# Informe de Diferencias entre Go y Rust

**1. Declaración de Estructuras**

* **Go**:

go

Copiar código

type Correo struct {

Email string

}

type Persona struct {

Nombre string

Correos []Correo

}

* **Rust**:

rust

Copiar código

struct Correo {

email: String,

}

struct Persona {

nombre: String,

correos: Vec<Correo>,

}

**Diferencias**:

* En Rust, las estructuras se declaran con la palabra clave struct y los nombres de los campos van en minúscula por convención.
* Los tipos de campo en Rust usan el tipo String para cadenas de texto y Vec<T> para vectores (listas dinámicas), mientras que Go usa string y slices ([]Correo).

**2. Creación e Inicialización de Estructuras**

* **Go**:

go

Copiar código

personas := []Persona{

{

Nombre: "Juan Perez",

Correos: []Correo{

{Email: "juan.perez@example.com"},

{Email: "juan.p@example.com"},

{Email: "jperez@example.com"},

},

},

// más personas

}

* **Rust**:

rust

Copiar código

let mut personas = vec![

Persona {

nombre: String::from("Juan Perez"),

correos: vec![

Correo { email: String::from("juan.perez@example.com") },

Correo { email: String::from("juan.p@example.com") },

Correo { email: String::from("jperez@example.com") },

],

},

// más personas

];

**Diferencias**:

* En Rust, se usa el macro vec![] para crear vectores, y String::from para inicializar cadenas de texto.
* Las instancias en Rust deben ser mutables (let mut) si se van a modificar más adelante.

**3. Impresión de Datos**

* **Go**:

go

Copiar código

fmt.Println("Lista de personas:")

imprimirPersonas(personas)

* **Rust**:

rust

Copiar código

println!("Lista de personas:");

imprimir\_personas(&personas);

**Diferencias**:

* Rust usa el macro println! para imprimir en consola.
* Las funciones en Rust se llaman de manera similar, pero las referencias (&) son explícitas para pasar datos sin mover la propiedad.

**4. Definición y Llamada a Funciones**

* **Go**:

go

Copiar código

func imprimirPersonas(personas []Persona) {

for \_, persona := range personas {

fmt.Printf("Nombre: %s\n", persona.Nombre)

for \_, correo := range persona.Correos {

fmt.Printf("Correo: %s\n", correo.Email)

}

fmt.Println()

}

}

* **Rust**:

rust

Copiar código

fn imprimir\_personas(personas: &Vec<Persona>) {

for persona in personas {

println!("Nombre: {}", persona.nombre);

for correo in &persona.correos {

println!("Correo: {}", correo.email);

}

println!();

}

}

**Diferencias**:

* Rust usa la palabra clave fn para definir funciones.
* Las referencias en Rust (&) son necesarias para indicar que no se está transfiriendo la propiedad de los datos.

**5. Manipulación de Datos y Referencias**

* **Go**:

go

Copiar código

func buscarPersona(personas []Persona, nombre string) \*Persona {

for i := range personas {

if personas[i].Nombre == nombre {

return &personas[i]

}

}

return nil

}

func cambiarCorreo(persona \*Persona, indice int, nuevoCorreo string) {

if indice >= 0 && indice < len(persona.Correos) {

persona.Correos[indice].Email = nuevoCorreo

}

}

* **Rust**:

rust

Copiar código

fn buscar\_persona<'a>(personas: &'a mut Vec<Persona>, nombre: &str) -> Option<&'a mut Persona> {

for persona in personas.iter\_mut() {

if persona.nombre == nombre {

return Some(persona);

}

}

None

}

fn cambiar\_correo(persona: &mut Persona, indice: usize, nuevo\_correo: &str) {

if indice < persona.correos.len() {

persona.correos[indice].email = String::from(nuevo\_correo);

}

}

**Diferencias**:

* Rust usa Option en lugar de nil para indicar la posibilidad de ausencia de un valor.
* Las referencias mutables (&mut) y lifetimes ('a) se utilizan en Rust para asegurar la seguridad de memoria y evitar referencias colgantes.
* La manipulación de vectores en Rust se hace con métodos como iter\_mut() para obtener iteradores mutables.

**Conclusión**

Rust y Go tienen diferencias significativas en la gestión de memoria, sintaxis y enfoque de programación. Rust se enfoca en la seguridad de memoria y las referencias explícitas, mientras que Go simplifica muchas de estas preocupaciones con un recolector de basura. La migración de código de Go a Rust implica adaptarse a estas diferencias y aprovechar las fortalezas de cada lenguaje.