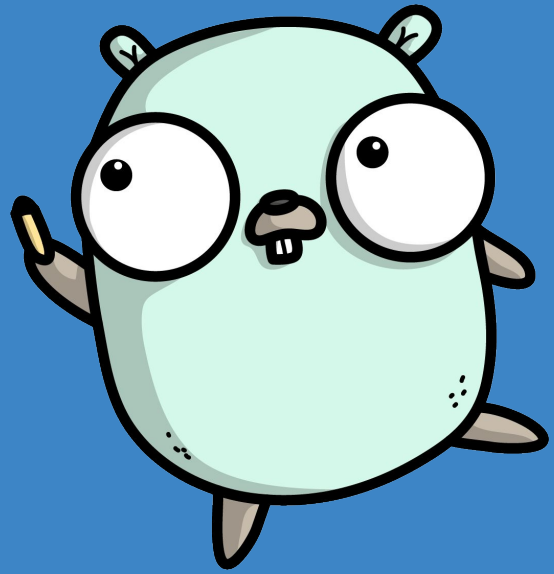
WHY GO ?

Anubha Kushwaha

@AnubhaKushwaha1

- **Concurrency**
  - built-in support for concurrency
- **Ease of deployment**
  - have a single binary to deploy
- **Performance**
  - more efficient and cheaper to create goroutines



GoRoutines

- Independently executing function
- Considered as very cheap thread
- Has its own call stack, which grows and shrinks

SHOWTIME !!

```
func awesome( statement string ) {

        for i := 0;i<10; i++ {

                fmt.Println( statement, i )
        }
}

func main( ) {

        go awesome( " We are awesome ")

        fmt.Println( " I don't have to wait for func awesome ")

        fmt.Println( " You are awesome enough, bye! ")

}
```

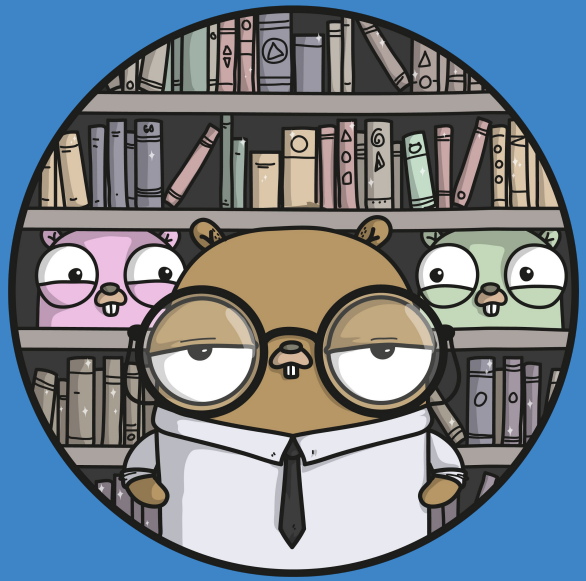
```
func awesome( statement string) {

        for i := 0;i<10; i++ {

                fmt.Println( statement, i )
        }
}
func main( ) {

        go awesome( " We are awesome ")

        fmt.Println( " I don't have to wait for func awesome ")

        time.Sleep( 2*time.Second )

        fmt.Println( " You are awesome enough, bye! ")

}
```

We already had Threads

- **Goroutines, memory and OS threads**
  - Go has a segmented stack that grows as needed. Go runtime does the scheduling, not the OS. The runtime multiplexes the goroutines onto a relatively small number of real OS threads.
- **Goroutines switch cost**
  - Goroutines are ***scheduled cooperatively*** and when a switch occurs, only 3 registers need to be saved/restored - Program Counter, Stack Pointer and DX.

COMMUNICATION

- prevent simultaneous access of shared resources by more than one goroutine.
- best to transfer data between goroutines using channels
- do not communicate by sharing memory instead share memory by communicating.



CHANNELS

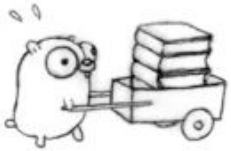
- provides a connection between two goroutines
- helps in synchronization
- type of element you can send through a channel, capacity (or buffer size) and direction of communication

```
func awesome( statement string, c chan string) {

        for i := 0;i<10; i++ {

                c <- fmt.Sprintf("%s %d", statement, i)
        }
}
func main( ) {

        c := make(chan string)

        go awesome( " We are awesome ", c)

        fmt.Println( " I don't have to wait for func awesome ")

        for i := 0;i<10; i++ {
                fmt.Println(<-c)
        }

        fmt.Println( " You are awesome enough, bye! ")
}
```

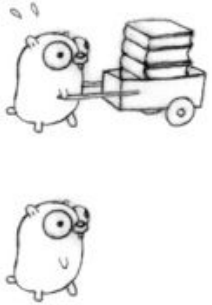
EXECUTING INDEPENDENTLY BUT THEY ARE COMMUNICATING

“Concurrency is about dealing with lots of things at once. Parallelism is about doing lots of things at once.”—Rob Pike

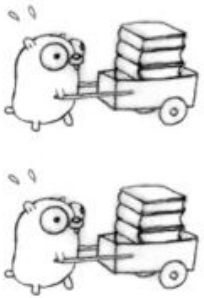
Move a pile of obsolete language manuals to the incinerator.



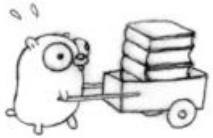
With only one gopher this will take too long.

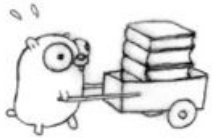


More gophers are not enough; they need more carts.

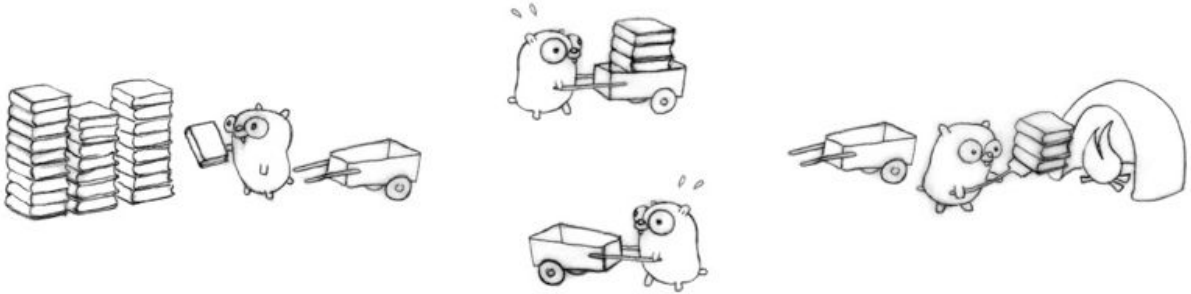


This will go faster, but there will be bottlenecks at the pile and incinerator.
Also need to synchronize the gophers.
A message (that is, a communication between the gophers) will do.


This will consume input twice as fast.

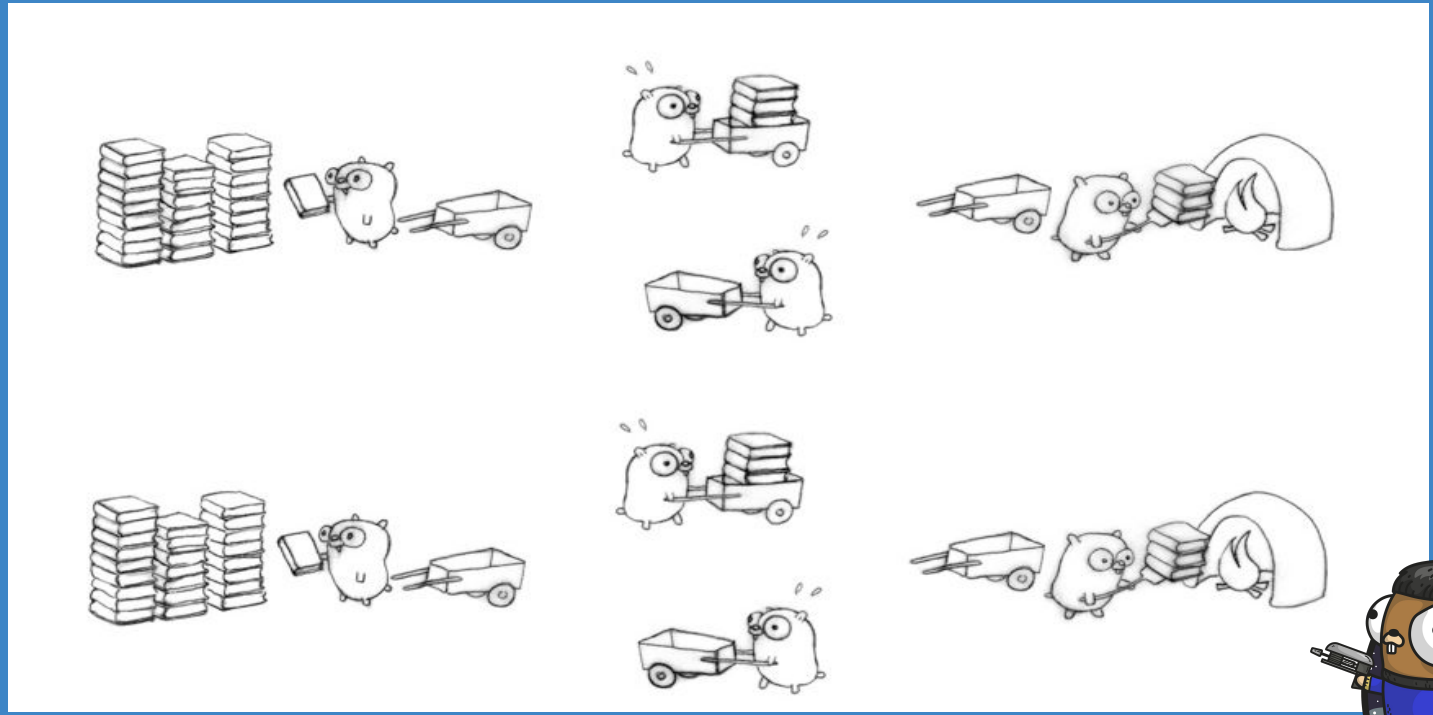


Four gophers in action for better flow, each doing one simple task.

If we arrange everything right (implausible but not impossible), that's four times faster than our original one-gopher design.

Improved performance by adding concurrent procedure to existing design

We can now parallelize on another axis and concurrency design makes it easy, all gophers busy



Good Reads

- https://blog.cloudflare.com/how-stacks-are-handled-in-go/
- https://blog.nindalf.com/posts/how-goroutines-work/
- https://groups.google.com/forum/#!topic/golang-nuts/j51G7ieoKh4
- https://thenewstack.io/go-programming-language-helps-docker-container-ecosystem/
- https://blog.golang.org/concurrency-is-not-parallelism