Distributed Communication 4th practice

Li Jianhao
lijianhao288@hotmail.com

1 Basics

1.1 Goroutine

Syntax: go $< FunctionCall >$

```
package main

import (
    "fmt"
    "time"
)

func main(){
    go printR("Hi␣1")
    go print("Hi␣2")
    time.Sleep(1 * time.Second)
}

func print(s string) {
    fmt.Println(s)
}


func printR(s string) string {
    fmt.Println(s)
    return s
}
```

Listing 1: Goroutine

```
Hi 1
Hi 2
```

Syntax: go func ($< Parametersandtheirtypes >$) {$< Functionbody >$} ($Arguments$)

```
package main

import (
    "fmt"
    "time"
)

func main(){
    go func () {
        fmt.Println("Hi␣1")
    }()

    go func (s string) {
        fmt.Println(s)
    }("Hi␣2")

    time.Sleep(1 * time.Second)
}
```

Listing 2: Goroutine2

```
Hi 1
Hi 2
```

1.2 WaitGroup

syntax:

1. import the package "sync"
2. var $< TypedValueName >$ sync.WaitGroup
3. $< TypedValueName >$.Add(1)
   //Before start each goroutine
4. defer $< TypedValueName >$.Done()
   //In the goroutine function body
5. $< TypedValueName >$.Wait()
   //At the place where we want to wait for all the goroutines

```
package main

import (
    "fmt"
    "sync"
)

func main(){

    var wg sync.WaitGroup

    for i := 0; i < 5; i++ {
        wg.Add(1)
        go func (a int) {
            defer wg.Done()
            fmt.Println("Hi",a)
        }(i)
    }

    wg.Wait()
}
```

Listing 3: WaitGroup

(The order may different)

```
Hi 4
Hi 1
Hi 0
Hi 2
Hi 3
```

At the place where we call the function Println, if we pass the i as the argument instead of a. The output will be:

```
Hi 5
Hi 5
Hi 5
Hi 5
Hi 5
```

1.3 Mutex

Syntax:

1. import the package "sync"
2. var $< TypedValueName >$ sync.Mutex
3. $< TypedValueName >$.Lock()
   //Before access the shared variable
4. $< TypedValueName >$.Unlock()
   //After access the shared variable

```
package main

import (
    "fmt"
    "sync"
)

func main(){

    var mu sync.Mutex
    var wg sync.WaitGroup

    var result = 0

    for i := 0; i < 1000; i++ {
        wg.Add(1)
        go func (a int) {
            defer wg.Done()
            mu.Lock()
            result += a
            mu.Unlock()
        }(i)
    }

    wg.Wait()

    fmt.Println(result)
}
```

Listing 4: Mutex

```
499500
```

If we do not use the Mutex:

```
package main

import (
    "fmt"
     "sync"
)

func main(){
    var wg sync.WaitGroup

    var result = 0

    for i := 0; i < 1000; i++ {
        wg.Add(1)
        go func (a int) {
            defer wg.Done()
            result += a
        }(i)
    }

    wg.Wait()

    fmt.Println(result)
}
```

Listing 5: NoMutex

The output will be wrong (May differnt when you run multiple time)

```
go run NoMutex.go
493000
go run NoMutex.go
489299
```

```
go run NoMutex.go
493126
```

Race condition

result += a (Contains three small operations: Read original value, Calculate new value, Write new value)
(Review: "result += a" equals "result = result + a")

Suppose:
result := 0
goroutine1: result += 1
goroutine2: result += 2

If we used Mutex, The result will be 3.

If we do not use Mutex, it can happen:
goroutine1 read result equals 0
goroutine2 read result equals 0
Then, goroutine 1 wants to write value 1 to the result, goroutine 2 wants to write value 2 to the result. We may get the wrong result 1 or 2.

1.4 AddUint64

int64: 64 bits, $-2^{63}\,to\,2^{63}-1$.
uint64: 64 bits, $0\,to\,2^{64}-1$.
Syntax: $< Name >$:= $< NewType >$($< Value >$)
Syntax:

1. import the package "sync/atomic"
2. var $< TypedValueName >$ uint64
3. atomic.AddUint64(&$< TypedValueName >$, $Value$)

```
package main

import (
    "fmt"
    "sync"
    "sync/atomic"
)

func main(){

    var result uint64
    var wg sync.WaitGroup

    for i := 0; i < 1000; i++ {
        wg.Add(1)
        go func (a int) {
            defer wg.Done()
            u:= uint64(a)
            atomic.AddUint64(&result, u)
        }(i)
    }

    wg.Wait()

    fmt.Println(result)
}
```

Listing 6: AddUint64

```
499500
```

2 Practice

2.1 p1

Step 1:

Include the packages: ”net/http”, ”time”. Include the function **linkTest** below. It takes a string (the URL) and return a string (The test result ”good” or ”bad”).

```
func linkTest(link string) string {
    client := http.Client{
        Timeout: 3*time.Second,
    }
    _, err := client.Get("http://"+link)
    if err != nil {
        resultString := "bad"
        return resultString
    }
    resultString := "good"
    return resultString
}
```

Step 2: Create a slice of string. Its name is **urls**. It has five initial elements: "www.google.com", "www.facebook.com", "www.asdfasfad.com", "www.ebay.com", "www.amazon.com".

Step 3: Create a map **urlMap** which maps the string to string. We use it to store the URL and its test results.

For each elements in the **urls**, we start a new goroutine to test the URL (Use the function **linkTest**) and write the result into the **urlMap**. The key is the URL, the value is the result that returned by the function **linkTest**.

Use the Mutex to deal with the mutual exclusion problem when accessing the **urlMap**.

Use the WaitGroup, let the main function wait until all the created goroutines finish. The main function will print out the **urlMap** after that.

The output will be like:

```
map[www.amazon.com:good www.asdfasfad.com:bad www.ebay.com:good www.facebook.com:good

www.google.com:good]
```

2.2 p2

Step 1: Create a new program file. Include the packages: "math/rand". Use the following statement to generate a random int between 0 and 1000. Assign this random int to variable r.

```
r := rand.Intn(1000)
```

Step 2: Declare a variable **result** of type uint64.

There is a For loop that will iterate r times. Each time we start a new goroutine that uses the AddUint64 to add 1 to the result.

Use the WaitGroup, let the main function wait until all the created goroutines finish. The main function will print out the boolean which shows if the result is equal to r. (hint: the compare need type conversion)