Control de flujos

Introducción

En esta sección del curso de programación en Go, vamos a explorar el control de flujo. El control de flujo es una parte importante de cualquier lenguaje de programación y permite que el programa tome decisiones en función de las condiciones y variables en el código.

En esta sección, vamos a cubrir varios temas importantes del control de flujo en Go. Estos incluyen:

- Operadores booleanos: Los operadores booleanos son una parte esencial de la programación, ya que permiten la evaluación de expresiones lógicas. En esta sección, aprenderás sobre los operadores booleanos en Go y cómo utilizarlos para tomar decisiones en tu código.
- Declaración If-Else: La declaración if-else es una de las formas más comunes de controlar el flujo de un programa. En esta sección, aprenderás cómo utilizar la declaración if-else en Go para tomar decisiones en función de las condiciones en tu programa.
- Declaración Switch: La declaración switch es otra forma común de controlar el flujo de un programa. En esta sección, aprenderás cómo utilizar la declaración switch en Go para tomar decisiones en función de múltiples condiciones.
- Bucle For: El bucle for es una estructura de control de flujo fundamental en cualquier lenguaje de programación. En esta sección, aprenderás cómo utilizar el bucle for en Go para repetir una sección de código un número determinado de veces.
- Uso de funciones: Las funciones son una parte importante de cualquier lenguaje de programación y permiten que el código sea modular y reutilizable. En esta sección, aprenderás cómo utilizar funciones en Go y cómo crear tus propias funciones personalizadas.
- Proyecto de la sección: Al final de la sección, tendrás la oportunidad de aplicar todo lo que has aprendido en un proyecto práctico. Este proyecto te permitirá crear un programa que utilice los conceptos de control de flujo que hemos cubierto en la sección.

Operadores booleanos

Los operadores relacionales y lógicos son utilizados en conjunto en las expresiones lógicas de Go para evaluar condiciones complejas y producir un resultado booleano (verdadero o falso).

Operadores de comparación

Se usan para comparar dos valores y devolver un valor booleano (`true` o `false`) según el resultado de la comparación. Los operadores de comparación incluyen:

- Igualdad `(==)`
- Desigualdad `(!=)`
- Mayor que `(>)`
- Menor que `(<)`
- Mayor o igual que `(>=)`
- Menor o igual que `(<=)`

Aquí hay algunos ejemplos de cómo podríamos utilizar los operadores de comparación en Go:

```
    // Comparación de números
    fmt.Println(1 == 1)   // true
    fmt.Println(1 != 2)   // true
    fmt.Println(2 < 3)    // true
    fmt.Println(3 > 4)    // false
    fmt.Println(4 <= 4)   // true
    fmt.Println(5 >= 6)   // false

    // Comparación de cadenas
    fmt.Println("hola" == "hola")       // true
    fmt.Println("hola" != "adios")      // true
    fmt.Println("abc" < "def")          // true
    fmt.Println("ghi" > "fgh")          // true
    fmt.Println("hij" <= "hij")         // true
    fmt.Println("klm" >= "klmno")       // false

    // Comparación de booleanos
    fmt.Println(true == true)           // true
    fmt.Println(false != true)          // true
    fmt.Println(true && false == false) // true
    fmt.Println(true || false == true)  // true
```

En este ejemplo, utilizamos los operadores de comparación para comparar números, cadenas y booleanos. En cada caso, el resultado de la comparación es un valor booleano que representa si la comparación es verdadera o falsa. Los valores booleanos también se pueden comparar utilizando los mismos operadores de comparación.

Operadores lógicos

Los operadores lógicos en Go son utilizados para evaluar expresiones lógicas y producir un resultado booleano (verdadero o falso). En Go, existen tres operadores lógicos: AND lógico (&&), OR lógico (||) y NOT lógico (!).

Operador AND lógico (&&):

El operador && evalúa dos expresiones booleanas y devuelve verdadero (true) si ambas expresiones son verdaderas, y devuelve falso (false) si alguna de las expresiones es falsa. Por ejemplo:

```
 x := true
y := false
z := x && y
fmt.Println(z) // Imprime false
```

En este ejemplo, la variable z tendrá el valor de falso (false), ya que la expresión x && y evalúa a falso, debido a que la variable y es falsa.

Operador OR lógico (||):

El operador || evalúa dos expresiones booleanas y devuelve verdadero (true) si al menos una de las expresiones es verdadera, y devuelve falso (false) si ambas expresiones son falsas. Por ejemplo:

```
 x := true
y := false
z := x || y
fmt.Println(z) // Imprime true
```

En este ejemplo, la variable z tendrá el valor de verdadero (true), ya que la expresión x || y evalúa a verdadero, debido a que la variable x es verdadera.

Operador NOT lógico (!):

El operador ! niega el valor booleano de una expresión, es decir, si una expresión es verdadera, la niega y devuelve falso, y si una expresión es falsa, la niega y devuelve verdadero. Por ejemplo:

```
 x := true
z := !x
fmt.Println(z) // Imprime false
```

En este ejemplo, la variable z tendrá el valor de falso (false), ya que la expresión !x niega el valor de x, que es verdadero, y devuelve falso.

Aquí hay un ejemplo en Go que utiliza solo operadores lógicos para realizar operaciones sin condiciones:

```
 x := true
y := false

// Negación
fmt.Println(!x) // false
fmt.Println(!y) // true

// AND lógico
fmt.Println(x && x) // true
fmt.Println(x && y) // false
fmt.Println(y && y) // false

// OR lógico
fmt.Println(x || x) // true
fmt.Println(x || y) // true
fmt.Println(y || y) // false
```

En este ejemplo, utilizamos los operadores lógicos !, && y || para realizar operaciones sin utilizar condiciones explícitas.

En la primera sección, utilizamos el operador de negación ! para invertir los valores de x e y.

En la sección de operaciones lógicas && y ||, realizamos operaciones booleanas utilizando los valores de x e y. En el caso del operador &&, el resultado será verdadero solo si ambos operandos son verdaderos, en otro caso, el resultado es falso. En el caso del operador ||, el resultado será verdadero si al menos uno de los operandos es verdadero, de lo contrario, el resultado será falso.

Expresiones

En programación, una expresión es una combinación de valores, variables, operadores y llamadas a funciones que se evalúa para producir un resultado. Las expresiones pueden ser tan simples como una variable que se asigna a un valor, o tan complejas como una ecuación matemática con múltiples operadores y variables.

El orden en que se resuelven las expresiones en un programa depende de la prioridad de los operadores y los paréntesis utilizados para agrupar las operaciones. El orden de resolución de las operaciones sigue las reglas matemáticas convencionales:

- Los paréntesis se evalúan primero. Las expresiones dentro de los paréntesis se resuelven antes que cualquier otra operación.
- Luego se resuelven las operaciones aritméticas, como la multiplicación, la división, la suma y la resta. La multiplicación y la división se resuelven antes que la suma y la resta.
- Finalmente, se resuelven las operaciones de comparación y los operadores lógicos.

Es importante tener en cuenta que los operadores con la misma prioridad se resuelven de izquierda a derecha. Por ejemplo, en la expresión 2 + 3 * 4, la multiplicación se resuelve primero debido a su mayor prioridad, y el resultado es 14. Si queremos que la suma se resuelva primero, debemos utilizar paréntesis para indicar la prioridad, como en (2 + 3) * 4, lo que da como resultado 20.

El orden de resolución de las expresiones es importante porque puede afectar el resultado final de un programa. Por lo tanto, es importante comprender la prioridad de los operadores y utilizar paréntesis para agrupar las operaciones de la manera adecuada.

Aquí hay un ejemplo en Go que utiliza expresiones con paréntesis, operadores aritméticos, operadores de comparación y operadores lógicos.

```
x := 5
y := 10
z := 15

// Expresión con paréntesis, operadores aritméticos, de comparación y lógicos
resultado := ((x+y)*z)/(x*y) > z && x != y

// Imprimir el resultado
fmt.Println(resultado) //False
```

En este ejemplo, definimos tres variables x, y y z con valores enteros. Luego, utilizamos estos valores para construir una expresión que incluye paréntesis, operadores aritméticos, operadores de comparación y operadores lógicos.

En lugar de utilizar la estructura de control if para evaluar la expresión, simplemente asignamos el resultado de la expresión a una variable llamada resultado. La expresión se evalúa de la misma manera que en el ejemplo anterior.

Finalmente, imprimimos el valor de la variable resultado utilizando la función fmt.Println(). Si la expresión se evalúa como verdadera, se imprimirá true. Si se evalúa como falsa, se imprimirá false.

Declaración If - Else

En Go, la estructura de control if ... else se utiliza para ejecutar un bloque de código si se cumple una condición booleana, y otro bloque de código si la condición no se cumple.

La sintaxis básica de la estructura if ... else es la siguiente:

```
if condicion {
  // código a ejecutar si la condición es verdadera
} else {
  // código a ejecutar si la condición es falsa
}
```

Por ejemplo, se puede utilizar la estructura if ... else para determinar si es mañana, tarde o noche:

```
    t := time.Now()

    if t.Hour() < 12 {
        fmt.Println("¡Mañana!")
    } else if t.Hour() < 17 {
        fmt.Println("¡Tarde!")
    } else {
        fmt.Println("¡Noche!")
    }
```

Estructura de condición If- Else acanzado.

```
    if t := time.Now(); t.Hour() < 12 {
        fmt.Println("¡Mañana!")
    } else if t.Hour() < 17 {
        fmt.Println("¡Tarde!")
    } else {
        fmt.Println("¡Noche!")
    }
```

Declaración Switch

En Go, la declaración switch se utiliza para tomar decisiones basadas en el valor de una expresión. La sintaxis básica de la declaración switch es la siguiente:

```
switch expression {
case value1:
  // código a ejecutar si expression == value1
case value2:
  // código a ejecutar si expression == value2
default:
  // código a ejecutar si ninguno de los casos anteriores se cumple
}
```

La expresión que se pasa a la declaración switch se evalúa y luego se compara con cada uno de los casos. Si el valor de la expresión coincide con uno de los valores de caso, se ejecuta el bloque de código correspondiente. Si ninguno de los casos coincide con el valor de la expresión, se ejecuta el bloque de código dentro del bloque default (opcional).

Aquí hay un ejemplo que muestra cómo se puede utilizar la declaración switch en Go:

```
    //os := runtime.GOOS;
    //os := "linux"
    switch os := runtime.GOOS; os {
    case "linux":
        fmt.Println("Go run -> Linux.")
    case "windows":
        fmt.Println("Go run -> Windows.")
    case "darwin":
        fmt.Println("Go run -> macOS.")
    default:
        fmt.Println("Go run -> Otro SO.")
    }
```

En este ejemplo, la variable dia se compara con cada uno de los casos dentro de la declaración switch. Si el valor de la variable dia coincide con uno de los casos, se imprime un mensaje correspondiente. Si no hay coincidencia, se imprime un mensaje dentro del bloque default.

También se puede utilizar una expresión en lugar de un valor en cada caso, lo que permite evaluar una expresión más compleja:

```
    switch t := time.Now(); {
    case t.Hour() < 12:
        fmt.Println("¡Mañana!")
    case t.Hour() < 17:
        fmt.Println("¡Tarde!")
    default:
        fmt.Println("¡Noche!")
    }
```

Bucle For

En Go, el bucle for es la única estructura de control de repetición disponible. Sin embargo, se puede utilizar de diferentes maneras para lograr diferentes funcionalidades.

Bucle infinito:

Para crear un bucle infinito en Go, se puede usar la sintaxis for {}:

```
for {
    // Código que se repetirá infinitamente
}
```

Bucle con una sola condición:

La sintaxis para crear un bucle con una sola condición en Go es la siguiente:

```
for condición {
    // Código que se repetirá mientras la condición sea verdadera
}
```

Por ejemplo, el siguiente bucle se repetirá mientras i sea menor que 10:

```
for i < 10 {
    fmt.Println(i)
    i++
}
```

Bucle típico de for:

La sintaxis típica de un bucle for en Go es la siguiente:

```
for inicialización; condición; actualización {
    // Código que se repetirá mientras la condición sea verdadera
}
```

Por ejemplo, el siguiente bucle imprimirá los números del 1 al 10:

```
for i := 1; i <= 10; i++ {
    fmt.Println(i)
}
```

En este caso, la inicialización es i := 1, la condición es i <= 10, y la actualización es i++. El bucle se repetirá mientras la condición sea verdadera, e incrementará i en 1 en cada iteración.

Break y continue

En Go, break y continue son palabras clave que se utilizan dentro de los bucles for para controlar el flujo de ejecución.

La palabra clave break se utiliza para salir de un bucle antes de que la condición de finalización se haya alcanzado. Cuando se ejecuta break, el control se transfiere a la siguiente instrucción después del bucle. Por ejemplo:

```
for i := 1; i <= 10; i++ {
    if i == 5 {
        break
    }
    fmt.Println(i)
}
```

En este caso, el bucle for se repetirá hasta que i sea igual a 5. Cuando i sea igual a 5, se ejecutará la instrucción break, lo que detendrá la ejecución del bucle y transferirá el control a la siguiente instrucción después del bucle.

Por otro lado, la palabra clave continue se utiliza para saltar a la siguiente iteración de un bucle sin ejecutar el código restante del cuerpo del bucle para la iteración actual. Cuando se ejecuta continue, el control se transfiere directamente al inicio del bucle para comenzar la siguiente iteración. Por ejemplo:

```
for i := 1; i <= 10; i++ {
    if i%2 == 0 {
        continue
    }
    fmt.Println(i)
}
```

En este caso, el bucle for imprimirá sólo los números impares del 1 al 10. Cuando i es un número par, se ejecutará la instrucción continue, lo que saltará directamente a la siguiente iteración del bucle sin ejecutar el código restante del cuerpo del bucle para la iteración actual.

Uso de funciones

En Go, se puede declarar una función utilizando la siguiente sintaxis:

```
func nombreFuncion(parametro1 tipoParametro1, parametro2 tipoParametro2) tipoRetorno {
    // Código de la función
    return valorRetorno
}
```

Donde:

- `func` : es la palabra clave que se utiliza para declarar una función.
- `nombreFuncion` : es el nombre que se le da a la función.
- `parametro1` , `parametro2` , etc.: son los parámetros que recibe la función, cada uno con su tipo correspondiente separado por coma.
- `tipoRetorno` : es el tipo de dato que devuelve la función. Si la función no devuelve nada, se utiliza la palabra clave void.
- `valorRetorno` : es el valor que devuelve la función si es que tiene un tipo de retorno definido.

Por ejemplo, para declarar una función que sume dos números enteros y devuelva el resultado, se podría utilizar el siguiente código:

```
func hello(name string) string {
    //fmt.Println("Hola, ", name)
    return "Hola, " + name
}
```

En este ejemplo, la función se llama sumar, recibe dos parámetros de tipo int llamados a y b, y devuelve un valor de tipo int que es la suma de a y b.

Funciones que devuelven múltiples valores

En Go, es posible que una función devuelva múltiples valores. Esto se logra simplemente especificando los tipos de los valores de retorno separados por comas en la declaración de la función.

Por ejemplo, la siguiente función llamada dividir toma dos números y devuelve el resultado de la división y el resto de la división como dos valores distintos:

```
func calc(a, b int) (sum, mul int) {
    sum = a + b
    mul = a * b
    return
}
```

En la declaración de la función, se especifica que la función devuelve dos valores de tipo int separados por coma: (int, int).

Dentro de la función, se realiza la división y se calcula el resto y se devuelven ambos valores utilizando la palabra clave return seguida de los valores separados por

coma: return cociente, resto.

Luego, al llamar a esta función, se pueden asignar los valores de retorno a dos variables separadas:

```
 sum, mul := calc(5, 3)
fmt.Println(sum) // Output: 8
fmt.Println(mul) // Output: 15
```

Proyecto de la sección

Desarrollar un programa que permita al usuario jugar el famoso juego "Adivina el número". El programa deberá incluir los siguientes elementos:

1. Generación de un número aleatorio entre 1 y 100 que será el número que el usuario deberá adivinar.
2. Preguntar al usuario que ingrese un número. Verificar si el número ingresado es igual, mayor o menor que el número generado aleatoriamente.
3. Si el número ingresado es igual al número generado, mostrar un mensaje de felicitación y preguntar si el usuario desea volver a jugar.
4. Si el número ingresado es mayor o menor que el número generado, mostrar un mensaje indicando la situación y permitir al usuario ingresar un nuevo número.
5. El usuario tendrá un número limitado de intentos para adivinar el número, por lo que se deberá mostrar cuántos intentos le quedan.
6. Si el usuario no adivina el número después de los intentos permitidos, mostrar un mensaje indicando que el juego ha terminado y preguntar si desea volver a jugar.
7. Si el usuario desea volver a jugar, el programa debe volver a generar un número aleatorio y comenzar el juego nuevamente.

El programa deberá utilizar los siguientes elementos de control de flujo:

1. Generar un número aleatorio usando la biblioteca "math/rand".
2. Utilizar la declaración "if - else" para verificar si el número ingresado es igual, mayor o menor que el número generado aleatoriamente.
3. Utilizar la declaración "for" para permitir al usuario tener un número limitado de intentos para adivinar el número.
4. Utilizar la declaración "break" para salir del bucle "for" si el usuario adivina el número o si se agotan los intentos.
5. Utilizar la declaración "switch" para preguntar al usuario si desea volver a jugar y ejecutar la acción correspondiente.
6. Utilizar funciones para estructurar el código y hacerlo más legible.

Código de proyecto

```
package main

import (
    "fmt"
    "math/rand"
    "time"
)

func main() {
    rand.Seed(time.Now().UnixNano())
    jugar()
}

func jugar() {
    numeroAleatorio := rand.Intn(100) + 1
    var numeroIngresado int
    var intentos int
    const maxIntentos = 10

    for intentos < maxIntentos {
        intentos++
        fmt.Printf("Ingresa un número (intentos restantes: %d): ", maxIntentos-intentos+1)
        fmt.Scanln(&numeroIngresado)

        if numeroIngresado == numeroAleatorio {
            fmt.Println("¡Felicitaciones, adivinaste el número!")
            jugarNuevamente()
            return
        } else if numeroIngresado < numeroAleatorio {
            fmt.Println("El número a adivinar es mayor.")
        } else if numeroIngresado > numeroAleatorio {
            fmt.Println("El número a adivinar es menor.")
        }
    }

    fmt.Println("Se acabaron los intentos. El número era:", numeroAleatorio)
    jugarNuevamente()
}

func jugarNuevamente() {
    var eleccion string
    fmt.Print("¿Quieres jugar nuevamente? (s/n): ")
    fmt.Scanln(&eleccion)

    switch eleccion {
    case "s":
        jugar()
    case "n":
        fmt.Println("¡Gracias por jugar!")
    default:
        fmt.Println("Elección inválida. Inténtalo nuevamente.")
        jugarNuevamente()
    }
}
```

Generar número aleatorio

Go tiene una biblioteca integrada para trabajar con números aleatorios. Esta biblioteca se llama `math/rand`. Para generar un número aleatorio en Go, primero necesitamos crear una fuente de números aleatorios o un generador de números aleatorios. Luego, podemos utilizar el generador para generar números aleatorios.

```
// Crea una nueva fuente de números aleatorios usando el tiempo actual como semilla
rand.Seed(time.Now().UnixNano())

// Genera un número aleatorio entre 0 y 99
randomNumber := rand.Intn(100)

fmt.Println(randomNumber)
```

1. Primero, se llama a la función `time.Now()` que devuelve el tiempo actual en formato `time.Time`.
2. Luego, se llama a la función `UnixNano()` de la estructura time.Time para obtener el tiempo actual en nanosegundos.
3. A continuación, se utiliza el valor obtenido como semilla para la fuente de números aleatorios llamando a la función `rand.Seed()`. Esto inicializa la fuente de números aleatorios con una semilla única que se basa en el tiempo actual.
4. Se utiliza la función `rand.Intn(100)` para generar un número aleatorio entero entre 0 y 99. Esta función devuelve un número entero pseudoaleatorio dentro del rango especificado.
5. El número aleatorio generado se almacena en la variable `randomNumber`.
6. Finalmente, se imprime el número aleatorio generado en la consola utilizando la función `fmt.Println()`.

En resumen, este código genera un número aleatorio entre 0 y 99 y lo imprime en la consola cada vez que se ejecuta, utilizando el tiempo actual como semilla para la fuente de números aleatorios. Esto garantiza que se generen números diferentes cada vez que se ejecuta el programa.

Resumen

En esta sección del curso de programación en Go, hemos explorado el control de flujo. Hemos cubierto varios temas importantes, que incluyen:

- Operadores booleanos: hemos aprendido acerca de los operadores booleanos en Go y cómo utilizarlos para tomar decisiones en nuestro código.
- Declaración If-Else: hemos aprendido cómo utilizar la declaración if-else en Go para tomar decisiones en función de las condiciones en nuestro programa.
- Declaración Switch: hemos aprendido cómo utilizar la declaración switch en Go para tomar decisiones en función de múltiples condiciones.
- Bucle For: hemos aprendido cómo utilizar el bucle for en Go para repetir una sección de código un número determinado de veces.
- Uso de funciones: hemos aprendido cómo utilizar funciones en Go y cómo crear nuestras propias funciones personalizadas.

Además, al final de la sección, hemos tenido la oportunidad de aplicar todo lo que hemos aprendido en un proyecto práctico, donde hemos creado un programa que utiliza los conceptos de control de flujo que hemos cubierto.

En general, esta sección ha sido una introducción muy útil al control de flujo en Go, y proporciona una base sólida para continuar aprendiendo y desarrollando habilidades en este lenguaje de programación.