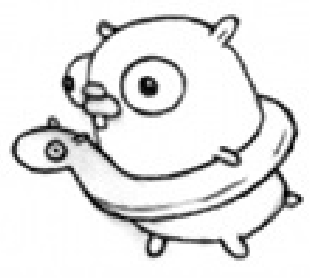
Faisons travailler des gophers ensemble

Présentation

Bruno Michel

- ☆ Directeur Technique d' af83
- ☆ Développeur de LinuxFr.org
- ☆ Ancien président de Ruby France
- ☆ github.com/nono
- ☆ twitter.com/brmichel



Gophers

« Gaufre (ou gauphre) est un nom vernaculaire ambigu en français, pouvant désigner des rongeurs [...] »

— Wikipédia

Go

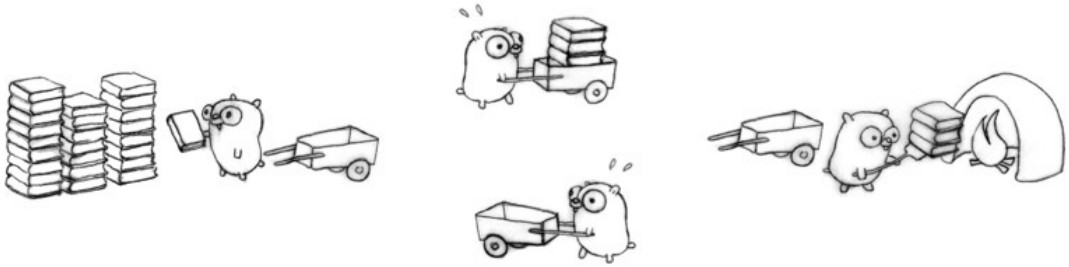
- ☆ Langage Open Source, géré par la communauté
- ☆ Une version stable, Go 1
- ☆ Langage moderne et agréable à utiliser
- ☆ Pour faire des logiciels simples, performants et fiables

Concurrence et parallélisme

Concurrence

« Programming as the composition of independently executing processes. »

— Rob Pike



Parallélisme

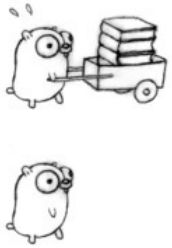
« Programming as the simultaneous execution of (possibly related) computations. »

— Rob Pike

Concurrence vs. parallélisme

Concurrence : une question de structure de données

Parallélisme : à propos de l'exécution



Les primitives de Go

Goroutine

```
func computation(n int) {
    time.Sleep(n * time.Milliseconds)
    fmt.Printf("%s seconds elapsed\n", n)
}
go computation(3)
go computation(1)
go computation(5)
computation(7)
```

Channels

```
func pingpong(ch chan int) {
    n := <-ch
    fmt.Printf("Received %d\n", n)
    ch <- n
}

func main() {
    ch := make(chan int)
    go pingpong(ch)
    ch <- 42
    <-ch
}
```

Buffered Channels

```
func pingpong(ch chan int) {
    n := <-ch
    fmt.Printf("Received %d\n", n)
    ch <- n
}

func main() {
    ch := make(chan int, 5)
    go pingpong(ch)
    ch <- 42
    <-ch
}
```

Select

```
select {
    case chan1 <- nb:
        fmt.Printf("c1")
    case str := <-chan2:
        fmt.Printf("c2")
    case str := <-chan3:
        fmt.Printf("c3")
}
```

Quelques patterns

Générateur

```go
func idGenerator() chan int {
    ids := make(chan int)
    go func() {
        id := 0
        for {
            ch <- id
            id++
        }
    }
    return ids
}

ids := idGenerator()
id1 := <-ids
id2 := <-ids
```

Timeout

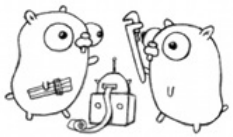
```
select {
    case n := <-ch:
        fmt.Printf("Received %d", n)
    case <-time.After(2 * time.Seconds)
        fmr.Printf("Too late")
}
```

Tickers

```
ticker := time.NewTicker(50 * time.Millisecond)
go func() {
    for t := range ticker.C {
        fmt.Println("Tick at", t)
    }
}()

time.Sleep(150 * time.Millisecond)
ticker.Stop()
```

Worker



```
func worker(in <-chan *Work,
            out chan<- *Work) {
    for w := range in {
        w.z = w.x * w.y
        Sleep(w.z)
        out <- w
    }
}
```

Load balancer

```
func Run() {
    in  := make(chan *Work)
    out := make(chan *Work)
    for i := 0; i < NumWorkers; i++ {
        go worker(in, out)
    }
    go sendLotsOfWork(in)
    receiveLotsOfResults(out)
}
```

Conclusion

Goroutines + Channels = <3

- ☆ Ni mutex
- ☆ Ni semaphore
- ☆ Mais c'est très puissant

Mise en application

External images for LinuxFr.org

https://github.com/nono/img-LinuxFr.org

Pour aller plus loin

Concurrency is not Parallelism (it's better) — Rob Pike

http://waza.heroku.com/