Go Academy

#5 concurrency and channels

www.3fs.si

Concurrency

- Why?
- What?
- Parallelism?

A model for software construction

- Easy to understand
- Easy to use
- Much nicer than parallelism with threads, semaphores, locks…

Simple function

```
func print(msg string) {
  for i := 0; ; i++ {
    fmt.Println(msg, i)
    time.Sleep(time.Duration(rand.Intn(1000)) * time.Millisecond)
  }
}
```

Running the function

```
func main() {
  print("Foo!")
}

func print(msg string) {
  for i := 0; ; i++ {
    fmt.Println(msg, i)
    time.Sleep(time.Duration(rand.Intn(1000)) * time.Millisecond)
  }
}
```

Running it concurrently

```go
func main() {
  go print("Foo!")
}

func print(msg string) {
  for i := 0; ; i++ {
    fmt.Println(msg, i)
    time.Sleep(time.Duration(rand.Intn(1000)) * time.Millisecond)
  }
}
```

Waiting for output

```
func main() {
  go print("Foo!")
  fmt.Println("Waiting...")
  time.Sleep(3 * time.Second)
  fmt.Println("Ending!")
}
```

Goroutines

- Independently executing function launched by a go statement
- It has its own call stack, which grows and shrinks as required
- It's very cheap. You can have thousands, hundreds of thousands of goroutines
- It's not a thread
- Goroutines are multiplexed dynamically onto threads as needed
- You can think of it as a very cheap thread

Communication

- Previous example cheated
- It just printed on the screen
- Real conversations require communication

Channels

- Channels are a typed conduit through which you can send and receive values with the channel operator: `<-`

- `ch <- v    // Send v to channel ch`

- `v := <-ch  // Receive from ch, and assign value to v`

- The "arrow" indicates the direction of data flow.

- Like maps and slices, channels must be created before use:

- `ch := make(chan int)`

Using channels

```
func main() {
    c := make(chan string)
    go print("Foo!", c)
    for i := 0; i < 5; i++ {
        fmt.Println(<-c)
    }
}

func print(msg string, c chan string) {
    for i := 0; ; i++ {
        c <- fmt.Sprint(msg, i)
        time.Sleep(time.Duration(rand.Intn(1000)) * time.Millisecond)
    }
}
```

Synchronisation

- When the main function executes `<–c`, it will wait for a value to be sent
- When the print function executes `c <–value`, it waits for a receiver to be ready
- Sender and receiver must both be ready to play their part in the communication. Otherwise we wait until they are
- Channels both communicate and synchronise
- All of the above is only true for unbuffered channels

Generator: function that returns a channel

```
func print(msg string) <-chan string { // returns receive-only channel
    c := make(chan string)
    go func() { // launch the goroutine from inside the unction
        for i := 0; ; i++ {
            c <- fmt.Sprint(msg, i)
            time.Sleep(time.Duration(rand.Intn(1000)) * time.Millisecond)
        }
    }()
    return c // return the channel
}
func main() {
    c := print("Foo!")
    for i := 0; i < 5; i++ {
        fmt.Println(<-c)
    }
}
```

Channels as handle on a service

- We can have more instances of the service

```
func main() {
    foo := print("Foo!")
    bar := print("Bar!")
    for i := 0; i < 5; i++ {
        fmt.Println(<-foo)
        fmt.Println(<-bar)
    }
}
```

Multiplexing

- Previous example ran in lockstep
- Use fan-in function to let whosever is ready to run

```
func fanIn(input1, input2 <-chan string) <-chan string {
    c := make(chan string)
    go func() { for { c <- <-input1 } }()
    go func() { for { c <- <-input2 } }()
    return c
}
func main() {
    foo := fanIn(print("Foo!"), print("Bar!"))
    for i := 0; i < 10; i++ {
        fmt.Println(<-c)
    }
}
```

Select

- The select statement provides another way to handle multiple channels
- Like a switch, but each case is a communication
- All channels are evaluated
- Selection blocks until one communication can proceed
- If multiple can proceed, select chooses pseudo randomly
- A default clause, if present executes immediately if no channel is ready

```
select {
case v1 := <-c1:
    fmt.Println("c1", v1)
case v2 := <-c2:
    fmt.Println("c2", v2)
case c3 <- 23:
    fmt.Println("sent 23 to c3")
default:
    fmt.Println("no one was ready")
}
```

Multiplexing using select

```
func fanIn(input1, input2 <-chan string) <-chan string {
    c := make(chan string)
    go func() {
        for {
            select {
            case s := <-input1:
                c <- s
            case s := <-input2:
                c <- s
            }
        }
    }()
    return c
}
```

Timeout using select

```
func main() {
    c := print("Foo!")
    for {
        select {
        case s := <-c:
            fmt.Println(s)
        case <-time.After(800 * time.Millisecond):
            fmt.Println("too slow")
            return
        }
    }
}
```

Timeout the whole conversation

```
func main() {
    c := print("Foo!")
    timeout := time.After(5 * time.Second)
    for {
        select {
        case s := <-c:
            fmt.Println(s)
        case <-timeout:
            fmt.Println("5 seconds elapsed")
            return
        }
    }
}
```

Quit channel

```
func print(msg string, quit <-chan bool) <-chan string {
    ...
        for i := 0; ; i++ {
            time.Sleep(time.Duration(rand.Intn(1000)) * time.Millisecond)
            select {
            case c <- fmt.Sprint(msg, i): //do nothing
            case <-quit: return
            }
        }
   ...
}
func main() {
    quit := make(chan bool)
    c := print("Foo!", quit)
    for i := 0; i < 5; i++ {
        fmt.Println(<-c)
    }
    quit <- true
}
```

Receive on quit channel

```
func print(msg string, quit chan bool) <-chan string {
    ...
            case <-quit:
                time.Sleep(time.Second)
                fmt.Println("cleanup")
                quit <- true
                return
            }
   ...
}
func main() {
    ...
    for i := 0; i < 5; i++ {
        fmt.Println(<-c)
    }
    quit <- true
    <-quit
    fmt.Println("done")
}
```

Buffered channels

- Same as channels, but with size: `ch := make(chan int, 10)`
- Writing to channel isn't blocking until the channel isn't full
- Reading from channel isn't blocking until the channel isn't empty

Basic buffered channel example

```go
func main() {
   c := make(chan int, 2)
   c <- 1
   c <- 2
   fmt.Println(<-c)
   fmt.Println(<-c)
}
```

Non working buffered channel example

```
func main() {
   c := make(chan int, 2)
   c <- 1
   c <- 2
   c <- 3
   fmt.Println(<-c)
   fmt.Println(<-c)
}
```

```
func main() {
   c := make(chan int, 2)
   c <- 1
   c <- 2
   fmt.Println(<-c)
   fmt.Println(<-c)
   fmt.Println(<-c)
}
```

```
fatal error: all goroutines are asleep - deadlock!
```

Logging

```
c := make(chan string)
func log(message string) {
   c <- message
}

func worker() {
  data := ""
  for {
    data += <- c
    if (len(data) > 10000) {
      //flush data to disk
      //empty memory
    }
  }
}
```

```
func main() {
  go worker()
}

func httpHandler(req *http.Request) {
  log(fmt.Sprint(req.RemoteAddr,
        req.URL.Path))

  //process request
}
```

Buffered Logging

```
c := make(chan string, 5)
func log(message string) {
  c <- message
}

func worker() {
  data := ""
  for {
    data += <- c
    if (len(data) > 10000) {
      //flush data to disk
      //empty memory
    }
  }
}
```

```
func main() {
  for i := 0; i < 5; i++ {
    go worker()
  }
}

func httpHandler(req *http.Request) {
  log(fmt.Sprint(req.RemoteAddr,
        req.URL.Path))

  //process request
}
```

Range and close

- A sender can `close` a channel to indicate that no more values will be sent
- Receivers can test whether a channel has been closed by assigning a second parameter to the receive expression: `v, ok := <-ch`
  - `ok` is false if there are no more values to receive and the channel is closed
- The loop `for i := range c` receives values from the channel repeatedly until it is closed
- You don't usually need to close channels. It's only necessary when the receiver must be told there are no more values coming, such as to terminate a range loop

Range and close example

```
func count(c chan int) {
    for i := 0; i < 10; i++ {
        c <- i
    }
    close(c)
}

func main() {
    c := make(chan int, 5)
    go count(c)

    for i := range c {
        fmt.Println(i)
    }
}
```

Conclusion

- Channels and goroutines are fun to play with and very powerfull, but don't overuse them
- Sometimes all you need is a reference counter
- Go has `sync` and `sync/atomic` packages that might solve your problems
- Always use the right tool for the job

Homework

- Continue work on your calculator
- Add support for parentheses: `(1+2)+(2*3)`
- Each parentheses group should get calculated in it's own goroutine