GOROUTINES AND CHANNELS IN PRACTICE

GUILHERME GARNIER

https://blog.guilhermegarnier.com

@gpgarnier

globo.com

We're hiring!

https://talentos.globo.com

tsuru


Java


python™


JS


Perl


C++

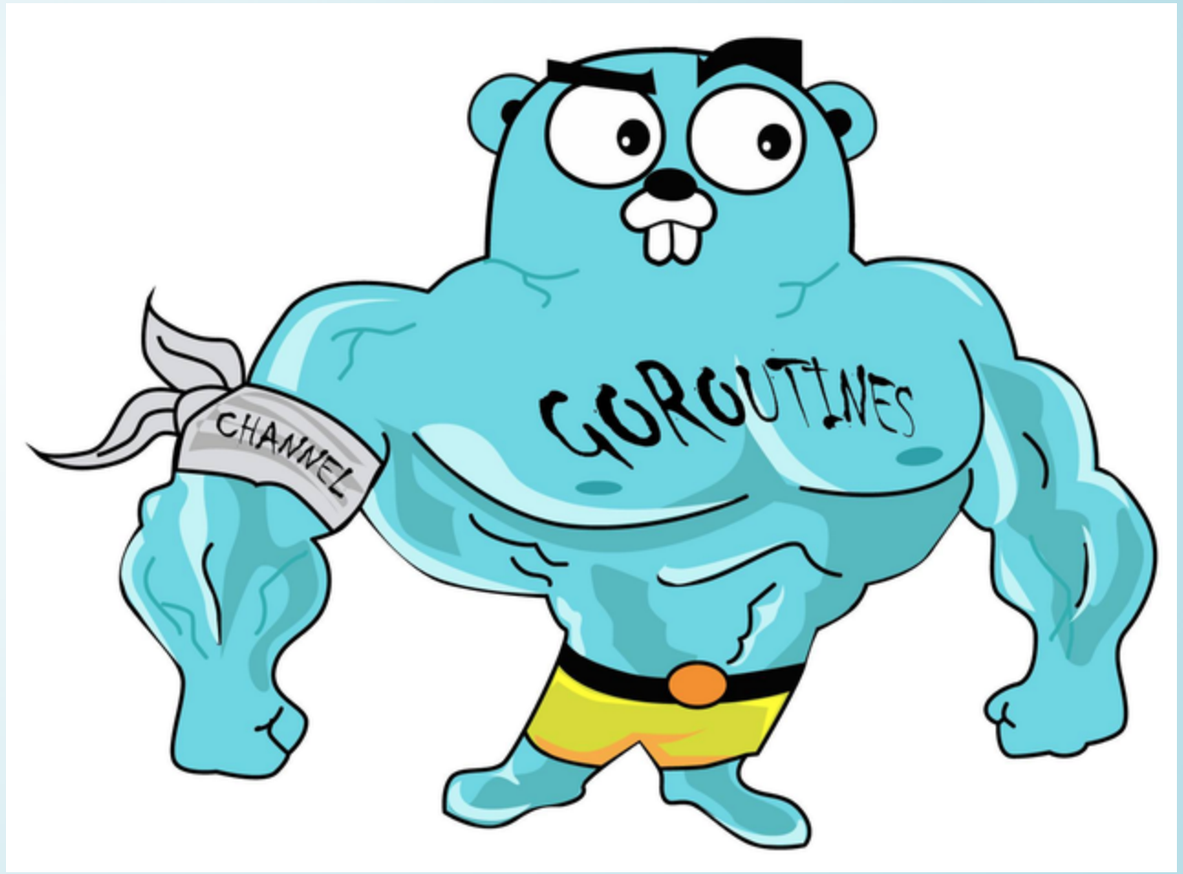

php

CONCURRENCY IS HARD

- Threads are expensive
- Hard to share state
- Hard to manage threads lifecycle

WHY IS GO DIFFERENT?

- simple concurrency model
- embraces concurrency with **goroutines**
- easy to share state with **channels**



WHAT'S A GOROUTINE?

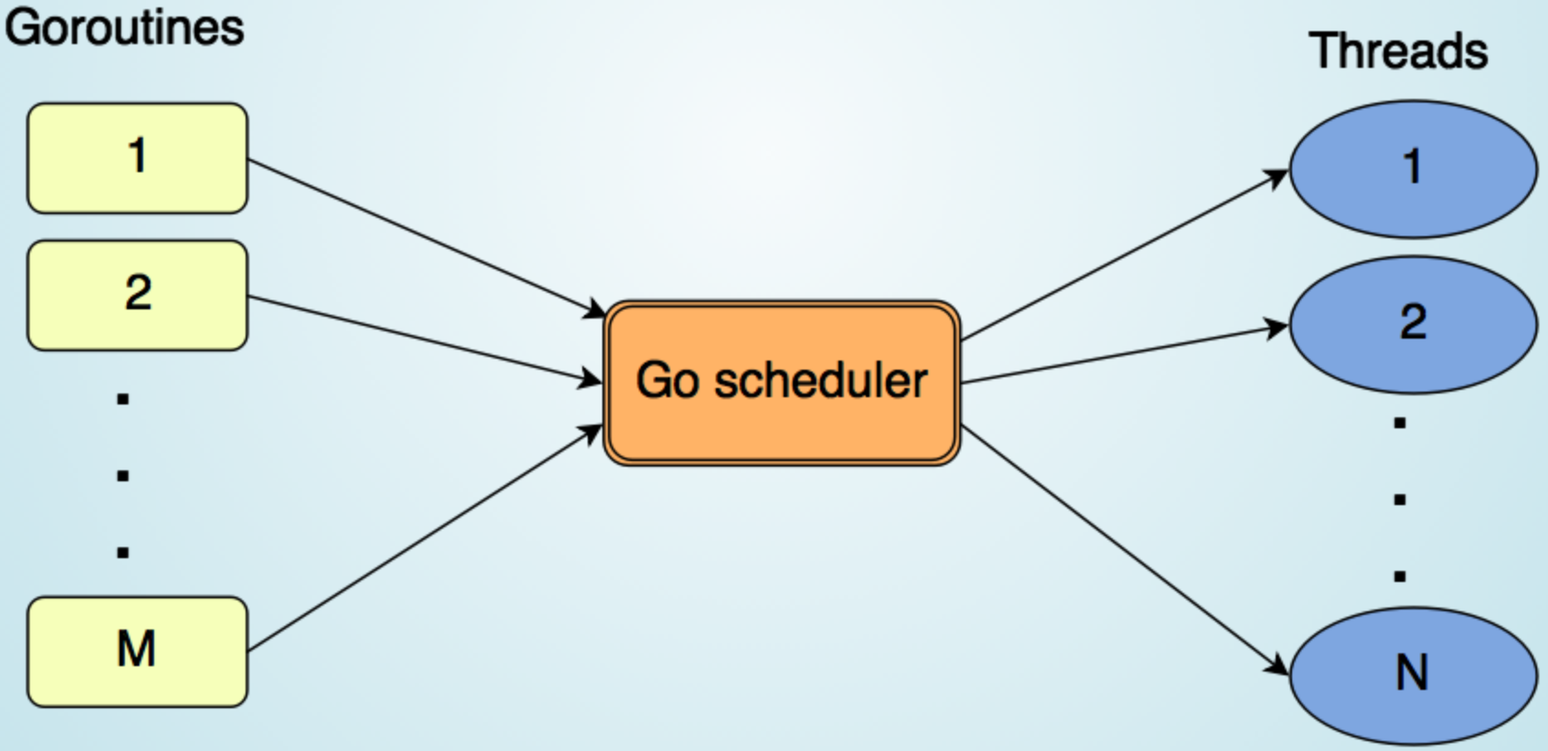
- Abstraction above OS threads
- Don't have an identity like thread IDs
- Not suspended unless sleeping, blocked or on a function call

SCHEDULING

	Goroutines	Threads
Scheduled by	Go runtime	OS kernel
Work distribution	defined in the code	defined by the Go scheduler

GO M:N SCHEDULER

Maps **M** goroutines to **N** threads



```
numGoroutines := 2
wg := sync.WaitGroup{}
wg.Add(numGoroutines)

for i := 0; i < numGoroutines; i++ {
    go func() {
        defer wg.Done()
        for i := 0; i < 100; i++ {
            for j := 0; j < 10000000; j++ {
            }
        }
    }()
}

wg.Wait()
```

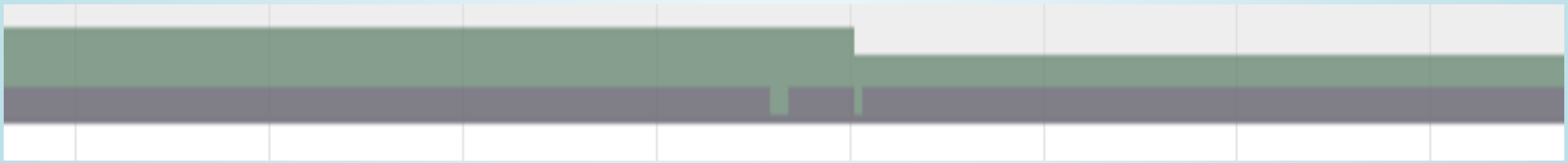
TRACING THE EXECUTION

Runnable goroutines

Running goroutines

`go run main.go` (multithread)

`GOMAXPROCS=1 go run main.go` (single thread)



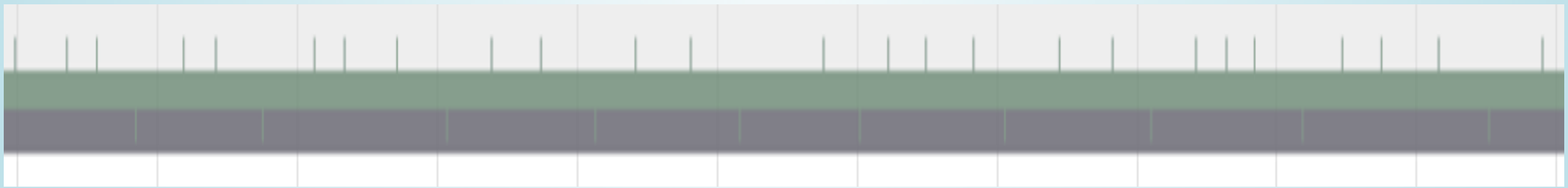
```
numGoroutines := 2
wg := sync.WaitGroup{}
wg.Add(numGoroutines)

for i := 0; i < numGoroutines; i++ {
    go func() {
        defer wg.Done()
        for i := 0; i < 100; i++ {
            for j := 0; j < 10000000; j++ {
            }
            time.Sleep(1 * time.Nanosecond)
        }
    }()
}

wg.Wait()
```

TRACING THE EXECUTION

Runnable goroutines

Running goroutines

`GOMAXPROCS=1 go run main.go` (single thread, with sleep)



STACK SIZE

THREADS

- fixed (~ 1-8 MB)
- wasteful for small jobs
- not enough for large jobs

GOROUTINES

- dynamic
- starts small (~ 4 kB)
- grows and shrinks when needed

GOROUTINES MEMORY USAGE

```
memConsumed := func() uint64 {...}
var ch <-chan interface{}
var wg sync.WaitGroup

const numGoroutines = 100000
wg.Add(numGoroutines)

before := memConsumed()
for i := 0; i < numGoroutines; i++ {
    go func() { wg.Done(); <-ch }()
}
wg.Wait()
after := memConsumed()

fmt.Printf("%.3f bytes", float64(after-before)/numGoroutines)
```

source: Concurrency in Go, page 44

GOROUTINES MEMORY USAGE

Mem (GB)	Goroutines
1	370 k
2	740 k
4	1.5 M
8	2.9 M
16	5.9 M
32	11.8 M
64	23.7 M

source: Concurrency in Go, page 44

STATE SHARING

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

https://blog.golang.org/share-memory-by-communicating

EXAMPLE: HEALTHCHECKS IN JAVA

```
public class Main {
    public static void main(String args[]) {
        String[] urls = {"url1", "url2", ...};
        Thread[] threads = new Thread[urls.length];
        for (int i = 0; i < urls.length; i++) {
            threads[i] = new Thread(new Healthcheck(urls[i]));
            threads[i].start();
        }

        try {
            for (Thread t : threads) {
                t.join();
            }
        } catch (Exception e) {}

        System.out.println(Healthcheck.getOkCount() + " ok, " +
            Healthcheck.getErrCount() + " errors");
    }
}
```

EXAMPLE: HEALTHCHECKS IN JAVA

```
class Healthcheck implements Runnable {
    private static int okCount = 0;
    private static int errCount = 0;

    public static synchronized void addOk() { Healthcheck.okCount++; }
    public static synchronized void addErr() { Healthcheck.errCount++; }
    public static int getOkCount() { return Healthcheck.okCount; }
    public static int getErrCount() { return Healthcheck.errCount; }

    public void run() {
        try {
            if (getStatusCode(this.url) == 200) {
                Healthcheck.addOk();
            } else {
                Healthcheck.addErr();
            }
        } catch(Exception e) {
            Healthcheck.addErr();
        }
    }

    private int getStatusCode(String url) throws Exception {
        ...
    }
}
```

EXAMPLE: HEALTHCHECKS IN GO

```
type Results struct {
    okCount  int
    errCount int
}

func main() {
    urls := []string{"url1", "url2", ...}

    r := runChecks(urls)
    fmt.Printf("%d ok, %d errors\n", r.okCount, r.errCount)
}
```

VERSION 1: SEQUENTIAL CODE

```
func runChecks(urls []string) Results {
    r := Results{}
    for _, url := range urls {
        resp, err := http.Head(url)
        if err != nil || resp.StatusCode != http.StatusOK {
            r.errCount++
        } else {
            r.okCount++
        }
    }

    return r
}
```

VERSION 2: USING GOROUTINES

```
func runChecks(urls []string) Results {
    r := Results{}
    lock := sync.Mutex{}
    for _, url := range urls {
        go func(url string) {
            resp, err := http.Head(url)
            if err != nil || resp.StatusCode != http.StatusOK {
                lock.Lock()
                r.errCount++
                lock.Unlock()
            } else {
                lock.Lock()
                r.okCount++
                lock.Unlock()
            }
        }(url)
    }

    // Wrong way to wait for goroutines to finish
    time.Sleep(2 * time.Second)

    return r
}
```

VERSION 3: USING CHANNELS

```
func runChecks(urls []string) Results {
    r := Results{}
    responses := make(chan bool, len(urls))
    for _, url := range urls {
        go func(url string) {
            resp, err := http.Head(url)
            success := err == nil && resp.StatusCode == http.StatusOK
            responses <- success
        }(url)
    }

    for i := 0; i < len(urls); i++ {
        success := <-responses
        if success {
            r.okCount++
        } else {
            r.errCount++
        }
    }

    return r
}
```

WHAT'S A CHANNEL?

- a conduit for streaming information
- concurrency-safe
- allows decoupling between sender and receiver

UNBUFFERED CHANNELS

- sender and receiver block each other
- provide guarantee of delivery

```
stream := make(chan int)
go func() {
    stream <- 1
}()
go func() {
    <-stream
}()
```

BUFFERED CHANNELS

- created with a maximum capacity
- sender and receiver block each other when it's full
- no guarantee of delivery

```
stream := make(chan int, 1)
go func() {
    stream <- 1
    stream <- 2 // blocks until the first receiver
}()
go func() {
    <-stream // blocks until the first sender
    <-stream
}()
```

CLOSED CHANNELS

- We can't send values anymore (`panic`!)
- We can still receive values sent before closing it
- If empty, receiving values gives the zero value of the channel type

```
ch := make(chan string, 2)
ch <- "foo"
ch <- "bar"
close(ch)

fmt.Println(<-ch) // "foo"
fmt.Println(<-ch) // "bar"
fmt.Println(<-ch) // ""
```

CLOSED CHANNELS

How to receive until the channel closes?

```
ch := make(chan string, 2)

go func() {
    ch <- "foo"
    ch <- "bar"
    close(ch)
}()

for v := range ch {
    fmt.Println(v)
}
```

If the channel is empty, `range` blocks until the channel is closed

MULTIPLEXING CHANNELS

```
select {
case v := <-ch1:
    fmt.Println("value read from ch1")
case v := <-ch2:
    fmt.Println("value read from ch2")
case ch1 <- 10:
    fmt.Println("value sent to ch1")
default:
    fmt.Println("channels not ready")
}
```

If more than one condition is ready, a `case` is randomly chosen

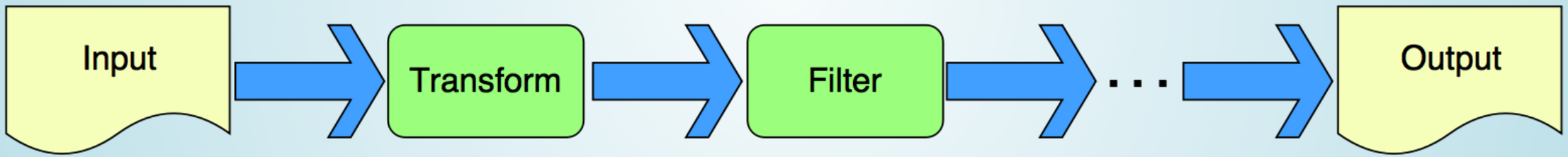
UNBLOCKING RECEIVES

```
ch := make(chan int)
select {
case v, ok := <-ch:
    if ok {
        fmt.Printf("Value is %d", v)
    } else {
        fmt.Print("channel is closed")
    }
default:
    fmt.Print("channel is empty")
}
```

TIMING OUT

```
ch := make(chan int)
select {
case v := <-ch:
    fmt.Printf("Value is %d", v)
case <- time.After(2*time.Second):
    fmt.Print("no data after 2 seconds")
}
```

PIPELINES



PIPELINES

```
// Inputs an infinite stream
randomNumbers := func() <-chan int {
    stream := make(chan int)
    go func() {
        defer close(stream) // This will never run
        for {
            stream <- rand.Intn(100)
        }
    }()
    return stream
}

// Transforms
double := func(input <-chan int) <-chan int {
    stream := make(chan int)
    go func() {
        defer close(stream)
        for i := range input {
            stream <- i * 2
        }
    }()
    return stream
}
```

PIPELINES

```
// Filters
onlyMultiplesOf10 := func(input <-chan int) <-chan int {
    stream := make(chan int)
    go func() {
        defer close(stream)
        for i := range input {
            if i%10 == 0 {
                stream <- i
            }
        }
    }()
    return stream
}

pipeline := onlyMultiplesOf10(double(randomNumbers()))
for i := range pipeline { // Infinite loop
    fmt.Println(i)
}
```

MANAGING GOROUTINES LIFECYCLE

LIMITING THE NUMBER OF RUNNING GOROUTINES

```
max := 3 // Max simultaneous goroutines
running := make(chan struct{}, max)

for url := range urls {
    running <- struct{}{} // waits for a free slot
    go func(url string) {
        defer func() {
            <-running // releases slot
        }()
        // do work
    }(url)
}
```

LIMITING THE NUMBER OF RUNNING GOROUTINES

```
max := 3 // Max simultaneous goroutines
running := make(chan struct{}, max)

go func() {
    for range time.Tick(100 * time.Millisecond) {
        fmt.Printf("%d goroutines running\n", len(running))
    }
}()

for url := range urls {
    running <- struct{}{} // waits for a free slot
    go func(url string) {
        defer func() {
            <-running // releases slot
        }()
        // do work
    }(url)
}
```

FORCING GOROUTINES TO STOP

Using a `done` channel

```
done := make(chan struct{})
go func() {
    defer func() {
        done <- struct{}{}
    }()
    bufio.NewReader(os.Stdin).ReadByte() // read input from stdin
}()

randomNumbers := func() <-chan int {
    stream := make(chan int)
    go func() {
        defer close(stream)
        for {
            select {
            case <-done:
                return
            default:
                stream <- rand.Intn(100)
            }
        }
    }()
    return stream
}
```

FORCING GOROUTINES TO STOP

Using context

```
ctx, cancelFunc := context.WithCancel(context.Background())
go func() {
    defer cancelFunc()
    bufio.NewReader(os.Stdin).ReadByte() // read input from stdin
}()

randomNumbers := func() <-chan int {
    stream := make(chan int)
    go func() {
        defer close(stream)
        for {
            select {
            case <-ctx.Done():
                return
            default:
                stream <- rand.Intn(100)
            }
        }
    }()
    return stream
}
```

FORCING GOROUTINES TO STOP

Using context with timeout

```
ctx, cancelFunc := context.WithTimeout(context.Background(), 10*time.Second)
go func() {
    defer cancelFunc()
    bufio.NewReader(os.Stdin).ReadByte() // read input from stdin
}()

randomNumbers := func() <-chan int {
    stream := make(chan int)
    go func() {
        defer close(stream)
        for {
            select {
            case <-ctx.Done(): // cancelFunc called or timeout reached
                return
            default:
                stream <- rand.Intn(100)
            }
        }
    }()
    return stream
}
```

COMMON CONCURRENCY PROBLEMS

DEADLOCK

```
ch1 := make(chan int)
ch2 := make(chan int)

go func() {
    n := <-ch2
    ch1 <- 2 * n
}()

n := <-ch1
ch2 <- 2 * n
```

DEADLOCK

```
$ go run main.go
fatal error: all goroutines are asleep - deadlock!

goroutine 1 [chan receive]:
main.main()
        main.go:12 +0xaa

goroutine 17 [chan receive]:
main.main.func1(0xc4200720c0, 0xc420072060)
        main.go:8 +0x3e
created by main.main
        main.go:7 +0x89
exit status 2
```

GOROUTINE LEAK

```
urls := make(chan string)

go func() {
    // defer close(urls)
    urls <- "http://example.com/1"
    urls <- "http://example.com/2"
}()

go func() {
    defer fmt.Println("This will never run")
    for url := range urls {
        http.Get(url)
        fmt.Println(url)
    }
}()
```

RACE DETECTOR TOOL

```
func runChecks(urls []string) Results {
    r := Results{}
    lock := sync.Mutex{}
    for _, url := range urls {
        go func(url string) {
            resp, err := http.Head(url)
            if err != nil || resp.StatusCode != http.StatusOK {
                lock.Lock()
                r.errCount++
                lock.Unlock()
            } else {
                lock.Lock()
                r.okCount++
                lock.Unlock()
            }
        }(url)
    }

    // Wrong way to wait for goroutines to finish
    time.Sleep(2 * time.Second)

    return r
}
```

RACE DETECTOR TOOL

```
$ go build -race
$ ./main
==================
WARNING: DATA RACE
Read at 0x00c4200ee2c0 by main goroutine:
    main.runChecks()
            main.go:35 +0x16b
    main.main()
            main.go:45 +0x87

Previous write at 0x00c4200ee2c0 by goroutine 7:
    main.runChecks.func1()
            main.go:26 +0x178

Goroutine 7 (finished) created at:
    main.runChecks()
            main.go:18 +0x123
    main.main()
            main.go:45 +0x87
==================
==================
WARNING: DATA RACE
...
==================
Found 2 data race(s)
```

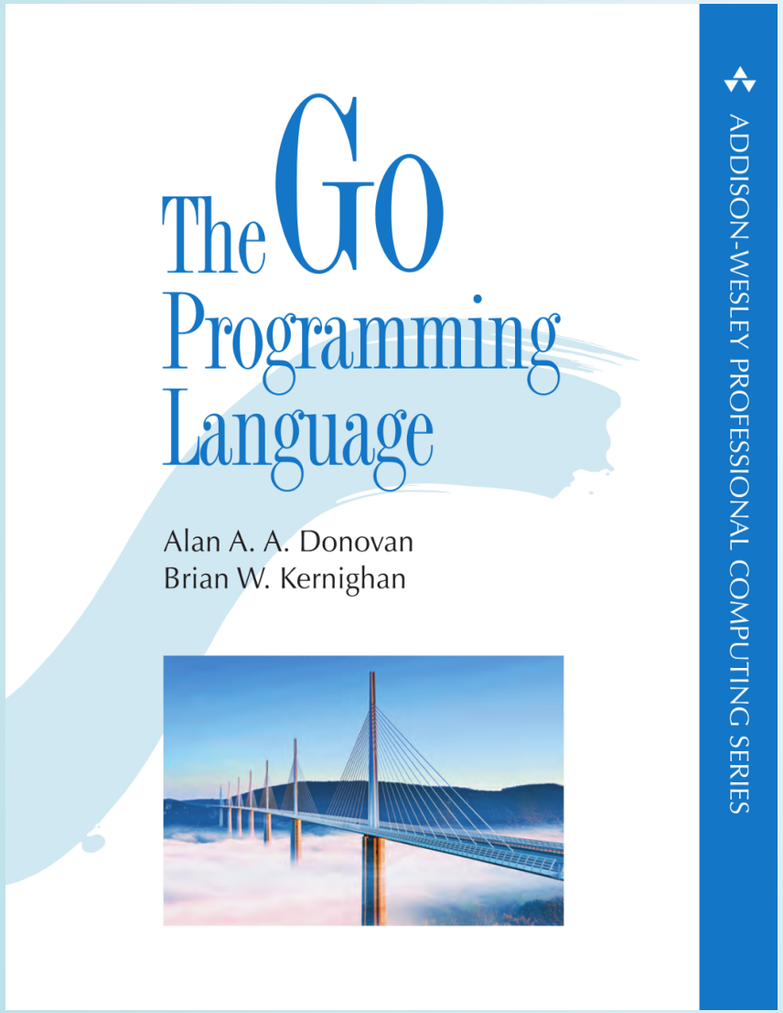
RACE DETECTOR TOOL

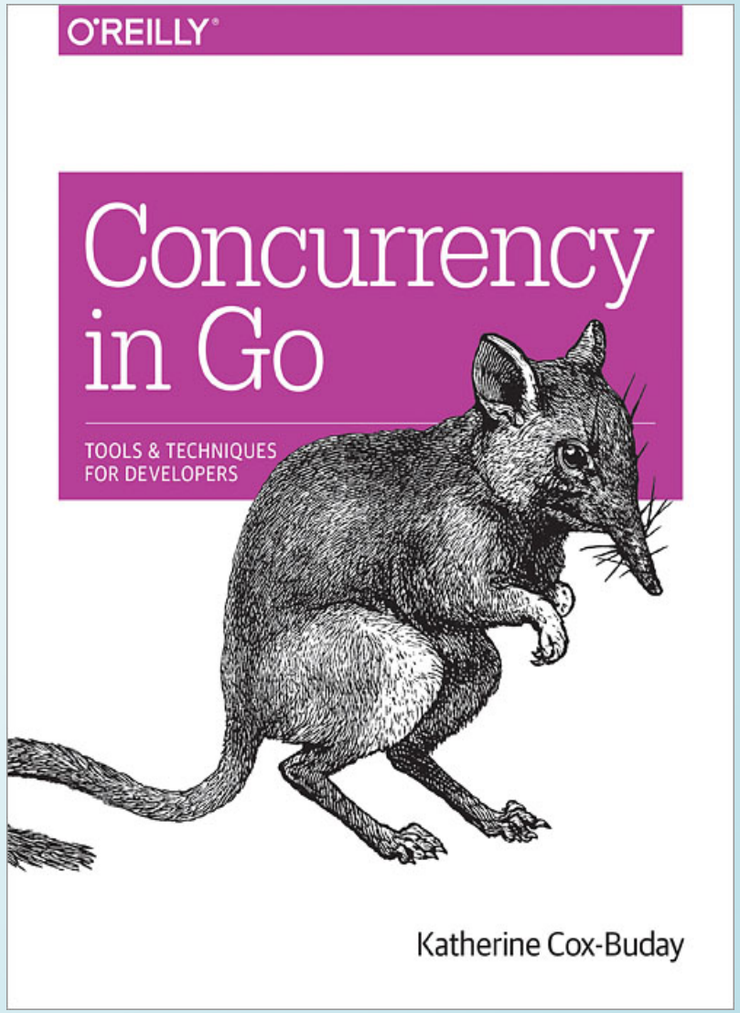
```
func runChecks(urls []string) Results {
    r := Results{lock: &sync.Mutex{}}
    var wg sync.WaitGroup
    for _, url := range urls {
        wg.Add(1)
        go func(url string) {
            defer wg.Done()
            resp, err := http.Head(url)
            if err != nil || resp.StatusCode != http.StatusOK {
                r.lock.Lock()
                r.errCount++
                r.lock.Unlock()
            } else {
                r.lock.Lock()
                r.okCount++
                r.lock.Unlock()
            }
        }(url)
    }

    wg.Wait() // Blocks until all `wg.Done()` calls
    return r
}
```

CONCLUSIONS

- Go treats concurrency as first-class citizen
- Goroutines make it easier to handle concurrent code
- Channels make it easier to share state and handle goroutines lifecycle
- Concurrency problems are still possible
- Do not abuse concurrency

REFERENCES


The Go Programming Language
Alan A. A. Donovan
Brian W. Kernighan
ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES


O'REILLY
Concurrency in Go
TOOLS & TECHNIQUES FOR DEVELOPERS
Katherine Cox-Buday

THANK YOU!

https://blog.guilhermegarnier.com

 @gpgarnier

Slides: https://blog.guilhermegarnier.com/talk-goroutines/