In this code:

* **numWorkers** defines the number of worker goroutines.
* **numJobs** defines the number of jobs to be processed.
* We create two channels: **jobs** for distributing jobs to workers and **results** for collecting results from workers.
* The **worker** function simulates processing a job and sends the result back to the main goroutine through the **results** channel.
* In the main function, we start the worker goroutines and distribute jobs among them (fan-out).
* We use a **sync.WaitGroup** to ensure that all workers have finished before closing the **results** channel (fan-in).
* Finally, we print the results received from the workers.

This code demonstrates a simple fan-in and fan-out concurrency pattern in Go using goroutines and channels.

package main

import (

"fmt"

"math/rand"

"sync"

"time"

)

func main() {

// Define the number of workers and the size of the job pool

numWorkers := 3

numJobs := 10

// Create channels for fan-out and fan-in

jobs := make(chan int, numJobs)

results := make(chan int, numJobs)

// Start the workers

var wg sync.WaitGroup

for i := 0; i < numWorkers; i++ {

wg.Add(1)

go worker(i, jobs, results, &wg)

}

// Fan-out: Generate jobs and distribute them among workers

go func() {

for i := 0; i < numJobs; i++ {

jobs <- i

}

close(jobs)

}()

// Fan-in: Collect results from workers

go func() {

wg.Wait()

close(results)

}()

// Print results

for res := range results {

fmt.Println("Result:", res)

}

}

func worker(id int, jobs <-chan int, results chan<- int, wg \*sync.WaitGroup) {

defer wg.Done()

for job := range jobs {

// Simulate processing time

time.Sleep(time.Duration(rand.Intn(100)) \* time.Millisecond)

fmt.Printf("Worker %d processed job %d\n", id, job)

results <- job \* 2 // Example processing: Double the job value

}

}