Goroutines ~ Range over Channels

최정운(coearth)
황태현(mandu)

Goroutines

A goroutine is a function that is capable of running concurrently with other functions.

```go
package main

import "fmt"

func f(from string) {
    for i := 0; i < 3; i++ {
        fmt.Println(from, ":", i)
    }
}

func main() {

    f("direct")


    go f("goroutine")



    go func(msg string) {
        fmt.Println(msg)
    }("going")

    var input string
    fmt.Scanln(&input)
    fmt.Println("done")
}
```

```
$ go run goroutines.go
direct : 0
direct : 1
direct : 2
goroutine : 0
going
goroutine : 1
goroutine : 2
<enter>
done
```

Goroutines

```go
package main

import "fmt"

func f(from string) {
    for i := 0; i < 3; i++ {
        fmt.Println(from, ":", i)
    }
}

func main() {

    f("direct")


    go f("goroutine")



    go func(msg string) {
        fmt.Println(msg)
    }("going")
}
```

Main function과 f라는 function이 동시에 실행되는데, main function이 먼저 끝나서 다른 goroutine들은 출력되지 않는다.

```
$ go run goroutines.go
direct : 0
direct : 1
direct : 2
```

Channels

```
package main

import (
    "fmt"
    "time"
)

func pinger(c chan string) {
    for i := 0; ; i++ {
        c <- "ping"
    }
}
func printer(c chan string) {
    for {
        msg := <- c
        fmt.Println(msg)
        time.Sleep(time.Second * 1)
    }
}
func main() {
    var c chan string = make(chan string)

    go pinger(c)
    go printer(c)

    var input string
    fmt.Scanln(&input)
}
```

Channels provide a way for two goroutines to communicate with one another.

선언 c := make(chan variable_type)

사용 c <- value(sending)
variable <- c(receiving)

ping
ping
ping

...

(enter)

Channels

```go
package main

import "fmt"

func sum(a []int, c chan int) {
    sum := 0
    for _, v := range a {
        sum += v
    }
    c <- sum // send sum to c
}

func main() {
    a := []int{7, 2, 8, -9, 4, 0}

    c := make(chan int)
    go sum(a[:len(a)/2], c)
    go sum(a[len(a)/2:], c)
    x, y := <-c, <-c // receive from c

    fmt.Println(x, y, x+y)
}
```

17 -5 12

```go
package main

import "fmt"

func sum(a []int, c chan int) {
    sum := 0
    for _, v := range a {
        sum += v
    }
    c <- sum // send sum to c
}

func main() {
    a := []int{7, 2, 8, -9, 4, 0}

    c := make(chan int)
    go sum(a[:len(a)/2], c)
    //go sum(a[len(a)/2:], c)
    x, y := <-c, <-c // receive from c

    fmt.Println(x, y, x+y)
}
```

Fatal error: all goroutines are asleep - deadlock!

Channel Buffering

A buffered channel is asynchronous ; sending or receiving a message will not wait unless the channel is already full.

```
package main

import "fmt"

func main() {

    messages := make(chan string, 2)

    messages <- "buffered"
    messages <- "channel"


    fmt.Println(<-messages)
    fmt.Println(<-messages)
}
```

선언 c := make(chan variable_type, size)

buffered
channel

Channel Synchronization

```go
package main

import "fmt"
import "time"

func worker(done chan bool) {
    fmt.Print("working...")
    time.Sleep(time.Second)
    fmt.Println("done")

    done <- true
}

func main() {

    done := make(chan bool, 1)
    go worker(done)

    <-done
}
```

채널의 특성을 이용해서 main과 worker라는 2개의 goroutine을 synchronize한 예이다.

```
$ go run channel-synchronization.go
working...done
```

Channel Directions

```go
package main

import "fmt"

func ping(pings chan<- string, msg string) {
    pings <- msg
}

func pong(pings <-chan string, pongs chan<- string) {
    msg := <-pings
    pongs <- msg
}

func main() {
    pings := make(chan string, 1)
    pongs := make(chan string, 1)
    ping(pings, "passed message")
    pong(pings, pongs)
    fmt.Println(<-pongs)
}
```

함수 argument 선언시

chan<- variable_type(only sending)

<-chan variable_type(only receiving)

```
$ go run channel-directions.go
passed message
```

Select

Go has a special statement called **select** which works like a **switch** but for channels.

```go
package main

import "time"
import "fmt"

func main() {

    c1 := make(chan string)
    c2 := make(chan string)

    go func() {
        time.Sleep(time.Second * 1)
        c1 <- "one"
    }()
    go func() {
        time.Sleep(time.Second * 2)
        c2 <- "two"
    }()

    for i := 0; i < 2; i++ {
        select {
        case msg1 := <-c1:
            fmt.Println("received", msg1)
        case msg2 := <-c2:
            fmt.Println("received", msg2)
        }
    }
}
```

사용

```
select {
case value := <- channel1:
        …
case value := <- channel2:
        …
case <-time.After(time):
        …
default:
        …
}
```

```
$ time go run select.go
received one
received two

real    0m2.245s
```

Timeouts

```go
package main

import "time"
import "fmt"

func main() {

    c1 := make(chan string, 1)
    go func() {
        time.Sleep(time.Second * 2)
        c1 <- "result 1"
    }()

    select {
    case res := <-c1:
        fmt.Println(res)
    case <-time.After(time.Second * 1):
        fmt.Println("timeout 1")
    }

    c2 := make(chan string, 1)
    go func() {
        time.Sleep(time.Second * 2)
        c2 <- "result 2"
    }()
    select {
    case res := <-c2:
        fmt.Println(res)
    case <-time.After(time.Second * 3):
        fmt.Println("timeout 2")
    }
}
```

Select에서 channel을 기다리는 시간을 정할 수 있다.

```
$ go run timeouts.go
timeout 1
result 2
```

Non-Blocking Channel Operations

```go
package main

import "fmt"

func main() {
    messages := make(chan string)
    signals := make(chan bool)

    select {
    case msg := <-messages:
        fmt.Println("received message", msg)
    default:
        fmt.Println("no message received")
    }

    msg := "hi"
    select {
    case messages <- msg:
        fmt.Println("sent message", msg)
    default:
        fmt.Println("no message sent")
    }

    select {
    case msg := <-messages:
        fmt.Println("received message", msg)
    case sig := <-signals:
        fmt.Println("received signal", sig)
    default:
        fmt.Println("no activity")
    }
}
```

기본적인 select에서는 channel 값을 기다리는데, default를 이용하면 channel 값을 기다리지 않는다.

```
$ go run non-blocking-channel-operations.go
no message received
no message sent
no activity
```

Closing Channels

```go
package main

import "fmt"

func main() {
    jobs := make(chan int, 5)
    done := make(chan bool)



    go func() {
        for {
            j, more := <-jobs
            if more {
                fmt.Println("received job", j)
            } else {
                fmt.Println("received all jobs")
                done <- true
                return
            }
        }
    }()

    for j := 1; j <= 3; j++ {
        jobs <- j
        fmt.Println("sent job", j)
    }
    close(jobs)
    fmt.Println("sent all jobs")

    <-done
}
```

사용 v, ok := <-ch

v에는 channel에 sending된 값이 들어간다.

ok에는 channel이 open되어있으면 true, close되어있으면 false가 들어간다.

```
$ go run closing-channels.go
sent job 1
received job 1
sent job 2
received job 2
sent job 3
received job 3
sent all jobs
received all jobs
```

Range over Channels

```go
package main

import (
    "fmt"
)

func fibonacci(n int, c chan int) {
    x, y := 0, 1
    for i := 0; i < n; i++ {
        c <- x
        x, y = y, x+y
    }
    close(c)
}

func main() {
    c := make(chan int, 10)
    go fibonacci(cap(c), c)
    for i := range c {
        fmt.Println(i)
    }
}
```

range를 이용하면 close되기 전까지를 cover할 수 있다.

```
0
1
1
2
3
5
8
13
21
34
```