



Goroutines and Networking

This lesson flashes back to the standard operations and their syntaxes defined on goroutines and networking.

WE'LL COVER THE FOLLOWING



-  Useful code snippets for channels
 - Creation
 - Looping over a channel with a `for-range`
 - Testing if a channel is closed
 - Using a channel to let the main program wait
 - Channel factory pattern
 - Stopping a goroutine
 - Simple timeout pattern
 - Using an in- and out-channel instead of locking
-  Useful code snippets for networking applications
 - Templating

A rule of thumb if you use *parallelism* to gain efficiency over serial computation:

"The amount of work done inside goroutines has to be much larger than the costs associated with creating goroutines and sending data back and forth between them."

- **Using buffered channels for performance:** A buffered channel can easily double its throughput depending on the context, and the performance gain can be 10x or more. You can further try to optimize by adjusting the capacity of the channel.
- **Limiting the number of items in a channel and packing them in arrays:** Channels become a bottleneck if you pass a lot of individual items through them. You can work around this by packing chunks of data into

arrays and then unpacking on the other end. This can give a speed gain of a factor 10x.



Useful code snippets for channels

Creation

```
ch := make(chan type, buf)
```

Looping over a channel with a `for-range`

```
for v := range ch {  
    // do something with v  
}
```

Testing if a channel is closed

```
//read channel until it closes or error-condition  
for {  
    if input, open := <-ch; !open {  
        break  
    }  
    fmt.Printf("%s ", input)  
}
```

Or, use the *looping over a channel method* where the detection is automatic.

Using a channel to let the main program wait

Using the semaphore pattern, you can let the main program wait until the goroutine completes:

```
ch := make(chan int) // Allocate a channel  
  
// Start something in a goroutine; when it completes, signal on the channel  
1  
go func() {  
    // doSomething  
    ch <- 1 // Send a signal; value does not matter  
}()  
doSomethingElseForAWhile()  
<-ch // Wait for goroutine to finish; discard sent value
```

Channel factory pattern

The function is a channel factory, and it starts a lambda function as goroutine, populating the channel:

```
func pump() chan int {
    ch := make(chan int)
    go func() {
        for i := 0; ; i++ {
            ch <- i
        }
    }()
    return ch
}
```

Stopping a goroutine


```
runtime.Goexit()
```

Simple timeout pattern

```
timeout := make(chan bool, 1)
go func() {
    time.Sleep(1e9) // one second
    timeout <- true
}()
select {
    case <-ch:
        // a read from ch has occurred
    case <-timeout:
        // the read from ch has timed out
}
```

Using an in- and out-channel instead of locking

```
func Worker(in, out chan *Task) {
    for {
        t := <-in
        process(t)
        out <- t
    }
}
```

 Useful code snippets for networking applications #

Templating #

Templating

- Make, parse and validate a template:

```
var strTempl = template.Must(template.New("TName").Parse(strTemplateHTML))
```

- Use the html filter to escape HTML special characters, when used in a web context:

```
{{html .}} or with a field fieldName {{ .FieldName |html }}
```

- Use template-caching.

This pretty much summarizes goroutines and networking. The suggestions you learned in the previous chapters are now summarized in the next lesson for you.