Sistemas Operativos 2

Grupo

14

Manual Técnico

Práctica 2

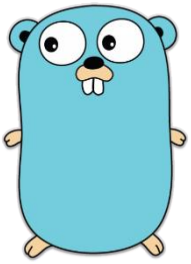
Benaventi Bernal Fuentes Roldan 201021212

Juan Pablo Rojas Chinchilla 201900289

Luis Daniel Ernesto Castellanos Galindo 201902238

Lourdes Rosario Velásquez Melini 201906564

Go



Es un lenguaje de programación concurrente y compilado inspirado en la sintaxis de C, que intenta ser dinámico como Python y con el rendimiento de C o C++. Ha sido desarrollado por Google y sus diseñadores iniciales fueron Robert Griesemer, Rob Pike y Ken Thompson.

Programación concurrente

Es la ejecución simultánea de múltiples tareas interactivamente. Estas tareas pueden ser un conjunto de procesos o hilos de ejecución creados por un único programa. Las tareas se pueden ejecutar en una sola CPU (multiprogramación), en varios procesadores, o en una red de computadores distribuidos.

Hilos

Se puede definir como una unidad básica de ejecución del Sistema Operativo para la utilización del CPU. Este es quien va al procesador y realiza todos los cálculos para que el programa se pueda ejecutar. Es necesario, ya que debe contar con al menos un hilo para que cualquier programa sea ejecutado.

Cada hilo tiene:

- ID del hilo
- Contador de programa
- Registros
- Stack

Multithreading

Es la capacidad para poder proporcionar múltiples hilos de ejecución al mismo tiempo. Una aplicación por lo general es implementada como un proceso separado con muchos hilos de control. Dentro de las semejanzas de hilo y multihilo es que poseen un solo bloque de control de proceso (PCB) y un solo espacio de direcciones de proceso.

Deadlock

También conocido como abrazo mortal, ocurre cuando un proceso espera un evento que nunca va a pasar. Aunque se puede dar por comunicación entre procesos, es más frecuente que se dé por manejo de recursos.

En este caso, deben cumplirse 4 condiciones para que se dé un "deadlock":

- Los procesos deben reclamar un acceso exclusivo a los recursos.
- Los procesos deben retener los recursos mientras esperan otros.
- Los recursos pueden no ser removidos de los procesos que esperan.
- Existe una cadena circular de procesos donde cada proceso retiene uno o más recursos que el siguiente proceso de la cadena necesita.

Gocolly

Es un marco de trabajo de rastreo web implementado con go. Actualmente tiene más de 3400 estrellas en github y ocupa el primer lugar en la versión go de los programas de rastreo. Gocolly es rápido y elegante. Puede iniciar más de 1K solicitudes por segundo en un solo núcleo; proporciona un conjunto de interfaces en forma de funciones de devolución de llamada, que pueden implementar cualquier tipo de rastreador; confiando en la biblioteca de goquery puede seleccionar elementos web como jquery.

Para su uso debemos instalarlo con el siguiente comando:

- go get -u github.com/gocolly/colly/...

Luego en el archivo del código debemos importar el paquete

- import "github.com/gocolly/colly"

El cuerpo principal de colly es el objeto Collector, que gestiona la comunicación de red y es responsable de ejecutar funciones de devolución de llamada adicionales mientras se ejecuta el trabajo. Para usar colly, primero debe inicializar Collector:

- c := colly.NewCollector()
-

```
47
48          c := colly.NewCollector()
```

Se usaron diferentes funciones para poder controlar las operaciones de recolección u obtener información, las cuales fueron:

- OnRequest: esta se llama antes que se inicie la solicitud

```
c.OnRequest(func(r *colly.Request) {
    fmt.Println("Visiting", r.URL)
})
```

- OnHTML: se llama si el contenido recibido es HTML

```
c.OnHTML("div#mw-content-text p", func(e *colly.HTMLElement) {
    conteo_palabras += len(strings.Split(e.Text, " "))
})

c.OnHTML("div#mw-content-text p a", func(e *colly.HTMLElement) {
    if conteo < Nr {
        aux = e.Request.AbsoluteURL(e.Attr("href"))
        results <- Task{aux, Nr - 1}
        conteo = conteo + 1
    }
})
```

Escribir Archivo

Tenemos la función para escribir el archivo el cual nombramos

```
func escribirArchivo(contenido Result) {
    // file, _ := os.OpenFile(nombre_archivo+".json", os.O_RDWR, 0644)
    resultFile.Results = append(resultFile.Results, contenido)
    file, _ := json.MarshalIndent(resultFile, "", " ")

    // response, _ := json.Marshal(resultFile)

    err1 := ioutil.WriteFile(nombre_archivo+".json", file, 0644)

    // _, err1 := file.WriteString(string(response))
    if err1 != nil {
        fmt.Println(err1)
    }
    // defer file.Close()
}
```

Main

Es la función que llama la ejecución del programa, inicializa todas las variables claves del programa.

```
func main() {
    init_values()
    jobs := make(chan Task, tamano_cola)
    results := make(chan Task, 1000000)
    for i := 0; i < cantidad_monos; i++ {
        go worker(jobs, results, i)
    }

    jobs <- Task{url_inicial, n_r}

    for r := range results {
        // before := <-results
        // escribirArchivo("origen: " + newSha(before.Url))
        // fmt.Println("Visitando resultados")
        // fmt.Println("id ", r)
        // fmt.Println(<-results)
        // fmt.Println(r)
        jobs <- r
        // jobs <- r
        // queue = append([]string{r.Url}, queue...)
        // fmt.Println(queue)
    }
}
```