

LEARN THE GO PROGRAMMING LANGUAGE

For experienced developers or
those of an adventurous nature

gotutorial.net
[@GoTutorialNet](https://twitter.com/GoTutorialNet)

[Matt Nunogawa](#)
[@amattn](https://twitter.com/amattn)

LESSON 05

Concurrency
Goroutines & Channels

v0.2 draft

CONCURRENCY IN GO

- The concurrency story is based on two primitives:
 - goroutines
 - channels
- And one control flow statement:
 - select

GOROUTINES

GOROUTINES ARE...

- Conceptually, lightweight threads (green threads)
- To be very specific, they are not processes or OS threads or coroutines
- goroutines **share memory**
- A running program consists of one or more goroutines
- Spec definition: "a function call as an independent concurrent thread of control"

MORE INFO

- http://golang.org/ref/spec#Go_statements
- http://golang.org/doc/effective_go.html#goroutines

JUST USE GO

7

<http://play.golang.org/p/loGHRPuuZ3>

```
go fmt.Println("immediately")
go Later()

func Later() {
    time.Sleep(20 * time.Second)
    fmt.Print("Some Time Later")
}
```

MAIN()

- A running program consists of one or more goroutines
 - Except that main() is very special
- **When main() exits, all goroutines are immediately terminated**
- In this particular case, main exits long before fmt has a chance to print anything to the console:

```
package main
import "fmt"
func main() {
    go fmt.Println("If a goroutine is in a forest")
}
// outputs nothing at all
```

8

<http://play.golang.org/p/-s9db0YdNq>

WAITGROUPS

9

```
func main() {  
    var wg sync.WaitGroup  
    for i := 0; i < 5; i++ {  
        wg.Add(1)  
        x := i  
        go func() {  
            time.Sleep(/*random*/) 1) increment our  
            fmt.Println(x)          wait group  
            wg.Done()              3) after a little nap,  
                                   our goroutines will  
                                   decrement the wait  
                                   group  
        }()  
    }  
    wg.Wait() 2) wait for our wait  
              group's internal  
              counter to hit zero  
} // will output 0-4 in random order
```

<http://golang.org/pkg/sync/#WaitGroup>

<http://play.golang.org/p/bKxRyUIOOw>

WAITGROUPS

```
func main() {  
    var wg sync.WaitGroup  
    for i := 0; i < 5; i++ {  
        wg.Add(1)  
        go func() {  
            time.Sleep(*random*)  
            fmt.Println(i)  
            wg.Done()  
        }()  
    }  
    wg.Wait()  
}  
// will output 5's
```

i increments to 5
while i is 5, the our
goroutines are
sleeping...
and when they
wake up, they will
all print "5"

10

<http://golang.org/pkg/sync/#WaitGroup>

http://play.golang.org/p/SO5tCUyH_X

<http://play.golang.org/p/bKxRyUIOOw>

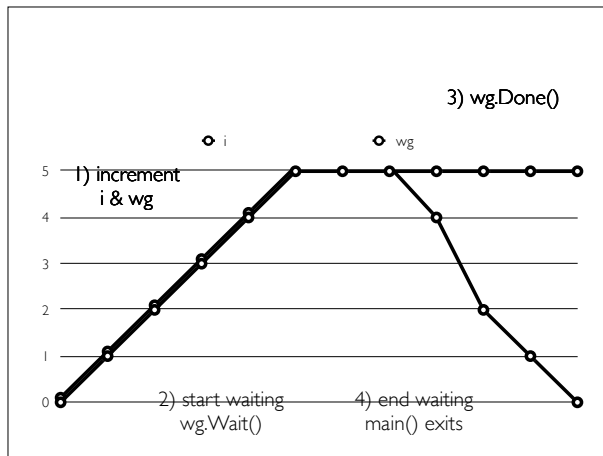
WAITGROUPS

11

```
func main() {  
    var wg sync.WaitGroup  
    for i := 0; i < 5; i++ {  
        wg.Add(1)           nuance!  
        x := i  
        go func() {  
            time.Sleep(/*random*/)   
            fmt.Println(x)  
            wg.Done()  
        }()  
    }  
    wg.Wait()  
}  
// will output 0-4 in random order
```

<http://golang.org/pkg/sync/#WaitGroup>

<http://play.golang.org/p/bKxRyUIOOw>



MORE THINGS

- goroutines are cheap
 - modern hardware can launch millions of them
 - stack size starts small and grow as necessary
- The scheduler can run goroutines on different processors/cores

MORE THINGS

14

```
# GOMAXPROCS env variable
# defaults to 1
export GOMAXPROCS=4

// or use the runtime package
import runtime

func main() {
    runtime.GOMAXPROCS(runtime.NumCPU())
}
```

CHANNELS

CHANNELS ARE...

- A type that provides communication
 - Channels are a reference type
- Conceptually they are bidirectional “typed pipes”
- Typically used to safely send/receive values across goroutines
- Also used for synchronization
 - send/receive are atomic operations

CREATION SYNTAX

- Use **make** to create channels

```
chInt      := make(chan int)
chShapes   := make(chan Shape)
chQs       := make(chan Quacker)
```

chQs is a variable that holds a channel than can tx/rx
anything that implements the Quacker interface

THE ARROW

- There is only a “leftward” arrow!
- To send data through a channel, use the arrow
`chInt <- 3`
- To receive data via a channel, use the arrow

```
var i int
i <-chInt // “pull” an int, store in i
<-chInt  // “pull” an int, discard it
```

BUFF/UNBUFF, ASYNC/SYNC

- By default, channels are unbuffered, or synchronous
- You can make asynchronous channels (buffered), but more on those later.

```
// buffer capacity of 42 ints:  
chInt := make(chan int, 42)
```

SYNCHRONOUS CHANNELS

- **Channel communication is atomic**

- A send operation on a channel blocks until a receiver is available for the same channel
- A receive operation for a channel blocks until a sender is available for the same channel
- When a channel is shared by two different goroutines, they will block until one is ready to send and the other is ready to receive.
- This is *synchronization* and it happens at the moment the data passes through the channel.

SYNCHRONOUS CHANNELS

```
func main() {  
    ch := make(chan int)  
    // we try to send 42, but we have no receivers  
    // the next line blocks and the whole thing hangs  
    ch <- 42  
}  
  
func main() {  
    ch := make(chan int)  
    // we try to receive an int, but we wait forever  
    // the next line blocks and the whole thing hangs  
    receivedInt := <-ch  
}
```

SYNCHRONOUS CHANNELS

```
func main() {  
    ch := make(chan int)  
    go func() {  
        // we try to send 42, but no receivers yet...  
        ch <- 42  
    }()  
  
    // after two seconds, pull a value from ch  
    time.Sleep(2 * time.Second)  
    receivedInt := <-ch  
    fmt.Println(receivedInt)  
}
```

RANDOM NUMBER PUMP

```
rn timer := make(chan int64)
go func() {
    for ;; {
        rn timer <- rand.Int63()
    }
}
```

But Mr. Go, why wouldn't you just call `rand.Int63()` directly?
Because this is a hypothetical example to drive a point...

HYPOTHETICAL HW-ASSISTED RANDOM NUMBER PUMP

```
// Assume we have a hardware entropy device.  
// Also assume that the hardware device can only be  
// read by one thread/process/goroutine at a time.
```

```
rnp := make(chan int64)  
go func() {  
    for ;; {  
        /* seed from HW device */  
        rand.Seed(GetHWEntropy())  
        rnp <- rand.Int63()  
    }  
}
```

Instead of mutexes we
can use a channel to
isolate access to
GetHWEntropy()

24

Even if 10,000 goroutines try to pull from the channel at the same time, conceptually, they are forced to wait their turn and be sent a random number one at a time.

IDIOMATIC, HYPOTHETICAL HW- ASSISTED RANDOM NUMBER PUMP

```
var rnp chan int64

// hide the global variable behind an accessor
func randomNumberPump() chan int64 {
    if rnp == nil {
        rnp = make(chan int64)
    }
    go func() {
        for ;; {
            /* seed from HW device */
            rand.Seed(GetHWEEntropy())
            rnp <- rand.Int63()
        }
    }()
    return rnp
}
```

Get used to "functions that return
channels." You will use them often

BUFFERED (ASYNCH) CHANNELS

- Buffered channels have an internal buffer with a fixed capacity:

```
// buffer capacity of 2 ints:  
chInt := make(chan int, 2)
```

- Buffered channels don't block, unless the buffer is full:

```
func main() {  
    ch := make(chan int, 2)  
    ch <- 42    // doesn't block  
    ch <- 43    // doesn't block  
    ch <- 44    // does block  
}
```

BUFFERED (ASYNCH) CHANNELS

- Buffer capacity cannot change
- Buffer capacity is not part of type

```
var someChannel chan int
var someOtherChannel chan int

someChannel = make(chan int, 2)
someChannel = make(chan int)
someOtherChannel = someChannel
someOtherChannel = make(chan int, 2)
```

<http://play.golang.org/p/QOMpYzkJWI>

CHANNEL EXAMPLES

- An implementation of the Sieve of Eratosthenes with channels:
 - http://golang.org/ref/spec#An_example_package
- <http://blog.golang.org/go-concurrency-patterns-timing-out-and>

28

http://en.wikipedia.org/wiki/Sieve_of_Eratosthenes

http://golang.org/ref/spec#An_example_package

FOR RANGE CHANNEL

- When receiving values indefinitely, we can use for and range
- This will loop and print out a random int64 every second:

```
func main() {  
    rnp := randomNumberPump()  
  
    for randomInt := range rnp {  
        fmt.Println(randomInt)  
        time.Sleep(1 * time.Second)  
    }  
}
```

<http://play.golang.org/p/qxmJABbes5>

CLOSING A CHANNEL

- Use the `close()` built-in:
`close(chInt)`
- Closing will terminate a `for range` statement
- Closed channels return the zero value when you try to pull from them
- You don't *have* to close a channel. It's not a file.
- Channels are garbage-collected regardless of being open or close
- You can use the two element variation to check if a channel is closed:
`i, isOpen := <-chInt`

CLOSING A CHANNEL

```
func main() {  
    rnp := randomNumberPump()  
  
    randomInt, isOpen := <-rnp  
    fmt.Println(randomInt, isOpen) // 55770067 true  
  
    close(rnp)  
  
    randomInt, isOpen = <-rnp  
    fmt.Println(randomInt, isOpen) // 0 false  
}
```

31

<http://play.golang.org/p/GxlpUK3w6L>

CLOSING A CHANNEL

```
func main() {  
    rnp := randomNumberPump()  
  
    go func() {  
        time.Sleep(5 * time.Second)  
        close(rnp)  
    }()  
  
    for randomInt := range rnp {  
        fmt.Println(randomInt)  
        time.Sleep(1 * time.Second)  
    }  
  
    fmt.Println("All Done")  
}
```

32

<http://play.golang.org/p/FhDOjiUBI9>

CHANNEL DIRECTIONALITY

- By default channels are bi-directional
- You can annotate a channel variable to make it unidirectional:

```
var recvOnly <-chan int  
var sendOnly chan<- int
```
- Under the hood, all channels are bidirectional, assigning a bidirectional channel to a unidirectional channels is a good practice for safety & understandability.

```
chInt := make(chan int)  
sendOnly = chInt  
recvOnly = chInt
```

SELECT

34

```
// Superficially, the select statement
// looks like a switch:

chInt := make(chan int)
chInt2 := make(chan int)
chString := make(chan string)

select {
case i := <-chInt:
    fmt.Printf("got an int %d", i)
case i2 := <-chInt2:
    fmt.Printf("got an int %d", i)
case s := <-chString:
    fmt.Printf("got a string %s", s)
default:
    fmt.Printf("default")
}
```

SELECT

Every case must be a send
or receive expression

All send/receive statements are
evaluated.

```
select {  
  case i := <-chInt:      If only one is ready, it is run.  
    fmt.Printf("got an int %d", i)  
  case i2 := <-chInt2:    If more than one is ready, one is  
    fmt.Printf("got an int %d", i) selected at random "fairly".  
  case s := <-chString:  
    fmt.Printf("got a string %s", s)  
  default:  
    fmt.Printf("default")    If none are ready, default is run.  
}  
                                if none are ready and there is no  
                                default clause, it blocks
```

35

The vast majority of the time, case statements are receives of some kind.

SELECT:TIMEOUT

```
chInt := make(chan int)

select {
  case i := <-chInt:
    fmt.Println("got an int", i)
  case <-time.After(5 * time.Second):
}
```

You don't have to use the
value you receive.

36

<http://play.golang.org/p/DGxc1qP9Sh>

REQ/RESP: BASIC SERVER

```
type Req struct {
    question string
    respChan chan *Resp
}

type Resp struct {
    responseCode int
    answer        string
}

func start(handler func(*Req), reqCh <-chan *Req) {
    for {
        go handler(<-reqCh)
    }
}
```

REQ/RESP: MAIN

```
func main() {  
    handler := func(req *Req) {  
        req.respChan <- &Resp{404, "I don't know the answer to '" + req.question + "'"}  
    }  
    reqCh := make(chan *Req)  
    respCh := make(chan *Resp)  
  
    go start(handler, reqCh)  
  
    go func() {  
        time.Sleep(1 * time.Second)  
        reqCh <- &Req{"What is your name?", respCh}  
        time.Sleep(1 * time.Second)  
        reqCh <- &Req{"What is your quest?", respCh}  
    }()  
  
    for {  
        select {  
        case resp := <- respCh:  
            fmt.Println(resp)  
        case <-time.After(3 * time.Second):  
            fmt.Println("timed out")  
            return // exit main  
        }  
    }  
}
```

38

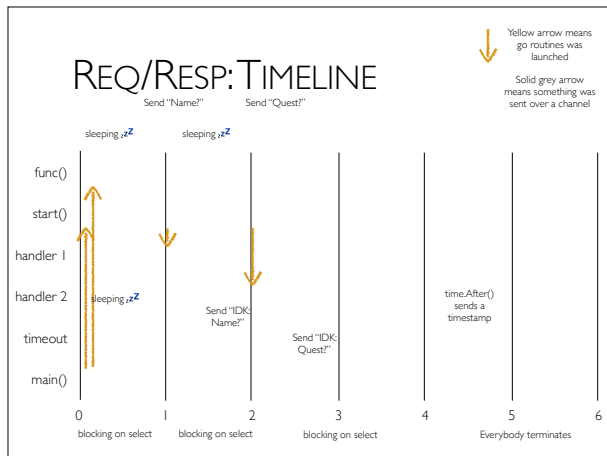
<http://play.golang.org/p/3RUAClgU1Q>

http://play.golang.org/p/U8T4U_ZIN8

39

<http://play.golang.org/p/llL4kVJmAz>

If it's not obvious, main() starts first and go



QUIT CHANNELS

```
func start(handler func(string) *Resp,  
    reqCh <-chan *Req,  
    quit <-chan bool) {  
    for {  
        select {  
        case req := <-requests:  
            req.respChan <- handler(req.question)  
        case <-quit:  
            return  
        }  
    }  
}
```


REQ/RESP: QUITTER

```
func main() {  
    handler := func(req *Req) {  
        req.respChan <- &Resp{404, "I don't know the answer to '"+req.question+"'"}  
    }  
    reqCh := make(chan *Req)  
    respCh := make(chan *Resp)  
  
    go start(handler, reqCh)  
  
    go func() {  
        time.Sleep(1 * time.Second)  
        reqCh <- &Req{"What is your name?", respCh}  
        time.Sleep(1 * time.Second)  
        reqCh <- &Req{"What is your quest?", respCh}  
    }()  
  
    quitCh := time.After(4 * time.Second)  
    for {  
        select {  
        case resp := <-respCh:  
            log.Println(resp)  
        case <-quitCh:  
            log.Println("quitting")  
            return // exit main  
        }  
    }  
}
```

41

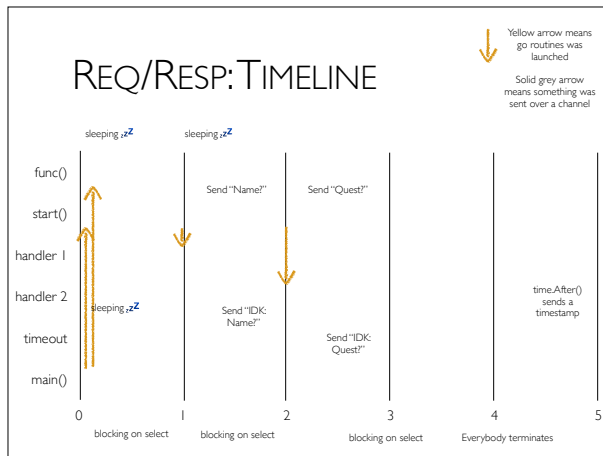
<http://play.golang.org/p/UF5Yoyexh0>

http://play.golang.org/p/hLKD9HFr_n

42

<http://play.golang.org/p/llL4kVJmAz>

If it's not obvious, main() starts first and go



PUTTING IT ALL TOGETHER

- Putting it all together:
 - channel send & receive gives you atomic communication
 - select gives you multiplexed receiving
 - channels as first class objects means that you can pass them around cheaply

THE GOLDEN RULE OF CONCURRENCY IN GO

44

Do not communicate by sharing memory.

Instead, share memory by communicating.

<http://blog.golang.org/share-memory-by-communicating>

MORE INFO

- <http://blog.golang.org/share-memory-by-communicating>
- <http://golang.org/doc/codewalk/sharemem/>
- <http://blog.golang.org/concurrency-is-not-parallelism>
- Go Concurrency patterns: <http://www.youtube.com/watch?v=f6kdp27TYZs>
- <http://blog.golang.org/advanced-go-concurrency-patterns>
- <http://blog.golang.org/go-concurrency-patterns-timing-out-and>
- <http://blog.golang.org/pipelines>
- http://golang.org/doc/effective_go.html#concurrency

LAST TIPS

- The learning curve is relatively steep
 - But worthwhile and powerful as you become familiar
- Performance is hard to predict: Use the profiler
- sync package has some useful primitives:
 - Once, Mutex, RWMutex, Cond

46

sync primitives are useful for simple problems... if you find yourself in mutex soup, it's a code smell and consider rethinking into a channel-based architecture.

THANK YOU, CREDITS & LICENSE

<http://gotutorial.net>
@GoTutorialNet

Matt Nunogawa
@amattn

- Much of the content is inspired by (and in some cases, outright taken from) a CCA3.0 Licensed (<http://creativecommons.org/licenses/by/3.0/us/>) 3 day Go Course by Rob Pike that predates Go 1.0 and is considered **out of date**:
 - <http://googlecode.com/hg-history/release-branch.r60/doc/GoCourseDay1.pdf>
 - <http://googlecode.com/hg-history/release-branch.r60/doc/GoCourseDay2.pdf>
 - <http://googlecode.com/hg-history/release-branch.r60/doc/GoCourseDay3.pdf>
- I owe many thanks to the many authors of Go and to Rob Pike in particular.
- These slides are Copyright 2013-2014 Matthew Nunogawa
- All content is licensed under the Creative Commons Attribution 4.0 License (<http://creativecommons.org/licenses/by/4.0/>)
 - attribution: Matt Nunogawa, Copyright 2013-2014 Matthew Nunogawa, <http://gotutorial.net>
- All code is licensed under a BSD License (<http://opensource.org/licenses/BSD-2-Clause>)

These are the slides that I used to learn go back in 2011.

“out of date”: The actually syntax has not significantly changed.

Some of the terminology is no longer in use, typically because after contact with the community, misunderstandings have occurred.

In the creation of these slides, I have, to the utmost of my ability, attempted to make sure that these are correct and updated. Any errors are likely my fault. I make no guarantee that these slides are correct or will remain correct under the inevitable progression of time.