Benchmarking, Accessing and Listening

This lesson provides an explanation on how to benchmark a goroutine, run a backend goroutine, and use reflection on channels.

WE'LL COVER THE FOLLOWING

- Benchmarking goroutines
- Concurrent access to objects by using a channel
- Using reflection to listen to a dynamically created set of channels

Benchmarking goroutines

In Chapter 11, we mentioned the principle of performing benchmarks on your functions in Go. Here, we apply it to a concrete example of a goroutine that is filled with ints and then read. The functions are called **N** times (e.g. N =1000000) with `testing.Benchmark`. The `BenchMarkResult` has a `String()` method for outputting its findings. The number **N** is decided upon by `go test`. It is taken to be high enough to get a reasonable benchmark result.

Of course, the same way of benchmarking also applies to ordinary functions. If you want to exclude certain parts of the code or you want to be more specific in what you are timing, you can stop and start the timer by calling the functions `testing.B.StopTimer()` and `testing.B.StartTimer()` as appropriate. The benchmarks will only be run if all your tests pass!

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

func main() {
  fmt.Println(" sync", testing.Benchmark(BenchmarkChannelSync).String())
  fmt.Println("buffered", testing.Benchmark(BenchmarkChannelBuffered).String())
}

func BenchmarkChannelSync(b *testing.B) { // makes buffered channel
```

```
func BenchmarkChannelSync(b *testing.B) { // makes buffered channel
  ch := make(chan int)
  go func() {
    for i := 0; i < b.N; i++ {
      ch <- i
    }
    close(ch)
  }()
  for _ = range ch { // iterating over channel without doing anything
  }
}

func BenchmarkChannelBuffered(b *testing.B) { // makes buffered channel with capacity of 128
  ch := make(chan int, 128)
  go func() {
    for i := 0; i < b.N; i++ {
      ch <- i
    }
    close(ch)

  }()
  for _ = range ch {
  }
}
```

Benchmark Channels

In order to do benchmarking, you have to import the testing package (see **line 4**). The `main()` calls first the `BenchmarkChannelSync()` function, and then the `BenchmarkChannelBuffered()` function; both require a parameter of type `*testing.B`.

Look at the header of the function `BenchmarkChannelSync()` at **line 12**. It makes an unbuffered channel `ch` and then starts a goroutine with an anonymous function, which puts a large number of successive integers on `ch`. In the `main()` routine at **line 20**, it iterates over the channel without doing anything.

Look at the header of function `BenchmarkChannelBuffered()` at **line 24**. It makes a buffered channel `ch` with a capacity of **128** and then performs the same action as `BenchmarkChannelSync()`

The results show clearly that using a buffered channel is much more efficient.

Concurrent access to objects by using a channel

To safeguard concurrent modifications of an object instead of using locking with a `sync` Mutex, we can also use a backend goroutine for the sequential execution of anonymous functions. Here is an example program:

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

type Person struct {
  Name string
  salary float64
  chF chan func()
}

func NewPerson(name string, salary float64) *Person {
  p := &Person{name, salary, make(chan func())}
  go p.backend()
  return p
}

func (p *Person) backend() {
  for f := range p.chF {
    f()
  }
}

// Set salary.
func (p *Person) SetSalary(sal float64) {
  p.chF <- func() { p.salary = sal }
}

// Retrieve salary.
func (p *Person) Salary() float64 {
  fChan := make(chan float64)
  p.chF <- func() { fChan <- p.salary }
  return <-fChan
}

func (p *Person) String() string {
  return "Person - name is: " + p.Name + " - salary is: " +
  strconv.FormatFloat(p.Salary(), 'f', 2, 64)
}

func main() {
  bs := NewPerson("Smith Bill", 2500.5)
  fmt.Println(bs)
  bs.SetSalary(4000.25)
  fmt.Println("Salary changed:")
  fmt.Println(bs)
}
```

Concurrent Access

In the program above, we have a type `Person` defined at **line 7**, which now contains a field `chF`, a channel of anonymous functions. A `Person` is initialized in the constructor method `NewPerson` at **line 14**. This method also starts a function `backend()` as a goroutine (**line 15**), and returns a pointer to a `Person` at **line 16**.

The `backend()` function executes (from **line 20** to **line 22**) all the functions placed on `chF` in an infinite loop, *effectively serializing them and thus providing safe concurrent access*. The methods that change (`SetSalary`, see its header at **line 26**) and retrieve the *salary* (Salary, see its header at **line 31**) create an anonymous function that does the changing or retrieving (resp. at **line 27** and **line 33**) and put this function on `chF`.

As we saw above, `backend()` will sequentially execute functions. Notice how in the method `Salary` (at **line 32** and **line 33**), the created closure function includes the channel `fChan`. The `String()` method on `Person`(from **line 37** to **line 40**), simply gets all info from a `Person` and puts that into a formatted string.

In `main()`, we create a specific `Person` at **line 43**, then print out the info. Then, we change the salary at **line 45** and again print the new info. Retrieving and changing the info is performed by `backend()` in the separate goroutine started at **line 15**. Because this goroutine can only execute one function at a time, a concurrency problem can not occur.

This is, of course, a simplified example, and it should not be applied in such simple cases. However, it shows how the problem could be tackled in more complex situations.

Using reflection to listen to a dynamically created set of channels

As the following example shows, you can combine the use of channels with reflection:

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

func produce(ch chan<- string, i int) {
  for j := 0; j < 5; j++ {
```

```
      ch <- fmt.Sprint(i*10 + j)
    }
    close(ch)
}

func main() {
    numChans := 4
    //I keep the channels in this slice, and want to "loop" over them in the select statement
    var chans = []chan string{}

    for i := 0; i < numChans; i++ {
      ch := make(chan string)
      chans = append(chans, ch)
      go produce(ch, i+1)
    }

    cases := make([]reflect.SelectCase, len(chans))
    for i, ch := range chans {
      cases[i] = reflect.SelectCase{Dir: reflect.SelectRecv, Chan: reflect.ValueOf(ch)}
    }

    remaining := len(cases)
    for remaining > 0 {
      chosen, value, ok := reflect.Select(cases)
      if !ok {
        // The chosen channel has been closed, so zero out the channel to disable the case
        cases[chosen].Chan = reflect.ValueOf(nil)
        remaining--
        continue
      }
      fmt.Printf("Read from channel %#v and received %s\n", chans[chosen], value.String())
    }

}
```

Reflection on Channels

In this code example, we define `numChans` channels of string and keep them in a slice `chance`, defined at **line 17**. From **line 19** to **line 23**, we make the channels, put them in the slice, and then for each channel, we start a goroutine `produce` on it at **line 22**.

Now, look at the header of the `produce()` function at **line 7**. It takes the channel `ch` for write-only, together with an integer `i` as a parameter. At **line 8**, a for-loop starts, which puts **5** strings on the channel (see **line 9**). Then, the channel is closed at **line 11**.

At **line 25** and beyond, we start using reflection, so we have to import that package first at **line 4**. We make a slice of `reflect.SelectCase` instances called `cases` at **line 25**, and use reflection on it, using for loop (from **line 26** to **line**

28). Then, in the for loop starting at **line 31**, we use `reflect.Select` to get the value out of the instance, to be printed out at **line 39**. If `ok` is false, then the channel has been closed. Therefore, we put `nil` in the reflection instance, count down (**line 36**), and continue the loop.

The use of reflection always has a negative impact on performance, so don't overuse it.

Now, that you know all the important concepts of goroutines and channels, there is a quiz in the next lesson for you to solve.