G

Go_Day09-1

Forked from an inaccessible project

Update from 5
Administrator authored 5 months ago

Name	Last commit	Last update
code-samples	Init commit	1 year ago
data-samples	Init commit	1 year ago
datasets	Init commit	1 year ago
materials	Update from 5	5 months ago
misc	Init commit	1 year ago
src	Init commit	1 year ago
.gitignore	Update from 3	1 year ago
CHANGELOG	Init commit	1 year ago
LICENSE	Init commit	1 year ago
README.md	Update from 4	7 months ago

README.md

Day 09 - Go Boot camp

Daily Routine

Contents

Chapter I

General rules

- Your programs should not quit unexpectedly (giving an error on a valid input). If this happens, your project will be considered non functional and will receive a 0 during the evaluation.
- We encourage you to create test programs for your project even though this work won't have to be submitted and won't be graded. It will give you a chance to easily test your work and your peers' work. You will find those tests especially useful during your defence. Indeed, during defence, you are free to use your tests and/or the tests of the peer you are evaluating.
- Submit your work to your assigned git repository. Only the work in the git repository will be graded.
- If your code is using external dependencies, it should use Go Modules for managing them

Chapter II

Rules of the day

- You should only turn ink `\*.go`, `\*_test.go` files and (in case of external dependencies) `go.mod + go.sum`
- Your code for this task should be buildable with just `go build`
- All your tests should be runnable by calling standard `go test ./...`

Chapter III

Intro

Sometimes we hear that there are some people who "can do several things in parallel". Even though that may be theoretically possible to do completely different things (say, with different hands), this phrase usually refers to people handling several tasks at the same time, but NOT in parallel. What do I mean by that?

"Parallel" in computer science usually means that progress is made on more than one task at the same time. But with people it is a bit different - the real trick is to keep the context and switch over from one task to another quickly. From the side it may even look like "parallelism", but it's not - it is *concurrency*, which is a bit more wide concept. And yes, it means concurrency can be inplemented using parallelism, but it can also work without it (like most people do).

When programming things to run in parallel, we are generally thinking of explicitly creating several separate threads and giving each of them a target function. But it is not how it works in case of Golang - it operates in terms of *concurrency*, which means we don't really need to think if and how actual parallelization is happening under the hood.

That gives us a lot of power, but with great power comes great responsibility...

Chapter IV

Exercise 00: Sleepsort

So, let's write a toy algorithm as an example. It is pretty much useless for production, but it helps to grasp the concept.

You need to write a function called `sleepSort` that accepts an unsorted slice of integers and returns an integer channel, where these numbers will be written one by one in sorted order. To test it, in main goroutine you should read and print output values from a returned channel and gracefully stop the application in the end.

The idea of Sleepsort (what makes it a "toy") is that we're spawning a number of goroutines equal to the size of an input slice. Then, each goroutine sleeps for amount of seconds equal to the received number. After that it wakes up and this number to the output channel. It's easy to notice that this way numbers will be returned in a sorted order.

Chapter V

Exercise 01: Spider-Sense

You probably remember how Peter Parker realised he now has superpowers when he woke up in the morning. Well, let's write our own spider (or crawler) for parsing the web. You need to implement a function `crawlWeb` which will accept an input channel (for sending in URLs) and return another channel for crawling results (pointers to web page body as a string). Also, at any moment in time there shouldn't be more than 8 goroutines querying pages in parallel.

But we want to be quick and flexible, so another requirement is to be able to stop the crawling process at any time by pressing Ctrl+C (and your code should perform a graceful shutdown). For that you may add more input arguments to `crawlWeb` function, which should be either `context.Context` for cancellation or `done` channel. If not interrupted, the program should gracefully stop after all given URLs are processed.

Chapter VI

Exercise 02: Dr. Octopus

Okay, so now we have to slain the villain! The main problem with Dr.Octopus is that he has a lot of tech tentacles, and it's hard to keep track of them all. Let's tie them together!

For this exercise, you need to write a function called `multiplex` which should be *variadic* (accepts a variable number of arguments). It should accept channels (`chan interface{}`) for arguments and return a single channel of the same type. The idea is to redirect any incoming messages from these input channels into just one single output channel, effectively implementing "fan-in" pattern.

As a proof of work, you should write a test on sample data which will explicitly show that all values randomly sent to any input channels are received further on the same output channel.

And...you've just defeated a villain!