Aprende Go en Y minutos

Nacho Pacheco

Published with GitBook

Tabla de contenido

Introducción

Esta es la traducción de Aprende Go en Y minutos, hice una petición de atracción en el repositorio oficial pero al parecer tarda un poco el proceso.

Es un artículo dedicado a programadores con un poco de experiencia en otros lenguajes. Por lo tanto, trata de evitar grandes explicaciones en los conceptos básicos salvo los específicos del lenguaje Go.

Go es un lenguaje que sus fuentes son utf-8, es decir, que acepta caracteres especiales como las vocales acentuadas y las eñes caracteísticas de nuestro idioma, por lo tanto, además traduje el código para una mejor comprensión, lo puedes probar y jugar con él en el parque de diversiones Go.

Aprende Go en Y minutos

Go fue creado por la necesidad de hacer el trabajo rápidamente. No es la última tendencia en informática, pero es la forma nueva y más rápida de resolver problemas reales.

Tiene conceptos familiares de lenguajes imperativos con tipado estático. Es rápido compilando y rápido al ejecutar, añade una concurrencia fácil de entender para las CPUs de varios núcleos de hoy día, y tiene características que ayudan con la programación a gran escala.

Go viene con una biblioteca estándar muy buena y una entusiasta comunidad.

```
// Comentario de una sola línea
/* Comentario
   multilínea */

// La cláusula `package` aparece al comienzo de cada fichero fuente.
// `main` es un nombre especial que declara un ejecutable en vez de una
// biblioteca.
package main

// La instrucción `import` declara los paquetes de bibliotecas referidos
// en este fichero.
import (
    "fmt"      // Un paquete en la biblioteca estándar de Go.
    "io/ioutil" // Implementa algunas útiles funciones de E/S.
    m "math"   // Biblioteca de matemáticas con alias local m.
    "net/http" // Sí, ¡un servidor web!
    "strconv"  // Conversiones de cadenas.
)

// Definición de una función. `main` es especial. Es el punto de entrada
// para el ejecutable. Te guste o no, Go utiliza llaves.
func main() {
    // Println imprime una línea a stdout.
    // Cualificalo con el nombre del paquete, fmt.
    fmt.Println("¡Hola mundo!")

    // Llama a otra función de este paquete.
    másAlláDelHola()
}

// Las funciones llevan parámetros entre paréntesis.
// Si no hay parámetros, los paréntesis siguen siendo obligatorios.
func másAlláDelHola() {
    var x int // Declaración de una variable.
              // Las variables se deben declarar antes de utilizarlas.
    x = 3     // Asignación de variable.
    // Declaración "corta" con := para inferir el tipo, declarar y asignar.
    y := 4
    suma, producto := aprendeMúltiple(x, y) // La función devuelve dos
                                            // valores.
    fmt.Println("suma:", suma, "producto:", producto) // Simple salida.
    aprendeTipos()                         // < y minutos, ¡aprende más!
}
```

```
// Las funciones pueden tener parámetros y (¡múltiples!) valores de
// retorno.
func aprendeMúltiple(x, y int) (suma, producto int) {
    return x + y, x * y // Devuelve dos valores.
}

// Algunos tipos incorporados y literales.
func aprendeTipos() {
    // La declaración corta suele darte lo que quieres.
    s := "¡Aprende Go!" // tipo cadena.
    s2 := `Un tipo cadena "puro" puede incluir
saltos de línea.` // mismo tipo cadena

    // Literal no ASCII. Los ficheros fuente de Go son UTF-8.
    g := 'Σ' // Tipo rune, un alias de int32, alberga un carácter unicode.
    f := 3.14195 // float64, el estándar IEEE-754 de coma flotante 64-bit.
    c := 3 + 4i  // complex128, representado internamente por dos float64.
    // Sintaxis Var con iniciadores.
    var u uint = 7 // Sin signo, pero la implementación depende del tamaño
                   // como en int.
    var pi float32 = 22. / 7

    // Sintáxis de conversión con una declaración corta.
    n := byte('\n') // byte es un alias para uint8.

    // Los arreglos tienen un tamaño fijo a la hora de compilar.
    var a4 [4]int           // Un arreglo de 4 ints, iniciados a 0.
    a3 := [...]int{3, 1, 5} // Un arreglo iniciado con un tamaño fijo de tres
                            // elementos, con valores 3, 1 y 5.
    // Los sectores tienen tamaño dinámico. Los arreglos y sectores tienen
    // sus ventajas y desventajas pero los casos de uso para los sectores
    // son más comunes.
    s3 := []int{4, 5, 9}     // Comparar con a3. No hay puntos suspensivos.
    s4 := make([]int, 4)     // Asigna sectores de 4 ints, iniciados a 0.
    var d2 [][]float64       // Solo declaración, sin asignación.
    bs := []byte("a sector") // Sintaxis de conversión de tipo.
    // Debido a que son dinámicos, los sectores pueden crecer bajo demanda.
    // Para añadir elementos a un sector, se utiliza la función incorporada
    // append().
    // El primer argumento es el sector al que se está anexando. Comúnmente,
    // la variable del arreglo se actualiza en su lugar, como en el
    // siguiente ejemplo.
    sec := []int{1, 2 , 3}      // El resultado es un sector de longitud 3.
    sec = append(sec, 4, 5, 6)  // Añade 3 elementos. El sector ahora tiene una
                                // longitud de 6.
    fmt.Println(sec) // El sector actualizado ahora es [1 2 3 4 5 6]
    // Para anexar otro sector, en lugar de la lista de elementos atómicos
    // podemos pasar una referencia a un sector o un sector literal como
    // este, con elipsis al final, lo que significa tomar un sector y
    // desempacar sus elementos, añadiéndolos al sector sec.
    sec = append(sec, []int{7, 8, 9} ...) // El segundo argumento es un
                                          // sector literal.
    fmt.Println(sec)  // El sector actualizado ahora es [1 2 3 4 5 6 7 8 9]
    p, q := aprendeMemoria() // Declara p, q para ser un tipo puntero a
                             // int.
    fmt.Println(*p, *q)      // * sigue un puntero. Esto imprime dos ints.

    // Los Mapas son arreglos asociativos dinámicos, como los hash o
```

```
    // diccionarios de otros lenguajes.
    m := map[string]int{"tres": 3, "cuatro": 4}
    m["uno"] = 1

    // Las variables no utilizadas en Go producen error.
    // El guión bajo permite "utilizar" una variable, pero descartar su
    // valor.
    _, _, _, _, _, _, _, _, _ = s2, g, f, u, pi, n, a3, s4, bs
    // Esto cuenta como utilización de variables.
    fmt.Println(s, c, a4, s3, d2, m)

    aprendeControlDeFlujo() // Vuelta al flujo.
}

// Es posible, a diferencia de muchos otros lenguajes tener valores de
// retorno con nombre en las funciones.
// Asignar un nombre al tipo que se devuelve en la línea de declaración de
// la función nos permite volver fácilmente desde múltiples puntos en una
// función, así como sólo utilizar la palabra clave `return`, sin nada
// más.
func aprendeRetornosNombrados(x, y int) (z int) {
    z = x * y
    return // aquí z es implícito, porque lo nombramos antes.
}

// Go posee recolector de basura. Tiene punteros pero no aritmética de
// punteros. Puedes cometer errores con un puntero nil, pero no
// incrementando un puntero.
func aprendeMemoria() (p, q *int) {
    // Los valores de retorno nombrados q y p tienen un tipo puntero
    // a int.
    p = new(int) // Función incorporada que reserva memoria.
    // La asignación de int se inicia a 0, p ya no es nil.
    s := make([]int, 20) // Reserva 20 ints en un solo bloque de memoria.
    s[3] = 7             // Asigna uno de ellos.
    r := -2              // Declara otra variable local.
    return &s[3], &r     // & toma la dirección de un objeto.
}

func cálculoCaro() float64 {
    return m.Exp(10)
}

func aprendeControlDeFlujo() {
    // La declaración If requiere llaves, pero no paréntesis.
    if true {
        fmt.Println("ya relatado")
    }
    // El formato está estandarizado por la orden "go fmt."
    if false {
        // Abadejo.
    } else {
        // Relamido.
    }
    // Utiliza switch preferentemente para if encadenados.
    x := 42.0
    switch x {
    case 0:
    case 1:
```

```
    case 42:
        // Los cases no se mezclan, no requieren de "break".
    case 43:
        // No llega.
    }
    // Como if, for no utiliza paréntesis tampoco.
    // Variables declaradas en for e if son locales a su ámbito.
    for x := 0; x < 3; x++ { // ++ es una instrucción.
        fmt.Println("iteración", x)
    }
    // aquí x == 42.

    // For es la única instrucción de bucle en Go, pero tiene formas
    // alternativas.
    for { // Bucle infinito.
        break    // ¡Solo bromeaba!
        continue // No llega.
    }

    // Puedes usar `range` para iterar en un arreglo, un sector, una
    // cadena, un mapa o un canal.
    // `range` devuelve o bien, un canal o de uno a dos valores (arreglo,
    // sector, cadena y mapa).
    for clave, valor := range map[string]int{"uno": 1, "dos": 2, "tres": 3} {
        // por cada par en el mapa, imprime la clave y el valor
        fmt.Printf("clave=%s, valor=%d\n", clave, valor)
    }

    // Como en for, := en una instrucción if significa declarar y asignar
    // primero, luego comprobar y > x.
    if y := cálculoCaro(); y > x {
        x = y
    }
    // Las funciones literales son "cierres".
    granX := func() bool {
        return x > 100 // Referencia a x declarada encima de la instrucción
                       // switch.
    }
    fmt.Println("granX:", granX()) // cierto (la última vez asignamos
                                   // 1e6 a x).
    x /= 1.3e3                     // Esto hace a x == 1300
    fmt.Println("granX:", granX()) // Ahora es falso.

    // Es más las funciones literales se pueden definir y llamar en línea,
    // actuando como un argumento para la función, siempre y cuando:
    // a) la función literal sea llamada inmediatamente (),
    // b) el tipo del resultado sea del tipo esperado del argumento
    fmt.Println("Suma dos números + doble: ",
        func(a, b int) int {
            return (a + b) * 2
        }(10, 2)) // Llamada con argumentos 10 y 2
    // => Suma dos números + doble: 24

    // Cuando lo necesites, te encantará.
    goto encanto
encanto:

    aprendeFunciónFábrica() // func devolviendo func es divertido(3)(3)
    aprendeADiferir()       // Un rápido desvío a una importante palabra clave.
```

```
    aprendeInterfaces()     // ¡Buen material dentro de poco!
}

func aprendeFunciónFábrica() {
    // Las dos siguientes son equivalentes, la segunda es más práctica
    fmt.Println(instrucciónFábrica("día")("Un bello", "de verano"))

    d := instrucciónFábrica("atardecer")
    fmt.Println(d("Un hermoso", "de verano"))
    fmt.Println(d("Un maravilloso", "de verano"))
}

// Los decoradores son comunes en otros lenguajes. Lo mismo se puede hacer
// en Go con funciónes literales que aceptan argumentos.
func instrucciónFábrica(micadena string) func(antes, después string) string {
    return func(antes, después string) string {
        return fmt.Sprintf("¡%s %s %s!", antes, micadena, después) // nueva cadena
    }
}

func aprendeADiferir() (ok bool) {
    // las instrucciones diferidas se ejecutan justo antes de que la
    // función regrese.
    defer fmt.Println("las instrucciones diferidas se ejecutan en orden inverso (PEPS).")
    defer fmt.Println("\nEsta línea se imprime primero debido a que")
    // Defer se usa comunmente para cerrar un fichero, por lo que la
    // función que cierra el fichero se mantiene cerca de la función que lo
    // abrió.
    return true
}

// Define Stringer como un tipo interfaz con un método, String.
type Stringer interface {
    String() string
}

// Define par como una estructura con dos campos int, x e y.
type par struct {
    x, y int
}

// Define un método en el tipo par. Par ahora implementa a Stringer.
func (p par) String() string { // p se conoce como el "receptor"
    // Sprintf es otra función pública del paquete fmt.
    // La sintaxis con punto se refiere a los campos de p.
    return fmt.Sprintf("(%d, %d)", p.x, p.y)
}

func aprendeInterfaces() {
    // La sintaxis de llaves es una "estructura literal". Evalúa a una
    // estructura iniciada. La sintaxis := declara e inicia p a esta
    // estructura.
    p := par{3, 4}
    fmt.Println(p.String()) // Llama al método String de p, de tipo par.
    var i Stringer          // Declara i como interfaz de tipo Stringer.
    i = p                   // Válido porque par implementa Stringer.
    // Llama al metodo String de i, de tipo Stringer. Misma salida que
    // arriba.
    fmt.Println(i.String())
```

```
    // Las funciones en el paquete fmt llaman al método String para
    // consultar un objeto por una representación imprimible de si
    // mismo.
    fmt.Println(p) // Salida igual que arriba. Println llama al método
                   // String.
    fmt.Println(i) // Salida igual que arriba.
    aprendeNúmeroVariableDeParámetros("¡gran", "aprendizaje", "aquí!")
}

// Las funciones pueden tener número variable de argumentos.
func aprendeNúmeroVariableDeParámetros(misCadenas ...interface{}) {
    // Itera en cada valor de los argumentos variables.
    // El espacio en blanco aquí omite el índice del argumento arreglo.
    for _, parámetro := range misCadenas {
        fmt.Println("parámetro:", parámetro)
    }

    // Pasa el valor de múltiples variables como parámetro variadic.
    fmt.Println("parámetros:", fmt.Sprintln(misCadenas...))
    aprendeManejoDeError()
}

func aprendeManejoDeError() {
    // ", ok" forma utilizada para saber si algo funcionó o no.
    m := map[int]string{3: "tres", 4: "cuatro"}
    if x, ok := m[1]; !ok { // ok será falso porque 1 no está en el mapa.
        fmt.Println("nada allí")
    } else {
        fmt.Print(x) // x sería el valor, si estuviera en el mapa.
    }
    // Un valor de error comunica más información sobre el problema aparte
    // de "ok".
    if _, err := strconv.Atoi("no-int"); err != nil { // _ descarta el
                                                      // valor
        // Imprime "strconv.ParseInt: parsing "no-int": invalid syntax".
        fmt.Println(err)
    }
    // Revisaremos las interfaces más adelante. Mientras tanto...
    aprendeConcurrencia()
}

// c es un canal, un objeto de comunicación concurrente seguro.
func inc(i int, c chan int) {
    c <- i + 1 // <- es el operador "enviar" cuando aparece un canal a la
               // izquierda.
}

// Utilizaremos inc para incrementar algunos números concurrentemente.
func aprendeConcurrencia() {
    // Misma función make utilizada antes para crear un sector. Make asigna
    // e inicia sectores, mapas y canales.
    c := make(chan int)
    // Inicia tres rutinasgo concurrentes. Los números serán incrementados
    // concurrentemente, quizás en paralelo si la máquina es capaz y está
    // correctamente configurada. Las tres envían al mismo canal.
    go inc(0, c) // go es una instrucción que inicia una nueva rutinago.
    go inc(10, c)
    go inc(-805, c)
```

```
    // Lee los tres resultados del canal y los imprime.
    // ¡No se puede saber en que orden llegarán los resultados!
    fmt.Println(<-c, <-c, <-c) // Canal a la derecha, <- es el operador
                                       // "recibe".

    cs := make(chan string)       // Otro canal, este gestiona cadenas.
    ccs := make(chan chan string) // Un canal de canales cadena.
    go func() { c <- 84 }()       // Inicia una nueva rutinago solo para
                                          // enviar un valor.
    go func() { cs <- "verboso" }() // Otra vez, para cs en esta ocasión.
    // Select tiene una sintáxis parecida a la instrucción switch pero cada
    // caso involucra una operacion con un canal. Selecciona un caso de
    // forma aleatoria de los casos que están listos para comunicarse.
    select {
    case i := <-c: // El valor recibido se puede asignar a una variable,
        fmt.Printf("es un %T", i)
    case <-cs:     // o el valor se puede descartar.
        fmt.Println("es una cadena")
    case <-ccs:    // Canal vacío, no está listo para la comunicación.
        fmt.Println("no sucedió.")
    }

    // En este punto un valor fue devuelto de c o cs. Una de las dos
    // rutinasgo que se iniciaron se ha completado, la otrá permancerá
    // bloqueada.

    aprendeProgramaciónWeb() // Go lo hace. Tú también quieres hacerlo.
}

// Una simple función del paquete http inicia un servidor web.
func aprendeProgramaciónWeb() {
// El primer parámetro es la direccinón TCP a la que escuchar.
    // El segundo parámetro es una interfaz, concretamente http.Handler.
    go func() {
        err := http.ListenAndServe(":8080", par{})
        fmt.Println(err) // no ignora errores
    }()
    consultaAlServidor()
}

// Hace un http.Handler de par implementando su único método, ServeHTTP.
func (p par) ServeHTTP(w http.ResponseWriter, r *http.Request) {
    // Sirve datos con un método de http.ResponseWriter.
    w.Write([]byte("¡Aprendiste Go en Y minutos!"))
}

func consultaAlServidor() {
    resp, err := http.Get("http://localhost:8080")
    fmt.Println(err)
    defer resp.Body.Close()
    cuerpo, err := ioutil.ReadAll(resp.Body)
    fmt.Printf("\nEl servidor web dijo: `%s`\n", string(cuerpo))
}
```

Más información

La raíz de todas las cosas sobre Go es el sitio web oficial de Go. Allí puedes seguir el tutorial, jugar interactivamente y leer mucho más.

La definición del lenguaje es altamente recomendada. Es fácil de leer y sorprendentemente corta (como la definición del lenguaje Go en estos días).

Puedes jugar con el código en el parque de diversiones Go. ¡Trata de cambiarlo y ejecutarlo desde tu navegador! Ten en cuenta que puedes utilizar https://play.golang.org como un REPL para probar cosas y el código en el navegador, sin ni siquiera instalar Go.

En la lista de lecturas para estudiantes de Go está el código fuente de la biblioteca estándar. Ampliamente documentado, que demuestra lo mejor del legible y comprensible Go, con su característico estilo y modismos. ¡O puedes hacer clic en un nombre de función en la documentación y aparecerá el código fuente!

Otro gran recurso para aprender Go está en Go con ejemplos.

Glosario

arreglo

`array` en inglés. En Go, un _arreglo_, _matriz_ o _vector_ es una zona de almacenamiento continuo, que contiene una serie de elementos del mismo tipo, los elementos del arreglo. ![Arreglo unidimensional de 10 elementos](img/arreglo.png)

1. Aprende Go