Współbieżność

23 March 2017

Filip Borkiewicz

Dlaczego?

Świat jest paralelny.

Wszystko co dzieje się wokół nas jest zbiorem niezależnie toczących się procesów.

To samo jest prawdą w informatyce:

- Networking, setki lub tysiące połączeń
- Wiele rdzeni
- Wielu użytkowników
- Wiele programów

Języki programowania powinny jakoś to odzwierciedlać.

Współbieżność

Sposób kompozycji programów, ogólny koncept zbioru niezależnie wykonywanych zadań (funkcji?).

Paralelizm

Jednoczesne wykonywanie wielu zadań (funkcji?), związanych ze sobą lub nie.

Współbieżność to nie paralelizm

"My program is slower when using multiple threads."

Współbieżność polega na poradzeniu sobie z wieloma zadaniami jednocześnie, paralelizm polega na wykonaniu wielu zadań jednocześnie.

Współbieżność dotyczy sposobu kompozycji i projektowania oprogramowania.

Współbieżność może (ale nie musi) prowadzić do rozwiązania problemu, które łatwo będzie sparalelizować.

Komunikacja

"Do not communicate by sharing memory; instead, share memory by communicating."

Tony Hoare: Communicating Sequential Processes

"This paper suggests that input and ouput are basic primitives of programming
and that parallel composition of communicating sequential processes is a
fundamental program structuring method."

- Squeak -> Newsqueak
- Limbo
- Erlang
- Go

Go

Compiled, statically typed, garbage-collected, object-oriented (?) programming language with CSP-inspired concurrency features.

Packages, rich official tooling, no generics.

```
a := 5
```

```
var a int = 5
```

Hello, world!

```
package main

import "fmt"

func main() {
    fmt.Println("Hello, □□")
}
```

Goroutines

Zwykłe wywołanie funkcji:

```
foo() // foo blocks
```

Keyword `go`:

```
go foo() // foo doesn't block
bar()
```

Podobne do wątków, ale znacznie lżejsze.

Normalnym jest tworzenie tysięcy lub dziesiątek tysięcy goroutines podczas działania programu.

Runtime dynamically schedules goroutines for execution on OS threads.

Channels

Channels are used to communicate between running goroutines and synchronize them.

Similar to pipes in the shell.

Channels are typed.

```
ch := make(chan int)
go func() {
    time.Sleep(5 * time.Second)
    ch <- 10
}
number := <-ch
```

Select

Podobne do instrukcji `switch`, gałęzie wybierane są w oparciu o gotowość do komunikacji, a nie o wartość jakiejś zmiennej.

Blokuje do momentu kiedy jeden z warunków zostanie spełniony.

Jeśli żaden kanał nie jest gotowy do komunikacji zostaje wybrany przypadek `default`.

```
select {
    case value1 := <-ch1:
        fmt.Println("Received from ch1:", value1)
    case value2 := <-ch2:
        fmt.Println("Received from ch2:", value2)
    case ch3 <- 7:
        fmt.Println("Sent to ch3:", 7)
    default:
        fmt.Println("Nobody was ready!")
}
```

Sequential

Praca wykonywana w pętli blokuje wykonywanie programu.

```
func main() {
    for i := 0; i < 5; i++ {
        fmt.Println("Hello", i)
        time.Sleep(time.Duration(rand.Float32() * float32(time.Second)))
    }

    fmt.Println("Exiting.")
}
```

Goroutine

Wykonujemy pracę w goroutine, możemy kontynuować od razu.

```
func main() {
    go func() {
        for i := 0; ; i++ {
            fmt.Println("Hello", i)
            time.Sleep(time.Duration(rand.Float32() * float32(time.Second)))
        }
    }()

    fmt.Println("Exiting.")
}
```

Goroutine

Kiedy program zakańcza działanie, zabijane są wszystkie goroutines.

```
func main() {
    go func() {
        for i := 0; ; i++ {
            fmt.Println("Hello", i)
            time.Sleep(time.Duration(rand.Float32() * float32(time.Second)))
        }
    }()

    fmt.Println("Exiting.")
    time.Sleep(5 * time.Second)
}
```

Communication using channels

Wykorzystujemy kanał do przesłania informacji pomiędzy osobnymi goroutines.

```
func main() {
    ch := make(chan int)

    go func() {
        for i := 0; ; i++ {
            ch <- i
            time.Sleep(time.Duration(rand.Float32() * float32(time.Second)))
        }
    }()

    for i := 0; i < 5; i++ {
        fmt.Println("Hello", <-ch)
    }
    fmt.Println("Exiting.")
}
```

Communication using channels

Channels are first-class values.

```
func generate(ch chan int) {
    for i := 0; ; i++ {
        ch <- i
        time.Sleep(time.Duration(rand.Float32() * float32(time.Second)))
    }
}

func main() {
    ch := make(chan int)

    go generate(ch)

    for i := 0; i < 5; i++ {
        fmt.Println("Hello", <-ch)
    }
    fmt.Println("Exiting.")
}
```

Channels can be used for synchronization

Kiedy funkcja `main` usiłuje przeczytać wartość `<-ch` blokuje do momentu kiedy jest to możliwe.

Kiedy funkcja `generate` stara się wysłać `ch <-` wartość do kanału także czeka do momentu kiedy jest to możliwe.

Kanały pozwalają nam przesyłać informacje, a także synchronizować program.

Istnieją kanały buforowane, które nie mogą służyć do synchronizacji.

"Do not communicate by sharing memory; instead, share memory by communicating."

Generators

Kanały to zwykłe wartości, możemy zwracać je z funkcji lub przekazywać je jako argumenty.

```
func generate() <-chan int {
    ch := make(chan int)
    go func() {
        for i := 0; ; i++ {
            ch <- i
            time.Sleep(time.Duration(rand.Float32() * float32(time.Second)))
        }
    }()
    return ch
}

func main() {
    ch := generate()
    for i := 0; i < 5; i++ {
        fmt.Println("Hello", <-ch)
    }
    fmt.Println("Exiting.")
}
```

Generators 2

To przypomina jakiś serwis, zmieńmy nazwę funkcji i odbierajmy `string`.

```
func communicate(name string) <-chan string {
    ch := make(chan string)
    go func() {
        for i := 0; ; i++ {
            ch <- fmt.Sprintf("%s: %d", name, i)
            time.Sleep(time.Duration(rand.Float32() * float32(time.Second)))
        }
    }()
    return ch
}

func main() {
    ch1 := communicate("Bob")
    ch2 := communicate("Alice")
    for i := 0; i < 5; i++ {
        fmt.Println(<-ch1)
        fmt.Println(<-ch2)
    }
    fmt.Println("Exiting.")
}
```

Fan in

```
func fanIn(ch1, ch2 <-chan string) <-chan string {
    out := make(chan string)
    go func() {
        for {
            out <- <-ch1
        }
    }()
    go func() {
        for {
            out <- <-ch2
        }
    }()
    return out
}

func main() {
    ch := fanIn(communicate("Bob"), communicate("Alice"))
    for i := 0; i < 10; i++ {
        fmt.Println(<-ch)
    }
    fmt.Println("Exiting.")
}
```

Fan in - select

```
func fanIn(ch1, ch2 <-chan string) <-chan string {
    out := make(chan string)
    go func() {
        for {
            select {
            case v := <-ch1:
                out <- v
            case v := <-ch2:
                out <- v
            }
        }
    }()
    return out
}

func main() {
    ch := fanIn(communicate("Bob"), communicate("Alice"))
    for i := 0; i < 10; i++ {
        fmt.Println(<-ch)
    }
    fmt.Println("Exiting.")
}
```

Timeout

Używamy kanału, który blokuje przez określony czas.

```
func main() {
    ch := fanIn(communicate("Bob"), communicate("Alice"))
    for {
        select {
        case message := <-ch:
            fmt.Println(message)
        case <-time.After(500 * time.Millisecond):
            fmt.Println("I can't wait anymore!")
            return
        }
    }
}
```

Timeout

Stworzenie kanału przed pętlą doprowadzi do zakończenia całej komunikacji po upłynięciu podanego czasu.

```
func main() {
    ch := fanIn(communicate("Bob"), communicate("Alice"))
    timeout := time.After(2 * time.Second)
    for {
        select {
        case message := <-ch:
            fmt.Println(message)
        case <-timeout:
            fmt.Println("I can't wait anymore!")
            return
        }
    }
}
```

Fan out

```
type Task struct {
    Id int
}

func Worker(in <-chan Task) {
    for {
        task := <-in
        fmt.Println("Processing task", task.Id)
        time.Sleep(time.Duration(rand.Float32() * float32(time.Second)))
    }
}

func main() {
    ch := make(chan Task)

    for i := 0; i < 4; i++ {
        go Worker(ch)
    }

    for i := 0; i < 10; i++ {
        ch <- Task{Id: i}
    }
}
```

Workers - mutex

Periodycznie pobieramy dane z listy adresów.

```
type Resource struct {
    url        string
    polling    bool
    lastPolled int64
}

type Resources struct {
    data []*Resource
    lock *sync.Mutex
}
```

Workers - mutex

```
func Poller(res *Resources) {
    for {
        res.lock.Lock()
        var r *Resource
        for _, v := range res.data {
            if v.polling { continue }
            if r == nil || v.lastPolled < r.lastPolled { r = v }
        }
        if r != nil {
            r.polling = true
        }
        res.lock.Unlock()
        if r == nil {
            continue
        }

        // poll the URL

        res.lock.Lock()
        r.polling = false
        r.lastPolled = time.Nanoseconds()
        res.lock.Unlock()
    }
}
```

Workers - channels

Fan out + fan in.

```
type Resource string

func Poller(in, out chan *Resource) {
    for r := range in {
        // poll the URL
        out <- r
    }
}
```

"Do not communicate by sharing memory; instead, share memory by communicating."

Goroutines są naprawdę szybkie

Przekazujemy wartość przez łańcuch goroutines.

```
const n = 100 * 1000

func main() {
    leftmost := make(chan int)
    right := leftmost // just init
    left := leftmost
    for i := 0; i < n; i++ {
        right = make(chan int)
        go func(left, right chan int) {
            right <- 1 + <-left
        }(left, right)
        left = right
    }
    go func() {
        leftmost <- 1
    }()
    fmt.Println(<-left)
}
```

Co widać w tych przykładach?

Współbieżność pozwala na wprowadzenie paralelizmu.

Współbieżność pozwala łatwo skalować problem.

Dzięki współbieżność tworzenie praktycznych, prostych i paralelnych rozwiązań jest prymitywne.

Dzięki zastosowaniu odpowiedniego sposobu komunikacji nie musimy martwić się o synchronizację, jest ona skutkiem ubocznym sposobu w który piszemy program.

Twitter

Users:
313 000 000 monthly active users.

Requests:
1 000 000 000 monthly visits to sites with embedded tweets.

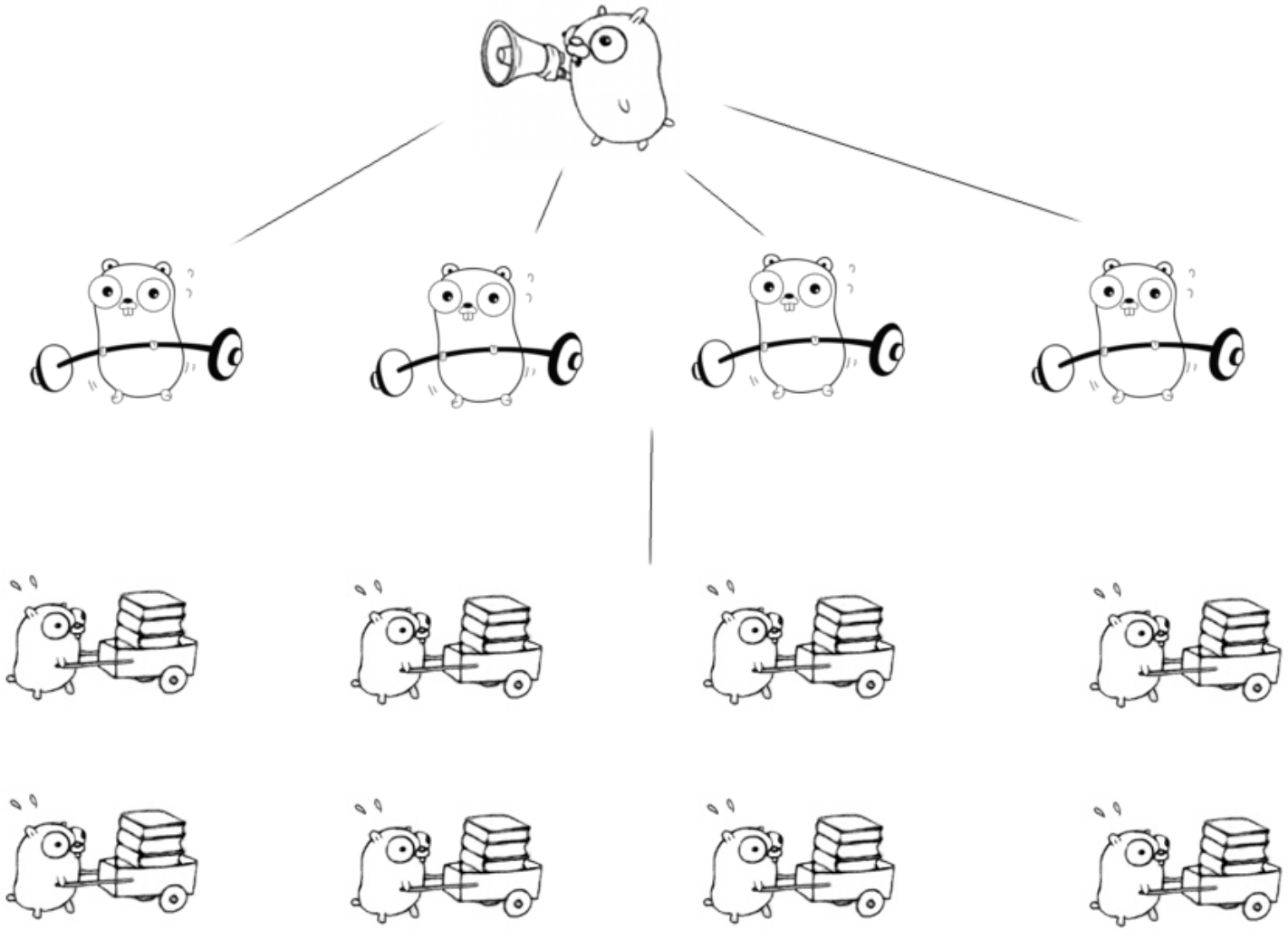
Tweets:
6000 tweets per second.

350 000 tweets per minute.

500 000 000 tweets per day.

180 000 000 000 tweets per year.

Twitter



Twittr! Czego potrzebujemy?

- Load balancer, który rozdziela przychodzące zapytania na serwery rendredujące
- Przechowywanie timeline'ów
- Odpytanie zreplikowanych serwerów przechowywujących timeline'y i zwrócenie pierwszej odpowiedzi (first-to-come)

First to come

Wysyłamy zapytanie do wszystkich serwerów które mogą ją zwrócić i odbieramy pierwszą odpowiedź.

```
func First(query string, replicas ...TimelineServer) Timeline {
    c := make(chan Timeline, len(replicas))
    for _, replica := range replicas {
        go func(s TimelineServer) {
            c <- s.Query(query)
        }(replica)
    }
    return <-c
}
```

Load balancer - fan out

```
type Task struct {
    Id int
}

func Worker(in <-chan Task) {
    for {
        task := <-in
        fmt.Println("Processing task", task.Id)
        time.Sleep(time.Duration(rand.Float32() * float32(time.Second)))
    }
}

func main() {
    ch := make(chan Task)

    for i := 0; i < 4; i++ {
        go Worker(ch)
    }

    for i := 0; i < 10; i++ {
        ch <- Task{Id: i}
    }
}
```

Load balancer - a better approach

Workery sygnalizują, że praca została wykonana wysyłając samych siebie przez dostarczony im kanał.

```
type Task struct {
    Id int
}

type Worker struct {
    tasks chan *Task
    load  int
}

func (w *Worker) work(done chan<- *Worker) {
    for {
        task := <-w.tasks
        fmt.Println("Processing task", task.Id)
        time.Sleep(time.Duration(rand.Float32() * float32(time.Second)))
        done <- w
    }
}
```

The actual balancer

Kanały pozwalają na synchronizację - nie używamy mutex'ów.

```
type Pool []*Worker

type LoadBalancer struct {
    pool *Pool
    done chan *Worker
}

func (b *LoadBalancer) Run(tasks chan *Task) {
    for {
        select {
        case task := <-tasks:
            b.dispatch(task)
        case worker := <-b.done:
            b.completed(worker)
        }
    }
}
```

Dispatch and completed

Dostęp do listy worker'ów i ich pól jest synchronizowany i nie ma możliwych zjawisk race condition.

```
func (b *LoadBalancer) dispatch(task *Task) {
    worker := b.pool.getLeastLoadedWorker()
    worker.tasks <- task
    worker.load++
}

func (b *LoadBalancer) completed(worker *Worker) {
    worker.load--
}
```

Timelines

Tworzymy potrzebne podstawowe struktury danych.

```
type UserID int

type Tweet struct {
    text string
    date time.Time
}

type User struct {
    timeline  []*Tweet
    followers []UserID
}
```

TimelineServer akceptuje requesty

Polecenia z zewnątrz przesyłane są kanałami.

```
type PutTweet struct {
    tweet  *Tweet
    target []UserID
}

type PutFollower struct {
    user   UserID
    target []UserID
}

type TimelineServer struct {
    Tweets  <-chan *PutTweet
    Follows <-chan *PutFollower
    users   map[UserID]*User
}
```

Funkcja run

Funkcja dodaje tweety i subskrypcje.

```
func (s *TimelineServer) run() {
    for {
        select {
        case putTweet := <-s.Tweets:
            for _, userID := range putTweet.target {
                s.users[userID].timeline = append(s.users[userID].timeline, putTweet.tweet)
            }
        case putFollower := <-s.Follows:
            for _, userID := range putFollower.target {
                s.users[userID].followers = append(s.users[userID].followers, putFollower.user)
            }
        }
    }
}
```

Dostęp z zewnątrz

Przesyłamy polecenia `PutTweet` i `PutFollower` do odpowiednich serwerów.

```
type Dispatcher struct {
    replicas []*TimelineServer
}

func (d *Dispatcher) follow(id, target UserID) {
    for replica := range d.findReplicas(target) {
        replica.Follows <- PutFollower{user: id, target: []UserID{target}}
    }
}

func (d *Dispatcher) tweet(id UserID, tweet *Tweet) {
    author := d.getFirstByID(id)
    for _, followerID := range author.followers {
        for replica := range d.findReplicas(followerID) {
            replica.Tweets <- PutTweet{tweet: tweet, target: author.followers}
        }
    }
}
```

Współbieżność w Go

Goroutines:
- tanie
- powszechnie używane

Channels:
- komunikacja
- synchronizacja

Dobrze wprowadzona komunikacja rozwiązuje problemy synchronizacji.

Wnioski

Złożone problemy łatwo rozbić na małe proste elementy.

Te elementy skomponowane współbieżnie prowadzą to prostych do zrozumienia, logicznych, skalujących się, a co najważniejsze poprawnych rozwiązań.

Współbieżność nie jest paralelizmem, ale potencjalnie prowadzi do paralelizmu.

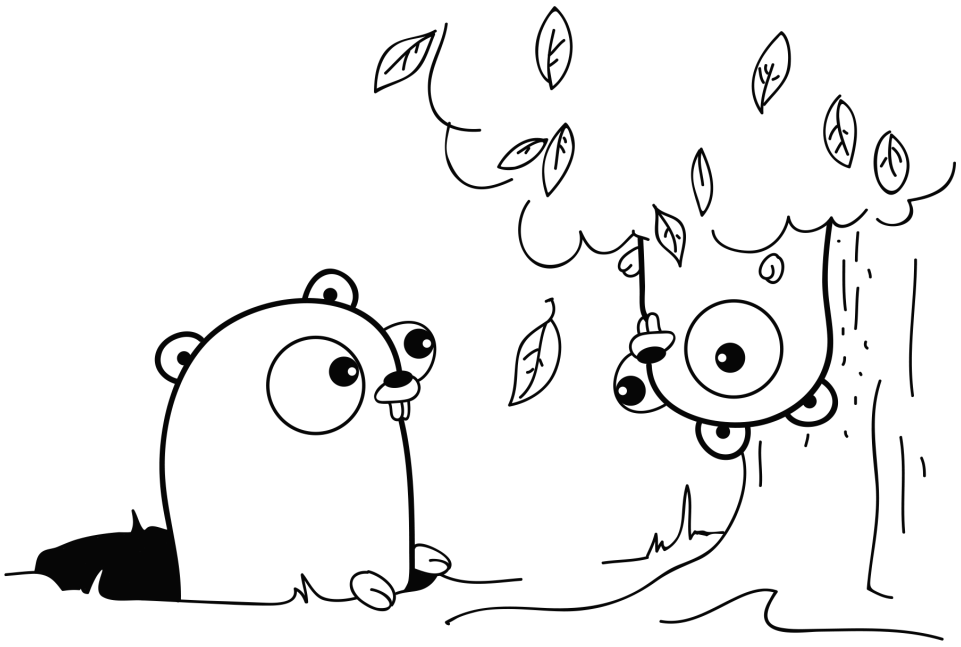
Współbieżność jest prosta i sprawia, że inne rzeczy są proste.

Co teraz?

golang.org/doc/ (https://golang.org/doc/)

"Concurrency is not parallelism" - Rob Pike (https://www.youtube.com/watch?v=cN_DpYBzKso)

"Go Concurrency Patterns" - Rob Pike (https://www.youtube.com/watch?v=f6kdp27TYZs)



Thank you

Filip Borkiewicz
https://0x46.net/ (https://0x46.net/)