Directory Synchronization using Concurrency in Go

19 March 2017

Zhibin Cai
Technical Overview

This presentation is about...

- Go language
- The **concurrency** feature in Go
- Final Project: Directory Synchronization just like Dropbox

Hello World

```
package main

import (
    "fmt"
)

func main() {
    fmt.Println("Hello World")
}
```

Run

Go Basics

```
func basics() {
    const N = 1024
    const str = "this is a 中文 string\n"

    var x, y *float32
    var ch = 'a'

    // Define and use a type, T.
    type T struct{ a, b int }
    var t0 *T = new(T)
    t1 := new(T) // declare and init

    if len(str) > 0 {
        ch = str[0]
    }
}
```

Methods

```
type Point struct {
    X, Y float64 // Upper cae means exported
}

// Scale the X and Y
func (p *Point) Scale(s float64) {
    p.X *= s
    p.Y *= s
}

// Abs value
func (p *Point) Abs() float64 {
    return math.Sqrt(p.X*p.X + p.Y*p.Y)
}

func main() {
    x := &Point{3, 3} // Get a ptr of an instance
    x.Scale(5)
    fmt.Printf("%f", x.Abs())
}
```

Run

Let's talk about concurrency

It's all about concurrency

- What is concurrency?
- Is concurrency also called parallelism?
- What is the feature of concurrency in Go?
- What is it good for?
- How am I going to use it in the project?

Example of Go concurrent code

Infinite function

```
func blah(msg string) {
    for i := 0; ; i++ { // Runs forever
        fmt.Println(msg, i)
        time.Sleep(time.Duration(rand.Intn(1e3)) * time.Millisecond)
    }
}

func main() {
    blah("Go Orange!")
}
```

Run

Goroutine

The go statement runs the function on the fly in the same address space and does not bloc
the called.
This is a go routine.
It just like you add & at the end of a shell command.
You just don't care.

```
func blah(msg string) {
    for i := 0; ; i++ { // Runs forever
        fmt.Println(msg, i)
        time.Sleep(time.Duration(rand.Intn(1e3)) * time.Millisecond)
    }
}

func main() {
    go blah("Go Orange!") // Launch a go routine
    fmt.Println("Hello there")
    time.Sleep(2 * time.Second) // This is bad code!
    fmt.Println("Time's Up! Game Over")
    // when return, it kills the go routine as well
}
```

Run

Channels

Channel is an approach of communication in go for goroutines to exchange data back and forth.

```
// Declaring and initializing
var c chan int
c = make(chan int)
// or it is equal to
c := make(chan int)

// You send a number to a channel
c <- 1

// Get a value from a channel
value = <- c
```

Using channel to get value

```
func blah(msg string, c chan string) {
    for i := 0; ; i++ { // Runs forever
        c <- fmt.Sprintf("%s %d", msg, i)
        time.Sleep(time.Duration(rand.Intn(1e3)) * time.Millisecond)
    }
}

func main() {
    c := make(chan string)
    go blah("Go Orange!", c)
    for i := 0; i < 5; i++ {
        fmt.Printf("From channel:  %q\n", <-c)
    }
    fmt.Println("Time's Up! Game Over")
    // when return, it kills the go routine as well
}
```

Run

Another Example: ping-pong

```
type Ball struct{ hits int }

func player(name string, table chan *Ball) {
    /* `for` means infinite loop */
    for {
        ball := <-table
        ball.hits++
        fmt.Println(name, ball.hits)
        time.Sleep(100 * time.Millisecond)
        table <- ball
    }
}

func main() {
    table := make(chan *Ball)
    go player("ping", table)
    go player("pong", table)

    table <- new(Ball) // Start the ping-pong game!
    time.Sleep(1 * time.Second)
    <-table // Grab the ball from the table
}
```

Run

Synchronization

When the main function executes `<-c`, it is blocked until a value is sent from that channel. Also in the `blah` function executing `` `c <- value` ``, it is also blocked until someone gets the value from itself.

Both of them are blocking operation.

The Go Approach

Don't communicate by sharing memory, share memory by communicating.

Go Concurrency Patterns

Pattern: Generator (Function that returns a channel)

```
func blah(slogan string) <-chan string {
    c := make(chan string)
    go func() {
        for i := 0; ; i++ {
            c <- fmt.Sprintf("%s %d", slogan, i)
            time.Sleep(time.Duration(rand.Intn(1e3)) * time.Millisecond)
        }
    }()

    return c // Return a channel to the caller
}

func main() {
    c := blah("Go Orange!") // Returns a channel!
    for i := 0; i < 5; i++ {
        fmt.Printf("Slogan: %q\n", <-c)
    }
    fmt.Println("Game over!")
}
```

Run

Generator works like a service

One function that returns a channel can be also regarded as a software service that run some job in background.

```
func main() {
    c := syncDirectory("Image") // Returns a channel
    d := syncDirectory("Music") // Returns another channel
    for i := 0; i < 5; i++ {
        fmt.Printf("Syncing directory: %q\n", <-c) // Image Syncing
        fmt.Printf("Syncing directory: %q\n", <-d) // Music Syncing
    }
    fmt.Println("Sync over...")
}
```

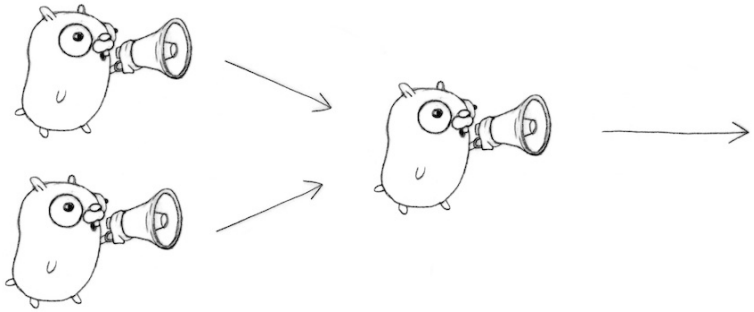
Run

- In this example, we have two independent instances of channel(service).

Fan-in function

```
func fanIn(service1, service2 <-chan string) <-chan string {
    c := make(chan string)
    go func() { for { c <- (<-service1) } }()
    go func() { for { c <- (<-service2) } }()
    return c
}

func main() {
    c := fanIn(syncDirectory("Image"), syncDirectory("Music"))
    for i := 0; i < 20; i++ {
        fmt.Printf("Sync Directory: %q\n", <-c)
    }
    fmt.Println("Sync over!")
}
```

Run



Select

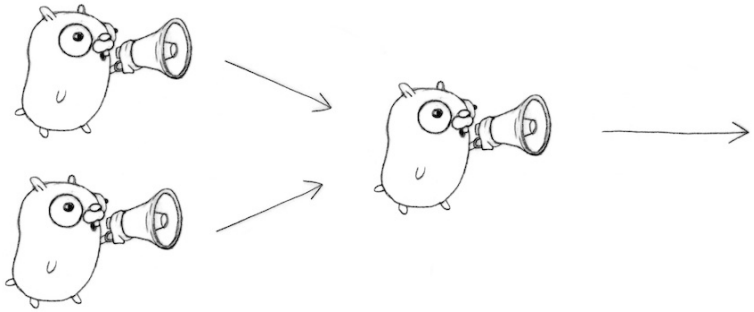
- A **Select** is like a `switch` statement in which the cases are communications.
- A simple example uses a second channel to tear down the server.

```
select {
    case v1 := <-c1:
        fmt.Printf("received %v from c1\n", v1)
    case v2 := <-c2:
        fmt.Printf("received %v from c2\n", v1)
    case c3 <- 23:
        fmt.Printf("sent %v to c3\n", 23)
    default:
        fmt.Printf("no one was ready to communicate\n")
}
```

Fan-in Function example

```
func fanIn(service1, service2 <-chan string) <-chan string {
    c := make(chan string)
    go func() {
        for {
            c <- (<-service1)
        }
    }()
    go func() {
        for {
            c <- (<-service2)
        }
    }()
    return c
}
```

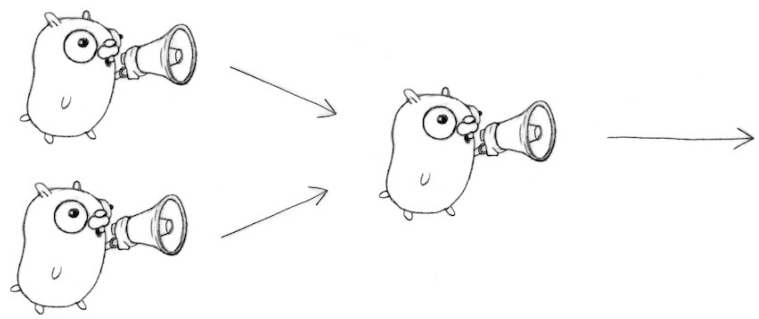
Run



Fan-in Function using Select

```
func fanIn(service1, service2 <-chan string) <-chan string {
    c := make(chan string)
    go func() {
        for {
            select {
            case s := <-service1:
                c <- s
            case s := <-service2:
                c <- s
            }
        }

    }()
    return c
}
```

Run

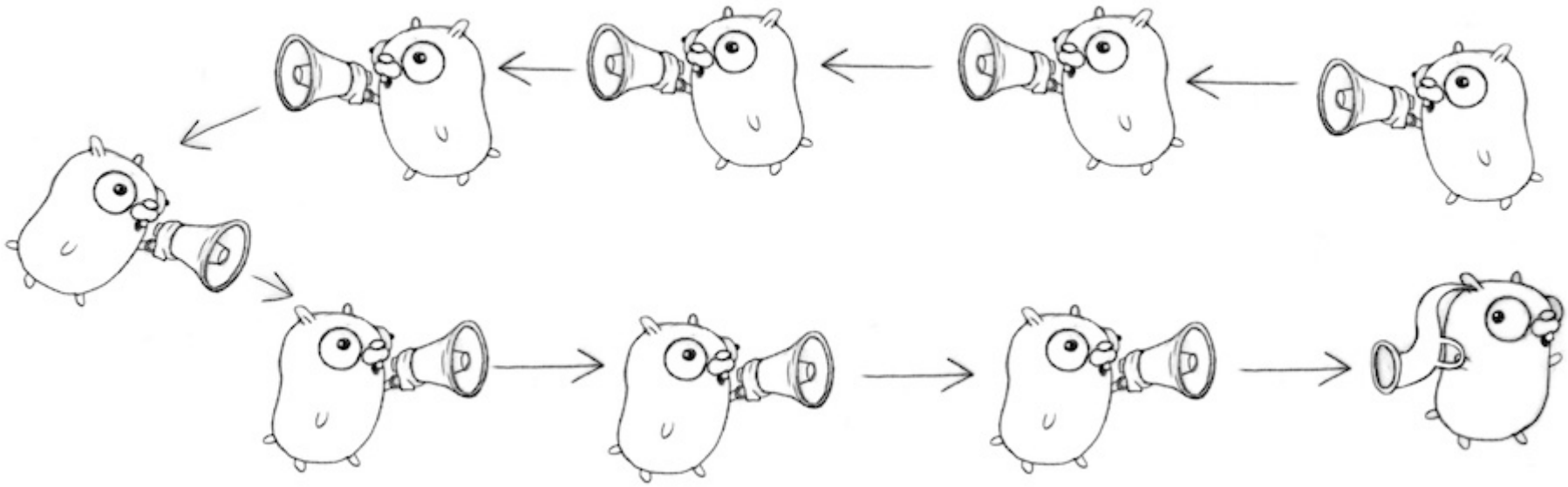


Quit for a server

We can use select and tell the server to stop when there is data sent to the quit channel.

```
func server(op int32, service chan string, quit chan bool) {
    for {
        select {
        case req := <-service:
            go run(op, req)
        case <-quit:
            return
        }
    }
}
```

Channel works just like Chinese Whisper Game



Reference: *Chinese Whisper Game* by Rob Pike

Daisy chain

We can run thousands of goroutines at once...

```
func f(left, right chan int) {
    left <- 1 + <-right
}

func main() {
    const n = 100000 // One hundred thousand goroutine!
    leftmost := make(chan int)
    right := leftmost
    left := leftmost
    for i := 0; i < n; i++ {
        right = make(chan int)
        go f(left, right)
        left = right
    }

    go func(c chan int) { c <- 1 }(right)
    fmt.Println(<-leftmost)
}
```

Run

Reference: *Daisy Chain* by Rob Pike

How to use Go concurrency in Directory Synchronization

Design the architecture first!

- Let's **fake** the services at the beginning.
- We called it `fakeDropbox`.

Fake Dropbox Example

What is it?

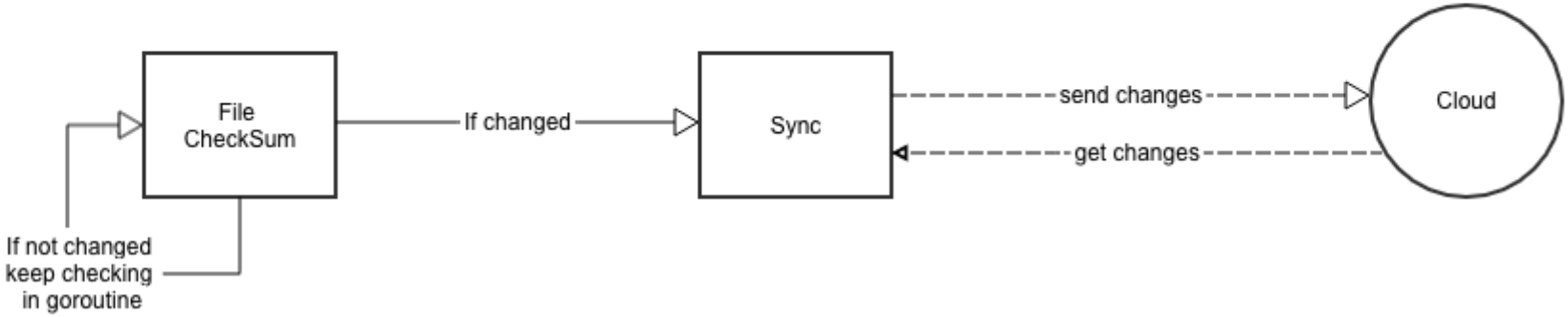
Given a series of files, sync the content with the corresponding files on the cloud.

We only cares about the delta of the files.
- If we are editting the plain text, we send the changes of the files.
- If we are handling the binary files, simply deal with the whole binary file.

Two major actions:

1. `SendChanges("Images")`
2. `GetChanges("Images")`

The process of synchronization



Dropbox sendChanges

We can mock this SendChanges(`directory`) function to **simulate** the sync function.

```
var (
    Directory = fakeSync("directory")
    Files     = fakeSync("files")
    Metadata  = fakeSync("metadata")
)

/* Sync job, which is a function returning the result */
type Sync func(directory string) Result

/* fakeSync - returns the Sync function */
func fakeSync(kind string) Sync {
    return func(directory string) Result {
        time.Sleep(time.Duration(rand.Intn(100)) * time.Millisecond)
        return Result{fmt.Sprintf("%s result for %q", kind, directory)}
    }
}
```

Fake Dropbox service v0.1 beta

This dropbox service only trigger the sync **sequentially**.

```
// Dropbox - A fake dropbox service
func Dropbox(directory string) (results []Result) {
    results = append(results, Directory(directory))
    results = append(results, Files(directory))
    results = append(results, Metadata(directory))
    return results
}
```

```
func main() {
    rand.Seed(time.Now().UnixNano())
    start := time.Now()
    results := Dropbox("~/fakeDropbox")
    elapsed := time.Since(start)
    fmt.Println(results)
    fmt.Println(elapsed)
}
```

Run

Fake Dropbox service v1.0

```
// Dropbox - A fake dropbox service
func Dropbox(directory string) (results []Result) {
    results = append(results, Directory(directory))
    results = append(results, Files(directory))
    results = append(results, Metadata(directory))
    return results
}
```

New version using a fan-in function

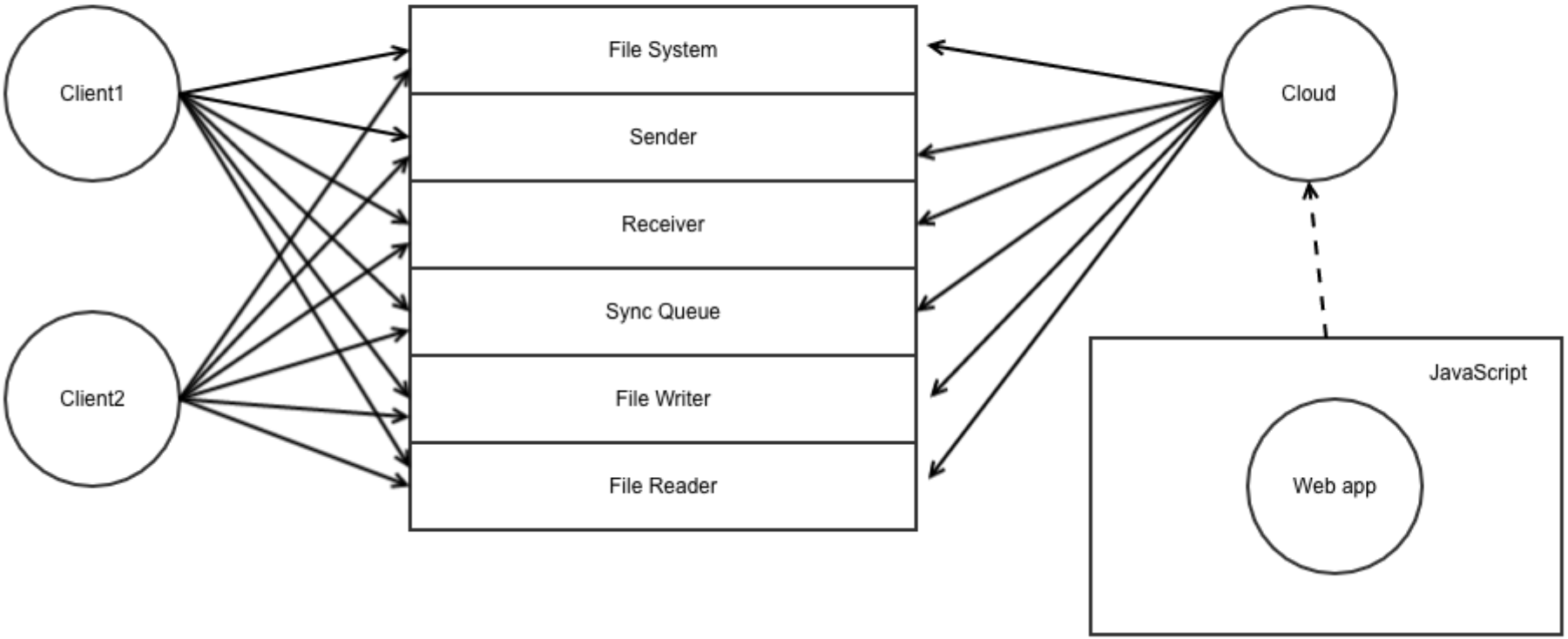
```
// Dropbox - A fake dropbox service
func Dropbox(directory string) (results []Result) {
    c := make(chan Result)
    go func() { c <- Directory(directory) }()
    go func() { c <- Files(directory) }()
    go func() { c <- Metadata(directory) }()

    for i := 0; i < 3; i++ {
        results = append(results, <-c)
    }
    return results
}
```

Run

Using Go with concurrency

- No mutexes / locks. No callbacks
- Cheap goroutines
- Communicate between goroutines using channel like a human

Future module/architecture



Thank you

Zhibin Cai
Technical Overview
zcai108@syr.edu (mailto:zcai108@syr.edu)
zhibincai.com