Concurrency: Goroutines and Channels

16 March 2020

Ákos Frohner

Overview

Part 1: (this) Goroutines and Channels
Part 2: (TODO) Parallel line count
Part 3: (TODO) Search "service"

Environment

Code:

github.com/hu-univ-golang/golang-elte-2020-public/concurrency(https://github.com/hu-univ-golang/golang-elte-2020-public/concurrency)

Examples are in **ex?** subdirectories with skeleton code.

Concurrency is not parallelism

- Parallelism: code executing at the same time.
- Concurrency: how to compose independent threads of execution.

talks.golang.org/2012/concurrency.slide (https://talks.golang.org/2012/concurrency.slide)

www.youtube.com/watch?v=f6kdp27TYZs (https://www.youtube.com/watch?v=f6kdp27TYZs)

Goroutines

- m:n onto OS threads by the Go runtime
- Small initial stack: 2kB
- GOMAXPROCS controls the number of executing goroutimes (~OS threads)

```
go func()
```

Starting goroutines

```go
package main

import (
    "fmt"
    "time"
)

func say(greeting string) {
    time.Sleep(100 * time.Millisecond)
    fmt.Println(greeting)
}

func main() {
    go say("hello")
    go say("world")
    time.Sleep(200 * time.Millisecond)
}
```

Run

Channels

```
ch := make(chan int) // Create a channel
```

The data flows in the direction of the arrow.

```
ch <- v    // Send v to channel ch
v := <-ch // Receive from ch and assign a value to v
```

Sender and receiver are synchronized at the communication.

Futures

```go
package main

import (
    "fmt"
    "time"
)

func translate(word string) <-chan string {
    ch := make(chan string)
    go func() {
        defer close(ch)
        time.Sleep(100 * time.Millisecond)
        ch <- word
    }()
    return ch
}

func main() {
    fmt.Println(<-translate("hello"), <-translate("world"))
}
```

Run

Futures

YES

```
hello := translate("hello")
world := translate("world")
fmt.Println(<-hello, <-world)
```

NO

```
hello := <-translate("hello") // translate("world") not started yet
world := <-translate("world")
fmt.Println(hello, world)
```

Buffered Channel

Channel may have a buffer or length:

```go
ch := make(chan int, 2)
ch <- 1
ch <- 2
fmt.Println(<-ch)
fmt.Println(<-ch)
```

tour.golang.org/concurrency/3 (https://tour.golang.org/concurrency/3)

Exercise 1: parallelize translation

```go
func translate(word string) string {
    time.Sleep(100 * time.Millisecond)
    return word
}
```

```go
func main() {
    text := []string{"hello", "world"}
    start := time.Now()

    // TODO: parallelize the translation of all words in 'text'
    for _, word := range text {
        fmt.Println(translate(word))
    }
    // END

    if time.Since(start) > time.Duration(len(text))*80*time.Millisecond {
        fmt.Println("Too late...")
    }
}
```

play.golang.org/p/PsCM5hUVBRo (https://play.golang.org/p/PsCM5hUVBRo)

Exercise 1 …

Exercise 1 solution

```
    // TODO: parallelize the translation of all words in 'text'
    translated := make(chan string, len(text))
    for _, word := range text {
        go func(word string) {
            translated <- translate(word)
        }(word)
    }
    for range text {
        fmt.Println(<-translated)
    }
    // END
```

play.golang.org/p/s9RnC5--yww (https://play.golang.org/p/s9RnC5--yww)

Close

A sender can close a channel to indicate no more values will be sent:

```
ch := make(chan int, 1)
ch <- 1
close(ch)
```

Receivers can test if there are more values to be received

```
v, open := <-ch
```

play.golang.org/p/ivferqxhmBR (https://play.golang.org/p/ivferqxhmBR)

Range

Loop over a channel while there are values to be received:

```
ch := make(chan int, 3)
ch <- 1
ch <- 2
ch <- 2
close(ch)
for v := range ch {
    fmt.Println(v)
}
```

Select

The select statement lets a goroutine wait on multiple communication operations.

A select blocks until one of its cases can run, then it executes that case. It chooses one at random if multiple are ready.

```
select {
    case translated := <- translate("hello"):
        fmt.Println(translated)
    case <-time.After(80 * time.Millisecond):
        fmt.Println("Too late...")
}
```

play.golang.org/p/yJtKTXBy7Kj (https://play.golang.org/p/yJtKTXBy7Kj)

Select default

Case, when no other case is ready:

```
select {
    case v := <-ch:
    default:
        // no sender for 'ch'
}
```

Select can be used for sending as well!

```
select {
    case ch <- 1:
    default:
        // no receiver is waiting for 'ch'
}
```

This is the non-blocking send.

Exercise 2: Hash

```go
func Hash(path string) ([]byte, error) {
    f, err := os.Open(path)
    if err != nil {
        return nil, err
    }
    defer f.Close()

    h := sha1.New()
    if _, err := io.Copy(h, f); err != nil {
        return nil, err
    }
    return h.Sum(nil), nil
}
```

Exercise 2: Files

```go
func Files() []string {
    var files []string
    flag.Parse()
    for _, path := range flag.Args() {
        // Walk will return no error, because all WalkFunc always returns nil.
        filepath.Walk(path, func(path string, info os.FileInfo, err error) error {
            if err != nil {
                fmt.Printf("ERROR: unable to access %q", path)
                return nil
            }
            if info.Mode()&os.ModeType != 0 {
                return nil // Not a regular file.
            }
            files = append(files, path)
            return nil
        })
    }
    return files
}
```

Exercise 2: parallelize checksum calculation

```go
    // TODO: parallelize the checksum calculation
    for _, path := range Files() {
        hash, err := Hash(path)
        if err != nil {
            fmt.Printf("ERROR: %s", err)
            continue
        }
        fmt.Printf("%x\t%s\n", hash, path)
    }
    // END
```

Exercise 2 ...

Thank you

Ákos Frohner
szamcsi@gmail.com (mailto:szamcsi@gmail.com)