Part 24: Select

10 AUGUST 2017

Welcome to tutorial no. 24 in Golang tutorial series.

What is *select*?

The `select` statement is used to choose from multiple send/receive channel operations. The select statement blocks until one of the send/receive operation is ready. If multiple operations are ready, one of them is chosen at random. The syntax is similar to `switch` except that each of the case statement will be a channel operation. Lets dive right into some code for better understanding.

Example

```
1  package main
2
3  import (
4      "fmt"
5      "time"
6  )
7
8  func server1(ch chan string) {
9      time.Sleep(6 * time.Second)
10     ch <- "from server1"
11 }
12 func server2(ch chan string) {
13     time.Sleep(3 * time.Second)
14     ch <- "from server2"
15
16 }
17 func main() {
18     output1 := make(chan string)
19     output2 := make(chan string)
20     go server1(output1)
21     go server2(output2)
22     select {
23     case s1 := <-output1:
24         fmt.Println(s1)
25     case s2 := <-output2:
26         fmt.Println(s2)
27     }
28 }
```

Run in playground

In the program above, the `server1` function in line no. 8 sleeps for 6 seconds then writes the text *from server1* to the channel `ch`. The `server2` function in line no. 12 sleeps for 3 seconds and then writes *from server2* to the channel `ch`.

The main function calls the go Goroutines `server1` and `server2` in line nos. 20 and 21 respectively.

In line no. 22, the control reaches the `select` statement. The `select` statement blocks until one of its cases is ready. In our program above, the `server1` Goroutine writes to the `output1` channel after 6 seconds whereas the `server2` writes to the `output2` channel after 3 seconds. So the select statement will block for 3 seconds and will wait for `server2` Goroutine to write to the `output2` channel. After 3 seconds, the program prints,

```
from server2
```

and then will terminate.

Practical use of select

The reason behind naming the functions in the above program as `server1` and `server2` is to illustrate the practical use of *select*.

Lets assume we have a mission critical application and we need to return the output to the user as quickly as possible. The database for this application is replicated and stored in different servers across the world. Assume that the functions `server1` and `server2` are in fact communicating with 2 such servers. The response time of each server is dependant on the load on each and the network delay. We send the request to both the servers and then wait on the corresponding channels for the response using the `select` statement. The server which responds first is chosen by the select and the other response is ignored. This way we can send the same request to multiple servers and return the quickest response to the user :).

Default case

The default case in a `select` statement is executed when none of the other case is ready. This is generally used to prevent the select statement from blocking.

```
1  package main
2
3  import (
4      "fmt"
5      "time"
6  )
7
8  func process(ch chan string) {
9      time.Sleep(10500 * time.Millisecond)
10     ch <- "process successful"
11 }
12
13 func main() {
14     ch := make(chan string)
15     go process(ch)
16     for {
17         time.Sleep(1000 * time.Millisecond)
18         select {
19         case v := <-ch:
20             fmt.Println("received value: ", v)
21             return
22         default:
23             fmt.Println("no value received")
24         }
25     }
26
27 }
```

Run in playground

In the program above, the `process` function in line no. 8 sleeps for 10500 milliseconds (10.5 seconds) and then writes `process successful` to the `ch` channel. This function is called concurrently in line no. 15 of the program.

After calling the `process` Goroutine concurrently, an infinite for loop is started in the main Goroutine. The infinite loop sleeps for 1000 milliseconds (1 second) during the start of each iteration and them performs a select operation. During the first 10500 milliseconds, the first case of the select statement namely `case v := <-ch:` will not be ready since the `process` Goroutine will write to the `ch` channel only after 10500 milliseconds. Hence the `default` case will be executed during this time and the program will print `no value received` 10 times.

After 10.5 seconds, the `process` Goroutine writes `process successful` to `ch` in line no. 10. Now the first case of the select statement will be executed and the program will print `received value:  process successful` and then it will terminate. This program will output,

```
no value received
no value received
no value received
no value received
no value received
no value received
no value received
no value received
no value received
no value received
received value:  process successful
```

Deadlock and default case

```
1  package main
2
3  func main() {
4      ch := make(chan string)
5      select {
6      case <-ch:
7      }
8  }
```

Run in playground

In the program above, we have created a channel `ch` in line no. 4. We try to read from this channel inside the select in line no. 6. The select statement will block forever since no other Goroutine is writing to this channel and hence will result in deadlock. This program will panic at runtime with the following message,

```
fatal error: all goroutines are asleep - deadlock!

goroutine 1 [chan receive]:
main.main()
    /tmp/sandbox416567824/main.go:6 +0x80
```

If a default case is present, this deadlock will not happen since the default case will be executed when no other case is ready. The program above is rewritten with a default case below.

```
package main

import "fmt"

func main() {
    ch := make(chan string)
    select {
    case <-ch:
    default:
        fmt.Println("default case executed")
    }
}
```

Run in playground

The above program will print,

```
default case executed
```

Similarly the default case will be executed even if the select has only nil channels.

```
1  package main
2
3  import "fmt"
4
5  func main() {
6      var ch chan string
7      select {
8      case v := <-ch:
9          fmt.Println("received value", v)
10     default:
11         fmt.Println("default case executed")
12
13     }
14 }
```

Run in playground

In the program above `ch` is `nil` and we are trying to read from `ch` in the select in line no. 8. If the `default` case was not present , the `select` would have blocked forever and caused a deadlock. Since we have a default case inside the select, it will be executed and the program will print,

```
default case executed
```

Random selection

When multiple cases in a `select` statement are ready, one of them will be executed at random.

```
1  package main
2
3  import (
4      "fmt"
5      "time"
6  )
7
8  func server1(ch chan string) {
9      ch <- "from server1"
10 }
11 func server2(ch chan string) {
12     ch <- "from server2"
13
14 }
15 func main() {
16     output1 := make(chan string)
17     output2 := make(chan string)
18     go server1(output1)
19     go server2(output2)
20     time.Sleep(1 * time.Second)
21     select {
22     case s1 := <-output1:
23         fmt.Println(s1)
24     case s2 := <-output2:
25         fmt.Println(s2)
26     }
27 }
```

Run in playground

In the program above, the `server1` and `server2` go routines are called in line no. 18 and 19 respectively. Then the main program sleeps for 1 second in line no. 20. When the control reaches the `select` statement in line no. 21, `server1` would have written `from server1` to the `output1` channel and `server2` would have written `from server2` to the `output2` channel and hence both the cases of the select statement are ready to be executed. If you run this program multiple times, the output will vary between `from server1` or `from server2` depending on which case is chosen in random.

Please run this program in your local system to get this randomness. If this program is run in the playground it will print the same output since the playground is deterministic.

Gotcha - Empty select

```
package main

func main() {
    select {}
}
```

Run in playground

What do you think will be the output of the program above?

We know that the select statement will block until one of its cases is executed. In this case the select statement doesn't have any cases and hence it will block forever resulting in a deadlock. This program will panic with the following output,

```
fatal error: all goroutines are asleep - deadlock!

goroutine 1 [select (no cases)]:
main.main()
    /tmp/sandbox299546399/main.go:4 +0x20
```

This brings us to an end of this tutorial. Have a good day.

Next tutorial - Mutex

Naveen Ramanathan

iOS developer at dietco.de and Golang enthusiast.