ""]]]]
//   [[[["" ""] ["" ""]] [["" ""] ["" ""]] [["" ""] ["" ""]]]
//    [[["" ""] ["" ""]] [["" ""] ["" ""]] [["" ""] ["" ""]]]
//    [[["" ""] ["" ""]] [["" ""] ["" ""]] [["" ""] ["" ""]]]
//    [[["" ""] ["" ""]] [["" ""] ["" ""]] [["" ""] ["" "All indexes to max"]]]]
```

Índices de matriz

Se debe acceder a los valores de las matrices utilizando un número que especifique la ubicación del valor deseado en la matriz. Este número se llama índice.

Los índices comienzan en **0** y terminan en **longitud de matriz -1** .

Para acceder a un valor, debe hacer algo como esto: `arrayName[index]` , reemplazando "índice" por el número correspondiente al rango del valor en su matriz.

Por ejemplo:

```
var array = [6]int {1, 2, 3, 4, 5, 6}

fmt.Println(array[-42]) // invalid array index -1 (index must be non-negative)
fmt.Println(array[-1]) // invalid array index -1 (index must be non-negative)
fmt.Println(array[0]) // > 1
fmt.Println(array[1]) // > 2
fmt.Println(array[2]) // > 3
fmt.Println(array[3]) // > 4
fmt.Println(array[4]) // > 5
fmt.Println(array[5]) // > 6
fmt.Println(array[6]) // invalid array index 6 (out of bounds for 6-element array)
fmt.Println(array[42]) // invalid array index 42 (out of bounds for 6-element array)
```

Para establecer o modificar un valor en la matriz, la forma es la misma.
Ejemplo:

```
var array = [6]int {1, 2, 3, 4, 5, 6}

fmt.Println(array) // > [1 2 3 4 5 6]

array[0] := 6
fmt.Println(array) // > [6 2 3 4 5 6]

array[1] := 5
fmt.Println(array) // > [6 5 3 4 5 6]

array[2] := 4
fmt.Println(array) // > [6 5 4 4 5 6]

array[3] := 3
fmt.Println(array) // > [6 5 4 3 5 6]

array[4] := 2
fmt.Println(array) // > [6 5 4 3 2 6]

array[5] := 1
fmt.Println(array) // > [6 5 4 3 2 1]
```

Lea Arrays en línea: https://riptutorial.com/es/go/topic/390/arrays

Capítulo 8: Autorización JWT en Go

Introducción

Los tokens web de JSON (JWT) son un método popular para representar reclamos de forma segura entre dos partes. Comprender cómo trabajar con ellos es importante al desarrollar aplicaciones web o interfaces de programación de aplicaciones.

Observaciones

context.Context y HTTP middleware están fuera del alcance de este tema, pero no obstante, esas almas curiosas y errantes deberían consultar https://github.com/goware/jwtauth , https://github.com/auth0/go-jwt- middleware , y https://github.com/dgrijalva/jwt-go .

Grandes felicitaciones a Dave Grijalva por su increíble trabajo en go-jwt.

Examples

Analizar y validar un token utilizando el método de firma HMAC

```
// sample token string taken from the New example
tokenString :=
"eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJmb28iOiJiYXIiLCJuYmYiOjE0NDQ0Nzg0MDB9.u1riaD1rW97opCoAuRCTy4w5
Zk-bh7vLiRIsrpU"

// Parse takes the token string and a function for looking up the key. The latter is
especially
// useful if you use multiple keys for your application.  The standard is to use 'kid' in the
// head of the token to identify which key to use, but the parsed token (head and claims) is
provided
// to the callback, providing flexibility.
token, err := jwt.Parse(tokenString, func(token *jwt.Token) (interface{}, error) {
    // Don't forget to validate the alg is what you expect:
    if _, ok := token.Method.(*jwt.SigningMethodHMAC); !ok {
        return nil, fmt.Errorf("Unexpected signing method: %v", token.Header["alg"])
    }

    // hmacSampleSecret is a []byte containing your secret, e.g. []byte("my_secret_key")
    return hmacSampleSecret, nil
})

if claims, ok := token.Claims.(jwt.MapClaims); ok && token.Valid {
    fmt.Println(claims["foo"], claims["nbf"])
} else {
    fmt.Println(err)
}
```

Salida:

```
bar 1.4444784e+09
```

(De la documentación , cortesía de Dave Grijalva.)

Creación de un token utilizando un tipo de notificaciones personalizado

El `StandardClaim` está integrado en el tipo personalizado para permitir la fácil codificación, análisis y validación de las reclamaciones estándar.

```
tokenString :=
"eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJmb28iOiJiYXIiLCJleHAiOjE1MDAwLCJpc3MiOiJ0ZXN0In0.HE7fK0xOQwFE

type MyCustomClaims struct {
    Foo string `json:"foo"`
    jwt.StandardClaims
}

// sample token is expired.  override time so it parses as valid
at(time.Unix(0, 0), func() {
    token, err := jwt.ParseWithClaims(tokenString, &MyCustomClaims{}, func(token *jwt.Token)
(interface{}, error) {
        return []byte("AllYourBase"), nil
    })

    if claims, ok := token.Claims.(*MyCustomClaims); ok && token.Valid {
        fmt.Printf("%v %v", claims.Foo, claims.StandardClaims.ExpiresAt)
    } else {
        fmt.Println(err)
    }
})
```

Salida:

```
bar 15000
```

(De la documentación , cortesía de Dave Grijalva.)

Creación, firma y codificación de un token JWT utilizando el método de firma HMAC

```
// Create a new token object, specifying signing method and the claims
// you would like it to contain.
token := jwt.NewWithClaims(jwt.SigningMethodHS256, jwt.MapClaims{
    "foo": "bar",
    "nbf": time.Date(2015, 10, 10, 12, 0, 0, 0, time.UTC).Unix(),
})

// Sign and get the complete encoded token as a string using the secret
tokenString, err := token.SignedString(hmacSampleSecret)

fmt.Println(tokenString, err)
```

Salida:

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJmb28iOiJiYXIiLCJuYmYiOjE0NDQ0Nzg0MDB9.u1riaD1rW97opCoAuRCTy4w58
```

```
Zk-bh7vLiRIsrpU <nil>
```

(De la documentación , cortesía de Dave Grijalva.)

Usando el tipo StandardClaims por sí mismo para analizar un token

El tipo `StandardClaims` está diseñado para ser integrado en sus tipos personalizados para proporcionar características de validación estándar. Puede usarlo solo, pero no hay forma de recuperar otros campos después de analizar. Vea el ejemplo de reclamos personalizados para el uso previsto.

```
mySigningKey := []byte("AllYourBase")

// Create the Claims
claims := &jwt.StandardClaims{
    ExpiresAt: 15000,
    Issuer:    "test",
}

token := jwt.NewWithClaims(jwt.SigningMethodHS256, claims)
ss, err := token.SignedString(mySigningKey)
fmt.Printf("%v %v", ss, err)
```

Salida:

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJleHAiOjE1MDAwLCJpc3MiOiJ0ZXN0In0.QsODzZu3lUZMVdhbO76u3Jv02iYCvE
<nil>
```

(De la documentación , cortesía de Dave Grijalva.)

Analizar los tipos de error usando cheques de campo de bits

```
// Token from another example.  This token is expired
var tokenString =
"eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJmb28iOiJiYXIiLCJleHAiOjE1MDAwLCJpc3MiOiJ0ZXN0In0.HE7fK0xOQwFEr


token, err := jwt.Parse(tokenString, func(token *jwt.Token) (interface{}, error) {
    return []byte("AllYourBase"), nil
})

if token.Valid {
    fmt.Println("You look nice today")
} else if ve, ok := err.(*jwt.ValidationError); ok {
    if ve.Errors&jwt.ValidationErrorMalformed != 0 {
        fmt.Println("That's not even a token")
    } else if ve.Errors&(jwt.ValidationErrorExpired|jwt.ValidationErrorNotValidYet) != 0 {
        // Token is either expired or not active yet
        fmt.Println("Timing is everything")
    } else {
        fmt.Println("Couldn't handle this token:", err)
    }
} else {
    fmt.Println("Couldn't handle this token:", err)
```

```
}
```

Salida:

```
Timing is everything
```

(De la documentación , cortesía de Dave Grijalva.)

Obteniendo el token del encabezado de autorización HTTP

```
type contextKey string

const (
    // JWTTokenContextKey holds the key used to store a JWT Token in the
    // context.
    JWTTokenContextKey contextKey = "JWTToken"

    // JWTClaimsContextKey holds the key used to store the JWT Claims in the
    // context.
    JWTClaimsContextKey contextKey = "JWTClaims"
)

// ToHTTPContext moves JWT token from request header to context.
func ToHTTPContext() http.RequestFunc {
    return func(ctx context.Context, r *stdhttp.Request) context.Context {
        token, ok := extractTokenFromAuthHeader(r.Header.Get("Authorization"))
        if !ok {
            return ctx
        }

        return context.WithValue(ctx, JWTTokenContextKey, token)
    }
}
```

(De go-kit / kit , cortesía de Peter Bourgon)

Lea Autorización JWT en Go en línea: https://riptutorial.com/es/go/topic/10161/autorizacion-jwt-en-go

Capítulo 9: Bucles

Introducción

Como una de las funciones más básicas de la programación, los bucles son una pieza importante para casi todos los lenguajes de programación. Los bucles permiten a los desarrolladores configurar ciertas partes de su código para que se repitan a través de una serie de bucles que se conocen como iteraciones. Este tema cubre el uso de varios tipos de bucles y aplicaciones de bucles en Go.

Examples

Lazo basico

`for` es la única sentencia de bucle en marcha, por lo que una implementación básica de bucle podría tener este aspecto:

```
// like if, for doesn't use parens either.
// variables declared in for and if are local to their scope.
for x := 0; x < 3; x++ { // ++ is a statement.
    fmt.Println("iteration", x)
}

// would print:
// iteration 0
// iteration 1
// iteration 2
```

Romper y continuar

Salir del bucle y continuar a la siguiente iteración también se admite en Go, como en muchos otros idiomas:

```
for x := 0; x < 10; x++ { // loop through 0 to 9
    if x < 3 { // skips all the numbers before 3
        continue
    }
    if x > 5 { // breaks out of the loop once x == 6
        break
    }
    fmt.Println("iteration", x)
}

// would print:
// iteration 3
// iteration 4
// iteration 5
```

Las declaraciones `break` y `continue` además aceptan una etiqueta opcional, que se utiliza para

identificar los bucles externos para apuntar con la declaración:

```
OuterLoop:
for {
    for {
        if allDone() {
            break OuterLoop
        }
        if innerDone() {
            continue OuterLoop
        }
        // do something
    }
}
```

Bucle condicional

La palabra clave `for` también se usa para los bucles condicionales, tradicionalmente `while` bucles en otros lenguajes de programación.

```
package main

import (
    "fmt"
)

func main() {
    i := 0
    for i < 3 { // Will repeat if condition is true
        i++
        fmt.Println(i)
    }
}
```

jugar en el patio de recreo

Saldrá:

```
1
2
3
```

Bucle infinito:

```
for {
    // This will run until a return or break.
}
```

Diferentes formas de For Loop

Forma simple utilizando una variable:

```
for i := 0; i < 10; i++ {
```

```
    fmt.Print(i, " ")
}
```

Usando dos variables (o más):

```
for i, j := 0, 0; i < 5 && j < 10; i, j = i+1, j+2 {
    fmt.Println(i, j)
}
```

Sin usar declaración de inicialización:

```
i := 0
for ; i < 10; i++ {
    fmt.Print(i, " ")
}
```

Sin una expresión de prueba:

```
for i := 1; ; i++ {
    if i&1 == 1 {
        continue
    }
    if i == 22 {
        break
    }
    fmt.Print(i, " ")
}
```

Sin incremento de expresión:

```
for i := 0; i < 10; {
    fmt.Print(i, " ")
    i++
}
```

Cuando se eliminan las tres expresiones de inicialización, prueba e incremento, el bucle se vuelve infinito:

```
i := 0
for {
    fmt.Print(i, " ")
    i++
    if i == 10 {
        break
    }
}
```

Este es un ejemplo de bucle infinito con contador inicializado con cero:

```
for i := 0; ; {
    fmt.Print(i, " ")
    if i == 9 {
        break
```

```
    }
    i++
}
```

Cuando solo se usa la expresión de prueba (actúa como un bucle while típico):

```
i := 0
for i < 10 {
    fmt.Print(i, " ")
    i++
}
```

Usando sólo la expresión de incremento:

```
i := 0
for ; ; i++ {
    fmt.Print(i, " ")
    if i == 9 {
        break
    }
}
```

Iterar sobre un rango de valores usando índice y valor:

```
ary := [5]int{0, 1, 2, 3, 4}
for index, value := range ary {
    fmt.Println("ary[", index, "] =", value)
}
```

Iterar sobre un rango usando solo índice:

```
for index := range ary {
    fmt.Println("ary[", index, "] =", ary[index])
}
```

Iterar sobre un rango usando solo índice:

```
for index, _ := range ary {
    fmt.Println("ary[", index, "] =", ary[index])
}
```

Iterar sobre un rango usando solo valor:

```
for _, value := range ary {
    fmt.Print(value, " ")
}
```

Iterar en un rango usando la clave y el valor para el mapa (puede que no esté en orden):

```
mp := map[string]int{"One": 1, "Two": 2, "Three": 3}
for key, value := range mp {
    fmt.Println("map[", key, "] =", value)
```

```
}
```

Iterar sobre un rango usando solo la tecla para el mapa (puede que no esté en orden):

```
for key := range mp {
    fmt.Print(key, " ") //One Two Three
}
```

Iterar sobre un rango usando solo la tecla para el mapa (puede que no esté en orden):

```
for key, _ := range mp {
    fmt.Print(key, " ") //One Two Three
}
```

Iterar en un rango usando solo el valor del mapa (puede que no esté en orden):

```
for _, value := range mp {
    fmt.Print(value, " ") //2 3 1
}
```

Iterar en un rango para canales (sale si el canal está cerrado):

```
ch := make(chan int, 10)
for i := 0; i < 10; i++ {
    ch <- i
}
close(ch)

for i := range ch {
    fmt.Print(i, " ")
}
```

Iterar sobre un rango para la cadena (da puntos de código Unicode):

```
utf8str := "B = \u00b5H" //B = µH
for _, r := range utf8str {
    fmt.Print(r, " ") //66 32 61 32 181 72
}
fmt.Println()
for _, v := range []byte(utf8str) {
    fmt.Print(v, " ") //66 32 61 32 194 181 72
}
fmt.Println(len(utf8str)) //7
```

Como puede ver, `utf8str` tiene 6 runas (puntos de código Unicode) y 7 bytes.

Bucle temporizado

```
package main

import(
    "fmt"
```

```
    "time"
)

func main() {
    for _ = range time.Tick(time.Second * 3) {
        fmt.Println("Ticking every 3 seconds")
    }
}
```

Lea Bucles en línea: https://riptutorial.com/es/go/topic/975/bucles

Capítulo 10: Buenas prácticas en la estructura del proyecto.

Examples

Proyectos Restfull API con Gin

> Gin es un framework web escrito en golang. Cuenta con una API similar a la de martini con un rendimiento mucho mejor, hasta 40 veces más rápido. Si necesitas rendimiento y buena productividad, te encantará la ginebra.

Habrá 8 paquetes + main.go

1. controladores
2. núcleo
3. libs
4. middlewares
5. público
6. enrutadores
7. servicios
8. pruebas
9. main.go

---

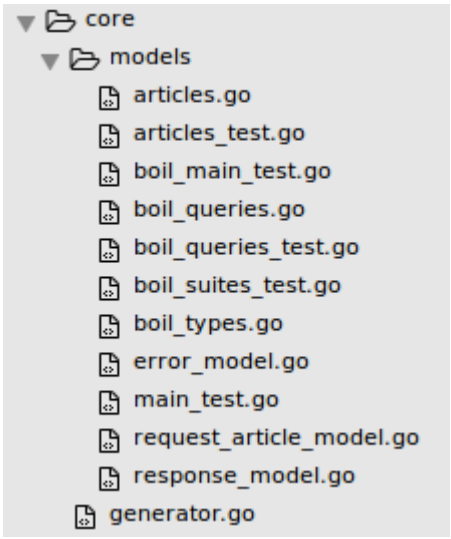
controladores

El paquete de controladores almacenará toda la lógica de la API. Cualquiera que sea tu API, tu lógica sucederá aquí

---

núcleo

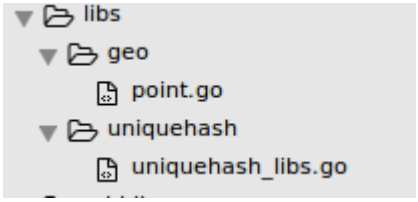
El paquete central almacenará todos sus modelos creados, ORM, etc.



---

libs

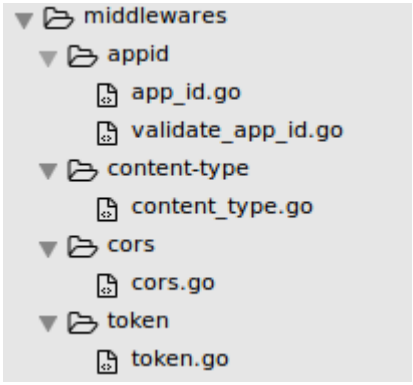
Este paquete almacenará cualquier biblioteca que se use en proyectos. Pero solo para la biblioteca creada / importada manualmente, que no está disponible cuando se usan los comandos `go get package_name` . Podría ser tu propio algoritmo hash, gráfico, árbol, etc.



---

middlewares

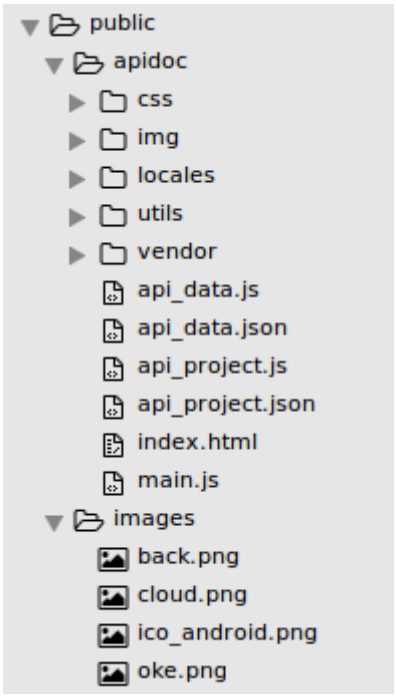
Este paquete almacena cada middleware que se usa en el proyecto, podría ser la creación / validación de cors, device-id, auth, etc.



---

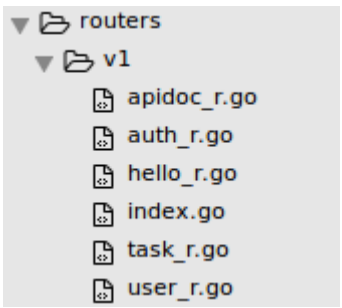
público

Este paquete almacenará todos los archivos públicos y estáticos, pueden ser html, css, javascript, imágenes, etc.



enrutadores

Este paquete almacenará todas las rutas en su API REST.



Ver código de ejemplo cómo asignar las rutas.

auth_r.go

```
import (
    auth "simple-api/controllers/v1/auth"
    "gopkg.in/gin-gonic/gin.v1"
)

func SetAuthRoutes(router *gin.RouterGroup) {

/**
 * @api {post} /v1/auth/login Login
 * @apiGroup Users
 * @apiHeader {application/json} Content-Type Accept application/json
 * @apiParam {String} username User username
 * @apiParam {String} password User Password
 * @apiParamExample {json} Input
 *    {
 *      "username": "your username",
 *        "password"     : "your password"
 *    }
 * @apiSuccess {Object} authenticate Response
 * @apiSuccess {Boolean} authenticate.success Status
 * @apiSuccess {Integer} authenticate.statuscode Status Code
 * @apiSuccess {String} authenticate.message Authenticate Message
 * @apiSuccess {String} authenticate.token Your JSON Token
 * @apiSuccessExample {json} Success
 *    {
 *        "authenticate": {
 *               "statuscode": 200,
 *              "success": true,
 *           "message": "Login Successfully",
 *              "token":
"eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiYWRtaW4iOnRyc

 *            }
 *      }
 * @apiErrorExample {json} List error
 *    HTTP/1.1 500 Internal Server Error
 */

    router.POST("/auth/login" , auth.Login)
}
```

Si ve, la razón por la que separo el controlador es para que podamos administrar cada uno de los enrutadores. Entonces puedo crear comentarios sobre la API, que con apidoc generará esto en documentación estructurada. Luego llamaré a la función en index.go en el paquete actual

index.go

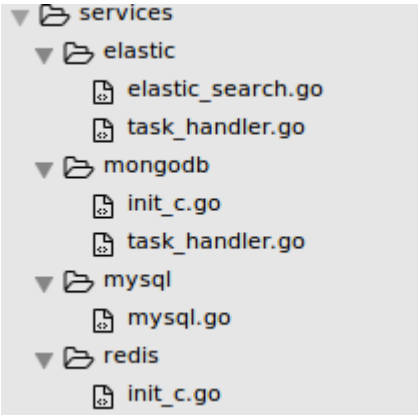
```
package v1


import (
    "gopkg.in/gin-gonic/gin.v1"
    token "simple-api/middlewares/token"
    appid "simple-api/middlewares/appid"
)
func InitRoutes(g *gin.RouterGroup)  {
```

```
    g.Use(appid.AppIDMiddleWare())
    SetHelloRoutes(g)
    SetAuthRoutes(g) // SetAuthRoutes invoked
    g.Use(token.TokenAuthMiddleWare())  //secure the API From this line to bottom with JSON
Auth
    g.Use(appid.ValidateAppIDMiddleWare())
    SetTaskRoutes(g)
    SetUserRoutes(g)
}
```

---

servicios

Este paquete almacenará cualquier configuración y configuración para usar en el proyecto de cualquier servicio usado, podría ser mongodb, redis, mysql, elasticsearch, etc.



---

main.go

La entrada principal de la API. Aquí se configurará cualquier configuración acerca de la configuración del entorno de desarrollo, sistemas, puertos, etc.

Ejemplo:
main.go

```
package main
import (
    "fmt"
    "net/http"
    "gopkg.in/gin-gonic/gin.v1"
    "articles/services/mysql"
    "articles/routers/v1"
    "articles/core/models"
)

var router  *gin.Engine;

func init() {
    mysql.CheckDB()
    router = gin.New();
    router.NoRoute(noRouteHandler())
    version1:=router.Group("/v1")
```

```
    v1.InitRoutes(version1)

}

func main() {
    fmt.Println("Server Running on Port: ", 9090)
    http.ListenAndServe(":9090",router)
}

func noRouteHandler() gin.HandlerFunc{
    return  func(c *gin.Context) {
    var statuscode     int
    var message        string         = "Not Found"
    var data         interface{} = nil
    var listError [] models.ErrorModel = nil
    var endpoint    string = c.Request.URL.String()
    var method        string = c.Request.Method

    var tempEr models.ErrorModel
    tempEr.ErrorCode     = 4041
    tempEr.Hints         = "Not Found !! \n Routes In Valid. You enter on invalid
Page/Endpoint"
    tempEr.Info            = "visit http://localhost:9090/v1/docs to see the available routes"
    listError             = append(listError,tempEr)
    statuscode             = 404
    responseModel := &models.ResponseModel{
        statuscode,
        message,
        data,
        listError,
        endpoint,
        method,
    }
    var content gin.H = responseModel.NewResponse();
    c.JSON(statuscode,content)
    }
}
```

ps: Cada código en este ejemplo, provienen de diferentes proyectos

---

ver proyectos de muestra en github

Lea Buenas prácticas en la estructura del proyecto. en línea: https://riptutorial.com/es/go/topic/9463/buenas-practicas-en-la-estructura-del-proyecto-

Capítulo 11: cgo

Examples

Cgo: tutorial de primeros pasos.

Algunos ejemplos para entender el flujo de trabajo de usar los enlaces de Go C

Qué

En Go puedes llamar a programas y funciones C usando cgo . De esta manera, puede crear fácilmente enlaces C a otras aplicaciones o bibliotecas que proporcionen C API.

Cómo

Todo lo que necesita hacer es agregar una `import "C"` al comienzo de su programa Go **justo** después de incluir su programa C:

```
//#include <stdio.h>
import "C"
```

Con el ejemplo anterior puedes usar el paquete `stdio` en Go.

Si necesita usar una aplicación que esté en su misma carpeta, use la misma sintaxis que en C (con la `"` lugar de `<>`)

```
//#include "hello.c"
import "C"
```

> **IMPORTANTE** : No **deje una nueva línea entre las** declaraciones de **`include`** **y de** **`import "C"`** o obtendrá este tipo de errores en la compilación:

```
# command-line-arguments
could not determine kind of name for C.Hello
could not determine kind of name for C.sum
```

El ejemplo

En esta carpeta puedes encontrar un ejemplo de enlaces en C. Tenemos dos "bibliotecas" de C muy simples llamadas `hello.c` :

```
//hello.c
#include <stdio.h>

void Hello(){
```

```
    printf("Hello world\n");
}
```

Eso simplemente imprime "hola mundo" en la consola y `sum.c`

```
//sum.c
#include <stdio.h>

int sum(int a, int b) {
    return a + b;
}
```

... eso toma 2 argumentos y devuelve su suma (no lo imprima).

Tenemos un programa `main.go` que hará uso de estos dos archivos. Primero los importamos como mencionamos antes:

```
//main.go
package main

/*
  #include "hello.c"
  #include "sum.c"
*/
import "C"
```

Hola Mundo!

Ahora estamos listos para usar los programas de C en nuestra aplicación Go. Primero probemos el programa Hello:

```
//main.go
package main

/*
  #include "hello.c"
  #include "sum.c"
*/
import "C"


func main() {
    //Call to void function without params
    err := Hello()
    if err != nil {
        log.Fatal(err)
    }
}

//Hello is a C binding to the Hello World "C" program. As a Go user you could
//use now the Hello function transparently without knowing that it is calling
//a C function
func Hello() error {
    _, err := C.Hello()    //We ignore first result as it is a void function
    if err != nil {
```

```
        return errors.New("error calling Hello function: " + err.Error())
    }

    return nil
}
```

Ahora ejecute el programa main.go utilizando `go run main.go` para obtener la impresión del programa C: "¡Hola mundo!". ¡Bien hecho!

Suma de ints

Hagámoslo un poco más complejo agregando una función que sume sus dos argumentos.

```
//sum.c
#include <stdio.h>

int sum(int a, int b) {
  return a + b;
}
```

Y lo llamaremos desde nuestra aplicación Go anterior.

```
//main.go
package main

/*
#include "hello.c"
#include "sum.c"
*/
import "C"

import (
    "errors"
    "fmt"
    "log"
)

func main() {
    //Call to void function without params
    err := Hello()
    if err != nil {
        log.Fatal(err)
    }

    //Call to int function with two params
    res, err := makeSum(5, 4)
    if err != nil {
        log.Fatal(err)
    }

    fmt.Printf("Sum of 5 + 4 is %d\n", res)
}

//Hello is a C binding to the Hello World "C" program. As a Go user you could
//use now the Hello function transparently without knowing that is calling a C
//function
func Hello() error {
```

```
    _, err := C.Hello() //We ignore first result as it is a void function
    if err != nil {
        return errors.New("error calling Hello function: " + err.Error())
    }

    return nil
}

//makeSum also is a C binding to make a sum. As before it returns a result and
//an error. Look that we had to pass the Int values to C.int values before using
//the function and cast the result back to a Go int value
func makeSum(a, b int) (int, error) {
    //Convert Go ints to C ints
    aC := C.int(a)
    bC := C.int(b)

    sum, err := C.sum(aC, bC)
    if err != nil {
        return 0, errors.New("error calling Sum function: " + err.Error())
    }

    //Convert C.int result to Go int
    res := int(sum)

    return res, nil
}
```

Echa un vistazo a la función "makeSum". Recibe dos parámetros `int` que deben convertirse a C `int` antes utilizando la función `C.int` . Además, la devolución de la llamada nos dará una C `int` y un error en caso de que algo salga mal. Necesitamos lanzar una respuesta en C a un int de Go usando `int()` .

Intenta ejecutar nuestra aplicación `go run main.go` usando `go run main.go`

```
$ go run main.go
Hello world!
Sum of 5 + 4 is 9
```

Generando un binario

Si intentas una compilación de Go, podrías obtener múltiples errores de definición.

```
$ go build
# github.com/sayden/c-bindings
/tmp/go-build329491076/github.com/sayden/c-bindings/_obj/hello.o: In function `Hello':
../../go/src/github.com/sayden/c-bindings/hello.c:5: multiple definition of `Hello'
/tmp/go-build329491076/github.com/sayden/c-
bindings/_obj/main.cgo2.o:/home/mariocaster/go/src/github.com/sayden/c-bindings/hello.c:5:
first defined here
/tmp/go-build329491076/github.com/sayden/c-bindings/_obj/sum.o: In function `sum':
../../go/src/github.com/sayden/c-bindings/sum.c:5: multiple definition of `sum`
/tmp/go-build329491076/github.com/sayden/c-
bindings/_obj/main.cgo2.o:/home/mariocaster/go/src/github.com/sayden/c-bindings/sum.c:5: first
defined here
collect2: error: ld returned 1 exit status
```

El truco consiste en referirse directamente al archivo principal cuando se utiliza `go build` :

```
$ go build main.go
$ ./main
Hello world!
Sum of 5 + 4 is 9
```

> Recuerde que puede proporcionar un nombre al archivo binario utilizando `-o` flag `go build -o my_c_binding main.go`

Espero que disfrutes este tutorial.

Lea cgo en línea: https://riptutorial.com/es/go/topic/6125/cgo

Capítulo 12: cgo

Examples

Llamando a la función C desde Go

Cgo permite la creación de paquetes Go que llaman código C
Para usar `cgo` escriba el código Go normal que importa un pseudo-paquete "C". El código Go puede hacer referencia a tipos como `C.int` o funciones como `C.Add` .
La importación de "C" está precedida inmediatamente por un comentario, ese comentario, denominado preámbulo, se usa como encabezado al compilar las partes C del paquete.
Tenga en cuenta que no debe haber líneas en blanco entre el comentario de `cgo` y la declaración de importación.
Tenga en cuenta que la `import "C"` no se puede agrupar con otras importaciones en una declaración de importación "factorizada" entre paréntesis. Debe escribir varias declaraciones de importación, como:

```
import "C"
import "fmt"
```

Y es un buen estilo usar la declaración de importación factorizada, para otras importaciones, como:

```
import "C"
import (
    "fmt"
    "math"
)
```

Ejemplo simple usando `cgo` :

```
package main

//int Add(int a, int b){
//    return a+b;
//}
import "C"
import "fmt"

func main() {
    a := C.int(10)
    b := C.int(20)
    c := C.Add(a, b)
    fmt.Println(c) // 30
}
```

Entonces `go build` , y ejecútalo, salida:

```
30
```

Para `cgo` paquetes `cgo` , simplemente use `go build` o `go install` como de costumbre. La `go tool` reconoce la importación especial en `"C"` y utiliza automáticamente `cgo` para esos archivos.

Cable C y Go en todas las direcciones.

Llamando al código C desde Go

```
package main

/*
// Everything in comments above the import "C" is C code and will be compiles with the GCC.
// Make sure you have a GCC installed.

int addInC(int a, int b) {
    return a + b;
}
 */
import "C"
import "fmt"

func main() {
       a := 3
       b := 5

       c := C.addInC(C.int(a), C.int(b))

       fmt.Println("Add in C:", a, "+", b, "=", int(c))
}
```

Llamando al código Go desde C

```
package main

/*
static inline int multiplyInGo(int a, int b) {
    return go_multiply(a, b);
}
 */
import "C"
import (
       "fmt"
)

func main() {
       a := 3
       b := 5

       c := C.multiplyInGo(C.int(a), C.int(b))

       fmt.Println("multiplyInGo:", a, "*", b, "=", int(c))
}

//export go_multiply
func go_multiply(a C.int, b C.int) C.int {
       return a * b
}
```

Tratar con punteros de función

```
package main

/*
int go_multiply(int a, int b);

typedef int (*multiply_f)(int a, int b);
multiply_f multiply;

static inline init() {
    multiply = go_multiply;
}

static inline int multiplyWithFp(int a, int b) {
    return multiply(a, b);
}
 */
import "C"
import (
       "fmt"
)

func main() {
       a := 3
       b := 5
       C.init(); // OR:
       C.multiply = C.multiply_f(go_multiply);

       c := C.multiplyWithFp(C.int(a), C.int(b))

       fmt.Println("multiplyInGo:", a, "+", b, "=", int(c))
}

//export go_multiply
func go_multiply(a C.int, b C.int) C.int {
       return a * b
}
```

Convertir tipos, estructuras de acceso y aritmética de punteros

De la documentación oficial de Go:

```
// Go string to C string
// The C string is allocated in the C heap using malloc.
// It is the caller's responsibility to arrange for it to be
// freed, such as by calling C.free (be sure to include stdlib.h
// if C.free is needed).
func C.CString(string) *C.char

// Go []byte slice to C array
// The C array is allocated in the C heap using malloc.
// It is the caller's responsibility to arrange for it to be
// freed, such as by calling C.free (be sure to include stdlib.h
// if C.free is needed).
func C.CBytes([]byte) unsafe.Pointer

// C string to Go string
func C.GoString(*C.char) string
```

```
// C data with explicit length to Go string
func C.GoStringN(*C.char, C.int) string

// C data with explicit length to Go []byte
func C.GoBytes(unsafe.Pointer, C.int) []byte
```

Cómo usarlo:

```
func go_handleData(data *C.uint8_t, length C.uint8_t) []byte {
       return C.GoBytes(unsafe.Pointer(data), C.int(length))
}

// ...

goByteSlice := []byte {1, 2, 3}
goUnsafePointer := C.CBytes(goByteSlice)
cPointer := (*C.uint8_t)(goUnsafePointer)

// ...

func getPayload(packet *C.packet_t) []byte {
       dataPtr := unsafe.Pointer(packet.data)
       // Lets assume a 2 byte header before the payload.
       payload := C.GoBytes(unsafe.Pointer(uintptr(dataPtr)+2), C.int(packet.dataLength-2))
       return payload
}
```

Lea cgo en línea: https://riptutorial.com/es/go/topic/6455/cgo

Capítulo 13: Cierres

Examples

Fundamentos de cierre

Un *cierre* es una función tomada junto con un entorno. La función es típicamente una función anónima definida dentro de otra función. El entorno es el alcance léxico de la función de encierro (una idea muy básica de un alcance léxico de una función sería el alcance que existe entre las llaves de la función).

```
func g() {
    i := 0
    f := func() { // anonymous function
        fmt.Println("f called")
    }
}
```

Dentro del cuerpo de una función anónima (por ejemplo, `f`) definida dentro de otra función (por ejemplo, `g`), las variables presentes en los ámbitos tanto de `f` como de `g` son accesibles. Sin embargo, es el alcance de `g` que forma parte del entorno del cierre (la parte de la función es `f`) y, como resultado, los cambios realizados en las variables en el alcance de `g` conservan sus valores (es decir, el entorno persiste entre las llamadas a `f`) .

Considere la siguiente función:

```
func NaturalNumbers() func() int {
    i := 0
    f:= func() int { // f is the function part of closure
        i++
        return i
    }
    return f
}
```

En la definición anterior, `NaturalNumbers` tiene una función interna `f` que devuelve `NaturalNumbers` . Dentro de `f` , se está accediendo a la variable `i` definida dentro del alcance de `NaturalNumbers` .

Obtenemos una nueva función de `NaturalNumbers` así:

```
n := NaturalNumbers()
```

Ahora `n` es un cierre. Es una función (definida por `f`) que también tiene un entorno asociado (alcance de `NaturalNumbers`).

En el caso de `n` , la parte del entorno solo contiene una variable: `i`

Como `n` es una función, puede ser llamada:

```
fmt.Println(n()) // 1
fmt.Println(n()) // 2
fmt.Println(n()) // 3
```

Como se desprende de la salida anterior, cada vez que se llama `n` , se incrementa `i` . `i` comienza en 0, y cada llamada a `n` ejecuta `i++` .

El valor de `i` se conserva entre las llamadas. Es decir, el medio ambiente, siendo parte del cierre, persiste.

Llamar de nuevo a `NaturalNumbers` crearía y devolvería una nueva función. Esto iniciaría una nueva `i` dentro de `NaturalNumbers` . Lo que significa que la función recién devuelta forma otro cierre que tiene la misma parte para la función (aún `f`) pero un entorno completamente nuevo (una `i` recién iniciada).

```
o := NaturalNumbers()

fmt.Println(n()) // 4
fmt.Println(o()) // 1
fmt.Println(o()) // 2
fmt.Println(n()) // 5
```

Ambos `n` y `o` son cierres que contienen la misma parte de función (lo que les da el mismo comportamiento), pero diferentes entornos. Por lo tanto, el uso de cierres permite que las funciones tengan acceso a un entorno persistente que se puede usar para retener información entre llamadas.

Otro ejemplo:

```
func multiples(i int) func() int {
    var x int = 0
    return func() int {
        x++
        // paramenter to multiples (here it is i) also forms
        // a part of the environment, and is retained
        return x * i
    }
}

two := multiples(2)
fmt.Println(two(), two(), two()) // 2 4 6

fortyTwo := multiples(42)
fmt.Println(fortyTwo(), fortyTwo(), fortyTwo()) // 42 84 126
```

Lea Cierres en línea: https://riptutorial.com/es/go/topic/2741/cierres

Capítulo 14: Cliente HTTP

Sintaxis

- resp, err: = http.Get (url) // Realiza una solicitud GET de HTTP con el cliente HTTP predeterminado. Se devuelve un error no nulo si la solicitud falla.
- resp, err: = http.Post (url, bodyType, body) // Realiza una solicitud HTTP POST con el cliente HTTP predeterminado. Se devuelve un error no nulo si la solicitud falla.
- resp, err: = http.PostForm (url, valores) // Realiza una solicitud POST del formulario HTTP con el cliente HTTP predeterminado. Se devuelve un error no nulo si la solicitud falla.

Parámetros

Parámetro	Detalles
resp	Una respuesta de tipo `*http.Response` a una solicitud HTTP
errar	Un `error` Si no es nulo, representa un error que ocurrió cuando se llamó a la función.
url	Una URL de tipo `string` para realizar una solicitud HTTP.
tipo de cuerpo	El tipo MIME de tipo `string` de la carga útil del cuerpo de una solicitud POST.
cuerpo	Un `io.Reader` (implementa `Read()`) que se leerá hasta que se alcance un error para enviarlo como la carga útil del cuerpo de una solicitud POST.
valores	Un mapa clave-valor de tipo `url.Values` . El tipo subyacente es una `map[string][]string` .

Observaciones

Es importante `defer resp.Body.Close()` después de cada solicitud HTTP que no devuelva un error no nulo, de lo contrario, se perderán recursos.

Examples

GET básico

Realice una solicitud GET básica e imprima el contenido de un sitio (HTML).

```
package main
```

```
import (
    "fmt"
    "io/ioutil"
    "net/http"
)

func main() {
    resp, err := http.Get("https://example.com/")
    if err != nil {
        panic(err)
    }

    // It is important to defer resp.Body.Close(), else resource leaks will occur.
    defer resp.Body.Close()

    data, err := ioutil.ReadAll(resp.Body)
    if err != nil {
        panic(err)
    }

    // Will print site contents (HTML) to output
    fmt.Println(string(data))
}
```

GET con parámetros de URL y una respuesta JSON

Una solicitud de las 10 publicaciones más recientes de StackOverflow activas que utilizan la API de Exchange Stack.

```
package main

import (
    "encoding/json"
    "fmt"
    "net/http"
    "net/url"
)

const apiURL = "https://api.stackexchange.com/2.2/posts?"

// Structs for JSON decoding
type postItem struct {
    Score int    `json:"score"`
    Link  string `json:"link"`
}

type postsType struct {
    Items []postItem `json:"items"`
}

func main() {
    // Set URL parameters on declaration
    values := url.Values{
        "order": []string{"desc"},
        "sort":  []string{"activity"},
        "site":  []string{"stackoverflow"},
    }

    // URL parameters can also be programmatically set
```

```
    values.Set("page", "1")
    values.Set("pagesize", "10")

    resp, err := http.Get(apiURL + values.Encode())
    if err != nil {
        panic(err)
    }

    defer resp.Body.Close()

    // To compare status codes, you should always use the status constants
    // provided by the http package.
    if resp.StatusCode != http.StatusOK {
        panic("Request was not OK: " + resp.Status)
    }

    // Example of JSON decoding on a reader.
    dec := json.NewDecoder(resp.Body)
    var p postsType
    err = dec.Decode(&p)
    if err != nil {
        panic(err)
    }

    fmt.Println("Top 10 most recently active StackOverflow posts:")
    fmt.Println("Score", "Link")
    for _, post := range p.Items {
        fmt.Println(post.Score, post.Link)
    }
}
```

Tiempo de espera de solicitud con un contexto

1.7+

El tiempo de espera de una solicitud HTTP con un contexto se puede lograr con solo la biblioteca estándar (no las subposiciones) en 1.7+:

```
import (
    "context"
    "net/http"
    "time"
)

req, err := http.NewRequest("GET", `https://example.net`, nil)
ctx, _ := context.WithTimeout(context.TODO(), 200 * time.Milliseconds)
resp, err := http.DefaultClient.Do(req.WithContext(ctx))
// Be sure to handle errors.
defer resp.Body.Close()
```

Antes de 1.7

```
import (
    "net/http"
    "time"
```

```
    "golang.org/x/net/context"
    "golang.org/x/net/context/ctxhttp"
)

ctx, err := context.WithTimeout(context.TODO(), 200 * time.Milliseconds)
resp, err := ctxhttp.Get(ctx, http.DefaultClient, "https://www.example.net")
// Be sure to handle errors.
defer resp.Body.Close()
```

Otras lecturas

Para obtener más información sobre el paquete de `context` , consulte Contexto .

PUT solicitud de objeto JSON

Lo siguiente actualiza un objeto de usuario a través de una solicitud PUT e imprime el código de estado de la solicitud:

```
package main

import (
    "bytes"
    "encoding/json"
    "fmt"
    "net/http"
)

type User struct {
    Name  string
    Email string
}

func main() {
    user := User{
        Name:  "John Doe",
        Email: "johndoe@example.com",
    }

    // initialize http client
    client := &http.Client{}

    // marshal User to json
    json, err := json.Marshal(user)
    if err != nil {
        panic(err)
    }

    // set the HTTP method, url, and request body
    req, err := http.NewRequest(http.MethodPut, "http://api.example.com/v1/user",
bytes.NewBuffer(json))
    if err != nil {
        panic(err)
    }

    // set the request header Content-Type for json
    req.Header.Set("Content-Type", "application/json; charset=utf-8")
```

```
    resp, err := client.Do(req)
    if err != nil {
        panic(err)
    }

    fmt.Println(resp.StatusCode)
}
```

Lea Cliente HTTP en línea: https://riptutorial.com/es/go/topic/1422/cliente-http

Capítulo 15: Codificación Base64

Sintaxis

- func (enc * base64.Encoding) Codificar (dst, src [] byte)
- func (enc * base64.Encoding) Decode (dst, src [] byte) (n int, error err)
- func (enc * base64.Encoding) EncodeToString (src [] byte) cadena
- func (enc * base64.Encoding) DecodeString (s string) ([] byte, error)

Observaciones

El paquete `encoding/base64` contiene varios codificadores integrados . La mayoría de los ejemplos en este documento usarán `base64.StdEncoding` , pero cualquier codificador (`URLEncoding` , `RawStdEncodign` , su propio codificador personalizado, etc.) puede ser sustituido.

Examples

Codificación

```
const foobar = `foo bar`
encoding := base64.StdEncoding
encodedFooBar := make([]byte, encoding.EncodedLen(len(foobar)))
encoding.Encode(encodedFooBar, []byte(foobar))
fmt.Printf("%s", encodedFooBar)
// Output: Zm9vIGJhcg==
```

Patio de recreo

Codificación a una cadena

```
str := base64.StdEncoding.EncodeToString([]byte(`foo bar`))
fmt.Println(str)
// Output: Zm9vIGJhcg==
```

Patio de recreo

Descodificación

```
encoding := base64.StdEncoding
data := []byte(`Zm9vIGJhcg==`)
decoded := make([]byte, encoding.DecodedLen(len(data)))
n, err := encoding.Decode(decoded, data)
if err != nil {
    log.Fatal(err)
}

// Because we don't know the length of the data that is encoded
```

```
// (only the max length), we need to trim the buffer to whatever
// the actual length of the decoded data was.
decoded = decoded[:n]

fmt.Printf("`%s`", decoded)
// Output: `foo bar`
```

Patio de recreo

Decodificar una cadena

```
decoded, err := base64.StdEncoding.DecodeString(`biws`)
if err != nil {
    log.Fatal(err)
}

fmt.Printf("%s", decoded)
// Output: n,,
```

Patio de recreo

Lea Codificación Base64 en línea: https://riptutorial.com/es/go/topic/4492/codificacion-base64

Capítulo 16: Comandos de ejecución

Examples

Tiempo de espera con interrupción y luego matar

```
c := exec.Command(name, arg...)
b := &bytes.Buffer{}
c.Stdout = b
c.Stdin = stdin
if err := c.Start(); err != nil {
    return nil, err
}
timedOut := false
intTimer := time.AfterFunc(timeout, func() {
    log.Printf("Process taking too long. Interrupting: %s %s", name, strings.Join(arg, " "))
    c.Process.Signal(os.Interrupt)
    timedOut = true
})
killTimer := time.AfterFunc(timeout*2, func() {
    log.Printf("Process taking too long. Killing: %s %s", name, strings.Join(arg, " "))
    c.Process.Signal(os.Kill)
    timedOut = true
})
err := c.Wait()
intTimer.Stop()
killTimer.Stop()
if timedOut {
    log.Print("the process timed out\n")
}
```

Ejecución de comando simple

```
// Execute a command a capture standard out. exec.Command creates the command
// and then the chained Output method gets standard out. Use CombinedOutput()
// if you want both standard out and standerr output
out, err := exec.Command("echo", "foo").Output()
if err != nil {
    log.Fatal(err)
}
```

Ejecutando un Comando luego Continuar y Esperar

```
cmd := exec.Command("sleep", "5")

// Does not wait for command to complete before returning
err := cmd.Start()
if err != nil {
    log.Fatal(err)
}

// Wait for cmd to Return
err = cmd.Wait()
```

```
log.Printf("Command finished with error: %v", err)
```

Ejecutando un comando dos veces

> Un Cmd no se puede reutilizar después de llamar a sus métodos Run, Output o CombinedOutput

Ejecutar un comando dos veces ***no*** *funcionará* :

```
cmd := exec.Command("xte", "key XF86AudioPlay")
_ := cmd.Run() // Play audio key press
// .. do something else
err := cmd.Run() // Pause audio key press, fails
```

> Error: exec: ya iniciado

Más bien, uno debe usar **dos** `exec.Command` **separados** . También es posible que necesite un cierto retraso entre los comandos.

```
cmd := exec.Command("xte", "key XF86AudioPlay")
_ := cmd.Run() // Play audio key press
// .. wait a moment
cmd := exec.Command("xte", "key XF86AudioPlay")
_ := cmd.Run() // Pause audio key press
```

Lea Comandos de ejecución en línea: https://riptutorial.com/es/go/topic/1097/comandos-de-ejecucion

Capítulo 17: Comenzando con el uso de Go Atom

Introducción

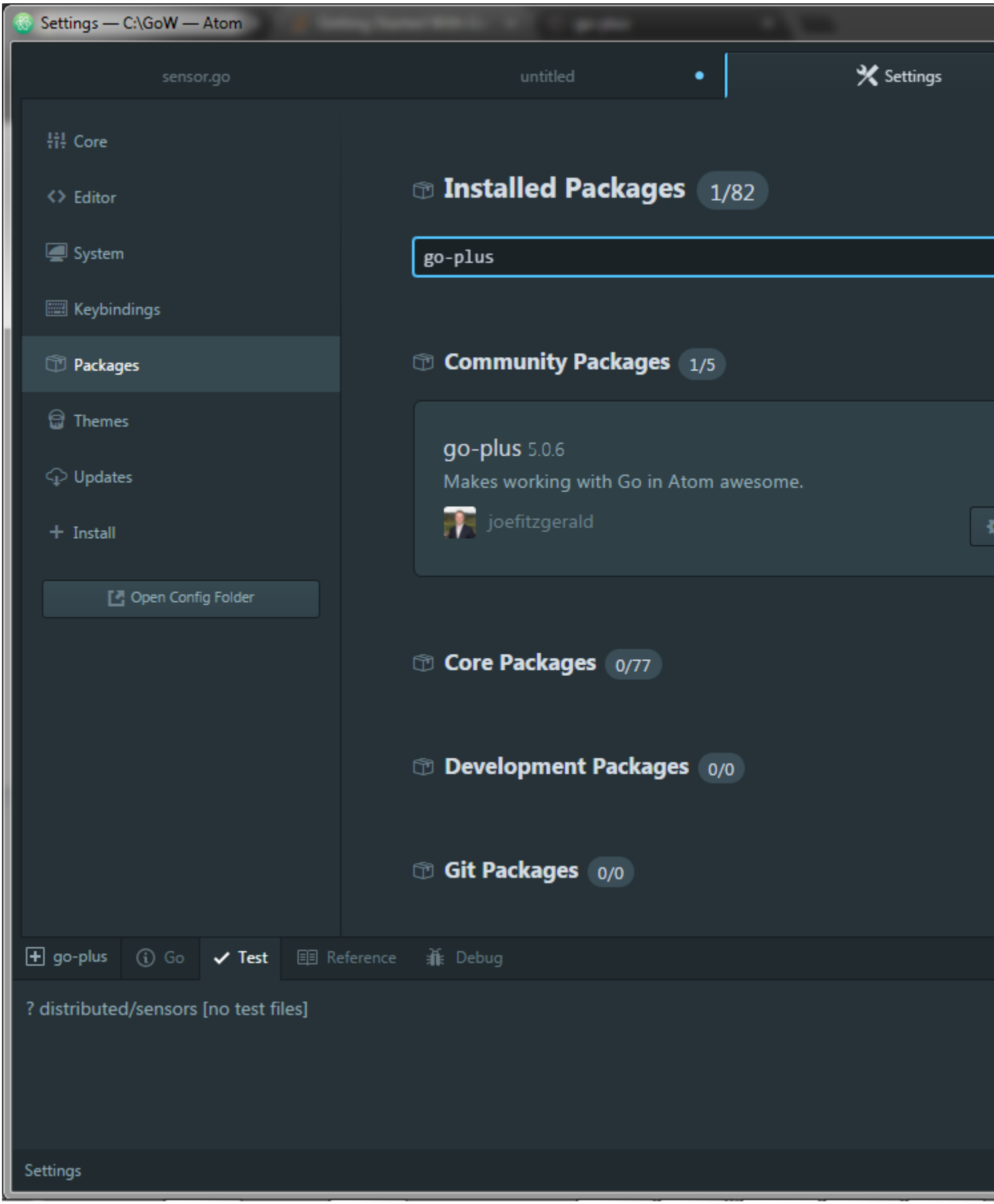
Después de instalar go (http://www.riptutorial.com/go/topic/198/getting-started-with-go) necesitará un entorno. Una forma eficiente y gratuita de comenzar es usar el editor de texto Atom (https://atom.io) y gulp. Una pregunta que tal vez cruzaste por tu mente es *¿por qué usar gulp?* Necesitamos un trago para el auto-completado. ¡Empecemos!

Examples

Obtener, instalar y configurar Atom & Gulp

1. Instalar atom Puedes obtener un átomo desde aquí.
2. Ir a la configuración del átomo (ctrl +,). Paquetes -> Instalar el paquete go-plus (go-plus)

Después de instalar go-plus en Atom:



3. Obtenga estas dependencias utilizando go get u otro administrador de dependencias: (abra una consola y ejecute estos comandos)

   ve a obtener -u golang.org/x/tools/cmd/goimports

ve a obtener -u golang.org/x/tools/cmd/gorename

ve a obtener -u github.com/sqs/goreturns

ve a obtener -u github.com/nsf/gocode

vaya a obtener -u github.com/alecthomas/gometalinter

ve a obtener -u github.com/zmb3/gogetdoc

ve a obtener -u github.com/rogpeppe/godef

ve a obtener -u golang.org/x/tools/cmd/guru

4. Instale Gulp (Gulpjs) usando npm o cualquier otro administrador de paquetes (gulp-getting-started-doc):

   $ npm instalar - trago global

Crear $ GO_PATH / gulpfile.js

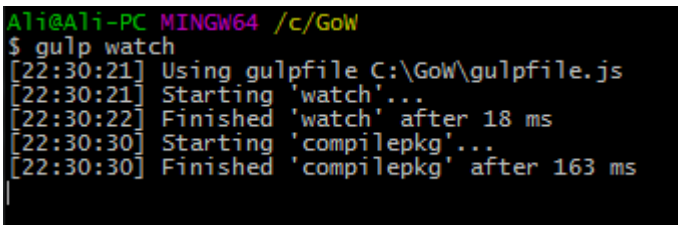
```
var gulp = require('gulp');
var path = require('path');
var shell = require('gulp-shell');

var goPath = 'src/mypackage/**/*.go';


gulp.task('compilepkg', function() {
  return gulp.src(goPath, {read: false})
    .pipe(shell(['go install <%= stripPath(file.path) %>'],
      {
          templateData: {
            stripPath: function(filePath) {
              var subPath = filePath.substring(process.cwd().length + 5);
              var pkg = subPath.substring(0, subPath.lastIndexOf(path.sep));
              return pkg;
            }
          }
      })
    );
});

gulp.task('watch', function() {
  gulp.watch(goPath, ['compilepkg']);
});
```

En el código anterior, definimos una tarea de *compliepkg* que se activará cada vez que se modifique cualquier archivo go en goPath (src / mypackage /) o en los subdirectorios. La tarea ejecutará el comando de shell, ir a instalar changed_file.go

Después de crear el archivo trago en la ruta de acceso y definir la tarea, abra una línea de comandos y ejecute:

trago reloj

Verás algo como esto cada vez que un archivo cambie:

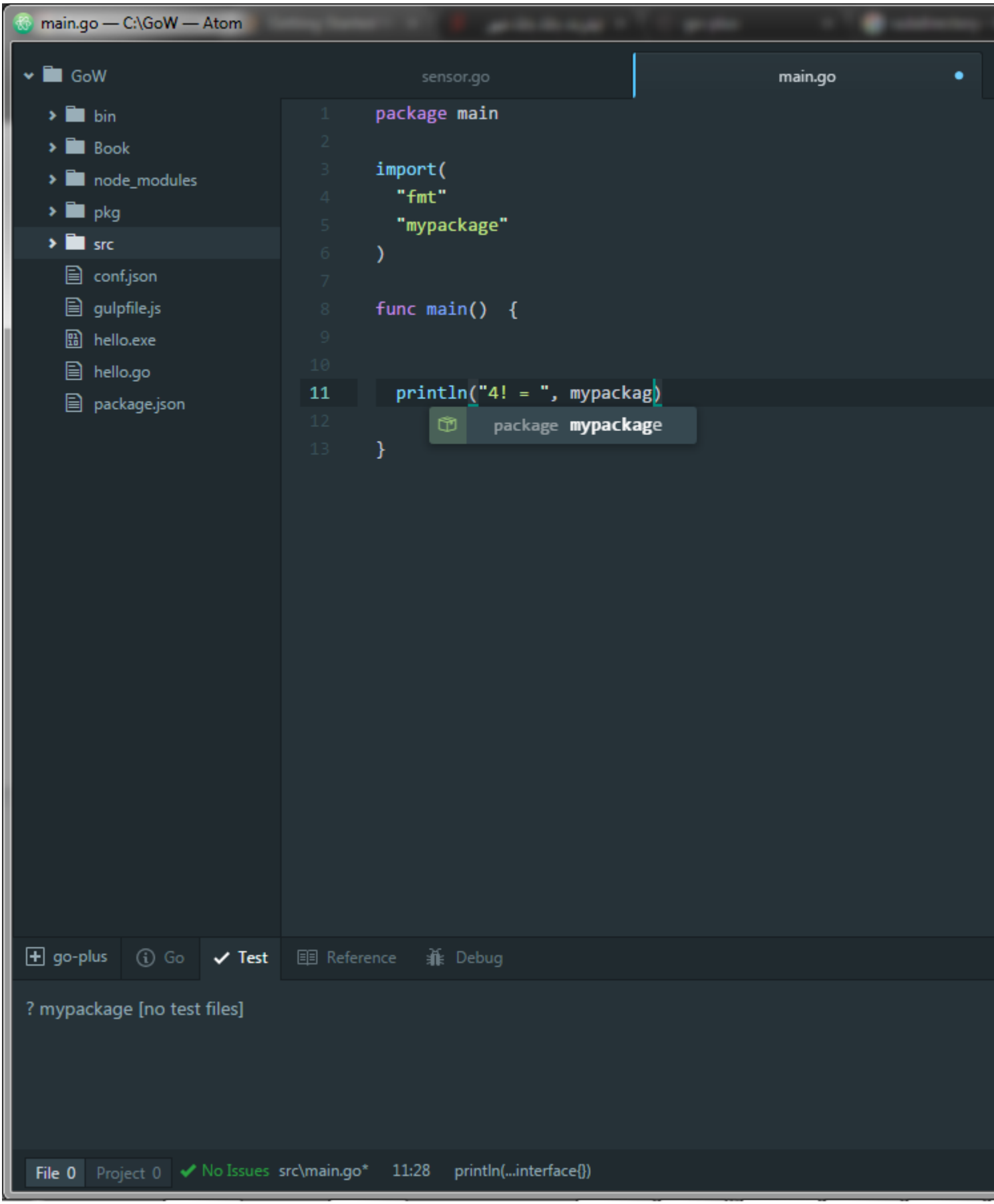


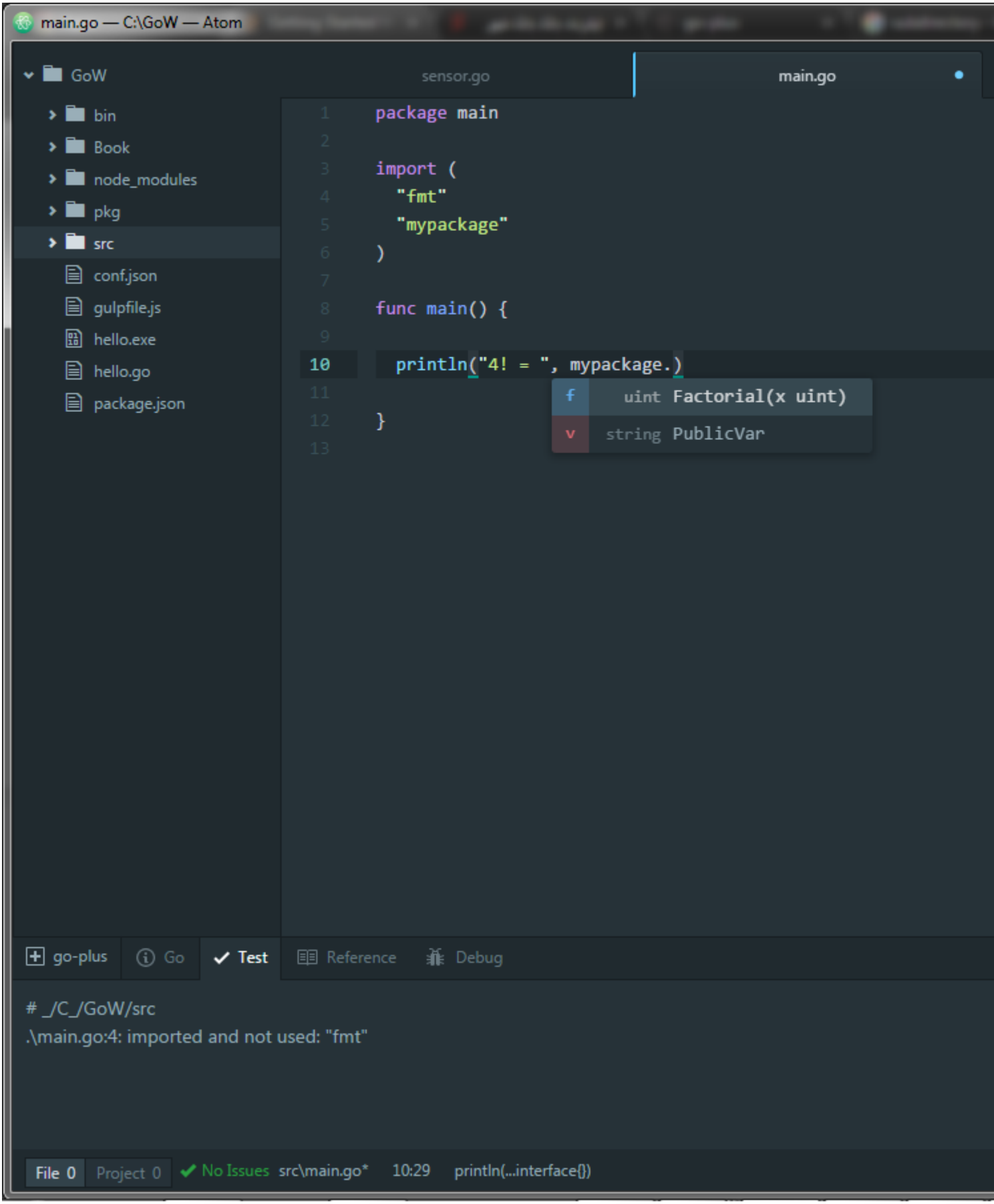
Crear $ GO_PATH / mypackage / source.go

```
package mypackage

var PublicVar string = "Hello, dear reader!"

//Calculates the factorial of given number recursively!
func Factorial(x uint) uint {
    if x == 0 {
        return 1
    }
    return x * Factorial(x-1)
}
```

Creando $ GO_PATH / main.go

Ahora puede comenzar a escribir su propio código go con autocompletado usando Atom y Gulp:


main.go — C:\GoW — Atom
GoW
bin
Book
node_modules
pkg
src
conf.json
gulpfile.js
hello.exe
hello.go
package.json
sensor.go
main.go
1 package main
2
3 import(
4 "fmt"
5 "mypackage"
6)
7
8 func main() {
9
10
11 println("4! = ", mypackag)
12
13 }
package mypackage
go-plus
Go
Test
Reference
Debug
? mypackage [no test files]
File 0
Project 0
No Issues
src\main.go*
11:28
println(...interface{})




```
package main

import (
    "fmt"
```

```
    "mypackage"
)

func main() {

    println("4! = ", mypackage.Factorial(4))

}
```

```
Ali@Ali-PC MINGW64 /c/GoW
$ go run src/main.go
4! =  24
```

Lea Comenzando con el uso de Go Atom en línea:
https://riptutorial.com/es/go/topic/8592/comenzando-con-el-uso-de-go-atom

Capítulo 18: Compilación cruzada

Introducción

El compilador Go puede producir binarios para muchas plataformas, es decir, procesadores y sistemas. A diferencia de la mayoría de los otros compiladores, no hay requisitos específicos para la compilación cruzada, es tan fácil de usar como la compilación regular.

Sintaxis

- GOOS = linux GOARCH = amd64 go build

Observaciones

Combinaciones de objetivos de arquitectura y sistema operativo compatibles (fuente)

$ GOOS	$ GOARCH
androide	brazo
Darwin	386
Darwin	amd64
Darwin	brazo
Darwin	brazo64
libélula	amd64
Freebsd	386
Freebsd	amd64
Freebsd	brazo
linux	386
linux	amd64
linux	brazo
linux	brazo64
linux	ppc64
linux	ppc64le

$ GOOS	$ GOARCH
linux	mips64
linux	mips64le
netbsd	386
netbsd	amd64
netbsd	brazo
openbsd	386
openbsd	amd64
openbsd	brazo
plan9	386
plan9	amd64
solaris	amd64
ventanas	386
ventanas	amd64

Examples

Compila todas las arquitecturas usando un Makefile

Este Makefile se cruzará compilando y comprimirá los ejecutables para Windows, Mac y Linux (ARM y x86).

```
# Replace demo with your desired executable name
appname := demo

sources := $(wildcard *.go)

build = GOOS=$(1) GOARCH=$(2) go build -o build/$(appname)$(3)
tar = cd build && tar -cvzf $(1)_$(2).tar.gz $(appname)$(3) && rm $(appname)$(3)
zip = cd build && zip $(1)_$(2).zip $(appname)$(3) && rm $(appname)$(3)

.PHONY: all windows darwin linux clean

all: windows darwin linux

clean:
    rm -rf build/

##### LINUX BUILDS #####
linux: build/linux_arm.tar.gz build/linux_arm64.tar.gz build/linux_386.tar.gz
```

```
build/linux_amd64.tar.gz

build/linux_386.tar.gz: $(sources)
    $(call build,linux,386,)
    $(call tar,linux,386)

build/linux_amd64.tar.gz: $(sources)
    $(call build,linux,amd64,)
    $(call tar,linux,amd64)

build/linux_arm.tar.gz: $(sources)
    $(call build,linux,arm,)
    $(call tar,linux,arm)

build/linux_arm64.tar.gz: $(sources)
    $(call build,linux,arm64,)
    $(call tar,linux,arm64)

##### DARWIN (MAC) BUILDS #####
darwin: build/darwin_amd64.tar.gz

build/darwin_amd64.tar.gz: $(sources)
    $(call build,darwin,amd64,)
    $(call tar,darwin,amd64)

##### WINDOWS BUILDS #####
windows: build/windows_386.zip build/windows_amd64.zip

build/windows_386.zip: $(sources)
    $(call build,windows,386,.exe)
    $(call zip,windows,386,.exe)

build/windows_amd64.zip: $(sources)
    $(call build,windows,amd64,.exe)
    $(call zip,windows,amd64,.exe)
```

(Tenga cuidado de que Makefile necesite pestañas duras, no espacios)

Recopilación cruzada simple con go build

Desde el directorio de su proyecto, ejecute el comando `go build` y especifique el sistema operativo y la arquitectura de destino con las variables de entorno `GOOS` y `GOARCH` :

Compilación para Mac (64 bits):

```
GOOS=darwin GOARCH=amd64 go build
```

Compilación para el procesador de Windows x86:

```
GOOS=windows GOARCH=386 go build
```

También es posible que desee establecer el nombre del archivo ejecutable de salida manualmente para realizar un seguimiento de la arquitectura:

```
GOOS=windows GOARCH=386 go build -o appname_win_x86.exe
```

A partir de la versión 1.7 y en adelante, puede obtener una lista de todas las combinaciones posibles de GOOS y GOARCH con:

```
go tool dist list
```

(o para un consumo más fácil de la máquina, `go tool dist list -json`)

Compilación cruzada utilizando gox

Otra solución conveniente para la compilación cruzada es el uso de `gox` : https://github.com/mitchellh/gox

Instalación

La instalación se realiza muy fácilmente ejecutando `go get github.com/mitchellh/gox` . El ejecutable resultante se coloca en el directorio binario de Go, por ejemplo, `/golang/bin` o `~/golang/bin` . Asegúrese de que esta carpeta sea parte de su ruta para poder utilizar el comando `gox` desde una ubicación arbitraria.

Uso

Desde dentro de la carpeta raíz de un proyecto de Go (donde realiza, por ejemplo, `go build`), ejecute `gox` para crear todos los binarios posibles para cualquier arquitectura (por ejemplo, x86, ARM) y sistema operativo (por ejemplo, Linux, macOS, Windows) que esté disponible.

Para compilar para un determinado sistema operativo, use, por ejemplo, `gox -os="linux"` lugar. También se podría definir la opción de arquitectura: `gox -osarch="linux/amd64" .`

Ejemplo simple: compilar helloworld.go para la arquitectura de brazo en una máquina Linux

Preparar helloworld.go (encuentra abajo)

```
package main

import "fmt"

func main(){
        fmt.Println("hello world")
}
```

Ejecute `GOOS=linux GOARCH=arm go build helloworld.go`

Copie el archivo `helloworld` generado (armar) en su máquina de destino.

Lea Compilación cruzada en línea: https://riptutorial.com/es/go/topic/1020/compilacion-cruzada

Capítulo 19: Concurrencia

Introducción

En Go, la concurrencia se logra mediante el uso de goroutines, y la comunicación entre goroutines generalmente se realiza con canales. Sin embargo, otros medios de sincronización, como las exclusiones mutuas y los grupos de espera, están disponibles, y deberían usarse siempre que sean más convenientes que los canales.

Sintaxis

- go doWork () // ejecuta la función doWork como una goroutine
- ch: = make (chan int) // declara nuevo canal de tipo int
- ch <- 1 // envío en un canal
- valor = <-ch // recibiendo de un canal

Observaciones

Goroutines en Go son similares a los hilos en otros idiomas en términos de uso. Internamente, Go crea una serie de subprocesos (especificados por `GOMAXPROCS`) y luego programa los goroutines para que se ejecuten en los subprocesos. Debido a este diseño, los mecanismos de concurrencia de Go son mucho más eficientes que los hilos en términos de uso de memoria y tiempo de inicialización.

Examples

Creando goroutines

Cualquier función se puede invocar como una goroutina prefijando su invocación con la palabra clave `go` :

```
func DoMultiply(x,y int) {
    // Simulate some hard work
    time.Sleep(time.Second * 1)
    fmt.Printf("Result: %d\n", x * y)
}

go DoMultiply(1,2) // first execution, non-blocking
go DoMultiply(3,4) // second execution, also non-blocking

// Results are printed after a single second only,
// not 2 seconds because they execute concurrently:
// Result: 2
// Result: 12
```

Tenga en cuenta que el valor de retorno de la función se ignora.

Hola mundo goroutine

Un solo canal, un solo goroutine, una escritura, una lectura.

```
package main

import "fmt"
import "time"

func main() {
    // create new channel of type string
    ch := make(chan string)

    // start new anonymous goroutine
    go func() {
        time.Sleep(time.Second)
        // send "Hello World" to channel
        ch <- "Hello World"
    }()
    // read from channel
    msg, ok := <-ch
    fmt.Printf("msg='%s', ok='%v'\n", msg, ok)
}
```

Ejecutalo en el patio

El canal `ch` es un **canal sin buffer o sincrónico** .

El `time.Sleep` está aquí para ilustrar la función `main()` **esperará** en el canal `ch` , lo que significa que la función literal ejecutada como goroutine tiene tiempo para enviar un valor a través de ese canal: el operador de recepción `<-ch` bloqueará la ejecución de `main()` . Si no fuera así, la goroutina se eliminaría cuando `main()` salga y no tendría tiempo de enviar su valor.

Esperando goroutines

Los programas de Go terminan cuando finaliza la función `main` , por lo que es una práctica común esperar a que todos los goroutines terminen. Una solución común para esto es usar un objeto sync.WaitGroup .

```
package main

import (
    "fmt"
    "sync"
)

var wg sync.WaitGroup // 1

func routine(i int) {
    defer wg.Done() // 3
    fmt.Printf("routine %v finished\n", i)
}

func main() {
    wg.Add(10) // 2
```

```
    for i := 0; i < 10; i++ {
        go routine(i) // *
    }
    wg.Wait() // 4
    fmt.Println("main finished")
}
```

Ejecutar el ejemplo en el patio de recreo.

Uso de WaitGroup en orden de ejecución:

1. Declaración de variable global. Hacerlo global es la forma más fácil de hacerlo visible a todas las funciones y métodos.
2. Aumentar el contador. Esto debe hacerse en la goroutine principal porque no hay garantía de que una goroutine recién iniciada se ejecute antes de las 4 debido a las garantías del modelo de memoria.
3. Disminuyendo el contador. Esto debe hacerse a la salida de una goroutina. Al utilizar una llamada diferida, nos aseguramos de que se llamará cada vez que la función finalice , sin importar cómo termine.
4. Esperando que el contador llegue a 0. Esto se debe hacer en la goroutine principal para evitar que el programa salga antes de que todos los goroutines hayan terminado.

* Los parámetros son evaluados antes de comenzar una nueva goroutina . Por lo tanto, es necesario definir sus valores explícitamente antes de `wg.Add(10)` para que el código de posible pánico no incremente el contador. Añadiendo 10 elementos al WaitGroup, por lo que esperará 10 elementos antes de que `wg.Wait` devuelva el control a `main()` goroutine. Aquí, el valor de i se define en el bucle for.

Usando cierres con goroutines en un bucle.

Cuando está en un bucle, la variable de bucle (val) en el siguiente ejemplo es una variable única que cambia de valor a medida que pasa por el bucle. Por lo tanto, uno debe hacer lo siguiente para pasar realmente cada valor de valores al goroutine:

```
for val := range values {
    go func(val interface{}) {
        fmt.Println(val)
    }(val)
}
```

Si tuviera que hacer solo go `func(val interface{}) { ... }()` sin pasar val, entonces el valor de `val` será el valor de `val` cuando se ejecuten los goroutines.

Otra forma de obtener el mismo efecto es:

```
for val := range values {
    val := val
    go func() {
        fmt.Println(val)
    }()
}
```

El `val := val` crea una nueva variable en cada iteración, a la que luego accede la goroutina.

Detener goroutines

```
package main

import (
    "log"
    "sync"
    "time"
)

func main() {
    // The WaitGroup lets the main goroutine wait for all other goroutines
    // to terminate. However, this is no implicit in Go. The WaitGroup must
    // be explicitely incremented prior to the execution of any goroutine
    // (i.e. before the `go` keyword) and it must be decremented by calling
    // wg.Done() at the end of every goroutine (typically via the `defer` keyword).
    wg := sync.WaitGroup{}

    // The stop channel is an unbuffered channel that is closed when the main
    // thread wants all other goroutines to terminate (there is no way to
    // interrupt another goroutine in Go). Each goroutine must multiplex its
    // work with the stop channel to guarantee liveness.
    stopCh := make(chan struct{})


    for i := 0; i < 5; i++ {
        // It is important that the WaitGroup is incremented before we start
        // the goroutine (and not within the goroutine) because the scheduler
        // makes no guarantee that the goroutine starts execution prior to
        // the main goroutine calling wg.Wait().
        wg.Add(1)
        go func(i int, stopCh <-chan struct{}) {
            // The defer keyword guarantees that the WaitGroup count is
            // decremented when the goroutine exits.
            defer wg.Done()

            log.Printf("started goroutine %d", i)

            select {
            // Since we never send empty structs on this channel we can
            // take the return of a receive on the channel to mean that the
            // channel has been closed (recall that receive never blocks on
            // closed channels).
            case <-stopCh:
                log.Printf("stopped goroutine %d", i)
            }
        }(i, stopCh)
    }

    time.Sleep(time.Second * 5)
    close(stopCh)
    log.Printf("stopping goroutines")
    wg.Wait()
    log.Printf("all goroutines stopped")
}
```

Ping pong con dos goroutines.

```
package main

import (
    "fmt"
    "time"
)

// The pinger prints a ping and waits for a pong
func pinger(pinger <-chan int, ponger chan<- int) {
    for {
        <-pinger
        fmt.Println("ping")
        time.Sleep(time.Second)
        ponger <- 1
    }
}

// The ponger prints a pong and waits for a ping
func ponger(pinger chan<- int, ponger <-chan int) {
    for {
        <-ponger
        fmt.Println("pong")
        time.Sleep(time.Second)
        pinger <- 1
    }
}

func main() {
    ping := make(chan int)
    pong := make(chan int)

    go pinger(ping, pong)
    go ponger(ping, pong)

    // The main goroutine starts the ping/pong by sending into the ping channel
    ping <- 1

    for {
        // Block the main thread until an interrupt
        time.Sleep(time.Second)
    }
}
```

Ejecuta una versión ligeramente modificada de este código en Go Playground

Lea Concurrencia en línea: https://riptutorial.com/es/go/topic/376/concurrencia

Capítulo 20: Constantes

Observaciones

Go admite constantes de caracteres, cadenas, valores booleanos y numéricos.

Examples

Declarando una constante

Las constantes se declaran como variables, pero usando la palabra clave `const` :

```
const Greeting string = "Hello World"
const Years int = 10
const Truth bool = true
```

Al igual que para las variables, los nombres que comienzan con mayúsculas se exportan (*público*), los nombres que comienzan con minúsculas no se exportan.

```
// not exported
const alpha string = "Alpha"
// exported
const Beta string = "Beta"
```

Las constantes se pueden usar como cualquier otra variable, excepto por el hecho de que el valor no se puede cambiar. Aquí hay un ejemplo:

```
package main

import (
    "fmt"
    "math"
)

const s string = "constant"

func main() {
    fmt.Println(s) // constant

    // A `const` statement can appear anywhere a `var` statement can.
    const n = 10
    fmt.Println(n)                          // 10
    fmt.Printf("n=%d is of type %T\n", n, n) // n=10 is of type int

    const m float64 = 4.3
    fmt.Println(m) // 4.3

    // An untyped constant takes the type needed by its context.
    // For example, here `math.Sin` expects a `float64`.
    const x = 10
    fmt.Println(math.Sin(x)) // -0.5440211108893699
```

```
}
```

Patio de recreo

Declaración de constantes múltiples

Puedes declarar múltiples constantes dentro del mismo bloque `const` :

```
const (
    Alpha = "alpha"
    Beta  = "beta"
    Gamma = "gamma"
)
```

E incrementa automáticamente las constantes con la palabra clave `iota` :

```
const (
    Zero = iota // Zero == 0
    One         // One  == 1
    Two         // Two  == 2
)
```

Para obtener más ejemplos del uso de `iota` para declarar constantes, consulte Iota .

También puede declarar constantes múltiples utilizando la asignación múltiple. Sin embargo, esta sintaxis puede ser más difícil de leer y generalmente se evita.

```
const Foo, Bar = "foo", "bar"
```

Constantes mecanografiadas vs. no tipificadas

Las constantes en Go pueden ser escritas o sin tipo. Por ejemplo, dada la siguiente cadena literal:

```
"bar"
```

se podría decir que el tipo del literal es una `string` , sin embargo, esto no es semánticamente correcto. En cambio, los literales son *constantes de cadena sin tipo* . Es una cadena (más correctamente, su *tipo predeterminado* es `string`), pero no es un **valor** Ir y, por lo tanto, no tiene ningún tipo hasta que se asigna o se usa en un contexto que se escribe. Esta es una distinción sutil, pero útil para entender.

Del mismo modo, si asignamos el literal a una constante:

```
const foo = "bar"
```

Permanece sin tipo ya que, por defecto, las constantes están sin tipo. También es posible declararlo como una *constante de cadena con tipo* :

```
const typedFoo string = "bar"
```

La diferencia entra en juego cuando intentamos asignar estas constantes en un contexto que tiene tipo. Por ejemplo, considere lo siguiente:

```
var s string
s = foo      // This works just fine
s = typedFoo // As does this

type MyString string
var mys MyString
mys = foo      // This works just fine
mys = typedFoo // cannot use typedFoo (type string) as type MyString in assignment
```

Lea Constantes en línea: https://riptutorial.com/es/go/topic/1047/constantes

Capítulo 21: Construir restricciones

Sintaxis

- // + construir etiquetas

Observaciones

Las etiquetas de compilación se utilizan para construir condicionalmente ciertos archivos en su código. Las etiquetas de compilación pueden ignorar los archivos que no desea que se compilen a menos que se incluyan explícitamente, o algunas etiquetas de compilación predefinidas se pueden usar para que un archivo solo se cree en una arquitectura o sistema operativo en particular.

Las etiquetas de compilación pueden aparecer en cualquier tipo de archivo de origen (no solo en Ir), pero deben aparecer cerca de la parte superior del archivo, precedidas solo por líneas en blanco y otros comentarios de línea. Estas reglas significan que en los archivos Go una restricción de compilación debe aparecer antes de la cláusula del paquete.

Una serie de etiquetas de compilación debe ir seguida de una línea en blanco.

Examples

Pruebas de integración separadas

Las restricciones de compilación se usan comúnmente para separar las pruebas de unidad normales de las pruebas de integración que requieren recursos externos, como una base de datos o acceso a la red. Para hacer esto, agregue una restricción de compilación personalizada en la parte superior del archivo de prueba:

```
// +build integration

package main

import (
    "testing"
)

func TestThatRequiresNetworkAccess(t *testing.T) {
    t.Fatal("It failed!")
}
```

El archivo de prueba no se compilará en el ejecutable de compilación a menos que se use la siguiente invocación de la `go test` :

```
go test -tags "integration"
```

Resultados:

```
$ go test
?       bitbucket.org/yourname/yourproject    [no test files]
$ go test -tags "integration"
--- FAIL: TestThatRequiresNetworkAccess (0.00s)
        main_test.go:10: It failed!
FAIL
exit status 1
FAIL    bitbucket.org/yourname/yourproject    0.003s
```

Optimizar implementaciones basadas en arquitectura.

Podemos optimizar una función xor simple solo para arquitecturas que admiten lecturas / escrituras no alineadas al crear dos archivos que definen la función y el prefijo con una restricción de compilación (para un ejemplo real del código xor que está fuera de alcance aquí, vea `crypto/cipher/xor.go` en la biblioteca estándar):

```
// +build 386 amd64 s390x

package cipher

func xorBytes(dst, a, b []byte) int { /* This function uses unaligned reads / writes to
optimize the operation */ }
```

y para otras arquitecturas:

```
// +build !386,!amd64,!s390x

package cipher

func xorBytes(dst, a, b []byte) int { /* This version of the function just loops and xors */ }
```

Lea Construir restricciones en línea: https://riptutorial.com/es/go/topic/2595/construir-restricciones

Capítulo 22: Contexto

Sintaxis

- escriba CancelFunc func ()
- func fondo () contexto
- func TODO () Context
- func WithCancel (Contexto principal) (Contexto ctx, cancelar CancelFunc)
- func WithDeadline (contexto principal, tiempo límite.Tiempo) (Context, CancelFunc)
- func WithTimeout (contexto principal, timeout time.Duration) (Context, CancelFunc)
- func WithValue (contexto principal, interfaz clave {}, interfaz val {})

Observaciones

El paquete de `context` (en Go 1.7) o el paquete `golang.org/x/net/context` (Pre 1.7) es una interfaz para crear contextos que se pueden usar para llevar los valores de alcance de la solicitud y los plazos entre los límites de la API y entre los servicios. Como una simple implementación de dicha interfaz.

aparte: la palabra "contexto" se usa para referirse a todo el árbol, oa hojas individuales en el árbol, por ejemplo. El `context.Context` real. Valores de `context.Context` .

En un nivel alto, un contexto es un árbol. Las nuevas hojas se agregan al árbol cuando se construyen (un `context.Context` con un valor padre), y las hojas nunca se eliminan del árbol. Cualquier contexto tiene acceso a todos los valores por encima de él (el acceso a los datos solo fluye hacia arriba), y si se cancela cualquier contexto, sus hijos también se cancelan (las señales de cancelación se propagan hacia abajo). La señal de cancelación se implementa mediante una función que devuelve un canal que se cerrará (legible) cuando se cancele el contexto; esto hace que los contextos sean una forma muy eficiente de implementar el patrón de concurrencia de la tubería y la cancelación , o tiempos de espera.

Por convención, las funciones que toman un contexto tienen el primer argumento `ctx context.Context` . Si bien esto es solo una convención, debe seguirse ya que muchas herramientas de análisis estático buscan específicamente este argumento. Dado que Context es una interfaz, también es posible convertir los datos existentes de tipo contexto (valores que se transmiten a lo largo de una cadena de llamadas de solicitud) en un contexto normal de Go y usarlos de una manera compatible hacia atrás con solo implementar algunos métodos. Además, los contextos son seguros para el acceso simultáneo, por lo que puedes usarlos desde muchos goroutines (ya sea que se ejecuten en subprocesos paralelos o como corrutines concurrentes) sin temor.

Otras lecturas

- https://blog.golang.org/context

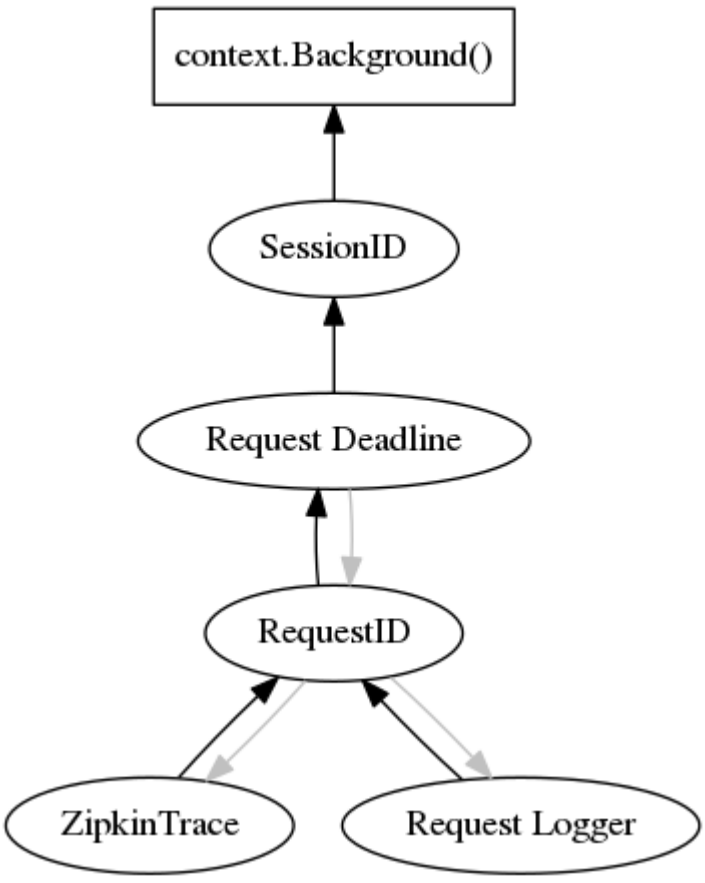
Examples

Árbol de contexto representado como un gráfico dirigido

Un árbol de contexto simple (que contiene algunos valores comunes que pueden ser objeto de un ámbito de solicitud e incluido en un contexto) construido a partir de un código de Go como el siguiente:

```
// Pseudo-Go
ctx := context.WithValue(
    context.WithDeadline(
        context.WithValue(context.Background(), sidKey, sid),
        time.Now().Add(30 * time.Minute),
    ),
    ridKey, rid,
)
trCtx := trace.NewContext(ctx, tr)
logCtx := myRequestLogging.NewContext(ctx, myRequestLogging.NewLogger())
```

Es un árbol que se puede representar como un gráfico dirigido que se ve así:



Cada contexto secundario tiene acceso a los valores de sus contextos principales, por lo que el acceso a los datos fluye hacia arriba en el árbol (representado por bordes negros). Las señales de cancelación, por otro lado, viajan por el árbol (si se cancela un contexto, también se cancelan todos sus hijos). El flujo de la señal de cancelación está representado por los bordes grises.

Usando un contexto para cancelar el trabajo

Pasar un contexto con un tiempo de espera (o con una función de cancelación) a una función de ejecución prolongada se puede usar para cancelar que las funciones funcionen:

```
ctx, _ := context.WithTimeout(context.Background(), 200*time.Millisecond)
for {
    select {
    case <-ctx.Done():
        return ctx.Err()
    default:
        // Do an iteration of some long running work here!
    }
}
```

Lea Contexto en línea: https://riptutorial.com/es/go/topic/2743/contexto

Capítulo 23: Criptografía

Introducción

Descubra cómo cifrar y descifrar datos con Go. Tenga en cuenta que este no es un curso sobre criptografía, sino cómo lograrlo con Go.

Examples

Cifrado y descifrado

Prefacio

Este es un ejemplo detallado sobre cómo cifrar y descifrar datos con Go. El código de uso se acorta, por ejemplo, no se menciona el manejo de errores. El proyecto de trabajo completo con manejo de errores e interfaz de usuario se puede encontrar en Github aquí .

Cifrado

Introducción y datos

Este ejemplo describe un cifrado y descifrado completo en Go. Para ello, necesitamos un dato. En este ejemplo, usamos nuestro propio `secret` estructura de datos:

```
type secret struct {
    DisplayName       string
    Notes             string
    Username          string
    EMail             string
    CopyMethod        string
    Password          string
    CustomField01Name string
    CustomField01Data string
    CustomField02Name string
    CustomField02Data string
    CustomField03Name string
    CustomField03Data string
    CustomField04Name string
    CustomField04Data string
    CustomField05Name string
    CustomField05Data string
    CustomField06Name string
    CustomField06Data string
}
```

A continuación, queremos cifrar tal `secret` . El ejemplo completo de trabajo se puede encontrar aquí (enlace a Github) . Ahora, el proceso paso a paso:

Paso 1

En primer lugar, necesitamos un tipo de contraseña maestra para proteger el secreto:
`masterPassword := "PASS"`

Paso 2

Todos los métodos criptográficos que trabajan con bytes en lugar de cadenas. Por lo tanto, construimos una matriz de bytes con los datos de nuestro secreto.

```
secretBytesDecrypted :=
[]byte(fmt.Sprintf("%s\n%s\n%s\n%s\n%s\n%s\n%s\n%s\n%s\n%s\n%s\n%s\n%s\n%s\n%s\n%s\n%s\n%s\n%s\n",
    artifact.DisplayName,
    strings.Replace(artifact.Notes, "\n", string(65000), -1),
    artifact.Username,
    artifact.EMail,
    artifact.CopyMethod,
    artifact.Password,
    artifact.CustomField01Name,
    artifact.CustomField01Data,
    artifact.CustomField02Name,
    artifact.CustomField02Data,
    artifact.CustomField03Name,
    artifact.CustomField03Data,
    artifact.CustomField04Name,
    artifact.CustomField04Data,
    artifact.CustomField05Name,
    artifact.CustomField05Data,
    artifact.CustomField06Name,
    artifact.CustomField06Data,
))
```

Paso 3

Creamos algo de sal para prevenir ataques a la mesa del arco iris, cf. Wikipedia : `saltBytes := uuid.NewV4().Bytes()` . Aquí, usamos un UUID v4 que no es predecible.

Etapa 4

Ahora, podemos derivar una clave y un vector a partir de la contraseña maestra y el salt aleatorio, con respecto al RFC 2898:

```
keyLength := 256
rfc2898Iterations := 6

keyVectorData := pbkdf2.Key(masterPassword, saltBytes, rfc2898Iterations,
(keyLength/8)+aes.BlockSize, sha1.New)
keyBytes := keyVectorData[:keyLength/8]
```

```
vectorBytes := keyVectorData[keyLength/8:]
```

Paso 5

El modo CBC deseado funciona con bloques enteros. Por lo tanto, tenemos que verificar si nuestros datos están alineados con un bloque completo. Si no, tenemos que rellenarla:

```
if len(secretBytesDecrypted)%aes.BlockSize != 0 {
    numberNecessaryBlocks := int(math.Ceil(float64(len(secretBytesDecrypted)) /
float64(aes.BlockSize)))
    enhanced := make([]byte, numberNecessaryBlocks*aes.BlockSize)
    copy(enhanced, secretBytesDecrypted)
    secretBytesDecrypted = enhanced
}
```

Paso 6

Ahora creamos un cifrado AES: `aesBlockEncrypter, aesErr := aes.NewCipher(keyBytes)`

Paso 7

Reservamos la memoria necesaria para los datos cifrados: `encryptedData := make([]byte, len(secretBytesDecrypted))` . En el caso de AES-CBC, los datos cifrados tenían la misma longitud que los datos no cifrados.

Paso 8

Ahora, debemos crear el cifrador y cifrar los datos:

```
aesEncrypter := cipher.NewCBCEncrypter(aesBlockEncrypter, vectorBytes)
aesEncrypter.CryptBlocks(encryptedData, secretBytesDecrypted)
```

Ahora, los datos encriptados están dentro de la variable `encryptedData` .

Paso 9

Los datos cifrados deben ser almacenados. Pero no solo los datos: sin la sal, los datos cifrados no se podrían descifrar. Por lo tanto, debemos usar algún tipo de formato de archivo para gestionar esto. Aquí, codificamos los datos cifrados como base64, cf. Wikipedia :

```
encodedBytes := make([]byte, base64.StdEncoding.EncodedLen(len(encryptedData)))
base64.StdEncoding.Encode(encodedBytes, encryptedData)
```

A continuación, definimos el contenido de nuestro archivo y nuestro propio formato de archivo. El formato se ve así: `salt[0x10]base64 content` . Primero, almacenamos la sal. Para marcar el comienzo del contenido base64, almacenamos el byte `10` . Esto funciona, porque base64 no usa

este valor. Por lo tanto, podríamos encontrar el inicio de base64 buscando la primera aparición de `10` desde el final hasta el principio del archivo.

```
fileContent := make([]byte, len(saltBytes))
copy(fileContent, saltBytes)
fileContent = append(fileContent, 10)
fileContent = append(fileContent, encodedBytes...)
```

Paso 10

Finalmente, podríamos escribir nuestro archivo: `writeErr := ioutil.WriteFile("my secret.data", fileContent, 0644)` .

Descifrado

Introducción y datos

En cuanto al cifrado, necesitamos algunos datos para trabajar. Por lo tanto, asumimos que tenemos un archivo cifrado y la estructura `secret` mencionada. El objetivo es leer los datos cifrados del archivo, descifrarlos y crear una instancia de la estructura.

Paso 1

El primer paso es idéntico al cifrado: necesitamos un tipo de contraseña maestra para descifrar el secreto: `masterPassword := "PASS"` .

Paso 2

Ahora, leemos los datos cifrados del archivo: `encryptedFileData, bytesErr := ioutil.ReadFile(filename)` .

Paso 3

Como se mencionó anteriormente, podríamos dividir la sal y los datos cifrados por el byte `10` delimitador, buscados hacia atrás desde el final hasta el principio:

```
for n := len(encryptedFileData) - 1; n > 0; n-- {
    if encryptedFileData[n] == 10 {
        saltBytes = encryptedFileData[:n]
        encryptedBytesBase64 = encryptedFileData[n+1:]
        break
    }
}
```

Etapa 4

A continuación, debemos decodificar los bytes codificados en base64:

```
decodedBytes := make([]byte, len(encryptedBytesBase64))
countDecoded, decodedErr := base64.StdEncoding.Decode(decodedBytes, encryptedBytesBase64)
encryptedBytes = decodedBytes[:countDecoded]
```

Paso 5

Ahora, podemos derivar una clave y un vector a partir de la contraseña maestra y el salt aleatorio, con respecto al RFC 2898:

```
keyLength := 256
rfc2898Iterations := 6

keyVectorData := pbkdf2.Key(masterPassword, saltBytes, rfc2898Iterations,
(keyLength/8)+aes.BlockSize, sha1.New)
keyBytes := keyVectorData[:keyLength/8]
vectorBytes := keyVectorData[keyLength/8:]
```

Paso 6

Cree un cifrado AES: `aesBlockDecrypter, aesErr := aes.NewCipher(keyBytes)` .

Paso 7

Reserve la memoria necesaria para los datos descifrados: `decryptedData := make([]byte, len(encryptedBytes))` . Por definición, tiene la misma longitud que los datos encriptados.

Paso 8

Ahora, crea el descifrador y descifra los datos:

```
aesDecrypter := cipher.NewCBCDecrypter(aesBlockDecrypter, vectorBytes)
aesDecrypter.CryptBlocks(decryptedData, encryptedBytes)
```

Paso 9

Convierta los bytes leídos en cadena: `decryptedString := string(decryptedData)` . Como necesitamos líneas, divida la cadena: `lines := strings.Split(decryptedString, "\n")` .

Paso 10

Construye un `secret` fuera de las líneas:

```
artifact := secret{}
artifact.DisplayName = lines[0]
artifact.Notes = lines[1]
artifact.Username = lines[2]
artifact.EMail = lines[3]
artifact.CopyMethod = lines[4]
artifact.Password = lines[5]
artifact.CustomField01Name = lines[6]
artifact.CustomField01Data = lines[7]
artifact.CustomField02Name = lines[8]
artifact.CustomField02Data = lines[9]
artifact.CustomField03Name = lines[10]
artifact.CustomField03Data = lines[11]
artifact.CustomField04Name = lines[12]
artifact.CustomField04Data = lines[13]
artifact.CustomField05Name = lines[14]
artifact.CustomField05Data = lines[15]
artifact.CustomField06Name = lines[16]
artifact.CustomField06Data = lines[17]
```

Finalmente, vuelva a crear los saltos de línea dentro del campo de notas: `artifact.Notes = strings.Replace(artifact.Notes, string(65000), "\n", -1)` .

Lea Criptografía en línea: https://riptutorial.com/es/go/topic/10065/criptografia

Capítulo 24: Cuerda

Introducción

Una cadena es, en efecto, un segmento de solo lectura de bytes. En Go, una cadena literal siempre contendrá una representación UTF-8 válida de su contenido.

Sintaxis

- variableName: = "Hello World" // declara una cadena
- variableName: = `Hello World` // declara una cadena literal en bruto
- variableName: = "Hola" + "Mundo" // concatena cadenas
- substring: = "Hello World" [0: 4] // obtener una parte de la cadena
- letter: = "Hello World" [6] // obtener un carácter de la cadena
- fmt.Sprintf ("% s", "Hello World") // formatea una cadena

Examples

Tipo de cadena

El tipo de `string` permite almacenar texto, que es una serie de caracteres. Hay múltiples formas de crear cadenas. Se crea una cadena literal escribiendo el texto entre comillas dobles.

```
text := "Hello World"
```

Debido a que las cadenas Go son compatibles con UTF-8, el ejemplo anterior es perfectamente válido. Las cadenas contienen bytes arbitrarios, lo que no significa necesariamente que todas las cadenas contengan UTF-8 válido, pero los literales de cadena siempre tendrán secuencias UTF-8 válidas.

El valor cero de las cadenas es una cadena vacía `""` .

Las cadenas se pueden concatenar usando el operador `+` .

```
text := "Hello " + "World"
```

Las cadenas también pueden definirse usando backticks `` . Esto crea un literal de cadena sin formato que significa que los caracteres no se escaparán.

```
text1 := "Hello\nWorld"
text2 := `Hello
World`
```

En el ejemplo anterior, `text1` escapa al carácter `\n` que representa una nueva línea, mientras que `text2` contiene el nuevo carácter de línea directamente. Si comparas `text1 == text2` el resultado

será `true` .

Sin embargo, `text2 := `Hello\nWorld`` no escapará al carácter `\n` , lo que significa que la cadena contiene el texto `Hello\nWorld` sin una nueva línea. Sería el equivalente de escribir `text1 := "Hello\\nWorld"` .

Formato de texto

El paquete `fmt` implementa funciones para imprimir y formatear texto usando *verbos de* formato. Los verbos se representan con un signo de porcentaje.

Verbos generales

```
%v    // the value in a default format
      // when printing structs, the plus flag (%+v) adds field names
%#v   // a Go-syntax representation of the value
%T    // a Go-syntax representation of the type of the value
%%    // a literal percent sign; consumes no value
```

Booleano

```
%t    // the word true or false
```

Entero:

```
%b    // base 2
%c    // the character represented by the corresponding Unicode code point
%d    // base 10
%o    // base 8
%q    // a single-quoted character literal safely escaped with Go syntax.
%x    // base 16, with lower-case letters for a-f
%X    // base 16, with upper-case letters for A-F
%U    // Unicode format: U+1234; same as "U+%04X"
```

Elementos de coma flotante y complejos:

```
%b    // decimalless scientific notation with exponent a power of two,
      // in the manner of strconv.FormatFloat with the 'b' format,
      // e.g. -123456p-78
%e    // scientific notation, e.g. -1.234456e+78
%E    // scientific notation, e.g. -1.234456E+78
%f    // decimal point but no exponent, e.g. 123.456
%F    // synonym for %f
%g    // %e for large exponents, %f otherwise
%G    // %E for large exponents, %F otherwise
```

Cadena y segmento de bytes (tratados de manera equivalente con estos verbos):

```
%s    // the uninterpreted bytes of the string or slice
%q    // a double-quoted string safely escaped with Go syntax
%x    // base 16, lower-case, two characters per byte
%X    // base 16, upper-case, two characters per byte
```

Puntero:

```
%p    // base 16 notation, with leading 0x
```

Usando los verbos, puedes crear cadenas que concatenen múltiples tipos:

```
text1 := fmt.Sprintf("Hello %s", "World")
text2 := fmt.Sprintf("%d + %d = %d", 2, 3, 5)
text3 := fmt.Sprintf("%s, %s (Age: %d)", "Obama", "Barack", 55)
```

La función `Sprintf` formatea la cadena en el primer parámetro, reemplazando los verbos con el valor de los valores en los siguientes parámetros y devuelve el resultado. Al igual que `Sprintf` , la función `Printf` también se formatea, pero en lugar de devolver el resultado, imprime la cadena.

paquete de cuerdas

- strings.Contains

  ```
  fmt.Println(strings.Contains("foobar", "foo")) // true
  fmt.Println(strings.Contains("foobar", "baz")) // false
  ```

- strings.HasPrefix

  ```
  fmt.Println(strings.HasPrefix("foobar", "foo")) // true
  fmt.Println(strings.HasPrefix("foobar", "baz")) // false
  ```

- strings.HasSuffix

  ```
  fmt.Println(strings.HasSuffix("foobar", "bar")) // true
  fmt.Println(strings.HasSuffix("foobar", "baz")) // false
  ```

- strings.Join

  ```
  ss := []string{"foo", "bar", "bar"}
  fmt.Println(strings.Join(ss, ", ")) // foo, bar, baz
  ```

- strings.Replace

  ```
  fmt.Println(strings.Replace("foobar", "bar", "baz", 1)) // foobaz
  ```

- strings.Split

  ```
  s := "foo, bar, bar"
  fmt.Println(strings.Split(s, ", ")) // [foo bar baz]
  ```

- strings.ToLower

  ```
  fmt.Println(strings.ToLower("FOOBAR")) // foobar
  ```

- strings.ToUpper

```
fmt.Println(strings.ToUpper("foobar")) // FOOBAR
```

- `strings.TrimSpace`

```
fmt.Println(strings.TrimSpace("  foobar  ")) // foobar
```

Más: https://golang.org/pkg/strings/ .

Lea Cuerda en línea: https://riptutorial.com/es/go/topic/9666/cuerda

Capítulo 25: Derivación

Examples

Cambiar declaraciones

Una simple declaración de `switch` :

```
switch a + b {
case c:
    // do something
case d:
    // do something else
default:
    // do something entirely different
}
```

El ejemplo anterior es equivalente a:

```
if a + b == c {
    // do something
} else if a + b == d {
    // do something else
} else {
    // do something entirely different
}
```

La cláusula `default` es opcional y se ejecutará si y solo si ninguno de los casos se compara como verdadero, incluso si no aparece en último lugar, lo que es aceptable. Lo siguiente es semánticamente igual al primer ejemplo:

```
switch a + b {
default:
    // do something entirely different
case c:
    // do something
case d:
    // do something else
}
```

Esto podría ser útil si tiene la intención de usar la declaración de `fallthrough` en la cláusula `default` , que debe ser la última declaración en un caso y hace que la ejecución del programa continúe con el siguiente caso:

```
switch a + b {
default:
    // do something entirely different, but then also do something
    fallthrough
case c:
    // do something
```

```
case d:
    // do something else
}
```

Una expresión de interruptor vacía es implícitamente `true` :

```
switch {
case a + b == c:
    // do something
case a + b == d:
    // do something else
}
```

Las instrucciones de cambio admiten una instrucción simple similar a las declaraciones `if` :

```
switch n := getNumber(); n {
case 1:
    // do something
case 2:
    // do something else
}
```

Los casos se pueden combinar en una lista separada por comas si comparten la misma lógica:

```
switch a + b {
case c, d:
    // do something
default:
    // do something entirely different
}
```

Si las declaraciones

Una simple declaración `if` :

```
if a == b {
    // do something
}
```

Tenga en cuenta que no hay paréntesis alrededor de la condición y que la llave de apertura `{` debe estar en la misma línea. Lo siguiente *no se* compilará:

```
if a == b
{
    // do something
}
```

Una declaración `if` haciendo uso de `else` :

```
if a == b {
    // do something
} else if a == c {
    // do something else
} else {
    // do something entirely different
}
```

Según la documentación de golang.org , "la expresión puede estar precedida por una declaración simple, que se ejecuta antes de que se evalúe la expresión". Las variables declaradas en esta simple declaración tienen el alcance de la instrucción `if` y no se puede acceder a ellas desde fuera:

```
if err := attemptSomething(); err != nil {
    // attemptSomething() was successful!
} else {
    // attemptSomething() returned an error; handle it
}
fmt.Println(err) // compiler error, 'undefined: err'
```

Tipo de cambio de instrucciones

Un simple interruptor de tipo:

```
// assuming x is an expression of type interface{}
switch t := x.(type) {
case nil:
    // x is nil
    // t will be type interface{}
case int:
    // underlying type of x is int
    // t will be int in this case as well
case string:
    // underlying type of x is string
    // t will be string in this case as well
case float, bool:
    // underlying type of x is either float or bool
    // since we don't know which, t is of type interface{} in this case
default:
    // underlying type of x was not any of the types tested for
    // t is interface{} in this type
}
```

Puede probar cualquier tipo, incluidos `error` , tipos definidos por el usuario, tipos de interfaz y tipos de funciones:

```
switch t := x.(type) {
case error:
    log.Fatal(t)
case myType:
    fmt.Println(myType.message)
case myInterface:
    t.MyInterfaceMethod()
```

```
case func(string) bool:
    if t("Hello world?") {
        fmt.Println("Hello world!")
    }
}
```

Goto declaraciones

Una instrucción `goto` transfiere el control a la instrucción con la etiqueta correspondiente dentro de la misma función. La ejecución de la instrucción `goto` no debe hacer que ninguna variable entre en el alcance que aún no estuviera dentro del alcance en el punto del `goto` .

por ejemplo, consulte el código fuente de la biblioteca estándar:
https://golang.org/src/math/gamma.go :

```
    for x < 0 {
        if x > -1e-09 {
            goto small
        }
        z = z / x
        x = x + 1
    }
    for x < 2 {
        if x < 1e-09 {
            goto small
        }
        z = z / x
        x = x + 1
    }

    if x == 2 {
        return z
    }

    x = x - 2
    p = (((((x*_gamP[0]+_gamP[1])*x+_gamP[2])*x+_gamP[3])*x+_gamP[4])*x+_gamP[5])*x + _gamP[6]
    q =
((((((x*_gamQ[0]+_gamQ[1])*x+_gamQ[2])*x+_gamQ[3])*x+_gamQ[4])*x+_gamQ[5])*x+_gamQ[6])*x +
_gamQ[7]
    return z * p / q

small:
    if x == 0 {
        return Inf(1)
    }
    return z / ((1 + Euler*x) * x)
```

Declaraciones de ruptura de continuar

La instrucción break, en la ejecución, hace que el bucle actual obligue a salir.

paquete principal

```
import "fmt"
```

```
func main() {
    i:=0
    for true {
      if i>2 {
        break
        }
    fmt.Println("Iteration : ",i)
    i++
    }
}
```

La instrucción continue, en ejecución, mueve el control al inicio del bucle.

```
import "fmt"

func main() {
    j:=100
    for j<110 {
     j++
     if j%2==0 {
        continue
        }
    fmt.Println("Var : ",j)
    }
}
```

Interruptor de interrupción / interrupción dentro del bucle

```
import "fmt"

func main() {
    j := 100

loop:
    for j < 110 {
        j++

        switch j % 3 {
        case 0:
            continue loop
        case 1:
            break loop
        }

        fmt.Println("Var : ", j)
    }
}
```

Lea Derivación en línea: https://riptutorial.com/es/go/topic/1342/derivacion

Capítulo 26: Desarrollando para múltiples plataformas con compilación condicional

Introducción

La compilación condicional basada en la plataforma viene en dos formas en Go, una con sufijos de archivo y la otra con etiquetas de compilación.

Sintaxis

- Después de " `// +build` ", puede seguir una sola plataforma o una lista
- Se puede revertir la plataforma precediéndola por `!` firmar
- Lista de plataformas separadas en el espacio son oradas juntas

Observaciones

Advertencias para las etiquetas de construcción:

- La restricción de `// +build` debe colocar en la parte superior del archivo, incluso antes de la cláusula del paquete.
- Debe ir seguido de una línea en blanco para separarse de los comentarios del paquete.

Lista de plataformas válidas para etiquetas de compilación y sufijos de archivos
androide
Darwin
libélula
Freebsd
linux
netbsd
openbsd
plan9
solaris
ventanas

Consulte la lista de `$GOOS` en https://golang.org/doc/install/source#environment para obtener la lista

de plataformas más actualizada.

Examples

Crear etiquetas

```
// +build linux

package lib

var OnlyAccessibleInLinux int // Will only be compiled in Linux
```

Niega una plataforma colocando `!` antes de eso:

```
// +build !windows

package lib

var NotWindows int // Will be compiled in all platforms but not Windows
```

La lista de plataformas se puede especificar separándolas con espacios

```
// +build linux darwin plan9

package lib

var SomeUnix int // Will be compiled in linux, darwin and plan9 but not on others
```

Sufijo de archivo

Si nombra su archivo `lib_linux.go` , todo el contenido de ese archivo solo se compilará en entornos de Linux:

```
package lib

var OnlyCompiledInLinux string
```

Definiendo comportamientos separados en diferentes plataformas.

Diferentes plataformas pueden tener implementaciones separadas del mismo método. Este ejemplo también ilustra cómo las etiquetas de compilación y los sufijos de archivos se pueden usar juntos.

Archivo `main.go` :

```
package main

import "fmt"

func main() {
```

```
    fmt.Println("Hello World from Conditional Compilation Doc!")
    printDetails()
}
```

`details.go` :

```
// +build !windows

package main

import "fmt"

func printDetails() {
    fmt.Println("Some specific details that cannot be found on Windows")
}
```

`details_windows.go` :

```
package main

import "fmt"

func printDetails() {
    fmt.Println("Windows specific details")
}
```

Lea Desarrollando para múltiples plataformas con compilación condicional en línea: https://riptutorial.com/es/go/topic/8599/desarrollando-para-multiples-plataformas-con-compilacion-condicional

Capítulo 27: E / S de consola

Examples

Leer entrada desde consola

Utilizando `scanf`

> Scanf escanea el texto leído de la entrada estándar, almacenando sucesivos valores separados por espacios en argumentos sucesivos según lo determine el formato. Devuelve el número de elementos escaneados con éxito. Si eso es menor que el número de argumentos, err informará por qué. Las líneas nuevas en la entrada deben coincidir con las líneas nuevas en el formato. La única excepción: el verbo% c siempre escanea la siguiente runa en la entrada, incluso si es un espacio (o pestaña, etc.) o nueva línea.

```
# Read integer
var i int
fmt.Scanf("%d", &i)

# Read string
var str string
fmt.Scanf("%s", &str)
```

Utilizando `scan`

> Escanear escanea el texto leído de la entrada estándar, almacenando sucesivos valores separados por espacios en argumentos sucesivos. Las líneas nuevas cuentan como espacio. Devuelve el número de elementos escaneados con éxito. Si eso es menor que el número de argumentos, err informará por qué.

```
# Read integer
var i int
fmt.Scan(&i)

# Read string
var str string
fmt.Scan(&str)
```

Utilizando `scanln`

> Sscanln es similar a Sscan, pero deja de escanear en una nueva línea y después del elemento final debe haber una nueva línea o EOF.

```
# Read string
var input string
fmt.Scanln(&input)
```

Usando `bufio`

```
# Read using Reader
reader := bufio.NewReader(os.Stdin)
text, err := reader.ReadString('\n')

# Read using Scanner
scanner := bufio.NewScanner(os.Stdin)
for scanner.Scan() {
    fmt.Println(scanner.Text())
}
```

Lea E / S de consola en línea: https://riptutorial.com/es/go/topic/9741/e---s-de-consola

Capítulo 28: El comando go

Introducción

El comando `go` es un programa de línea de comandos que permite la administración del desarrollo de Go. Permite la creación, ejecución y prueba de código, así como una variedad de otras tareas relacionadas con Go.

Examples

Corre

`go run` ejecutará un programa sin crear un archivo ejecutable. Sobre todo útil para el desarrollo. `run` solo ejecutará paquetes cuyo *nombre de paquete* sea **main** .

Para demostrarlo, usaremos un ejemplo simple de Hello World `main.go` :

```
package main

import fmt

func main() {
    fmt.Println("Hello, World!")
}
```

Ejecutar sin compilar en un archivo:

```
go run main.go
```

Salida:

```
Hello, World!
```

Ejecutar varios archivos en el paquete

Si el paquete es **principal** y se divide en varios archivos, uno debe incluir los otros archivos en el comando de `run` :

```
go run main.go assets.go
```

Ir a construir

`go build` compilará un programa en un archivo ejecutable.

Para demostrarlo, usaremos un ejemplo simple de Hello World main.go:

```
package main

import fmt

func main() {
    fmt.Println("Hello, World!")
}
```

Compila el programa:

```
go build main.go
```

`build` crea un programa ejecutable, en este caso: `main` o `main.exe` . A continuación, puede ejecutar este archivo para ver la salida `Hello, World!` . También puede copiarlo en un sistema similar que no tenga instalado Go, *hacerlo ejecutable* y ejecutarlo allí.

Especifique el sistema operativo o la arquitectura en la construcción:

Puede especificar qué sistema o arquitectura construirá modificando el `env` antes de `build` :

```
env GOOS=linux go build main.go # builds for Linux
env GOARCH=arm go build main.go # builds for ARM architecture
```

Construir múltiples archivos

Si su paquete se divide en varios archivos **y** el nombre del paquete es **main** (es decir, *no es un paquete importable*), debe especificar todos los archivos para compilar:

```
go build main.go assets.go # outputs an executable: main
```

Construyendo un paquete

Para construir un paquete llamado `main` , simplemente puede usar:

```
go build . # outputs an executable with name as the name of enclosing folder
```

Ir limpio

`go clean` limpiará todos los archivos temporales creados al invocar `go build` en un programa. También limpiará los archivos sobrantes de Makefiles.

Ir fmt

`go fmt` formateará el código fuente de un programa de una manera ordenada e idiomática que sea fácil de leer y entender. Se recomienda utilizar `go fmt` en cualquier fuente antes de enviarlo para

que el público lo vea o se comprometa con un sistema de control de versiones, para facilitar la lectura.

Para formatear un archivo:

```
go fmt main.go
```

O todos los archivos en un directorio:

```
go fmt myProject
```

También puede usar `gofmt -s` (**no** `go fmt`) para intentar simplificar cualquier código que pueda.

`gofmt` (**no** `go fmt`) también se puede usar para refactorizar el código. Entiende a Go, por lo que es más poderoso que usar una simple búsqueda y reemplazo. Por ejemplo, dado este programa (`main.go`):

```
package main

type Example struct {
    Name string
}

func (e *Example) Original(name string) {
    e.Name = name
}

func main() {
    e := &Example{"Hello"}
    e.Original("Goodbye")
}
```

Puede reemplazar el método `Original` con `Refactor` con `gofmt` :

```
gofmt -r 'Original -> Refactor' -d main.go
```

Lo que producirá el diff:

```
-func (e *Example) Original(name string) {
+func (e *Example) Refactor(name string) {
     e.Name = name
 }

 func main() {
     e := &Example{"Hello"}
-    e.Original("Goodbye")
+    e.Refactor("Goodbye")
 }
```

Ir a buscar

`go get` descarga los paquetes nombrados por las rutas de importación, junto con sus

dependencias. A continuación, instala los paquetes nombrados, como 'instalar'. Get también acepta banderas de compilación para controlar la instalación.

> ve a github.com/maknahar/phonecountry

Al retirar un nuevo paquete, get crea el directorio de destino `$GOPATH/src/<import-path>` . Si el GOPATH contiene varias entradas, get usa la primera. Del mismo modo, instalará binarios compilados en `$GOPATH/bin` .

Al retirar o actualizar un paquete, busque una rama o etiqueta que coincida con la versión instalada localmente de Go. La regla más importante es que si la instalación local está ejecutando la versión "go1", obtenga búsquedas de una rama o etiqueta llamada "go1". Si no existe tal versión, recupera la versión más reciente del paquete.

Cuando se utiliza `go get` , el indicador `-d` hace que se descargue pero no se instala el paquete dado. La bandera `-u` le permitirá actualizar el paquete y sus dependencias.

Nunca obtenga verificaciones o actualizaciones de códigos almacenados en directorios de proveedores.

Ve env

`go env [var ...]` imprime go información del entorno.

> Por defecto imprime toda la información.

```
$go env
```

```
GOARCH="amd64"
GOBIN=""
GOEXE=""
GOHOSTARCH="amd64"
GOHOSTOS="darwin"
GOOS="darwin"
GOPATH="/Users/vikashkv/work"
GORACE=""
GOROOT="/usr/local/Cellar/go/1.7.4_1/libexec"
GOTOOLDIR="/usr/local/Cellar/go/1.7.4_1/libexec/pkg/tool/darwin_amd64"
CC="clang"
GOGCCFLAGS="-fPIC -m64 -pthread -fno-caret-diagnostics -Qunused-arguments -fmessage-length=0 -
fdebug-prefix-map=/var/folders/xf/t3j24fjd2b7bv8c9gdr_0mj80000gn/T/go-build785167995=/tmp/go-
build -gno-record-gcc-switches -fno-common"
CXX="clang++"
CGO_ENABLED="1"
```

> Si uno o más nombres de variables se dan como argumentos, imprime el valor de cada variable nombrada en su propia línea.

```
$go env GOOS GOPATH
```

```
darwin
/Users/vikashkv/work
```

Lea El comando go en línea: https://riptutorial.com/es/go/topic/4828/el-comando-go

Capítulo 29: Enchufar

Introducción

Go proporciona un mecanismo de complemento que se puede usar para vincular dinámicamente otro código Go en tiempo de ejecución.

A partir de Go 1.8, solo se puede usar en Linux.

Examples

Definir y usar un plugin.

```
package main

import "fmt"

var V int

func F() { fmt.Printf("Hello, number %d\n", V) }
```

Esto se puede construir con:

```
go build -buildmode=plugin
```

Y luego cargado y usado desde su aplicación:

```
p, err := plugin.Open("plugin_name.so")
if err != nil {
    panic(err)
}

v, err := p.Lookup("V")
if err != nil {
    panic(err)
}

f, err := p.Lookup("F")
if err != nil {
    panic(err)
}

*v.(*int) = 7
f.(func())() // prints "Hello, number 7"
```

Ejemplo de *The State of Go 2017* .

Lea Enchufar en línea: https://riptutorial.com/es/go/topic/9150/enchufar

Capítulo 30: Enviar / recibir correos electrónicos

Sintaxis

- Función PlainAuth (identidad, nombre de usuario, contraseña, cadena de host) Aut.
- func SendMail (cadena addr, un Auth, de cadena, a [] cadena, msg [] byte) error

Examples

Enviando correo electrónico con smtp.SendMail ()

Enviar correo electrónico es bastante simple en Go. Ayuda a comprender el RFC 822, que especifica el estilo en el que debe estar un correo electrónico, el código que aparece a continuación envía un correo electrónico compatible con RFC 822.

```
package main

import (
    "fmt"
    "net/smtp"
)

func main() {
    // user we are authorizing as
    from := "someuser@example.com"

    // use we are sending email to
    to := "otheruser@example.com"

    // server we are authorized to send email through
    host := "mail.example.com"

    // Create the authentication for the SendMail()
    // using PlainText, but other authentication methods are encouraged
    auth := smtp.PlainAuth("", from, "password", host)

    // NOTE: Using the backtick here ` works like a heredoc, which is why all the
    // rest of the lines are forced to the beginning of the line, otherwise the
    // formatting is wrong for the RFC 822 style
    message := `To: "Some User" <someuser@example.com>
From: "Other User" <otheruser@example.com>
Subject: Testing Email From Go!!

This is the message we are sending. That's it!
`

    if err := smtp.SendMail(host+":25", auth, from, []string{to}, []byte(message)); err != nil
{
        fmt.Println("Error SendMail: ", err)
        os.Exit(1)
    }
```

```
    fmt.Println("Email Sent!")
}
```

Lo anterior enviará un mensaje como el siguiente:

```
To: "Other User" <otheruser@example.com>
From: "Some User" <someuser@example.com>
Subject: Testing Email From Go!!

This is the message we are sending. That's it!
.
```

Lea Enviar / recibir correos electrónicos en línea: https://riptutorial.com/es/go/topic/5912/enviar---recibir-correos-electronicos

Capítulo 31: Estructuras

Introducción

Las estructuras son conjuntos de varias variables empaquetadas juntas. La estructura en sí misma es solo un *paquete que* contiene variables y que las hace fácilmente accesibles.

A diferencia de en C, las estructuras de Go pueden tener métodos adjuntos. También les permite implementar interfaces. Eso hace que las estructuras de Go sean similares a los objetos, pero faltan (probablemente intencionalmente) algunas características importantes conocidas en lenguajes orientados a objetos como la herencia.

Examples

Declaración Básica

Una estructura básica se declara de la siguiente manera:

```
type User struct {
    FirstName, LastName string
    Email               string
    Age                 int
}
```

Cada valor se llama un campo. Los campos generalmente se escriben uno por línea, con el nombre del campo precediendo a su tipo. Los campos consecutivos del mismo tipo se pueden combinar, como `FirstName` y `LastName` en el ejemplo anterior.

Campos exportados frente a no exportados (privado frente a público)

Los campos de Struct cuyos nombres comienzan con una letra mayúscula se exportan. Todos los demás nombres no son exportados.

```
type Account struct {
    UserID      int    // exported
    accessToken string // unexported
}
```

Solo se puede acceder a los campos no exportados por código dentro del mismo paquete. Como tal, si alguna vez accede a un campo de un paquete *diferente* , su nombre debe comenzar con una letra mayúscula.

```
package main

import "bank"

func main() {
```

```
    var x = &bank.Account{
        UserID: 1,          // this works fine
        accessToken: "one", // this does not work, since accessToken is unexported
    }
}
```

Sin embargo, desde el paquete del `bank` , puede acceder tanto a UserId como a accessToken sin problemas.

El `bank` paquetes podría implementarse así:

```
package bank

type Account struct {
    UserID int
    accessToken string
}

func ProcessUser(u *Account) {
    u.accessToken = doSomething(u) // ProcessUser() can access u.accessToken because
                                   // it's defined in the same package
}
```

Composición e incrustación

La composición proporciona una alternativa a la herencia. Una estructura puede incluir otro tipo por nombre en su declaración:

```
type Request struct {
    Resource string
}

type AuthenticatedRequest struct {
    Request
    Username, Password string
}
```

En el ejemplo anterior, `AuthenticatedRequest` contendrá cuatro miembros públicos: `Resource` , `Request` , `Username` y `Password` .

Las estructuras compuestas se pueden instanciar y utilizar de la misma manera que las estructuras normales:

```
func main() {
    ar := new(AuthenticatedRequest)
    ar.Resource = "example.com/request"
    ar.Username = "bob"
    ar.Password = "P@ssw0rd"
    fmt.Printf("%#v", ar)
}
```

jugar en el patio de recreo

Incrustación

En el ejemplo anterior, la `Request` es un campo incrustado. La composición también se puede lograr mediante la incrustación de un tipo diferente. Esto es útil, por ejemplo, para decorar un Struct con más funcionalidad. Por ejemplo, continuando con el ejemplo de Recursos, queremos una función que formatee el contenido del campo de Recursos para prefijarlo con `http://` o `https://` . Tenemos dos opciones: crear los nuevos métodos en AuthenticatedRequest o **incrustarlos** desde una estructura diferente:

```
type ResourceFormatter struct {}

func(r *ResourceFormatter) FormatHTTP(resource string) string {
    return fmt.Sprintf("http://%s", resource)
}
func(r *ResourceFormatter) FormatHTTPS(resource string) string {
    return fmt.Sprintf("https://%s", resource)
}


type AuthenticatedRequest struct {
    Request
    Username, Password string
    ResourceFormatter
}
```

Y ahora la función principal podría hacer lo siguiente:

```
func main() {
    ar := new(AuthenticatedRequest)
    ar.Resource = "www.example.com/request"
    ar.Username = "bob"
    ar.Password = "P@ssw0rd"

    println(ar.FormatHTTP(ar.Resource))
    println(ar.FormatHTTPS(ar.Resource))

    fmt.Printf("%#v", ar)
}
```

Observe que el `AuthenticatedRequest` que tiene una estructura incorporada `ResourceFormatter` .

Pero la desventaja es que no puedes acceder a objetos fuera de tu composición. Por lo tanto, `ResourceFormatter` no puede acceder a los miembros de `AuthenticatedRequest` .

jugar en el patio de recreo

Métodos

Los métodos de Struct son muy similares a las funciones:

```
type User struct {
    name string
```

```
}

func (u User) Name() string {
    return u.name
}

func (u *User) SetName(newName string) {
    u.name = newName
}
```

La única diferencia es la adición del método receptor. Se puede declarar como una instancia del tipo o un puntero a una instancia del tipo. Como `SetName()` muta la instancia, el receptor debe ser un puntero para realizar un cambio permanente en la instancia.

Por ejemplo:

```
package main

import "fmt"

type User struct {
    name string
}

func (u User) Name() string {
    return u.name
}

func (u *User) SetName(newName string) {
    u.name = newName
}

func main() {
    var me User

    me.SetName("Slim Shady")
    fmt.Println("My name is", me.Name())
}
```

Ir al patio de recreo

Estructura anónima

Es posible crear una estructura anónima:

```
data := struct {
    Number int
    Text   string
} {
    42,
    "Hello world!",
}
```

Ejemplo completo:

```
package main

import (
    "fmt"
)

func main() {
    data := struct {Number int; Text string}{42, "Hello world!"} // anonymous struct
    fmt.Printf("%+v\n", data)
}
```

jugar en el patio de recreo

Etiquetas

Los campos Struct pueden tener etiquetas asociadas con ellos. Estas etiquetas pueden ser leídas por el `reflect` paquete para obtener información personalizada especificada sobre un campo por el desarrollador.

```
struct Account {
    Username      string `json:"username"`
    DisplayName   string `json:"display_name"`
    FavoriteColor string `json:"favorite_color,omitempty"`
}
```

En el ejemplo anterior, las etiquetas se usan para cambiar los nombres de clave utilizados por el paquete `encoding/json` al calcular o desmarcar JSON.

Si bien la etiqueta puede tener cualquier valor de cadena, se considera una práctica recomendada usar los pares de `key:"value"` separados por espacios:

```
struct StructName {
    FieldName int `package1:"customdata,moredata" package2:"info"`
}
```

Las etiquetas de estructura utilizadas con el paquete `encoding/xml` y `encoding/json` se utilizan en todo el libarary estándar.

Haciendo copias struct.

Una estructura puede simplemente copiarse usando una asignación.

```
type T struct {
    I int
    S string
}

// initialize a struct
t := T{1, "one"}

// make struct copy
u := t // u has its field values equal to t
```

```
if u == t { // true
    fmt.Println("u and t are equal") // Prints: "u and t are equal"
}
```

En el caso anterior, `'t'` y 'u' ahora son objetos separados (valores de estructura).

Dado que `T` no contiene ningún tipo de referencia (segmentos, mapa, canales) ya que sus campos, `t` y `u` anteriores pueden modificarse sin afectarse entre sí.

```
fmt.Printf("t.I = %d, u.I = %d\n", t.I, u.I) // t.I = 100, u.I = 1
```

Sin embargo, si `T` contiene un tipo de referencia, por ejemplo:

```
type T struct {
    I  int
    S  string
    xs []int // a slice is a reference type
}
```

Luego, una copia por asignación simple copiaría también el valor del campo de tipo de sector en el nuevo objeto. Esto daría como resultado dos objetos diferentes que se refieren al mismo objeto de sector.

```
// initialize a struct
t := T{I: 1, S: "one", xs: []int{1, 2, 3}}

// make struct copy
u := t // u has its field values equal to t
```

Dado que tanto u como t hacen referencia a la misma división a través de su campo xs, la actualización de un valor en la división de un objeto reflejaría el cambio en la otra.

```
// update a slice field in u
u.xs[1] = 500

fmt.Printf("t.xs = %d, u.xs = %d\n", t.xs, u.xs)
// t.xs = [1 500 3], u.xs = [1 500 3]
```

Por lo tanto, se debe tener mucho cuidado para garantizar que esta propiedad de tipo de referencia no produzca un comportamiento involuntario.

Para copiar los objetos anteriores, por ejemplo, se podría realizar una copia explícita del campo de división:

```
// explicitly initialize u's slice field
u.xs = make([]int, len(t.xs))
// copy the slice values over from t
copy(u.xs, t.xs)

// updating slice value in u will not affect t
u.xs[1] = 500
```

```
fmt.Printf("t.xs = %d, u.xs = %d\n", t.xs, u.xs)
// t.xs = [1 2 3], u.xs = [1 500 3]
```

Literales de Struct

Un valor de un tipo de estructura se puede escribir utilizando un *literal de estructura* que especifica valores para sus campos.

```
type Point struct { X, Y int }
p := Point{1, 2}
```

El ejemplo anterior especifica cada campo en el orden correcto. Lo que no es útil, porque los programadores tienen que recordar los campos exactos en orden. Más a menudo, una estructura se puede inicializar enumerando algunos o todos los nombres de campo y sus valores correspondientes.

```
anim := gif.GIF{LoopCount: nframes}
```

Los campos omitidos se establecen en el valor cero para su tipo.

Nota: **Las dos formas no se pueden mezclar en el mismo literal.**

Estructura vacía

Una estructura es una secuencia de elementos con nombre, campos llamados, cada uno de los cuales tiene un nombre y un tipo. La estructura vacía no tiene campos, como esta estructura vacía anónima:

```
var s struct{}
```

O como este tipo de estructura vacía nombrada:

```
type T struct{}
```

Lo interesante de la estructura vacía es que, su tamaño es cero (prueba The Go Playground):

```
fmt.Println(unsafe.Sizeof(s))
```

Esto imprime `0` , por lo que la estructura vacía no toma memoria. así que es una buena opción para salir del canal, como (prueba The Go Playground):

```
package main

import (
    "fmt"
    "time"
)

func main() {
```

```
    done := make(chan struct{})
    go func() {
        time.Sleep(1 * time.Second)
        close(done)
    }()

    fmt.Println("Wait...")
    <-done
    fmt.Println("done.")
}
```

Lea Estructuras en línea: https://riptutorial.com/es/go/topic/374/estructuras

Capítulo 32: Expansión en línea

Observaciones

La expansión en línea es una optimización común en el código compilado que prioriza el rendimiento sobre el tamaño binario. Permite al compilador reemplazar una llamada de función con el cuerpo real de la función; efectivamente copiar / pegar código de un lugar a otro en tiempo de compilación. Dado que el sitio de llamada se expande para contener solo las instrucciones de la máquina que el compilador generó para la función, no tenemos que realizar una LLAMADA o EMPUJAR (el equivalente x86 de una declaración GOTO o un empuje de marco de pila) o su equivalente en otra arquitecturas

El inliner toma decisiones acerca de si debe o no incluir una función basada en una serie de heurísticas, pero en general, ir en línea de forma predeterminada. Debido a que el inliner se deshace de las llamadas a funciones, puede decidir de manera efectiva dónde se le permite al programador anticiparse a una goroutina.

Las llamadas a funciones no se incluirán si cualquiera de las siguientes afirmaciones es verdadera (también hay muchas otras razones, esta lista está incompleta):

- Las funciones son variad (por ejemplo, tienen `...` args)
- Las funciones tienen una "máxima pereza" mayor que el presupuesto (se repiten demasiado o no se pueden analizar por alguna otra razón)
- Contienen `panic` , se `recover` , o `defer`

Examples

Deshabilitando la expansión en línea

La expansión en línea se puede desactivar con el `go:noinline` pragma. Por ejemplo, si construimos el siguiente programa simple:

```
package main

func printhello() {
    println("Hello")
}

func main() {
    printhello()
}
```

obtenemos una salida que se ve así (recortada para facilitar la lectura):

```
$ go version
go version go1.6.2 linux/amd64
$ go build main.go
```

```
$ ./main
Hello
$ go tool objdump main
TEXT main.main(SB) /home/sam/main.go
        main.go:7       0x401000        64488b0c25f8ffffff      FS MOVQ FS:0xfffffff8, CX
        main.go:7       0x401009        483b6110                CMPQ 0x10(CX), SP
        main.go:7       0x40100d        7631                    JBE 0x401040
        main.go:7       0x40100f        4883ec10                SUBQ $0x10, SP
        main.go:8       0x401013        e8281f0200              CALL runtime.printlock(SB)
        main.go:8       0x401018        488d1d01130700          LEAQ 0x71301(IP), BX
        main.go:8       0x40101f        48891c24                MOVQ BX, 0(SP)
        main.go:8       0x401023        48c744240805000000      MOVQ $0x5, 0x8(SP)
        main.go:8       0x40102c        e81f290200              CALL runtime.printstring(SB)
        main.go:8       0x401031        e89a210200              CALL runtime.printnl(SB)
        main.go:8       0x401036        e8851f0200              CALL runtime.printunlock(SB)
        main.go:9       0x40103b        4883c410                ADDQ $0x10, SP
        main.go:9       0x40103f        c3                      RET
        main.go:7       0x401040        e87b9f0400              CALL
runtime.morestack_noctxt(SB)
        main.go:7       0x401045        ebb9                    JMP main.main(SB)
        main.go:7       0x401047        cc                      INT $0x3
        main.go:7       0x401048        cc                      INT $0x3
        main.go:7       0x401049        cc                      INT $0x3
        main.go:7       0x40104a        cc                      INT $0x3
        main.go:7       0x40104b        cc                      INT $0x3
        main.go:7       0x40104c        cc                      INT $0x3
        main.go:7       0x40104d        cc                      INT $0x3
        main.go:7       0x40104e        cc                      INT $0x3
        main.go:7       0x40104f        cc                      INT $0x3
…
```

Tenga en cuenta que no hay `CALL` al `printhello` . Sin embargo, si luego construimos el programa con el pragma en su lugar:

```
package main

//go:noinline
func printhello() {
    println("Hello")
}

func main() {
    printhello()
}
```

La salida contiene la función Printhello y un `CALL main.printhello` :

```
$ go version
go version go1.6.2 linux/amd64
$ go build main.go
$ ./main
Hello
$ go tool objdump main
TEXT main.printhello(SB) /home/sam/main.go
        main.go:4       0x401000        64488b0c25f8ffffff      FS MOVQ FS:0xfffffff8, CX
        main.go:4       0x401009        483b6110                CMPQ 0x10(CX), SP
        main.go:4       0x40100d        7631                    JBE 0x401040
        main.go:4       0x40100f        4883ec10                SUBQ $0x10, SP
```

```
        main.go:5       0x401013        e8481f0200              CALL runtime.printlock(SB)
        main.go:5       0x401018        488d1d01130700          LEAQ 0x71301(IP), BX
        main.go:5       0x40101f        48891c24                MOVQ BX, 0(SP)
        main.go:5       0x401023        48c744240805000000      MOVQ $0x5, 0x8(SP)
        main.go:5       0x40102c        e83f290200              CALL runtime.printstring(SB)
        main.go:5       0x401031        e8ba210200              CALL runtime.printnl(SB)
        main.go:5       0x401036        e8a51f0200              CALL runtime.printunlock(SB)
        main.go:6       0x40103b        4883c410                ADDQ $0x10, SP
        main.go:6       0x40103f        c3                      RET
        main.go:4       0x401040        e89b9f0400              CALL
runtime.morestack_noctxt(SB)
        main.go:4       0x401045        ebb9                    JMP main.printhello(SB)
        main.go:4       0x401047        cc                      INT $0x3
        main.go:4       0x401048        cc                      INT $0x3
        main.go:4       0x401049        cc                      INT $0x3
        main.go:4       0x40104a        cc                      INT $0x3
        main.go:4       0x40104b        cc                      INT $0x3
        main.go:4       0x40104c        cc                      INT $0x3
        main.go:4       0x40104d        cc                      INT $0x3
        main.go:4       0x40104e        cc                      INT $0x3
        main.go:4       0x40104f        cc                      INT $0x3

TEXT main.main(SB) /home/sam/main.go
        main.go:8       0x401050        64488b0c25f8ffffff      FS MOVQ FS:0xfffffff8, CX
        main.go:8       0x401059        483b6110                CMPQ 0x10(CX), SP
        main.go:8       0x40105d        7606                    JBE 0x401065
        main.go:9       0x40105f        e89cffffff              CALL main.printhello(SB)
        main.go:10      0x401064        c3                      RET
        main.go:8       0x401065        e8769f0400              CALL
runtime.morestack_noctxt(SB)
        main.go:8       0x40106a        ebe4                    JMP main.main(SB)
        main.go:8       0x40106c        cc                      INT $0x3
        main.go:8       0x40106d        cc                      INT $0x3
        main.go:8       0x40106e        cc                      INT $0x3
        main.go:8       0x40106f        cc                      INT $0x3
…
```

Lea Expansión en línea en línea: https://riptutorial.com/es/go/topic/2718/expansion-en-linea

Capítulo 33: Explotación florestal

Examples

Impresión básica

Go tiene una biblioteca de registro incorporada conocida como `log` con un método de uso común `Print` y sus variantes. Puedes importar la librería y luego hacer alguna impresión básica:

```
package main

import "log"

func main() {

    log.Println("Hello, world!")
    // Prints 'Hello, world!' on a single line

    log.Print("Hello, again! \n")
    // Prints 'Hello, again!' but doesn't break at the end without \n

    hello := "Hello, Stackers!"
    log.Printf("The type of hello is: %T \n", hello)
    // Allows you to use standard string formatting. Prints the type 'string' for %T
    // 'The type of hello is: string
}
```

Iniciar sesión para archivar

Es posible especificar el destino del registro con algo que establezca la interfaz io.Writer. Con eso podemos iniciar sesión en el archivo:

```
package main

import (
    "log"
    "os"
)

func main() {
    logfile, err := os.OpenFile("test.log", os.O_RDWR|os.O_CREATE|os.O_APPEND, 0666)
    if err != nil {
        log.Fatalln(err)
    }
    defer logfile.Close()

    log.SetOutput(logfile)
    log.Println("Log entry")
}
```

Salida:

```
$ cat test.log
2016/07/26 07:29:05 Log entry
```

Iniciar sesión en syslog

También es posible iniciar sesión en syslog con `log/syslog` esta manera:

```
package main

import (
    "log"
    "log/syslog"
)

func main() {
    syslogger, err := syslog.New(syslog.LOG_INFO, "syslog_example")
    if err != nil {
        log.Fatalln(err)
    }

    log.SetOutput(syslogger)
    log.Println("Log entry")
}
```

Después de ejecutar podremos ver esa línea en syslog:

```
Jul 26 07:35:21 localhost syslog_example[18358]: 2016/07/26 07:35:21 Log entry
```

Lea Explotación florestal en línea: https://riptutorial.com/es/go/topic/3724/explotacion-florestal

Capítulo 34: Fmt

Examples

Larguero

La interfaz `fmt.Stringer` requiere un solo método, `String() string` para ser satisfecha. El método de cadena define el formato de cadena "nativo" para ese valor, y es la representación predeterminada si el valor se proporciona a cualquiera de las `fmt` formateo o impresión de paquetes de `fmt` .

```
package main

import (
    "fmt"
)

type User struct {
    Name  string
    Email string
}

// String satisfies the fmt.Stringer interface for the User type
func (u User) String() string {
    return fmt.Sprintf("%s <%s>", u.Name, u.Email)
}

func main() {
    u := User{
        Name:  "John Doe",
        Email: "johndoe@example.com",
    }

    fmt.Println(u)
    // output: John Doe <johndoe@example.com>
}
```

Playground

Fundamento básico

El paquete fmt implementa E / S formateadas usando *verbos de* formato:

```
%v    // the value in a default format
%T    // a Go-syntax representation of the type of the value
%s    // the uninterpreted bytes of the string or slice
```

Funciones de formato

Hay **4** tipos de funciones principales en `fmt` y varias variaciones dentro.

Impresión

```
fmt.Print("Hello World")        // prints: Hello World
fmt.Println("Hello World")      // prints: Hello World\n
fmt.Printf("Hello %s", "World") // prints: Hello World
```

Sprint

```
formattedString := fmt.Sprintf("%v %s", 2, "words") // returns string "2 words"
```

Huella

```
byteCount, err := fmt.Fprint(w, "Hello World") // writes to io.Writer w
```

`Fprint` puede ser usado, dentro de los manejadores `http` :

```
func handler(w http.ResponseWriter, r *http.Request) {
    fmt.Fprintf(w, "Hello %s!", "Browser")
}   // Writes: "Hello Browser!" onto http response
```

Escanear

Escanear escanea texto leído desde la entrada estándar.

```
var s string
fmt.Scanln(&s) // pass pointer to buffer
// Scanln is similar to fmt.Scan(), but it stops scanning at new line.
fmt.Println(s) // whatever was inputted
```

Interfaz de largueros

Cualquier valor que tiene un `String()` método implementa la `fmt` **inteface** `Stringer`

```
type Stringer interface {
        String() string
}
```

Lea Fmt en línea: https://riptutorial.com/es/go/topic/2938/fmt

Capítulo 35: Funciones

Introducción

Las funciones en Go proporcionan un código organizado y reutilizable para realizar un conjunto de acciones. Las funciones simplifican el proceso de codificación, evitan la lógica redundante y hacen que el código sea más fácil de seguir. Este tema describe la declaración y la utilización de funciones, argumentos, parámetros, declaraciones de devolución y ámbitos en Go.

Sintaxis

- func () // tipo de función sin argumentos y sin valor de retorno
- func (x int) int // acepta entero y devuelve un entero
- func (a, b int, z float32) bool // acepta 2 enteros, un float y devuelve un booleano
- func (prefijo cadena, valores ... int) // función "variadic" que acepta una cadena y uno o más números enteros
- func () (int, bool) // función que devuelve dos valores
- func (a, b int, z float64, opt ... interface {}) (success bool) // acepta 2 enteros, uno flotante y uno o más números de interfaces y devuelve un valor booleano (que ya está inicializado dentro de la función))

Examples

Declaración Básica

Una función simple que no acepta ningún parámetro y no devuelve ningún valor:

```
func SayHello() {
    fmt.Println("Hello!")
}
```

Parámetros

Una función puede declarar opcionalmente un conjunto de parámetros:

```
func SayHelloToMe(firstName, lastName string, age int) {
    fmt.Printf("Hello, %s %s!\n", firstName, lastName)
    fmt.Printf("You are %d", age)
}
```

Observe que el tipo para `firstName` se omite porque es idéntica a `lastName` .

Valores de retorno

Una función puede devolver uno o más valores al llamante:

```
func AddAndMultiply(a, b int) (int, int) {
    return a+b, a*b
}
```

El segundo valor de retorno también puede ser el error var:

```
import errors

func Divide(dividend, divisor int) (int, error) {
    if divisor == 0 {
        return 0, errors.New("Division by zero forbidden")
    }
    return dividend / divisor, nil
}
```

Dos cosas importantes deben ser notadas:

- Los paréntesis pueden omitirse para un único valor de retorno.
- Cada declaración de `return` debe proporcionar un valor para **todos los** valores de retorno declarados.

Valores de retorno nombrados

Los valores de retorno se pueden asignar a una variable local. Una declaración de `return` vacía se puede usar para devolver sus valores actuales. Esto se conoce como retorno *"desnudo"* . Las declaraciones de devoluciones desnudas se deben usar solo en funciones cortas, ya que dañan la legibilidad en funciones más largas:

```
func Inverse(v float32) (reciprocal float32) {
    if v == 0 {
        return
    }
    reciprocal = 1 / v
    return
}
```

jugar en el patio de recreo

```
//A function can also return multiple values
func split(sum int) (x, y int) {
    x = sum * 4 / 9
    y = sum - x
    return
}
```

jugar en el patio de recreo

Dos cosas importantes deben ser notadas:

- Los paréntesis alrededor de los valores de retorno son **obligatorios** .
- Siempre se debe proporcionar una declaración de `return` vacía.

Funciones y cierres literales

Una simple función literal, imprimiendo `Hello!` al stdout

```
package main

import "fmt"

func main() {
    func(){
        fmt.Println("Hello!")
    }()
}
```

jugar en el patio de recreo

Una función literal, imprimiendo el argumento `str` a stdout:

```
package main

import "fmt"

func main() {
    func(str string) {
        fmt.Println(str)
    }("Hello!")
}
```

jugar en el patio de recreo

Una función literal, cerrando sobre la variable `str` :

```
package main

import "fmt"

func main() {
    str := "Hello!"
    func() {
        fmt.Println(str)
    }()
}
```

jugar en el patio de recreo

Es posible asignar una función literal a una variable:

```
package main

import (
    "fmt"
```

```
)

func main() {
    str := "Hello!"
    anon := func() {
        fmt.Println(str)
    }
    anon()
}
```

jugar en el patio de recreo

Funciones variables

Se puede llamar a una función variad con cualquier número de argumentos **finales** . Esos elementos se almacenan en una rebanada.

```
package main

import "fmt"

func variadic(strs ...string) {
     // strs is a slice of string
     for i, str := range strs {
         fmt.Printf("%d: %s\n", i, str)
     }
}

func main() {
     variadic("Hello", "Goodbye")
     variadic("Str1", "Str2", "Str3")
}
```

jugar en el patio de recreo

También puedes darle una porción a una función variad, con . . .

```
func main() {
     strs := []string {"Str1", "Str2", "Str3"}

     variadic(strs...)
}
```

jugar en el patio de recreo

Lea Funciones en línea: https://riptutorial.com/es/go/topic/373/funciones

Capítulo 36: Goroutines

Introducción

Un goroutine es un hilo ligero administrado por el tiempo de ejecución Go.

ir f (x, y, z)

comienza una nueva carrera de goroutine

f (x, y, z)

La evaluación de f, x, y, z ocurre en el goroutine actual y la ejecución de f ocurre en el nuevo goroutine.

Los goroutines se ejecutan en el mismo espacio de direcciones, por lo que el acceso a la memoria compartida debe estar sincronizado. El paquete de sincronización proporciona primitivas útiles, aunque no las necesitarás mucho en Go, ya que hay otras primitivas.

Referencia: https://tour.golang.org/concurrency/1

Examples

Programa Básico Goroutines

```
package main

import (
    "fmt"
    "time"
)

func say(s string) {
    for i := 0; i < 5; i++ {
        time.Sleep(100 * time.Millisecond)
        fmt.Println(s)
    }
}

func main() {
    go say("world")
    say("hello")
}
```

Un goroutine es una función que es capaz de ejecutarse simultáneamente con otras funciones. Para crear un goroutine usamos la palabra clave `go` seguido de una invocación de función:

```
package main

import "fmt"
```

```
func f(n int) {
  for i := 0; i < 10; i++ {
    fmt.Println(n, ":", i)
  }
}

func main() {
  go f(0)
  var input string
  fmt.Scanln(&input)
}
```

Generalmente, la llamada a la función ejecuta todas las declaraciones dentro del cuerpo de la función y regresa a la siguiente línea. Pero, con los goroutines, regresamos inmediatamente a la siguiente línea ya que no se espera a que se complete la función. Por lo tanto, se incluye una llamada a una función `Scanln` , de lo contrario, el programa se ha salido sin imprimir los números.

Lea Goroutines en línea: https://riptutorial.com/es/go/topic/9776/goroutines

Capítulo 37: Hora

Introducción

El paquete Go `time` proporciona funcionalidad para medir y mostrar el tiempo.

Este paquete proporciona una estructura de tiempo. `time.Time` , que permite almacenar y realizar cálculos en fechas y horas.

Sintaxis

- hora.Fecha (2016, hora.Diciembre, 31, 23, 59, 59, 999, hora.UTC) // inicializar
- date1 == date2 // devuelve `true` cuando los 2 son el mismo momento
- date1! = date2 // devuelve `true` cuando los 2 son momentos diferentes
- date1.Before (date2) // devuelve `true` cuando el primero es estrictamente anterior al segundo
- date1.After (date2) // devuelve `true` cuando el primero es estrictamente después del segundo

Examples

Tiempo de retorno. Tiempo cero valor cuando la función tiene un error

```
const timeFormat = "15 Monday January 2006"

func ParseDate(s string) (time.Time, error) {
    t, err := time.Parse(timeFormat, s)
    if err != nil {
        // time.Time{} returns January 1, year 1, 00:00:00.000000000 UTC
        // which according to the source code is the zero value for time.Time
        // https://golang.org/src/time/time.go#L23
        return time.Time{}, err
    }
    return t, nil
}
```

Análisis de tiempo

Si tiene una fecha almacenada como una cadena, deberá analizarla. Use `time.Parse` .

```
//           time.Parse(   format    , date to parse)
date, err := time.Parse("01/02/2006",  "04/08/2017")
if err != nil {
    panic(err)
}

fmt.Println(date)
// Prints 2017-04-08 00:00:00 +0000 UTC
```

El primer parámetro es el diseño en el que la cadena almacena la fecha y el segundo parámetro es la cadena que contiene la fecha. `01/02/2006` es lo mismo que decir que el formato es `MM/DD/YYYY` .

El diseño define el formato mostrando cómo se interpretaría el tiempo de referencia, que se definirá como `Mon Jan 2 15:04:05 -0700 MST 2006` si fuera el valor; Sirve como un ejemplo del formato de entrada. Entonces se hará la misma interpretación a la cadena de entrada.

Puede ver las constantes definidas en el paquete de tiempo para saber cómo escribir la cadena de diseño, pero tenga en cuenta que las constantes no se exportan y no se pueden usar fuera del paquete de tiempo.

```
const (
    stdLongMonth             // "January"
    stdMonth                 // "Jan"
    stdNumMonth              // "1"
    stdZeroMonth             // "01"
    stdLongWeekDay           // "Monday"
    stdWeekDay               // "Mon"
    stdDay                   // "2"
    stdUnderDay              // "_2"
    stdZeroDay               // "02"
    stdHour                  // "15"
    stdHour12                // "3"
    stdZeroHour12            // "03"
    stdMinute                // "4"
    stdZeroMinute            // "04"
    stdSecond                // "5"
    stdZeroSecond            // "05"
    stdLongYear              // "2006"
    stdYear                  // "06"
    stdPM                    // "PM"
    stdpm                    // "pm"
    stdTZ                    // "MST"
    stdISO8601TZ             // "Z0700"  // prints Z for UTC
    stdISO8601SecondsTZ      // "Z070000"
    stdISO8601ShortTZ        // "Z07"
    stdISO8601ColonTZ        // "Z07:00" // prints Z for UTC
    stdISO8601ColonSecondsTZ // "Z07:00:00"
    stdNumTZ                 // "-0700"  // always numeric
    stdNumSecondsTz          // "-070000"
    stdNumShortTZ            // "-07"    // always numeric
    stdNumColonTZ            // "-07:00" // always numeric
    stdNumColonSecondsTZ     // "-07:00:00"
)
```

Comparando el tiempo

En algún momento deberá saber, con objetos de 2 fechas, si hay correspondientes a la misma fecha, o encontrar qué fecha es posterior a la otra.

En **Go** , hay 4 maneras de comparar fechas:

- `date1 == date2` , devuelve `true` cuando los 2 son el mismo momento
- `date1 != date2` , devuelve `true` cuando los 2 son momentos diferentes

- `date1.Before(date2)` , devuelve `true` cuando el primero es estrictamente anterior al segundo
- `date1.After(date2)` , devuelve `true` cuando el primero es estrictamente después del segundo

  ADVERTENCIA: Cuando los 2 tiempos de comparación son iguales (o corresponden a la misma fecha exacta), las funciones `After` y `Before` devolverán `false` , ya que una fecha no es ni el antes ni el después de sí mismo.

  - `date1 == date1` , devuelve `true`
  - `date1 != date1` , devuelve `false`
  - `date1.After(date1)` , devuelve `false`
  - `date1.Before(date1)` , devuelve `false`

  CONSEJOS: Si necesita saber si una fecha es anterior o igual a otra, solo necesita combinar los 4 operadores

  - `date1 == date2 && date1.After(date2)` , devuelve `true` cuando date1 es posterior o igual a date2
    o utilizando `! (date1.Before(date2))`
  - `date1 == date2 && date1.Before(date2)` , devuelve `true` cuando date1 es anterior o igual a date2 o usando `!(date1.After(date2))`

Algunos ejemplos para ver cómo usar:

```
// Init 2 dates for example
var date1 = time.Date(2009, time.November, 10, 23, 0, 0, 0, time.UTC)
var date2 = time.Date(2017, time.July, 25, 16, 22, 42, 123, time.UTC)
var date3 = time.Date(2017, time.July, 25, 16, 22, 42, 123, time.UTC)

bool1 := date1.Before(date2) // true, because date1 is before date2
bool2 := date1.After(date2) // false, because date1 is not after date2

bool3 := date2.Before(date1) // false, because date2 is not before date1
bool4 := date2.After(date1) // true, because date2 is after date1

bool5 := date1 == date2 // false, not the same moment
bool6 := date1 == date3 // true, different objects but representing the exact same time

bool7 := date1 != date2 // true, different moments
bool8 := date1 != date3 // false, not different moments

bool9 := date1.After(date3) // false, because date1 is not after date3 (that are the same)
bool10:= date1.Before(date3) // false, because date1 is not before date3 (that are the same)

bool11 := !(date1.Before(date3)) // true, because date1 is not before date3
bool12 := !(date1.After(date3)) // true, because date1 is not after date3
```

Lea Hora en línea: https://riptutorial.com/es/go/topic/8860/hora

Capítulo 38: Imágenes

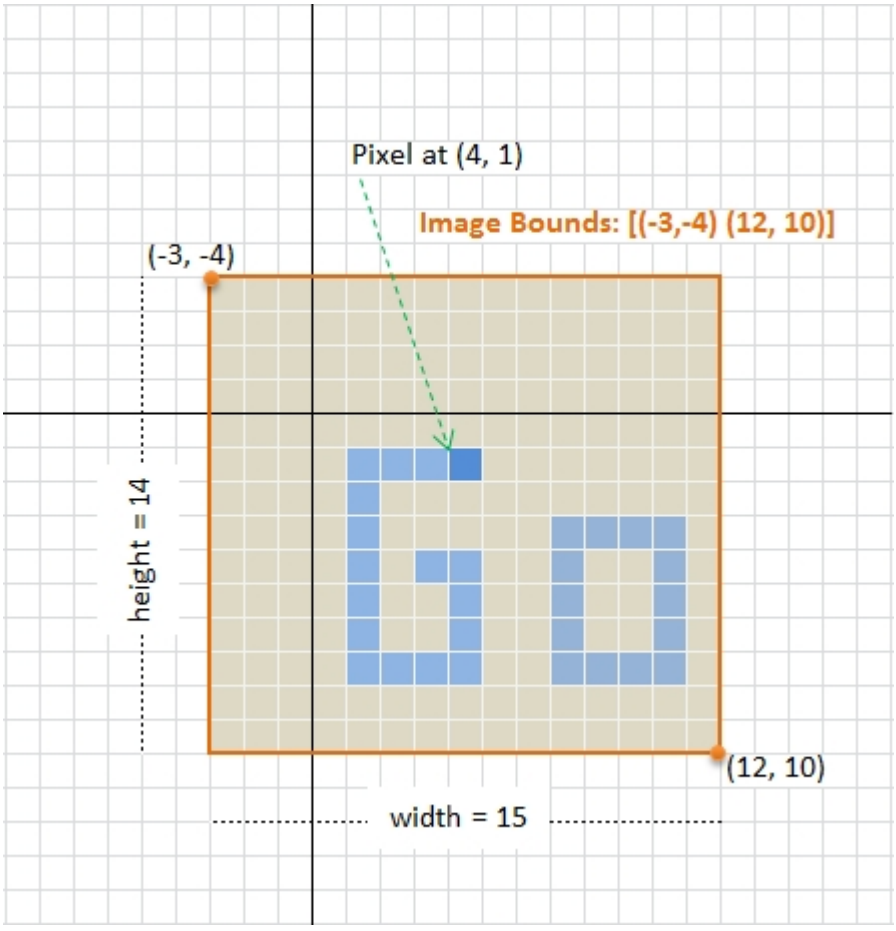
Introducción

El paquete de imágenes proporciona funcionalidades básicas para trabajar con imágenes 2-D. Este tema describe varias operaciones básicas cuando se trabaja con imágenes, como leer y escribir un formato de imagen particular, recortar, acceder y modificar *píxeles* , conversión de color, cambio de tamaño y filtrado de imágenes básico.

Examples

Conceptos básicos

Una imagen representa una cuadrícula rectangular de elementos de imagen (*píxel*). En el paquete de imágenes , el píxel se representa como uno de los colores definidos en el paquete de imágenes / colores . La geometría 2-D de la imagen se representa como `image.Rectangle` , mientras que `image.Point` denota una posición en la cuadrícula.



La figura anterior ilustra los conceptos básicos de una imagen en el paquete. Una imagen de tamaño 15x14 píxeles tiene *límites* rectangulares iniciados en *la* esquina *superior izquierda* (por ejemplo, coordenada (-3, -4) en la figura anterior), y sus ejes aumentan a la derecha y hacia abajo a *la* esquina *inferior derecha* (por ejemplo, coordenadas (12, 10) en la figura). Tenga en cuenta

que los límites **no necesariamente comienzan o contienen el punto (0,0)** .

Imagen relacionada con el *tipo*

En `Go` , una imagen siempre implementa la siguiente `image.Image` Interfaz de imagen

```
type Image interface {
    // ColorModel returns the Image's color model.
    ColorModel() color.Model
    // Bounds returns the domain for which At can return non-zero color.
    // The bounds do not necessarily contain the point (0, 0).
    Bounds() Rectangle
    // At returns the color of the pixel at (x, y).
    // At(Bounds().Min.X, Bounds().Min.Y) returns the upper-left pixel of the grid.
    // At(Bounds().Max.X-1, Bounds().Max.Y-1) returns the lower-right one.
    At(x, y int) color.Color
}
```

en el que la interfaz `color.Color` se define como

```
type Color interface {
    // RGBA returns the alpha-premultiplied red, green, blue and alpha values
    // for the color. Each value ranges within [0, 0xffff], but is represented
    // by a uint32 so that multiplying by a blend factor up to 0xffff will not
    // overflow.
    //
    // An alpha-premultiplied color component c has been scaled by alpha (a),
    // so has valid values 0 <= c <= a.
    RGBA() (r, g, b, a uint32)
}
```

y `color.Model` es una interfaz declarada como

```
type Model interface {
    Convert(c Color) Color
}
```

Accediendo a la dimensión de la imagen y píxel.

Supongamos que tenemos una imagen almacenada como `img` variable, luego podemos obtener la dimensión y el píxel de la imagen de la siguiente manera:

```
// Image bounds and dimension
b := img.Bounds()
width, height := b.Dx(), b.Dy()
// do something with dimension ...

// Corner co-ordinates
top := b.Min.Y
left := b.Min.X
bottom := b.Max.Y
right := b.Max.X
```

```
// Accessing pixel. The (x,y) position must be
// started from (left, top) position not (0,0)
for y := top; y < bottom; y++ {
    for x := left; x < right; x++ {
        cl := img.At(x, y)
        r, g, b, a := cl.RGBA()
        // do something with r,g,b,a color component
    }
}
```

Tenga en cuenta que en el paquete, el valor de cada componente `R,G,B,A` encuentra entre `0-65535` (`0x0000 - 0xffff`), **no entre `0-255` .**

Cargando y guardando imagen

En la memoria, una imagen puede verse como una matriz de píxeles (color). Sin embargo, cuando una imagen se almacena en un almacenamiento permanente, puede almacenarse tal como está (formato RAW), Bitmap u otros formatos de imagen con un algoritmo de compresión particular para ahorrar espacio de almacenamiento, por ejemplo, PNG, JPEG, GIF, etc. Al cargar una imagen con un formato particular, la imagen se debe *decodificar* a imagen. `image.Image` con el algoritmo correspondiente. Una función `image.Decode` declarada como

```
func Decode(r io.Reader) (Image, string, error)
```

Se proporciona para este uso particular. Para poder manejar varios formatos de imagen, antes de llamar a la función `image.Decode` , el decodificador debe registrarse a través de `image.RegisterFormat` función definida como

```
func RegisterFormat(name, magic string,
    decode func(io.Reader) (Image, error), decodeConfig func(io.Reader) (Config, error))
```

Actualmente, el paquete de imágenes admite tres formatos de archivo: JPEG , GIF y PNG . Para registrar un decodificador, agregue lo siguiente

```
import _ "image/jpeg"    //register JPEG decoder
```

al paquete `main` la aplicación. En algún lugar de su código (no es necesario en el paquete `main`), para cargar una imagen JPEG, use los siguientes fragmentos:

```
f, err := os.Open("inputimage.jpg")
if err != nil {
    // Handle error
}
defer f.Close()

img, fmtName, err := image.Decode(f)
if err != nil {
    // Handle error
}

// `fmtName` contains the name used during format registration
```

```
// Work with `img` ...
```

Guardar en PNG

Para guardar una imagen en un formato particular, el *codificador* correspondiente debe importarse explícitamente, es decir,

```
import "image/png"    //needed to use `png` encoder
```

luego se puede guardar una imagen con los siguientes fragmentos de código:

```
f, err := os.Create("outimage.png")
if err != nil {
    // Handle error
}
defer f.Close()

// Encode to `PNG` with `DefaultCompression` level
// then save to file
err = png.Encode(f, img)
if err != nil {
    // Handle error
}
```

Si desea especificar un nivel de compresión distinto del nivel de compresión `DefaultCompression` , cree un *codificador* , por ejemplo

```
enc := png.Encoder{
    CompressionLevel: png.BestSpeed,
}
err := enc.Encode(f, img)
```

Guardar en JPEG

Para guardar en formato `jpeg` , usa lo siguiente:

```
import "image/jpeg"

// Somewhere in the same package
f, err := os.Create("outimage.jpg")
if err != nil {
    // Handle error
}
defer f.Close()

// Specify the quality, between 0-100
// Higher is better
opt := jpeg.Options{
    Quality: 90,
}
err = jpeg.Encode(f, img, &opt)
if err != nil {
```

```
    // Handle error
}
```

Guardar en GIF

Para guardar la imagen en un archivo GIF, use los siguientes fragmentos.

```
import "image/gif"

// Samewhere in the same package
f, err := os.Create("outimage.gif")
if err != nil {
    // Handle error
}
defer f.Close()

opt := gif.Options {
    NumColors: 256,
    // Add more parameters as needed
}

err = gif.Encode(f, img, &opt)
if err != nil {
    // Handle error
}
```

Recortar imagen

La mayor parte del tipo de imagen en el paquete de imagen tiene un `SubImage(r Rectangle) Image` imagen de `SubImage(r Rectangle) Image` , excepto `image.Uniform` . En base a este hecho, podemos implementar una función para recortar una imagen arbitraria de la siguiente manera

```
func CropImage(img image.Image, cropRect image.Rectangle) (cropImg image.Image, newImg bool) {
    //Interface for asserting whether `img`
    //implements SubImage or not.
    //This can be defined globally.
    type CropableImage interface {
        image.Image
        SubImage(r image.Rectangle) image.Image
    }

    if p, ok := img.(CropableImage); ok {
        // Call SubImage. This should be fast,
        // since SubImage (usually) shares underlying pixel.
        cropImg = p.SubImage(cropRect)
    } else if cropRect = cropRect.Intersect(img.Bounds()); !cropRect.Empty() {
        // If `img` does not implement `SubImage`,
        // copy (and silently convert) the image portion to RGBA image.
        rgbaImg := image.NewRGBA(cropRect)
        for y := cropRect.Min.Y; y < cropRect.Max.Y; y++ {
            for x := cropRect.Min.X; x < cropRect.Max.X; x++ {
                rgbaImg.Set(x, y, img.At(x, y))
            }
        }
        cropImg = rgbaImg
        newImg = true
```

```
    } else {
        // Return an empty RGBA image
        cropImg = &image.RGBA{}
        newImg = true
    }

    return cropImg, newImg
}
```

Tenga en cuenta que la imagen recortada puede compartir sus píxeles subyacentes con la imagen original. Si este es el caso, cualquier modificación de la imagen recortada afectará a la imagen original.

Convertir imagen en color a escala de grises

Algún algoritmo de procesamiento de imágenes digitales, como la detección de bordes, la información transportada por la intensidad de la imagen (es decir, el valor en escala de grises) es suficiente. El uso de información de color (`R, G, B` canal `R, G, B`) puede proporcionar un resultado ligeramente mejor, pero la complejidad del algoritmo aumentará. Por lo tanto, en este caso, debemos convertir la imagen en color a una imagen en escala de grises antes de aplicar dicho algoritmo.

El siguiente código es un ejemplo de conversión de una imagen arbitraria a una imagen en escala de grises de 8 bits. La imagen se recupera de una ubicación remota mediante `net/http` paquete `net/http` , se convierte a escala de grises y finalmente se guarda como imagen PNG.

```
package main

import (
    "image"
    "log"
    "net/http"
    "os"

    _ "image/jpeg"
    "image/png"
)

func main() {
    // Load image from remote through http
    // The Go gopher was designed by Renee French. (http://reneefrench.blogspot.com/)
    // Images are available under the Creative Commons 3.0 Attributions license.
    resp, err := http.Get("http://golang.org/doc/gopher/fiveyears.jpg")
    if err != nil {
        // handle error
        log.Fatal(err)
    }
    defer resp.Body.Close()

    // Decode image to JPEG
    img, _, err := image.Decode(resp.Body)
    if err != nil {
        // handle error
        log.Fatal(err)
    }
```

```
    log.Printf("Image type: %T", img)

    // Converting image to grayscale
    grayImg := image.NewGray(img.Bounds())
    for y := img.Bounds().Min.Y; y < img.Bounds().Max.Y; y++ {
        for x := img.Bounds().Min.X; x < img.Bounds().Max.X; x++ {
            grayImg.Set(x, y, img.At(x, y))
        }
    }

    // Working with grayscale image, e.g. convert to png
    f, err := os.Create("fiveyears_gray.png")
    if err != nil {
        // handle error
        log.Fatal(err)
    }
    defer f.Close()

    if err := png.Encode(f, grayImg); err != nil {
        log.Fatal(err)
    }
}
```

La conversión de color se produce cuando se asigna un píxel a través de `Set(x, y int, c color.Color)` que se implementa en `image.go` como

```
func (p *Gray) Set(x, y int, c color.Color) {
    if !(Point{x, y}.In(p.Rect)) {
        return
    }

    i := p.PixOffset(x, y)
    p.Pix[i] = color.GrayModel.Convert(c).(color.Gray).Y
}
```

en el cual, `color.GrayModel` se define en `color.go` como

```
func grayModel(c Color) Color {
    if _, ok := c.(Gray); ok {
        return c
    }
    r, g, b, _ := c.RGBA()

    // These coefficients (the fractions 0.299, 0.587 and 0.114) are the same
    // as those given by the JFIF specification and used by func RGBToYCbCr in
    // ycbcr.go.
    //
    // Note that 19595 + 38470 + 7471 equals 65536.
    //
    // The 24 is 16 + 8. The 16 is the same as used in RGBToYCbCr. The 8 is
    // because the return value is 8 bit color, not 16 bit color.
    y := (19595*r + 38470*g + 7471*b + 1<<15) >> 24

    return Gray{uint8(y)}
}
```

Sobre la base de los hechos anteriores, la intensidad `Y` se calcula con la siguiente fórmula:

Luminancia: Y = 0.299 **R** + 0.587 **G** + 0.114 **B**

Si queremos aplicar diferentes fórmulas / algoritmos para convertir un color en una intensidad, por ejemplo

Media: Y = (**R** + **G** + **B**) / 3
Luma: Y = 0.2126 **R** + 0.7152 **G** + 0.0722 **B**
Brillo: Y = (min (**R** , **G** , **B**) + max (**R** , **G** , **B**)) / 2

entonces, los siguientes fragmentos pueden ser utilizados.

```
// Converting image to grayscale
grayImg := image.NewGray(img.Bounds())
for y := img.Bounds().Min.Y; y < img.Bounds().Max.Y; y++ {
    for x := img.Bounds().Min.X; x < img.Bounds().Max.X; x++ {
        R, G, B, _ := img.At(x, y).RGBA()
        //Luma: Y = 0.2126*R + 0.7152*G + 0.0722*B
        Y := (0.2126*float64(R) + 0.7152*float64(G) + 0.0722*float64(B)) * (255.0 / 65535)
        grayPix := color.Gray{uint8(Y)}
        grayImg.Set(x, y, grayPix)
    }
}
```

El cálculo anterior se realiza mediante la multiplicación de punto flotante, y ciertamente no es el más eficiente, pero es suficiente para demostrar la idea. El otro punto es, cuando se llama a `Set(x, y int, c color.Color)` con `color.Gray` como tercer argumento, el modelo de color no realizará la conversión de color como se puede ver en la función `grayModel` anterior.

Lea Imágenes en línea: https://riptutorial.com/es/go/topic/10557/imagenes

Capítulo 39: Instalación

Examples

Instalar en Linux o Ubuntu

```
$ sudo apt-get update
$ sudo apt-get install -y build-essential git curl wget
$ wget https://storage.googleapis.com/golang/go<versions>.gz
```

Puedes encontrar las listas de versiones aquí .

```
# To install go1.7 use
$ wget https://storage.googleapis.com/golang/go1.7.linux-amd64.tar.gz


# Untar the file
$ sudo tar -C /usr/local -xzf go1.7.linux-amd64.tar.gz
$ sudo chown -R $USER:$USER /usr/local/go
$ rm go1.5.4.linux-amd64.tar.gz
```

Actualizar `$GOPATH`

```
$ mkdir $HOME/go
```

Agregue las siguientes dos líneas al final del archivo ~ / .bashrc

```
export GOPATH=$HOME/go
export PATH=$GOPATH/bin:/usr/local/go/bin:$PATH
```

```
$ nano ~/.bashrc
  export GOPATH=$HOME/go
  export PATH=$GOPATH/bin:/usr/local/go/bin:$PATH

$ source ~/.bashrc
```

Ahora están listos para ir, prueba tu versión de go usando:

```
$ go version
go version go<version> linux/amd64
```

Lea Instalación en línea: https://riptutorial.com/es/go/topic/5776/instalacion

Capítulo 40: Instalación

Observaciones

Descargando go

Visite la Lista de descargas y encuentre el archivo correcto para su sistema operativo. Los nombres de estas descargas pueden ser un tanto crípticos para los nuevos usuarios.

Los nombres están en el formato go [versión]. [Sistema operativo] - [arquitectura]. [Archivo]

Para la versión, desea elegir la más reciente disponible. Estas deben ser las primeras opciones que veas.

Para el sistema operativo, esto se explica por sí mismo, excepto para los usuarios de Mac, donde el sistema operativo se llama "darwin". Esto lleva el nombre de la parte de código abierto del sistema operativo utilizado por las computadoras Mac .

Si está ejecutando una máquina de 64 bits (que es la más común en las computadoras modernas), la parte de "arquitectura" del nombre del archivo debe ser "amd64". Para máquinas de 32 bits, será "386". Si estás en un dispositivo ARM como una Raspberry Pi, querrás "armv6l".

Para la parte de "archivo", los usuarios de Mac y Windows tienen dos opciones porque Go proporciona instaladores para esas plataformas. Para Mac, probablemente quieras "pkg". Para Windows, probablemente quieras "msi".

Entonces, por ejemplo, si estoy en una máquina Windows de 64 bits y quiero descargar Go 1.6.3, la descarga que quiero se llamará:

```
go1.6.3.windows-amd64.msi
```

Extraer los archivos descargados.

Ahora que tenemos un archivo Go descargado, necesitamos extraerlo en algún lugar.

Mac y Windows

Dado que se proporcionan instaladores para estas plataformas, la instalación es fácil. Simplemente ejecute el instalador y acepte los valores predeterminados.

Linux

No hay un instalador para Linux, por lo que se requiere más trabajo. Deberías haber descargado un archivo con el sufijo ".tar.gz". Este es un archivo comprimido, similar a un archivo ".zip".

Necesitamos extraerlo. Extraeremos los archivos de Go a `/usr/local` porque es la ubicación recomendada.

Abra una terminal y cambie los directorios al lugar donde descargó el archivo. Esto es probablemente en `Downloads` . Si no, reemplace el directorio en el siguiente comando adecuadamente.

```
cd Downloads
```

Ahora, ejecute lo siguiente para extraer el archivo en `/usr/local` , reemplazando `[filename]` con el nombre del archivo que descargó.

```
tar -C /usr/local -xzf [filename].tar.gz
```

Configuración de variables de entorno

Hay un paso más por delante antes de que estés listo para comenzar a desarrollar. Necesitamos establecer variables de entorno, que es información que los usuarios pueden cambiar para que los programas tengan una mejor idea de la configuración del usuario.

Windows

Necesitas configurar `GOPATH` , que es la carpeta en la que estarás haciendo el trabajo de Go.

Puede configurar las variables de entorno a través del botón "Variables de entorno" en la pestaña "Avanzadas" del panel de control "Sistema". Algunas versiones de Windows proporcionan este panel de control a través de la opción "Configuración avanzada del sistema" dentro del panel de control "Sistema".

El nombre de su nueva variable de entorno debe ser "GOPATH". El valor debe ser la ruta completa a un directorio en el que desarrollará el código Go. Una carpeta llamada "go" en su directorio de usuarios es una buena opción.

Mac

Necesitas configurar `GOPATH` , que es la carpeta en la que estarás haciendo el trabajo de Go.

Edite un archivo de texto llamado ".bash_profile", que debería estar en su directorio de usuarios, y agregue la siguiente nueva línea al final, reemplazando `[work area]` con una ruta completa a un directorio en el que le gustaría trabajar. Vaya ".bash_profile" no existe, créelo. Una carpeta llamada "go" en su directorio de usuarios es una buena opción.

```
export GOPATH=[work area]
```

Linux

Como Linux no tiene un instalador, requiere un poco más de trabajo. Necesitamos mostrar el

terminal donde se encuentran el compilador Go y otras herramientas, y debemos configurar `GOPATH` , que es una carpeta en la que estarás trabajando con Go.

Edite un archivo de texto llamado ".profile", que debería estar en su directorio de usuarios, y agregue la siguiente línea al final, reemplazando `[work area]` con una ruta completa a un directorio en el que le gustaría trabajar. Vaya ". .profile "no existe, créelo. Una carpeta llamada "go" en su directorio de usuarios es una buena opción.

Luego, en otra línea nueva, agregue lo siguiente a su archivo ".profile".

```
export PATH=$PATH:/usr/local/go/bin
```

¡Terminado!

Si las herramientas de Go aún no están disponibles en el terminal, intente cerrar esa ventana y abrir una nueva ventana de terminal.

Examples

Ejemplo .profile o .bash_profile

```
# This is an example of a .profile or .bash_profile for Linux and Mac systems
export GOPATH=/home/user/go
export PATH=$PATH:/usr/local/go/bin
```

Lea Instalación en línea: https://riptutorial.com/es/go/topic/6213/instalacion

Capítulo 41: Interfaces

Observaciones

Las interfaces en Go son solo conjuntos de métodos fijos. Un tipo implementa *implícitamente* una interfaz si su conjunto de métodos es un superconjunto de la interfaz. *No hay declaración de intenciones.*

Examples

Interfaz simple

En Go, una interfaz es solo un conjunto de métodos. Usamos una interfaz para especificar un comportamiento de un objeto dado.

```
type Painter interface {
    Paint()
}
```

El tipo de implementación **no necesita** declarar que está implementando la interfaz. Basta con definir métodos de la misma firma.

```
type Rembrandt struct{}

func (r Rembrandt) Paint() {
    // use a lot of canvas here
}
```

Ahora podemos usar la estructura como interfaz.

```
var p Painter
p = Rembrandt{}
```

Una interfaz puede ser satisfecha (o implementada) por un número arbitrario de tipos. También un tipo puede implementar un número arbitrario de interfaces.

```
type Singer interface {
     Sing()
}

type Writer interface {
     Write()
}

type Human struct{}

func (h *Human) Sing() {
    fmt.Println("singing")
}
```

```
func (h *Human) Write() {
    fmt.Println("writing")
}



type OnlySinger struct{}
func (o *OnlySinger) Sing() {
    fmt.Println("singing")
}
```

Aquí, The `Human` struct satisface tanto la interfaz de `Singer` como la de `Writer` , pero la estructura `OnlySinger` solo satisface la interfaz de `Singer` .

Interfaz vacía

Hay un tipo de interfaz vacío, que no contiene métodos. Lo declaramos como `interface{}` . Esto no contiene métodos por lo que cada `type` satisface. Por lo tanto, la interfaz vacía puede contener cualquier valor de tipo.

```
var a interface{}
var i int = 5
s := "Hello world"

type StructType struct {
    i, j int
    k string
}


// all are valid statements
a = i
a = s
a = &StructType{1, 2, "hello"}
```

El caso de uso más común para las interfaces es asegurar que una variable admita uno o más comportamientos. Por el contrario, el caso de uso principal de la interfaz vacía es definir una variable que pueda contener cualquier valor, independientemente de su tipo concreto.

Para recuperar estos valores como sus tipos originales, solo tenemos que hacer

```
i = a.(int)
s = a.(string)
m := a.(*StructType)
```

o

```
i, ok := a.(int)
s, ok := a.(string)
m, ok := a.(*StructType)
```

`ok` indica si la `interface a` es convertible a un tipo dado. Si no es posible lanzar `ok` , será `false` .

Valores de interfaz

Si declara una variable de una interfaz, puede almacenar cualquier tipo de valor que implemente los métodos declarados por la interfaz.

Si declaramos `h` de la `interface Singer` , puede almacenar un valor de tipo `Human` o `OnlySinger`. Esto se debe a que todos ellos implementan los métodos especificados por la interfaz de `Singer` .

```
var h Singer
h = &human{}

h.Sing()
```

Determinación del tipo subyacente desde la interfaz

A veces puede ser útil saber qué tipo subyacente se ha pasado. Esto se puede hacer con un interruptor de tipo. Esto supone que tenemos dos estructuras:

```
type Rembrandt struct{}

func (r Rembrandt) Paint() {}

type Picasso struct{}

func (r Picasso) Paint() {}
```

Eso implementa la interfaz de Painter:

```
type Painter interface {
    Paint()
}
```

Luego podemos usar este interruptor para determinar el tipo subyacente:

```
func WhichPainter(painter Painter) {
    switch painter.(type) {
    case Rembrandt:
        fmt.Println("The underlying type is Rembrandt")
    case Picasso:
        fmt.Println("The underlying type is Picasso")
    default:
        fmt.Println("Unknown type")
    }
}
```

Verificación en tiempo de compilación si un tipo satisface una interfaz

Las interfaces e implementaciones (tipos que implementan una interfaz) están "desconectadas". Por lo tanto, es una pregunta legítima cómo verificar en tiempo de compilación si un tipo implementa una interfaz.

Una forma de pedirle al compilador que compruebe que el tipo `T` implementa la interfaz `I` es

intentar una asignación utilizando el valor cero para `T` o el puntero a `T` , según corresponda. Y podemos elegir asignar al identificador en blanco para evitar la basura innecesaria:

```
type T struct{}

var _ I = T{}       // Verify that T implements I.
var _ I = (*T)(nil) // Verify that *T implements I.
```

Si `T` o `*T` no implementa `I` , será un error de tiempo de compilación.

Esta pregunta también aparece en las preguntas frecuentes oficiales: ¿Cómo puedo garantizar que mi tipo satisfaga una interfaz?

Tipo de interruptor

Los interruptores de tipo también se pueden usar para obtener una variable que coincida con el tipo de caso:

```
func convint(v interface{}) (int,error) {
    switch u := v.(type) {
    case int:
        return u, nil
    case float64:
        return int(u), nil
    case string:
        return strconv(u)
    default:
        return 0, errors.New("Unsupported type")
    }
}
```

Aserción de tipo

Puede acceder al tipo de datos real de la interfaz con Type Assertion.

```
interfaceVariable.(DataType)
```

Ejemplo de estructura `MyType` que implementa la interfaz `Subber` :

```
package main

import (
    "fmt"
)

type Subber interface {
    Sub(a, b int) int
}

type MyType struct {
    Msg string
}
```

```
//Implement method Sub(a,b int) int
func (m *MyType) Sub(a, b int) int {
    m.Msg = "SUB!!!"

    return a - b;
}

func main() {
    var interfaceVar Subber = &MyType{}
    fmt.Println(interfaceVar.Sub(6,5))
    fmt.Println(interfaceVar.(*MyType).Msg)
}
```

Sin `.(*MyType)` no podríamos acceder a `Msg` Field. Si probamos `interfaceVar.Msg` mostrará un error de compilación:

```
interfaceVar.Msg undefined (type Subber has no field or method Msg)
```

Ir interfaces de un aspecto matemático

En matemáticas, especialmente la *teoría de conjuntos* , tenemos una colección de cosas que se denomina *conjunto* y nombramos esas cosas como *elementos* . Mostramos un conjunto con su nombre como A, B, C, ... o explícitamente poniendo su miembro en notación de refuerzo: {a, b, c, d, e}. Supongamos que tenemos un elemento arbitrario xy un conjunto Z, la pregunta clave es: "¿Cómo podemos entender que x es miembro de Z o no?". El matemático responde a esta pregunta con un concepto: **Característica característica** de un conjunto. *La propiedad característica* de un conjunto es una expresión que describe el conjunto completamente. Por ejemplo, tenemos un conjunto de *números naturales* que es {0, 1, 2, 3, 4, 5, ...}. Podemos describir este conjunto con esta expresión: $\{a_n \mid a_0 = 0, a_n = a_{n-1} + 1\}$. En la última expresión $a_0 = 0, a_n = a_{n-1} + 1$ es la propiedad característica del conjunto de números naturales. **Si tenemos esta expresión, podemos construir este conjunto completamente** . Vamos a describir el conjunto de *números pares* de esta manera. Sabemos que este conjunto está formado por estos números: {0, 2, 4, 6, 8, 10, ...}. Con una mirada entendemos que todos estos números son también un *número natural* , en otras palabras, *si agregamos algunas condiciones adicionales a la propiedad característica de los números naturales, podemos construir una nueva expresión que describa este conjunto* . Entonces podemos describir con esta expresión: {n | n es un miembro de números naturales *y* el recordatorio de n en 2 es cero}. Ahora podemos crear un filtro que obtenga la propiedad característica de un conjunto y filtre algunos elementos deseados para devolver elementos de nuestro conjunto. Por ejemplo, si tenemos un filtro de números naturales, tanto los números naturales como los números pares pueden pasar este filtro, pero si tenemos un filtro de números pares, algunos elementos como 3 y 137871 no pueden pasar el filtro.

La definición de interfaz en Go es como definir la propiedad característica y el mecanismo de uso de interfaz como argumento de una función es como un filtro que detecta que el elemento es miembro de nuestro conjunto deseado o no. Permite describir este aspecto con código:

```
type Number interface {
    IsNumber() bool // the implementation filter "meysam" from 3.14, 2 and 3
}
```

```
type NaturalNumber interface {
    Number
    IsNaturalNumber() bool // the implementation filter 3.14 from 2 and 3
}

type EvenNumber interface {
    NaturalNumber
    IsEvenNumber() bool // the implementation filter 3 from 2
}
```

La propiedad característica de `Number` es todas las estructuras que tienen el método `IsNumber` , para `NaturalNumber` son todas las que tienen los métodos `IsNumber` e `IsNaturalNumber` y, finalmente, para `EvenNumber` son todos los tipos que tienen los `IsNumber` , `IsNaturalNumber` e `IsEvenNumber` . Gracias a esta interpretación de la interfaz, podemos entender fácilmente que, dado que la `interface{}` no tiene ninguna propiedad característica, acepta todos los tipos (porque no tiene ningún filtro para distinguir los valores).

Lea Interfaces en línea: https://riptutorial.com/es/go/topic/1221/interfaces

Capítulo 42: Iota

Introducción

Iota proporciona una forma de declarar constantes numéricas a partir de un valor inicial que crece monótonamente. Iota se puede usar para declarar máscaras de bits que se usan a menudo en la programación de redes y sistemas y otras listas de constantes con valores relacionados.

Observaciones

El identificador `iota` se utiliza para asignar valores a listas de constantes. Cuando se usa iota en una lista, comienza con un valor de cero, y se incrementa en uno para cada valor en la lista de constantes y se restablece en cada palabra clave `const` . A diferencia de las enumeraciones de otros idiomas, iota se puede usar en expresiones (por ejemplo, `iota + 1`) que permite una mayor flexibilidad.

Examples

Uso simple de iota

Para crear una lista de constantes, asigne un valor `iota` a cada elemento:

```
const (
  a = iota // a = 0
  b = iota // b = 1
  c = iota // c = 2
)
```

Para crear una lista de constantes de forma abreviada, asigne un valor de `iota` al primer elemento:

```
const (
  a = iota // a = 0
  b        // b = 1
  c        // c = 2
)
```

Usando iota en una expresión

`iota` se puede usar en expresiones, por lo que también se puede usar para asignar valores que no sean simples enteros incrementales a partir de cero. Para crear constantes para unidades SI, use este ejemplo de Effective Go :

```
type ByteSize float64

const (
```

```
    _           = iota // ignore first value by assigning to blank identifier
    KB ByteSize = 1 << (10 * iota)
    MB
    GB
    TB
    PB
    EB
    ZB
    YB
)
```

Valores de salto

El valor de `iota` aún se incrementa para cada entrada en una lista constante, incluso si no se usa iota:

```
const ( // iota is reset to 0
    a = 1 << iota  // a == 1
    b = 1 << iota  // b == 2
    c = 3          // c == 3  (iota is not used but still incremented)
    d = 1 << iota  // d == 8
)
```

también se incrementará incluso si no se crea una constante, lo que significa que el identificador vacío se puede usar para omitir valores por completo:

```
const (
  a = iota // a = 0
  _        // iota is incremented
  b        // b = 2
)
```

El primer bloque de código se tomó de Go Spec (CC-BY 3.0).

Uso de iota en una lista de expresiones.

Debido a que `iota` se incrementa después de cada `ConstSpec` , los valores dentro de la misma lista de expresiones tendrán el mismo valor para `iota` :

```
const (
    bit0, mask0 = 1 << iota, 1<<iota - 1  // bit0 == 1, mask0 == 0
    bit1, mask1                           // bit1 == 2, mask1 == 1
    _, _                                  // skips iota == 2
    bit3, mask3                           // bit3 == 8, mask3 == 7
)
```

Este ejemplo fue tomado de Go Spec (CC-BY 3.0).

Uso de iota en una máscara de bits.

Iota puede ser muy útil al crear una máscara de bits. Por ejemplo, para representar el estado de una conexión de red que puede ser segura, autenticada y / o lista, podríamos crear una máscara

de bits como la siguiente:

```
const (
    Secure = 1 << iota // 0b001
    Authn              // 0b010
    Ready              // 0b100
)

ConnState := Secure|Authn // 0b011: Connection is secure and authenticated, but not yet Ready
```

Uso de iota en const.

Esta es una enumeración para la creación de const. Ir compilador inicia iota desde 0 e incrementa en uno para cada constante siguiente. El valor se determina en tiempo de compilación en lugar de tiempo de ejecución. Debido a esto, no podemos aplicar iota a las expresiones que se evalúan en tiempo de ejecución.

Programa para usar iota en const.

```
package main

import "fmt"

const (
    Low = 5 * iota
    Medium
    High
)

func main() {
    // Use our iota constants.
    fmt.Println(Low)
    fmt.Println(Medium)
    fmt.Println(High)
}
```

Pruébalo en Go Playground

Lea Iota en línea: https://riptutorial.com/es/go/topic/2865/iota

Capítulo 43: JSON

Sintaxis

- Func Marshal (v interfaz {}) ([] byte, error)
- func Unmarshal (data [] byte, v interface {}) error

Observaciones

El paquete `"encoding/json"` Package json implementa la codificación y decodificación de objetos JSON en `Go` .

Los tipos en JSON junto con sus tipos concretos correspondientes en Go son:

Tipo JSON	Ir tipo concreto
booleano	bool
números	float64 o int
cuerda	cuerda
nulo	nulo

Examples

Codificación JSON básica

`json.Marshal` del paquete `"encoding/json"` codifica un valor para JSON.

El parámetro es el valor a codificar. Los valores devueltos son una matriz de bytes que representan la entrada codificada JSON (en caso de éxito), y un error (en caso de error).

```
decodedValue := []string{"foo", "bar"}

// encode the value
data, err := json.Marshal(decodedValue)

// check if the encoding is successful
if err != nil {
    panic(err)
}

// print out the JSON-encoded string
// remember that data is a []byte
fmt.Println(string(data))
// "["foo","bar"]"
```

Patio de recreo

Aquí hay algunos ejemplos básicos de codificación para tipos de datos incorporados:

```
var data []byte

data, _ = json.Marshal(1)
fmt.Println(string(data))
// 1

data, _ = json.Marshal("1")
fmt.Println(string(data))
// "1"

data, _ = json.Marshal(true)
fmt.Println(string(data))
// true

data, _ = json.Marshal(map[string]int{"London": 18, "Rome": 30})
fmt.Println(string(data))
// {"London":18,"Rome":30}
```

Patio de recreo

Codificar variables simples es útil para entender cómo funciona la codificación JSON en Go. Sin embargo, en el mundo real, es probable que codifique datos más complejos almacenados en estructuras .

Decodificación JSON básica

`json.Unmarshal` del paquete `"encoding/json"` decodifica un valor JSON en el valor señalado por la variable dada.

Los parámetros son el valor para decodificar en `[]bytes` y una variable para usar como almacenamiento para el valor sin serializar. El valor devuelto es un error (en caso de fallo).

```
encodedValue := []byte(`{"London":18,"Rome":30}`)

// generic storage for the decoded JSON
var data map[string]interface{}

// decode the value into data
// notice that we must pass the pointer to data using &data
err := json.Unmarshal(encodedValue, &data)

// check if the decoding is successful
if err != nil {
    panic(err)
}

fmt.Println(data)
map[London:18 Rome:30]
```

Patio de recreo

Observe cómo en el ejemplo anterior sabíamos de antemano tanto el tipo de clave como el valor. Pero este no es siempre el caso. De hecho, en la mayoría de los casos, el JSON contiene tipos de valores mixtos.

```
encodedValue := []byte(`{"city":"Rome","temperature":30}`)

// generic storage for the decoded JSON
var data map[string]interface{}

// decode the value into data
if err := json.Unmarshal(encodedValue, &data); err != nil {
    panic(err)
}

// if you want to use a specific value type, we need to cast it
temp := data["temperature"].(float64)
fmt.Println(temp) // 30
city := data["city"].(string)
fmt.Println(city) // "Rome"
```

Patio de recreo

En el último ejemplo anterior, usamos un mapa genérico para almacenar el valor decodificado. Debemos usar una `map[string]interface{}` porque sabemos que las claves son cadenas, pero no sabemos el tipo de sus valores de antemano.

Este es un enfoque muy simple, pero también es extremadamente limitado. En el mundo real, generalmente decodificaría un JSON en un tipo de `struct` definido a la medida .

Decodificación de datos JSON de un archivo

Los datos JSON también se pueden leer desde archivos.

Supongamos que tenemos un archivo llamado `data.json` con el siguiente contenido:

```
[
    {
      "Name" : "John Doe",
      "Standard" : 4
    },
    {
      "Name" : "Peter Parker",
      "Standard" : 11
    },
    {
      "Name" : "Bilbo Baggins",
      "Standard" : 150
    }
]
```

El siguiente ejemplo lee el archivo y decodifica el contenido:

```
package main
```

```
import (
    "encoding/json"
    "fmt"
    "log"
    "os"
)

type Student struct {
    Name     string
    Standard int `json:"Standard"`
}

func main() {
    // open the file pointer
    studentFile, err := os.Open("data.json")
    if err != nil {
        log.Fatal(err)
    }
    defer studentFile.Close()

    // create a new decoder
    var studentDecoder *json.Decoder = json.NewDecoder(studentFile)
    if err != nil {
        log.Fatal(err)
    }

    // initialize the storage for the decoded data
    var studentList []Student

    // decode the data
    err = studentDecoder.Decode(&studentList)
    if err != nil {
        log.Fatal(err)
    }

    for i, student := range studentList {
        fmt.Println("Student", i+1)
        fmt.Println("Student name:", student.Name)
        fmt.Println("Student standard:", student.Standard)
    }
}
```

El archivo `data.json` debe estar en el mismo directorio del programa ejecutable Go. Lea la documentación de E / S de Go File para obtener más información sobre cómo trabajar con archivos en Go.

Usando estructuras anónimas para decodificar

El objetivo de usar estructuras anónimas es descodificar solo la información que nos importa sin ensuciar nuestra aplicación con tipos que se usan solo en una sola función.

```
jsonBlob := []byte(`
  {
    "_total": 1,
    "_links": {
      "self":
"https://api.twitch.tv/kraken/channels/foo/subscriptions?direction=ASC&limit=25&offset=0",
      "next":
```

```
"https://api.twitch.tv/kraken/channels/foo/subscriptions?direction=ASC&limit=25&offset=25"
    },
    "subscriptions": [
      {
        "created_at": "2011-11-23T02:53:17Z",
        "_id": "abcdef0000000000000000000000000000000000",
        "_links": {
          "self": "https://api.twitch.tv/kraken/channels/foo/subscriptions/bar"
        },
        "user": {
          "display_name": "bar",
          "_id": 123456,
          "name": "bar",
          "staff": false,
          "created_at": "2011-06-16T18:23:11Z",
          "updated_at": "2014-10-23T02:20:51Z",
          "logo": null,
          "_links": {
            "self": "https://api.twitch.tv/kraken/users/bar"
          }
        }
      }
    ]
  }
`)

var js struct {
    Total int `json:"_total"`
    Links struct {
        Next string `json:"next"`
    } `json:"_links"`
    Subs []struct {
        Created string `json:"created_at"`
        User    struct {
            Name string `json:"name"`
            ID   int    `json:"_id"`
        } `json:"user"`
    } `json:"subscriptions"`
}

err := json.Unmarshal(jsonBlob, &js)
if err != nil {
    fmt.Println("error:", err)
}
fmt.Printf("%+v", js)
```

Salida: `{Total:1`
`Links:{Next:https://api.twitch.tv/kraken/channels/foo/subscriptions?direction=ASC&limit=25&offset=25}`
`Subs:[{Created:2011-11-23T02:53:17Z User:{Name:bar ID:123456}}]}`

Patio de recreo

Para el caso general, consulte también:
http://stackoverflow.com/documentation/go/994/json/4111/encoding-decoding-go-structs

Configurando campos de estructura JSON

Considere el siguiente ejemplo:

```
type Company struct {
    Name     string
    Location string
}
```

Ocultar / Omitir ciertos campos

Para exportar `Revenue` y `Sales` , pero ocultarlos de la codificación / decodificación, use `json:"-"` o cambie el nombre de la variable para comenzar con una letra minúscula. Tenga en cuenta que esto evita que la variable sea visible fuera del paquete.

```
type Company struct {
    Name     string `json:"name"`
    Location string `json:"location"`
    Revenue  int    `json:"-"`
    sales    int
}
```

Ignorar los campos vacíos

Para evitar que la `Location` se incluya en el JSON cuando se establece en su valor cero, agregue `,omitempty` a la etiqueta `json` .

```
type Company struct {
    Name     string `json:"name"`
    Location string `json:"location,omitempty"`
}
```

Ejemplo en Playground

Estructuras de cálculo con campos privados.

Como buen desarrollador, ha creado la siguiente estructura con campos exportados y no exportados:

```
type MyStruct struct {
    uuid string
    Name string
}
```

Ejemplo en Playground: https://play.golang.org/p/Zk94Il2ANZ

Ahora desea convertir a `Marshal()` esta estructura en JSON válido para almacenamiento en algo así como etcd. Sin embargo, dado que `uuid` no se exporta, el `json.Marshal()` omite. ¿Qué hacer? Utilice una estructura anónima y la interfaz `json.MarshalJSON()` . Aquí hay un ejemplo:

```
type MyStruct struct {
    uuid string
    Name string
}
```

```
func (m MyStruct) MarshalJSON() ([]byte, error {
    j, err := json.Marshal(struct {
        Uuid string
        Name string
    }{
        Uuid: m.uuid,
        Name: m.Name,
    })
    if err != nil {
            return nil, err
    }
    return j, nil
}
```

Ejemplo en Playground: https://play.golang.org/p/Bv2k9GgbzE

Codificación / Decodificación utilizando Go structs

Supongamos que tenemos la siguiente `struct` que define un tipo de `City` :

```
type City struct {
    Name string
    Temperature int
}
```

Podemos codificar / decodificar los valores de la ciudad usando el paquete `encoding/json` .

En primer lugar, necesitamos usar los metadatos de Go para indicar al codificador la correspondencia entre los campos de estructura y las claves JSON.

```
type City struct {
    Name string `json:"name"`
    Temperature int `json:"temp"`
    // IMPORTANT: only exported fields will be encoded/decoded
    // Any field starting with a lower letter will be ignored
}
```

Para mantener este ejemplo simple, declararemos una correspondencia explícita entre los campos y las claves. Sin embargo, puede usar varias variantes de los metadatos de `json:` como se explica en los documentos .

IMPORTANTE: Solo los campos exportados (campos con nombre de capital) serán serializados / deserializados. Por ejemplo, si el nombre del campo ***t** emperatura* será ignorado, incluso si se establece el `json` metadatos.

Codificación

Para codificar una estructura de `City` , use `json.Marshal` como en el ejemplo básico:

```
// data to encode
city := City{Name: "Rome", Temperature: 30}
```

```
// encode the data
bytes, err := json.Marshal(city)
if err != nil {
    panic(err)
}

fmt.Println(string(bytes))
// {"name":"Rome","temp":30}
```

Patio de recreo

Descodificación

Para decodificar una estructura de `City` , use `json.Unmarshal` como en el ejemplo básico:

```
// data to decode
bytes := []byte(`{"name":"Rome","temp":30}`)

// initialize the container for the decoded data
var city City

// decode the data
// notice the use of &city to pass the pointer to city
if err := json.Unmarshal(bytes, &city); err != nil {
    panic(err)
}

fmt.Println(city)
// {Rome 30}
```

Patio de recreo

Lea JSON en línea: https://riptutorial.com/es/go/topic/994/json

Capítulo 44: Lectores

Examples

Usando bytes.Lector para leer desde una cadena

Una implementación de la interfaz `io.Reader` se puede encontrar en el paquete de `bytes` . Permite utilizar un segmento de bytes como fuente de un lector. En este ejemplo, el segmento de bytes se toma de una cadena, pero es más probable que se haya leído desde un archivo o conexión de red.

```
message := []byte("Hello, playground")

reader := bytes.NewReader(message)

bs := make([]byte, 5)
n, err := reader.Read(bs)
if err != nil {
    log.Fatal(err)
}

fmt.Printf("Read %d bytes: %s", n, bs)
```

Ir al patio de recreo

Lea Lectores en línea: https://riptutorial.com/es/go/topic/7000/lectores

Capítulo 45: Los canales

Introducción

Un canal contiene valores de un tipo dado. Los valores se pueden escribir en un canal y leer desde él, y circulan dentro del canal en orden de primero en entrar, primero en salir. Hay una distinción entre los canales almacenados en búfer, que pueden contener varios mensajes, y los canales no almacenados, que no pueden. Los canales se usan normalmente para comunicarse entre goroutines, pero también son útiles en otras circunstancias.

Sintaxis

- make (chan int) // crea un canal sin búfer
- make (chan int, 5) // crea un canal con buffer con una capacidad de 5
- cerrar (ch) // cierra un canal "ch"
- ch <- 1 // escribe el valor de 1 en un canal "ch"
- val: = <-ch // lee un valor del canal "ch"
- val, ok: = <-ch // sintaxis alternativa; ok es un bool que indica si el canal está cerrado

Observaciones

Un canal que contiene la estructura vacía `make(chan struct{})` es un mensaje claro para el usuario de que no se transmite información a través del canal y que se utiliza exclusivamente para la sincronización.

Con respecto a los canales no almacenados, la escritura de un canal se bloqueará hasta que se produzca la lectura correspondiente de otro goroutine. Lo mismo es cierto para un bloqueo de lectura de canal mientras se espera un escritor.

Examples

Utilizando rango

Al leer varios valores de un canal, el uso de `range` es un patrón común:

```
func foo() chan int {
    ch := make(chan int)

    go func() {
        ch <- 1
        ch <- 2
        ch <- 3
        close(ch)

    }()
```

```
    return ch
}

func main() {
    for n := range foo() {
        fmt.Println(n)
    }

    fmt.Println("channel is now closed")
}
```

Patio de recreo

Salida

```
1
2
3
channel is now closed
```

Tiempos de espera

Los canales se utilizan a menudo para implementar tiempos de espera.

```
func main() {
    // Create a buffered channel to prevent a goroutine leak. The buffer
    // ensures that the goroutine below can eventually terminate, even if
    // the timeout is met. Without the buffer, the send on the channel
    // blocks forever, waiting for a read that will never happen, and the
    // goroutine is leaked.
    ch := make(chan struct{}, 1)

    go func() {
        time.Sleep(10 * time.Second)
        ch <- struct{}{}
    }()

    select {
    case <-ch:
        // Work completed before timeout.
    case <-time.After(1 * time.Second):
        // Work was not completed after 1 second.
    }
}
```

Coordinadores goroutines

Imagine un goroutine con un proceso de dos pasos, donde el hilo principal debe hacer algo de trabajo entre cada paso:

```
func main() {
    ch := make(chan struct{})
    go func() {
        // Wait for main thread's signal to begin step one
        <-ch
```

```
        // Perform work
        time.Sleep(1 * time.Second)

        // Signal to main thread that step one has completed
        ch <- struct{}{}

        // Wait for main thread's signal to begin step two
        <-ch

        // Perform work
        time.Sleep(1 * time.Second)

        // Signal to main thread that work has completed
        ch <- struct{}{}
    }()

    // Notify goroutine that step one can begin
    ch <- struct{}{}

    // Wait for notification from goroutine that step one has completed
    <-ch

    // Perform some work before we notify
    // the goroutine that step two can begin
    time.Sleep(1 * time.Second)

    // Notify goroutine that step two can begin
    ch <- struct{}{}

    // Wait for notification from goroutine that step two has completed
    <-ch
}
```

Buffer vs vs no buffer

```
func bufferedUnbufferedExample(buffered bool) {
    // We'll declare the channel, and we'll make it buffered or
    // unbuffered depending on the parameter `buffered` passed
    // to this function.
    var ch chan int
    if buffered {
        ch = make(chan int, 3)
    } else {
        ch = make(chan int)
    }

    // We'll start a goroutine, which will emulate a webserver
    // receiving tasks to do every 25ms.
    go func() {
        for i := 0; i < 7; i++ {
            // If the channel is buffered, then while there's an empty
            // "slot" in the channel, sending to it will not be a
            // blocking operation. If the channel is full, however, we'll
            // have to wait until a "slot" frees up.
            // If the channel is unbuffered, sending will block until
            // there's a receiver ready to take the value. This is great
            // for goroutine synchronization, not so much for queueing
            // tasks for instance in a webserver, as the request will
```

```
            // hang until the worker is ready to take our task.
            fmt.Println(">", "Sending", i, "...")
            ch <- i
            fmt.Println(">", i, "sent!")
            time.Sleep(25 * time.Millisecond)
        }
        // We'll close the channel, so that the range over channel
        // below can terminate.
        close(ch)
    }()

    for i := range ch {
        // For each task sent on the channel, we would perform some
        // task. In this case, we will assume the job is to
        // "sleep 100ms".
        fmt.Println("<", i, "received, performing 100ms job")
        time.Sleep(100 * time.Millisecond)
        fmt.Println("<", i, "job done")
    }
}
```

ir al patio

Bloqueo y desbloqueo de canales.

Por defecto, la comunicación a través de los canales está sincronizada; Cuando envías algún valor debe haber un receptor. De lo contrario, obtendrás un `fatal error: all goroutines are asleep - deadlock!` como sigue:

```
package main

import "fmt"

func main() {
    msg := make(chan string)
    msg <- "Hey There"
    go func() {
        fmt.Println(<-msg)
    }()
}
```

Pero hay una solución de uso: usar canales en búfer:

```
package main

import "fmt"
import "time"

func main() {
    msg :=make(chan string, 1)
    msg <- "Hey There!"
    go func() {
        fmt.Println(<-msg)
    }()
    time.Sleep(time.Second * 1)
}
```

Esperando que el trabajo termine

Una técnica común para usar canales es crear una cantidad de trabajadores (o consumidores) para leer desde el canal. Usar un sync.WaitGroup es una manera fácil de esperar a que esos trabajadores terminen de correr.

```
package main

import (
    "fmt"
    "sync"
    "time"
)

func main() {
    numPiecesOfWork := 20
    numWorkers := 5

    workCh := make(chan int)
    wg := &sync.WaitGroup{}

    // Start workers
    wg.Add(numWorkers)
    for i := 0; i < numWorkers; i++ {
        go worker(workCh, wg)
    }

    // Send work
    for i := 0; i < numPiecesOfWork; i++ {
        work := i % 10 // invent some work
        workCh <- work
    }

    // Tell workers that no more work is coming
    close(workCh)

    // Wait for workers to finish
    wg.Wait()

    fmt.Println("done")
}

func worker(workCh <-chan int, wg *sync.WaitGroup) {
    defer wg.Done() // will call wg.Done() right before returning

    for work := range workCh { // will wait for work until workCh is closed
        doWork(work)
    }
}

func doWork(work int) {
    time.Sleep(time.Duration(work) * time.Millisecond)
    fmt.Println("slept for", work, "milliseconds")
}
```

Lea Los canales en línea: https://riptutorial.com/es/go/topic/1263/los-canales

Capítulo 46: Manejo de errores

Introducción

En Go, las situaciones inesperadas se manejan utilizando **errores** , no excepciones. Este enfoque es más similar al de C, usando errno, que al de Java u otros lenguajes orientados a objetos, con sus bloques try / catch. Sin embargo, un error no es un entero sino una interfaz.

Una función que puede fallar normalmente devuelve un **error** como último valor de retorno. Si este error no es **nulo** , algo salió mal y la persona que llama a la función debería actuar en consecuencia.

Observaciones

Observe cómo en Go no *genera* un error. En su lugar, *devuelve* un error en caso de fallo.

Si una función puede fallar, el último valor devuelto es generalmente un tipo de `error` .

```
// This method doesn't fail
func DoSomethingSafe() {
}

// This method can fail
func DoSomething() (error) {
}

// This method can fail and, when it succeeds,
// it returns a string.
func DoAndReturnSomething() (string, error) {
}
```

Examples

Creando un valor de error

La forma más sencilla de crear un error es mediante el uso del paquete de `errors` .

```
errors.New("this is an error")
```

Si desea agregar información adicional a un error, el paquete `fmt` también proporciona un método útil de creación de errores:

```
var f float64
fmt.Errorf("error with some additional information: %g", f)
```

Aquí hay un ejemplo completo, donde el error es devuelto desde una función:

```
package main

import (
    "errors"
    "fmt"
)

var ErrThreeNotFound = errors.New("error 3 is not found")

func main() {
    fmt.Println(DoSomething(1)) // succeeds! returns nil
    fmt.Println(DoSomething(2)) // returns a specific error message
    fmt.Println(DoSomething(3)) // returns an error variable
    fmt.Println(DoSomething(4)) // returns a simple error message
}

func DoSomething(someID int) error {
    switch someID {
    case 3:
        return ErrThreeNotFound
    case 2:
        return fmt.Errorf("this is an error with extra info: %d", someID)
    case 1:
        return nil
    }

    return errors.New("this is an error")
}
```

Abrir en el patio

Creando un tipo de error personalizado

En Go, un error está representado por cualquier valor que pueda describirse como una cadena. Cualquier tipo que implemente la interfaz de `error` incorporada es un error.

```
// The error interface is represented by a single
// Error() method, that returns a string representation of the error
type error interface {
    Error() string
}
```

El siguiente ejemplo muestra cómo definir un nuevo tipo de error usando un literal compuesto de cadena.

```
// Define AuthorizationError as composite literal
type AuthorizationError string

// Implement the error interface
// In this case, I simply return the underlying string
func (e AuthorizationError) Error() string {
    return string(e)
}
```

Ahora puedo usar mi tipo de error personalizado como error:

```
package main

import (
    "fmt"
)

// Define AuthorizationError as composite literal
type AuthorizationError string

// Implement the error interface
// In this case, I simply return the underlying string
func (e AuthorizationError) Error() string {
    return string(e)
}

func main() {
    fmt.Println(DoSomething(1)) // succeeds! returns nil
    fmt.Println(DoSomething(2)) // returns an error message
}

func DoSomething(someID int) error {
    if someID != 1 {
        return AuthorizationError("Action not allowed!")
    }

    // do something here

    // return a nil error if the execution succeeded
    return nil
}
```

Devolviendo un error

En Go no *generas* un error. En su lugar, *devuelve* un `error` en caso de fallo.

```
// This method can fail
func DoSomething() error {
    // functionThatReportsOK is a side-effecting function that reports its
    // state as a boolean. NOTE: this is not a good practice, so this example
    // turns the boolean value into an error. Normally, you'd rewrite this
    // function if it is under your control.
    if ok := functionThatReportsOK(); !ok {
        return errors.New("functionThatReportsSuccess returned a non-ok state")
    }

    // The method succeeded. You still have to return an error
    // to properly obey to the method signature.
    // But in this case you return a nil error.
    return nil
}
```

Si el método devuelve varios valores (y la ejecución puede fallar), entonces la convención estándar es devolver el error como el último argumento.

```
// This method can fail and, when it succeeds,
// it returns a string.
func DoAndReturnSomething() (string, error) {
```

```
    if os.Getenv("ERROR") == "1" {
        return "", errors.New("The method failed")
    }

    s := "Success!"

    // The method succeeded.
    return s, nil
}

result, err := DoAndReturnSomething()
if err != nil {
    panic(err)
}
```

Manejando un error

En Go los errores pueden devolverse desde una llamada de función. La convención es que si un método puede fallar, el último argumento devuelto es un `error` .

```
func DoAndReturnSomething() (string, error) {
    if os.Getenv("ERROR") == "1" {
        return "", errors.New("The method failed")
    }

    // The method succeeded.
    return "Success!", nil
}
```

Utiliza múltiples asignaciones de variables para verificar si el método falló.

```
result, err := DoAndReturnSomething()
if err != nil {
    panic(err)
}

// This is executed only if the method didn't return an error
fmt.Println(result)
```

Si no está interesado en el error, simplemente puede ignorarlo asignándolo a _ .

```
result, _ := DoAndReturnSomething()
fmt.Println(result)
```

Por supuesto, ignorar un error puede tener serias implicaciones. Por lo tanto, esto generalmente no se recomienda.

Si tiene varias llamadas a métodos, y uno o más métodos en la cadena pueden devolver un error, debe propagar el error al primer nivel que puede manejarlo.

```
func Foo() error {
    return errors.New("I failed!")
}
```

```
func Bar() (string, error) {
    err := Foo()
    if err != nil {
        return "", err
    }

    return "I succeeded", nil
}

func Baz() (string, string, error) {
    res, err := Bar()
    if err != nil {
        return "", "", err
    }

    return "Foo", "Bar", nil
}
```

Recuperándose del pánico

Un error común es declarar un sector y comenzar a solicitar índices sin inicializarlo, lo que lleva a un pánico de "índice fuera de rango". El siguiente código explica cómo recuperarse del pánico sin salir del programa, que es el comportamiento normal de un pánico. En la mayoría de las situaciones, devolver un error de esta manera en lugar de salir del programa en una situación de pánico solo es útil para fines de desarrollo o prueba.

```
type Foo struct {
    Is []int
}

func main() {
    fp := &Foo{}
    if err := fp.Panic(); err != nil {
        fmt.Printf("Error: %v", err)
    }
    fmt.Println("ok")
}

func (fp *Foo) Panic() (err error) {
    defer PanicRecovery(&err)
    fp.Is[0] = 5
    return nil
}

func PanicRecovery(err *error) {

    if r := recover(); r != nil {
        if _, ok := r.(runtime.Error); ok {
             //fmt.Println("Panicing")
             //panic(r)
             *err = r.(error)
        } else {
            *err = r.(error)
        }
    }
}
```

El uso de una función separada (en lugar del cierre) permite la reutilización de la misma función en otras funciones propensas al pánico.

Lea Manejo de errores en línea: https://riptutorial.com/es/go/topic/785/manejo-de-errores

Capítulo 47: Mapas

Introducción

Los mapas son tipos de datos utilizados para almacenar pares clave-valor no ordenados, de modo que buscar el valor asociado a una clave dada es muy eficiente. Las llaves son únicas. La estructura de datos subyacente crece según sea necesario para acomodar nuevos elementos, por lo que el programador no tiene que preocuparse por la administración de la memoria. Son similares a lo que otros lenguajes denominan tablas hash, diccionarios o matrices asociativas.

Sintaxis

- var mapName map [KeyType] ValueType // declara un mapa
- var mapName = map [KeyType] ValueType {} // declara y asigna un mapa vacío
- var mapName = map [KeyType] ValueType {key1: val1, key2: val2} // declara y asigna un Map
- mapName: = make (map [KeyType] ValueType) // declara e inicializa el mapa de tamaño predeterminado
- mapName: = make (map [KeyType] ValueType, length) // declara e inicializa el mapa de tamaño de *longitud*
- mapName: = map [KeyType] ValueType {} // auto-declara y asigna un mapa vacío con: =
- mapName: = map [KeyType] ValueType {key1: value1, key2: value2} // declarar automáticamente y asignar un mapa con: =
- valor: = mapName [clave] // Obtener valor por clave
- value, hasKey: = mapName [key] // Obtener valor por clave, 'hasKey' es 'true' si la clave existe en el mapa
- mapName [clave] = valor // Establecer valor por clave

Observaciones

Go proporciona un tipo de `map` incorporado que implementa una *tabla hash* . Los mapas son el tipo de datos asociativos incorporados de Go (también llamados *hashes* o *diccionarios* en otros idiomas).

Examples

Declarar e inicializar un mapa

Usted define un mapa usando el `map` palabras clave, seguido de los tipos de sus claves y sus valores:

```
// Keys are ints, values are ints.
var m1 map[int]int // initialized to nil
```

```
// Keys are strings, values are ints.
var m2 map[string]int // initialized to nil
```

Los mapas son tipos de referencia y, una vez definidos, tienen un *valor cero* de `nil` . Las escrituras en mapas nulos entrarán en pánico y las lecturas siempre devolverán el valor cero.

Para inicializar un mapa, use la función `make` :

```
m := make(map[string]int)
```

Con la forma de `make` de dos parámetros, es posible especificar una capacidad de entrada inicial para el mapa, anulando la capacidad predeterminada:

```
m := make(map[string]int, 30)
```

Alternativamente, puede declarar un mapa, inicializarlo a su valor cero, y luego asignarle un valor literal más tarde, lo que ayuda si calibra la estructura en json para producir un mapa vacío en el retorno.

```
m := make(map[string]int, 0)
```

También puede hacer un mapa y establecer su valor inicial entre llaves (`{}`).

```
var m map[string]int = map[string]int{"Foo": 20, "Bar": 30}

fmt.Println(m["Foo"]) // outputs 20
```

Todas las siguientes declaraciones dan como resultado que la variable se enlaza al mismo valor.

```
// Declare, initializing to zero value, then assign a literal value.
var m map[string]int
m = map[string]int{}

// Declare and initialize via literal value.
var m = map[string]int{}

// Declare via short variable declaration and initialize with a literal value.
m := map[string]int{}
```

También podemos usar un *mapa literal* para crear un nuevo mapa con algunos pares de clave / valor iniciales .

El tipo de clave puede ser cualquier tipo comparable ; En particular, esto excluye funciones, mapas y segmentos . El tipo de valor puede ser cualquier tipo, incluidos los tipos personalizados o la `interface{}` .

```
type Person struct {
    FirstName string
    LastName  string
}
```

```
// Declare via short variable declaration and initialize with make.
m := make(map[string]Person)

// Declare, initializing to zero value, then assign a literal value.
var m map[string]Person
m = map[string]Person{}

// Declare and initialize via literal value.
var m = map[string]Person{}

// Declare via short variable declaration and initialize with a literal value.
m := map[string]Person{}
```

Creando un mapa

Uno puede declarar e inicializar un mapa en una sola declaración usando un *literal compuesto* .

Usando el tipo automático de declaración de variable corta:

```
mapIntInt := map[int]int{10: 100, 20: 100, 30: 1000}
mapIntString := map[int]string{10: "foo", 20: "bar", 30: "baz"}
mapStringInt := map[string]int{"foo": 10, "bar": 20, "baz": 30}
mapStringString := map[string]string{"foo": "one", "bar": "two", "baz": "three"}
```

El mismo código, pero con tipos de variables:

```
var mapIntInt = map[int]int{10: 100, 20: 100, 30: 1000}
var mapIntString = map[int]string{10: "foo", 20: "bar", 30: "baz"}
var mapStringInt = map[string]int{"foo": 10, "bar": 20, "baz": 30}
var mapStringString = map[string]string{"foo": "one", "bar": "two", "baz": "three"}
```

También puedes incluir tus propias estructuras en un mapa:

Puedes usar tipos personalizados como valor:

```
// Custom struct types
type Person struct {
  FirstName, LastName string
}

var mapStringPerson = map[string]Person{
  "john": Person{"John", "Doe"},
  "jane": Person{"Jane", "Doe"}}
mapStringPerson := map[string]Person{
  "john": Person{"John", "Doe"},
  "jane": Person{"Jane", "Doe"}}
```

Tu estructura también puede ser la *clave* del mapa:

```
type RouteHit struct {
    Domain string
    Route  string
}
```

```
var hitMap = map[RouteHit]int{
  RouteHit{"example.com","/home"}: 1,
  RouteHit{"example.com","/help"}: 2}
hitMap := map[RouteHit]int{
  RouteHit{"example.com","/home"}: 1,
  RouteHit{"example.com","/help"}: 2}
```

Puede crear un mapa vacío simplemente no ingresando ningún valor entre los corchetes `{}` .

```
mapIntInt := map[int]int{}
mapIntString := map[int]string{}
mapStringInt := map[string]int{}
mapStringString := map[string]string{}
mapStringPerson := map[string]Person{}
```

Puede crear y usar un mapa directamente, sin la necesidad de asignarlo a una variable. Sin embargo, deberá especificar tanto la declaración como el contenido.

```
// using a map as argument for fmt.Println()
fmt.Println(map[string]string{
  "FirstName": "John",
  "LastName": "Doe",
  "Age": "30"})

// equivalent to
data := map[string]string{
  "FirstName": "John",
  "LastName": "Doe",
  "Age": "30"}
fmt.Println(data)
```

Valor cero de un mapa

El valor cero de un `map` es `nil` y tiene una longitud de `0` .

```
var m map[string]string
fmt.Println(m == nil) // true
fmt.Println(len(m) ==0) // true
```

Un mapa `nil` no tiene claves ni se pueden agregar claves. Un mapa `nil` comporta como un mapa vacío si se lee desde, pero provoca un pánico en el tiempo de ejecución si se escribe.

```
var m map[string]string

// reading
m["foo"] == "" // true. Remember "" is the zero value for a string
_, ok = m["foo"] // ok == false

// writing
m["foo"] = "bar" // panic: assignment to entry in nil map
```

No debe intentar leer o escribir en un mapa de valor cero. En su lugar, inicialice el mapa (con `make`

o asignación) antes de usarlo.

```
var m map[string]string
m = make(map[string]string) // OR m = map[string]string{}
m["foo"] = "bar"
```

Iterando los elementos de un mapa.

```
import fmt

people := map[string]int{
  "john": 30,
  "jane": 29,
  "mark": 11,
}

for key, value := range people {
  fmt.Println("Name:", key, "Age:", value)
}
```

Tenga en cuenta que al iterar sobre un mapa con un bucle de rango, el orden de iteración no se especifica y no se garantiza que sea igual de una iteración a la siguiente.

También puede descartar las claves o los valores del mapa, si está buscando simplemente agarrar las claves o simplemente tomar los valores.

Iterando las teclas de un mapa.

```
people := map[string]int{
  "john": 30,
  "jane": 29,
  "mark": 11,
}

for key, _ := range people {
  fmt.Println("Name:", key)
}
```

Si solo está buscando las claves, ya que son el primer valor, simplemente puede colocar el guión bajo:

```
for key := range people {
  fmt.Println("Name:", key)
}
```

Tenga en cuenta que al iterar sobre un mapa con un bucle de rango, el orden de iteración no se especifica y no se garantiza que sea igual de una iteración a la siguiente.

Eliminar un elemento del mapa

La función incorporada de `delete` elimina el elemento con la clave especificada de un mapa.

```
people := map[string]int{"john": 30, "jane": 29}
fmt.Println(people) // map[john:30 jane:29]

delete(people, "john")
fmt.Println(people) // map[jane:29]
```

Si el `map` es `nil` o no existe tal elemento, `delete` no tiene efecto.

```
people := map[string]int{"john": 30, "jane": 29}
fmt.Println(people) // map[john:30 jane:29]

delete(people, "notfound")
fmt.Println(people) // map[john:30 jane:29]

var something map[string]int
delete(something, "notfound") // no-op
```

Contando elementos del mapa

La función incorporada `len` devuelve el número de elementos en un `map`

```
m := map[string]int{}
len(m) // 0

m["foo"] = 1
len(m) // 1
```

Si una variable apunta a un mapa `nil` , `len` devuelve 0.

```
var m map[string]int
len(m) // 0
```

Acceso concurrente de mapas

Los mapas en go no son seguros para la concurrencia. Debe tomar un candado para leer y escribir en ellos si va a acceder a ellos al mismo tiempo. Generalmente, la mejor opción es usar `sync.RWMutex` porque puede tener bloqueos de lectura y escritura. Sin embargo, un `sync.Mutex` también podría ser utilizado.

```
type RWMap struct {
    sync.RWMutex
    m map[string]int
}

// Get is a wrapper for getting the value from the underlying map
func (r RWMap) Get(key string) int {
    r.RLock()
    defer r.RUnlock()
    return r.m[key]
}

// Set is a wrapper for setting the value of a key in the underlying map
func (r RWMap) Set(key string, val int) {
```

```
    r.Lock()
    defer r.Unlock()
    r.m[key] = val
}

// Inc increases the value in the RWMap for a key.
//   This is more pleasant than r.Set(key, r.Get(key)++)
func (r RWMap) Inc(key string) {
    r.Lock()
    defer r.Unlock()
    r.m[key]++
}

func main() {

    // Init
    counter := RWMap{m: make(map[string]int)}

    // Get a Read Lock
    counter.RLock()
    _ = counter.["Key"]
    counter.RUnlock()

    // the above could be replaced with
    _ = counter.Get("Key")

    // Get a write Lock
    counter.Lock()
    counter.m["some_key"]++
    counter.Unlock()

    // above would need to be written as
    counter.Inc("some_key")
}
```

La compensación de las funciones de envoltura es entre el acceso público del mapa subyacente y el uso correcto de los bloqueos apropiados.

Creación de mapas con cortes como valores.

```
m := make(map[string][]int)
```

El acceso a una clave no existente devolverá una porción nula como un valor. Dado que los segmentos nulos actúan como segmentos de longitud cero cuando se usan con funciones `append` u otras funciones incorporadas, normalmente no es necesario verificar para ver si existe una clave:

```
// m["key1"] == nil && len(m["key1"]) == 0
m["key1"] = append(m["key1"], 1)
// len(m["key1"]) == 1
```

Al eliminar una clave del mapa, la clave vuelve a ser nula:

```
delete(m, "key1")
// m["key1"] == nil
```

Verificar elemento en un mapa

Para obtener un valor del mapa, solo tienes que hacer algo como: 00

```
value := mapName[ key ]
```

Si el mapa contiene la clave, devuelve el valor correspondiente.
Si no, devuelve el valor cero del tipo de valor del mapa (`0` si es un mapa de valores `int` , `""` si es un mapa de valores de `string` ...)

```
m  := map[string]string{"foo": "foo_value", "bar": ""}
k  := m["foo"]  // returns "foo_value" since that is the value stored in the map
k2 := m["bar"]  // returns "" since that is the value stored in the map
k3 := m["nop"]  // returns "" since the key does not exist, and "" is the string type's zero
value
```

Para diferenciar entre valores vacíos y claves no existentes, puede usar el segundo valor devuelto del acceso al mapa (usando el `value, hasKey := map["key"]` like `value, hasKey := map["key"]`).

Este segundo valor es de tipo `boolean` , y será:

- `true` cuando el valor está en el mapa,
- `false` cuando el mapa no contiene la clave dada.

Mira el siguiente ejemplo:

```
value, hasKey = m[ key ]
if hasKey {
    // the map contains the given key, so we can safely use the value
    // If value is zero-value, it's because the zero-value was pushed to the map
} else {
    // The map does not have the given key
    // the value will be the zero-value of the map's type
}
```

Iterando los valores de un mapa.

```
people := map[string]int{
  "john": 30,
  "jane": 29,
  "mark": 11,
}

for _, value := range people {
  fmt.Println("Age:", value)
}
```

Tenga en cuenta que al iterar sobre un mapa con un bucle de rango, el orden de iteración no se especifica y no se garantiza que sea igual de una iteración a la siguiente.

Copiar un mapa

Al igual que los cortes, los mapas contienen **referencias** a una estructura de datos subyacente. Entonces, al asignar su valor a otra variable, solo se pasará la referencia. Para copiar el mapa, es necesario crear otro mapa y copiar cada valor:

```
// Create the original map
originalMap := make(map[string]int)
originalMap["one"] = 1
originalMap["two"] = 2

// Create the target map
targetMap := make(map[string]int)

// Copy from the original map to the target map
for key, value := range originalMap {
  targetMap[key] = value
}
```

Usando un mapa como conjunto

Algunos idiomas tienen una estructura nativa para conjuntos. Para hacer un conjunto en Go, se recomienda usar un mapa del tipo de valor del conjunto a una estructura vacía (`map[Type]struct{}`).

Por ejemplo, con cuerdas:

```
// To initialize a set of strings:
greetings := map[string]struct{}{
    "hi":    {},
    "hello": {},
}

// To delete a value:
delete(greetings, "hi")

// To add a value:
greetings["hey"] = struct{}{}

// To check if a value is in the set:
if _, ok := greetings["hey"]; ok {
    fmt.Println("hey is in greetings")
}
```

Lea Mapas en línea: https://riptutorial.com/es/go/topic/732/mapas

Capítulo 48: Métodos

Sintaxis

- func (t T) exampleOne (i int) (n int) {return i} // esta función recibirá una copia de la estructura
- func (t * T) exampleTwo (i int) (n int) {return i} // este método recibirá el puntero a la estructura y podrá modificarlo

Examples

Metodos basicos

Los métodos en Go son como funciones, excepto que tienen *receptor* .

Normalmente el receptor es algún tipo de estructura o tipo.

```
package main

import (
    "fmt"
)

type Employee struct {
    Name string
    Age  int
    Rank int
}

func (empl *Employee) Promote() {
    empl.Rank++
}

func main() {

    Bob := new(Employee)

    Bob.Rank = 1
    fmt.Println("Bobs rank now is: ", Bob.Rank)
    fmt.Println("Lets promote Bob!")

    Bob.Promote()

    fmt.Println("Now Bobs rank is: ", Bob.Rank)

}
```

Salida:

```
Bobs rank now is:  1
Lets promote Bob!
Now Bobs rank is:  2
```

Métodos de encadenamiento

Con los métodos en golang puede hacer el método "encadenar" pasando el puntero al método y devolviendo el puntero a la misma estructura como esta:

```
package main

import (
    "fmt"
)

type Employee struct {
    Name string
    Age  int
    Rank int
}

func (empl *Employee) Promote() *Employee {
    fmt.Printf("Promoting %s\n", empl.Name)
    empl.Rank++
    return empl
}

func (empl *Employee) SetName(name string) *Employee {
    fmt.Printf("Set name of new Employee to %s\n", name)
    empl.Name = name
    return empl
}

func main() {

    worker := new(Employee)

    worker.Rank = 1

    worker.SetName("Bob").Promote()

    fmt.Printf("Here we have %s with rank %d\n", worker.Name, worker.Rank)

}
```

Salida:

```
Set name of new Employee to Bob
Promoting Bob
Here we have Bob with rank 2
```

Operadores de incremento-decremento como argumentos en los métodos

Aunque Go admite operadores ++ y - y se encuentra que el comportamiento es casi similar a c / c ++, las variables con dichos operadores no se pueden pasar como argumento para funcionar.

```
    package main

    import (
        "fmt"
```

```
)

func abcd(a int, b int) {
 fmt.Println(a," ",b)
}
func main() {
    a:=5
    abcd(a++,++a)
}
```

Salida: error de sintaxis: inesperado ++, esperando una coma o)

Lea Métodos en línea: https://riptutorial.com/es/go/topic/3890/metodos

Capítulo 49: mgo

Introducción

mgo (pronunciado como mango) es un controlador MongoDB para el lenguaje Go que implementa una selección rica y bien probada de características bajo una API muy simple que sigue los modismos estándar de Go.

Observaciones

Documentación API

[https://gopkg.in/mgo.v2◆◆1]

Examples

Ejemplo

```
package main

import (
        "fmt"
        "log"
        "gopkg.in/mgo.v2"
        "gopkg.in/mgo.v2/bson"
)

type Person struct {
        Name string
        Phone string
}

func main() {
        session, err := mgo.Dial("server1.example.com,server2.example.com")
        if err != nil {
                panic(err)
        }
        defer session.Close()

        // Optional. Switch the session to a monotonic behavior.
        session.SetMode(mgo.Monotonic, true)

        c := session.DB("test").C("people")
        err = c.Insert(&Person{"Ale", "+55 53 8116 9639"},
                   &Person{"Cla", "+55 53 8402 8510"})
        if err != nil {
                log.Fatal(err)
        }

        result := Person{}
        err = c.Find(bson.M{"name": "Ale"}).One(&result)
        if err != nil {
```

```
                log.Fatal(err)
        }

        fmt.Println("Phone:", result.Phone)
}
```

Lea mgo en línea: https://riptutorial.com/es/go/topic/8898/mgo

Capítulo 50: Middleware

Introducción

En Go Middleware se puede usar para ejecutar código antes y después de la función del controlador. Utiliza el poder de las interfaces de una sola función. Se puede introducir en cualquier momento sin afectar al otro middleware. Por ejemplo: el registro de autenticación se puede agregar en etapas posteriores del desarrollo sin alterar el código existente.

Observaciones

La **firma del middleware** debe ser (http.ResponseWriter, * http.Request), es decir, de tipo **http.handlerFunc** .

Examples

Función normal del manejador

```
func loginHandler(w http.ResponseWriter, r *http.Request) {
            // Steps to login
}


func main() {
    http.HandleFunc("/login", loginHandler)
    http.ListenAndServe(":8080", nil)
}
```

Middleware Calcular el tiempo requerido para que handlerFunc se ejecute

```
// logger middlerware that logs time taken to process each request
func Logger(h http.Handler) http.Handler {
    return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
        startTime := time.Now()
        h.ServeHttp(w,r)
        endTime := time.Since(startTime)
        log.Printf("%s %d %v", r.URL, r.Method, endTime)
    })
}

func loginHandler(w http.ResponseWriter, r *http.Request) {
            // Steps to login
}


func main() {
    http.HandleFunc("/login", Logger(loginHandler))
    http.ListenAndServe(":8080", nil)
}
```

CORS Middleware

```
func CORS(h http.Handler) http.Handler {
    return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
        origin := r.Header.Get("Origin")
        w.Header().Set("Access-Control-Allow-Origin", origin)
        if r.Method == "OPTIONS" {
            w.Header().Set("Access-Control-Allow-Credentials", "true")
            w.Header().Set("Access-Control-Allow-Methods", "GET,POST")

            w.RespWriter.Header().Set("Access-Control-Allow-Headers", "Content-Type, X-CSRF-
Token, Authorization")
            return
        } else {
            h.ServeHTTP(w, r)
        }
    })
}

func main() {
    http.HandleFunc("/login", Logger(CORS(loginHandler)))
    http.ListenAndServe(":8080", nil)
}
```

Auth Middleware

```
func Authenticate(h http.Handler) http.Handler {
    return CustomHandlerFunc(func(w *http.ResponseWriter, r *http.Request) {
        // extract params from req
        // post params | headers etc
        if CheckAuth(params) {
            log.Println("Auth Pass")
            // pass control to next middleware in chain or handler func
            h.ServeHTTP(w, r)
        } else {
            log.Println("Auth Fail")
            // Responsd Auth Fail
        }
    })
}
```

Controlador de recuperación para evitar que el servidor se bloquee

```
func Recovery(h http.Handler) http.Handler {
    return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request){
        defer func() {
            if err := recover(); err != nil {
                // respondInternalServerError
            }
        }()
        h.ServeHTTP(w , r)
    })
}
```

Lea Middleware en línea: https://riptutorial.com/es/go/topic/9343/middleware

Capítulo 51: Mutex

Examples

Bloqueo mutex

El bloqueo mutex en Go le permite asegurarse de que solo una goroutina a la vez tenga un bloqueo:

```
import "sync"

func mutexTest() {
    lock := sync.Mutex{}
    go func(m *sync.Mutex) {
        m.Lock()
        defer m.Unlock()   // Automatically unlock when this function returns
        // Do some things
    }(&lock)

    lock.Lock()
    // Do some other things
    lock.Unlock()
}
```

El uso de `Mutex` permite evitar condiciones de carrera, modificaciones concurrentes y otros problemas asociados con múltiples rutinas concurrentes que operan en los mismos recursos. Tenga en cuenta que `Mutex.Unlock()` puede ejecutarse por cualquier rutina, no solo por la rutina que obtuvo el bloqueo. También tenga en cuenta que la llamada a `Mutex.Lock()` no fallará si otra rutina mantiene el bloqueo; se bloqueará hasta que se libere el bloqueo.

Consejo: Siempre que esté pasando una variable Mutex a una función, siempre pásela como un puntero. De lo contrario, se hace una copia de su variable, lo que anula el propósito del Mutex. Si está utilizando una versión anterior de Go (<1.7), el compilador no le advertirá sobre este error.

Lea Mutex en línea: https://riptutorial.com/es/go/topic/2607/mutex

Capítulo 52: Pánico y Recuperación

Observaciones

Este artículo supone el conocimiento de los conceptos básicos de aplazamiento.

Para el manejo de errores ordinarios, lea el tema sobre el manejo de errores.

Examples

Pánico

Un pánico detiene el flujo de ejecución normal y sale de la función actual. Cualquier llamada diferida se ejecutará antes de pasar el control a la siguiente función superior en la pila. La función de cada pila saldrá y ejecutará llamadas diferidas hasta que el pánico se maneje usando un `recover()` diferido `recover()` , o hasta que el pánico alcance `main()` y finalice el programa. Si esto ocurre, el argumento proporcionado para entrar en pánico y un seguimiento de pila se imprimirá en `stderr` .

```
package main

import "fmt"

func foo() {
    defer fmt.Println("Exiting foo")
    panic("bar")
}

func main() {
    defer fmt.Println("Exiting main")
    foo()
}
```

Salida:

```
Exiting foo
Exiting main
panic: bar


goroutine 1 [running]:
panic(0x128360, 0x1040a130)
    /usr/local/go/src/runtime/panic.go:481 +0x700
main.foo()
    /tmp/sandbox550159908/main.go:7 +0x160
main.main()
    /tmp/sandbox550159908/main.go:12 +0x120
```

Es importante tener en cuenta que el `panic` aceptará cualquier tipo como parámetro.

Recuperar

Recuperar como su nombre lo indica, puede intentar recuperarse de un `panic` . La recuperación *debe* intentarse en una declaración diferida ya que el flujo de ejecución normal se ha detenido. La instrucción de `recover` debe aparecer *directamente* dentro del encierro de la función diferida. No se aceptarán las declaraciones de recuperación en funciones llamadas por llamadas de función diferidas. La llamada `recover()` devolverá el argumento provisto al pánico inicial, si el programa está actualmente en pánico. Si el programa no tiene pánico actualmente, `recover()` devolverá `nil` .

```
package main

import "fmt"

func foo() {
    panic("bar")
}

func bar() {
    defer func() {
        if msg := recover(); msg != nil {
            fmt.Printf("Recovered with message %s\n", msg)
        }
    }()
    foo()
    fmt.Println("Never gets executed")
}

func main() {
    fmt.Println("Entering main")
    bar()
    fmt.Println("Exiting main the normal way")
}
```

Salida:

```
Entering main
Recovered with message bar
Exiting main the normal way
```

Lea Pánico y Recuperación en línea: https://riptutorial.com/es/go/topic/4350/panico-y-recuperacion

Capítulo 53: Paquetes

Examples

Inicialización de paquetes

El paquete puede tener métodos de `init` que se ejecutan **solo una vez** antes de main.

```
package usefull

func init() {
    // init code
}
```

Si solo desea ejecutar la inicialización del paquete sin hacer referencia a nada, use la siguiente expresión de importación.

```
import _ "usefull"
```

Gestionando dependencias de paquetes.

Una forma común de descargar las dependencias de Go es mediante el comando `go get <package>` , que guardará el paquete en el directorio global / shared `$GOPATH/src` . Esto significa que una única versión de cada paquete se vinculará a cada proyecto que lo incluya como una dependencia. Esto también significa que cuando un nuevo desarrollador implemente su proyecto, `go get` la última versión de cada dependencia.

Sin embargo, puede mantener el entorno de compilación consistente, adjuntando todas las dependencias de un proyecto en el directorio del `vendor/` . Mantener las dependencias vendidas comprometidas junto con el repositorio de su proyecto le permite realizar versiones de dependencia por proyecto y proporcionar un entorno coherente para su compilación.

Así se verá la estructura de tu proyecto:

```
$GOPATH/src/
├── github.com/username/project/
|   ├── main.go
|   ├── vendor/
|   |   ├── github.com/pkg/errors
|   |   ├── github.com/gorilla/mux
```

Usando diferentes paquetes y nombres de carpetas

Está perfectamente bien usar un nombre de paquete que no sea el nombre de la carpeta. Si lo hacemos, todavía tenemos que importar el paquete en función de la estructura del directorio, pero después de la importación debemos referirnos a él por el nombre que usamos en la cláusula del paquete.

Por ejemplo, si tiene una carpeta `$GOPATH/src/mypck` , y en ella tenemos un archivo `a.go` :

```
package apple

const Pi = 3.14
```

Utilizando este paquete:

```
package main

import (
    "mypck"
    "fmt"
)

func main() {
    fmt.Println(apple.Pi)
}
```

A pesar de que esto funciona, debería tener una buena razón para desviar el nombre del paquete del nombre de la carpeta (o puede convertirse en una fuente de malentendidos y confusión).

¿Para qué sirve esto?

Sencillo. Un nombre de paquete es un identificador Go:

```
identifier = letter { letter | unicode_digit } .
```

Lo que permite usar letras Unicode en identificadores, por ejemplo, `αβ` es un identificador válido en Go. Los nombres de carpetas y archivos no son manejados por Go, sino por el sistema operativo, y los diferentes sistemas de archivos tienen diferentes restricciones. En realidad, hay muchos sistemas de archivos que no permitirían todos los identificadores Go válidos como nombres de carpetas, por lo que no podría nombrar sus paquetes lo que de otro modo permitiría la especificación de idioma.

Al tener la opción de usar nombres de paquetes diferentes a los de las carpetas que contienen, tiene la opción de nombrar realmente a sus paquetes lo que permite la especificación de idioma, independientemente del sistema operativo y de archivos subyacente.

Importando paquetes

Puede importar un solo paquete con la declaración:

```
import "path/to/package"
```

o agrupar múltiples importaciones juntas:

```
import (
    "path/to/package1"
    "path/to/package2"
```

```
)
```

Esto se verá en los correspondientes `import` caminos en el interior de la `$GOPATH` para `.go` archivos y le permite acceder a los nombres exportados a través `packagename.AnyExportedName` .

También puede acceder a los paquetes locales dentro de la carpeta actual introduciendo los paquetes con `./` . En un proyecto con una estructura como esta:

```
project
├── src
│   ├── package1
│   │   └── file1.go
│   └── package2
│       └── file2.go
└── main.go
```

Puede llamar a esto en `main.go` para importar el código en `file1.go` y `file2.go` :

```
import (
    "./src/package1"
    "./src/package2"
)
```

Dado que los nombres de paquetes pueden colisionar en diferentes bibliotecas, es posible que desee asignar un alias de un paquete a un nombre nuevo. Puede hacer esto prefijando su declaración de importación con el nombre que desea usar.

```
import (
    "fmt" //fmt from the standardlibrary
    tfmt "some/thirdparty/fmt" //fmt from some other library
)
```

Esto le permite acceder a la antigua `fmt` paquete mediante `fmt.*` Y el segundo `fmt` paquete utilizando `tfmt.*` .

También puede importar el paquete en el propio espacio de nombres, de modo que puede hacer referencia a los nombres exportados sin el `package.` prefijo usando un solo punto como alias:

```
import (
    . "fmt"
)
```

El ejemplo anterior importa `fmt` en el espacio de nombres global y le permite llamar, por ejemplo, directamente a `Printf` : Playground

Si importa un paquete pero no utiliza ninguno de sus nombres exportados, el compilador Go imprimirá un mensaje de error. Para evitar esto, puede establecer el alias en el guión bajo:

```
import (
    _ "fmt"
)
```

Esto puede ser útil si no accede directamente a este paquete pero necesita las funciones de `init` para ejecutarse.

Nota:

Como los nombres de los paquetes se basan en la estructura de la carpeta, cualquier cambio en los nombres de las carpetas y las referencias de importación (incluida la distinción entre mayúsculas y minúsculas) causará un error de compilación "colisión de importación insensible a mayúsculas y minúsculas" en Linux y OS-X, que es difícil de rastrear y corregir (el mensaje de error es un tanto críptico para simples mortales, ya que trata de transmitir lo contrario: la comparación falló debido a la sensibilidad del caso).

ej: "ruta / a / paquete1" vs "ruta / a / paquete1"

Ejemplo en vivo: https://github.com/akamai-open/AkamaiOPEN-edgegrid-golang/issues/2

Lea Paquetes en línea: https://riptutorial.com/es/go/topic/401/paquetes

Capítulo 54: Perfilado usando la herramienta go pprof

Observaciones

Para más información sobre los programas go, visite el blog go .

Examples

Perfil básico de cpu y memoria.

Agregue el siguiente código en su programa principal.

```
var cpuprofile = flag.String("cpuprofile", "", "write cpu profile `file`")
var memprofile = flag.String("memprofile", "", "write memory profile to `file`")

func main() {
    flag.Parse()
    if *cpuprofile != "" {
        f, err := os.Create(*cpuprofile)
        if err != nil {
            log.Fatal("could not create CPU profile: ", err)
        }
        if err := pprof.StartCPUProfile(f); err != nil {
            log.Fatal("could not start CPU profile: ", err)
        }
        defer pprof.StopCPUProfile()
    }
    ...
    if *memprofile != "" {
        f, err := os.Create(*memprofile)
        if err != nil {
            log.Fatal("could not create memory profile: ", err)
        }
        runtime.GC() // get up-to-date statistics
        if err := pprof.WriteHeapProfile(f); err != nil {
            log.Fatal("could not write memory profile: ", err)
        }
        f.Close()
    }
}
```

después de eso, **construya** el programa go si se agrega en main `go build main.go` Ejecute el programa principal con los indicadores definidos en el código `main.exe -cpuprofile cpu.prof -memprof mem.prof` . Si se realiza el perfilado para los casos de prueba, ejecute las pruebas con los mismos indicadores `go test -cpuprofile cpu.prof -memprofile mem.prof`

Memoria básica de perfiles

```
var memprofile = flag.String("memprofile", "", "write memory profile to `file`")
```

```
func main() {
    flag.Parse()
    if *memprofile != "" {
        f, err := os.Create(*memprofile)
        if err != nil {
            log.Fatal("could not create memory profile: ", err)
        }
        runtime.GC() // get up-to-date statistics
        if err := pprof.WriteHeapProfile(f); err != nil {
            log.Fatal("could not write memory profile: ", err)
        }
        f.Close()
    }
}
```

```
go build main.go
main.exe -memprofile mem.prof
go tool pprof main.exe mem.prof
```

Establecer la tasa de perfil de CPU / bloque

```
// Sets the CPU profiling rate to hz samples per second
// If hz <= 0, SetCPUProfileRate turns off profiling
runtime.SetCPUProfileRate(hz)

// Controls the fraction of goroutine blocking events that are reported in the blocking
profile
// Rate = 1 includes every blocking event in the profile
// Rate <= 0 turns off profiling
runtime.SetBlockProfileRate(rate)
```

Uso de puntos de referencia para crear perfil

Para un paquete no principal y principal, en **lugar de agregar indicadores dentro del código** , escriba **puntos** de **referencia** en el paquete de prueba, por ejemplo:

```
func BenchmarkHello(b *testing.B) {
    for i := 0; i < b.N; i++ {
        fmt.Sprintf("hello")
    }
}
```

A continuación, ejecute la prueba con la bandera de perfil

> ir prueba -cpuprofile cpu.prof -bench =.

Y los puntos de referencia se ejecutarán y crearán un archivo prof con el nombre de archivo cpu.prof (en el ejemplo anterior).

Accediendo al archivo de perfil

Una vez que se ha generado un archivo prof, se puede acceder al archivo prof con las **herramientas go** :

    ir herramienta pprof cpu.prof

Esto entrará en una interfaz de línea de comandos para explorar el `profile`

Los comandos comunes incluyen:

```
(pprof) top
```

Enumera los procesos principales en la memoria.

```
(pprof) peek
```

Enumera todos los procesos, usa *expresiones regulares* para restringir la búsqueda.

```
(pprof) web
```

Abre un gráfico (en formato svg) del proceso.

Un ejemplo del comando `top` :

```
69.29s of 100.84s total (68.71%)
Dropped 176 nodes (cum <= 0.50s)
Showing top 10 nodes out of 73 (cum >= 12.03s)
      flat  flat%   sum%        cum   cum%
    12.44s 12.34% 12.34%     27.87s 27.64%  runtime.mapaccess1
    10.94s 10.85% 23.19%     10.94s 10.85%  runtime.duffcopy
     9.45s  9.37% 32.56%     54.61s 54.16%  github.com/tester/test.(*Circle).Draw
     8.88s  8.81% 41.36%      8.88s  8.81%  runtime.aeshashbody
     7.90s  7.83% 49.20%     11.04s 10.95%  runtime.mapaccess1_fast64
     5.86s  5.81% 55.01%      9.59s  9.51%  github.com/tester/test.(*Circle).isCircle
     5.03s  4.99% 60.00%      8.89s  8.82%  github.com/tester/test.(*Circle).openCircle
     3.14s  3.11% 63.11%      3.14s  3.11%  runtime.aeshash64
     3.08s  3.05% 66.16%      7.85s  7.78%  runtime.mallocgc
     2.57s  2.55% 68.71%     12.03s 11.93%  runtime.memhash
```

Lea Perfilado usando la herramienta go pprof en línea:
https://riptutorial.com/es/go/topic/7748/perfilado-usando-la-herramienta-go-pprof

Capítulo 55: Piscinas de trabajadores

Examples

Grupo de trabajadores simple

Una simple implementación de pool de trabajadores:

```
package main

import (
    "fmt"
    "sync"
)

type job struct {
    // some fields for your job type
}

type result struct {
    // some fields for your result type
}

func worker(jobs <-chan job, results chan<- result) {
    for j := range jobs {
        var r result
        // do some work
        results <- r
    }
}

func main() {
    // make our channels for communicating work and results
    jobs := make(chan job, 100) // 100 was chosen arbitrarily
    results := make(chan result, 100)

    // spin up workers and use a sync.WaitGroup to indicate completion
    wg := sync.WaitGroup
    for i := 0; i < runtime.NumCPU; i++ {
        wg.Add(1)
        go func() {
            defer wg.Done()
            worker(jobs, results)
        }()
    }

    // wait on the workers to finish and close the result channel
    // to signal downstream that all work is done
    go func() {
        defer close(results)
        wg.Wait()
    }()

    // start sending jobs
    go func() {
        defer close(jobs)
```

```
        for {
            jobs <- getJob()   // I haven't defined getJob() and noMoreJobs()
            if noMoreJobs() {  // they are just for illustration
                break
            }
        }
    }()

    // read all the results
    for r := range results {
        fmt.Println(r)
    }
}
```

Cola de trabajos con grupo de trabajadores

Una cola de trabajos que mantiene un grupo de trabajadores, útil para hacer cosas como el procesamiento en segundo plano en servidores web:

```
package main

import (
  "fmt"
  "runtime"
  "strconv"
  "sync"
  "time"
)

// Job - interface for job processing
type Job interface {
  Process()
}

// Worker - the worker threads that actually process the jobs
type Worker struct {
  done             sync.WaitGroup
  readyPool        chan chan Job
  assignedJobQueue chan Job

  quit chan bool
}

// JobQueue - a queue for enqueueing jobs to be processed
type JobQueue struct {
  internalQueue     chan Job
  readyPool         chan chan Job
  workers           []*Worker
  dispatcherStopped sync.WaitGroup
  workersStopped    sync.WaitGroup
  quit              chan bool
}

// NewJobQueue - creates a new job queue
func NewJobQueue(maxWorkers int) *JobQueue {
  workersStopped := sync.WaitGroup{}
  readyPool := make(chan chan Job, maxWorkers)
  workers := make([]*Worker, maxWorkers, maxWorkers)
  for i := 0; i < maxWorkers; i++ {
```

```
    workers[i] = NewWorker(readyPool, workersStopped)
  }
  return &JobQueue{
    internalQueue:     make(chan Job),
    readyPool:         readyPool,
    workers:           workers,
    dispatcherStopped: sync.WaitGroup{},
    workersStopped:    workersStopped,
    quit:              make(chan bool),
  }
}

// Start - starts the worker routines and dispatcher routine
func (q *JobQueue) Start() {
  for i := 0; i < len(q.workers); i++ {
    q.workers[i].Start()
  }
  go q.dispatch()
}

// Stop - stops the workers and sispatcher routine
func (q *JobQueue) Stop() {
  q.quit <- true
  q.dispatcherStopped.Wait()
}

func (q *JobQueue) dispatch() {
  q.dispatcherStopped.Add(1)
  for {
    select {
    case job := <-q.internalQueue: // We got something in on our queue
      workerChannel := <-q.readyPool // Check out an available worker
      workerChannel <- job           // Send the request to the channel
    case <-q.quit:
      for i := 0; i < len(q.workers); i++ {
        q.workers[i].Stop()
      }
      q.workersStopped.Wait()
      q.dispatcherStopped.Done()
      return
    }
  }
}

// Submit - adds a new job to be processed
func (q *JobQueue) Submit(job Job) {
  q.internalQueue <- job
}

// NewWorker - creates a new worker
func NewWorker(readyPool chan chan Job, done sync.WaitGroup) *Worker {
  return &Worker{
    done:             done,
    readyPool:        readyPool,
    assignedJobQueue: make(chan Job),
    quit:             make(chan bool),
  }
}

// Start - begins the job processing loop for the worker
func (w *Worker) Start() {
```

```
  go func() {
    w.done.Add(1)
    for {
      w.readyPool <- w.assignedJobQueue // check the job queue in
      select {
      case job := <-w.assignedJobQueue: // see if anything has been assigned to the queue
        job.Process()
      case <-w.quit:
        w.done.Done()
        return
      }
    }
  }()
}

// Stop - stops the worker
func (w *Worker) Stop() {
  w.quit <- true
}

///////////////// Example /////////////////

// TestJob - holds only an ID to show state
type TestJob struct {
  ID string
}

// Process - test process function
func (t *TestJob) Process() {
  fmt.Printf("Processing job '%s'\n", t.ID)
  time.Sleep(1 * time.Second)
}

func main() {
  queue := NewJobQueue(runtime.NumCPU())
  queue.Start()
  defer queue.Stop()

  for i := 0; i < 4*runtime.NumCPU(); i++ {
    queue.Submit(&TestJob{strconv.Itoa(i)})
  }
}
```

Lea Piscinas de trabajadores en línea: https://riptutorial.com/es/go/topic/4182/piscinas-de-trabajadores

Capítulo 56: Plantillas

Sintaxis

- t, err: = template.Parse (`{{.MyName .MyAge}}`)
- t.Execute (os.Stdout, struct {MyValue, MyAge string} {"John Doe", "40.1"})

Observaciones

Golang proporciona paquetes como:

1. `text/template`
2. `html/template`

para implementar plantillas basadas en datos para generar resultados textuales y HTML.

Examples

Los valores de salida de la estructura de la estructura a la salida estándar utilizando una plantilla de texto

```
package main

import (
    "log"
    "text/template"
    "os"
)

type Person struct{
    MyName string
    MyAge int
}

var myTempContents string= `
This person's name is : {{.MyName}}
And he is {{.MyAge}} years old.
`

func main() {
    t,err := template.New("myTemp").Parse(myTempContents)
    if err != nil{
        log.Fatal(err)
    }
    myPersonSlice := []Person{ {"John Doe",41},{"Peter Parker",17} }
    for _,myPerson := range myPersonSlice{
        t.Execute(os.Stdout,myPerson)
    }
}
```

Patio de recreo

Definiendo funciones para llamar desde plantilla

```
package main

import (
    "fmt"
    "net/http"
    "os"
    "text/template"
)

var requestTemplate string = `
{{range $i, $url := .URLs}}
{{ $url }} {{(status_code $url)}}
{{ end }}`

type Requests struct {
    URLs []string
}

func main() {
    var fns = template.FuncMap{
        "status_code": func(x string) int {
            resp, err := http.Head(x)
            if err != nil {
                return -1
            }
            return resp.StatusCode
        },
    }

    req := new(Requests)
    req.URLs = []string{"http://godoc.org", "http://stackoverflow.com", "http://linux.org"}

    tmpl := template.Must(template.New("getBatch").Funcs(fns).Parse(requestTemplate))
    err := tmpl.Execute(os.Stdout, req)
    if err != nil {
        fmt.Println(err)
    }
}
```

Aquí usamos nuestra función definida `status_code` para obtener el código de estado de la página web directamente desde la plantilla.

Salida:

```
http://godoc.org 200

http://stackoverflow.com 200

http://linux.org 200
```

Lea Plantillas en línea: https://riptutorial.com/es/go/topic/1402/plantillas

Capítulo 57: Programación orientada a objetos

Observaciones

La interfaz no se puede implementar con receptores de puntero porque `*User` no es `User`

Examples

Estructuras

Go admite tipos definidos por el usuario en forma de estructuras y alias de tipo. Las estructuras son tipos compuestos, los elementos componentes de los datos que constituyen el tipo de estructura se denominan *campos* . un campo tiene un tipo y un nombre que debe ser unqiue.

```
package main

type User struct {
    ID uint64
    FullName string
    Email    string
}

func main() {
    user := User{
        1,
        "Zelalem Mekonen",
        "zola.mk.27@gmail.com",
    }

    fmt.Println(user) // {1 Zelalem Mekonen zola.mk.27@gmail.com}
}
```

Esta es también una sintaxis legal para definir estructuras.

```
type User struct {
    ID uint64
    FullName, Email string
}

user := new(User)

user.ID = 1
user.FullName = "Zelalem Mekonen"
user.Email = "zola.mk.27@gmail.com"
```

Estructuras embebidas

Debido a que una estructura es también un tipo de datos, se puede usar como un campo

anónimo, la estructura externa puede acceder directamente a los campos de la estructura incorporada incluso si la estructura proviene de un paquete diferente. este comportamiento proporciona una manera de derivar parte o la totalidad de su implementación de otro tipo o conjunto de tipos.

```
package main

type Admin struct {
    Username, Password string
}

type User struct {
    ID uint64
    FullName, Email string
    Admin // embedded struct
}

func main() {
    admin := Admin{
        "zola",
        "supersecretpassword",
    }

    user := User{
        1,
        "Zelalem Mekonen",
        "zola.mk.27@gmail.com",
        admin,
    }

    fmt.Println(admin) // {zola supersecretpassword}

    fmt.Println(user) // {1 Zelalem Mekonen zola.mk.27@gmail.com {zola supersecretpassword}}

    fmt.Println(user.Username) // zola

    fmt.Println(user.Password) // supersecretpassword
}
```

Métodos

En Go un método es

> una función que actúa sobre una variable de cierto tipo, llamada el receptor

el receptor puede ser cualquier cosa, no solo `structs` sino incluso una `function` , tipos de alias para tipos integrados como `int` , `string` , `bool` puede tener un método, una excepción a esta regla es que las `interfaces` (que se explican más adelante) no pueden tener métodos, ya que interfaz es una definición abstracta y un método es una implementación, al intentar generar un error de compilación.

combinando `structs` y `methods` puede obtener un equivalente cercano de una `class` en programación orientada a objetos.

Un método en Go tiene la siguiente firma.

```
func (name receiverType) methodName(paramterList) (returnList) {}
```

```
package main

type Admin struct {
    Username, Password string
}

func (admin Admin) Delete() {
    fmt.Println("Admin Deleted")
}

type User struct {
    ID uint64
    FullName, Email string
    Admin
}

func (user User) SendEmail(email string) {
    fmt.Printf("Email sent to: %s\n", user.Email)
}

func main() {
    admin := Admin{
        "zola",
        "supersecretpassword",
    }

    user := User{
        1,
        "Zelalem Mekonen",
        "zola.mk.27@gmail.com",
        admin,
    }

    user.SendEmail("Hello") // Email sent to: zola.mk.27@gmail.com

    admin.Delete() // Admin Deleted
}
```

Puntero Vs Valor receptor

El receptor de un método suele ser un puntero por motivos de rendimiento porque no haríamos una copia de la instancia, como sería el caso en el receptor de valores, esto es especialmente cierto si el tipo de receptor es una estructura. Otra razón para hacer que el receptor escriba un puntero sería para que pudiéramos modificar los datos a los que apunta el receptor.

se utiliza un receptor de valor para evitar la modificación de los datos que contiene el receptor, un receptor de vara puede causar un impacto en el rendimiento si el receptor es una estructura grande.

```
package main

type User struct {
    ID uint64
    FullName, Email string
}
```

```
// We do no require any special syntax to access field because receiver is a pointer
func (user *User) SendEmail(email string) {
    fmt.Printf("Sent email to: %s\n", user.Email)
}

// ChangeMail will modify the users email because the receiver type is a ponter
func (user *User) ChangeEmail(email string) {
    user.Email = email;
}

func main() {
    user := User{
        1,
        "Zelalem Mekonen",
        "zola.mk.27@gmail.com",
    }

    user.SendEmail("Hello") // Sent email to: zola.mk.27@gmail.com

    user.ChangeEmail("zola@gmail.com")

    fmt.Println(user.Email) // zola@gmail.com
}
```

Interfaz y polimorfismo

Las interfaces proporcionan una manera de especificar el comportamiento de un objeto, si algo puede hacer esto, se puede usar aquí. una interfaz define un conjunto de métodos, pero estos métodos no contienen código, ya que son abstractos o se deja en manos del usuario de la interfaz. a diferencia de la mayoría de los lenguajes, las interfaces orientadas a objetos pueden contener variables en Go.

El polimorfismo es la esencia de la programación orientada a objetos: la capacidad de tratar objetos de diferentes tipos de manera uniforme siempre que se adhieran a la misma interfaz. Las interfaces Go proporcionan esta capacidad de una manera muy directa e intuitiva.

```
package main

type Runner interface {
    Run()
}

type Admin struct {
    Username, Password string
}

func (admin Admin) Run() {
    fmt.Println("Admin ==> Run()");
}

type User struct {
    ID uint64
    FullName, Email string
}

func (user User) Run() {
```

```
    fmt.Println("User ==> Run()")
}

// RunnerExample takes any type that fullfils the Runner interface
func RunnerExample(r Runner) {
    r.Run()
}

func main() {
    admin := Admin{
        "zola",
        "supersecretpassword",
    }

    user := User{
        1,
        "Zelalem Mekonen",
        "zola.mk.27@gmail.com",
    }

    RunnerExample(admin)

    RunnerExample(user)

}
```

Lea Programación orientada a objetos en línea:
https://riptutorial.com/es/go/topic/8801/programacion-orientada-a-objetos

Capítulo 58: Protobuf en Go

Introducción

Protobuf o Protocol Buffer codifica y decodifica datos para que diferentes aplicaciones o módulos escritos en idiomas diferentes puedan intercambiar la gran cantidad de mensajes de forma rápida y confiable sin sobrecargar el canal de comunicación. Con protobuf, el rendimiento es directamente proporcional al número de mensajes que tiende a enviar. Comprime el mensaje para enviarlo en un formato binario serializado al proporcionarle las herramientas para codificar el mensaje en el origen y decodificarlo en el destino.

Observaciones

Hay dos pasos de usar **protobuf** .

1. Primero debes compilar las definiciones del buffer de protocolo.
2. Importe las definiciones anteriores, con la biblioteca de soporte en su programa.

soporte gRPC

Si un archivo de proto especifica servicios RPC, se puede indicar a protoc-gen-go que genere un código compatible con gRPC (http://www.grpc.io/) . Para hacer esto, pasa el parámetro de `plugins` a protoc-gen-go; la forma habitual es insertarlo en el argumento --go_out para protoc:

```
protoc --go_out=plugins=grpc:. *.proto
```

Examples

Usando Protobuf con Go

El mensaje que desea serializar y enviar que puede incluir en un archivo **test.proto** , que contiene

```
package example;

enum FOO { X = 17; };

message Test {
  required string label = 1;
  optional int32 type = 2 [default=77];
  repeated int64 reps = 3;
  optional group OptionalGroup = 4 {
    required string RequiredField = 5;
  }
}
```

Para compilar la definición del búfer de protocolo, ejecute protoc con el parámetro --go_out configurado en el directorio al que desea enviar el código Go.

```
protoc --go_out=. *.proto
```

Para crear y jugar con un objeto de prueba del paquete de ejemplo,

```
package main

import (
    "log"

    "github.com/golang/protobuf/proto"
    "path/to/example"
)

func main() {
    test := &example.Test {
        Label: proto.String("hello"),
        Type:  proto.Int32(17),
        Reps:  []int64{1, 2, 3},
        Optionalgroup: &example.Test_OptionalGroup {
            RequiredField: proto.String("good bye"),
        },
    }
    data, err := proto.Marshal(test)
    if err != nil {
        log.Fatal("marshaling error: ", err)
    }
    newTest := &example.Test{}
    err = proto.Unmarshal(data, newTest)
    if err != nil {
        log.Fatal("unmarshaling error: ", err)
    }
    // Now test and newTest contain the same data.
    if test.GetLabel() != newTest.GetLabel() {
        log.Fatalf("data mismatch %q != %q", test.GetLabel(), newTest.GetLabel())
    }
    // etc.
}
```

Para pasar parámetros adicionales al complemento, use una lista de parámetros separados por comas separados del directorio de salida por dos puntos:

```
protoc --go_out=plugins=grpc,import_path=mypackage:. *.proto
```

Lea Protobuf en Go en línea: https://riptutorial.com/es/go/topic/9729/protobuf-en-go

Capítulo 59: Pruebas

Introducción

Go viene con sus propias instalaciones de prueba que tiene todo lo necesario para ejecutar pruebas y puntos de referencia. A diferencia de la mayoría de los otros lenguajes de programación, a menudo no hay necesidad de un marco de prueba separado, aunque algunos existen.

Examples

Prueba basica

`main.go` :

```
package main

import (
    "fmt"
)

func main() {
    fmt.Println(Sum(4,5))
}

func Sum(a, b int) int {
    return a + b
}
```

`main_test.go` :

```
package main

import (
    "testing"
)

// Test methods start with `Test`
func TestSum(t *testing.T) {
    got := Sum(1, 2)
    want := 3
    if got != want {
        t.Errorf("Sum(1, 2) == %d, want %d", got, want)
    }
}
```

Para ejecutar la prueba solo usa el comando `go test` :

```
$ go test
ok      test_app    0.005s
```

Use la bandera `-v` para ver los resultados de cada prueba:

```
$ go test -v
=== RUN   TestSum
--- PASS: TestSum (0.00s)
PASS
ok      _/tmp    0.000s
```

Use la ruta `./...` para probar los subdirectorios recursivamente:

```
$ go test -v ./...
ok      github.com/me/project/dir1    0.008s
=== RUN   TestSum
--- PASS: TestSum (0.00s)
PASS
ok      github.com/me/project/dir2    0.008s
=== RUN   TestDiff
--- PASS: TestDiff (0.00s)
PASS
```

Ejecutar una prueba particular:
Si hay varias pruebas y desea ejecutar una prueba específica, se puede hacer así:

```
go test -v -run=<TestName> // will execute only test with this name
```

Ejemplo:

```
go test -v run=TestSum
```

Pruebas de referencia

Si desea medir los puntos de referencia, agregue un método de prueba como este:

`sum.go` :

```
package sum

// Sum calculates the sum of two integers
func Sum(a, b int) int {
    return a + b
}
```

`sum_test.go` :

```
package sum

import "testing"

func BenchmarkSum(b *testing.B) {
    for i := 0; i < b.N; i++ {
        _ = Sum(2, 3)
    }
```

```
}
```

Entonces para ejecutar un simple benchmark:

```
$ go test -bench=.
BenchmarkSum-8    2000000000             0.49 ns/op
ok      so/sum    1.027s
```

Pruebas unitarias de mesa

Este tipo de prueba es una técnica popular para realizar pruebas con valores de entrada y salida predefinidos.

Crea un archivo llamado `main.go` con contenido:

```
package main

import (
    "fmt"
)

func main() {
    fmt.Println(Sum(4, 5))
}

func Sum(a, b int) int {
    return a + b
}
```

Después de ejecutarlo con, verá que la salida es `9` . Aunque la función `Sum` parece bastante simple, es una buena idea probar su código. Para hacer esto, creamos otro archivo llamado `main_test.go` en la misma carpeta que `main.go` , que contiene el siguiente código:

```
package main

import (
    "testing"
)

// Test methods start with Test
func TestSum(t *testing.T) {
    // Note that the data variable is of type array of anonymous struct,
    // which is very handy for writing table-driven unit tests.
    data := []struct {
        a, b, res int
    }{
        {1, 2, 3},
        {0, 0, 0},
        {1, -1, 0},
        {2, 3, 5},
        {1000, 234, 1234},
    }

    for _, d := range data {
        if got := Sum(d.a, d.b); got != d.res {
```

```
            t.Errorf("Sum(%d, %d) == %d, want %d", d.a, d.b, got, d.res)
        }
    }
}
```

Como puede ver, se crea un segmento de estructuras anónimas, cada una con un conjunto de entradas y el resultado esperado. Esto permite crear una gran cantidad de casos de prueba en un solo lugar, luego ejecutarse en un bucle, reduciendo la repetición del código y mejorando la claridad.

Pruebas de ejemplo (auto documentar pruebas)

Este tipo de pruebas se aseguran de que su código se compile correctamente y aparecerá en la documentación generada para su proyecto. Además de eso, las pruebas de ejemplo pueden afirmar que su prueba produce un resultado adecuado.

`sum.go` :

```
package sum

// Sum calculates the sum of two integers
func Sum(a, b int) int {
    return a + b
}
```

`sum_test.go` :

```
package sum

import "fmt"

func ExampleSum() {
    x := Sum(1, 2)
    fmt.Println(x)
    fmt.Println(Sum(-1, -1))
    fmt.Println(Sum(0, 0))

    // Output:
    // 3
    // -2
    // 0
}
```

Para ejecutar su prueba, ejecute `go test` en la carpeta que contiene esos archivos O coloque esos dos archivos en una subcarpeta denominada `sum` y luego, desde la carpeta principal, ejecute `go test ./sum` . En ambos casos obtendrás una salida similar a esta:

```
ok      so/sum    0.005s
```

Si se está preguntando cómo está probando su código, aquí hay otra función de ejemplo, que realmente falla la prueba:

```
func ExampleSum_fail() {
    x := Sum(1, 2)
    fmt.Println(x)

    // Output:
    // 5
}
```

Cuando ejecutas `go test` , obtienes el siguiente resultado:

```
$ go test
--- FAIL: ExampleSum_fail (0.00s)
got:
3
want:
5
FAIL
exit status 1
FAIL    so/sum    0.006s
```

Si desea ver la documentación de su paquete de `sum` , simplemente ejecute:

```
go doc -http=:6060
```

y navegue a http: // localhost: 6060 / pkg / FOLDER / sum / , donde *FOLDER* es la carpeta que contiene el paquete de la `sum` (en este ejemplo, `so`). La documentación para el método de suma se ve así:

Package sum

```
import "so/sum"
```


Overview ▾

Package sum is a sample package for test purpos

Index ▾

Examples

Package files

- Una función tearDown hace un rollback.

Esta es una buena opción cuando no puede modificar su base de datos y necesita crear un objeto que simule un objeto traído de la base de datos o necesite iniciar una configuración en cada prueba.

Un ejemplo estúpido sería:

```
// Standard numbers map
var numbers map[string]int = map[string]int{"zero": 0, "three": 3}

// TestMain will exec each test, one by one
func TestMain(m *testing.M) {
    // exec setUp function
    setUp("one", 1)
    // exec test and this returns an exit code to pass to os
    retCode := m.Run()
    // exec tearDown function
    tearDown("one")
    // If exit code is distinct of zero,
    // the test will be failed (red)
    os.Exit(retCode)
}

// setUp function, add a number to numbers slice
func setUp(key string, value int) {
    numbers[key] = value
}

// tearDown function, delete a number to numbers slice
func tearDown(key string) {
    delete(numbers, key)
}

// First test
func TestOnePlusOne(t *testing.T) {
    numbers["one"] = numbers["one"] + 1

    if numbers["one"] != 2 {
        t.Error("1 plus 1 = 2, not %v", value)
    }
}

// Second test
func TestOnePlusTwo(t *testing.T) {
    numbers["one"] = numbers["one"] + 2

    if numbers["one"] != 3 {
        t.Error("1 plus 2 = 3, not %v", value)
    }
}
```

Otro ejemplo sería preparar la base de datos para probar y hacer la reversión

```
 // ID of Person will be saved in database
personID := 12345
// Name of Person will be saved in database
personName := "Toni"
```

```
func TestMain(m *testing.M) {
    // You create an Person and you save in database
    setUp(&Person{
            ID:   personID,
            Name: personName,
            Age:  19,
        })
    retCode := m.Run()
    // When you have executed the test, the Person is deleted from database
    tearDown(personID)
    os.Exit(retCode)
}

func setUp(P *Person) {
    // ...
    db.add(P)
    // ...
}

func tearDown(id int) {
    // ...
    db.delete(id)
    // ...
}

func getPerson(t *testing.T) {
    P := Get(personID)

    if P.Name != personName {
        t.Error("P.Name is %s and it must be Toni", P.Name)
    }
}
```

Ver cobertura de código en formato HTML

Ejecute `go test` como lo hace normalmente, pero con la `coverprofile` . Luego usa la `go tool` para ver los resultados como HTML.

```
    go test -coverprofile=c.out
    go tool cover -html=c.out
```

Lea Pruebas en línea: https://riptutorial.com/es/go/topic/1234/pruebas

Capítulo 60: Punteros

Sintaxis

- puntero: = & variable // obtener puntero de variable
- variable: = * puntero // obtener variable desde puntero
- * puntero = valor // establecer el valor de la variable a través del puntero
- puntero: = nuevo (Struct) // obtener puntero de nueva estructura

Examples

Punteros básicos

Go admite punteros , lo que le permite pasar referencias a valores y registros dentro de su programa.

```
package main

import "fmt"

// We'll show how pointers work in contrast to values with
// 2 functions: `zeroval` and `zeroptr`. `zeroval` has an
// `int` parameter, so arguments will be passed to it by
// value. `zeroval` will get a copy of `ival` distinct
// from the one in the calling function.
func zeroval(ival int) {
    ival = 0
}

// `zeroptr` in contrast has an `*int` parameter, meaning
// that it takes an `int` pointer. The `*iptr` code in the
// function body then _dereferences_ the pointer from its
// memory address to the current value at that address.
// Assigning a value to a dereferenced pointer changes the
// value at the referenced address.
func zeroptr(iptr *int) {
    *iptr = 0
}
```

Una vez que estas funciones están definidas, puedes hacer lo siguiente:

```
func main() {
    i := 1
    fmt.Println("initial:", i) // initial: 1

    zeroval(i)
    fmt.Println("zeroval:", i) // zeroval: 1
    // `i` is still equal to 1 because `zeroval` edited
    // a "copy" of `i`, not the original.

    // The `&i` syntax gives the memory address of `i`,
    // i.e. a pointer to `i`. When calling `zeroptr`,
```

```
    // it will edit the "original" `i`.
    zeroptr(&i)
    fmt.Println("zeroptr:", i) // zeroptr: 0

    // Pointers can be printed too.
    fmt.Println("pointer:", &i) // pointer: 0x10434114
}
```

Prueba este codigo

Puntero v. Métodos de valor

Métodos de puntero

Los métodos de puntero se pueden llamar incluso si la variable no es un puntero.

Según la especificación de Go ,

> . . . una referencia a un método sin interfaz con un receptor de puntero que usa un valor direccionable tomará automáticamente la dirección de ese valor: `t.Mp` es equivalente a `(&t).Mp` .

Puedes ver esto en este ejemplo:

```
package main

import "fmt"

type Foo struct {
    Bar int
}

func (f *Foo) Increment() {
    f.Bar += 1
}

func main() {
    var f Foo

    // Calling `f.Increment` is automatically changed to `(&f).Increment` by the compiler.
    f = Foo{}
    fmt.Printf("f.Bar is %d\n", f.Bar)
    f.Increment()
    fmt.Printf("f.Bar is %d\n", f.Bar)

    // As you can see, calling `(&f).Increment` directly does the same thing.
    f = Foo{}
    fmt.Printf("f.Bar is %d\n", f.Bar)
    (&f).Increment()
    fmt.Printf("f.Bar is %d\n", f.Bar)
}
```

Juegalo

Métodos de valor

De manera similar a los métodos de puntero, los métodos de valor pueden invocarse incluso si la variable en sí misma no es un valor.

Según la especificación de Go ,

> . . . una referencia a un método sin interfaz con un receptor de valores que utiliza un puntero `pt.Mv` automáticamente la referencia de que el puntero: `pt.Mv` es equivalente a `(*pt).Mv` .

Puedes ver esto en este ejemplo:

```
package main

import "fmt"

type Foo struct {
    Bar int
}

func (f Foo) Increment() {
    f.Bar += 1
}

func main() {
    var p *Foo

    // Calling `p.Increment` is automatically changed to `(*p).Increment` by the compiler.
    // (Note that `*p` is going to remain at 0 because a copy of `*p`, and not the original
`*p` are being edited)
    p = &Foo{}
    fmt.Printf("(*p).Bar is %d\n", (*p).Bar)
    p.Increment()
    fmt.Printf("(*p).Bar is %d\n", (*p).Bar)

    // As you can see, calling `(*p).Increment` directly does the same thing.
    p = &Foo{}
    fmt.Printf("(*p).Bar is %d\n", (*p).Bar)
    (*p).Increment()
    fmt.Printf("(*p).Bar is %d\n", (*p).Bar)
}
```

Juegalo

Para obtener más información sobre los métodos de puntero y valor, visite la sección Go Spec en Valores del método , o consulte la sección Effective Go acerca de los punteros v. Valores .

Nota 1: Los paréntesis (`()`) alrededor de `*p` y `&f` antes de los selectores como `.Bar` están allí para propósitos de agrupación, y deben mantenerse.

Nota 2: Aunque los punteros se pueden convertir a valores (y viceversa) cuando son los

receptores de un método, no se convierten automáticamente a otros cuando son argumentos dentro de una función.

Desreferenciación de punteros

Los punteros se pueden **eliminar** de **referencia** agregando un asterisco * antes de un puntero.

```
package main

import (
    "fmt"
)

type Person struct {
    Name string
}

func main() {
    c := new(Person) // returns pointer
    c.Name = "Catherine"
    fmt.Println(c.Name) // prints: Catherine
    d := c
    d.Name = "Daniel"
    fmt.Println(c.Name) // prints: Daniel
    // Adding an Asterix before a pointer dereferences the pointer
    i := *d
    i.Name = "Ines"
    fmt.Println(c.Name) // prints: Daniel
    fmt.Println(d.Name) // prints: Daniel
    fmt.Println(i.Name) // prints: Ines
}
```

Las rebanadas son punteros a segmentos de matriz

Los segmentos son **punteros** a matrices, con la longitud del segmento y su capacidad. Se comportan como punteros, y asignando su valor a otra porción, asignarán la dirección de memoria. Para **copiar** un valor de sector a otro, use la función de **copia** incorporada: `func copy(dst, src []Type) int` (devuelve la cantidad de elementos copiados).

```
package main

import (
    "fmt"
)

func main() {
    x := []byte{'a', 'b', 'c'}
    fmt.Printf("%s", x)       // prints: abc
    y := x
    y[0], y[1], y[2] = 'x', 'y', 'z'
    fmt.Printf("%s", x)       // prints: xyz
    z := make([]byte, len(x))
    // To copy the value to another slice, but
    // but not the memory address use copy:
    _ = copy(z, x)            // returns count of items copied
    fmt.Printf("%s", z)       // prints: xyz
```

```
    z[0], z[1], z[2] = 'a', 'b', 'c'
    fmt.Printf("%s", x)       // prints: xyz
    fmt.Printf("%s", z)       // prints: abc
}
```

Punteros simples

```
func swap(x, y *int) {
  *x, *y = *y, *x
}

func main() {
  x := int(1)
  y := int(2)
  // variable addresses
  swap(&x, &y)
  fmt.Println(x, y)
}
```

Lea Punteros en línea: https://riptutorial.com/es/go/topic/1239/punteros

Capítulo 61: Rebanadas

Introducción

Una porción es una estructura de datos que encapsula una matriz para que el programador pueda agregar tantos elementos como sea necesario sin tener que preocuparse por la administración de la memoria. Las divisiones se pueden cortar en subdivisiones de manera muy eficiente, ya que las divisiones resultantes apuntan a la misma matriz interna. Los programadores de Go a menudo aprovechan esto para evitar copiar arreglos, lo que normalmente se haría en muchos otros lenguajes de programación.

Sintaxis

- slice: = make ([] type, len, cap) // crea un nuevo slice
- slice = append (slice, item) // agregar un elemento a una división
- slice = append (slice, items ...) // anexar slice of items a una porción
- len: = len (slice) // obtiene la longitud de un segmento
- cap: = cap (slice) // obtiene la capacidad de un slice
- elNum: = copy (dst, slice) // copia el contenido de un sector a otro sector

Examples

Anexando a rebanar

```
slice = append(slice, "hello", "world")
```

Sumando dos rebanadas juntas

```
slice1 := []string{"!"}
slice2 := []string{"Hello", "world"}
slice  := append(slice1, slice2...)
```

Correr en el patio de juegos ir

Eliminando elementos / "rebanando" rodajas

Si necesita eliminar uno o más elementos de una división, o si necesita trabajar con una subdivisión de otra existente; Puedes usar el siguiente método.

> Los siguientes ejemplos usan slice de int, pero eso funciona con todo tipo de slice.

Entonces para eso necesitamos una porción, de donde eliminaremos algunos elementos:

```
slice := []int{1, 2, 3, 4, 5, 6}
```

```
// > [1 2 3 4 5 6]
```

Necesitamos también los índices de elementos para eliminar:

```
// index of first element to remove (corresponding to the '3' in the slice)
var first = 2

// index of last element to remove (corresponding to the '5' in the slice)
var last = 4
```

Y así podemos "cortar" la porción, eliminando elementos no deseados:

```
// keeping elements from start to 'first element to remove' (not keeping first to remove),
// removing elements from 'first element to remove' to 'last element to remove'
// and keeping all others elements to the end of the slice
newSlice1 := append(slice[:first], slice[last+1:]...)
// > [1 2 6]

// you can do using directly numbers instead of variables
newSlice2 := append(slice[:2], slice[5:]...)
// > [1 2 6]

// Another way to do the same
newSlice3 := slice[:first + copy(slice[first:], slice[last+1:])]
// > [1 2 6]

// same that newSlice3 with hard coded indexes (without use of variables)
newSlice4 := slice[:2 + copy(slice[2:], slice[5:])]
// > [1 2 6]
```

Para eliminar solo un elemento, solo hay que colocar el índice de este elemento como el primer Y como el último índice que se eliminará, así:

```
var indexToRemove = 3
newSlice5 := append(slice[:indexToRemove], slice[indexToRemove+1:]...)
// > [1 2 3 5 6]

// hard-coded version:
newSlice5 := append(slice[:3], slice[4:]...)
// > [1 2 3 5 6]
```

Y también puedes eliminar elementos desde el principio de la división:

```
newSlice6 := append(slice[:0], slice[last+1:]...)
// > [6]

// That can be simplified into
newSlice6 := slice[last+1:]
// > [6]
```

También puedes eliminar algunos elementos del final de la división:

```
newSlice7 := append(slice[:first], slice[first+1:len(slice)-1]...)
// > [1 2]
```

```
// That can be simplified into
newSlice7 := slice[:first]
// > [1 2]
```

Si la nueva división tiene que contener exactamente los mismos elementos que la primera, puede usar la misma cosa pero con la `last := first-1` .
(Esto puede ser útil en caso de que sus índices se hayan calculado previamente)

Longitud y capacidad

Las rebanadas tienen tanto longitud como capacidad. La longitud de una división es el número de elementos que hay *actualmente* en la división, mientras que la capacidad es la cantidad de elementos que la división *puede contener* antes de que sea necesario reasignarla.

Al crear una división utilizando la función integrada `make()` , puede especificar su longitud y, opcionalmente, su capacidad. Si la capacidad no se especifica explícitamente, será la longitud especificada.

```
var s = make([]int, 3, 5) // length 3, capacity 5
```

Puede verificar la longitud de un sector con la función `len()` incorporada:

```
var n = len(s) // n == 3
```

Puede verificar la capacidad con la función `cap()` incorporada:

```
var c = cap(s) // c == 5
```

Los elementos creados por `make()` se establecen en el valor cero para el tipo de elemento de la división:

```
for idx, val := range s {
    fmt.Println(idx, val)
}
// output:
// 0 0
// 1 0
// 2 0
```

Ejecutalo en play.golang.org

No puede acceder a elementos más allá de la longitud de una porción, incluso si el índice está dentro de la capacidad:

```
var x = s[3] // panic: runtime error: index out of range
```

Sin embargo, siempre que la capacidad exceda la longitud, puede agregar nuevos elementos sin reasignar:

```
var t = []int{3, 4}
s = append(s, t) // s is now []int{0, 0, 0, 3, 4}
n = len(s) // n == 5
c = cap(s) // c == 5
```

Si se agrega a una porción que carece de la capacidad para aceptar los nuevos elementos, la matriz subyacente se reasignará para usted con suficiente capacidad:

```
var u = []int{5, 6}
s = append(s, u) // s is now []int{0, 0, 0, 3, 4, 5, 6}
n = len(s) // n == 7
c = cap(s) // c > 5
```

Por lo tanto, generalmente es una buena práctica asignar suficiente capacidad al crear una porción, si sabe cuánto espacio necesitará, para evitar reasignaciones innecesarias.

Copiando contenidos de una rebanada a otra rebanada

Si desea copiar el contenido de un sector en un sector inicialmente vacío, se pueden seguir los siguientes pasos para lograrlo:

1. Crear la porción de origen:

```
var sourceSlice []interface{} = []interface{}{"Hello",5.10,"World",true}
```

2. Crea la porción de destino, con:

- Longitud = longitud de sourceSlice

```
var destinationSlice []interface{} = make([]interface{},len(sourceSlice))
```

3. Ahora que la matriz subyacente de la porción de destino es lo suficientemente grande para acomodar todos los elementos de la porción de origen, podemos proceder a copiar los elementos utilizando la `copy` incorporada:

```
copy(destinationSlice,sourceSlice)
```

Creando Rebanadas

Los segmentos son la forma típica en que los programadores almacenan las listas de datos.

Para declarar una variable de división, use la sintaxis de `[]Type` .

```
var a []int
```

Para declarar e inicializar una variable de división en una línea, use la sintaxis de `[]Type{values}` .

```
var a []int = []int{3, 1, 4, 1, 5, 9}
```

Otra forma de inicializar un sector es con la función `make` . Tiene tres argumentos: el `Type` de sector (o mapa), la `length` y la `capacity` .

```
a := make([]int, 0, 5)
```

Puedes agregar elementos a tu nueva rebanada usando el `append` .

```
a = append(a, 5)
```

Verifique el número de elementos en su porción usando `len` .

```
length := len(a)
```

Verifica la capacidad de tu rebanada usando la `cap` . La capacidad es el número de elementos asignados actualmente para estar en la memoria para el sector. Siempre se puede agregar a una porción a capacidad, ya que Go creará automáticamente una porción más grande para usted.

```
capacity := cap(a)
```

Puede acceder a los elementos de un sector utilizando la sintaxis de indexación típica.

```
a[0]  // Gets the first member of `a`
```

También puede utilizar un bucle `for` sobre segmentos con `range` . La primera variable es el índice en la matriz especificada, y la segunda variable es el valor para el índice.

```
for index, value := range a {
    fmt.Println("Index: " + index + " Value: " + value)  // Prints "Index: 0 Value: 5" (and
continues until end of slice)
}
```

Ir al patio de recreo

Filtrando una rebanada

Para filtrar una porción sin asignar una nueva matriz subyacente:

```
// Our base slice
slice := []int{ 1, 2, 3, 4 }
// Create a zero-length slice with the same underlying array
tmp := slice[:0]

for _, v := range slice {
  if v % 2 == 0 {
    // Append desired values to slice
    tmp = append(tmp, v)
  }
}

// (Optional) Reassign the slice
```

```
slice = tmp // [2, 4]
```

Valor cero de la rebanada

El valor cero de slice es `nil` , que tiene la longitud y la capacidad `0` . Una porción `nil` no tiene una matriz subyacente. Pero también hay segmentos no nulos de longitud y capacidad `0` , como `[]int{}` o `make([]int, 5)[5:]` .

Cualquier tipo que tenga valores nulos se puede convertir en un segmento `nil` :

```
s = []int(nil)
```

Para probar si una rebanada está vacía, use:

```
if len(s) == 0 {
    fmt.Ptintf("s is empty.")
}
```

Lea Rebanadas en línea: https://riptutorial.com/es/go/topic/733/rebanadas

Capítulo 62: Reflexión

Observaciones

Los documentos `reflect` son una gran referencia. En la programación general de computadoras, la **reflexión** es la capacidad de un programa para **examinar** la estructura y el comportamiento de **sí mismo** en **`runtime`** de **`runtime`** .

Basado en su estricto sistema de `static type` , Go lang tiene algunas reglas (leyes de reflexión)

Examples

Básico reflejar.Valor de uso

```
import "reflect"

value := reflect.ValueOf(4)

// Interface returns an interface{}-typed value, which can be type-asserted
value.Interface().(int) // 4

// Type gets the reflect.Type, which contains runtime type information about
// this value
value.Type().Name() // int

value.SetInt(5) // panics -- non-pointer/slice/array types are not addressable

x := 4
reflect.ValueOf(&x).Elem().SetInt(5) // works
```

Estructuras

```
import "reflect"

type S struct {
    A int
    b string
}

func (s *S) String() { return s.b }

s := &S{
    A: 5,
    b: "example",
}

indirect := reflect.ValueOf(s) // effectively a pointer to an S
value := indirect.Elem()       // this is addressable, since we've derefed a pointer

value.FieldByName("A").Interface() // 5
```

```
value.Field(2).Interface()         // "example"

value.NumMethod()    // 0, since String takes a pointer receiver
indirect.NumMethod() // 1

indirect.Method(0).Call([]reflect.Value{})              // "example"
indirect.MethodByName("String").Call([]reflect.Value{}) // "example"
```

Rebanadas

```
import "reflect"

s := []int{1, 2, 3}

value := reflect.ValueOf(s)

value.Len()                // 3
value.Index(0).Interface() // 1
value.Type().Kind()        // reflect.Slice
value.Type().Elem().Name() // int

value.Index(1).CanAddr()   // true -- slice elements are addressable
value.Index(1).CanSet()    // true -- and settable
value.Index(1).Set(5)

typ := reflect.SliceOf(reflect.TypeOf("example"))
newS := reflect.MakeSlice(typ, 0, 10) // an empty []string{} with capacity 10
```

reflect.Value.Elem ()

```
import "reflect"

// this is effectively a pointer dereference

x := 5
ptr := reflect.ValueOf(&x)
ptr.Type().Name() // *int
ptr.Type().Kind() // reflect.Ptr
ptr.Interface()   // [pointer to x]
ptr.Set(4)        // panic

value := ptr.Elem() // this is a deref
value.Type().Name() // int
value.Type().Kind() // reflect.Int
value.Set(4)        // this works
value.Interface()   // 4
```

Tipo de valor - paquete "reflejar"

reflect.TypeOf se puede usar para verificar el tipo de variables cuando se comparan

```
package main
```

```
import (
    "fmt"
    "reflect"
)
type Data struct {
 a int
}
func main() {
    s:="hey dude"
    fmt.Println(reflect.TypeOf(s))

    D := Data{a:5}
    fmt.Println(reflect.TypeOf(D))

}
```

Salida:
cuerda
datos principales

Lea Reflexión en línea: https://riptutorial.com/es/go/topic/1854/reflexion

Capítulo 63: Seleccione y Canales

Introducción

La palabra clave `select` proporciona un método fácil para trabajar con canales y realizar tareas más avanzadas. Se utiliza con frecuencia para varios propósitos: - Tiempo de espera de manejo. - Cuando hay varios canales para leer, la selección se leerá aleatoriamente desde un canal que tenga datos. - Proporcionar una manera fácil de definir qué sucede si no hay datos disponibles en un canal.

Sintaxis

- seleccione {}
- seleccione {caso verdadero:}
- seleccione {case incomingData: = <-someChannel:}
- seleccione {predeterminado:}

Examples

Simple Seleccione Trabajar con Canales

En este ejemplo, creamos una goroutine (una función que se ejecuta en un subproceso independiente) que acepta un parámetro `chan` , y simplemente realiza un bucle, enviando información al canal cada vez.

En lo `main` tenemos un bucle `for` y un `select` . La `select` bloqueará el procesamiento hasta que una de las declaraciones de `case` convierta en verdadera. Aquí hemos declarado dos casos; la primera es cuando la información llega a través del canal, y la otra es si no ocurre ningún otro caso, lo que se conoce como `default` .

```
// Use of the select statement with channels (no timeouts)
package main

import (
    "fmt"
    "time"
)

// Function that is "chatty"
// Takes a single parameter a channel to send messages down
func chatter(chatChannel chan<- string) {
    // Clean up our channel when we are done.
    // The channel writer should always be the one to close a channel.
    defer close(chatChannel)

    // loop five times and die
    for i := 1; i <= 5; i++ {
        time.Sleep(2 * time.Second) // sleep for 2 seconds
```

```
        chatChannel <- fmt.Sprintf("This is pass number %d of chatter", i)
    }
}

// Our main function
func main() {
    // Create the channel
    chatChannel := make(chan string, 1)

    // start a go routine with chatter (separate, non blocking)
    go chatter(chatChannel)

    // This for loop keeps things going while the chatter is sleeping
    for {
        // select statement will block this thread until one of the two conditions below is
met
        // because we have a default, we will hit default any time the chatter isn't chatting
        select {
        // anytime the chatter chats, we'll catch it and output it
        case spam, ok := <-chatChannel:
            // Print the string from the channel, unless the channel is closed
            // and we're out of data, in which case exit.
            if ok {
                fmt.Println(spam)
            } else {
                fmt.Println("Channel closed, exiting!")
                return
            }
        default:
            // print a line, then sleep for 1 second.
            fmt.Println("Nothing happened this second.")
            time.Sleep(1 * time.Second)
        }
    }
}
```

Pruébalo en el Go Playground!

Usando seleccionar con tiempos de espera

Así que aquí, he eliminado los bucles `for` , e hice un **tiempo** de **espera** agregando un segundo `case` a la `select` que regresa después de 3 segundos. Debido a que la `select` solo espera hasta que CUALQUIER caso sea verdadero, el segundo `case` dispara, y luego nuestro script termina, y el `chatter()` ni siquiera tiene la oportunidad de terminar.

```
// Use of the select statement with channels, for timeouts, etc.
package main

import (
    "fmt"
    "time"
)

// Function that is "chatty"
//Takes a single parameter a channel to send messages down
func chatter(chatChannel chan<- string) {
    // loop ten times and die
    time.Sleep(5 * time.Second) // sleep for 5 seconds
```

```
    chatChannel<- fmt.Sprintf("This is pass number %d of chatter", 1)
}

// out main function
func main() {
    // Create the channel, it will be taking only strings, no need for a buffer on this
project
    chatChannel := make(chan string)
    // Clean up our channel when we are done
    defer close(chatChannel)

    // start a go routine with chatter (separate, no blocking)
    go chatter(chatChannel)

    // select statement will block this thread until one of the two conditions below is met
    // because we have a default, we will hit default any time the chatter isn't chatting
    select {
    // anytime the chatter chats, we'll catch it and output it
    case spam := <-chatChannel:
        fmt.Println(spam)
    // if the chatter takes more than 3 seconds to chat, stop waiting
    case <-time.After(3 * time.Second):
        fmt.Println("Ain't no time for that!")
    }
}
```

Lea Seleccione y Canales en línea: https://riptutorial.com/es/go/topic/3539/seleccione-y-canales

Capítulo 64: Señales OS

Sintaxis

- func Notify (c chan <- os.Signal, sig ... os.Signal)

Parámetros

Parámetro	Detalles
c chan <- os.Signal	`channel` receptor específicamente del tipo `os.Signal` ; creado fácilmente con `sigChan := make(chan os.Signal)`
sig ... os.Signal	Lista de tipos de `os.Signal` para capturar y enviar este `channel` . Consulte https://golang.org/pkg/syscall/#pkg-constants para obtener más opciones.

Examples

Asignar señales a un canal

Muchas veces tendrá motivos para detectar cuándo el sistema operativo le indica a su programa que se detenga en el sistema operativo y tome algunas medidas para preservar el estado o limpiar su aplicación. Para lograr esto, puede usar el paquete `os/signal` de la biblioteca estándar. A continuación se muestra un ejemplo simple de asignación de todas las señales del sistema a un canal, y luego cómo reaccionar ante esas señales.

```
package main

import (
    "fmt"
    "os"
    "os/signal"
)

func main() {
    // create a channel for os.Signal
    sigChan := make(chan os.Signal)

    // assign all signal notifications to the channel
    signal.Notify(sigChan)

    // blocks until you get a signal from the OS
    select {
    // when a signal is received
    case sig := <-sigChan:
        // print this line telling us which signal was seen
        fmt.Println("Received signal from OS:", sig)
    }
}
```

Cuando ejecute el script anterior, creará un canal y luego lo bloqueará hasta que ese canal reciba una señal.

```
$ go run signals.go
^CReceived signal from OS: interrupt
```

La `^C` anterior es el comando de teclado `CTRL+C` que envía la señal `SIGINT` .

Lea Señales OS en línea: https://riptutorial.com/es/go/topic/4497/senales-os

Capítulo 65: Servidor HTTP

Observaciones

`http.ServeMux` proporciona un multiplexor que llama a los controladores para las solicitudes HTTP.

Las alternativas al multiplexor de biblioteca estándar incluyen:

- Gorila mux

Examples

HTTP Hello World con servidor personalizado y mux

```
package main

import (
    "log"
    "net/http"
)

func main() {

    // Create a mux for routing incoming requests
    m := http.NewServeMux()

    // All URLs will be handled by this function
    m.HandleFunc("/", func(w http.ResponseWriter, r *http.Request) {
        w.Write([]byte("Hello, world!"))
    })

    // Create a server listening on port 8000
    s := &http.Server{
        Addr:    ":8000",
        Handler: m,
    }

    // Continue to process new requests until an error occurs
    log.Fatal(s.ListenAndServe())
}
```

Presione `Ctrl` + `C` para detener el proceso.

Hola Mundo

La forma típica de comenzar a escribir servidores web en golang es usar el módulo `net/http` biblioteca estándar.

También hay un tutorial para ello aquí .

El siguiente código también lo usa. Aquí está la implementación de servidor HTTP más simple

posible. Responde `"Hello World"` a cualquier solicitud HTTP.

Guarde el siguiente código en un archivo `server.go` en sus `server.go` de trabajo.

```
package main

import (
    "log"
    "net/http"
)

func main() {
    // All URLs will be handled by this function
    // http.HandleFunc uses the DefaultServeMux
    http.HandleFunc("/", func(w http.ResponseWriter, r *http.Request) {
        w.Write([]byte("Hello, world!"))
    })

    // Continue to process new requests until an error occurs
    log.Fatal(http.ListenAndServe(":8080", nil))
}
```

Puede ejecutar el servidor utilizando:

```
$ go run server.go
```

O puedes compilar y correr.

```
$ go build server.go
$ ./server
```

El servidor escuchará el puerto especificado (`:8080`). Puedes probarlo con cualquier cliente HTTP. Aquí hay un ejemplo con `cURL` :

```
curl -i http://localhost:8080/
HTTP/1.1 200 OK
Date: Wed, 20 Jul 2016 18:04:46 GMT
Content-Length: 13
Content-Type: text/plain; charset=utf-8

Hello, world!
```

Presione `Ctrl` + `C` para detener el proceso.

Usando una función de manejador

`HandleFunc` registra la función del controlador para el patrón dado en el servidor mux (enrutador).

Puede pasar a definir una función anónima, como hemos visto en el ejemplo básico de *Hello World* :

```
http.HandleFunc("/", func(w http.ResponseWriter, r *http.Request) {
    fmt.Fprintln(w, "Hello, world!")
```

```
}
```

Pero también podemos pasar un tipo `HandlerFunc` . En otras palabras, podemos pasar cualquier función que respete la siguiente firma:

```
func FunctionName(w http.ResponseWriter, req *http.Request)
```

Podemos reescribir el ejemplo anterior pasando la referencia a un `HandlerFunc` previamente definido. Aquí está el ejemplo completo:

```
package main

import (
    "fmt"
    "net/http"
)

// A HandlerFunc function
// Notice the signature of the function
func RootHandler(w http.ResponseWriter, req *http.Request) {
    fmt.Fprintln(w, "Hello, world!")
}

func main() {
    // Here we pass the reference to the `RootHandler` handler function
    http.HandleFunc("/", RootHandler)
    panic(http.ListenAndServe(":8080", nil))
}
```

Por supuesto, puede definir varios manejadores de funciones para diferentes rutas.

```
package main

import (
    "fmt"
    "log"
    "net/http"
)

func FooHandler(w http.ResponseWriter, req *http.Request) {
    fmt.Fprintln(w, "Hello from foo!")
}

func BarHandler(w http.ResponseWriter, req *http.Request) {
    fmt.Fprintln(w, "Hello from bar!")
}

func main() {
    http.HandleFunc("/foo", FooHandler)
    http.HandleFunc("/bar", BarHandler)

    log.Fatal(http.ListenAndServe(":8080", nil))
}
```

Aquí está la salida usando `cURL` :

```
➜  ~ curl -i localhost:8080/foo
HTTP/1.1 200 OK
Date: Wed, 20 Jul 2016 18:23:08 GMT
Content-Length: 16
Content-Type: text/plain; charset=utf-8

Hello from foo!

➜  ~ curl -i localhost:8080/bar
HTTP/1.1 200 OK
Date: Wed, 20 Jul 2016 18:23:10 GMT
Content-Length: 16
Content-Type: text/plain; charset=utf-8

Hello from bar!

➜  ~ curl -i localhost:8080/
HTTP/1.1 404 Not Found
Content-Type: text/plain; charset=utf-8
X-Content-Type-Options: nosniff
Date: Wed, 20 Jul 2016 18:23:13 GMT
Content-Length: 19

404 page not found
```

Crear un servidor HTTPS

Generar un certificado

Para ejecutar un servidor HTTPS, es necesario un certificado. La generación de un certificado autofirmado con `openssl` se realiza ejecutando este comando:

```
openssl req -x509 -newkey rsa:4096 -sha256 -nodes -keyout key.pem -out cert.pem -subj
"/CN=example.com" -days 3650`
```

Los parámetros son:

- `req` Usa la herramienta de solicitud de certificado
- `x509` crea un certificado autofirmado
- `newkey rsa:4096` Crea una nueva clave y certificado utilizando los algoritmos RSA con una longitud de clave de `4096` bits
- `sha256` Fuerza los algoritmos de hash SHA256 que los principales navegadores consideran seguros (en el año 2017)
- `nodes` Desactiva la protección de contraseña para la clave privada. Sin este parámetro, su servidor tuvo que pedirle la contraseña cada vez que se inicia.
- `keyout` Nombra el archivo donde escribir la clave
- `out` los nombres del archivo donde escribir el certificado
- `subj` define el nombre de dominio para el cual este certificado es válido
- `days` Fow ¿cuántos días debe ser válido este certificado? `3650` son aprox. 10 años.

Nota: Se puede usar un certificado autofirmado, por ejemplo, para proyectos internos, depuración,

pruebas, etc. Cualquier navegador por ahí mencionará que este certificado no es seguro. Para evitar esto, el certificado debe estar firmado por una autoridad de certificación. En su mayoría, esto no está disponible de forma gratuita. Una excepción es el movimiento "Let's Encrypt": https://letsencrypt.org

El código Go necesario

Puede manejar la configuración de TLS para el servidor con el siguiente código. `cert.pem` y `key.pem` son su certificado y clave SSL, que se generaron con el comando anterior.

```
package main

import (
    "log"
    "net/http"
)

func main() {
    http.HandleFunc("/", func(w http.ResponseWriter, r *http.Request) {
        w.Write([]byte("Hello, world!"))
    })

    log.Fatal(http.ListenAndServeTLS(":443","cert.pem","key.pem", nil))
}
```

Respondiendo a una solicitud HTTP usando plantillas

Las respuestas se pueden escribir en un `http.ResponseWriter` usando plantillas en Go. Esto demuestra como una herramienta útil si desea crear páginas dinámicas.

(Para saber cómo funcionan las plantillas en Go, visite la página de documentación de Go Templates).

Continuando con un ejemplo simple para utilizar el `html/template` para responder a una solicitud HTTP:

```
package main

import(
    "html/template"
    "net/http"
    "log"
)

func main(){
    http.HandleFunc("/",WelcomeHandler)
    http.ListenAndServe(":8080",nil)
}

type User struct{
    Name string
    nationality string //unexported field.
}
```

```
func check(err error){
    if err != nil{
        log.Fatal(err)
    }
}

func WelcomeHandler(w http.ResponseWriter, r *http.Request){
    if r.Method == "GET"{
        t,err := template.ParseFiles("welcomeform.html")
        check(err)
        t.Execute(w,nil)
    }else{
        r.ParseForm()
        myUser := User{}
        myUser.Name = r.Form.Get("entered_name")
        myUser.nationality = r.Form.Get("entered_nationality")
        t, err := template.ParseFiles("welcomeresponse.html")
        check(err)
        t.Execute(w,myUser)
    }
}
```

Donde, los contenidos de

1. `welcomeform.html` son:

```
<head>
    <title> Help us greet you </title>
</head>
<body>
    <form method="POST" action="/">
        Enter Name: <input type="text" name="entered_name">
        Enter Nationality: <input type="text" name="entered_nationality">
        <input type="submit" value="Greet me!">
    </form>
</body>
```

1. `welcomeresponse.html` son:

```
<head>
    <title> Greetings, {{.Name}} </title>
</head>
<body>
    Greetings, {{.Name}}.<br>
    We know you are a {{.nationality}}!
</body>
```

Nota:

1. Asegúrese de que los archivos `.html` estén en el directorio correcto.
2. Cuando se puede visitar `http://localhost:8080/` después de iniciar el servidor.
3. Como se puede ver después de enviar el formulario, el paquete de plantilla no pudo analizar el campo de nacionalidad no *exportado* de la estructura, como se esperaba.

Sirviendo contenido usando ServeMux

Un simple servidor de archivos estáticos se vería así:

```
package main

import (
    "net/http"
)

func main() {
    muxer := http.NewServeMux()
    fileServerCss := http.FileServer(http.Dir("src/css"))
    fileServerJs := http.FileServer(http.Dir("src/js"))
    fileServerHtml := http.FileServer(http.Dir("content"))
    muxer.Handle("/", fileServerHtml)
    muxer.Handle("/css", fileServerCss)
    muxer.Handle("/js", fileServerJs)
    http.ListenAndServe(":8080", muxer)
}
```

Manejo del método http, acceso a cadenas de consulta y cuerpo de solicitud

Aquí hay un ejemplo simple de algunas tareas comunes relacionadas con el desarrollo de una API, diferenciando el Método HTTP de la solicitud, accediendo a los valores de cadena de consulta y accediendo al cuerpo de la solicitud.

Recursos

- interfaz http.Handler
- http.ResponseWriter
- http.Request
- Método disponible y constantes de estado

```
package main

import (
    "fmt"
    "io/ioutil"
    "log"
    "net/http"
)

type customHandler struct{}

// ServeHTTP implements the http.Handler interface in the net/http package
func (h customHandler) ServeHTTP(w http.ResponseWriter, r *http.Request) {

    // ParseForm will parse query string values and make r.Form available
    r.ParseForm()

    // r.Form is map of query string parameters
    // its' type is url.Values, which in turn is a map[string][]string
    queryMap := r.Form
```

```
    switch r.Method {
    case http.MethodGet:
        // Handle GET requests
        w.WriteHeader(http.StatusOK)
        w.Write([]byte(fmt.Sprintf("Query string values: %s", queryMap)))
        return
    case http.MethodPost:
        // Handle POST requests
        body, err := ioutil.ReadAll(r.Body)
        if err != nil {
            // Error occurred while parsing request body
            w.WriteHeader(http.StatusBadRequest)
            return
        }
        w.WriteHeader(http.StatusOK)
        w.Write([]byte(fmt.Sprintf("Query string values: %s\nBody posted: %s", queryMap,
body)))
        return
    }

    // Other HTTP methods (eg PUT, PATCH, etc) are not handled by the above
    // so inform the client with appropriate status code
    w.WriteHeader(http.StatusMethodNotAllowed)
}

func main() {
    // All URLs will be handled by this function
    // http.Handle, similarly to http.HandleFunc
    // uses the DefaultServeMux
    http.Handle("/", customHandler{})

    // Continue to process new requests until an error occurs
    log.Fatal(http.ListenAndServe(":8080", nil))
}
```

Ejemplo de salida de rizo:

```
$ curl -i 'localhost:8080?city=Seattle&state=WA' -H 'Content-Type: text/plain' -X GET
HTTP/1.1 200 OK
Date: Fri, 02 Sep 2016 16:36:24 GMT
Content-Length: 51
Content-Type: text/plain; charset=utf-8

Query string values: map[city:[Seattle] state:[WA]]%

$ curl -i 'localhost:8080?city=Seattle&state=WA' -H 'Content-Type: text/plain' -X POST -d
"some post data"
HTTP/1.1 200 OK
Date: Fri, 02 Sep 2016 16:36:35 GMT
Content-Length: 79
Content-Type: text/plain; charset=utf-8

Query string values: map[city:[Seattle] state:[WA]]
Body posted: some post data%

$ curl -i 'localhost:8080?city=Seattle&state=WA' -H 'Content-Type: text/plain' -X PUT
HTTP/1.1 405 Method Not Allowed
Date: Fri, 02 Sep 2016 16:36:41 GMT
Content-Length: 0
Content-Type: text/plain; charset=utf-8
```

Lea Servidor HTTP en línea: https://riptutorial.com/es/go/topic/756/servidor-http

Capítulo 66: SQL

Observaciones

Para obtener una lista de los controladores de base de datos SQL, consulte el artículo oficial de Go wiki *SQLDrivers* .

Los controladores SQL son importados y prefijados por _ , de modo que *solo* están disponibles para el controlador.

Examples

Preguntando

Este ejemplo muestra cómo consultar una base de datos con `database/sql` , tomando como ejemplo una base de datos MySql.

```
package main

import (
    "log"
    "fmt"
    "database/sql"
    _ "github.com/go-sql-driver/mysql"
)

func main() {
    dsn := "mysql_username:CHANGEME@tcp(localhost:3306)/dbname"

    db, err := sql.Open("mysql", dsn)
    if err != nil {
        log.Fatal(err)
    }
    defer sql.Close()

    rows, err := db.Query("select id, first_name from user limit 10")
    if err != nil {
        log.Fatal(err)
    }
    defer rows.Close()

    for rows.Next() {
        var id int
        var username string
        if err := rows.Scan(&id, &username); err != nil {
            log.Fatal(err)
        }
        fmt.Printf("%d-%s\n", id, username)
    }
}
```

MySQL

Para habilitar MySQL, se necesita un controlador de base de datos. Por ejemplo github.com/go-sql-driver/mysql .

```
import (
    "database/sql"
    _ "github.com/go-sql-driver/mysql"
)
```

Abriendo una base de datos

La apertura de una base de datos es específica de la base de datos, aquí hay ejemplos de algunas bases de datos.

Sqlite 3

```
file := "path/to/file"
db_, err := sql.Open("sqlite3", file)
if err != nil {
    panic(err)
}
```

MySql

```
dsn := "mysql_username:CHANGEME@tcp(localhost:3306)/dbname"
db, err := sql.Open("mysql", dsn)
if err != nil {
    panic(err)
}
```

MongoDB: conectar y insertar y eliminar y actualizar y consultar

```
package main

import (
    "fmt"
    "time"

    log "github.com/Sirupsen/logrus"
    mgo "gopkg.in/mgo.v2"
    "gopkg.in/mgo.v2/bson"
)

var mongoConn *mgo.Session

type MongoDB_Conn struct {
    Host string `json:"Host"`
    Port string `json:"Port"`
    User string `json:"User"`
    Pass string `json:"Pass"`
    DB   string `json:"DB"`
}

func MongoConn(mdb MongoDB_Conn) (*mgo.Session, string, error) {
    if mongoConn != nil {
```

```
        if mongoConn.Ping() == nil {
            return mongoConn, nil
        }
    }
    user := mdb.User
    pass := mdb.Pass
    host := mdb.Host
    port := mdb.Port
    db := mdb.DB
    if host == "" || port == "" || db == "" {
        log.Fatal("Host or port or db is nil")
    }
    url := fmt.Sprintf("mongodb://%s:%s@%s:%s/%s", user, pass, host, port, db)
    if user == "" {
        url = fmt.Sprintf("mongodb://%s:%s/%s", host, port, db)
    }
    mongo, err := mgo.DialWithTimeout(url, 3*time.Second)
    if err != nil {
        log.Errorf("Mongo Conn Error: [%v], Mongo ConnUrl: [%v]",
            err, url)
        errTextReturn := fmt.Sprintf("Mongo Conn Error: [%v]", err)
        return &mgo.Session{}, errors.New(errTextReturn)
    }
    mongoConn = mongo
    return mongoConn, nil
}

func MongoInsert(dbName, C string, data interface{}) error {
    mongo, err := MongoConn()
    if err != nil {
        log.Error(err)
        return err
    }
    db := mongo.DB(dbName)
    collection := db.C(C)
    err = collection.Insert(data)
    if err != nil {
        return err
    }
    return nil
}

func MongoRemove(dbName, C string, selector bson.M) error {
    mongo, err := MongoConn()
    if err != nil {
        log.Error(err)
        return err
    }
    db := mongo.DB(dbName)
    collection := db.C(C)
    err = collection.Remove(selector)
    if err != nil {
        return err
    }
    return nil
}

func MongoFind(dbName, C string, query, selector bson.M) ([]interface{}, error) {
    mongo, err := MongoConn()
    if err != nil {
        return nil, err
```

```
    }
    db := mongo.DB(dbName)
    collection := db.C(C)
    result := make([]interface{}, 0)
    err = collection.Find(query).Select(selector).All(&result)
    return result, err
}

func MongoUpdate(dbName, C string, selector bson.M, update interface{}) error {
    mongo, err := MongoConn()
    if err != nil {
        log.Error(err)
        return err
    }
    db := mongo.DB(dbName)
    collection := db.C(C)
    err = collection.Update(selector, update)
    if err != nil {
        return err
    }
    return nil
}
```

Lea SQL en línea: https://riptutorial.com/es/go/topic/1273/sql

Capítulo 67: Texto + HTML Plantillas

Examples

Plantilla de elemento único

Tenga en cuenta el uso de `{{.}}` Para generar el elemento dentro de la plantilla.

```
package main

import (
    "fmt"
    "os"
    "text/template"
)

func main() {
    const (
        letter = `Dear {{.}}, How are you?`
    )

    tmpl, err := template.New("letter").Parse(letter)
    if err != nil {
        fmt.Println(err.Error())
    }

    tmpl.Execute(os.Stdout, "Professor Jones")
}
```

Resultados en:

```
Dear Professor Jones, How are you?
```

Plantilla de elemento múltiple

Tenga en cuenta el uso de `{{range .}}` Y `{{end}}` para desplazarse por la colección.

```
package main

import (
    "fmt"
    "os"
    "text/template"
)

func main() {
    const (
        letter = `Dear {{range .}}{{.}}, {{end}} How are you?`
    )

    tmpl, err := template.New("letter").Parse(letter)
    if err != nil {
        fmt.Println(err.Error())
```

```
    }

    tmpl.Execute(os.Stdout, []string{"Harry", "Jane", "Lisa", "George"})
}
```

Resultados en:

```
Dear Harry, Jane, Lisa, George,  How are you?
```

Plantillas con lógica personalizada.

En este ejemplo, un mapa de función denominado `funcMap` se suministra a la plantilla a través del método `Funcs()` y luego se invoca dentro de la plantilla. Aquí, la función `increment()` se usa para evitar la falta de una función menor o igual en el lenguaje de plantillas. Observe en la salida cómo se maneja el artículo final en la colección.

A `-` al principio `{{-` o end `-}}` se usa para recortar espacios en blanco y se puede usar para ayudar a que la plantilla sea más legible.

```
package main

import (
    "fmt"
    "os"
    "text/template"
)

var funcMap = template.FuncMap{
    "increment": increment,
}

func increment(x int) int {
    return x + 1
}

func main() {
    const (
        letter = `Dear {{with $names := .}}
        {{- range $i, $val := $names}}
            {{- if lt (increment $i) (len $names)}}
                {{- $val}}, {{else -}} and {{$val}}{{end}}
        {{- end}}{{end}}; How are you?`
    )

    tmpl, err := template.New("letter").Funcs(funcMap).Parse(letter)
    if err != nil {
        fmt.Println(err.Error())
    }

    tmpl.Execute(os.Stdout, []string{"Harry", "Jane", "Lisa", "George"})
}
```

Resultados en:

```
Dear Harry, Jane, Lisa, and George; How are you?
```

Plantillas con estructuras

Observe cómo se obtienen los valores de campo utilizando `{{.FieldName}}`.

```
package main

import (
    "fmt"
    "os"
    "text/template"
)

type Person struct {
    FirstName string
    LastName  string
    Street    string
    City      string
    State     string
    Zip       string
}

func main() {
    const (
        letter = `------------------------------
{{range .}}{{.FirstName}} {{.LastName}}
{{.Street}}
{{.City}}, {{.State}} {{.Zip}}

Dear {{.FirstName}},
    How are you?

------------------------------
{{end}}`
    )

    tmpl, err := template.New("letter").Parse(letter)
    if err != nil {
        fmt.Println(err.Error())
    }

    harry := Person{
        FirstName: "Harry",
        LastName:  "Jones",
        Street:    "1234 Main St.",
        City:      "Springfield",
        State:     "IL",
        Zip:       "12345-6789",
    }

    jane := Person{
        FirstName: "Jane",
        LastName:  "Sherman",
        Street:    "8511 1st Ave.",
        City:      "Dayton",
        State:     "OH",
        Zip:       "18515-6261",
    }

    tmpl.Execute(os.Stdout, []Person{harry, jane})
}
```

Resultados en:

```
------------------------------
Harry Jones
1234 Main St.
Springfield, IL 12345-6789

Dear Harry,
    How are you?

------------------------------
Jane Sherman
8511 1st Ave.
Dayton, OH 18515-6261

Dear Jane,
    How are you?

------------------------------
```

Plantillas HTML

Tenga en cuenta la importación de diferentes paquetes.

```
package main

import (
    "fmt"
    "html/template"
    "os"
)

type Person struct {
    FirstName string
    LastName  string
    Street    string
    City      string
    State     string
    Zip       string
    AvatarUrl string
}

func main() {
    const (
        letter = `<html><body><table>
<tr><th></th><th>Name</th><th>Address</th></tr>
{{range .}}
<tr>
<td><img src="{{.AvatarUrl}}"></td>
<td>{{.FirstName}} {{.LastName}}</td>
<td>{{.Street}}, {{.City}}, {{.State}} {{.Zip}}</td>
</tr>
{{end}}
</table></body></html>`
    )

    tmpl, err := template.New("letter").Parse(letter)
    if err != nil {
```

```
        fmt.Println(err.Error())
    }

    harry := Person{
        FirstName: "Harry",
        LastName:  "Jones",
        Street:    "1234 Main St.",
        City:      "Springfield",
        State:     "IL",
        Zip:       "12345-6789",
        AvatarUrl: "harry.png",
    }

    jane := Person{
        FirstName: "Jane",
        LastName:  "Sherman",
        Street:    "8511 1st Ave.",
        City:      "Dayton",
        State:     "OH",
        Zip:       "18515-6261",
        AvatarUrl: "jane.png",
    }

    tmpl.Execute(os.Stdout, []Person{harry, jane})
}
```

Resultados en:

```
<html><body><table>
<tr><th></th><th>Name</th><th>Address</th></tr>

<tr>
<td><img src="harry.png"></td>
<td>Harry Jones</td>
<td>1234 Main St., Springfield, IL 12345-6789</td>
</tr>

<tr>
<td><img src="jane.png"></td>
<td>Jane Sherman</td>
<td>8511 1st Ave., Dayton, OH 18515-6261</td>
</tr>

</table></body></html>
```

Cómo las plantillas HTML evitan la inyección de código malicioso

Primero, esto es lo que puede suceder cuando se usa `text/template` para HTML. Tenga en cuenta la propiedad `FirstName` de Harry).

```
package main

import (
    "fmt"
    "html/template"
    "os"
)
```

```
type Person struct {
    FirstName string
    LastName  string
    Street    string
    City      string
    State     string
    Zip       string
    AvatarUrl string
}

func main() {
    const (
        letter = `<html><body><table>
<tr><th></th><th>Name</th><th>Address</th></tr>
{{range .}}
<tr>
<td><img src="{{.AvatarUrl}}"></td>
<td>{{.FirstName}} {{.LastName}}</td>
<td>{{.Street}}, {{.City}}, {{.State}} {{.Zip}}</td>
</tr>
{{end}}
</table></body></html>`
    )

    tmpl, err := template.New("letter").Parse(letter)
    if err != nil {
        fmt.Println(err.Error())
    }

    harry := Person{
        FirstName: `Harry<script>alert("You've been hacked!")</script>`,
        LastName:  "Jones",
        Street:    "1234 Main St.",
        City:      "Springfield",
        State:     "IL",
        Zip:       "12345-6789",
        AvatarUrl: "harry.png",
    }

    jane := Person{
        FirstName: "Jane",
        LastName:  "Sherman",
        Street:    "8511 1st Ave.",
        City:      "Dayton",
        State:     "OH",
        Zip:       "18515-6261",
        AvatarUrl: "jane.png",
    }

    tmpl.Execute(os.Stdout, []Person{harry, jane})
}
```

Resultados en:

```
<html><body><table>
<tr><th></th><th>Name</th><th>Address</th></tr>

<tr>
<td><img src="harry.png"></td>
```

```
<td>Harry<script>alert("You've been hacked!")</script> Jones</td>
<td>1234 Main St., Springfield, IL 12345-6789</td>
</tr>

<tr>
<td><img src="jane.png"></td>
<td>Jane Sherman</td>
<td>8511 1st Ave., Dayton, OH 18515-6261</td>
</tr>

</table></body></html>
```

El ejemplo anterior, si se accede desde un navegador, daría como resultado que el script se ejecute y se genere una alerta. Si, en cambio, se importó el `html/template` lugar de `text/template` , la secuencia de comandos se sanearía de forma segura:

```
<html><body><table>
<tr><th></th><th>Name</th><th>Address</th></tr>

<tr>
<td><img src="harry.png"></td>
<td>Harry&lt;script&gt;alert(&#34;You&#39;ve been hacked!&#34;)&lt;/script&gt; Jones</td>
<td>1234 Main St., Springfield, IL 12345-6789</td>
</tr>

<tr>
<td><img src="jane.png"></td>
<td>Jane Sherman</td>
<td>8511 1st Ave., Dayton, OH 18515-6261</td>
</tr>

</table></body></html>
```

El segundo resultado se vería confuso cuando se cargaba en un navegador, pero no daría como resultado la ejecución de un script potencialmente malicioso.

Lea Texto + HTML Plantillas en línea: https://riptutorial.com/es/go/topic/3888/texto-plus-html-plantillas

Capítulo 68: Tipo de conversiones

Examples

Conversión de tipo básico

Hay dos estilos básicos de conversión de tipos en Go:

```
// Simple type conversion
var x := Foo{}    // x is of type Foo
var y := (Bar)Foo // y is of type Bar, unless Foo cannot be cast to Bar, then compile-time
error occurs.
// Extended type conversion
var z,ok := x.(Bar)    // z is of type Bar, ok is of type bool - if conversion succeeded, z
has the same value as x and ok is true. If it failed, z has the zero value of type Bar, and ok
is false.
```

Implementación de la interfaz de prueba

Como Go utiliza la implementación de la interfaz implícita, no obtendrá un error en tiempo de compilación si su estructura no implementa una interfaz que tenía la intención de implementar. Puede probar la implementación explícitamente utilizando conversión de tipos: escriba la interfaz de MyInterface {Thing ()}

```
type MyImplementer struct {}

func (m MyImplementer) Thing() {
    fmt.Println("Huzzah!")
}

// Interface is implemented, no error. Variable name _ causes value to be ignored.
var _ MyInterface = (*MyImplementer)nil

type MyNonImplementer struct {}

// Compile-time error - cannot case because interface is not implemented.
var _ MyInterface = (*MyNonImplementer)nil
```

Implementar un sistema de unidades con tipos

Este ejemplo ilustra cómo se puede usar el sistema de tipos de Go para implementar algún sistema de unidades.

```
package main

import (
    "fmt"
)

type MetersPerSecond float64
```

```
type KilometersPerHour float64

func (mps MetersPerSecond) toKilometersPerHour() KilometersPerHour {
    return KilometersPerHour(mps * 3.6)
}

func (kmh KilometersPerHour) toMetersPerSecond() MetersPerSecond {
    return MetersPerSecond(kmh / 3.6)
}

func main() {
    var mps MetersPerSecond
    mps = 12.5
    kmh := mps.toKilometersPerHour()
    mps2 := kmh.toMetersPerSecond()
    fmt.Printf("%vmps = %vkmh = %vmps\n", mps, kmh, mps2)
}
```

Abrir en el patio

Lea Tipo de conversiones en línea: https://riptutorial.com/es/go/topic/2851/tipo-de-conversiones

Capítulo 69: trozo

Introducción

Gob es un método de serialización específico de Go. Tiene soporte para todos los tipos de datos Go, excepto para canales y funciones. Gob también codifica la información de tipo en la forma serializada, lo que lo hace diferente de, digamos, XML es que es mucho más eficiente.

La inclusión de información de tipo hace que la codificación y la decodificación sean bastante robustas a las diferencias entre el codificador y el decodificador.

Examples

¿Cómo codificar los datos y escribir en un archivo con gob?

```
package main

import (
    "encoding/gob"
    "os"
)

type User struct {
    Username string
    Password string
}

func main() {

    user := User{
        "zola",
        "supersecretpassword",
    }

    file, _ := os.Create("user.gob")

    defer file.Close()

    encoder := gob.NewEncoder(file)

    encoder.Encode(user)

}
```

¿Cómo leer datos de archivo y decodificar con go?

```
package main

import (
    "encoding/gob"
    "fmt"
    "os"
```

```
)

type User struct {
    Username string
    Password string
}

func main() {

    user := User{}

    file, _ := os.Open("user.gob")

    defer file.Close()

    decoder := gob.NewDecoder(file)

    decoder.Decode(&user)

    fmt.Println(user)

}
```

¿Cómo codificar una interfaz con gob?

```
package main

import (
    "encoding/gob"
    "fmt"
    "os"
)

type User struct {
    Username string
    Password string
}

type Admin struct {
    Username string
    Password string
    IsAdmin  bool
}

type Deleter interface {
    Delete()
}

func (u User) Delete() {
    fmt.Println("User ==> Delete()")
}

func (a Admin) Delete() {
    fmt.Println("Admin ==> Delete()")
}

func main() {

    user := User{
```

```
        "zola",
        "supersecretpassword",
    }

    admin := Admin{
        "john",
        "supersecretpassword",
        true,
    }

    file, _ := os.Create("user.gob")

    adminFile, _ := os.Create("admin.gob")

    defer file.Close()

    defer adminFile.Close()

    gob.Register(User{}) // registering the type allows us to encode it

    gob.Register(Admin{}) // registering the type allows us to encode it

    encoder := gob.NewEncoder(file)

    adminEncoder := gob.NewEncoder(adminFile)

    InterfaceEncode(encoder, user)

    InterfaceEncode(adminEncoder, admin)

}

func InterfaceEncode(encoder *gob.Encoder, d Deleter) {

    if err := encoder.Encode(&d); err != nil {
        fmt.Println(err)
    }

}
```

¿Cómo decodificar una interfaz con gob?

```
package main

import (
    "encoding/gob"
    "fmt"
    "log"
    "os"
)

type User struct {
    Username string
    Password string
}

type Admin struct {
    Username string
    Password string
```

```
    IsAdmin  bool
}

type Deleter interface {
    Delete()
}

func (u User) Delete() {
    fmt.Println("User ==> Delete()")
}

func (a Admin) Delete() {
    fmt.Println("Admin ==> Delete()")
}

func main() {

    file, _ := os.Open("user.gob")

    adminFile, _ := os.Open("admin.gob")

    defer file.Close()

    defer adminFile.Close()

    gob.Register(User{}) // registering the type allows us to encode it

    gob.Register(Admin{}) // registering the type allows us to encode it

    var admin Deleter

    var user Deleter

    userDecoder := gob.NewDecoder(file)

    adminDecoder := gob.NewDecoder(adminFile)

    user = InterfaceDecode(userDecoder)

    admin = InterfaceDecode(adminDecoder)

    fmt.Println(user)

    fmt.Println(admin)

}

func InterfaceDecode(decoder *gob.Decoder) Deleter {

    var d Deleter

    if err := decoder.Decode(&d); err != nil {
        log.Fatal(err)
    }

    return d

}
```

Lea trozo en línea: https://riptutorial.com/es/go/topic/8820/trozo

Capítulo 70: Valores cero

Observaciones

Una cosa a tener en cuenta: los tipos que tienen un valor cero no nulo como cadenas, ints, flotantes, bools y estructuras no se pueden establecer en nil.

Examples

Valores básicos de cero

Las variables en Go siempre se inicializan, ya sea que les de un valor de inicio o no. Cada tipo, incluidos los tipos personalizados, tiene un valor cero al que están establecidos si no se les asigna un valor.

```
var myString string      // "" - an empty string
var myInt int64          // 0 - applies to all types of int and uint
var myFloat float64      // 0.0 - applies to all types of float and complex
var myBool bool          // false
var myPointer *string    // nil
var myInter interface{}  // nil
```

Esto también se aplica a mapas, segmentos, canales y tipos de funciones. Estos tipos se inicializarán a cero. En las matrices, cada elemento se inicializa al valor cero de su tipo respectivo.

Valores de cero más complejos

En rebanadas el valor cero es una rebanada vacía.

```
var myIntSlice []int    // [] - an empty slice
```

Utilice `make` para crear una división rellena con valores, cualquier valor creado en la división se establece en el valor cero del tipo de la división. Por ejemplo:

```
myIntSlice := make([]int, 5)    // [0, 0, 0, 0, 0] - a slice with 5 zeroes
fmt.Println(myIntSlice[3])
// Prints 0
```

En este ejemplo, `myIntSlice` es un segmento `int` que contiene 5 elementos que son todos 0 porque ese es el valor cero para el tipo `int` .

También puede crear una división con `new` , esto creará un puntero a una división.

```
myIntSlice := new([]int)        // &[] - a pointer to an empty slice
*myIntSlice = make([]int, 5)    // [0, 0, 0, 0, 0] - a slice with 5 zeroes
fmt.Println((*myIntSlice)[3])
```

```
// Prints 0
```

Nota: los punteros de `myIntSlice[3]` no admiten la indexación, por lo que no puede acceder a los valores utilizando `myIntSlice[3]` , en su lugar, debe hacerlo como `(*myIntSlice)[3]` .

Valores cero de Struct

Al crear una estructura sin inicializarla, cada campo de la estructura se inicializa a su valor cero respectivo.

```
type ZeroStruct struct {
    myString string
    myInt    int64
    myBool   bool
}

func main() {
    var myZero = ZeroStruct{}
    fmt.Printf("Zero values are: %q, %d, %t\n", myZero.myString, myZero.myInt, myZero.myBool)
    // Prints "Zero values are: "", 0, false"
}
```

Valores Cero Arreglos

Según el blog Go :

> Las matrices no necesitan inicializarse explícitamente; el valor cero de una matriz es una matriz lista para usar cuyos elementos están ellos mismos en cero

Por ejemplo, `myIntArray` se inicializa con el valor cero de `int` , que es 0:

```
var myIntArray [5]int     // an array of five 0's: [0, 0, 0, 0, 0]
```

Lea Valores cero en línea: https://riptutorial.com/es/go/topic/6069/valores-cero

Capítulo 71: Valores cero

Examples

Explicación

Los valores cero o la inicialización cero son simples de implementar. Viniendo de lenguajes como Java, puede parecer complicado que algunos valores sean `nil` y otros no. En resumen de Valor cero: La especificación del lenguaje de programación Go :

> Los punteros, funciones, interfaces, segmentos, canales y mapas son los únicos tipos que pueden ser nulos. El resto se inicializa en falso, cero o cadenas vacías según sus respectivos tipos.

Si una función comprueba alguna condición, pueden surgir problemas:

```
func isAlive() bool {
    //Not implemented yet
    return false
}
```

El valor cero será falso incluso antes de la implementación. Las pruebas unitarias dependientes del retorno de esta función podrían dar falsos positivos / negativos.

Una solución típica es devolver también un error, que es idiomático en Go:

```
package main

import "fmt"

func isAlive() (bool, error) {
    //Not implemented yet
    return false, fmt.Errorf("Not implemented yet")
}

func main() {
    _, err := isAlive()
    if err != nil {
        fmt.Printf("ERR: %s\n", err.Error())
    }

}
```

jugar en el patio de recreo

Al devolver tanto una estructura como un error, se necesita una estructura de usuario para la devolución, que no es muy elegante. Hay dos opciones de contador:

- Trabajar con interfaces: devuelve nil devolviendo una interfaz.
- Trabajar con punteros: un puntero **puede** ser `nil`

Por ejemplo, el siguiente código devuelve un puntero:

```
func(d *DB) GetUser(id uint64) (*User, error) {
    //Some error ocurred
    return nil, err
}
```

Lea Valores cero en línea: https://riptutorial.com/es/go/topic/6379/valores-cero

Capítulo 72: Variables

Sintaxis

- var x int // declara la variable x con el tipo int
- var s string // declarar variable s con tipo string
- x = 4 // define el valor de x
- s = "foo" // define el valor de s
- y: = 5 // declara y define y deduce su tipo a int
- f: = 4.5 // declara y define f inferiendo su tipo a float64
- b: = "bar" // declara y define b inferiendo su tipo a cadena

Examples

Declaración Variable Básica

Go es un lenguaje de tipo estático, lo que significa que generalmente tiene que declarar el tipo de las variables que está utilizando.

```
// Basic variable declaration. Declares a variable of type specified on the right.
// The variable is initialized to the zero value of the respective type.
var x int
var s string
var p Person // Assuming type Person struct {}

// Assignment of a value to a variable
x = 3

// Short declaration using := infers the type
y := 4

u := int64(100)    // declare variable of type int64 and init with 100
var u2 int64 = 100 // declare variable of type int64 and init with 100
```

Asignación de variables múltiples

En Go, puedes declarar múltiples variables al mismo tiempo.

```
// You can declare multiple variables of the same type in one line
var a, b, c string

var d, e string = "Hello", "world!"

// You can also use short declaration to assign multiple variables
x, y, z := 1, 2, 3

foo, bar := 4, "stack" // `foo` is type `int`, `bar` is type `string`
```

Si una función devuelve varios valores, también puede asignar valores a las variables en función

de los valores de retorno de la función.

```
func multipleReturn() (int, int) {
    return 1, 2
}

x, y := multipleReturn() // x = 1, y = 2

func multipleReturn2() (a int, b int) {
    a = 3
    b = 4
    return
}

w, z := multipleReturn2() // w = 3, z = 4
```

Identificador en blanco

Go lanzará un error cuando haya una variable que no esté en uso para animarte a escribir un código mejor. Sin embargo, hay algunas situaciones en las que realmente no es necesario utilizar un valor almacenado en una variable. En esos casos, utiliza un "identificador en blanco" _ para asignar y descartar el valor asignado.

A un identificador en blanco se le puede asignar un valor de cualquier tipo, y se usa más comúnmente en funciones que devuelven valores múltiples.

Valores de retorno múltiples

```
func SumProduct(a, b int) (int, int) {
    return a+b, a*b
}

func main() {
    // I only want the sum, but not the product
    sum, _ := SumProduct(1,2) // the product gets discarded
    fmt.Println(sum) // prints 3
}
```

Utilizando `range`

```
func main() {

    pets := []string{"dog", "cat", "fish"}

    // Range returns both the current index and value
    // but sometimes you may only want to use the value
    for _, pet := range pets {
        fmt.Println(pet)
    }

}
```

Comprobando el tipo de una variable

Hay algunas situaciones en las que no estará seguro de qué tipo es una variable cuando se devuelve desde una función. Siempre puede verificar el tipo de una variable usando `var.(type)` si no está seguro de qué tipo es:

```
x := someFunction() // Some value of an unknown type is stored in x now

switch x := x.(type) {
    case bool:
        fmt.Printf("boolean %t\n", x)             // x has type bool
    case int:
        fmt.Printf("integer %d\n", x)             // x has type int
    case string:
        fmt.Printf("pointer to boolean %s\n", x) // x has type string
    default:
        fmt.Printf("unexpected type %T\n", x)     // %T prints whatever type x is
}
```

Lea Variables en línea: https://riptutorial.com/es/go/topic/674/variables

Capítulo 73: Venta

Observaciones

La venta es un método para asegurar que todos los paquetes de terceros que usa en su proyecto Go sean consistentes para todos los que desarrollan para su aplicación.

Cuando su paquete Go importa otro paquete, el compilador normalmente verifica `$GOPATH/src/` para la ruta del proyecto importado. Sin embargo, si su paquete contiene una carpeta llamada `vendor` , el compilador registrará esa carpeta *primero* . Esto significa que puede importar paquetes de otras partes dentro de su propio repositorio de código, sin tener que modificar su código.

La venta es una característica estándar en Go 1.6 y superior. En Go 1.5, debe configurar la variable de entorno de `GO15VENDOREXPERIMENT=1` para habilitar la venta.

Examples

Use govendor para agregar paquetes externos

Govendor es una herramienta que se utiliza para importar paquetes de terceros en su repositorio de código de una manera que sea compatible con la venta de Golang.

Digamos, por ejemplo, que está utilizando un paquete de terceros `bosun.org/slog` :

```
package main

import "bosun.org/slog"

func main() {
    slog.Infof("Hello World")
}
```

La estructura de su directorio puede verse como:

```
$GOPATH/src/
├── github.com/me/helloworld/
|   ├── hello.go
├── bosun.org/slog/
|   ├── ... (slog files)
```

Sin embargo, es posible que alguien que `github.com/me/helloworld` no tenga una `$GOPATH/src/bosun.org/slog/` , lo que hace que *su* compilación falle debido a que faltan paquetes.

Ejecutar el siguiente comando en su indicador de comando tomará todos los paquetes externos de su paquete de Go y empaquetará los bits necesarios en una carpeta de proveedor:

```
govendor add +e
```

Esto le indica a Govendor que agregue todos los paquetes externos a su repositorio actual.

La estructura del directorio de su aplicación ahora sería:

```
$GOPATH/src/
├── github.com/me/helloworld/
|   ├── vendor/
|   |   ├── bosun.org/slog/
|   |   |   ├── ... (slog files)
|   ├── hello.go
```

y aquellos que clonen su repositorio también tomarán los paquetes de terceros requeridos.

Uso de basura para gestionar ./vendor

`trash` es una herramienta de venta minimalista que configura con el archivo `vendor.conf` . Este ejemplo es para la `trash` sí:

```
# package
github.com/rancher/trash

github.com/Sirupsen/logrus                      v0.10.0
github.com/urfave/cli                           v1.18.0
github.com/cloudfoundry-incubator/candiedyaml   99c3df8
https://github.com/imikushin/candiedyaml.git
github.com/stretchr/testify                     v1.1.3
github.com/davecgh/go-spew                      5215b55
github.com/pmezard/go-difflib                   792786c
golang.org/x/sys                                a408501
```

La primera línea sin comentarios es el paquete que administramos ./vendor (nota: esto puede ser literalmente cualquier paquete en su proyecto, no solo el raíz).

Las líneas comentadas comienzan con `#` .

Cada línea no vacía y sin comentarios enumera una dependencia. Solo el paquete "raíz" de la dependencia necesita ser listado.

Después de que el nombre del paquete va a la versión (confirmación, etiqueta o rama) y, opcionalmente, a la URL del repositorio de paquetes (de forma predeterminada, se deduce del nombre del paquete).

Para completar su directorio ./vendor, necesita tener el archivo `vendor.conf` en el directorio actual y simplemente ejecutar:

```
$ trash
```

Trash clonará las bibliotecas vendidas en `~/.trash-cache` (por defecto), comprueba las versiones solicitadas, copia los archivos en `./vendor` dir y `./vendor` **los paquetes no importados y los archivos de prueba** . Este último paso mantiene su ./vendor lean y mean y ayuda a ahorrar espacio en su repositorio de proyecto.

Nota: a partir de v0.2.5, la papelera está disponible para Linux y macOS, y solo admite git para recuperar paquetes, ya que git es el más popular, pero estamos trabajando para agregar todos los demás que `go get` soporte.

Usar golang / dep

Golang / Dep es una herramienta de gestión de dependencias prototipo. Pronto será una herramienta oficial de versionamiento. Estado actual **alfa** .

Uso

Obtener la herramienta a través de

```
$ go get -u github.com/golang/dep/...
```

El uso típico en un nuevo repositorio podría ser

```
$ dep init
$ dep ensure -update
```

Para actualizar una dependencia a una nueva versión, puede ejecutar

```
$ dep ensure github.com/pkg/errors@^0.8.0
```

Tenga en cuenta que los formatos de archivo de manifiesto y bloqueo **ya se han finalizado** . Estos seguirán siendo compatibles incluso cuando la herramienta cambie.

vendor.json utilizando la herramienta Govendor

```
# It creates vendor folder and vendor.json inside it
govendor init

# Add dependencies in vendor.json
govendor fetch <dependency>

# Usage on new repository
# fetch depenencies in vendor.json
govendor sync
```

Ejemplo vendor.json

```
{

"comment": "",
"ignore": "test",
"package": [
    {
        "checksumSHA1": "kBeNcaKk56FguvPSUCEaH6AxpRc=",
        "path": "github.com/golang/protobuf/proto",
        "revision": "2bba0603135d7d7f5cb73b2125beeda19c09f4ef",
```

```
        "revisionTime": "2017-03-31T03:19:02Z"
    },
    {
        "checksumSHA1": "1DRAxdlWzS4U0xKN/yQ/fdNN7f0=",
        "path": "github.com/syndtr/goleveldb/leveldb/errors",
        "revision": "8c81ea47d4c41a385645e133e15510fc6a2a74b4",
        "revisionTime": "2017-04-09T01:48:31Z"
    }
],
"rootPath": "github.com/sample"

}
```

Lea Venta en línea: https://riptutorial.com/es/go/topic/978/venta

Capítulo 74: XML

Observaciones

Si bien muchos de los usos del paquete `encoding/xml` incluyen cálculo de referencias y desvinculación en una `struct` Go, vale la pena señalar que esto no es un mapeo directo. La documentación del paquete dice:

> El mapeo entre elementos XML y estructuras de datos es inherentemente defectuoso: un elemento XML es una colección de valores anónimos dependiente de la orden, mientras que una estructura de datos es una colección de valores nombrados independiente de la orden.

Para pares simples, desordenados, clave-valor, el uso de una codificación diferente, como Gob's o JSON, puede ser un mejor ajuste. Para datos ordenados o flujos de datos basados en eventos / devoluciones de llamadas, XML puede ser la mejor opción.

Examples

Descodificación / no básica de elementos anidados con datos.

Los elementos XML a menudo se anidan, tienen datos en atributos y / o como datos de caracteres. La forma de capturar estos datos es usando `,attr` y `,chardata` respectivamente, para esos casos.

```
var doc = `
<parent>
  <child1 attr1="attribute one"/>
  <child2>and some cdata</child2>
</parent>
`

type parent struct {
    Child1 child1 `xml:"child1"`
    Child2 child2 `xml:"child2"`
}

type child1 struct {
    Attr1 string `xml:"attr1,attr"`
}

type child2 struct {
    Cdata1 string `xml:",cdata"`
}

func main() {
    var obj parent
    err := xml.Unmarshal([]byte(doc), &obj)
    if err != nil {
        log.Fatal(err)
    }
```

```
    fmt.Println(obj.Child2.Cdata1)

}
```

Playground

Lea XML en línea: https://riptutorial.com/es/go/topic/1846/xml

Capítulo 75: YAML

Examples

Creando un archivo de configuración en formato YAML

```
import (
    "io/ioutil"
    "path/filepath"

    "gopkg.in/yaml.v2"
)

func main() {
    filename, _ := filepath.Abs("config/config.yml")
    yamlFile, err := ioutil.ReadFile(filename)
    var config Config
    err = yaml.Unmarshal(yamlFile, &config)
    if err != nil {
        panic(err)
    }
    //env can be accessed from config.Env
}

type Config struct {
    Env                 string `yaml:"env"`
}

//config.yml should be placed in config/config.yml for example, and needs to have the
following line for the above example:
//env: test
```

Lea YAML en línea: https://riptutorial.com/es/go/topic/2503/yaml

Creditos

S. No	Capítulos	Contributors
1	Empezando con Go	4444, alejosocorro, Alexander, Amitay Stern, Andrej Bencic, Andrii Abramov, burfl, Burhan Ali, cat, Cody Gustafson, Community, David G., Dmitri Goldring, Feckmore, Florian Hämmerle, Franck Dernoncourt, Gerep, Greg Bray, hellyale, Hunter, James Taylor, Jared Hooper, Jon Chan, Katamaritaco, Mark Henderson, Matt, mbb, MegaTom, mmlb, mnoronha, mohan08p, Nir, nix, nouney, patterns, Pavel Nikolov, ProfNandaa, Quentin Skousen, Radouane ROUFID, Rahul Nair, RamenChef, raulsntos, Sam Whited, seriousdev, Simone Carletti, skunkmb, sztanpet, Tanmay Garg, Topo, Unapiedra, Vikash, Xavier Nicollet
2	Agrupación de memoria	Elijah Sarver, Grzegorz Żur, Kenny Grant
3	Análisis de archivos CSV	Ainar-G
4	Análisis de argumentos de línea de comando y banderas	Ingve, Pavel Kazhevets, Sam Whited
5	Aplazar	abhink, Adrian, Sam Whited, Vikash
6	Archivo I / O	1lann, Andres Kütt, greatwolf, Grzegorz Żur, koblas, noisewaterphd, Quentin Skousen, Sam Whited
7	Arrays	NatNgs, nouney, Noval Agung Prayogo, Sam Whited
8	Autorización JWT en Go	AniSkywalker
9	Bucles	1lann, burfl, Community, ivan73, jayantS, Jon Chan, mgh, MohamedAlaa, RamenChef, Sam Whited, Steven Maude, Thomas Gerot
10	Buenas prácticas en la estructura del proyecto.	Iman Tumorang
11	cgo	MaC, Vojtech Kane

12	Cierres	abhink
13	Cliente HTTP	1lann, dmportella, Lanzafame, Sam Whited, SommerEngineering
14	Codificación Base64	Nathan Osman, RamenChef, Sam Whited
15	Comandos de ejecución	Krzysztof Kowalczyk, Kyle Brandt, Nevermore
16	Comenzando con el uso de Go Atom	Ali M, Danny Chen, Katamaritaco
17	Compilación cruzada	Jordan, Katamaritaco, mbb, mohan08p, RamenChef, Riley Guerin, SH', Siu Ching Pong -Asuka Kenji-, SommerEngineering, sztanpet, Zoyd
18	Concurrencia	Chris Lucas, Community, Florian Hämmerle, flyingfinger, Grzegorz Żur, Harshal Sheth, Ilya, Inanc Gumus, Kyle Brandt, Nathan Osman, Roland Illig, Ryan Kelln, Tim S. Van Haren, VonC, zianwar, Zoyd
19	Constantes	Pavel Nikolov, RamenChef, Sam Whited, Simone Carletti
20	Construir restricciones	4444, RamenChef, Sam Whited, seriousdev
21	Contexto	Ingaz, Sam Whited
22	Criptografía	SommerEngineering
23	Cuerda	Ainar-G, NatNgs, raulsntos
24	Derivación	burfl, Community, ganesh kumar, Ingve, nk2ge5k
25	Desarrollando para múltiples plataformas con compilación condicional	ecem
26	E / S de consola	Abhilekh Singh
27	El comando go	ganesh kumar, Harshal Sheth, Ingve, Lanzafame, Mayank Patel , Nevermore, Quentin Skousen, Sam Whited, theflametrooper, Vikash
28	Enchufar	Sam Whited
29	Enviar / recibir correos electrónicos	Utahcon

30	Estructuras	abhink, Amitay Stern, Anthony Atkinson, Blixt, burfl, cizixs, Community, FredMaggiowski, Howl, Ingve, Kin, MaC, Mark Henderson, matt.s, mohan08p, Nathan Osman, nouney, Patrick, Quentin Skousen, radbrawler, RamenChef, Roland Illig, Simone Carletti, sunkuet02, Vojtech Kane, Wojciech Kazior
31	Expansión en línea	Sam Whited
32	Explotación florestal	Grzegorz Żur, Jon Chan, Nathan Osman, Pavel Kazhevets, Sam Whited
33	Fmt	Lanzafame, Nevermore, Sam Whited
34	Funciones	Boris Le Méec, Dmytro Sadovnychyi, Grzegorz Żur, jayantS, LeoTao, Nathan Osman, nouney, palestamp, RamenChef, Right leg, Thomas Gerot
35	Goroutines	mohan08p
36	Hora	Lanzafame, NatNgs, raulsntos
37	Imágenes	putu
38	Instalación	sadlil
39	Interfaces	Cody Roseborough, dotctor, Francis Norton, Grzegorz Żur, icza, Ingve, meysam, Mike, ptman, sadlil, Sam Whited, Wendy Adi
40	Iota	4444, Florian Hämmerle, Ingve, mohan08p, Sam Whited, Wojciech Kazior, Zoyd
41	JSON	Dmitry Udod, Joe, Jon Chan, Kyle Brandt, Nathan Osman, RamenChef, Sam Whited, shayan, Simone Carletti, sztanpet, Tanmay Garg, Utahcon
42	Lectores	Mike Houston
43	Los canales	Chris Lucas, Howl, Jeremy, Kwarrtz, metmirr, RamenChef, Rodolfo Carvalho, Zoyd
44	Manejo de errores	bowsersenior, elevine, Elijah Sarver, Florian Hämmerle, groob, Ingve, Joe, Kin, Paul Hankin, Quentin Skousen, Sam Whited, Simone Carletti, Sridhar, Surreal Dreams, Vervious, Zoyd
45	Mapas	Abhay, abhink, Amitay Stern, Brendan, burfl, chowey, Chris Lucas, cizixs, Community, creker, Dair, Dmitri Goldring, gbulmer , Hugo, James, JepZ, Joe, Kaedys, Kamil Kisiel, Kyle Brandt, Mark Henderson, matt.s, Milo Christiansen, NatNgs, Oleg Sklyar, radbrawler, RamenChef, Roland Illig, Sam Whited, seh, Simone Carletti, skunkmb, Surreal Dreams, Vojtech Kane, Zoyd

		, Zyerah
46	Métodos	ganesh kumar, Pavel Kazhevets
47	mgo	Florian Hämmerle, Sourabh
48	Middleware	Ankit Deshpande
49	Mutex	Adrian, Prutswonder
50	Pánico y Recuperación	JunLe Meng, Kaedys, Kristoffer Sall-Storgaard, Sam Whited
51	Paquetes	dmportella, Grzegorz Żur, icza, Michael, Nathan Osman, RadicalFish, RamenChef, skunkmb, tkausl
52	Perfilado usando la herramienta go pprof	mbb, Nevermore, radbrawler
53	Piscinas de trabajadores	burfl, photoionized, seriousdev
54	Plantillas	Pavel Kazhevets, RamenChef, Tanmay Garg
55	Programación orientada a objetos	Davyd Dzhahaiev, Sam Whited, zola
56	Protobuf en Go	mohan08p
57	Pruebas	Adrian, Ankit Deshpande, Harshal Sheth, ivan.sim, Jared Ririe, Nathan Osman, Omid, Pavel Nikolov, Rodolfo Carvalho, seriousdev, Toni Villena, Zoyd
58	Punteros	David Hoelzer, Jon Chan, Joost, Mal Curtis, metmirr, Nevermore, skunkmb
59	Rebanadas	1lann, Benjamin Kadish, burfl, cizixs, Grzegorz Żur, Guillaume, Jared Hooper, Joost, Jukurrpa, Kyle Brandt, Mark Henderson, NatNgs, RamenChef, Simone Carletti, skunkmb, Tanmay Garg, Zoyd
60	Reflexión	ganesh kumar, mammothbane, radbrawler
61	Seleccione y Canales	Harshal Sheth, Kaedys, RamenChef, Sam Whited, Utahcon
62	Señales OS	Community, Sam Whited, Utahcon
63	Servidor HTTP	Chief, frigo americain, Jon Erickson, Kin, Nathan Osman, rogerdpack, Sam Whited, Sascha, seriousdev, Simone Carletti,

		SommerEngineering, Tanmay Garg, Zhinkk
64	SQL	Adrian, artamonovdev, bernardn, Francesco Pasa, Nevermore, Sam Whited, Sascha, Tanmay Garg, wrfly
65	Texto + HTML Plantillas	Stephen Rudolph
66	Tipo de conversiones	Adrian, Florian Hämmerle
67	trozo	zola
68	Valores cero	Harshal Sheth, raulsntos, Surreal Dreams
69	Variables	Community, FredMaggiowski, Jon Chan, Simone Carletti
70	Venta	Abhilekh Singh, Boris Le Méec, burfl, Dmitri Goldring, Ivan Mikushin, Mark Henderson, Martin Campbell, Michael, Sam Whited, Vardius
71	XML	ivarg, Sam Whited
72	YAML	Nathan Osman, Orr, Sam Whited