Golang Training Course

Ngày: 8st June 2021

Outline

Input

- Go Routines
- Reflection

Output

- Hiểu được làm việc với Go Routines/và sử dụng Reflection trong Golang

Content

Go Routines trong Golang

- Go là ngôn ngữ hỗ trợ xử lý Concurrency (Xử lý song song/bất đồng bộ) với Go Routines.
- Go Routines khác với các ngôn ngữ khác như Java/C# như sau:
    - Go Routines sử dụng Go Runtime thay vì OS Runtime để quản lý các tác vụ. Các tác vụ được đưa vào Global Run Queue của Go Runtime, sau đó được Go Scheduler lần lượt mang đi xử lý tại các OS Thread.
    - Đặc điểm của Go Routines là xử lý nhanh, và nhẹ hơn so với các ngôn ngữ khác, đặc biệt là việc sử dụng bộ nhớ của CPU.
    - Khi chương trình Go được khởi tạo, Go Runtime sẽ thực hiện kiểm tra số lượng OS Thread trên mỗi Core của máy, và thực hiện Consume trực tiếp vào các OS Thread này, bởi vậy việc khởi tạo Go Routines khá là rẻ.
- Để làm việc với Go Routines, bạn sử dụng keyword là `go` trước bất kỳ function nào muốn sử dụng.
- Tuy nhiên, Go Routines cần có Cold Start-time (thời gian cần để khởi chạy) để thực hiện chạy, bởi vậy bạn cần cẩn thận sử dụng chúng.
- Go Routines hỗ trợ làm việc với Go Channel, bạn có thể hiểu Go Channel là 1 đường ống (Queue) dùng để chứa thông tin, giúp các Go Routines có thể nói chuyện với nhau. Go Channel hỗ trợ Buffered (Có giới hạn phần tử), Unbuffered (Không giới hạn phần tử).

Các ví dụ sử dụng Go Routines

- Go Routines đơn giản:

```
package main

import (
    "context"
    "time"
    "fmt"
)
```

```
func print(i int) {
    fmt.Println(i)
}

func main() {
    ctx, cancel := context.WithTimeout(context.Background(), 5 *
time.Second)
    defer cancel()

    for i := 0; i < 10;i++ {
        go print(i)
    }

    <- ctx.Done()
}
```

Kết quả trả ra:

```
6
3
5
1
8
0
4
9
2
7
```

> Bạn có thể thắc mắc tại sao kết quả không được in theo thứ tự, khi bạn đang chạy 1 vòng for-loop từ 0 đến 10?
>
> Bạn có thể hiểu như sau: Trong máy tính của bạn có rất nhiều Thread tại mỗi Core, ví dụ bạn có 2 Thread trên mỗi Core, và có 4 Core tổng tất cả. Vậy chúng ta sẽ có 8 Thread để xử lý. Công việc xử lý sẽ do Go Scheduler thực hiện phân bổ tác vụ cho các OS Thread xử lý, nên có thể có các OS Thread xử lý nhanh hơn so với OS Thread khác, do tác vụ được xử lý nhanh hơn. Bởi vậy theo quy tắc FIFO, thì tác vụ nào xong trước thì sẽ trả kết quả trước.

- Sử dụng Go Routines với Go Channel:

```
package main

import (
    "fmt"
    "sync/atomic"
)

func consume(c chan int32, done chan bool, max int32) {
```

```
    var count int32
    for {
        select {
        case i := <-c:
            fmt.Println(i)
            atomic.AddInt32(&count, 1)
            if count == max {
                done <- true
            }
        case <-done:
            return
        }
    }
}

func produce(c chan int32, max int32) {
    var i int32
    for i = 0; i < max; i++ {
        c <- i
    }
}

func main() {
    var (
        max  int32 = 10
        ci         = make(chan int32)
        done       = make(chan bool, 1)
    )

    go produce(ci, max)
    go consume(ci, done, max)

    <-done
}
```

Kết quả được in ra:

```
0
1
2
3
4
5
6
7
8
9
10
```

> Ở ví dụ trên bạn có thể thấy Channel dùng để giao tiếp giữa 2 Go Routines. Một Channel dùng để chứa các phần tử Integer, producer() có nhiệm vụ sản xuất ra các phần tử và đẩy vào Channel để cho consumer() thực hiện lắng nghe và in ra kết quả. Khi đã đến limit, có nghĩa toàn bộ kết quả đã được nhận xong, thì thực hiện thông báo kết thúc để kết thúc chương trình.

- Sử dụng Go Routines với sync.WaitGroup

```
package main

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

func main() {
    var (
        count int32
        i     int32
        wg    = sync.WaitGroup{}
    )

    for i = 0; i < 10; i++ {
        wg.Add(1)
        go func(i int32) {
            defer wg.Done()
            fmt.Println(i)
            atomic.AddInt32(&count, 1)
        }(i)
    }

    wg.Wait()

    fmt.Println(count)
}
```

> WaitGroup được sử dụng trong trường hợp bạn muốn chờ một nhóm Go Routines cần thực hiện xử lý xong để lấy kết quả, và thực hiện Statement tiếp theo, ví dụ bạn thực hiện Counting bất đồng bộ của 10 đầu kết quả khác nhau bởi Go Routines, việc này cần thực hiện xong được thông báo bởi Wait() để thực hiện lệnh tiếp theo. Để định nghĩa có bao nhiêu Go Routines cần phải chờ, bạn dùng Add(), khi đã xử lý xong 1 Go Routines nào đó thì bạn thông báo Done(). Nên dùng kết hợp với defer cho Done()

- Sử dụng Go Routines với sync.Mutex

```
package main

import (
    "fmt"
    "sync"
```

```
)

type calculation struct {
    sum int32
}

func (c *calculation) increase(wg *sync.WaitGroup) {
    defer wg.Done()
    c.sum++
}

func (c *calculation) print() {
    fmt.Println(c.sum)
}

func main() {
    var (
        wg = sync.WaitGroup{}
    )

    c := &calculation{}

    for i := 0; i < 500; i++ {
        wg.Add(1)
        go c.increase(&wg)
    }

    wg.Wait()

    c.print()
}
```

Với chương trình trên, bạn nghĩ rằng kết qủa trả ra sẽ là 500? Thật không may, là không đúng, kết quả có thể ít hơn 500. Tại vì khi chạy một khối lượng lớn Go Routines và việc thay đổi kết quả tại 1 Pointer, có thể sẽ không có kết quả như ý muốn, bởi vì bạn đang thực hiện ghi đè song song tại cùng 1 địa chỉ. Để khắc phục điểm này ta dùng 2 cách:

Cách 1, sử dụng với `atomic`: thay vì `sum++` thì dùng `atomic.AddInt32(&c.sum, 1)` tuy nhiên atomic chỉ sử dụng cho Counter, đối với trường hợp là một kiểu dữ liệu khác, thì dùng ta sử dụng `sync.Mutex`

Sửa lại như sau:

```
package main

import (
    "fmt"
    "sync"
)

type calculation struct {
    sum int32
```

```
    mu  sync.Mutex
}

func (c *calculation) increase(wg *sync.WaitGroup) {
    defer wg.Done()
    c.mu.Lock()
    defer c.mu.Unlock()
    c.sum++
}

func (c *calculation) print() {
    fmt.Println(c.sum)
}

func main() {
    var (
        wg = sync.WaitGroup{}
    )

    c := &calculation{}

    for i := 0; i < 500; i++ {
        wg.Add(1)
        go c.increase(&wg)
    }

    wg.Wait()

    c.print()
}
```

> Đối với chương trình trên, với việc sử dụng Lock() và Unlock() sẽ giúp các Go Routines phải chờ lần lượt để thực hiện ghi kết quả vào 1 giá trị nào đấy thay vì ghi không có kiểm soát. Kết quả trả ra sẽ đúng như mong muốn

- Sử dụng Go Routines với `errgroup`

```
package main

import (
    "errors"
    "fmt"
    "time"

    "golang.org/x/sync/errgroup"
)

func main() {
    eg := &errgroup.Group{}

    eg.Go(func() error {
```

```
        for i := 0; i < 5; i++ {
            fmt.Println(i)
            time.Sleep(1 * time.Second)
        }
        return nil
    })

    eg.Go(func() error {
        time.Sleep(1 * time.Second)
        return errors.New("error cause of something wrong")
    })

    fmt.Println(eg.Wait().Error())
}
```

> errorgroup sử dụng để bắt lỗi khi thực hiện chạy nhiều Go Routines cùng một lúc, ví dụ bạn chạy 2, 3 Go Routines khác nhau, tuy nhiên không may có 1 Go Routines trả kết quả lỗi, sau khi chạy xong toàn bộ Go Routines, thì bạn capture được lỗi và thực hiện tiếp công việc khác

Reflection

Reflection là công cụ cực kì mạnh mẽ để xử lý với `interface{}` trong Go. Một số ví dụ áp dụng như kiểm tra Kind của một dữ liệu là Struct hay các kiểu dữ liệu khác, thực hiện Set giá trị cho một Interface khác từ một Interface đang có.

Ví dụ:

```
type ProjectionOption func(m map[string]interface{}) error

func Projection(i interface{}, opts ...ProjectionOption)
(map[string]interface{}, error) {
    rf := reflect.ValueOf(i).Elem()
    if t := rf.Kind(); t != reflect.Struct {
        return nil, fmt.Errorf("type of s: %s is not a struct",
t.String())
    }

    m := make(map[string]interface{})

    for i := 0; i < rf.NumField(); i++ {
        m[stringutil.ToSnakeCase(rf.Type().Field(i).Name)] =
rf.Field(i).Interface()
    }

    for _, opt := range opts {
        if err := opt(m); err != nil {
            return nil, err
        }
    }

    return m, nil
```

```
}

func WithoutFields(f ...string) ProjectionOption {
    return func(m map[string]interface{}) error {
        for _, v := range f {
            if _, ok := m[v]; !ok {
                return fmt.Errorf("field %s is not found to projection",
v)
            }
            delete(m, v)
        }
        return nil
    }
}
```

> Ví dụ trên dùng Reflection để thực hiện chuyển một Struct ra thành một `map[string]interface{}` tự động (Projection). Function này thực hiện Scan toàn bộ Properties trong Struct để lấy Name sau đó Convert Name sang Snake Case rồi gán giá trị tương ứng.