Concurrency in Go

Share by Communicating

23 April 2014

Frank Roberts
CAD Engineer, Cypress Semiconductor

Before I start, an update:

Cypress Semiconductor

- I'm a CAD Engineer.
- The CAD group maintains the design tool chain and methodology.

As a CAD Engineer, I have to know a little bit of everything.

- Electrical Engineering -> I understand the design flow.
- Computer Science -> I model problems and solutions in software.

Concurrent programming is hard

Common model: Shared memory

- Run multiple routines concurrently operating on shared data
- Must control access to shared state
- Must synchronize actions of concurrent routines

Share by Communicating

Go encourages a model that avoids these problems

- Don't share state between concurrent routines
- Let the communication mechanism handle synchronization

Go's concurrency model reduces to this slogan:

Do not communicate by sharing memory, *share memory by communicating*.

Go provides *goroutines* and *channels* to support this model.

Goroutines

Goroutines enable concurrency

Like threads, but lighter

Spawn one by prefixing a function call with the *go* keyword

Similar to backgrounding a process in Linux

Scheduled onto OS threads by the Go runtime

Goroutines share an address space, but sharing data structures is not idiomatic.

Channels

Channels facilitate communication and synchronization between goroutines.

Think Unix pipes, but type-safe.

Channels may be of any type, including channel.

Channels may be buffered to any level, including 0.

Reads on a channel block until there is a value to receive.

Writes block until the receiver executes a read or there is space in the buffer.

Concurrent Program Design

Think in pipelines

Model programs as directed graphs:

- Goroutines are nodes
- Channels are edges

Do not share state. Sharing state defeats the purpose.

Concurrency != parallelism:

- concurrency: structuring a program as independently executing routines
- parallelism: executing calculations in parallel for efficiency

A Logger

```
func main() {
  logger := startLogger()   // Start a logger instance.
  logger <- "Hello, World"  // Send the string "Hello, World" to the logger.
  close(logger)             // Disallow more sends on channel.
}

func startLogger() (chan string) {
  c := make(chan string) // Create a channel of strings.
  go func () {           // Define the logger as an anonymous function.
    for s := range c {   // Iterate until channel is closed.
      fmt.Println(s);    // Log the message.
    }
  }()                    // Call the logger function (as a goroutine).
  return c               // Return the channel.
}
```

Run

Odd Numbers

```
func main() {
  runtime.GOMAXPROCS(runtime.NumCPU())  // Allow parallelism.
  odds := []int{1,2,3,4,5,6,7,8,9}      // A list of odd(?) numbers.
  done := make(chan int)                // Create a channel to gather goroutines.
  for _, n := range odds {
    go checkOdd(n, done)                // Report whether n is odd.
  }

  for i := 0; i < len(odds); i++ {      // Wait on other goroutines to exit.
    <-done
  }
  fmt.Println("All done.")
}

func checkOdd(n int, done chan int) {
  if n % 2 == 0 {
    fmt.Printf("%d is not odd.\n", n)
  } else {
    fmt.Printf("%d is odd.\n", n)
  }
  done <- 1                             // Signal completion
}
```

Run

Logging Odd Numbers

```
func main() {
  runtime.GOMAXPROCS(runtime.NumCPU())  // Allow parallelism.
  odds := []int{1,2,3,4,5,6,7,8,9}      // A list of odd(?) numbers.
  done := make(chan int)                // Create a channel to gather goroutines.
  logger := startLogger()               // Start a logger.
  for _, n := range odds {
    go checkOdd(n, logger, done)        // Report whether n is odd.
  }

  for i := 0; i < len(odds); i++ {      // Wait on other goroutines to exit.
    <-done
  }
  fmt.Println("All done.")
  close(logger)
}

func checkOdd(n int, logger chan string, done chan int) {
  if n % 2 == 0 {
    logger <- fmt.Sprintf("%d is not odd.", n)
  } else {
    logger <- fmt.Sprintf("%d is odd.", n)
  }
  done <- 1                             // Signal Completion
}
```

Run

Select

Wait for operations on a set of channels.

Semantically similar to select() system call.

Syntactically similar to a switch statement.

Evaluate all channel operations, choose one to proceed.

Simple Select Example

```
func wait_rand() (chan int) {
    delay := time.Duration(rand.Int() % 10)
    c := make(chan int)
    go func() {
        <-time.After(delay * time.Second)
        c <- int(delay)
        close(c)
    }()
    return c
}

func handleChannel(proc []chan int, open bool, i, delay int) (int) {
    finished := 0
    if open {
        fmt.Printf("proc[%d] waited %d seconds\n", i, delay)
    } else {
        finished = 1
        proc[i] = nil
    }
    return finished
}
```

Simple Select Example

```
func main() {
    rand.Seed(time.Now().UnixNano())
    proc := []chan int{wait_rand(), wait_rand(), wait_rand()}

    finished := 0
    for i := 0; finished < len(proc); i++ {
        select {
            case delay, open := <-proc[0]:
                finished += handleChannel(proc, open, 0, delay)
            case delay, open := <-proc[1]:
                finished += handleChannel(proc, open, 1, delay)
            case delay, open := <-proc[2]:
                finished += handleChannel(proc, open, 2, delay)
            case <-time.After(time.Second):
                fmt.Printf("Select iteration %d\n", i)
        }
    }
}
```

Run

My Go Concurrency Projects

A NodeScape client

- Spawns a new Goroutine for each property that it measures
- Re-reads its configuration file at a configurable interval
- Configurable, per-property communication intervals

A distributed checksum calculator

- Server splits space into regions
- Server spawns a goroutine to manage each region
- Region managers re-issue the region until it's successfully computed
- Client program computes multiple regions in parallel

References

Go Homepage

www.golang.org/ (http://www.golang.org/)

Go Playground

www.golang.org/play (http://www.golang.org/play)

Questions?

23 April 2014
Tags: Go Concurrency (#ZgotmplZ)

Frank Roberts
CAD Engineer, Cypress Semiconductor
frank@jafrro.net (mailto:frank@jafrro.net)
http://www.jafrro.net (http://www.jafrro.net)