Programming in Go

Matt Holiday
Christmas 2020

Concurrency Gotchas

Concurrency problems

#1: race conditions, where unprotected read & 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 *some* deadlocks automatically; with `-race` it can find *some* 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: channel errors

- trying to send on a closed channel
- trying to send or receive on a nil channel
- closing a nil channel
- closing a channel twice

Concurrency problems

#5: other errors

- closure capture
- misuse of `Mutex`
- misuse of `WaitGroup`
- misuse of `select`

A good taxonomy of Go concurrency errors may be found in this paper:
https://cseweb.ucsd.edu/~yiying/GoStudy-ASPLOS19.pdf

Many of the errors are basic & should easily be found by review;
maybe we'll get static analysis tools to help find them

Gotchas 1: Data race

We've already seen this, but here it is again

```
var nextID = 0

func handler(w http.ResponseWriter, r *http.Request) {
    fmt.Fprintf(w, "<h1>You got %v<h1>", nextID)

    // unsafe - data race
    nextID++
}

func main() {
    http.HandleFunc("/", handler)
    if err := http.ListenAndServe(":8080", nil); err != nil {
        log.Fatal(err)
    }
}
```

Gotchas 2: Deadlock

Go can usually detect when no goroutine is able to make progress; here the main goroutine is blocked on a channel it can never read

```
func main() {
    ch := make(chan bool)

    go func(ok bool) {
        fmt.Println("STARTED")

        if ok {
            ch <- ok
        }
    }(false)

    <-ch
    fmt.Println("DONE")
}
```

Gotchas 2: Deadlock

Locking a mutex and then failing to unlock it afterwards;
the fix is to use defer at the point of locking

```
var m sync.Mutex
done := make(chan bool)

go func() {
    m.Lock()  // not unlocked!
}()

go func() {
    time.Sleep(1)
    m.Lock()
    defer m.Unlock()
    done <- true
}()
<-done
```

Gotchas 2: Deadlock

Locking mutexes in the wrong order will often result in deadlock;
the fix is **always** to lock them in the same order everywhere

```
var m1, m2 sync.Mutex
done := make(chan bool)
go func() {
    m1.Lock(); defer m1.Unlock()
    time.Sleep(1)
    m2.Lock(); defer m2.Unlock()
    done <- true
}()
go func() {
    m2.Lock(); defer m2.Unlock()
    time.Sleep(1)
    m1.Lock(); defer m1.Unlock()
    done <- true
}()
<-done; <-done
```

Gotchas 3: Goroutine leak

In this example, a timeout leaves the goroutine hanging forever; the correct solution is to make a buffered channel

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

    go func() {
        . . .      // work that takes too long
        ch <- fn() // blocking send
    }()

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

Gotchas 4: Incorrect use of WaitGroup

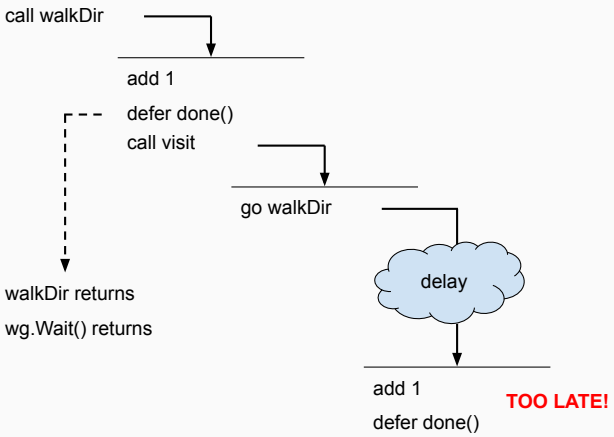
Always, always, always call `Add` before `go` or `Wait`

```
func walkDir(dir string, pairs chan<- pair, ...) {
    wg.Add(1)                                          // BIG MISTEAK
    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* the goroutine may fail (too late)



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()
```

Gotchas 5: Closure capture

A goroutine closure shouldn't capture a **mutating** variable

```
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)
}
```

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
- a “done” channel is *just another channel*
- **available channels are selected at random**

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`

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

Four 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
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!** **Review!** **Test!**