Enrique Ulises Báez Gómez Tagle
0241823@up.edu.mx¿

Concurrency Patterns in Go

1. Patrones de concurrencia

- **Worker Pool Pattern** -> Consiste en crear un grupo de goroutines («workers») que procesan tareas concurrentemente, limitando la cantidad de operaciones simultáneas.
  - Utilidades:
    - Procesamiento masivo de tareas, como manejar múltiples solicitudes HTTP de un servidor web.
    - Procesamiento concurrente de imágenes o cualquier tipo de dato en grandes volúmenes.
  - Código clave:

```go
// Definición de worker que procesa tareas
func worker(id int, jobs <-chan int, results chan<- int) {
    for job := range jobs {
        // Procesa cada trabajo aquí
        results <- job * 2
    }
}
```

- **Pipeline Pattern** -> Un patrón donde una serie de etapas de procesamiento son ejecutadas concurrentemente. Cada etapa recibe datos, los transforma y pasa los resultados a la siguiente etapa.
  - Aplicaciones comunes:
    - Pipelines de procesamiento de datos en sistemas ETL (Extract, Transform, Load).
    - Procesamiento de imágenes en aplicaciones multimedia.
  - Beneficios:
    - Mejora la modularidad y facilita la implementación de flujos de procesamiento de datos complejos.
  - Ejemplo:

```go
// Etapas del pipeline que procesan y transforman datos secuencialmente
go func() {
    for data := range input {
        stage1Output <- data * 2
    }
}()
```

- **Fan-out/Fan-in Pattern** -> Este patrón distribuye (fan-out) tareas a múltiples goroutines para procesarlas en paralelo y luego recolecta (fan-in) los resultados agregados en una sola goroutine.
  - Aplicaciones típicas:
    - Scraping web concurrente, donde múltiples sitios web son escaneados simultáneamente.
    - Agregación de datos en sistemas de IoT, donde múltiples sensores generan datos concurrentemente.
  - Ejemplo:

```go
// Recolecta y distribuye tareas concurrentes
for i := 0; i < numWorkers; i++ {
    go worker(i, tasks, results)
}
```

- **Mutex Pattern** -> Protege recursos compartidos utilizando un mutex (`sync.Mutex`), asegurando acceso exclusivo para evitar condiciones de carrera.
  - Escenarios:
    - Cuando múltiples goroutines acceden y modifican una misma variable o recurso al mismo tiempo.
  - Ejemplo:

```
var mu sync.Mutex
mu.Lock()
// Código seguro aquí
mu.Unlock()
```

- **Semaphore Pattern** -> Controla el número de goroutines concurrentes que pueden acceder a un recurso limitado utilizando semáforos.
  - Uso común:
    - Limitar el número de conexiones a una base de datos para evitar sobrecarga del sistema.
  - Implementación:

```
var sem = make(chan struct{}, 5) // Permite 5 goroutines
```

- **Barrier Pattern** -> Sincroniza un conjunto de goroutines en puntos específicos de su ejecución, asegurando que todas alcanzan cierto estado antes de proceder.
  - Uso:
    - Cuando se requiere que todas las goroutines completen una parte de su trabajo antes de que alguna avance a la siguiente etapa.
- **WaitGroup Pattern** -> Permite esperar a que un conjunto de goroutines termine antes de proceder, útil en casos donde se necesita que todas las goroutines completen su trabajo antes de recoger los resultados.
  - Ejemplo:

```
var wg sync.WaitGroup
wg.Add(1)
go func() {
    defer wg.Done()
    // Código concurrente
}()
wg.Wait()
```

2. Paralelismo vs. Concurrencia

- **Paralelismo** -> La ejecución simultánea de múltiples tareas usando varios núcleos de CPU. Es ideal para tareas computacionalmente intensivas, como simulaciones científicas o procesamiento de grandes volúmenes de datos.
  - Escenario típico: Renders gráficos o simulaciones matemáticas complejas.
- **Concurrencia** -> Se refiere a la capacidad de un sistema para manejar múltiples tareas que se solapan en el tiempo, aunque no necesariamente corran en paralelo.
  - Mejora la eficiencia de recursos y la capacidad de respuesta del sistema.
  - Escenario típico: Aplicaciones web que manejan múltiples solicitudes de usuarios simultáneamente.

3. Patrones Clave para Concurrencia Efectiva

- **Task Decomposition** -> Descomposición de tareas complejas en subtareas más simples que pueden ejecutarse concurrentemente.

- Beneficios:
  - Mejor modularidad y uso eficiente de los núcleos de CPU.
- Ejemplo:

```
go processItem(itemID)
```

- **Worker Pools** -> Un patrón que asigna un número fijo de goroutines para manejar tareas, evitando el agotamiento de recursos.
  - Uso típico:
    - Procesamiento masivo de datos en sistemas distribuidos.
  - Ejemplo:

```
go func() {
    worker(task, results)
}()
```

- **Context y Cancelación** -> Utilización del contexto (`context.Context`) para gestionar la cancelación y el tiempo de vida de las goroutines, evitando la ejecución innecesaria de tareas concurrentes cuando ya no se necesitan.
  - Ejemplo:

```
ctx, cancel := context.WithCancel(context.Background())
go worker(ctx)
cancel() // Cancela la goroutine cuando no se necesita más
```

4. Desafíos y Optimización en Código Concurrente

- **Errores comunes** -> Los errores como **deadlocks** (cuando dos o más goroutines esperan indefinidamente) y **race conditions** (cuando varias goroutines acceden a un recurso simultáneamente) son típicos en la programación concurrente.
  - Solución:
    - Uso de patrones como mutex y semáforos para sincronizar el acceso a recursos compartidos.
- **Optimización del rendimiento** -> Es crucial minimizar el uso de goroutines cuando no son necesarias, así como emplear patrones de concurrencia adecuados para balancear la carga del sistema y evitar la sobrecarga.