Golang Session

- Harsh Dusane

Topic : Golang Goroutines

Goroutines

- Concurrency in Golang is the ability for functions to run independent of each other. Goroutines are functions that are run concurrently. Golang provides Goroutines as a way to handle operations concurrently.
- New goroutines are created by the go statement.

```
sum()     // A normal function call that executes sum synchronously and waits for completing it
go sum()  // A goroutine that executes sum asynchronously and doesn't wait for completing it
```

Creating Goroutines

```go
package main
import (
        "fmt"
        "io/ioutil"
        "log"
        "net/http"
        "time"
)
func responseSize(url string) {
        fmt.Println("Step1: ", url)
        response, err := http.Get(url)
        if err != nil {
                log.Fatal(err)
        }
        fmt.Println("Step2: ", url)
        defer response.Body.Close()
        fmt.Println("Step3: ", url)
        body, err := ioutil.ReadAll(response.Body)
        if err != nil {
                log.Fatal(err)
        }
        fmt.Println("Step4: ", len(body))
}
func main() {
        go responseSize("https://www.golangprograms.com")
        go responseSize("https://coderwall.com")
        go responseSize("https://stackoverflow.com")
        time.Sleep(10 * time.Second)
}
```

Waiting for Goroutines to Finish Execution

```go
package main
import (
        "fmt"
        "io/ioutil"
        "log"
        "net/http"
        "sync"
)
// WaitGroup is used to wait for the program to finish goroutines.
var wg sync.WaitGroup
func responseSize(url string) {
        // Schedule the call to WaitGroup's Done to tell goroutine is completed.
        defer wg.Done()
        fmt.Println("Step1: ", url)
        response, err := http.Get(url)
        if err != nil {
                log.Fatal(err)
        }
        fmt.Println("Step2: ", url)
        defer response.Body.Close()
        fmt.Println("Step3: ", url)
        body, err := ioutil.ReadAll(response.Body)
        if err != nil {
                log.Fatal(err)
        }
        fmt.Println("Step4: ", len(body))
}
func main() {
        // Add a count of three, one for each goroutine.
        wg.Add(3)
        fmt.Println("Start Goroutines")
        go responseSize("https://www.golangprograms.com")
        go responseSize("https://stackoverflow.com")
        go responseSize("https://coderwall.com")
        // Wait for the goroutines to finish.
        wg.Wait()
        fmt.Println("Terminating Program")
}
```

Fetch Values from Goroutines

```
package main
import (
        "fmt"
        "io/ioutil"
        "log"
        "net/http"
        "sync"
)
// WaitGroup is used to wait for the program to finish goroutines.
var wg sync.WaitGroup
func responseSize(url string, nums chan int) {
        // Schedule the call to WaitGroup's Done to tell goroutine is completed.
        defer wg.Done()
        response, err := http.Get(url)
        if err != nil {
                log.Fatal(err)
        }
        defer response.Body.Close()
        body, err := ioutil.ReadAll(response.Body)
        if err != nil {
                log.Fatal(err)
        }
        // Send value to the unbuffered channel
        nums <- len(body)
}
func main() {
        nums := make(chan int) // Declare a unbuffered channel
        wg.Add(1)
        go responseSize("https://www.golangprograms.com", nums)
        fmt.Println(<-nums) // Read the value from unbuffered channel
        wg.Wait()
        close(nums) // Closes the channel
}
```

Play and Pause Execution of Goroutine

```
import (
        "fmt"
        "sync"
        "time"
)
var i int
func work() {
        time.Sleep(250 * time.Millisecond)
        i++
        fmt.Println(i)
}

func main() {
        var wg sync.WaitGroup
        wg.Add(1)
        command := make(chan string)
        go routine(command, &wg)
        time.Sleep(1 * time.Second)
        command <- "Pause"
        time.Sleep(1 * time.Second)
        command <- "Play"
        time.Sleep(1 * time.Second)
        command <- "Stop"
        wg.Wait()
}
```

```
func routine(command <-chan string, wg *sync.WaitGroup) {
        defer wg.Done()
        var status = "Play"
        for {
                select {
                case cmd := <-command:
                        fmt.Println(cmd)
                        switch cmd {
                        case "Stop":
                                return
                        case "Pause":
                                status = "Pause"
                        default:
                                status = "Play"
                        }
                default:
                        if status == "Play" {
                                work()
                        }
                }
        }
}
```

Fix Race Condition using Atomic Functions

```
package main
import (
        "fmt"
        "runtime"
        "sync"
        "sync/atomic"
)
var (
        counter int32          // counter is a variable incremented by all goroutines.
        wg      sync.WaitGroup // wg is used to wait for the program to finish.
)
func main() {
        wg.Add(3) // Add a count of two, one for each goroutine.
        go increment("Python")
        go increment("Java")
        go increment("Golang")
        wg.Wait() // Wait for the goroutines to finish.
        fmt.Println("Counter:", counter)
}
func increment(name string) {
        defer wg.Done() // Schedule the call to Done to tell main we are done.
        for range name {
                atomic.AddInt32(&counter, 1)
                runtime.Gosched() // Yield the thread and be placed back in queue.
        }
}
```

Define Critical Sections using Mutex

```
package main
import (
        "fmt"
        "sync"
)
var (
        counter int32           // counter is a variable incremented by all goroutines.
        wg      sync.WaitGroup // wg is used to wait for the program to finish.
        mutex   sync.Mutex     // mutex is used to define a critical section of code.
)
func main() {
        wg.Add(3) // Add a count of two, one for each goroutine.
        go increment("Python")
        go increment("Go Programming Language")
        go increment("Java")
        wg.Wait() // Wait for the goroutines to finish.
        fmt.Println("Counter:", counter)
}
func increment(lang string) {
        defer wg.Done() // Schedule the call to Done to tell main we are done.
        for i := 0; i < 3; i++ {
                mutex.Lock()
                {
                        fmt.Println(lang)
                        counter++
                }
                mutex.Unlock()
        }
}
```