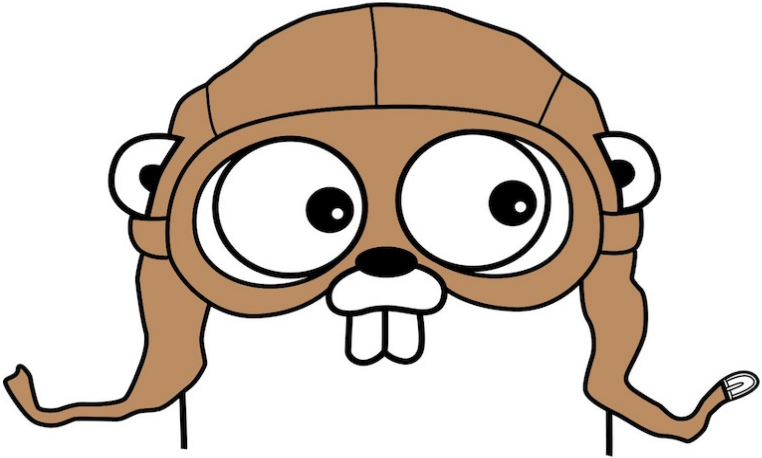
Intro to Concurrency in Go

Ioannis Sermetziadis
Backend Engineer

June 26, 2020

Roadmap

- Introduction to Go
- Introduction to Concurrency
- Concurrency + Go
- Why Go?



Introduction to Go

Language history

- Design started in 2007 @ Google
- Robert Griesemer, Rob Pike, and Ken Thompson, Ian Lance Taylor and Russ Cox
- 1.0 version on 03/2013, 6-month release cycle
- Latest version 1.14 on 02/2020
- Open sourced since 2009
- Inspired from paper Communicating Sequential Processes (Hoare, 1978)
- Concurrency features of other languages, like Limbo (Bell Labs)

Core values

- Programming productivity for multi-core, networked, large systems
- Code readability - required on large and complex codebases
- Simplicity / Minimalism
- Go 1 is fixed to language specification
- Address criticism of other languages by combining best-practices
- Solving the painpoints of large systems, like build time, dependency management, readability

Characteristics

- Static typing
- Garbage collection
- Modular - versioned modules and dependency management
- Cross-compile from / to different platforms using `$GOOS` and `$GOARCH`
- Provides binaries for freeBSD, linux, macOS and windows

Hello World!

```go
package main

import (
        "fmt"
)

func main() {
        fmt.Print("Hello DevStaff!")
}
```

Hello panic!

```
package main

import "time"

func main() {
	go func() {
		panic("failed!")
	}()

	time.Sleep(time.Second)
}
```

Hello HTTP!

- applies implicit concurrency
- a goroutine triggered per HTTP connection

```go
func helloHandler(w http.ResponseWriter, req *http.Request) {
	fmt.Fprintf(w, "hello DevStaffer!\n")
}

func main() {
	log.Println("listening...")
	http.HandleFunc("/", helloHandler)
	err := http.ListenAndServe("localhost:12345", nil)
	if err != nil {
		log.Fatal("ListenAndServe: ", err)
	}
}
```

HTTP client

- HTTP client is thread-safe - no extra effort to support calls by multiple goroutines

```go
resp, err := http.Get("http://localhost:12345")
if err != nil {
        log.Fatalln(err)
}

defer resp.Body.Close()

body, err := ioutil.ReadAll(resp.Body)
if err != nil {
        log.Fatalln(err)
}

log.Println(fmt.Sprintf("received: %s", string(body)))
```

Introduction to Concurrency

- Concurrency abstracts independently executing workloads and improves responsiveness
- "out of order" / non-deterministic execution that still produces consistently the same result
- Can make programs faster or slower
- Sequential thinking and sequential design are simpler, but don't model real-world problems
- Async callbacks are not easy to read and understand

Concurrency vs Parallelism

- Concurrency = when two or more tasks start, run and complete in overlapping time periods
- Tasks need to be interruptable
- Parallelism = when two or more tasks run at the same time
- Tasks need to be independent

CPU or IO bound

- CPU-bound workloads require parallelism because CPU cores are always busy
- Not all CPU-bound workloads (algorithms) are suitable for concurrency - its expensive to break-up work or combine results
- IO-bound workloads don't need parallelism - more threads than cores can improve workload execution
- IO-bound workloads are suitable for concurrency

OS Scheduler

- Software component responsible for allocating CPU time to processes or threads
- A Process can spawn many Threads allocated to 1 or more CPU cores
- Context-switch happens when a Thread is deallocated from a CPU core and another Thread is allocated
- Thread states - Waiting, Runnable, Executing
- Preemptive scheduling - time sliced scheduling - preempt/interrupt and resume

Concurrency in Go

goroutines (**go**)
channels (**chan**)
select (**select**)

Goroutine

- Goroutine also known as light-weight process or green thread
- Derived from coroutines and inspired by the Communicating Sequential Processes paper

Goroutines example

```go
const jobs = 100000

func main() {
	defer logSpan(time.Now())

	// WaitGroup synchronises on a collection of goroutines signal
	wg := sync.WaitGroup{}
	wg.Add(jobs)

	for i := 0; i < jobs; i++ {
		// spawn a go routine for each job
		go func() {
			// simulate some work
			time.Sleep(1 * time.Second)
			// signal finish
			wg.Done()
		}()
	}

	log.Println("waiting jobs to complete...")
	wg.Wait()
}
```

Channel

- Enable goroutine communication in blocking or non-blocking way
- Typed primitives that support send and receive operations and allow goroutines to **synchronise** and **communicate**
- Operations are blocking by default (unbuffered)
- Non-blocking send operation is possible by defining channel buffer capacity
- Goroutines can synchronize on channels without explicit locking, by blocking on send and receive
- Wired into the language API (e.g. time package)

Channel operations

- Receive operation using `for`, `select` or simple receive expression
- Receiver can query the state of the channel, using an extra return value ‘v, **ok** := <- ch‘
- A closed channel will return zero values (of channel type) on receive operation
- Send operation to a closed channel with trigger panic - sender should close the channel when it stops sending
- Channel directions increase type safety by statically checking the operation
- Write will block when buffer is full and read when buffer is empty

Producer-Consumer example

```go
func produce(msgs chan<- msg) {
	for {
		w := rand.Intn(100)
		msgs <- msg{
			value: w,
		}
		time.Sleep(time.Duration(w) * time.Millisecond)
	}
}

func consume(msgs <-chan msg) {
	for msg := range msgs {
		fmt.Println(msg.value)
	}
}

func main() {
	var msgs = make(chan msg)
	go produce(msgs)
	go consume(msgs)
	time.Sleep(2 * time.Second)
}
```

Select

- Control statement similar to switch, but decision is made based on the ability to communicate
- Goroutines can operate (send or receive) on multiple channels
- If possible to operate on multiple channels, chooses pseudo-randomly
- **default** clause provides safety for immediate outcome

Select example

```
func main() {
        input := make(chan int)
        output := make(chan int)

        select {
        case input <- 1:
        case _ = <-output:
        // case <-time.After(time.Second):
        //         log.Println("timed out")
        default:
                log.Println("no one was ready")
        }
}
```

Speadtest in Go

```
func main() {
        for i := 0; i < 10; i++ {
                var winner string
                select {
                case winner = <-ping("Google"):
                case winner = <-ping("Facebook"):
                case winner = <-ping("Twitter"):
                case winner = <-ping("LinkedIn"):
                case <-time.After(300 * time.Millisecond):
                        log.Println("none responded in time")
                        continue
                }
                log.Printf("got response from %s\n", winner)
        }
}
```

Why Go?

Go Scheduler

- Non-preemptive/co-operative scheduler that depends on user-space events used for scheduling decisions
- Part of the Go runtime built into each Go application
- Prevents thread context-switches by allocating work
- Uses Global Run Queue (GRQ) and Local Run Queue (LRQ) to keep track of the goroutines state and dynamically assign to Threads
- Goroutines are context-switched on and off a Thread, similar to Threads context-switched on a CPU core
- When a Goroutine is blocked (system call, IO, synchronisation), it gets de-allocated and moved to the network poller

OS vs Go scheduler

- Work Stealing - Goroutines are balanced across all available threads to avoid OS context-switches
- When a thread LRQ is empty, scheduler steals work from other thread LRQ
- In C, OS thread context-switching costs ∼1000 nanoseconds or ∼17k instructions - cache misses also possible
- In Go, Goroutine context-switching prevents thread content-switching - cost is ∼150 nanoseconds or ∼2.4k instructions
- The Go scheduler turns the IO-bound workload into CPU-bound work that leverages CPU capacity
- Preemptive vs Non-preemptive scheduling - timesliced execution vs cooperation

Data race detection

- Multiple goroutines access the same variable and at least one is a write
- The result of a program will vary randomly based on the order to access

Data race examples:

- Read of loop counter variable from child goroutines
- Accidentally shared variable
- Unprotected global variable

```go
func main() {
        for i := 0; i < 1000; i++ {
                state := 10
                wg := sync.WaitGroup{}
                wg.Add(2)
                go multiply(&state, &wg, 10)
                go deduct(&state, &wg, 2)

                wg.Wait()
                fmt.Println(state)
        }
}
```

Deadlock errors

- A group of goroutines are waiting for each other and none of them is able to proceed
- Possible in message-passing or shared-memory concurrency models
- Go detects only when the whole program freezes not for subset of goroutines
- Channel deadlocks are easier to detect than Lock deadlocks

```
func main() {
        ch := make(chan int)
        ch <- 1
        fmt.Println(<-ch)
}
```

Links

Tour of Go
`tour.golang.org/welcome/1`
Playground
`play.golang.org/`
Go Concurrency Patterns
`talks.golang.org/2012/concurrency.slide`
Advanced Go Concurrency Patterns
`blog.golang.org/io2013-talk-concurrency`
Scheduling In Go
`www.ardanlabs.com/blog/2018/08/scheduling-in-go-part1.html`