GO on the toilet

episode #11

Concurrency is not parallelism left side

① I hope you've started reading right here. This is the beginning of this episode. Let's get started with acknowledgment that both parallelism and concurrency deal with the same domain of doing many things "at the same time". →②

③ This is concurrent work in action. You are reading two things at the same time, but you can be focused only on one of them at the time. This shows that a computer with a single CPU is able to perform concurrent tasks. Concurrency is really about how your program is structured. If a human being would be able to read one page with one eye, and the second with the other we would have parallel computing. Similarly, it is easy to add parallelism to the program that is written concurrently by using more processors. →④

⑤

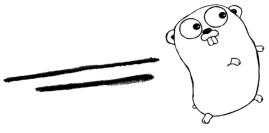
```
func main() {
    files, _ := ioutil.ReadDir(".")

    thumbnailsToUpload := make(chan string)
    uploadDone := make(chan bool)
    go upload(thumbnailsToUpload, uploadDone)

    wg := sync.WaitGroup{}
    for _, file := range files {
       if strings.HasSuffix(file.Name(), ".png") {
             wg.Add(1)
             go thumbnail(file.Name(), thumbnailsToUpload, &wg)
       }
    }
    // Wait for all thumbnail jobs to finish.
    wg.Wait()
    // Close toUpload channel so that upload goroutine will
    // know when to stop iterating over toUpload channel.
    close(toUpload)
    // Wait for uploadDone channel so receive so that program
    // will now finish before launching all uploads.
    <-uploadDone
}
```

Let's see how **`thumbnail`** and **`upload`** are defined. →⑥

⑦ This short program leverages go asynchronous model. It uses goroutines in order to make it faster. →⑧



Concurrency is not parallelism right side

② Did you see what just happened? You've just switched a context. You've just started working on different task of reading an article to the right site. ③←

④ Let's think of an example of concurrent work to do that is small enough that will fit on one episode, yet realistic enough that you can relate to it. How about a thumbnailer? The idea will be to have a program that makes a thumbnail for every png file in given directory and uploads them somewhere. ⑤←

⑥

```
func thumbnail(path string, thumbnails chan<- string, wg *sync.WaitGroup) {
    defer wg.Done()

    // .. magic that generates smaller version of an image under `path`.
    thumbnailPath := resizeImg(path)

    thumbnails <- thumbnailPath
}
```

Resizing an image is a CPU-bound task.

```
func upload(paths <-chan string, done chan<- bool) {
    for path := range paths {
       go uploadSingle(path)
    }
    done <- true
}
```

This one is IO-bound, it has make a remote call. ⑦←

⑧ What is worth noting here is that this program can be run on a single core (by setting environment variable **`GOMAXPROCS=1`**) and we can still save time on executing upload goroutines because inside this goroutine we don't wait for the result of upload operation to finish (which is naive, but that's not the point).

Starting from Go 1.5 Release **`GOMAXPROCS`** environment variable defaults to the amount of CPU cores on the machine. If we have multi-core machine (which we probably have) and if we do not change the value of **`GOMAXPROCS`**, thumbnail function will also gain performance because the CPU will be able to do calculations exactly at the same time for many goroutines.