concurrency

concurrency in GO

part of the language

lightweight thread execution model

safe inter-thread communication

> “Do not communicate by sharing memory; instead, share memory by communicating”

feels natural (after a while)

sometimes hard to get right

goroutines

function executed concurrently

```
go func() {
    time.Sleep(1 * time.Second)
    fmt.Println("Hello, I am a go routine!")
}()
fmt.Println("Hello World!")

go gopher.Speak()
```

memory sharing

```
var a string
go func() { a = "hello" }()
fmt.Println(a)

values := []int{1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
for _, n := range values {
      go func() {
            fmt.Println(n)
       }()
}
> 10 10 10 10 10 10 10 10 10 10
```

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

pass values

e.g. pass variables to goroutines

```
values := []int{1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
for _, n := range values {
        go func(val int) {
                fmt.Println(val)
        }(n)
}
```

channels

use channels to send/receive data

sequentializing data flow

```
ch := make(chan int)

ch <- 1 // send
result := <- ch // receive
```

synchronize

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

doMoreWork()
<- ch // synchronization
```

sending data

```
var a string
go func() { a = "hello" }()
fmt.Println(a)
```

```
ch := make(chan string)

var a string
go func() {
        ch <- "hello"
}()

a = <-ch
fmt.Println(a)
```

channel types

```
ch := make(chan Person)

ch1 := make(chan int) //unbuffered channel
ch2 := make(chan int, 0)  //unbuffered channel
ch3 := make(chan *Person, 5)  // buffered channel

chan<- Person // send only
<-chan Person // receive only
```

semaphore with channels

```
var sem = make(chan int, MaxOutstanding)
func handle(r *Request) {
    sem <- 1 // Wait for active queue to drain.
    process(r) // May take a long time.
    <-sem // Done; enable next request to run.
}

func Serve(queue chan *Request) {
    for {
        req := <-queue
        go handle(req) // Don't wait for handle to finish.
    }
}
```

select

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

select {
case s := <-ch1:
        fmt.Printf("Received string from channel 1: %v", s)
case s := <-ch2:
        fmt.Printf("Received string from channel 2: %v", s)
}
```

ticker

```
ticker := time.NewTicker(2 * time.Second)
personCh := make(chan Person)

go func() {
    for {
        select {
        case p := <-personCh:
            fmt.Printf("Received person %+v", p)
        case <-ticker.C:
            fmt.Printf("Received tick")
        }
    }
}()
time.Sleep(10 * time.Second)
personCh <- Person{"Gopher", 4}
```

atomic operations

most operations are not atomic

i.e. maps are not synchronized

very easy to create nasty bugs

see packages "sync" & "sync/atomic"

better: use channels

mutexes

```
import "sync"

var mutex sync.Mutex
mutex.Lock()
mutex.Unlock()

var rwmutex sync.RWMutex
rwmutex.Lock()
rwmutex.Unlock()

rwmutex.RLock()
rwmutex.RUnlock()
```

mutex example

```
type TSInc struct {
        n int
        mtx sync.RWMutex
}

func (ts *TSInc) Inc() {
        ts.mtx.Lock()
        ts.n = ts.n + 1
        ts.mtx.Unlock()
}

func (ts *TSInc) Val() int {
        ts.mtx.RLock()
        val := ts.n
        ts.mtx.RUnlock()
        return val
}
```

defer

defer execution to the moment the surrounding function returns

in case of return & panic (error condition)

```
func (ts *TSInc) Val() int {
        ts.mtx.RLock()
        val := ts.n
        ts.mtx.RUnlock()
        return val
}

func (ts *TSInc) Val() int {
        ts.mtx.RLock()
        defer ts.mtx.RUnlock()
        return ts.n
}
```

panic, recover

built in functions to handle error conditions

```
func panic(interface{})
func recover() interface{}
```

panic interrupts the control flow and terminates all goroutines in the current scope

Not exceptions. Do not use for control flow.

panic example

```
func main() {
   defer func() {
      fmt.Printf("stopping...\n")
      if panicData := recover(); panicData != nil {
         fmt.Printf("Don't panic. Recovering from: %v\n", panicData)
      }
   }()

   panic("OMG OMG OMG!")
   fmt.Printf("Hello!")
}
```

gotchas

concurrency gets complicated quickly

whole websites with examples

https://go-traps.appspot.com

http://devs.cloudimmunity.com/gotchas-and-common-mistakes-in-go-golang/

care when handling with pointers

care when handling with closures

example

```
personCh := make(chan Person)
doneCh := make(chan int)

go func(ch chan<- Person) {
    time.Sleep(1 * time.Second) //work

    ch <- Person{"Gopher", 4} //send data

    time.Sleep(2 * time.Second) //work more

    doneCh <- 1 // notify
}(personCh)

go func(ch <-chan Person) {
    p := <-ch

    fmt.Println(p)
}(personCh)

fmt.Println("Started workers. Waiting ...")
<-doneCh //wait
fmt.Println("Done.")
```

session 4 – concurrency

https://github.com/iigorr/go-workshop

4-concurrency/README.md + CheatSheet.md

4.1 – SpecialMap

- race conditions
- mutexes

4.2 – Multiplex & Timeout

- goroutines
- channels

process end

```
func main() {
    go func() {
        time.Sleep(1 * time.Second)
        fmt.Println("Hello, I am a go routine!")
    }()
    fmt.Println("Hello World!")
}
```

> Hello World!

goroutine leak

```
func processStuff(inputChan chan int) {
	for {
		input := <-inputChan
		fmt.Println(input, "Hello World!!")
	}
}

ch := make(chan int, 0)
go processStuff(ch)
for i := 0; i < n; i++ {
	ch <- i
	time.Sleep(2 * time.Second)
	fmt.Printf("Number of goroutines: %d\n", runtime.NumGoroutine())
}
```