Go Lang Practical

IN5170/IN9170 – Models of concurrency

Go Lang Practical lecture 11

Fall 2022

5. October, 2022



History of Go

- first plans around 2007
- "IPO": end of 2009
- Precursor languages, resp. inspired by:
  - C
  - CSP / Occam
  - At Bell Labs
    - Squeak, Newsqueak
    - Limbo
    - Alef
  - Erlang, Concurrent ML

Go's non-revolutionary feature mix

- imperative
- object-oriented (?)
- compiled
- concurrent (goroutines)
- "strongishly" typed
- garbage collected
- portable
- higher-order functions and closures

Perhaps OO it is, but not as you know it

Go's *heterodox* take on OO

- *no* classes
- not even objects, officially
- no (class) inheritance
- interfaces as types[a]
- code reuse encouraged by
  - embedding
- name of an interface type $\neq$ interface type itself

[a] We concentrate here on the "OO" part of Go's type system, i.e., the interfaces. There are other types too, obviously.

What's a type?

Well, depends on whom you ask:

- compiler & run-time system?
  - a hint for the compiler of memory usage & representation layout?
  - piece of meta-data about a chunk of memory
- semanticist?
  - what's the *meaning* of a type?
- programmer?
  - types make my programs more safe, it's a partial specification
  - type systems stand in the way of my expert mastering of code
- orthodoxion oo'er?
  - a type is more or less a class (at least the more interesting ones/custom types)

Type systems

- important part of the "static analysis phase" of a compiler
- static vs. dynamic
- decidable typing (when statically typed)
- "Strong" typing

Milner's dictum

" well-typed programs cannot go wrong ".[a]

[a] That phrase corresponds to "safe" typing or type safety, not "strong" typing.

- balancing flexibility, safety, notational overhead, etc
- polymorphism

Typing: Static vs Dynamic

- **Static**: Types checked before run-time
- **Dynamic**: Types checked on the fly, during execution,Dynamically typed languages dont check types until execution.
- Static & Compiled and Dynamic & Interpreted are quite similar. But remember it is when types are checked vs. when source code is translated.
- **Type-checking has nothing to do with the language being compiled or interpreted!**

Dynamic, Interpreted

```
def check(a):    if a > 0:    print 'checked'
else:   print "3" + 5
check(2)
```

Because Python is both interpreted and dynamically typed, it only translates and type-checks code it is executing on. The else block never executes, so $\ddot{3} \ddot{+} 5$ is never even looked at!

Typing: Static vs Dynamic

What if it was statically typed?

A type error would be thrown before the code is even run. It still performs type-checking before run-time even though it is interpreted.

What if it was compiled?

The else block would be translated/looked at before run-time, but because it is dynamically typed it would not throw an error!

```
package main
import ("fmt")
func check(a int) {  if (a > 0) {fmt.Println("Hi")  }
        else { fmt.Println("3" + 5)  }}
        func main() {  check(2)}
```

The types are checked before running (static) and the type error is immediately caught!

Concurrency

Shared var's considered harmful

Go's concurrency mantra

"Don't communicate by sharing memory, share memory by communicating!"

- concurrency vs. parallelism

Go concurrency

goroutines + channels

- claimed to be "easy"
- first-class, typed channels

Goroutines

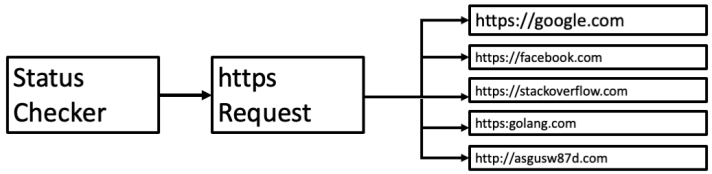
- Go's name for its unit of concurrency
- executing function calls asynchronously

- goroutine vs. threads
- "green threads"
- "lightweight" threads
- "threads minus monitor communication"
- goroutine dies when parent dies

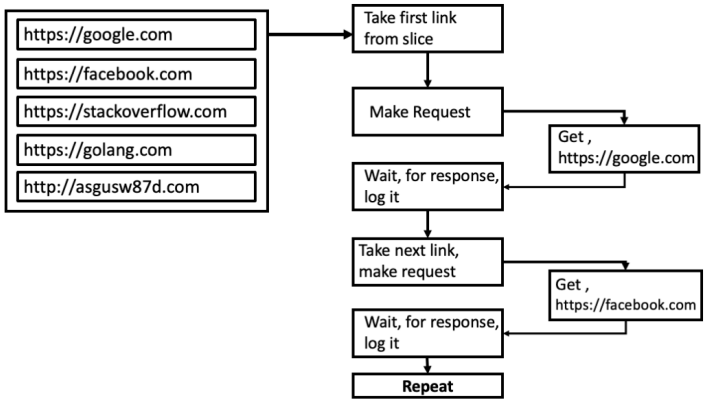
3 ways to call a function

- $f(x)$ – ordinary (synchronous) function call, where f is a defined function or a functional definition
- **go** $f(x)$ – called as an asynchronous process, i.e. go-routine (the go-routine will die when its parent process dies!
- **defer** $f(x)$ – the call is delayed until the end of the process

Without Go Routines - Example1



Without Go Routines - Example1



Go Routines - Example1

```
func main() {
        links := []string{
                "https://google.com",
                "https://facebook.com",
                "https://golang.org",
                "https://stackoverflow.com",
                "https://chinmayiprabhu.com",
        }
        c := make(chan string)

        for _, link := range links {
                go checklink(link, c)

        }
        for i := 0; i < len(links); i++ {
                fmt.Println(<-c)
        }

}
```

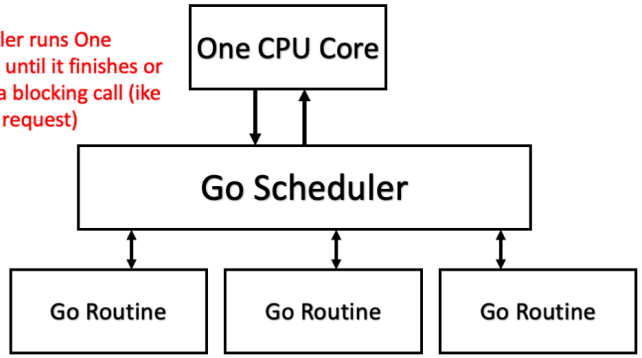
Go Routines- Example1 cont'd

```
func checklink(link string, c chan string) {
        _, err := http.Get(link)
        if err != nil {
                fmt.Println(link, "might␣be␣down!")
                c <- "␣Yep,␣Its␣down!"
                return
        }
        fmt.Println(link, "is␣up")
        c <- "␣yep,␣Its␣up!"
}
```

- Added the go keyword before each call of function *checklink*
- The five checklink goroutines starts up concurrently and five calls to http.Get are made concurrently as well.
- The program doesn't wait until one response comes back before sending out the next request.

Go Routines- Explained

Scheduler runs One routine until it finishes or makes a blocking call (ike a HTTP request)



Waiting for Go routines to finish in Go

Add() method

The Add method is used to add a counter to the WaitGroup.

Done() method

The Done method of WaitGroup is scheduled using a defer statement to decrement the WaitGroup counter.

Wait() method

The Wait method of the WaitGroup type waits for the program to finish all goroutines. The Wait method is called inside the main function, which blocks execution until the WaitGroup counter reaches the value of zero and ensures that all goroutines are executed.

Waiting for Go routines to finish

```
    func main() {
        var wg sync.WaitGroup
        var i int = -1
        var file string
        for i, file = range os.Args[1:] {
                wg.Add(1)
                go func(filename string) {
                        compress(filename)
                        wg.Done()
                }(file)
        }
        wg.Wait()
        fmt.Printf("compressed %d files \n", i+1)
}
```

Waiting for Go routines to finish

```
  unc compress(filename string) error {
        in, err := os.Open(filename)
        if err != nil {
                return err}
        defer in.Close()
        out, err := os.Create(filename + ".gz")
        if err != nil {
                return err}
        defer out.Close()
        gzout := gzip.NewWriter(out)
        _, err = io.Copy(gzout, in)
        gzout.Close()
        return err
}
```

- The WaitGroup type of sync package, is used to wait for the program to finish all goroutines launched from the main function.
- It uses a counter that specifies the number of goroutines, and Wait blocks the execution of the program until the WaitGroup counter is zero.

Channels

- "named pipes"
- FIFO, bounded, non-lossy communication
- crucial data type with synchronization power (see later)
- taking a back-seat:
  - locks
  - mutexes
  - monitors
  - semaphores...
- channels: *first-class* data
  - channels can send (reference to) channels
  - can be passed around by functions ....
  - inspired by CSP (and CCS, and, actually π)
- *directed* channels

Simple channel example in Go

```
func main() {
        messages := make(chan string, 0)  // declare + initialize

        go func() { messages <- "ping" }() // send
        msg := <-messages                  // receive
        fmt.Println(msg)
}
```

Channels

```
chan := make(chan int, buffersize)  // declare & initialize chanr
chan <- x                           // send (valuef) over chan
<- chan                             // receive from chan
y = <- chan                         // or :=
```

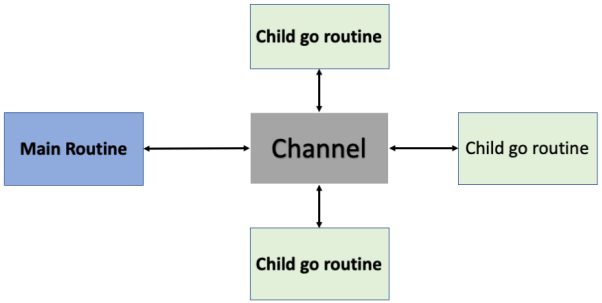
Run command: +go `run program.go` (compile and run program

Channels in Go

- Channels provide a way to send messages form one go routine to another.
- Go channels work like sockets between go routines within a single application. Like network sockets, they can be unidirectional or bidirectional.Â
- Channels are created with *make*, just like maps and slices.
- The arrow operator (<-) is used both to signify the direction of a channel and to send or receive data over a channel
- The select statement can watch multiple channels (zero or more). Until something happens, it will wait (or execute a default statement, if supplied).
- When a channel has an event, the select statement will execute that event.

Channels in Go

- Channels are bidirectional by default



Channel operations

- create channels (with capacity)
- close a channel[1]
- send and receive
- choice over channel communication
- different from `switch`
  - synchonization statement (`select {}`)!
  - no *first match*
- typical use: select over input

```
select {
    case i1 = <-c1:
        ...
    case i2 = <-c2:
  ...
  }
```

- "mixed" choice: possible

[1] don't forget, otherwise deadlocking!

Select Operation on Channels in Go

```
    func main() {
        done := time.After(30 * time.Second)
        echo := make(chan []byte)
        go readStdin(echo)
        for {
                select {
                case buf := <-echo:
                        os.Stdout.Write(buf)
                case <-done:
                        fmt.Println("␣Timed␣out")
                        os.Exit(0)}}
}

func readStdin(out chan<- []byte) {
        for {
                data := make([]byte, 1024)
                l, _ := os.Stdin.Read(data)
                if l > 0 {

                        out <- data
                }}
}
```

Lock implementation Channels in Go

```
    func main() {
        lock := make(chan bool, 1)
        for i := 1; i < 7; i++ {
                go worker(i, lock)
        }
        time.Sleep(10 * time.Second)
}

func worker(id int, lock chan bool) {
        fmt.Printf("%dwantsthelock\n", id)
        lock <- true
        fmt.Printf("%dhasthelock\n", id)
        time.Sleep(500 * time.Millisecond)
        fmt.Printf("%d is releasing the lock \n", id)
        <-lock
}
```

Producer Consumer Implementation in Go

```
    const producerCount int = 4
    const consumerCount int = 3

func produce(link chan<- string, id int, wg *sync.WaitGroup) {
        defer wg.Done()
for _, msg := range messages[id] {
link <- msg
        }
}

func consume(link <-chan string, id int, wg *sync.WaitGroup) {
        defer wg.Done()
        for msg := range link {
fmt.Printf("Message␣\"v\"␣is␣consumed␣by␣consumer␣v\n", msg, id)
        }
}


}}
```

Producer Consumer Implementation in Go

```
func main() {
        link := make(chan string)
        wp := &sync.WaitGroup{}
        wc := &sync.WaitGroup{}

        wp.Add(producerCount)
        wc.Add(consumerCount)

        for i := 0; i < producerCount; i++ {
                go produce(link, i, wp)
        }

        for i := 0; i < consumerCount; i++ {
                go consume(link, i, wc)
        }

        wp.Wait()
        close(link)
        wc.Wait()
```

Dining Philosophers by channels

```
// Goes from thinking to hungry to eating
// done eating then starts over.
func (p philosopher) eat() {
        defer eatWgroup.Done()
        for j := 0; j < 3; j++ {
                p.leftFork.Lock()
                p.rightFork.Lock()
                say("eating", p.id)
                time.Sleep(time.Second)
                p.rightFork.Unlock()
                p.leftFork.Unlock()
                say("finished eating", p.id)
                time.Sleep(time.Second)}

}
```

Dining Philosophers by channels:

Dining Philosophers by channels:

```
        // Create forks
        forks := make([]*fork, count)
        for i := 0; i < count; i++ {
                forks[i] = new(fork)
        }
        // Create philospoher,
        assign them 2 forks, send them to the dining table
        philosophers := make([]*philosopher, count)
        for i := 0; i < count; i++ {
        philosophers[i] = &philosopher{
        id: i, leftFork: forks[i], rightFork: forks[(i+1)%count]}
        eatWgroup.Add(1)
        go philosophers[i].eat()
        }
        eatWgroup.Wait()
}
```

References I