Programming in Go

Matt Holiday
Christmas 2020

Select

Select

`select` allows any “ready” alternative to proceed among

- a channel we can read from
- a channel we can write to
- a `default` action that’s always ready

Most often `select` runs in a loop so we keep trying

We can put a timeout or “done” channel into the `select`

- ♥ We can compose channels as synchronization primitives!
- ♠ Traditional primitives (mutex, condition variable) can’t be composed

Select example: read two channels

```
func main() {
    chans := []chan int{ make(chan int), make(chan int) }
    for i := range chans {
        go func(i int, ch chan int) {
            for {
                time.Sleep(time.Duration(i) * time.Second); ch <- i
            }
        }(i+1, chans[i])
    }
    for i := 0; i < 12; i++ {
        select {
        case m0 := <-chans[0]:
            fmt.Println("received", m0)
        case m1 := <-chans[1]:
            fmt.Println("received", m1)
        }
    }
}
```

Select example: read two channels

```
2009/11/10 23:00:01 received 1
2009/11/10 23:00:02 received 2
2009/11/10 23:00:02 received 1
2009/11/10 23:00:03 received 1
2009/11/10 23:00:04 received 2
2009/11/10 23:00:04 received 1
2009/11/10 23:00:05 received 1
2009/11/10 23:00:06 received 1
2009/11/10 23:00:06 received 2
2009/11/10 23:00:07 received 1
2009/11/10 23:00:08 received 1
2009/11/10 23:00:08 received 2
```

Simple example with timeout

```
func main() {
    stopper := time.After(5 * time.Second)
    . . .

    for _, url := range list {
        go get(url, results)      // start a CSP process
    }


    for range list {              // read from the channel
        select {
        case r := <-results:
            log.Printf("%-20s %s\n", r.url, r.latency)
        case <-stopper:
            log.Fatal("timeout")
        }
    }
}
```

Select with a periodic timer

```
const tickRate = 2 * time.Second

func main() {
    log.Println("start")
    ticker := time.NewTicker(tickRate).C  // periodic
    stopper := time.After(5 * tickRate)   // one shot

loop:
    for {
        select {
        case <-ticker:
            log.Println("tick")
        case <-stopper:
            break loop
        }
    }
    log.Println("finish")
}
```

Select with a periodic timer

```
2009/11/10 23:00:00 start
2009/11/10 23:00:02 tick
2009/11/10 23:00:04 tick
2009/11/10 23:00:06 tick
2009/11/10 23:00:08 tick
2009/11/10 23:00:10 tick
2009/11/10 23:00:10 finish
```

Select example: default

In a `select` block, the `default` case is always ready and will be chosen if no other case is:

```
func sendOrDrop(data []byte) {
    select {
    case ch <- data;
        // sent ok; do nothing

    default:
        log.Printf("overflow: drop %d bytes", len(data))
    }
}
```

Don't use `default` inside a loop — the `select` will busy wait and waste CPU

Select example: running a subprocess

```
func start(name, args ...string) error {
    cmd := exec.Command(name, args...)
    stderr, _ := cmd.StderrPipe()        // ignoring errors
    _ := cmd.Start()                     // ditto :-)
    done := make(chan string)

    go scrape(stderr, done)

    select {
    case result := <-done:
        if result == "running" {
            return nil
        }
        return fmt.Errorf("failed: %s", result)
    case <-time.After(20 * time.Second):
        return fmt.Errorf("timeout")
    }
}
```

Select example: running a subprocess

```
func scrape(in io.ReadCloser, done ch<- string) {
    defer in.Close()
    defer close(done)

    for {
        d := bufferedRead(in)   // details omitted

        if strings.Contains(d, "Exception") {
            done <- strings.TrimSuffix(d, "\n")
            return
        }

        if strings.Contains(d, "server is now running") {
            done <- "running"
            return
        }
    }
}
```