Go Concurrency

Programming Models for Emerging Platforms

Concurrency

- Allows developers to **logically** separate co-running program tasks
  - Web server : ?
  - Firefox : ?
  - Google Maps : ?

Concurrency

- Allows developers to **logically** separate co-running program tasks
  - Web server : Each visit to google.com
  - Firefox : Each open tab
  - Google Maps : UI thread, GPS thread, direction lookup thread

Concurrency

- Allows developers to **logically** separate co-running program tasks
  - Web server : Each visit to google.com
  - Firefox : Each open tab
  - Google Maps : UI thread, GPS thread, direction lookup thread
- **Concurrent programming is not new, but we have “new” concurrent paradigms**

We have seen

- Pthreads
  - Mutexes, Semaphores, Condition Variables
- Java Threads
  - synchronized blocks, Locks, Conditions
- Cilk
- ForkJoin
- Message Passing Interface

We have seen

- Most everything so far has had the concurrent programming model “tacked on”
- First look at a model where concurrency was designed along with the language
  - But I will show you how it’s still kind of broken :)

Goroutines

- A function that concurrently executes with other functions
- Multiple goroutines multiplexed onto multiple threads
  - 4 goroutines could share the same 1 physical thread
  - 4 goroutines could share 4 physical threads
- Think Cilk

Goroutines

```
func shout(name string) {
  fmt.Printf("HELLO %s\n", name)
}

func main() {
  go shout("Anthony")

  go func() {
    fmt.Printf("HELLO Michael\n")
  }()

  go func(name string) {
    fmt.Printf("HELLO %s\n", name)
  }("Cassie")

  for { // Prevent program exit
  }
}
```

go “function call” will start a goroutine

Goroutines

```go
func shout(name string) {
  fmt.Printf("HELLO %s\n", name)
}

func main() {
  go shout("Anthony")

  go func() {
    fmt.Printf("HELLO Michael\n")
  }()

  go func(name string) {
    fmt.Printf("HELLO %s\n", name)
  }("Cassie")

  for { // Prevent program exit
  }
}
```

go “function call” will start a goroutine

function literals make the “spawn” convenient (compare with cilk)

Don’t forget to “call” the function

Goroutines

```
func shout(name string) {
  fmt.Printf("HELLO %s\n", name)
}

func main() {
  go shout("Anthony")

  go func() {
    fmt.Printf("HELLO Michael\n")
  }()

  go func(name string) {
    fmt.Printf("HELLO %s\n", name)
  }("Cassie")

  for { // Prevent program exit
  }
}
```

go “function call” will start a goroutine

Remember that function literals can have arguments

Pass “Cassie” as parameter

Goroutines

- Goroutines have no notion of “return”
  - They execute the supplied function. That’s it.
- Goroutines **do** share memory…

Goroutines

```go
var n = 0

func step() {
  for i := 0; i < 10000; i++ {
    n++
  }
  fmt.Printf("Current:%d\n", n)
}

func main() {
  for i := 0; i < 10; i++ {
    go step()
  }
  for {
  }
}
```

Shared package variabled

Race condition

Goroutines

- Example (race.go)

Goroutines

- Goroutines have no notion of “return”
  - They execute the supplied function. That’s it.
- Goroutines **do** share memory
  - go build -race race.go (builds with race detector)
  - **sync** package contains abstractions to managing goroutines in a thread-like fashion
    - mutex, condition, WaitGroups
  - **We will not use the shared memory model**

Go Concurrnecy

- Concurrent programming in many environments is made difficult by the subtleties required to implement correct access to shared variables. Go encourages a different approach in which shared values are passed around on channels and, in fact, never actively shared by separate threads of execution. Only one goroutine has access to the value at any given time. Data races cannot occur, by design. To encourage this way of thinking we have reduced it to a slogan:
  - **Do not communicate by sharing memory; instead, share memory by communicating.**

Go Channels

- **Typed** message passing model
  - `chan int       // pass ints`
  - `chan string    // pass strings`
  - `chan []node    // pass slices of nodes`

Go Channels

- **Typed** message passing model
  - `chan int        // pass ints`
  - `chan string     // pass strings`
  - `chan []node     // pass slices of nodes`
- Unbuffered (blocking) by default
  - `c := make(chan int)`
  - `c := make(chan int, 0)`
- Buffered
  - `c := make(chan int, 100)`

Channel Basics

```go
var c = make(chan int)

func fact(f int) {
  v := 1
  for f > 0 {
    v *= f
    f--
  }
  c <- v
}

func main() {
  go fact(4)
  go fact(5)
  f4 := <-c
  f5 := <-c
  fmt.Printf("f(4) = %d, f(5) = %d\n", f4, f5)
}
```

Use make to init channel

Send v over channel

Receive next value from channel

Channel Basics

```
var c = make(chan int)

func fact(f int) {
  v := 1
  for f > 0 {
    v *= f
    f--
  }
  c <- v
}

func main() {
  go fact(4)
  go fact(5)
  f4 := <-c
  f5 := <-c
  fmt.Printf("f(4) = %d, f(5) = %d\n", f4, f5)
}
```

Safe to share channels among goroutines

What do you expect for output?

Channel Basics

- Channels implement a first-in, first-out (FIFO) queue
- If nothing to receive, reading from channel will block
- Channels are first class values in Go

Channel Basics

- Channels implement a first-in, first-out (FIFO) queue
- If nothing to receive, reading from channel will block
- **Channels are first class values in Go**

Goroutines

- Example (channels.go, gen.go)

Channel Select

- **select** provides a way to multiplex channel communication
  - part of the reason channels are built into the language
  - think of it as a **switch** for channels
- select allows further multiplexing...

Channel Select (timeout)

```
func main() {
  primeC := make(chan int)
  go primeSearcher(1000, primeC)

  select {
  case s := <-primeC:
    fmt.Printf("Found prime %d\n", s)
  case <-time.After(10 * time.Second):
    fmt.Printf("No primes found after
                  10 seconds, exiting\n")
    return
  }
}
```

time.After creates a channel that sends message after delay

Use this as a timeout

Channel Select (nonblocking)

```
func main() {
  primeC := make(chan int)
  go primeSearcher(1000, primeC)

  for {
    select {
    case s := <-primeC:
      fmt.Printf("Found prime %d\n", s)
    default:
      fmt.Printf(“Waiting...”)
      time.Sleep(time.Second)
    }
  }
}
```

If all other cases do not have message available, default is called

Use this for non-blocking communication

Ending Communication

- Channels can be closed with `close(c)`, where `c` is some channel
  - Note that channels do not *need* to be closed, but sometimes is useful for synchronization
- Receivers can receive from and check for a closed channel
- Senders CAN NOT send to a closed channel (panic)

Closing Channel (ok comma)

```
func fibonacci(n int, c chan int) {
    x, y := 0, 1
    for i := 0; i < n; i++ {
        c <- x
        x, y = y, x+y
    }
    close(c)
}

func main() {
    c := make(chan int)
    go fibonacci(20, c)
    for {
        v, ok := <- c
        if !ok {
            return
        }
        fmt.Println(v)
    }
}
```

Close channel to indicate no more values coming

ok command idiom that we've seen several times

v gets zero value, ok gets false if channel is closed

Closing Channel (range)

```
func fibonacci(n int, c chan int) {
    x, y := 0, 1
    for i := 0; i < n; i++ {
        c <- x
        x, y = y, x+y
    }
    close(c)
}

func main() {
    c := make(chan int)
    go fibonacci(20, c)
    for v := range c {
        fmt.Println(v)
    }
}
```

range operation on channel will receive until channel closes

Deadlock

- It’s still possible to deadlock goroutines with channels

Channel Deadlock

```
func main() {
  c1, c2 := make(chan int), make(chan int)
  go func() {
    <-c2
    c1 <- 0
  }()
  <-c1
  c2 <- 0
  fmt.Printf("Done!\n")
}
```

Immediate block on channel receive

What happens on the run in Go? (deadlock.go)

A Case Study (Google)

- Google Search
  - Take a “query”, returns Results
  - Concurrently searches web, images, video etc
  - We can abstract some of the details away and see a realworld use of channels

Google Search 1.0

```
func Google(query string) (results []Result) {
  results = append(results, Web(query))
  results = append(results, Image(query))
  results = append(results, Video(query))
  return
}
```

Google Search 1.0

```
func Google(query string) (results []Result) {
  results = append(results, Web(query))
  results = append(results, Image(query))
  results = append(results, Video(query))
  return
}
```

What issues do you see?

Google Search 1.0

```
func Google(query string) (results []Result) {
  results = append(results, Web(query))
  results = append(results, Image(query))
  results = append(results, Video(query))
  return
}
```

Not concurrent!

Google Search 2.0

```
func Google(query string) (results []Result) {
  c := make(chan Result)
  go func() { c <- Web(query) } ()
  go func() { c <- Image(query) } ()
  go func() { c <- Video(query) } ()
  for i := 0; i < 3; i++ {
    result := <-c
    results = append(results, result)
  }
  return
}
```

What issues do you see?

Google Search 2.0

```
func Google(query string) (results []Result) {
  c := make(chan Result)
  go func() { c <- Web(query) } ()
  go func() { c <- Image(query) } ()
  go func() { c <- Video(query) } ()
  for i := 0; i < 3; i++ {
    result := <-c
    results = append(results, result)
  }
  return
}
```

Slow search slows down
entire search

Google Search 2.1

```
func Google(query string) (results []Result) {
  c := make(chan Result)
  go func() { c <- Web(query) } ()
  go func() { c <- Image(query) } ()
  go func() { c <- Video(query) } ()
  timeout := time.After(80 * time.Millisecond)
  for i := 0; i < 3; i++ {
    select {
    case result := <-c:
      results = append(results, result)
    case <-timeout:
      fmt.Println("timed out")
    return
    }
  }
  return
}
```

Google Search 2.1

```
func Google(query string) (results []Result) {
  c := make(chan Result)
  go func() { c <- Web(query) } ()
  go func() { c <- Image(query) } ()
  go func() { c <- Video(query) } ()
  timeout := time.After(80 * time.Millisecond)
  for i := 0; i < 3; i++ {
    select {
    case result := <-c:
      results = append(results, result)
    case <-timeout:
      fmt.Println("timed out")
    return
    }
  }
  return
}
```

What issues do you see?

Google Search 2.1

```
func Google(query string) (results []Result) {
  c := make(chan Result)
  go func() { c <- Web(query) } ()
  go func() { c <- Image(query) } ()
  go func() { c <- Video(query) } ()
  timeout := time.After(80 * time.Millisecond)
  for i := 0; i < 3; i++ {
    select {
    case result := <-c:
      results = append(results, result)
    case <-timeout:
      fmt.Println("timed out")
    return
    }
  }
  return
}
```

Slow results get discarded

Google Search 3.0

```
func Google(query string) (results []Result) {
  c := make(chan Result)
  go func() { c <- First(query, Web1, Web2) } ()
  go func() { c <- First(query, Image1, Image2) } ()
  go func() { c <- First(query, Video1, Video2) } ()
  timeout := time.After(80 *
    time.Millisecond)
  for i := 0; i < 3; i++ {
    select {
    case result := <-c:
      results = append(results, result)
    case <-timeout:
      fmt.Println("timed out")
    return
    }
  }
  return
}
```

```
func First(query string, replicas ...Search) Result {
  c := make(chan Result)
  searchReplica := func(i int) {
    c <- replicas[i](query)
  }
  for i := range replicas {
    go searchReplica(i)
  }
  return <-c
}
```

Google Search 3.0

```
func Google(query string) (results []Result) {
  c := make(chan Result)
  go func() { c <- First(query, Web1, Web2) } ()
  go func() { c <- First(query, Image1, Image2) } ()
  go func() { c <- First(query, Video1, Video2) } ()
  timeout := time.After(80 *
    time.Millisecond)
  for i := 0; i < 3; i++ {
    select {
    case result := <-c:
      results = append(results, result)
    case <-timeout:
      fmt.Println("timed out")
    return
    }
  }
  return
}
```

```
func First(query string, replicas ...Search) Result {
  c := make(chan Result)
  searchReplica := func(i int) {
    c <- replicas[i](query)
  }
  for i := range replicas {
    go searchReplica(i)
  }
  return <-c
}
```

No locks, no condition variables, no callbacks.

Acknowledgments

- https://golang.org/doc/effective_go.html
- https://golang.org/ref/spec
- https://golangcode.com/sleeping-with-go/
- https://tour.golang.org/concurrency/4

- Rob Pike, "Go Concurrency Patterns'' (Creative Commons Attribution 3.0 License)
- Sameer Ajmani, "Advanced Go Concurrency Patterns"