Concurrency Patterns

26 May 2015

Aaron Schlesinger
Sr. Engineer, Iron.io

About me

- Database & backend systems engineer at Iron.io
- Writing Go for ~2 yrs, JVM for a while during/before that
- Distributed systems at Zynga, StackMob, PayPal, now Iron.io
- C -> C++ -> Python/PHP -> Java -> Scala -> Go

"It's almost too easy to write concurrency bugs in Go"

Because Go has powerful concurrency primitives.

I'm discussing today how to use them responsibly.

Today

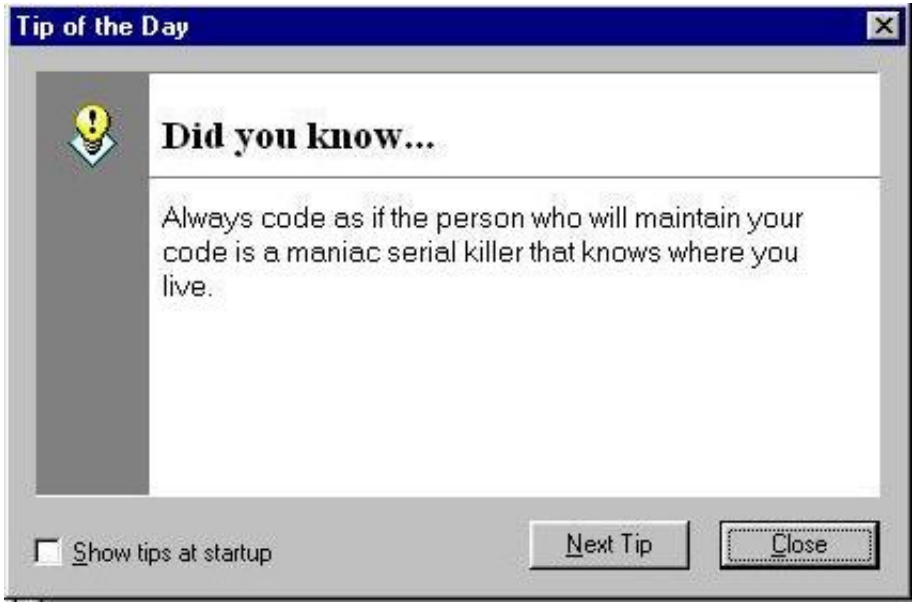
Conventions & patterns for:

- `sync.Mutex`, chan and `sync.WaitGroup`
- Timeouts (http://blog.golang.org/go-concurrency-patterns-timing-out-and), cancellation and net.Context (http://godoc.org/golang.org/x/net/context)
- For-select loops

Something old

Locks have a time and a place. We found a few at Iron.io (building a distributed DB).

If you use locks:



But prefer sharing memory by communicating.

Go-specific conventions

- `defer` unlocking wherever possible
- Document ownership in your **public** interface
- Abstract mutual exclusion if it crosses an exported `func`

Worth paying attention to conventions from other communities (e.g. lock acquisition order).

Documenting Mutual Exclusion

```
package main

import "errors"

var ErrNotReserved = errors.New("not reserved")

// Reserver is a map[string]interface{} where each key/value pair
// can be reserved by callers for mutually exclusive read-modify-write
// operations.
type Reserver interface {
    // GetAndReserve waits for the key to be unreserved, then reserves it for mutual exclusion.
    // On success, returns the current state and reservation ID. Use the latter in
    // future calls that require a reservation ID. If a non-nil error is returned, no
    // reservation is established and the returned value and reservation ID are invalid.
    GetAndReserve(key string) (val interface{}, reservationID string, err error)

    // SetReserved sets the value of the given key if reservationID is valid
    // and points to the current reservation. After this call returns successfully,
    // the caller doesn't have ownership of key and reservationID is invalid.
    // Returns ErrNotReserved if the reservation ID is invalid or not the current reservation.
    // On any non-nil error, neither the value nor the current reservation are changed.
    SetReserved(key, reservationID string, value interface{}) error
}
```

Channels

```
Share memory by communicating.
```

- Channels + goroutines are "easy" but powerful enough to build real systems
- They're built in for a reason. Use them by default
- When in doubt, ask why you *shouldn't* use them to communicate between goroutines

Conventions for Channels

- Document sends and receives across `func` boundaries: who and how?
- Enlist the compiler. Use directional channels
- Don't return a chan unless the `func` is a generator (https://talks.golang.org/2012/concurrency.slide#25)
- `close` is a useful signal to callers. Use it and document it

Example

```
package main

import "sync"

// WatchChanges will watch the state of the given request. ch will send after
// each request state change and will be closed after the request is removed from
// the request state database. Sends on ch from the same goroutine as the caller.
//
// Returns ErrNotFound if the request is not reserved at call time.
// WatchChanges will do no operations on ch if any non-nil error is returned.
func WatchChanges(reqID string, ch chan<- int) error

// WatchAll watches for all events on the given request.
//
// The WaitGroup will be done after the request is reserved, and the channel
// will send on each state change, then be closed when the request is released.
//
// The channel will send from a new, internal goroutine, which you are not responsible
// for managing.
func WatchAll(reqID string) (*sync.WaitGroup, <-chan int)
```

On Documentation

- Documentation may establish contracts or invariants that code can't or won't
- Code should be as self-documenting as possible, but don't let Godoc be empty
- The remainder of this talk has mostly runnable code

WaitGroup

If you're waiting for a chan to close or receive a `struct{}`, can you use a `sync.WaitGroup` instead?

Use these as:

- Notifiers for one-time events
- Rendezvous points
- Helpers to write deterministic tests

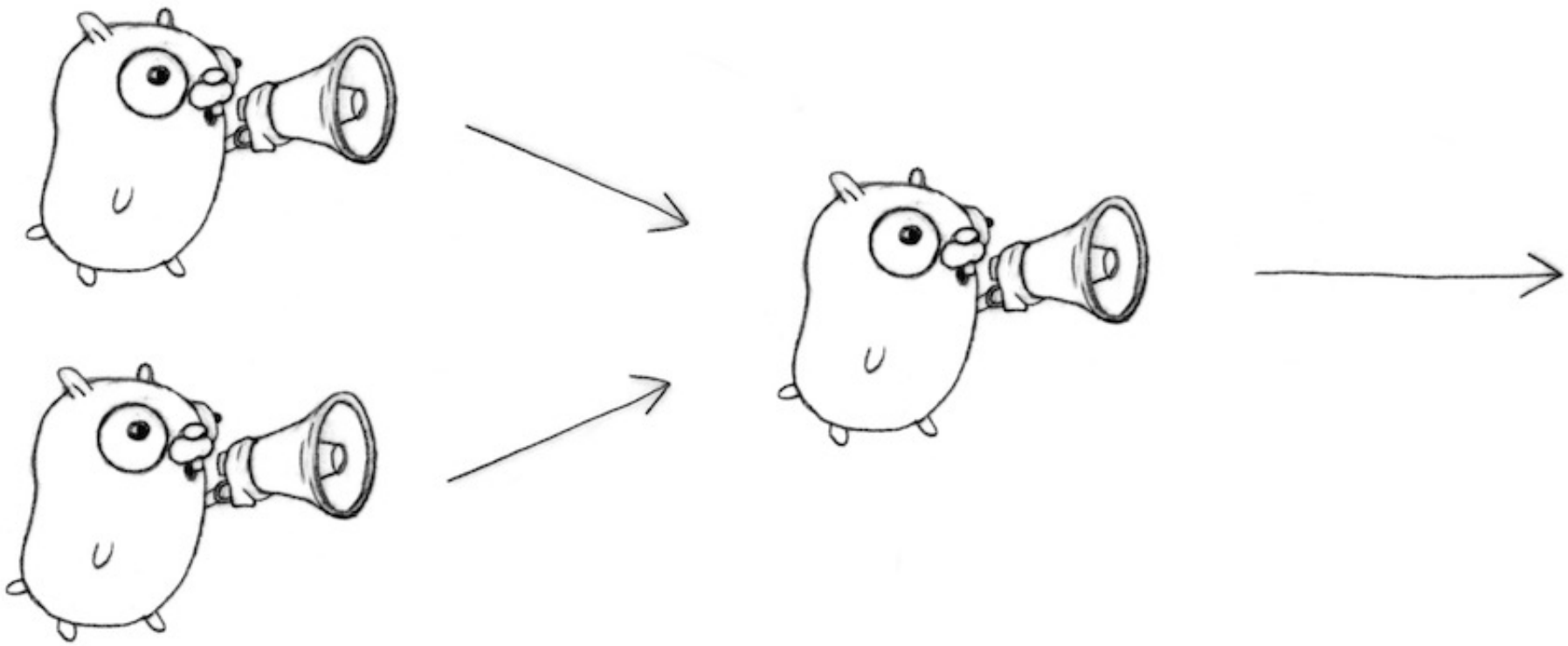
Notification of an event

```
package main

import "sync"

// startLoop starts a loop in a goroutine. the returned WaitGroup is done
// after the first loop iteration has started
func startLoop(n int) *sync.WaitGroup {
    var wg sync.WaitGroup
    wg.Add(1)
    go func() {
        first := true
        for {
            // do some work here
            if first {
                wg.Done()
                first = false
            }
        }
    }()
    return &wg
}
```

Revisiting fan-in



(Taken from https://talks.golang.org/2012/concurrency.slide#28 (https://talks.golang.org/2012/concurrency.slide#28), credit Renée French)

Why?

If you can do work concurrently, do it. There's no excuse not to.

How I'm defining fan-in:

- A code pattern to gather results from 2 or more goroutines
- An algorithm to follow for converting sequential code to concurrent

Details

- Read about it under "Fan-out, fan-in" section at https://blog.golang.org/pipelines (https://blog.golang.org/pipelines)
- `sync.WaitGroup` and a few channels make fan-in simple & understandable
- In many cases, you can get an easy latency win without changing an exported `func`

Sequential datastore queries

```
func GetAll() []int {
    ints := make([]int, 10)
    for i := 0; i < 10; i++ {
        ints[i] = datastoreGet() // sleeps for <= 1sec, then returns a random int
    }
    return ints
}
```

Run

Concurrent datastore queries

```
func GetAll() []int {
    wg, ch := getWGAndChan(10) // get a waitgroup that has 10 added to it, and a chan int
    for i := 0; i < 10; i++ {
        c := make(chan int)
        go datastoreGet(c)  // sends an int on c then closes after sleeping <= 1 sec
        go func() {
            defer wg.Done() // mark this iteration done after receiving on c
            ch <- <-c       // enhancement: range of c if >1 results
        }()
    }
    go func() {
        wg.Wait() // wait for all datastoreGets to finish
        close(ch) // then close the main channel
    }()
    ints := make([]int, 10)
    i := 0
    for res := range ch { // stream all results from each datastoreGet into the slice
        ints[i] = res // GetAll can be a generator if you're willing to change API.
        i++           // that lets you push results back to the caller.
    }
    return ints
}
```

Run

Production issues

Specifically, issues in long-running systems.

- They will happen
- Test for them
- Even if you can't, be proactive and try to fail gracefully

Timeouts

Your system shouldn't grind to a halt on channel sends.

```
Aarons-MacBook-Pro:Desktop arschles$ go run asleep.go
fatal error: all goroutines are asleep - deadlock!

goroutine 1 [chan receive]:
main.main()
        /Users/arschles/Desktop/asleep.go:12 +0x10b

goroutine 5 [chan receive]:
main.func·001()
        /Users/arschles/Desktop/asleep.go:9 +0x69
created by main.main
        /Users/arschles/Desktop/asleep.go:10 +0xc1
exit status 2
```

Select on the channel and a timer. Or, do better...

Use net.Context

`net.Context` (https://blog.golang.org/context) has a nice interface in front of timers.

- I'm cheating here. I said everything would be done with the standard library
- Contexts are more than a timer. They provide a nice interface with some extras
- You could build and test your own context in a few hours, from the stdlib
- That's my excuse

Context interface

With (excellent) comments taken out for brevity.

```
import "time"

type Context interface {
    Deadline() (deadline time.Time, ok bool)

    Done() <-chan struct{}

    Err() error

    Value(key interface{}) interface{}
}
```

(Copyright (c) 2009 The Go Authors. All rights reserved. Please see the full license (https://github.com/golang/net/blob/master/LICENSE) for more.)

Using contexts

- They add cancellation
- They build a tree of control

`net.Context` is a good universal tool for timeouts/cancellation in a large codebase.

Contexts in a distributed system

The Tail at Scale.

- Jeff Dean talk/paper. I originally saw it at a Ricon 2013 Talk (https://www.youtube.com/watch?v=C_PxVdQmfpk)
- Hedged requests: do a few identical GET (e.g. no side effects) requests, cancel remaining requests after first returns

Rob showed a variant in https://talks.golang.org/2012/concurrency.slide#50

(https://talks.golang.org/2012/concurrency.slide#50)

Adding cancellation

```
func main() {
    ch := make(chan int)
    ctx, cancel := context.WithTimeout(context.Background(), 10*time.Millisecond)
    defer cancel()
    for i := 0; i < 10; i++ {
        // get sleeps for a random duration <= 100ms,
        // then sends a random int on ch. stops if ctx.Done() receives.
        go get(ctx, ch)
    }
    select {
    case i := <-ch:
        fmt.Printf("got result %d\n", i)
    case <-ctx.Done():
        fmt.Println("got no result")
    }
}
```

Run

That was the naïve implementation

But, it's not too hard to get "fancy"

- Don't send 2nd request until 1st is past 95th percentile expected latency (2 contexts - one cancel)
- Cancel in-flight requests (pass context to RPC subsystem)
- Target-target communication (pass info on other in-flight requests over RPC)

Putting it all together

For-select loops put together almost all of the concepts in here.

Possible applications:

- Event loops
- GC
- Sequential state mutation (like an actor)

For-select loop mechanics

- Run a (possibly infinite) loop in a goroutine
- Generally select on 2+ channels in each iteration
- Sometimes pass long running operations to other goroutines

Patterns

- Ack before and after real work is done. testing is easier and rate limiting/backpressure is easy
- If you're ticking, wrap `time.Ticker` to add ack
- `net.Context` for cancellation
- `sync.WaitGroup` for started and stopped

A for-select poller

```
func poll(ctx context.Context) (*sync.WaitGroup, *sync.WaitGroup, <-chan string) {
    var start, end sync.WaitGroup // start & end notifications to multiple parties
    start.Add(1)
    end.Add(1)
    ch := make(chan string)
    go func() {
        defer close(ch)
        defer end.Done()
        start.Done()
        for {
            time.Sleep(5 * time.Millisecond)
            select {
            case <-ctx.Done():
                return
            case ch <- "element " + strconv.Itoa(rand.Int()):
            }
        }
    }()
    return &start, &end, ch
}
```

Driving the poller

```
func main() {
    ctx, cancel := context.WithTimeout(context.Background(), 10*time.Millisecond)
    defer cancel()
    mainCh, wg := makeThings(10) // make a chan string and a wg that has 10 added to it
    for i := 0; i < 10; i++ {
        start, _, ch := poll(ctx)
        start.Wait()
        go func() {
            defer wg.Done()
            for str := range ch {
                mainCh <- str
            }
        }()
    }
    go func() {
        wg.Wait()
        close(mainCh)
    }()
    printCh(mainCh) // loops on mainCh until it's closed
}
```

Run

Notes

- The poller is missing the ack
- We have a small utility at Iron.io to add acks to `time.Ticker` and `time.Timer`
- Exercise left to the reader

Conclusion

Go has really good built in concurrency primitives.

I believe we (the community) are starting to build good patterns & tools to *responsibly* build on them.

If you take one thing away

Use `net.Context` in your codebase.

Or, at least try it.

It's simple and powerful, and follows the "Go way."

If you take two things away

Add reading and understanding Go Concurrency Patterns (https://talks.golang.org/2012/concurrency.slide).

Thank you

Aaron Schlesinger
Sr. Engineer, Iron.io
aaron@iron.io (mailto:aaron@iron.io)
http://github.com/arschles (http://github.com/arschles)
@arschles (http://twitter.com/arschles)