



CLOUDFLARE™

Go Concurrency

March 27, 2013

John Graham-Cumming

Fundamentals

- goroutines
 - Very lightweight processes
 - All scheduling handled internally by the Go runtime
 - Unless you are CPU bound you do not have to think about scheduling
- Channel-based communication
 - The right way for goroutines to talk to each other
- Synchronization Primitives
 - For when a channel is too heavyweight
 - Not covered in this talk

goroutines

- “Lightweight”
 - Starting 10,000 goroutines on my MacBook Pro took 22ms
 - Allocated memory increased by 3,014,000 bytes (301 bytes per goroutine)
 - <https://gist.github.com/jgrahamc/5253020>
- Not unusual at CloudFlare to have a single Go program running 10,000s of goroutines with 1,000,000s of goroutines created during life program.
- So, go `yourFunc ()` as much as you like.

Channels

- Quick syntax review

`c := make(chan bool)` – Makes an unbuffered channel of bools

`c <- x` – Sends a value on the channel

`<- c` – Waits to receive a value on the channel

`x = <- c` – Waits to receive a value and stores it in x

`x, ok = <- c` – Waits to receive a value; ok will be false if channel is closed and empty.

Unbuffered channels are best

- They provide both communication and synchronization

```
func from(connection chan int) {
    connection <- rand.Intn(100)
}

func to(connection chan int) {
    i := <- connection
    fmt.Printf("Someone sent me %d\n", i)
}

func main() {
    cpus := runtime.NumCPU()
    runtime.GOMAXPROCS(cpus)

    connection := make(chan int)
    go from(connection)
    go to(connection)
}
```

Using channels for signaling (1)

- Sometimes just closing a channel is enough

```
c := make(chan bool)

go func() {
    // ... do some stuff
    close(c)
}()

// ... do some other stuff
<- c
```

Using channels for signaling (2)

- Close a channel to coordinate multiple goroutines

```
func worker(start chan bool) {  
    <- start  
    // ... do stuff  
}  
  
func main() {  
    start := make(chan bool)  
  
    for i := 0; i < 100; i++ {  
        go worker(start)  
    }  
  
    close(start)  
  
    // ... all workers running now  
}
```

Select

- Select statement enables sending/receiving on multiple channels at once

```
select {  
  case x := <- somechan:  
    // ... do stuff with x  
  
  case y, ok := <- someOtherchan:  
    // ... do stuff with y  
    // check ok to see if someOtherChan  
    // is closed  
  
  case outputChan <- z:  
    // ... ok z was sent  
  
  default:  
    // ... no one wants to communicate  
}
```


Common idiom: for/select

```
for {
  select {
    case x := <- somechan:
      // ... do stuff with x

    case y, ok := <- someOtherchan:
      // ... do stuff with y
      // check ok to see if someOtherChan
      // is closed

    case outputChan <- z:
      // ... ok z was sent

    default:
      // ... no one wants to communicate
  }
}
```

Using channels for signaling (4)

- Close a channel to terminate multiple goroutines

```
func worker(die chan bool) {
    for {
        select {
            // ... do stuff cases
            case <- die:
                return
        }
    }
}

func main() {
    die := make(chan bool)
    for i := 0; i < 100; i++ {
        go worker(die)
    }
    close(die)
}
```

Using channels for signaling (5)

- Terminate a goroutine and verify termination

```
func worker(die chan bool) {
    for {
        select {
            // ... do stuff cases
            case <- die:
                // ... do termination tasks
                die <- true
                return
        }
    }
}

func main() {
    die := make(chan bool)
    go worker(die)
    die <- true
    <- die
}
```

Example: unique ID service

- Just receive from `id` to get a unique ID
- Safe to share `id` channel across routines

```
id := make(chan string)

go func() {
    var counter int64 = 0
    for {
        id <- fmt.Sprintf("%x", counter)
        counter += 1
    }
}()

x := <- id // x will be 1
x = <- id  // x will be 2
```

Example: memory recycler

```
func recycler(give, get chan []byte) {
    q := new(list.List)

    for {
        if q.Len() == 0 {
            q.PushFront(make([]byte, 100))
        }

        e := q.Front()

        select {
        case s := <-give:
            q.PushFront(s[:0])

        case get <- e.Value.([]byte):
            q.Remove(e)
        }
    }
}
```

Timeout

```
func worker(start chan bool) {  
    for {  
        timeout := time.After(30 * time.Second)  
        select {  
            // ... do some stuff  
  
            case <- timeout:  
                return  
        }  
    }  
}
```

```
func worker(start chan bool) {  
    timeout := time.After(30 * time.Second)  
    for {  
        select {  
            // ... do some stuff  
  
            case <- timeout:  
                return  
        }  
    }  
}
```

Heartbeat

```
func worker(start chan bool) {  
    heartbeat := time.Tick(30 * time.Second)  
    for {  
        select {  
            // ... do some stuff  
  
            case <- heartbeat:  
                // ... do heartbeat stuff  
        }  
    }  
}
```

Example: network multiplexor

- Multiple goroutines can send on the same channel

```
func worker(messages chan string) {
    for {
        var msg string // ... generate a message
        messages <- msg
    }
}

func main() {
    messages := make(chan string)
    conn, _ := net.Dial("tcp", "example.com")

    for i := 0; i < 100; i++ {
        go worker(messages)
    }
    for {
        msg := <- messages
        conn.Write([]byte(msg))
    }
}
```


Example: first of N

- Dispatch requests and get back the first one to complete

```
type response struct {
    resp *http.Response
    url string
}

func get(url string, r chan response) {
    if resp, err := http.Get(url); err == nil {
        r <- response{resp, url}
    }
}

func main() {
    first := make(chan response)
    for _, url := range []string{"http://code.jquery.com/jquery-1.9.1.min.js",
        "http://cdnjs.cloudflare.com/ajax/libs/jquery/1.9.1/jquery.min.js",
        "http://ajax.googleapis.com/ajax/libs/jquery/1.9.1/jquery.min.js",
        "http://ajax.aspnetcdn.com/ajax/jQuery/jquery-1.9.1.min.js"} {
        go get(url, first)
    }

    r := <- first
    // ... do something
}
```

range

- Can be used to consume all values from a channel

```
func generator(strings chan string) {
    strings <- "Five hour's New York jet lag"
    strings <- "and Cayce Pollard wakes in Camden Town"
    strings <- "to the dire and ever-decreasing circles"
    strings <- "of disrupted circadian rhythm."
    close(strings)
}

func main() {
    strings := make(chan string)
    go generator(strings)

    for s := range strings {
        fmt.Printf("%s ", s)
    }
    fmt.Printf("\n");
}
```

Passing a 'response' channel

```
type work struct {
    url string
    resp chan *http.Response
}

func getter(w chan work) {
    for {
        do := <- w
        resp, _ := http.Get(do.url)
        do.resp <- resp
    }
}

func main() {
    w := make(chan work)

    go getter(w)

    resp := make(chan *http.Response)
    w <- work{"http://cdnjs.cloudflare.com/jquery/1.9.1/jquery.min.js",
        resp}

    r := <- resp
}
```

Buffered channels

- Can be useful to create queues
- But make reasoning about concurrency more difficult

```
c := make(chan bool, 100)
```

Example: an HTTP load balancer

- Limited number of HTTP clients can make requests for URLs
- Unlimited number of goroutines need to request URLs and get responses
- Solution: an HTTP request load balancer

A URL getter

```
type job struct {  
    url string  
    resp chan *http.Response  
}  
  
type worker struct {  
    jobs chan *job  
    count int  
}  
  
func (w *worker) getter(done chan *worker) {  
    for {  
        j := <- w.jobs  
        resp, _ := http.Get(j.url)  
        j.resp <- resp  
        done <- w  
    }  
}
```

A way to get URLs

```
func get(jobs chan *job, url string, answer chan string) {
    resp := make(chan *http.Response)
    jobs <- &job{url, resp}
    r := <- resp
    answer <- r.Request.URL.String()
}

func main() {
    jobs := balancer(10, 10)
    answer := make(chan string)
    for {
        var url string
        if _, err := fmt.Scanln(&url); err != nil {
            break
        }
        go get(jobs, url, answer)
    }
    for u := range answer {
        fmt.Printf("%s\n", u)
    }
}
```

A load balancer

```
func balancer(count int, depth int) chan *job {
    jobs := make(chan *job)
    done := make(chan *worker)
    workers := make([]*worker, count)

    for i := 0; i < count; i++ {
        workers[i] = &worker{make(chan *job,
            depth), 0}
        go workers[i].getter(done)
    }

    go func() {
        for {
            var free *worker
            min := depth
            for _, w := range workers {
                if w.count < min {
                    free = w
                    min = w.count
                }
            }

            var jobsource chan *job
            if free != nil {
                jobsource = free.jobs
            }
        }
    }()

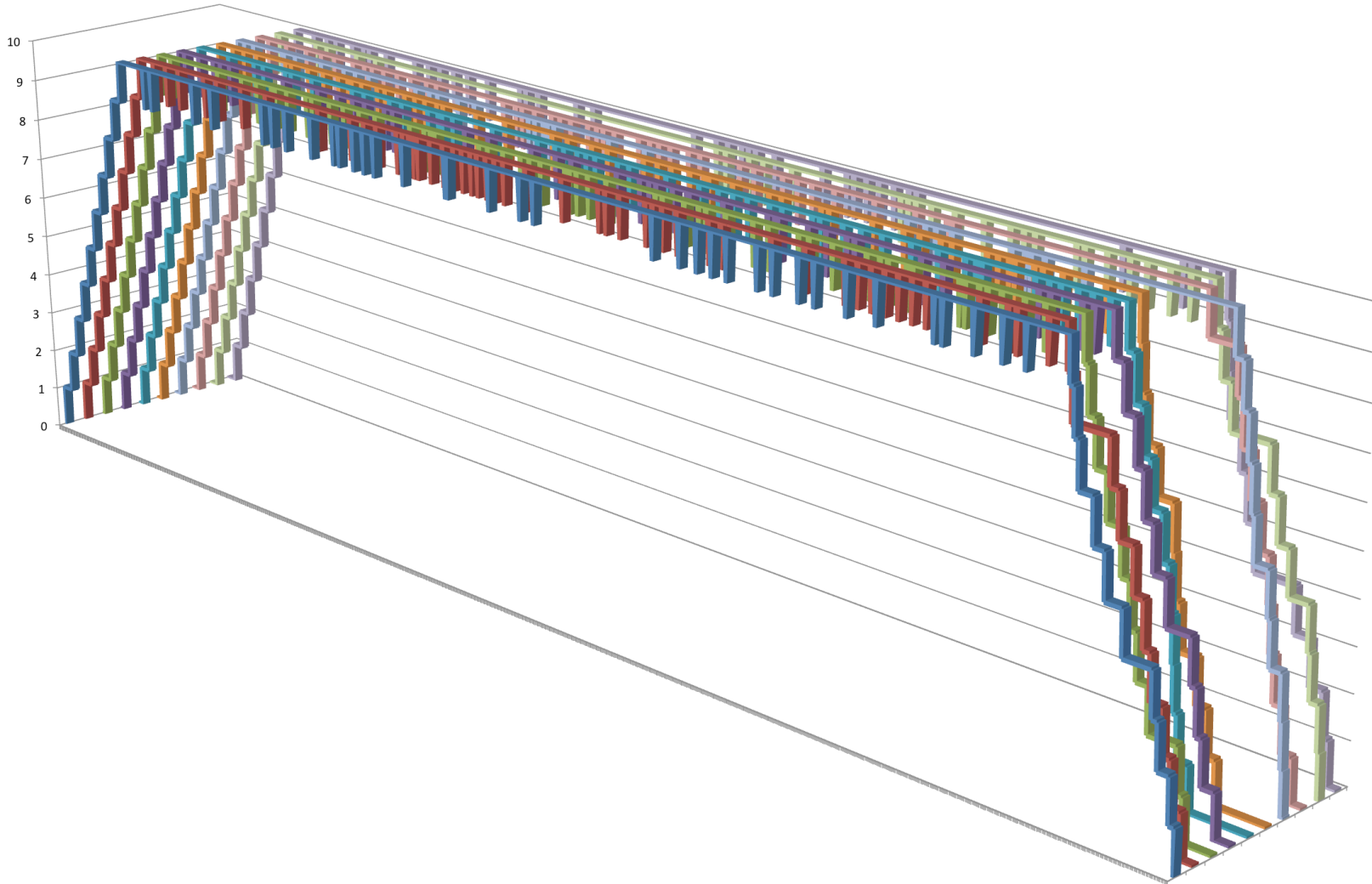
    return jobs
}
```

```
select {
case j := <- jobsource:
    free.jobs <- j
    free.count++

case w := <- done:
    w.count--
}

}()
```


Top 500 web sites loaded



THANKS

The Go Way: “small sequential pieces joined by channels”