핵심! Go 프로그래밍

- 2일차

목차

고루틴

- 프로세스(Process)
  - 명령어, 사용자 데이터, 시스템 영역, 실행 과정에서 수집한 다양한 종류의 리소스로 구성된 독립적 실행 단위
- 프로그램(Program)
  - 프로세스의 명령어와 사용자 데이터를 초기화하는데 사용할 명령어와 데이터를 담은 파일
- 스레드(thread)
  - 프로그램이나 프로세서보다 좀 더 가볍고 작은 실행단위
  - 프로세스에 의해 생성되며 독립적 제어 흐름과 스택을 가짐
  - 프로세스의 일부분

고루틴

- Go 루틴(goroutine)
  - Go 프로그램에서 동시에 실행할 수 있는 최소 단위
  - OS에서 관리하는게 아닌 go runtime에서 관리
  - 프로세스 > 스레드 > goroutine
  - 특징
    - 굉장히 가벼움(4k) -> 보통 m 단위
    - 이론상 몇 만개를 사용해도 가능
    - 크기가 동적으로 증가
    - 채널을 활용한 동기화

고루틴

```go
package main

import (
    "fmt"
    "time"
)

func test(){
    fmt.Println( a...: "hi")
}
func main() {
    go test()
    time.Sleep( d: 10)
}
```

```
hi
```

```go
package main

import (
    "fmt"
    "time"
)

func test(num int){
    fmt.Print( a...: "goroutine Num : ", num,  " ")
}
func main() {
    for i := 0 ; i < 10 ; i++{
        go test(i)
    }
    time.Sleep( d: 1000000000)
}
```

```
<4 go setup calls>
goroutine Num : 9 goroutine Num : 6 goroutine Num : 7 goroutine Num : 8 goroutine Num : 2 goroutine Num : 3 goroutine Num : 1 goroutine Num : 0 goroutine Num : 4 goroutine Num : 5
Process finished with exit code 0
```

고루틴

```go
package main

import (
    "fmt"
    "runtime"
    "time"
)

func test(num int){
    fmt.Print( a...: "goroutine Num : ", num,  " ")
}
func main() {
    fmt.Println(runtime.GOMAXPROCS( n: 0))
    runtime.GOMAXPROCS(runtime.NumCPU())
    fmt.Println(runtime.GOMAXPROCS( n: 0))
    for i := 0 ; i < 10 ; i++{
        go test(i)
    }
    time.Sleep( d: 1000000000)
}
```

```
20
20
goroutine Num : 9 goroutine Num : 1 goroutine Num : 2 goroutine Num : 3 goroutine Nu
```

고루틴

```go
package main

import (
    "fmt"
    "time"
)

func main() {
    for i := 0 ; i < 10 ; i++{
        go func() {
            fmt.Println( a...: "closer : ", i)
        }()
    }
    time.Sleep( d: 1000000000)
}
```

```
<4 go setup calls>
closer :  10
closer :  10
closer :  10
closer :  10
closer :  10
closer :  10
closer :  10
closer :  10
closer :  10
closer :  10
```

```go
package main

import (
    "fmt"
    "time"
)

func main() {
    for i := 0 ; i < 10 ; i++{
        go func() {
            fmt.Println( a...: "closer : ", i)
        }()
        time.Sleep( d: 1)
    }
    time.Sleep( d: 1000000000)
}
```

```
<4 go setup calls>
closer :  0
closer :  1
closer :  2
closer :  3
closer :  4
closer :  5
closer :  6
closer :  7
closer :  8
closer :  9
```

고루틴

```go
package main

import (
    "fmt"
    "time"
)

func main() {
    for i := 0 ; i < 10 ; i++{
        go func(num int) {
            fmt.Println( a...: "closer : ", num)
        }(i)
    }
    time.Sleep( d: 1000000000)
}
```

```
<4 go setup calls>
closer :  9
closer :  0
closer :  1
closer :  2
closer :  5
closer :  6
closer :  7
closer :  8
closer :  3
closer :  4
```

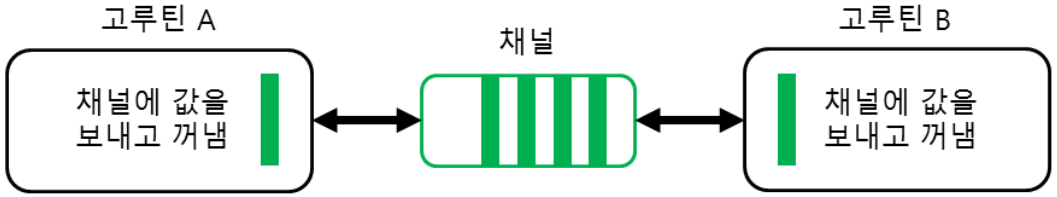
```go
package main

import (
    "fmt"
    "time"
)

func main() {
    buffer := []int{1,2,3,4,5}
    for i := 0 ; i < 10 ; i++{
        go func(num int) {
            buffer[3] *= num+1
            fmt.Println(num, "closer : ", i, buffer)
        }(i)
        //time.Sleep(1)
    }
    time.Sleep( d: 1000000000)
}
```

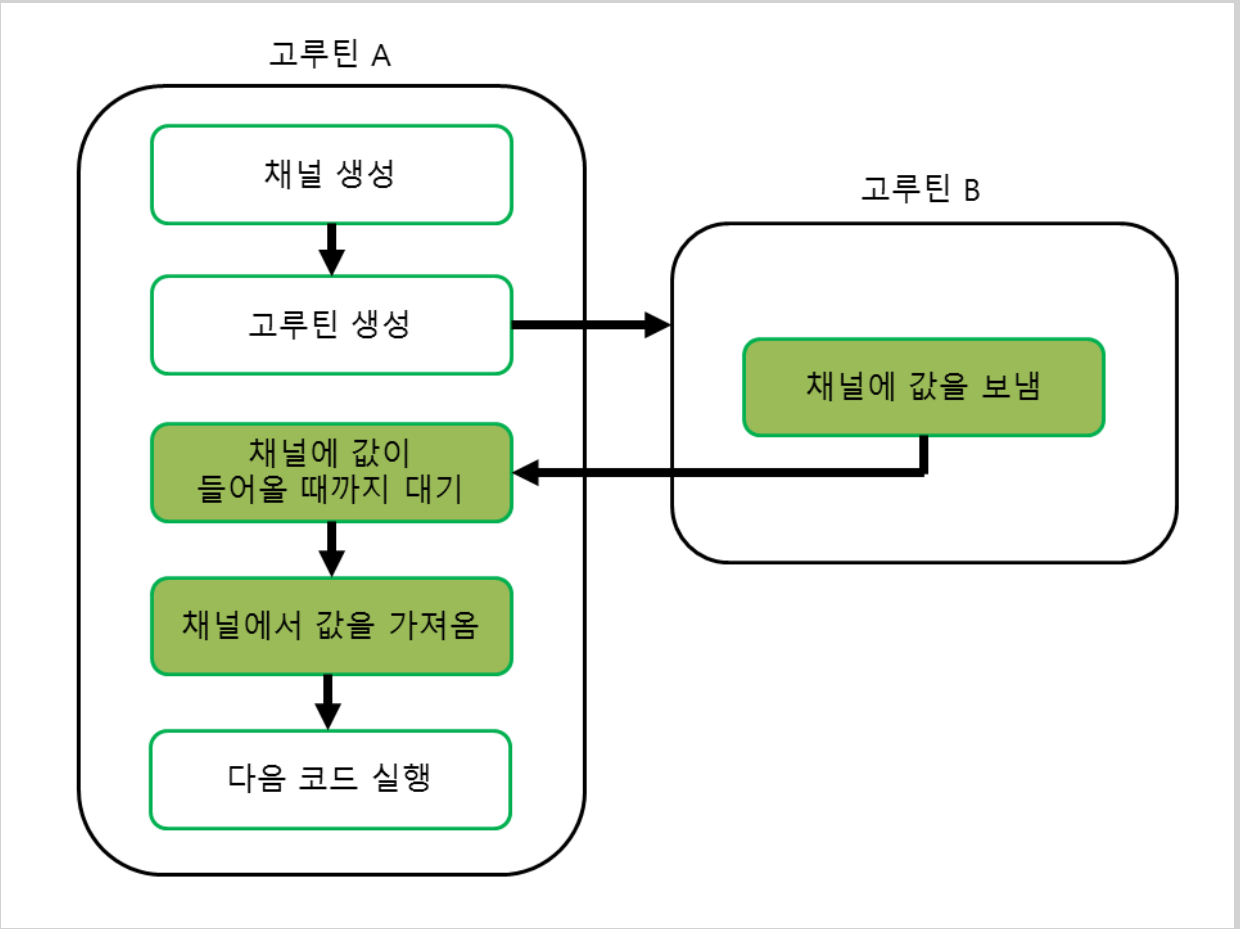
```
9 closer :  10 [1 2 3 160 5]
2 closer :  10 [1 2 3 7680 5]
0 closer :  10 [1 2 3 160 5]
1 closer :  10 [1 2 3 320 5]
5 closer :  10 [1 2 3 2073600 5]
3 closer :  10 [1 2 3 160 5]
4 closer :  10 [1 2 3 345600 5]
8 closer :  10 [1 2 3 345600 5]
7 closer :  10 [1 2 3 7680 5]
6 closer :  10 [1 2 3 14515200 5]
```

채널

- 채널(Channel)
  - Goroutine 끼리 데이터를 주고 받기 위한 통신 메커니즘
  - 특징
    - 모든 타입을 채널로 사용할 수 있음(Reference Type)
    - 각 채널마다 특정 데이터 타입으로 데이터를 교환(원소 타입, element type)
    - 채널로 데이터를 보내는 Goroutine이 존재해야 함
    - 채널 선언은 chan, 닫으려면 close
    - 채널의 방향을 설정해야 함

채널





출처: http://pyrasis.com/book/GoForTheReallyImpatient/Unit34

채널

```go
package main

import "fmt"

func sum(a ,b int, c chan int){
    c <- a + b
}
func main() {
    c := make(chan int)
    go sum( a: 1, b: 2,c)
    n := <- c
    fmt.Println(n)

}
```

```
<4 go setup calls>
3
```

```go
package main

import "fmt"

func sum(a ,b int, c chan int){
    c <- a + b
}
func test(c chan string){
    c <- "hihi"
}
func main() {
    channel := make(chan int)
    var channelString = make(chan string)
    go sum( a: 1, b: 2,channel)
    go test(channelString)
    n := <- channel
    temp := <- channelString

    fmt.Println(n, temp)

}
```

```
<4 go setup calls>
3 hihi
```

채널

```go
package main

import (
    "fmt"
    "time"
)

func main() {
    channel := make(chan int)
    count := 3
    go func() {
        for i:= 0 ; i < count ; i++{
            channel <- i
            fmt.Println( a...: "Goroutine : ", i)
            time.Sleep( d: 1000000000)
        }
    }()

    for i:= 0 ; i < count ; i++{
        <- channel
        fmt.Println( a...: "Main Func : ", i)
    }
}
```

```
<4 go setup calls>
Goroutine :  0
Main Func :  0
Goroutine :  1
Main Func :  1
Goroutine :  2
Main Func :  2
```

- 동기 채널
- 값이 들어오면, 받아서 사용

채널

```go
package main

import (
    "fmt"
)

func main() {
    channel := make(chan int, 5)
    count := 10
    go func() {
        for i:= 0 ; i < count ; i++{
            channel <- i
            fmt.Println( a...: "Goroutine : ", i)
        }
    }()

    for i:= 0 ; i < count ; i++{
        <- channel
        fmt.Println( a...: "Main Func : ", i)
    }
}

```

```
<4 go setup calls>
Goroutine :  0
Goroutine :  1
Goroutine :  2
Goroutine :  3
Goroutine :  4
Goroutine :  5
Main Func :  0
Main Func :  1
Main Func :  2
Main Func :  3
Main Func :  4
Main Func :  5
Main Func :  6
Goroutine :  6
Goroutine :  7
Goroutine :  8
Goroutine :  9
Main Func :  7
Main Func :  8
Main Func :  9
```

- 비동기 채널
- 버퍼에 값이 있으면, 알아서 사용
- 보통 버퍼가 가득 차면 값을 꺼내서 사용

채널

```go
package main

import (
    "fmt"
)

func main() {
    channel := make(chan int)
    count := 10
    go func() {
        for i:= 0 ; i < count ; i++{
            channel <- i
            fmt.Println( a...: "input Channel")
            //time.Sleep(1)
        }
        close(channel)
    }()

    for i := range channel{
        fmt.Println( a...: "out channel : ", i)
    }


}
```

```
input Channel
out channel :  0
out channel :  1
input Channel
input Channel
out channel :  2
out channel :  3
input Channel
input Channel
out channel :  4
out channel :  5
input Channel
input Channel
out channel :  6
out channel :  7
input Channel
input Channel
out channel :  8
out channel :  9
input Channel
```

- Print 함수와 화면 표시 순서가 다름
- Main goroutine이 먼저 실행 된다고 해도, 화면 IO 부분에서 순서가 바뀔 수 있음

채널

```go
package main

import (
    "fmt"
)

func main() {
    channel := make(chan int)
    count := 10
    go func() {
        for i:= 0 ; i < count ; i++{
            channel <- i
            fmt.Println( a...: "input Channel")
            //time.Sleep(1)
        }
        //close(channel)
    }()

    for i := range channel{
        fmt.Println( a...: "out channel : ", i)
    }

}
```

```
input Channel
out channel :  0
out channel :  1
input Channel
input Channel
out channel :  2
out channel :  3
input Channel
input Channel
out channel :  4
out channel :  5
input Channel
input Channel
out channel :  6
out channel :  7
input Channel
input Channel
out channel :  8
out channel :  9
input Channel
fatal error: all goroutines are asleep - deadlock!

goroutine 1 [chan receive]:
main.main()
    C:/Users/holly/Desktop/workspace_go/src/main/2day.go:19 +0x11d
```

- 채널을 안 닫아주면 Deadlock 발생
- 무한정 대기하기 때문

채널

```go
package main

import "fmt"

func main() {
    channel := make(chan int)

    go func() {
        channel <- 1
    }()

    a, ok := <- channel
    fmt.Println(a, ok)

    close(channel)
    a, ok = <- channel
    fmt.Println(a, ok)

}
```

```
<4 go setup calls>
1 true
0 false
```

- 채널 오픈 유무 확인

채널

```go
package main

import (
    "fmt"
    "time"
)

func test1(c <-chan int){
    fmt.Println(<-c)
}
func test2(c chan<- int){
    c <- 10
}
func main() {
    channel := make(chan int)

    go test1(channel)
    go test2(channel)

    time.Sleep( d: 1000000)
}
```

```
<4 go setup calls>
10
```

```go
package main

import (
    "fmt"
    "time"
)

func test1(c chan<- int){
    fmt.Println(<-c)
}
func test2(c <-chan int){
    c <- 10
}
func main() {
    channel := make(chan int)

    go test1(channel)
    go test2(channel)

    time.Sleep( d: 1000000)
}
```

```
<3 go setup calls>
# command-line-arguments
.\2day.go:9:14: invalid operation: <-c (receive from send-only type chan<- int)
.\2day.go:12:4: invalid operation: c <- 10 (send to receive-only type <-chan int)
```

- <-chan : 보내기 전용
- chan<- : 받기 전용

셀렉트

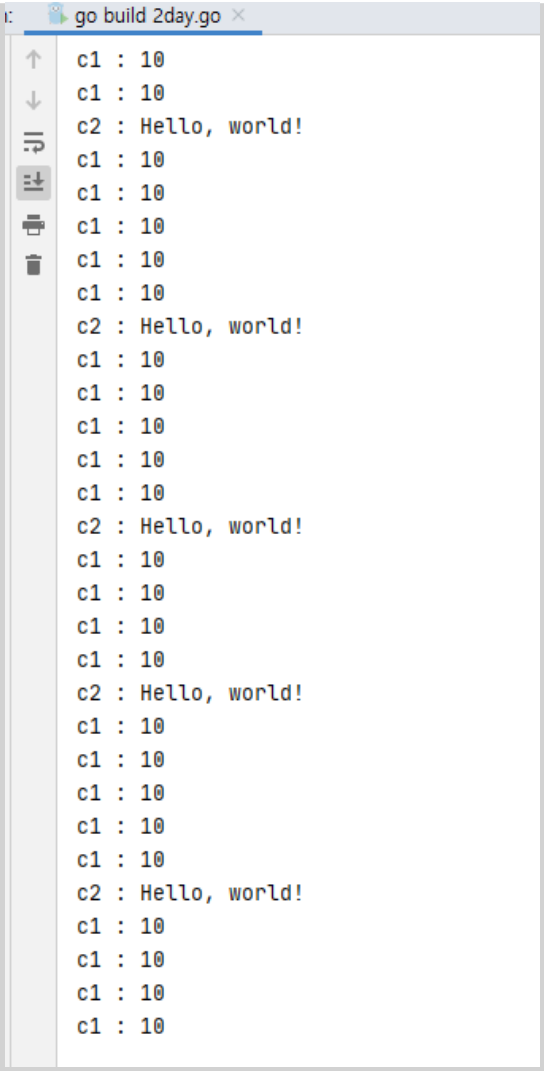
- 셀렉트(Select)
  - 여러 개의 채널을 하나의 Select block에서 다룰 수 있음
  - 특징
    - Switch문과 같은 형식을 가짐 (인자가 Channel)
    - Goroutine 연산 여러 개를 컨트롤 가능

셀렉트

- 셀렉트(Select)
  - 여러 개의 채널을 하나의 Select block에서 다룰 수 있음
  - 특징
    - Switch문과 같은 형식을 가짐 (인자가 Channel)
    - Goroutine 연산 여러 개를 컨트롤 가능

셀렉트

```go
package main

import ...

func main() {
    c1 := make(chan int)
    c2 := make(chan string)
    go func() {
        for {
            i := <-c1
            fmt.Println( a...: "c1 :", i)
            time.Sleep(100 * time.Millisecond)
        }
    }()
    go func() {
        for {
            c2 <- "Hello, world!"
            time.Sleep(500 * time.Millisecond)
        }
    }()
    go func() {
        for { // 무한 루프
            select {
            case c1 <- 10:
            case s := <-c2:
                fmt.Println( a...: "c2 :", s)
            }
        }
    }()
    time.Sleep(10 * time.Second)
}
```



셀렉트

```go
package main

import ...

func main() {
    c1 := make(chan int)
    c2 := make(chan string)
    go func() {
        for {
            i := <-c1
            fmt.Println( a...: "c1 :", i)
            time.Sleep(100 * time.Millisecond)
        }
    }()
    go func() {
        for {
            c2 <- "Hello, world!"
            time.Sleep(500 * time.Millisecond)
        }
    }()
    go func() {
        for { // 무한 루프
            select {
            case <-time.After(50 * time.Millisecond):
                fmt.Println( a...: "timeOut!")
                return
            case c1 <- 10:
            case s := <-c2:
                fmt.Println( a...: "c2 :", s)


            }
        }
    }()
    time.Sleep(10 * time.Second)
}
```

```
<4 go setup calls>
c2 : Hello, world!
c1 : 10
timeOut!
```