Distributed Communication 6th practice

Li Jianhao
lijianhao288@hotmail.com

1 Basics

1.1 Select

Syntax:
select {
case $< ChannelOperation >$:
...
default:
}

Channel Operations:

1. $< - < ChannelName >$
2. $< Message > := < - < ChannelName >$
3. $< ChannelName > < - < Message >$

1.2 Select timeout

func After(d Duration) $< -$chan Time

$< -$chan
chan$< -$

```
package main                                                    1
import (                                                        2
    "fmt"                                                       3
    "time"                                                      4
)                                                               5
                                                                6
func main() {                                                   7
    c := make(chan int)                                         8
                                                                9
```

```
    go func() {
        c <- 1
        fmt.Println("g1 sent 1")
    }()

    select {
    case msg := <-c:
        fmt.Println("Main received ", msg)
    case <- time.After(3 * time.Second):
        fmt.Println("Timeout , Quit")
    }
}
```

Listing 1: Select timeout 1

```
g1 sent 1
Main received  1
```

```
package main
import (
    "fmt"
    "time"
)

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

    go func() {
        time.Sleep(4 * time.Second)
        c <- 1
        fmt.Println("g1 sent 1")
    }()

    select {
    case msg := <-c:
        fmt.Println("Main received ", msg)
    case <- time.After(3 * time.Second):
        fmt.Println("Timeout , Quit")
    }
}
```

Listing 2: Select timeout 2

```
Timeout , Quit
```

1.3 Select in for loop

```
package main
import (
    "fmt"
    "time"
)

func main() {
    c1 := make(chan int)
```

```
    c2 := make(chan int)

    go func() {
        for i := 0; i < 5; i++ {
            time.Sleep(500 * time.Millisecond)
            c1 <- i
            fmt.Println("g1 sent ", i)
        }
    }()

    go func() {
        for i := 0; i < 5; i++ {
            time.Sleep(500 * time.Millisecond)
            c2 <- i
            fmt.Println("g2 sent ", i)
        }
    }()

    L: for {
        select {
        case msg := <-c1:
            fmt.Println("Main received ", msg, " from g1")
        case msg := <-c2:
            fmt.Println("Main received ", msg, " from g2")
        case <- time.After(3 * time.Second):
            fmt.Println("Timeout , Quit")
            break L
        }
    }
}
```

Listing 3: Select in for loop

```
Main received  0  from g1
Main received  0  from g2
g1 sent  0
g2 sent  0
g1 sent  1
Main received  1  from g1
Main received  1  from g2
g2 sent  1
g1 sent  2
Main received  2  from g1
Main received  2  from g2
g2 sent  2
g1 sent  3
Main received  3  from g1
Main received  3  from g2
g2 sent  3
g2 sent  4
Main received  4  from g2
Main received  4  from g1
g1 sent  4
Timeout , Quit
```

1.4 Select default

```
package main
import (
    "fmt"
    "time"
)

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

    go func() {
        time.Sleep(2 * time.Second)
        c <- 1
        fmt.Println("g1␣sent␣1")
    }()

    L: for {
        select {
        case msg := <-c:
            fmt.Println("Main␣received␣", msg)
            break L
        default:
            time.Sleep(500 * time.Millisecond)
            fmt.Println("Default")
        }
    }
}
```

Listing 4: Select Default

```
Default
Default
Default
Default
Main received  1
```

1.5 Select response

```
package main
import (
    "fmt"
    "time"
)

func main() {
    add := make(chan request)
    multiply := make(chan request)

    go func() {
        for i := 0; i < 3; i++ {
            time.Sleep(500 * time.Millisecond)
            req := request{i+2,i+3,make(chan int)}
            add <- req
            result:= <- req.resp
            fmt.Println("(add)␣g1␣sent", req.first, req.second, "received", result)
        }
    }()
```

```
    go func() {
        for i := 0; i < 3; i++ {
            time.Sleep(500 * time.Millisecond)
            req := request{i+2,i+3,make(chan int)}
            multiply <- req
            result:= <- req.resp
            fmt.Println("(multiply) g2 sent", req.first, req.second, "received", result)
        }
    }()

    L: for {
        select {
        case msg := <-add:
            result:= msg.first + msg.second
            msg.resp <- result
            fmt.Println("(add) Main received ", msg.first, msg.second,"sent result ",result)
        case msg := <-multiply:
            result:= msg.first * msg.second
            msg.resp <- result
            fmt.Println("(multiply) Main received ", msg.first, msg.second,"sent result ",result)
        case <- time.After(3 * time.Second):
            fmt.Println("Timeout, Quit")
            break L
        }
    }
}

type request struct{
    first int
    second int
    resp chan int
}
```

Listing 5: Select response

```
(add) g1 sent 2 3 received 5
(add) Main received  2 3 sent result  5
(multiply) g2 sent 2 3 received 6
(multiply) Main received  2 3 sent result  6
(multiply) Main received  3 4 sent result  12
(add) g1 sent 3 4 received 7
(multiply) g2 sent 3 4 received 12
(add) Main received  3 4 sent result  7
(multiply) Main received  4 5 sent result  20
(add) g1 sent 4 5 received 9
(multiply) g2 sent 4 5 received 20
(add) Main received  4 5 sent result  9
Timeout, Quit
```

2 Practice

2.1 p1

Create a channel of channel of string named **c** (Channel of string inside the channel). This channel does not have a buffer.

The main function starts a new goroutine, let's call it g1. g1 receive a message from the **c**, the message is stored in variable **insideC**. Print out "g1 receive the channel inside". Send a "Hello" to the **insideC**. Print out "g1 sent Hello to the channel inside".

The main function creates a channel of string named **in**. This channel does not have a buffer.

The main function defines a select with a 3-second timeout. When it is time out, it prints "Timeout, Quit" and breaks the select. There is a case try to send the channel **in** to the channel **c**. In this case:

1. Print out "Main send the channel inside".
2. Receive message from the channel **in**, store the value in variable **resp**.
3. Print out "Main received" and the **resp**.

2.2 p2

Create a struct **request**. Its fields: first (type string), second (type string), resp (type channel of string).

The main function creates two channels of request. Both channels do not have a buffer. One named **attach**, another one named **reverse**.

The main function starts a new goroutine, let's call it g1. g1 has a for loop which iterates 3 times. In each iteration:

1. sleep 500 millisecond
2. convert iterator i to string, store the string in **s1**
3. convert i+1 to string, store the string in **s2**

4. create a typed value of request, named as **req**. **s1** as the first, **s2** as the second, a new channel of string as resp.

5. send the **req** to channel **attach**.

6. receive message from the **resp** of **req**. store the received value as **result**.

7. print out "(attach) g1 sent", first of **req**, second of **req**, "received", and the **result**.

The main function starts another new goroutine, let's call it g2. It is similar to g1. There are two difference: The **req** will be sent to the channel **reverse**. The print out is different. It prints out "(reverse) g2 sent", first of **req**, second of **req**, "received", and the **result**.

The main function uses the for loop and the select to receive requests from the channel **attach** and **reverse**. It has a 3-second timeout. If timeout, print "Timeout, Quit" and break the for loop. For each request received from the channel **attach**:

1. attach the first and the second (the first is on the left). The result is stored in the variable **result**.

2. send the **result** to the request's **resp**.

3. print out "(attach) Main received", request's first, request's second, "sent result", and the **result**.

For each request received from the channel **reverse**:

1. attach the first and the second (the first is on the right). The result is stored in the variable **result**.

2. send the **result** to the request's **resp**.

3. print out "(reverse attach) Main received", request's first, request's second, "sent result", and the **result**.