Resumen sobre Go Clase 1

En el mundo IT donde las tecnologías avanzan y cambian continuamente, los lenguajes de programación juegan un papel crucial. Donde vemos que hay momentos que cierto idioma es preponderante, aparecen otros que irrumpen. Los que ven ese cambio son los que estan a la vanguardia del desarrollo. Hoy se buscan soluciones eficientes, escalables y seguras. Uno de estos lenguajes que vienen ganando relevancia y cumple con lo que comentamos más arriba, es GO (Golang), un lenguaje creado por Google en 2009. En este 2024 se catapultó como una de las mejores opciones para desarrolladores y empresas.

Go:

1. **Simplicidad y continuidad de aprendizaje**: Go fue diseñado con una sintaxis limpia y fácil de leer. Es un lenguaje de bajo nivel que permite control sobre el hardware directamente sin intermediarios, pero con una estructura sencilla que reduce complejidades de lenguajes como C o C++.

2. **Altamente eficiente**: Go es conocido por su eficiencia en la concurrencia y el paralelismo gracias a sus goroutines, permitiendo ejecutar múltiples tareas simultáneamente con un bajo consumo de recursos.

3. **Popularidad en la industria**: Go ha sido adoptado por grandes empresas tecnológicas como Google, Netflix, Uber, Dropbox, entre otras. Su comunidad crece día a día.

4. **Ideal para microservicios y la nube**: Su capacidad para manejar arquitecturas de microservicios lo hace ideal para aplicaciones modulares y escalables.

5. **Gran rendimiento y seguridad**: Go combina rendimiento con seguridad, ofreciendo administración de memoria automática (garbage collection).

6. **Soporte y comunidad activa**: Go cuenta con una comunidad activa, y el respaldo de Google asegura su continuidad y soporte.

7. **Desarrollo web eficiente**: Frameworks como Gin y Echo permiten construir APIs RESTful de manera rápida y eficiente.

¿Dónde se usa Go?

- **Microservicios**
- **Infraestructura y herramientas de sistema** (Kubernetes, Docker)
- **Desarrollo web**

- **Servicios en la nube**
- **Aplicaciones en tiempo real**
- **Big Data y Machine Learning**
- **Juegos**
- **DevOps**
- **APIs de backend**

¿Quién lo usa?

Grandes empresas tecnológicas como Google, Netflix, Uber, Mercado Libre, y muchas otras.

Módulos en Go:

Desde Go 1.11, los módulos de Go permiten gestionar las dependencias sin necesidad de GOPATH. Los archivos `go.mod` y `go.sum` contienen las versiones y hashes de las dependencias. El comando `go mod init` inicializa un nuevo módulo en cualquier carpeta.

Tipos de Datos en Go

Go es un lenguaje de programación estáticamente tipado, lo que significa que el tipo de una variable se determina en tiempo de compilación. Los principales tipos de datos en Go incluyen:

1. Tipos Numéricos

- **int**: Entero con tamaño dependiente de la arquitectura (32 o 64 bits).
- **int8, int16, int32, int64**: Enteros con tamaño específico en bits.
- **uint**: Entero sin signo.
- **uint8, uint16, uint32, uint64**: Enteros sin signo con tamaño específico.
- **float32, float64**: Números en coma flotante.
- **complex64, complex128**: Números complejos.
- **byte**: Alias de uint8.
- **rune**: Alias de int32, usado para representar un solo carácter Unicode.
- **bool**: Representa valores booleanos (`true` o `false`).
- **string**: Una secuencia de caracteres Unicode inmutable. Puede ser concatenada usando el operador `+`.

Constantes

const PI float64 = 3.14159265359 // Constante de tipo float64 (precisión doble) const E float32 = 2.71828 // Constante de tipo float32 (precisión simple) const greeting string = "Hola" // Constante de tipo string const isActive bool = true // Constante de tipo bool const maxUsers int = 100 // Constante de tipo int

Programa Basico

```
package main

import "fmt"

func main() {
	myvariable := "hola mundo"
	fmt.Println(myvariable)
}
```

Print

En Go, tanto `println` como `printf` son funciones que se utilizan para mostrar información por consola, pero tienen diferencias clave en su uso:

1. `println`:
    - Es una función incorporada en Go (builtin), pero no es parte de la librería estándar `fmt`. Su propósito principal es imprimir valores en la consola con un formato básico, añadiendo automáticamente un salto de línea al final.
    - No permite formatear la salida de manera detallada, simplemente imprime los valores que se le pasan separados por un espacio.
    - No es común en producción, ya que está diseñada más para uso interno o depuración.

`2.printf`:

- Es parte del paquete `fmt`, que proporciona funciones más completas para formatear cadenas de texto.
- A diferencia de `println`, `printf` permite especificar un formato detallado utilizando verbos de formato, como `%d` para enteros, `%s` para cadenas, `%f` para números en punto flotante, etc. Además, **no** añade un salto de línea al final automáticamente.
- Es ideal cuando necesitas controlar la forma en que se muestran los datos.

Resumen sobre Go Clase 2

Bloques En Go

En Go, los **bloques** se utilizan para agrupar declaraciones y crear ámbitos de variables, generalmente delimitados por llaves `{}`. A continuación se explican sus funciones principales:

1. **Ámbito de variables**:
   Las variables declaradas dentro de un bloque solo son accesibles dentro de ese bloque. Esto es útil para limitar el alcance de variables y evitar conflictos con otras variables fuera del bloque

   2.Agrupación lógica:
   Los bloques permiten agrupar lógicamente varias declaraciones de código, como dentro de funciones, condicionales o bucles. Bloques En Go

Convertir en Go

Convertir de número a string y viceversa en Go// import "strconv"

En Go, para convertir entre números y cadenas (`string`), puedes usar las siguientes funciones y métodos:

---

num := 123

str := strconv.Itoa(num) // "123"

---

num := 123.45

str := strconv.FormatFloat(num, 'f', 2, 64)

---

str := "123"

num, err := strconv.Atoi(str)

if err != nil { // Manejo del error }

---

str := "123.45"

num, err := strconv.ParseFloat(str, 64)

if err != nil { // Manejo del error }

Ingreso de datos por teclado

En Go, puedes ingresar datos por teclado utilizando el paquete `fmt` para leer entradas del usuario. Aquí te dejo un resumen de cómo hacerlo:

Texto:

var input string

fmt.Print("Ingresa un texto: ")

fmt.Scanln(&input) // Lee una línea de entrada y la almacena en la variable 'input'

fmt.Println("Has ingresado:", input)

Numeros:

var num int

fmt.Print("Ingresa un número entero: ")

fmt.Scanln(&num) // Lee una línea de entrada y la almacena como entero

fmt.Println("El número ingresado es:", num)

Operadores

Operadores Relacionales:

- == : Igual a
- != : No igual a
- > : Mayor que
- < : Menor que
- >= : Mayor o igual que
- <= : Menor o igual que

Operadores Lógicos:

- && : Y lógico (AND)
- || : O lógico (OR)
- ! : Negación lógica (NOT)

Sentencias de Control

En Go, las sentencias de control permiten dirigir el flujo de ejecución del programa. A continuación, te detallo las principales sentencias de control:

1. `if`

La sentencia `if` se usa para ejecutar un bloque de código si se cumple una condición. Se puede combinar con `else` para manejar el caso en que la condición no se cumpla.

2. `else if`

Permite probar múltiples condiciones. Si la primera condición no es verdadera, se evaluarán las siguientes con `else if`. Finalmente, se puede utilizar un `else` para cubrir todos los demás casos.

3. `switch`

La sentencia `switch` se utiliza para simplificar la evaluación de múltiples condiciones. Compara el valor de una expresión con varias opciones (`case`) y ejecuta el código asociado al caso correspondiente. Tiene un `default` para manejar todos los otros valores no cubiertos por los casos.

4. `for`

Es el único bucle en Go y puede funcionar como un bucle `while` o `do-while` en otros lenguajes. Se usa para repetir un bloque de código un número determinado de veces o mientras una condición sea verdadera.

`break`

Termina de forma inmediata el bucle o la sentencia `switch` en la que se encuentra.

`continue`

Salta la iteración actual de un bucle y continúa con la siguiente iteración

Tipos de Colecciones; Conjuntos

Arrays:

- **Definición**: Un array en Go es una estructura de datos que almacena un número fijo de elementos del mismo tipo. El tamaño del array es parte de su tipo, lo que significa que los arrays no pueden cambiar de tamaño una vez que se han definido.
- **Características**:
    - Su tamaño es fijo y se especifica en el momento de la declaración.
    - Los elementos de un array se acceden por su índice, comenzando desde 0.
    - Los arrays en Go se pasan por valor, lo que significa que al pasar un array a una función, se crea una copia.

Slices:

- **Definición**: Un slice es una estructura más flexible que los arrays, ya que su tamaño puede cambiar dinámicamente. Un slice es una referencia a una porción (o todo) de un array.
- **Características**:
    - No tienen un tamaño fijo, pueden crecer o reducirse.
    - Se componen de un puntero a un array subyacente, una longitud y una capacidad (que puede ser mayor que la longitud actual).
    - Los slices son más utilizados que los arrays debido a su flexibilidad.
    - Se pasan por referencia, lo que significa que los cambios en un slice pueden afectar el array subyacente.
    - Tienen funciones y operaciones integradas como `append` (para añadir elementos) y `len` (para obtener la longitud).

Maps:

- **Definición**: Un map en Go es una estructura de datos que asocia claves (keys) con valores. Es similar a un diccionario en otros lenguajes.
- **Características**:
    - Las claves deben ser de un tipo que sea comparable (por ejemplo, `int`, `string`, etc.).
    - Los valores pueden ser de cualquier tipo.
    - No tiene un tamaño fijo, los elementos pueden ser añadidos o eliminados dinámicamente.
    - El acceso a los valores se realiza a través de las claves, y si una clave no existe, devuelve el valor por defecto del tipo del valor.

- Los maps en Go son eficientes y se utilizan para búsquedas rápidas de pares clave-valor.

Estos tres tipos de estructuras de datos son fundamentales en Go para organizar y manejar conjuntos de datos de forma eficiente y flexible.

Resumen sobre Go Clase 3

Funciones

En Go, las funciones son bloques de código reutilizables que se pueden definir para realizar tareas específicas. A continuación, te explico los conceptos clave sobre funciones definidas por el usuario:

1. Definición de funciones

Las funciones en Go se definen utilizando la palabra clave `func`, seguida del nombre de la función, una lista de parámetros (si los hay) y el tipo de valor que la función devuelve (si aplica). Las funciones pueden o no devolver valores, y pueden tener parámetros opcionales o requeridos.

2. Parámetros

Las funciones pueden aceptar cero o más parámetros. Cada parámetro tiene un nombre y un tipo. Si hay varios parámetros del mismo tipo consecutivo, se pueden agrupar bajo un solo tipo para simplificar la sintaxis. Los parámetros son pasados por valor, a menos que se utilicen punteros para permitir la modificación del valor original.

3. Valores de retorno

Las funciones en Go pueden devolver uno o más valores. Si hay más de un valor de retorno, estos se agrupan entre paréntesis. Los valores de retorno también pueden ser nombrados dentro de la declaración de la función, lo que hace que el código sea más claro y permite devolver automáticamente dichos valores al final de la función.

4. Funciones sin retorno

Las funciones no están obligadas a devolver un valor. Si no se especifica ningún tipo de retorno, la función simplemente ejecuta su bloque de código y finaliza.

Structs

En Go, los structs (estructuras) son tipos de datos compuestos que permiten agrupar múltiples campos bajo un solo nombre. Cada campo de un `struct` puede tener un nombre y un tipo, y pueden ser de tipos diferentes.

Características de los Structs en Go:

1. **Definición**: Un `struct` es una colección de campos (o propiedades) que pueden tener tipos diferentes. Se utiliza para representar datos complejos, agrupando variables relacionadas.
2. **Campos**: Los campos de un `struct` tienen un nombre y un tipo. Se pueden definir tantos campos como sea necesario, y cada campo es accesible por su nombre.
3. **Acceso a campos**: Los campos se acceden utilizando la notación de punto (`.`). Si tienes una instancia de un `struct`, puedes leer o modificar sus campos directamente usando este operador.
4. **Comparación**: Los `structs` pueden ser comparados usando los operadores `==` y `!=` siempre que todos los campos del `struct` sean comparables. Los `structs` con campos no comparables (como `slices` o `maps`) no pueden compararse directamente.
5. **Punteros y referencias**: En Go, puedes usar punteros a `structs` para evitar copiar estructuras grandes al pasarlas como parámetros a funciones. Esto permite modificar los campos del `struct` original desde la función.
6. **Métodos asociados**: Los `structs` pueden tener métodos asociados. Esto significa que puedes definir funciones que operan sobre una instancia de un `struct` específico, asociando el método a la estructura.
7. **Anidación**: Los `structs` pueden contener otros `structs` como campos. Esto permite crear estructuras jerárquicas complejas.
8. **Campos exportados e inexportados**: Si el nombre de un campo comienza con una letra mayúscula, ese campo es exportado y puede ser accedido desde otros paquetes. Si comienza con una letra minúscula, es inexportado y solo es accesible dentro del mismo paquete.

Los `structs` en Go son muy utilizados para definir tipos de datos personalizados y organizar mejor los datos dentro de un programa, facilitando la creación de estructuras complejas y claras.

Resumen sobre Go Clase 4

Structs segunda parte

Structs en Go segunda parte

Una **struct** en Go es un tipo de dato compuesto que permite agrupar variables bajo un mismo nombre, las cuales pueden tener diferentes tipos. Las structs se utilizan para modelar entidades más complejas que requieren almacenar varias propiedades. Cada propiedad dentro de una struct se conoce como un **campo**, y cada campo tiene un nombre y un tipo de dato.

- **Definición**: Las structs permiten encapsular datos y agrupar atributos relacionados en un solo tipo. Se definen utilizando la palabra clave `struct` y permiten un diseño más organizado y modular en la programación.

- **Métodos**: Go permite asociar métodos a structs, lo que significa que una struct puede tener comportamiento además de datos. Los métodos se definen sobre un receptor (la instancia de la struct) y permiten manipular o acceder a los datos dentro de la misma.

Generics en Go

Los **generics** en Go permiten escribir funciones y tipos de datos que pueden trabajar con cualquier tipo de dato sin tener que especificarlo de antemano. Este es un concepto clave para mejorar la reutilización de código y evitar la duplicación, ya que permite crear algoritmos que funcionan con diferentes tipos sin comprometer el rendimiento.

- **Parámetros de Tipo**: Con los generics, las funciones y tipos pueden definir **parámetros de tipo**. Esto se hace utilizando una sintaxis que indica que un valor o un tipo será genérico, es decir, podrá adaptarse a diferentes tipos en tiempo de compilación.

- **Ventajas**: Los generics permiten evitar la duplicación de código, proporcionando una manera más flexible de definir funciones y tipos que trabajen con múltiples tipos de datos. Esto promueve un diseño más limpio y un código más fácil de mantener.

- **Limitaciones**: Aunque los generics son poderosos, no se comportan exactamente como los genéricos en otros lenguajes como Java o C#. Por ejemplo, en Go no es posible realizar operaciones arbitrarias sobre los tipos genéricos a menos que se hayan definido las restricciones adecuadas.

En resumen, **las structs** son fundamentales para organizar datos complejos en Go, mientras que **los generics** mejoran la flexibilidad y la reutilización del código al permitir que funciones y tipos trabajen con diferentes tipos de datos.

Punteros

Concepto de Punteros

Un **puntero** es una variable que almacena la dirección de memoria de otra variable. En lugar de contener directamente un valor, un puntero "apunta" a la ubicación en la memoria donde está almacenado ese valor.

- **Declaración de un puntero**: En Go, un puntero se declara utilizando el símbolo `*` seguido del tipo de dato que apunta. Por ejemplo, un puntero a un entero se declara como `*int`.
- **Operador de dirección (`&`)**: Para obtener la dirección de memoria de una variable (es decir, crear un puntero que apunte a esa variable), se usa el operador &. Esto permite asignar la referencia de una variable a un puntero.
- **Operador de desreferencia (`*`)**: Para acceder al valor almacenado en la dirección de memoria a la que apunta un puntero, se utiliza el operador de desreferencia (`*`). Esto permite leer o modificar el valor al que apunta el puntero.

Ventajas de los Punteros

- **Eficiencia**: Los punteros permiten trabajar con grandes estructuras de datos sin necesidad de copiarlas. Al pasar un puntero a una función, se pasa una referencia a los datos, no una copia de los mismos, lo que mejora el rendimiento.
- **Modificación de valores**: Al usar punteros, es posible modificar el valor original de una variable desde diferentes lugares en el código. Esto es útil para funciones que necesitan modificar directamente los valores que reciben como argumentos.

Punteros y Tipos en Go

Go es un lenguaje que, por diseño, evita la complejidad de la manipulación directa de memoria que se ve en otros lenguajes como C o C++. Algunas de las características clave de los punteros en Go incluyen:

- **Punteros nulos**: Un puntero que no apunta a ninguna dirección válida se inicializa con el valor `nil`, que indica que no tiene un valor asignado.
- **Pasaje de parámetros por referencia**: En Go, las variables se pasan a las funciones por valor, lo que significa que se crea una copia de la variable original. Sin embargo, si pasas un puntero a una función, la función puede modificar el valor original de la variable.

Limitaciones y Seguridad

- **Go no permite aritmética de punteros**: A diferencia de lenguajes como C o C++, en Go no se puede realizar aritmética de punteros (es decir, incrementar o decrementar direcciones de memoria), lo que aumenta la seguridad al evitar errores de manipulación de memoria.
- **Recogida de basura (garbage collection)**: Go tiene un recolector de basura integrado, lo que significa que la administración de memoria es más segura y los punteros no necesitan ser liberados manualmente.

En resumen, los punteros en Go permiten manipular datos de manera eficiente al pasar referencias en lugar de valores, lo que reduce el uso de memoria y permite la modificación directa de los datos. Sin embargo, Go ofrece una administración de memoria segura, limitando la complejidad asociada con la manipulación directa de punteros.

Resumen sobre Go Clase 5

Bibliotecas

La biblioteca estándar de Go organiza el código en paquetes, cada uno diseñado para realizar tareas específicas y reutilizables en otros programas. Todo programa en Go debe tener un paquete principal llamado `main`, que incluye la función `main` como punto de inicio.

Esta biblioteca ofrece herramientas para operaciones comunes, como la entrada/salida (con el paquete `fmt`), manipulación de cadenas, conexiones HTTP, y más. Para usar un paquete, se importa al inicio del archivo, y solo son accesibles públicamente las funciones que inician con mayúscula.

Los paquetes están organizados en carpetas, y algunos requieren rutas completas al importarlos (por ejemplo, `net/http`). Para importar varios paquetes, es posible agruparlos en una sola declaración. Además, Go permite asignar alias a paquetes para evitar colisiones de nombres.

Por ahora, se utiliza la biblioteca estándar, pero más adelante se cubrirá cómo crear paquetes propios para reutilización y cómo usar paquetes de terceros.

Strings

El paquete strings de la biblioteca estándar de Go ofrece funciones para manipular cadenas de caracteres.

Conversión de Mayúsculas/Minúsculas: Permite convertir una cadena a mayúsculas o minúsculas completas.

Buscar Subcadenas: Con Index, se puede verificar si una palabra está contenida en otra y obtener la posición donde comienza.

Repetición de Cadenas: La función Repeat permite repetir una palabra un número específico de veces.

Reemplazo de Subcadenas: Con Replace, se puede cambiar una subcadena por otra en toda la cadena o solo en las primeras apariciones especificadas.

Go ofrece más de 40 funciones en el paquete strings para distintas manipulaciones, como dividir cadenas (Split), verificar si una cadena contiene otra (Contains), contar ocurrencias (Count), y comparar prefijos (HasPrefix).

Strconv

La conversión entre cadenas de texto (strings) y tipos numéricos es una actividad común en el desarrollo de programas. Go facilita esta tarea a través de su biblioteca estándar, que incluye un paquete especializado para realizar conversiones de datos de tipo string a numéricos y viceversa, cubriendo las necesidades más habituales en esta área.

Otras bibliotecas

`time`

El paquete `time` proporciona funcionalidades para trabajar con fechas, horas, y duraciones. Permite:

- Obtener la fecha y hora actuales (`time.Now()`).

- Formatear y analizar fechas y horas (`Format` y `Parse`).
- Realizar operaciones entre fechas (`Add`, `Sub`) y medir el tiempo transcurrido.
- Utilizar temporizadores y intervalos de tiempo (`Ticker` y `Timer`), útiles para programar eventos.

`math`

El paquete `math` contiene funciones matemáticas generales y constantes. Permite:

- Realizar operaciones como raíces cuadradas, potencias, logaritmos, trigonometría, y exponenciales.
- Usar constantes matemáticas (`Pi`, `E`, entre otras).
- Trabajar con funciones especiales como `Abs` (valor absoluto), `Ceil` (redondeo hacia arriba), `Floor` (redondeo hacia abajo), y `Round` (redondeo al entero más cercano).

`math/rand`

El subpaquete `math/rand` se especializa en la generación de números aleatorios. Permite:

- Generar números aleatorios de diferentes tipos (`Int`, `Float`, etc.).
- Controlar la secuencia de generación de números aleatorios usando semillas (`Seed`), lo que permite reproducir secuencias de aleatoriedad.
- Generar números aleatorios en rangos específicos y utilizar distribuciones personalizadas para simulaciones y pruebas.

Estas bibliotecas son fundamentales para el manejo de tiempo, cálculos matemáticos y generación de aleatoriedad en Go.

Creación de paquetes propios

En Go, crear paquetes propios permite organizar y reutilizar código en diferentes partes de un proyecto o incluso en otros proyectos. Aquí tienes un resumen de cómo hacerlo:

1. Estructura de Directorios

Los paquetes en Go están organizados en directorios. Para crear un paquete propio, crea una carpeta dentro de tu proyecto con el nombre del paquete.

2. Definir el Paquete

Cada archivo `.go` en el directorio del paquete debe comenzar con la declaración `package` seguido del nombre del paquete.

3. Importar y Usar el Paquete

Para utilizar el paquete en otro archivo (como `main.go`), debes importarlo. Go busca el paquete en el mismo directorio de trabajo o en el `GOPATH`.

Buenas Prácticas al crear un paquete

- **Nombrado Consistente**: Usa nombres claros y consistentes para los paquetes y archivos.
- **Exportación**: Solo exporta las funciones necesarias (en mayúsculas).
- **Documentación**: Agrega comentarios a las funciones exportadas para facilitar su uso.

Este enfoque permite que el código sea modular, mantenible y reutilizable en cualquier proyecto de Go.

Resumen sobre Go Clase 6

Programación Concurrente

Las Goroutines en Go permiten la programación concurrente, ejecutando funciones de manera simultánea en lugar de secuencial. A diferencia de las llamadas a funciones tradicionales, una Goroutine permite que el programa continúe su flujo sin esperar a que la función finalice, lo cual es útil para procesos largos o que requieren mucho tiempo, como búsquedas en disco o consultas de red.

Ventajas de las Goroutines:

1. **Ejecución sin bloqueo**: El programa no queda detenido por tareas complejas o de alto tiempo de espera, como la búsqueda en un disco duro.
2. **Aprovechamiento de múltiples procesadores**: En sistemas con varios núcleos, las Goroutines pueden ejecutarse en paralelo, acelerando el rendimiento general.

Este enfoque hace que los programas sean más eficientes y rápidos, especialmente en tareas que involucran operaciones de E/S (entrada/salida) o procesos de cómputo intensivo.

Goroutines - tipo WaitGroup

El paquete sync dispone de un struct llamado WaitGroup con una serie de métodos.

Este struct y sus métodos nos permiten llevar un conteo de las goroutines en ejecución.

Dispone de los siguientes métodos:

- Add : incrementa la cantidad de goroutines activos.
- Done : Resta en uno la cantidad de goroutines activas.
- Wait : No finaliza esta función hasta que el contador de goroutines llegue a cero.
-

Goroutines - NumCPU y GOMAXPROCS

La eficiencia en la ejecución de programas escritos en Go empleando Goroutines se ve beneficiada cuando se ejecutan en equipos con múltiples procesadores físicos.

Si queremos conocer la cantidad de procesadores físicos de tu computadora podemos acceder a la función NumCPU

Goroutines - Channels

Como hemos visto las Goroutines se ejecutan en forma concurrente (si hay varios procesadores pueden estar ejecutándose en forma simultanea), en el problema anterior cuando ordenamos dos vectores en forma concurrente no teníamos ningún problemas de sincronización ya que son dos problemas completamente independientes.

En muchas situaciones la ejecución de una Goroutine depende del estado de ejecución de otra Goroutine.

Cuando dos o más Goroutines deben comunicarse o acceder a recursos comunes el lenguaje Go propone el mecanismo de Channels (canales) para que se comuniquen e intercambien recursos.

Un Channel es un mecanismo de comunicación que permite a una Goroutine enviar valores a otro goroutine.

Adicional New y Make

New()

La función `new()` en Go es bastante específica en su uso y, en realidad, no es tan común en el código típico de Go, ya que existen formas más idiomáticas de crear instancias de tipos de datos. Sin embargo, algunas situaciones donde `new()` podría ser útil o preferible:

Asignación de memoria para un puntero a un valor cero de un tipo simple:

1. Si necesitas un puntero a un tipo básico (como `int`, `float64`, `bool`, etc.) inicializado en su valor cero, puedes usar `new()`. Por ejemplo, `new(int)` te da un puntero a un entero con valor `0`.
2. Si trabajas con estructuras grandes y prefieres usar punteros para evitar copias innecesarias, `new()` te ayuda a crear un puntero directamente. Esto es útil cuando deseas manipular una estructura grande en funciones sin duplicar datos.
3. Al construir paquetes o funciones que deben devolver punteros a estructuras, `new()` puede resultar útil para crear un puntero a un valor en cero sin necesidad de inicializar un valor específico. Este caso es típico en código que necesita flexibilidad en la creación de tipos arbitrarios.
4. En algunos casos, especialmente al trabajar con funciones de API o cuando un valor puede ser opcional, `new()` puede ayudar a indicar explícitamente que un valor existe aunque esté en su estado cero. Esto permite diferenciar entre "valor inexistente" (nil) y "valor existente con valor cero".

Usa `new()` si específicamente necesitas un **puntero** a tipo o estructura nicializada en cero y quieres evitar crear el puntero manualmente con `&`. (`p := New(int)`)

Usa el tipo o la estructura si solo necesitas el **valor directo** de la estructura. Puedes obtener un puntero a este valor usando `&` cuando sea necesario (var x int; `p:= &x;`)

Make()

La función `make()` en Go es utilizada para crear y inicializar algunos tipos de datos específicos: slices (porciones), maps (mapas) y channels (canales). Estos tipos de datos necesitan más que solo asignación de memoria; requieren una inicialización adicional que `make()` proporciona automáticamente.

¿Qué hace `make()`?

- **Inicializa** y asigna la memoria necesaria para **slices**, **maps** y **channels**.
- Devuelve un valor ya **inicializado y listo para usarse**, no un puntero.

- La memoria y los punteros internos de estos tipos se configuran en el momento de llamar a `make()`.

¿Por qué `make()` es necesario solo para estos tipos?

- A diferencia de otros tipos de datos en Go, **slices**, **maps** y **channels** son tipos de referencia. Esto significa que tienen estructuras de datos internas (como punteros y capacidades) que necesitan ser configuradas para que funcionen correctamente.
- `make()` asegura que se asignen e inicialicen estas estructuras internas, haciendo que el slice, map o canal esté listo para su uso inmediato.

Con `make()` puedes definir la capacidad de antemano y evitar realocaciones frecuentes de memoria si estás trabajando con datos grandes.

Supón que necesitas un slice con una capacidad fija pero que empiece vacío. Puedes inicializarlo con `make()` especificando tanto la longitud como la capacidad. Esto es ventajoso cuando sabes que el slice necesitará expandirse sin necesidad de realocar memoria, ya que la capacidad ya está definida. Si lo definis tradicionalmente cada vez que se expanda pedira más memoria.

Con `maps`, el uso de `make()` no solo es conveniente sino **obligatorio** si deseas asignar pares clave-valor inmediatamente. Si intentas agregar elementos a un mapa no inicializado, el programa fallará.

Para canales, `make()` es igualmente necesario. Si intentas enviar o recibir datos en un canal que no fue inicializado con `make()`, el programa fallará.

Diferencias clave entre `make()` y `new()`

- `make()`: Solo es para slices, maps y channels; inicializa estructuras internas y devuelve un valor inicializado.
- `new()`: Es más general y se usa para cualquier tipo de dato, asigna memoria y devuelve un puntero, sin inicialización adicional.

Errores en Golang

Es un caso bastante común que recibamos un número como cadena de texto, pero debamos realizar algún proceso con él que requiera tratarlo como un entero, algunos lenguajes de programación permiten hacer esto de manera implícita, operaciones como `"2" + 4` son totalmente válidas, aunque el resultado puede no ser el esperado. Hablando específicamente de Go, como lenguaje estrictamente tipado, requiere que la conversión se realice explícitamente.

```
numero, err := strconv.Atoi(cadena)
```

Pero esto rara vez sucede en casos reales, normalmente no se tiene total control del contenido de la la cadena, lo que significa que existe la posibilidad de que se produzca un error y el programa debería estar preparado para manejarlo.

En una variable llamada `err`, es común usar ese identificador para los errores, evaluamos si su valor es diferente de `nil`, debido a que de cumplirse esa condición significaría que ha ocurrido un error, dentro de la condicional podemos tomar las acciones que consideremos pertinentes, en este caso se imprime un mensaje por consola que muestra el contenido del error y el `return` termina la ejecución del programa, después de todo no es recomendable continuar un proceso que depende de un valor en el que ha tenido lugar un error.

En algunos casos, es posible que queramos tomar medidas más fuertes como detener la ejecución total del programa, podríamos generar conscientemente un `panic`, y frenar la ejecucion de nuestro programa.

Resumen sobre Go Clase 7

Adicional Interfaz

Una interfaz en programación es un contrato que define un conjunto de métodos que una clase debe implementar, sin proporcionar la implementación de esos métodos. Es una forma de especificar cómo deben comportarse las clases que la implementan. En Golang no existen las clases, lo más parecido en comportamiento son las structs.

Métodos Abstractos: Una interfaz puede contener métodos abstractos, que son métodos sin implementación. Las structs que las usan deben proporcionar la implementación de estos métodos.

Las interfaces son una herramienta fundamental en la programación de objetos y estructuras, ya que promueven buenas prácticas de diseño, como la separación de responsabilidades y la reutilización de código.

Scripting

1. os

- `os.Create(name string) (*File, error)`
  - **Propósito**: Crea un nuevo archivo con el nombre especificado y devuelve un puntero al archivo.
  - **Uso**: Para crear archivos vacíos o sobrescribir archivos existentes.
- `os.Remove(name string) error`
  - **Propósito**: Elimina el archivo o directorio especificado.
  - **Uso**: Para borrar archivos o directorios no deseados.
- `os.Mkdir(name string, perm FileMode) error`
  - **Propósito**: Crea un nuevo directorio con los permisos especificados.
  - **Uso**: Para crear nuevas carpetas en el sistema de archivos.
- `os.Open(name string) (*File, error)`
  - **Propósito**: Abre el archivo especificado y devuelve un puntero al archivo.
  - **Uso**: Para leer archivos existentes.
- `os.Stat(name string) (FileInfo, error)`
  - **Propósito**: Devuelve información sobre el archivo o directorio, como tamaño y permisos.
  - **Uso**: Para obtener metadatos sobre archivos.

2. OS/FILE ; IO

- **os.ReadFile(filename string) ([]byte, error)**
  - **Propósito**: `Lee el contenido de un archivo y lo devuelve como un slice de bytes.`
  - **Uso**: `Para cargar archivos completos en memoria.`
- **os.WriteFile(filename string, data []byte, perm FileMode) error**
  - **Propósito**: Escribe datos en un archivo, creando el archivo si no existe.
  - **Uso**: Para guardar datos en archivos de manera sencilla.

- **io.Copy(dst Writer, src Reader) (int64, error)**
  - **Propósito**: Copia datos de un Reader a un Writer.
  - **Uso**: Para clonar el contenido de un archivo a otro.

3. `os/exec`

- `exec.Command(name string, arg ...string) *Cmd`
  - **Propósito**: Crea un nuevo comando para ejecutar el programa especificado con los argumentos dados.
  - **Uso**: Para ejecutar comandos de shell o programas externos.
- `(*Cmd).CombinedOutput() ([]byte, error)`
  - **Propósito**: Ejecuta el comando y devuelve la salida estándar y la salida de error combinadas.
  - **Uso**: Para obtener la salida de un comando en una sola llamada.
- `(*Cmd).Run() error`
  - **Propósito**: Ejecuta el comando sin capturar su salida.
  - **Uso**: Para ejecutar comandos donde la salida no es necesaria.

4. `path/filepath`

- `filepath.Base(path string) string`
  - **Propósito**: Devuelve el último elemento de una ruta, que generalmente es el nombre del archivo.
  - **Uso**: Para extraer el nombre del archivo de una ruta completa.
- `filepath.Dir(path string) string`
  - **Propósito**: Devuelve el directorio padre de la ruta especificada.
  - **Uso**: Para obtener la carpeta que contiene el archivo.
- `filepath.Join(elem ...string) string`
  - **Propósito**: Une varios elementos de ruta en una sola ruta, manejando correctamente los separadores de directorio.
  - **Uso**: Para construir rutas de manera portátil.
- `filepath.Walk(root string, walkFn WalkFunc) error`
  - **Propósito**: Recorre un árbol de directorios y ejecuta una función en cada archivo y directorio encontrado.
  - **Uso**: Para realizar operaciones en todos los archivos dentro de un directorio.

Resumen

Estos comandos y funciones en Go te permitirán realizar diversas tareas relacionadas con la manipulación de archivos y la ejecución de comandos, similar a las capacidades de Python en los módulos mencionados.

Peticion HTTP

Usamos `http.Get` para hacer una solicitud GET a una API externa que devuelve información sobre el dólar en formato JSON. Al recibir la respuesta, leemos el cuerpo (`resp.Body`) y lo deserializa en un slice de mapas (`[]map[string]interface{}`) mediante `json.Unmarshal`. Luego, extrae e imprime el valor de la clave `"venta"` del primer objeto en el slice, que representa el precio de venta del dólar. Los paquetes `fmt`, `encoding/json`, `io`, y `net/http` se usan para manejar la salida, la deserialización de JSON, la entrada/salida y la solicitud HTTP, respectivamente.

Conexion a BBDD

En Go para conectar a una base de datos MySQL y realiza una inserción de datos. Utilizamos `sql.Open` para establecer la conexión, pasando las credenciales y detalles de la conexión. Tras verificar la conexión con `db.Ping`, definimos una estructura `Producto` con los campos `Nombre` y `Precio`. Luego, ejecuta una consulta SQL de inserción (`INSERT INTO`) para agregar un nuevo producto. Utiliza `db.Exec` para insertar y `result.LastInsertId()` para obtener e imprimir el ID del producto recién insertado. Los paquetes `fmt`, `log`, y `database/sql` se usan para manejar la salida, los errores y la conexión SQL respectivamente.

Resumen sobre Go Clase 8 y 9

Arquitectura Web y Principios de REST

Arquitectura Web:

La arquitectura web se refiere a la forma en que se estructuran y organizan los componentes de una aplicación web. Existen varias arquitecturas, siendo una de las más comunes la **arquitectura en capas**, que se divide en:

- **Capa de presentación (Frontend)**: Es la interfaz que el usuario ve e interactúa. Aquí se gestionan las solicitudes del usuario y se presentan las respuestas del servidor. Esta capa puede estar formada por tecnologías como HTML, CSS, JavaScript y frameworks como React, Angular, o Vue.js.
- **Capa de lógica de negocio (Backend)**: Esta capa maneja la lógica principal de la aplicación. Procesa las solicitudes recibidas de la capa de presentación, interactúa con las bases de datos o servicios externos, y devuelve respuestas. Esta capa está construida con lenguajes como Go, Python, Java, entre otros.
- **Capa de datos**: Es la responsable del almacenamiento, recuperación y manejo de la información. Utiliza bases de datos relacionales (como MySQL, PostgreSQL) o no relacionales (como MongoDB).

REST (Representational State Transfer):

REST es un estilo arquitectónico para diseñar servicios web. Sus principios fundamentales incluyen:

- **Recursos Identificados por URLs**: Cada recurso en una API REST debe tener una URL única que lo identifique. Por ejemplo, `https://api.ejemplo.com/usuarios/123`.
- **Métodos HTTP**: Los métodos estándar de HTTP son utilizados para realizar operaciones sobre los recursos:
    - `GET`: Recuperar información de un recurso.
    - `POST`: Crear un nuevo recurso.

    - `PUT`: Actualizar un recurso existente.
    - `DELETE`: Eliminar un recurso.
- **Stateless (Sin estado)**: Cada solicitud HTTP debe ser independiente, y no debe depender de ninguna información almacenada en el servidor entre peticiones. Toda la información necesaria para procesar la solicitud debe ir incluida en la misma.
- **Representación de los recursos**: Los recursos pueden ser representados en distintos formatos, siendo los más comunes **JSON** o **XML**.
- **Cacheable (Cachable)**: Las respuestas deben ser explícitamente etiquetadas como cacheables o no cacheables, lo que mejora la eficiencia al reducir la necesidad de consultas repetidas al servidor.
- **Interfaz uniforme**: Los servicios REST deben tener una interfaz uniforme que facilite su uso, lo que implica tener un diseño predecible y consistente de los recursos y los métodos HTTP.

APIs con Go

Implementación de Endpoints:

En una API RESTful, un "endpoint" es una URL que está asociada a una operación específica. Cada endpoint tiene un método HTTP (GET, POST, PUT, DELETE) y realiza una acción en un recurso o conjunto de recursos.

Ejemplo de un endpoint típico:

- `GET /usuarios`: Devuelve una lista de todos los usuarios.
- `POST /usuarios`: Crea un nuevo usuario.
- `PUT /usuarios/{id}`: Actualiza los datos de un usuario con el ID especificado.
- `DELETE /usuarios/{id}`: Elimina el usuario con el ID especificado.

Manejo de Errores:

El manejo de errores es crucial para proporcionar una experiencia de usuario adecuada y asegurar que las respuestas de la API sean claras y consistentes. En REST, los errores deben ser comunicados con códigos de estado HTTP adecuados, como:

- `404 Not Found`: El recurso solicitado no se encontró.
- `400 Bad Request`: La solicitud es incorrecta o incompleta.
- `500 Internal Server Error`: Hubo un error en el servidor al procesar la solicitud.

Además, es importante proporcionar un mensaje de error detallado en el cuerpo de la respuesta.

Framework Gin en Go

Gin es un framework web minimalista y rápido para Go (también conocido como Golang), que facilita la construcción de aplicaciones y APIs web de alto rendimiento. Gin se destaca por su simplicidad, flexibilidad, y excelente manejo de rendimiento, lo que lo convierte en una opción popular para desarrolladores que crean APIs RESTful.

Principales Características de Gin:

1. **Alto rendimiento**: Gin está diseñado para ser rápido, y se encuentra entre los frameworks web más rápidos disponibles para Go. Su eficiencia es clave para aplicaciones que requieren alta concurrencia y respuestas rápidas.
2. **Enrutamiento eficiente**: Gin proporciona un enrutador altamente optimizado que permite manejar rutas de manera rápida y sencilla. Utiliza árboles de enrutamiento para una búsqueda eficiente de las rutas que permite manejar cientos de rutas con mínimo costo.
3. **Middlewares**: Gin soporta middlewares que pueden ser utilizados para realizar tareas comunes como la autenticación, validación de entradas, logging, etc. Los middlewares son funciones que se ejecutan antes o después del manejo de la solicitud.
4. **Manejo de errores**: El manejo de errores es sencillo con Gin. Permite capturar errores globales de la aplicación y manejar respuestas coherentes con códigos de estado HTTP apropiados.
5. **Soporte para JSON y XML**: Aunque el formato más común es JSON, Gin también permite manejar otros tipos de respuestas, como XML, lo cual es útil para interactuar con diferentes clientes que pueden requerir otros formatos de datos.
6. **Soporte para renderizado de plantillas**: Gin ofrece soporte para la renderización de plantillas HTML, lo que lo hace adecuado para aplicaciones web tradicionales que requieren mostrar páginas HTML.
7. **Validación de datos**: El framework incluye funcionalidades para la validación de datos dentro de los parámetros de las solicitudes. Esto es especialmente útil cuando se reciben datos desde el cliente, asegurando que cumplen con las reglas definidas (como tipos de datos, longitud mínima, etc.).

8. **Documentación de API**: A través de la integración con herramientas como **Swagger**, Gin facilita la creación de documentación automática de las APIs, lo cual es vital para las mejores prácticas en el desarrollo de servicios RESTful.

Adicional Templates con Gin

En **Gin**, los templates permiten renderizar vistas HTML dinámicas, insertando datos desde el servidor en plantillas HTML. Gin utiliza el paquete **`html/template`** de Go para manejar las plantillas.

1. **Cargar Plantillas:**
   - Se cargan utilizando **`LoadHTMLGlob`** o **`LoadHTMLFiles`** para leer archivos `.html` desde un directorio o archivo específico.
   - Ejemplo: `r.LoadHTMLGlob("templates/*")` carga todas las plantillas desde el directorio `templates`.
2. **Renderizar Plantillas:**
   - Se usa **`c.HTML(statusCode, "template.html", data)`** para renderizar una plantilla y pasarle datos.
   - Los datos se pasan como un mapa, por ejemplo: `gin.H{"title": "Página de Inicio"}`.
3. **Sintaxis de Plantillas:**
   - **`{{ }}`**: Utilizado para insertar datos dentro de las plantillas HTML. Ejemplo: `{{ .title }}` inserta el valor de `title`.
4. **Plantillas Parciales:**
   - Puedes tener un layout común con elementos como cabecera y pie de página, y luego insertar contenido específico en cada plantilla usando bloques como `{{ block "content" . }}{{ end }}`.
5. **Funciones Customizadas:**
   - Se pueden registrar funciones personalizadas con **`template.FuncMap`** para ser utilizadas dentro de las plantillas.

Con estos elementos, las plantillas en Gin permiten generar contenido dinámico y reutilizable, separando la lógica del servidor y la presentación del cliente de manera eficiente.

Adicional Funciones Anonimas

Las **funciones anónimas** en Go son funciones que no tienen un nombre explícito. Se utilizan principalmente cuando necesitas una función de forma inmediata y no es necesario

referirse a ella por un nombre. Son útiles en varias situaciones, como pasar funciones como parámetros, retornarlas desde otras funciones o definir comportamientos inline sin complicar el código.

Características principales:

Definición: Una función anónima se declara de forma similar a una función tradicional, pero sin un nombre. Se puede ejecutar inmediatamente o ser asignada a una variable.

func() { fmt.Println("Hola desde una función anónima") }

Uso como valor: Las funciones anónimas pueden ser asignadas a variables o pasadas como parámetros a otras funciones.

saludar := func(nombre string) { fmt.Println("Hola", nombre) } saludar("Juan")

Cierres (Closures): Una función anónima puede capturar y recordar el estado de las variables que la rodean en su entorno. Esto se conoce como un **closure** y permite que la función acceda a estas variables incluso después de que el entorno original haya finalizado.

Ventajas:

- **Simplicidad y Concisión:** Permiten escribir funciones "ad-hoc" sin la necesidad de declarar una función con nombre.
- **Flexibilidad:** Son muy útiles en contextos como callbacks, procesamiento de datos o funciones de orden superior.
- **Captura de contexto:** A través de los **closures**, permiten mantener acceso al contexto donde se crearon, lo cual es útil para mantener estados entre invocaciones

En resumen, las funciones anónimas en Go son una herramienta poderosa para escribir código limpio y eficiente, permitiendo flexibilidad en la programación, especialmente cuando se trabajan con funciones como parámetros o cuando se necesita capturar y mantener un estado a lo largo del tiempo.

Adicional WSL

WSL (Windows Subsystem for Linux) es una característica de Windows que permite ejecutar un entorno de Linux directamente sobre Windows sin necesidad de máquinas virtuales o configuraciones dual-boot. WSL ofrece un kernel de Linux completo y acceso a herramientas y aplicaciones de Linux, lo que permite a los desarrolladores usar herramientas y scripts de Linux sin abandonar el entorno de Windows.

Hay dos versiones principales de WSL:

1. **WSL 1**: Utiliza una capa de compatibilidad para ejecutar binarios de Linux en el sistema Windows, pero no tiene un kernel de Linux real.
2. **WSL 2**: Introduce un kernel de Linux completo, mejorando la compatibilidad y el rendimiento, permitiendo ejecutar aplicaciones más complejas de Linux en Windows.

WSL es útil para desarrolladores que necesitan trabajar con herramientas de Linux, como aquellos que desarrollan en Python, Node.js, o herramientas de administración de sistemas, todo mientras siguen usando Windows.