Concurrency in Go

Aditya Thakur

ECS 140A Programming Languages – Winter 2019

Goroutines

- Syntactically, a `go` statement is ordinary function or method call prefixed with the `go` keyword.
- A `go` statement causes a function to be called in a newly created `goroutine.`
- A `goroutine` is similar to a thread.

```
f()  // call f(); wait for it to return
go f()  // create a new goroutine that calls f(); don't wait
```

- https://play.golang.org/p/f0sLl05L4f6

WaitGroup

- A WaitGroup waits for a collection of goroutines to finish.
- The main goroutine calls Add to set the number of goroutines to wait for.
- Then each of the goroutines runs and calls Done when finished.
- At the same time, Wait can be used to block until all goroutines have finished.

https://golang.org/pkg/sync/#WaitGroup

https://play.golang.org/p/D9E-hYs_rui

https://play.golang.org/p/YAkNNB3wA1A

Channels

Channels are the pipes that connect concurrent goroutines. You can send values into channels from one goroutine and receive those values into another goroutine.

```
ch := make(chan int) // ch has type ‘chan int’
ch <- x // send statement
X = <-ch  // receive expression in assignment
<-ch // receive statement; result is discarded
```

Closing Channels

- To close a channel `ch`, call the `close` function
  ```
  close(ch)
  ```
- Sending on a closed channel will lead to a panic
- Receives on a closed channel yield the values that have been sent until no more values are left.
- Any receive operations thereafter complete immediately and yield the zero value of the channels' element type.

Closing Channels

- Receivers can test whether a channel has been closed by assigning a second parameter to the receive expression: after

```
v, ok := <-ch
```

`ok` is false if there are no more values to receive and the channel is closed.

Channels

Channels can be unbuffered or buffered

```
ch = make(chan int) // unbuffered channel
ch = make(chan int, 0) // unbuffered channel
ch = make(chan int, 4) // buffered channel, capacity 4
```

Unbuffered Channels

Sends and receives block until the other side is ready.

This allows goroutines to synchronize without explicit locks or condition variables.

https://play.golang.org/p/bRGMAqinovA

https://play.golang.org/p/lSM5G7RRYbv

https://play.golang.org/p/WjY-i7UY-Pp

Buffered Channels

- A buffered channel has a queue of elements
  - Send inserts at the back of the queue, receive removes an element from the front
- If the channel is full, the send operation blocks until space is made available by a receive
- If a channel is empty, the receive operation blocks until a value is sent by another go routine

- https://play.golang.org/p/4ej1ZbvdKPs

Range over channels

- The loop `for i := range c` receives values from the channel repeatedly until it is closed.

- https://play.golang.org/p/SrsWBdMWJHJ

Select

- The `select` statement lets a goroutine wait on multiple communication operations.

https://play.golang.org/p/VoqPiRaTR-X

https://play.golang.org/p/ZdSOPe1Gj13

Default in Select

- The default case in a select is run if no other case is ready.

https://play.golang.org/p/KgpsNnkdjZo

Timeouts with select

https://play.golang.org/p/MgcfA-xpJO9

Timers

- https://play.golang.org/p/pybl9hRvJq2

Sync.Mutex

https://play.golang.org/p/Z_Td6Kn_hMT

More code

https://github.com/adonovan/gopl.io/tree/master/ch9