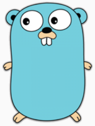
Programming in Go
Lesson 5: Concurrency

Matt Holiday
14 May 2019

Cardinal Peak



Lesson #5

What we'll cover today:

- Homework #4
- Concurrent programming in Go
- Goroutines
- Channels
- Select
- Mutexes and package sync

Homework #4

```go
package main

import (
    "fmt"
    "log"
    "net/http"
    "strconv"
)

// NOTE: don't do this in real life
type dollars float32

func (d dollars) String() string {
    return fmt.Sprintf("$%.2f", d)
}

type database map[string]dollars
```

Homework #4

```go
func (db database) list(w http.ResponseWriter,
                        req *http.Request) {
    for item, price := range db {
        fmt.Fprintf(w, "%s: %s\n", item, price)
    }
}

func (db database) add(w http.ResponseWriter,
                       req *http.Request) {
    item := req.URL.Query().Get("item")
    price := req.URL.Query().Get("price")

    if _, ok := db[item]; ok {
        w.WriteHeader(http.StatusBadRequest) // 400

        fmt.Fprintf(w, "duplicate item: %q\n", item)
        return
    }
```

Homework #4

```
    if f64, err := strconv.ParseFloat(price, 32); err != nil {
        w.WriteHeader(http.StatusBadRequest) // 400

        fmt.Fprintf(w, "invalid price: %q\n", price)
    } else {
        db[item] = dollars(f64)

        fmt.Fprintf(w, "added %s with price %s\n", item,
                    dollars(f64))
    }
}

func (db database) update(w http.ResponseWriter,
                          req *http.Request) {
    item := req.URL.Query().Get("item")
    price := req.URL.Query().Get("price")
```

Homework #4

```
    if _, ok := db[item]; !ok {
        w.WriteHeader(http.StatusNotFound) // 404

        fmt.Fprintf(w, "no such item: %q\n", item)
        return
    }

    if f64, err := strconv.ParseFloat(price, 32); err != nil {
        w.WriteHeader(http.StatusBadRequest) // 400

        fmt.Fprintf(w, "invalid price: %q\n", price)
    } else {
        db[item] = dollars(f64)

        fmt.Fprintf(w, "new price %s for %s\n", dollars(f64),
                    item)
    }
}
```

Homework #4

```go
func (db database) fetch(w http.ResponseWriter,
                         req *http.Request) {
    item := req.URL.Query().Get("item")

    if _, ok := db[item]; !ok {
        w.WriteHeader(http.StatusNotFound) // 404

        fmt.Fprintf(w, "no such item: %q\n", item)
        return
    }

    fmt.Fprintf(w, "item %s has price %s\n", item,
                db[item])
}
```

Homework #4

```go
func (db database) drop(w http.ResponseWriter,
                        req *http.Request) {
    item := req.URL.Query().Get("item")

    if _, ok := db[item]; !ok {
        w.WriteHeader(http.StatusNotFound) // 404

        fmt.Fprintf(w, "no such item: %q\n", item)
        return
    }

    delete(db, item)
    fmt.Fprintf(w, "dropped %s\n", item)
}
```

Homework #4

```go
func main() {
    db := database{"shoes": 50, "socks": 5}

    http.HandleFunc("/list", db.list)
    http.HandleFunc("/create", db.add)
    http.HandleFunc("/update", db.update)
    http.HandleFunc("/delete", db.drop)
    http.HandleFunc("/read", db.fetch)

    log.Fatal(http.ListenAndServe("localhost:8000", nil))
}
```

Share memory by communicating

Concurrency

It’s when you need to deal with two things at the same time

Parallelism expands that — two things happen at the same time

You can have concurrency with a single-core processor

We need it to take advantage of multi-core processing: chips aren’t getting faster as fast as they used to

It’s not avoidable — so how to deal with it?

Communicating sequential processes

A model for thinking about concurrency that makes it *less* hard

It’s always hard: the human brain wasn’t made to think this way

There’s a lot of push now for event-driven or “reactive” models — they’re **hard** hard

How about a simple program that talks to another program?

That’s what CSP is about — break your program into nanoservices

Goroutines

A goroutine isn't a thread, but you can think of it that way

It's easy to start a goroutine

The trick is knowing how the goroutine will stop:

- you have a well-defined loop terminating condition
- you have a channel that closes to signal completion
- you let it run until the program stops

But you need to make sure it doesn't get blocked by mistake

Channels

A channel is like a one-way socket or a Unix pipe

It’s a method of synchronization as well as communication

We know that a write always *happens before* a read

It’s also a vehicle for transferring ownership of data, so that only one goroutine at a time is writing the data

“Don’t communicate by sharing memory; instead, **share memory by communicating.”** — Rob Pike

Simple example

```
results := make(chan int)

// asynchronously yield a stream of integers

go func(limit int, out chan<- int) {
    for i := 0; i < limit; i++ {
        out <- i
    }

    close(out)  // else we would deadlock
}(10, results)

// receive them when they're ready

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

Channels

A goroutine blocks waiting to read an empty channel

A goroutine blocks waiting to write to a full channel

Some channels have buffers, so that a writer doesn’t wait

We need to make sure goroutines don’t block forever

One way to do that is multiplexing with `select`

Select

`select` allows any “ready” alternative to proceed among

- a channel we can read from
- a channel we can write to
- a `default` action that’s always ready

Most often `select` runs in a loop so we keep trying

We can put a timeout or “done” channel into the `select`

That gives us a way out

Select

```go
for {
    select {
    case x, ok := <- data:
        if !ok {
            break
        }

        // use the data ...

    case <-time.After(30 * time.Second):
        // safety timeout (one-shot channel)
        return
    }
}
```

If we put a `default` here, we'd busy wait

Two ways to block a goroutine

An empty `select` blocks but consumes no CPU

```
for {}       // blocks by spinning

select {}    // blocks by sleeping
```

Asynchronous Logging Example

How channels solve problems

Example thanks to Bill Kennedy's Behavior of Channels

We'll build a program where the log device may have a problem

When this happens, we don't want the program to hang . . .

Logger with an issue

```go
type device struct {
    problem bool
}

func (d *device) Write(p []byte) (int, error) {
    for d.problem {
        time.Sleep(time.Second)
    }

    return fmt.Println(string(p))
}

func main() {
    var d device
    var l log.Logger

    l.SetOutput(&d)
    . . .
```

Logger with an issue

```
    . . .
    for i := 0; i < 10; i++ {
        go func(id int) {
            for {
                l.Println(fmt.Sprintf("%d: log data", id))
                time.Sleep(100 * time.Millisecond)
            }
        }(i)
    }

    sigChan := make(chan os.Signal, 1)
    signal.Notify(sigChan, os.Interrupt)

    for {
        <- sigChan
        d.problem = !d.problem
    }
}
```

Custom logger

```go
type Logger struct {
    ch chan string
    wg sync.WaitGroup
}

func (l *Logger) Stop() {
    close(l.ch)
    l.wg.Wait()
}

func (l *Logger) Println(s string) {
    select {
    l.ch <- s + "\n":
        // do nothing
    default:
        fmt.Println("DROP")
    }
}
```

Custom logger

```go
func New(w io.Writer, cap int) *Logger {
    l := Logger{
        ch: make(chan string, cap)
    }

    l.wg.Add(1)

    go func() {
        for v := range l.ch {
            fmt.Fprintf(w, v)
        }

        l.wg.Done()
    }()

    return &l
}
```

Custom logger

```go
func main() {
    d := device{}
    l := New(&d, 10)

    // same code as before; spin off 10 goroutines
    // and accept ctrl-C to create a problem
    . . .
}
```

If we hit `Ctrl-C` now, the buffer fills and we drop logs ...

Buffered channels and a `select` allowed us to avoid deadlock

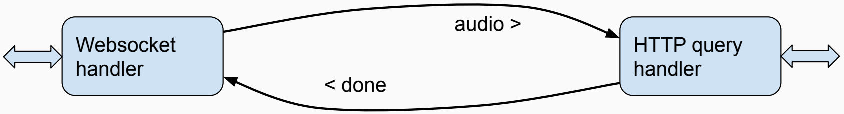
Audio Forwarding Example

Audio forwarding

In this example, we read audio from a websocket and forward it to an HTTP connection

We need to stop forwarding once we've received the correct indication back from the web server, or detected end-of-speech

We need a new goroutine, a couple of channels, and a `select`



Audio forwarding

```go
func (h *houndASR) AddAudio(audio []byte) error {
    if h.eos { return types.ErrEndOfSpeech }
    if h.done == nil { // start up on first chunk
        h.done = make(chan bool, 1)
        h.input = make(chan []byte, 32)
        go h.runQuery(&sender{ch: h.input})
    }

    h.input <- audio

    select {
    case done, ok := <-h.done:
        if !ok || done { h.eos = true }
    default:
        if h.tracker.DetectedEOS(audio) { h.eos = true }
    }

    if h.eos { close(h.input) }; return nil
}
```

Audio forwarding

```go
func (h *houndASR) runQuery(sdr *sender) {
    req, _ := http.NewRequest("POST", h.voiceURL, sdr)
    resp, _ := h.client.Do(req)
    reader := bufio.NewReader(resp.Body)
    defer resp.Body.Close()

    for {
        bytes, _ := reader.ReadBytes('\n') // chunked resp
        data := transcript{}

        if err := json.Unmarshal(bytes), &data); err != nil {
            break
        }
        if data.Done {
            h.done <- true
        }
        . . .
    }
}
```

Audio forwarding

```go
type sender struct {
    ch chan []byte
}

func (s *sender) Read(p []byte) (n int, err error) {
    v, ok := <-s.ch  // blocking read

    if !ok {
        return 0, io.EOF
    }

    // assume for now it fits
    return copy(p, v), nil
}

func (s *sender) Close() error {
    return nil
}
```

Conventional Synchronization

Conventional synchronization

Package `sync` with `Mutex`, `WaitGroup`, etc.

Package `sync/atomic` for atomic scalar reads & writes

We saw a use of wait groups in the 2nd example above

Mutexes must be locked & unlocked to give protection

Mutexes block a goroutine when locked by another

Mutexes in action

```go
type safeMap struct {
    sync.Mutex          // not safe to copy
    m map[string]int
}

// so methods must take a pointer, not a value
func (s *safeMap) incr(key string) {
    s.Lock()
    defer s.Unlock()

    // only one goroutine can execute this
    // code at the same time, guaranteed
    s.m[key]++
}
```

Using `defer` is a good habit, avoids mistakes

Atomic primitives

```go
import "sync"

func do() int {
    var n int64
    var w sync.WaitGroup

    for i := 0; i < 1000; i++ {
        w.Add(1)

        go func() {
            n++                             // DATA RACE
            w.Done()
        }()
    }

    w.Wait()
    return n
}
```

Atomic primitives

```go
import ( "sync"; "sync/atomic" )

func do() int {
    var n int64
    var w sync.WaitGroup

    for i := 0; i < 1000; i++ {
        w.Add(1)

        go func() {
            atomic.AddInt64(&n, 1)       // fixed
            w.Done()
        }()
    }

    w.Wait()
    return n
}
```

Some Gotchas

Concurrency problems

#1: race conditions, where unprotected writes overlap

- must be some data that is written to
- could be a read-modify-write operation
- and two goroutines can do it at the same time

#2: deadlock, when no goroutine can make progress

- goroutines could all be blocked on empty channels
- goroutines could all be blocked waiting on a mutex
- GC could be prevented from running (busy loop)

Go detects most deadlocks; with `-race` it can find data races

Concurrency problems

#3: goroutine leak

- goroutine hangs on a empty or blocked channel
- not deadlock: other goroutines make progress
- often found by looking at pprof output

When you start a goroutine, **always know how/when it will end**

#4: other errors

- misuse of `Mutex`
- misuse of `WaitGroup`
- misuse of `select`

Gotchas 1: Deadlock

A goroutine is **preemptible** only when it starts a (non-inlined) function call, blocks on a channel or mutex, or makes a blocking system call

Potential Problems

If your goroutine isn't preemptible, garbage collection will never run, because it must first "stop the world"

However, if the `main()` function exits, all goroutines terminate

Gotchas 1: Deadlock example

```go
go func() {
    var i byte

    for i = 0; i <= 255; i++ {
        // infinite loop does nothing
        // doesn't get elided
        // and can't be preempted
    }
}()

runtime.Gosched()    // yield execution
runtime.GC()         // force GC

// DEADLOCK

fmt.Println("Done")  // never happens
```

Gotchas 1: Deadlock fixed

```go
go func() {
    var i byte

    for i = 0; i <= 255; i++ {
        // infinite loop does nothing
        // doesn't get elided

        // but we can yield
        if (i == 123) {
            runtime.Gosched()
        }
    }
}()

runtime.Gosched()    // yield execution
runtime.GC()         // force GC
fmt.Println("Done")  // prints Done
```

Gotchas 2: Closure capture

A closure shouldn't capture a **mutating** variable, e.g. a loop index

If it does, it will get the wrong value!

```
for i := 0; i < 10; i++ {   // WRONG
    go func() {
        fmt.Println(i)
    }()
}
```

Instead, **pass the variable's value as a parameter**

```
for i := 0; i < 10; i++ {  // RIGHT
    go func(i int) {
        fmt.Println(i)
    }(i)
}
```

Gotchas 3: Goroutine leak

In this example, a timeout leaves the goroutine hanging forever

```
func finishReq(timeout time.Duration) *obj {
    ch := make(chan obj)

    go func() {
        ch <- fn() // blocking send
    }()

    select {
    case rslt := <-ch:
        return rslt
    case <-time.After(timeout):
        return nil
    }
}
```

The correct solution is to make a buffered channel

Gotchas 4: Incorrect use of WaitGroup

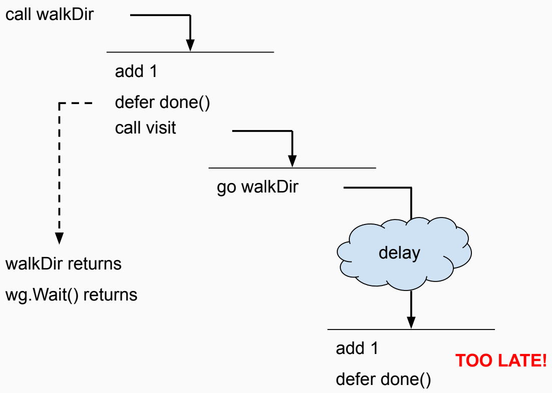
Always, always, always call `Add` before `go` or `Wait`

```
func walkDir(dir string, pairs chan<- pair, ...) {
    wg.Add(1)                                                  // WRONG
    defer wg.Done()

    visit := func(p string, fi os.FileInfo, ...) {
        if fi.Mode().IsDir() && p != dir {
            go walkDir(p, pairs, wg, limits)
        . . .
}

err := walkDir(dir, paths, wg)
wg.Wait()
```

Adding too late may cause `Wait` to return too soon

Gotchas 4: Incorrect use of WaitGroup

Adding inside `walkDir` (and not before) may have a timing problem



Gotchas 4: Incorrect use of WaitGroup

Always, always, always call `Add` before `go` or `Wait`

```
func walkDir(dir string, pairs chan<- pair, ...) {
    defer wg.Done()

    visit := func(p string, fi os.FileInfo, ...) {
        if fi.Mode().IsDir() && p != dir {
            wg.Add(1)                                   // RIGHT
            go walkDir(p, pairs, wg, limits)
        . . .
}

wg.Add(1)                                               // RIGHT
err := walkDir(dir, paths, wg)
wg.Wait()
```

Adding too late may cause `Wait` to return too soon

Some bad code for your delectation

```go
func do() {
    var wg sync.WaitGroup
    sem := make(chan int, 4)  // 4 simultaneous workers

    for i := 0; i < 10; i++ {
        wg.Add(1); sem <- 1

        go func() {
            defer func() {
                wg.Done(); <- sem
            }()

            // some work
        }()
    }

    wg.Wait(); close(sem)
}
```

Some bad code for your delectation

1. the order of `wg.Done` and `<-sem` is confusing
2. read from a closed channel is safe, but `close` isn't needed
3. we acquire the semaphore too soon, limiting the number of goroutines (not the number of simultaneous workers)

```go
for i := 0; i < 10; i++ {
    wg.Add(1)

    go func() {
        defer wg.Done()

        sem <- 1
        // some work
        <- sem
    }()
}
```

Select problems

`select` can be challenging and lead to mistakes:

- `default` is always active
- a `nil` channel is always ignored
- a full channel (for send) is skipped over
- available channels are selected at random
- the “done” channel is just another channel

Anatomy of a select mistake: #1

Mistake #1: skipping a full channel to default and losing a message

```
for {
    x := socket.Read()

    select {
    case output <- x:
        . . .

    default:
        return
    }
}
```

The code was written assuming we'd skip `output` only if it was set to `nil` (as no longer needed)

We also skip if `output` is full, and lose this and future messages

Anatomy of a select mistake: #2

Mistake #2: reading a “done” channel and aborting when input is backed up on another channel — that input is lost

```
for {
    select {
    case x := <- input:
        . . .

    case <- done:
        return
    }
}
```

There’s no guarantee we read all of `input` before reading `done`

Better: use `done` only for an error abort; close `input` on EOF

Some thoughts

Three considerations when using concurrency:

1. Don’t start a goroutine without knowing how it will stop
2. Acquire locks/semaphores as late as possible; release them in the reverse order as soon as possible (and without fail)
3. Don’t wait for non-parallel work that you could do yourself

```
func do() int {
    ch := make(chan int)

    go func() { ch <- 1 }()

    return <-ch
}
```

4. **Simplify!** Make everything clear and easy to read!

Channel state reference

State	Receive	Send	Close
Nil	Block*	Block*	Panic
Empty	Block	Write	Close
Partly Full	Read	Write	Readable until empty
Full	Read	Block	
Closed	Default Value**	Panic	
Receive-only	OK	Compile Error	
Send-only	Compile Error	OK	

* `select` ignores a nil channel since it would always block

** Reading a closed channel returns `(<default-value>, !ok)`

Homework

Homework #5

Take the program you wrote for Homework #4, and write another program using goroutines that drives it with multiple readers & writers until it breaks due to the race conditions that were left unsolved.

You will want to run the server with Go's `-race` option to identify the failures. You may wish to write your driver program in the form of a unit test. More info on the race detector is on the Go site.

(Re-use your HTTP read logic from Homework #3. Feel free to extend your server & client to use JSON instead of URL parameters if you need more challenge.)

Then add mutexes to the database so that it is protected against race conditions and no longer fails with concurrent updates.