package main import ( "fmt" "time" ) // worker это наш рабочий, который принимает два канала: // jobs - канал задач, это входные данные для обработки // results - канал результатов, это результаты работы воркера func worker(doneCh chan struct{}, id int, jobs <-chan int, results chan<- int) { for { select { case <-doneCh: fmt.Println(id, "Worker DONE CHANNEL STOP") return case job, ok := <-jobs: if !ok { jobs = nil continue } // для наглядности будем выводить какой рабочий начал работу и его задачу fmt.Println("рабочий", id, "запущен задача", job) // немного замедлим выполнение рабочего time.Sleep(time.Second) // для наглядности выводим какой рабочий завершил какую задачу fmt.Println("рабочий", id, "закончил задача", job) // отправляем результат в канал результатов results <- job + 1 } } } func main() { // допустим у вас 10 задач, которые нужно выполнить const numJobs = 10 // создаем буферизованный канал для принятия задач в воркер jobs := make(chan int, numJobs) // создаем буферизованный канал для отправки результатов results := make(chan int, numJobs) // создаём сигнальный канал doneCh := make(chan struct{}) //defer close(doneCh) // создаем и запускаем 3 воркера, это и есть пул, // передаем id, это для наглядности, канал задач и канал результатов for w := 1; w <= 3; w++ { go worker(doneCh, w, jobs, results) } // в канал задач отправляем какие-то данные // задач у нас 5, а воркера 3, значит одновременно решается только 3 задачи for j := 1; j <= numJobs; j++ { jobs <- j } // как вы помните, закрываем канал на стороне отправителя close(jobs) // забираем из канала результатов результаты ;) for a := 1; a <= numJobs; a++ { if a == 9 { close(doneCh) return } <-results } }

Первая ошибка в примере - слишком рано закрыт канал jobs. Он буферизованный, но при этом в воркере стоит проверка его закрытия, которая фактически шунтирует этот канал. Поэтому воркер начинает играть вхолостую сразу, как только закрывается jobs. А в канале при этом все еще есть данные. Чтобы убедиться, достаточно поставить отладку перед continue. jobs = nil там вообще не нужен. Вариант решения - сделать канал небуферизованным. Тогда его закрытие не приведет к потеряшкам на коде

if !ok {

continue

}

if a == 9 - это просто костыль какой-то

Вот работающий пример:

package main

import (

"fmt"

"time"

)

// worker это наш рабочий, который принимает два канала:

// jobs - канал задач, это входные данные для обработки

// results - канал результатов, это результаты работы воркера

func worker(doneCh chan struct{}, id int, jobs <-chan int, results chan<- int) {

for {

select {

case <-doneCh:

fmt.Println(id, "Worker DONE CHANNEL STOP")

return

case job, ok := <-jobs:

if !ok {

continue

}

// для наглядности будем выводить какой рабочий начал работу и его задачу

fmt.Println("рабочий", id, "запущен задача", job)

// немного замедлим выполнение рабочего

time.Sleep(time.Second)

// для наглядности выводим какой рабочий завершил какую задачу

fmt.Println("рабочий", id, "закончил задача", job)

// отправляем результат в канал результатов

results <- job + 1

}

}

}

func main() {

// допустим у вас 10 задач, которые нужно выполнить

const numJobs = 10

// создаем буферизованный канал для принятия задач в воркер

jobs := make(chan int, numJobs)

// создаем буферизованный канал для отправки результатов

results := make(chan int, numJobs)

// создаём сигнальный канал

doneCh := make(chan struct{})

//defer close(doneCh)

// создаем и запускаем 3 воркера, это и есть пул,

// передаем id, это для наглядности, канал задач и канал результатов

for w := 1; w <= 3; w++ {

go worker(doneCh, w, jobs, results)

}

// в канал задач отправляем какие-то данные

// задач у нас 5, а воркера 3, значит одновременно решается только 3 задачи

for j := 1; j <= numJobs; j++ {

jobs <- j

}

// забираем из канала результатов результаты ;)

for a := 1; a <= numJobs; a++ {

<-results

}

// как вы помните, закрываем канал на стороне отправителя

close(jobs)

close(doneCh)

fmt.Println("sleeping")

time.Sleep(time.Second)

}

Ну и вместо time.Sleep(time.Second) в нормальном коде ставится wg.Wait()

Еще, кстати, она проблема - это все тот же злополучный

if !ok {

continue

}

Если в нем не делать хотя бы минимальный time.Sleep, то цикл будет многократно проворачиваться прежде, чем словит <-doneCh