Goroutine

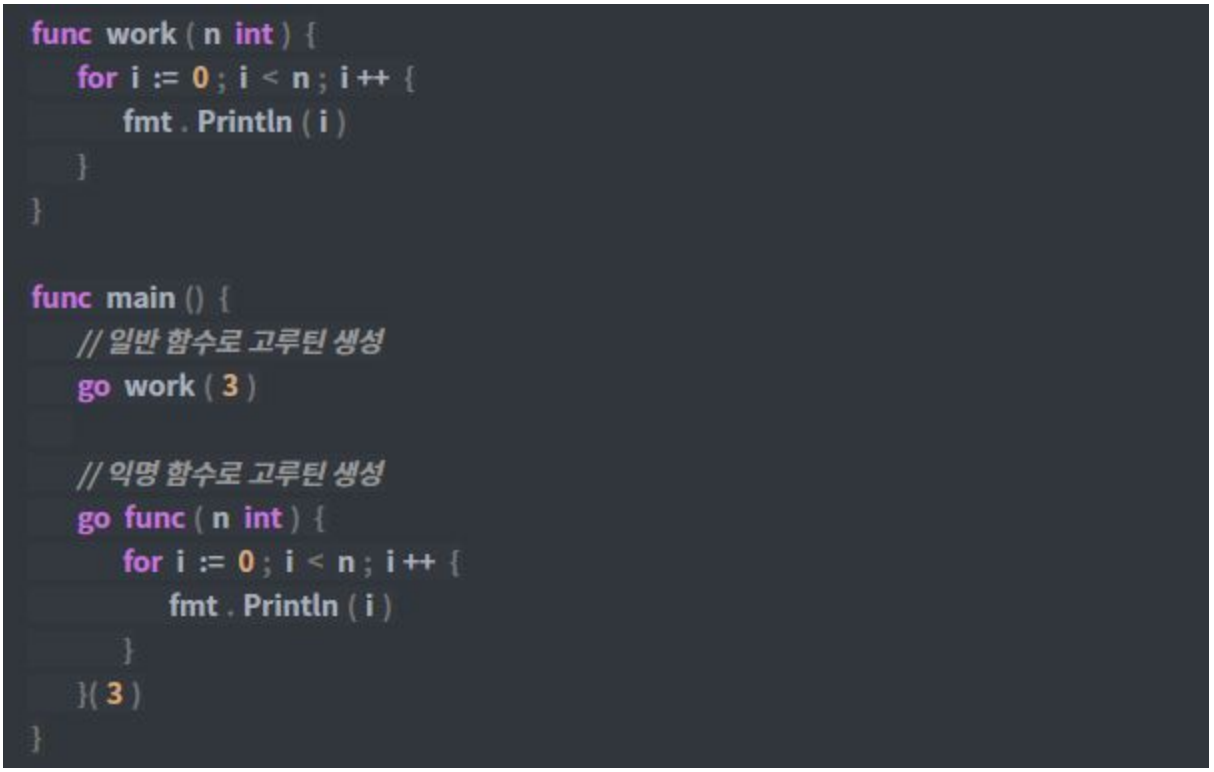
- Go 런타임에 의해 관리되는 경량 스레드

- Goroutine을 사용하면 비동기적으로 여러개의 함수 실행 가능

- 생성 방법 (go 키워드 사용)
    - 일반 함수 사용
        - go work (3)
    - 익명 함수 사용
        - go func (n int)

- main 함수와는 독립적으로 실행되지만 main 함수가 종료되면 모든 고루틴 종료

- Goroutine보다 main이 먼저 종료되는걸 방지하기 위해 sync 라이브러리에 있는 WaltGroup이라는 세마포어를 활용해 goroutine의 종료를 대기시킴
- 예제



```
func work ( n int ) {
    for i := 0 ; i < n ; i ++ {
        fmt . Println ( i )
    }
}

func main () {
    // 일반 함수로 고루틴 생성
    go work ( 3 )

    // 익명 함수로 고루틴 생성
    go func ( n int ) {
        for i := 0 ; i < n ; i ++ {
            fmt . Println ( i )
        }
    }( 3 )
}
```

Channel

- Goroutine끼리 데이터를 주고 받아야 하는 경우 사용
- Goroutine들을 연결해주는 일종의 pipe
- chan 키워드로 생성할 수 있으며 채널에 들어가는 데이터는 모든 타입이 가능
- 예제

```
ourChannel chan int
```

정수를 주고받는 채널

```
func routine1 ( ourChannel chan string ) {
    // 채널에 값을 넣습니다.
    ourChannel <- "data"
}

func routine2 ( ourChannel chan string ) {
    // 채널로부터 값을 받습니다.
    fmt . Println ( <- ourChannel )
    // 출력값 : data
}

func main () {
    // string 채널을 위한 메모리를 할당합니다.
    ourChannel := make ( chan string )

    go routine1 ( ourChannel )
    go routine2 ( ourChannel )
}
```

< 양방향 채널 (Default) >

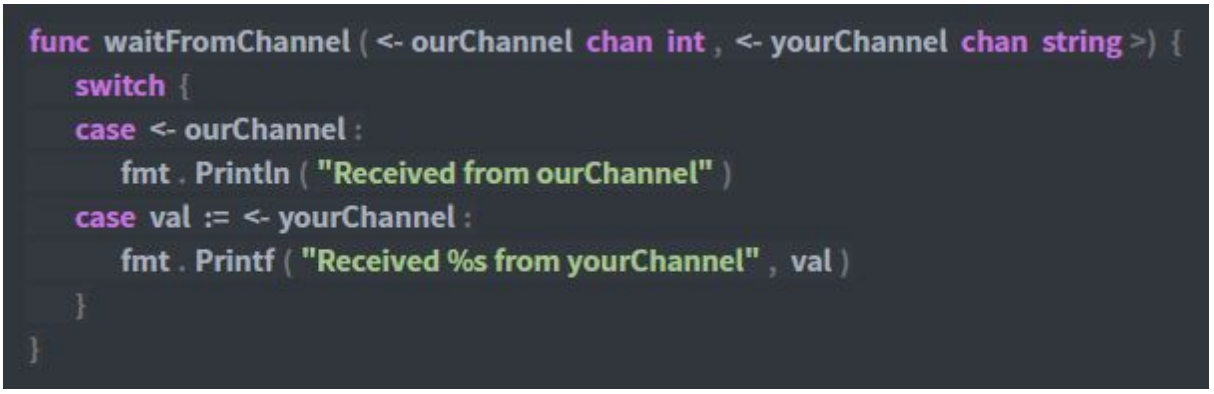
```
// 보내기 전용 단방향 채널을 사용합니다.
func routine1 ( ourChannel <- chan string ) {
    // 채널에 값을 넣습니다.
    ourChannel <- "data"
}

// 받기 전용 단방향 채널을 사용합니다.
func routine2 ( <- ourChannel chan string ) {
    // 채널로부터 값을 받습니다.
    fmt . Println ( <- ourChannel )
    // 출력값 : data
}

func main () {
    ...
}
```

< 단방향 채널 >

어떤 Goroutine이 특정 채널로부터 값을 받을 때까지 대기해야 하는 상황이 있다. 이때 switch/case 문을 사용할 수 있다.



```
func waitFromChannel(<-ourChannel chan int, <-yourChannel chan string>) {
    switch {
    case <-ourChannel:
        fmt.Println("Received from ourChannel")
    case val := <-yourChannel:
        fmt.Printf("Received %s from yourChannel", val)
    }
}
```