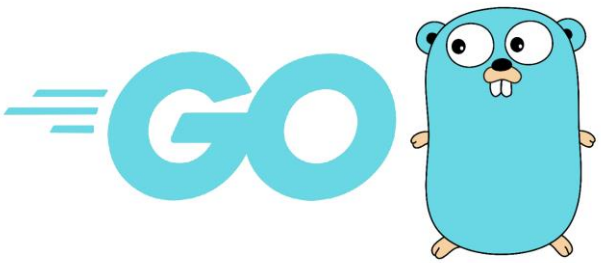
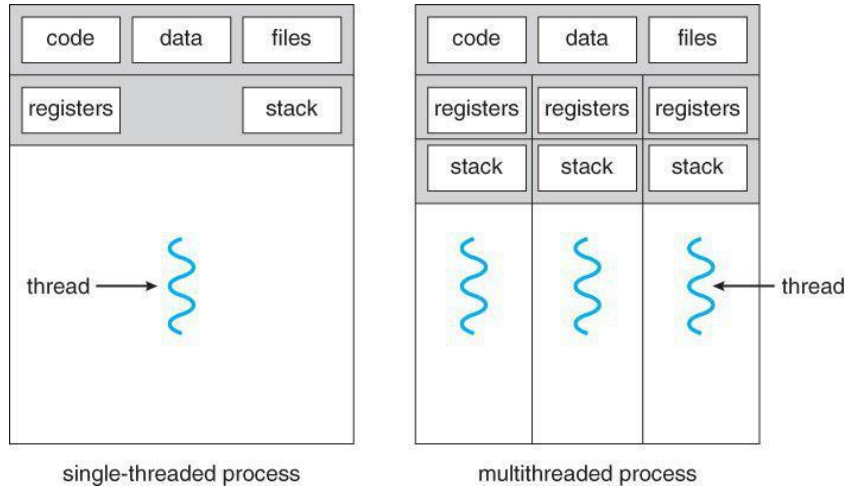
併發



併發

併發(Goroutines)

- 執行緒(線程)(thread)是作業系統能夠進行運算排程的最小單位。大部分情況下，它被包含在行程之中，是行程中的實際運作單位。一條執行緒指的是程序(行程)(Process)中，一個單一順序的控制流，一個行程可以並行多個執行緒，每條執行緒並列執行不同的任務。
- 執行緒是獨立排程和分派的基本單位。一個Process可以向作業系統取得多個執行緒(threads)，多個執行緒可以達到類似平行處理的效果。



併發

併發(Goroutines)

- **Goroutine是輕量級的線程(Thread)。**
- 而main func則是程式當前的、主要的goroutine。想要讓Go併發非常的容易，只要寫好一個func，**在呼叫時多一個go關鍵字即可。**

```go
package main

import (
    "fmt"
    "time"
)

func main() {
    go test()
}

func test() {
    time.Sleep(time.Second * 1)
    fmt.Println("嚇嚇叫")
}
```

奇怪，怎麼沒有輸出？ 原因是主程式func main確實已經執行完畢，但是被go出去的func還在time.Sleep，而且睡著了(一秒鐘)，所以被go出去的func睡飽發現main主線程已經早已結束返回，自然看不見fmt.Print的輸出了。

稍微改一下，給主線程睡一會，這樣就可以了。

```go
func main() {
    go test()
    time.Sleep(time.Second * 1)
}

func test() {
    fmt.Println("嚇嚇叫")
}

/* result:
嚇嚇叫
*/
```

https://go.dev/play/p/4PjHbBSr8HX

併發

Go併發過程

- GO的併發會用到多個核心下去執行，試著執行以下的程式看看，輸出是一段一段的，一下 O 一下 - 交錯著，代表兩邊的線程都很努力的想把值Print出來。

```
func main() {
    go print1()
    go print2()
    time.Sleep(time.Second)
}

func print1() {
    for i := 0; i < 1000; i++ {
        fmt.Print("O")
    }
}

func print2() {
    for i := 0; i < 1000; i++ {
        fmt.Print("-")
    }
}

/* result:
OOOOOOOOOOOOOOOOOOOOOOOOO---------OO---------
---OOOOOOOOOO...
*/
```

https://play.golang.org/p/Y29gVHLFTUC

併發

Go併發過程

- 在這邊可以看到 Print 裡面的值，只印出了幾個就中斷了。
- 原因是，任何情況下，**只要main thread(主線程)中斷或結束，所有的子線程(goroutine)都會中斷**

```go
package main

import (
    "fmt"
    "time"
)

func Print() {
    for i := 0; i < 10; i++ {
        fmt.Println(i)
        time.Sleep(1 * time.Second)
    }
}

func main() {

    go Print()
    time.Sleep(1 * time.Second)
    fmt.Println("Hello world")

}
```

https://play.golang.org/p/7AHRUQrU9CL

併發

runtime.GOMAXPROCS

- runtime.GOMAXPROCS(n) 這一參數限制程式執行時 CPU用到的最大核心數量。如果設置小於1，等於沒設，預設值是電腦核心數。
- 但限制一核心之後，為什麼還是可以把兩個print func都印到呢，怎麼不是只印出一個直到時間到？說好的單核心？原來是Go Routine會去排程，執行A線程一小段時間後會跳到線程B去，這才公平合理嘛！不然CPU資源都被其中一個線程給佔住，作業系統就卡死啦。所以看到的輸出會是O很多，再來 - 很多，兩者都連續印很多的情況下交錯著。

https://play.golang.org/p/idAyE6AsCJ1

併發

搭配 sync.WaitGroup

- 在前面例子，用讓程式碼睡一下的方法，使得併發得以實現，但在實務上，比較少直接使用這個方法，這裏要介紹另外一個方法，是使用 sync 套件裡的方法 WaitGroup，來實現併發：

- 在程式碼尾端加上wg.Wait()，達成一些條件，才可以往後執行，而這個條件，就是收到wg.Done()的呼叫次數，而這個次數，即是 wg.Add(5) 裡的數字 5，取決於要執行幾個 goroutine，請確保這個數字要正確。

```go
package main

import (
    "fmt"
    "sync"
)

var wg sync.WaitGroup

func count(s string) {
    fmt.Println(s)
    wg.Done()
}

func main() {
    wg.Add(5)
    go count("1")
    go count("2")
    go count("3")
    go count("4")
    go count("5")
    wg.Wait()
}
```

https://go.dev/play/p/MTqvyifjTLi

併發

搭配 sync.WaitGroup

- Add() `wg.Add(2)`
  - 增加WaitGroup裡可以容納幾個Thread
- Wait() `wg.Wait()`
  - 等待WaitGroup裡的Thread執行完畢再繼續進行
- Done() 或是 Add(-1) `wg.Done()` `wg.Add(-1)`
  - 使WaitGroup的容量-1

```go
package main

import (
    "fmt"
    "sync"
)

func foo() {
    for i := 1; i <= 10; i++ {
        fmt.Println("Foo: ", i)
    }
    wg.Done()
}

func bar() {
    for i := 1; i <= 10; i++ {
        fmt.Println("Foo: ", i)
    }
    wg.Done()
}

var wg sync.WaitGroup

func main() {
    wg.Add(2)
    go foo()
    go bar()
    wg.Wait()
}
```

https://play.golang.org/p/K7vwXs0UnV

併發

錯誤恐慌(Panic)

- 錯誤恐慌(Panic) 跟退出程序是什麼關係？
- Panic 是發生了預期之外的事情，導致異常、錯誤的產生，退出程序的同時回傳錯誤代碼2 (Process finished with exit code 2)。可以透過panic的func來主動引起錯誤發生。要注意的是若在併發線程中發生了panic，也會導致主程式也異常結束。

```go
func main() {
    fmt.Println("程式開始")

    go p()
    time.Sleep(time.Second * 1)
    fmt.Println("主程式順利結束")
}

func p() {
    fmt.Println("即將發生空難...")
    panic("空難")
}
```

```
/* result:
程式開始
即將發生空難...
panic: 空難

goroutine 18 [running]:
main.p()
    /tmp/sandbox810050981/prog.go:18 +0x95
created by main.main
    /tmp/sandbox810050981/prog.go:11 +0x91
*/
```

併發

匿名併發

- 匿名併發 = 匿名函式加上go併發

```go
func main() {
    go func() { // 把這裡的go併發去掉，就變成匿名函式了
        for i := 0; i < 10000; i++ {
            fmt.Print("匿名併發函式 ")
        }
    }() // 這邊有小括號有點奇怪對不？ 因為他是一個函式，在呼叫時需要()

    for i := 0; i < 10000; i++ {
        fmt.Print("main ")
    }

    time.Sleep(10000)
}
```

不一定每次都能期待被併發出去的func順利執行完任務並且會傳值回來，人家主線程main func早就打烊、結束營業啦，就算回傳了也沒辦法接到。

所以說，被go出去的func 沒有return回傳值，因為不能期待每次都能成功回傳值、不能期待主線程每次都等他們完成。

那要回傳值給main func知道，該怎麼辦呢？此時就需要自己建立**通道(channel)**了。

https://play.golang.org/p/yjlln9Th00a

併發

通道(Channel)

- 因為被併發出去的func不會回傳值，若想要在併發線程之間交流、傳遞資料、發送訊息，就必須要使用通道(Channel)。
- 通道分成兩種：有Buffer 跟 無Buffer
  - **有儲存空間限制的通道 vs 無限制的通道**
- 通道也能分成另外兩種：單向 跟 雙向
  - **只有一方能傳資料 vs 主動call方 (caller) 與 被call方 (callee)都能傳資料**
  - **通道預設是雙向的，除非將它變成單向道。**
- 通道也是有型別 (Type) 之分的，**建立通道時記得要加**。

併發

通道(Channel)

- 透過channel、全域變數進行溝通：

```
package main

import (
        "fmt"
        "time"
)

func say(s string, val  chan int) {
        for i := 0; i < 2; i++ {
                time.Sleep(100 * time.Millisecond)
                 fmt.Printf("say %s \n" ,s)

        }
val <- 1 //注入資料1
}
func say2(s string, val  chan int) {
        for i := 0; i < 2; i++ {
                time.Sleep(100 * time.Millisecond)
                fmt.Printf("say2 %s \n" ,s)

        }
val <- 2 //注入資料1
}

func main() {
val := make(chan int)
        go say("a", val )
        go say2("a", val )
    x := <-val  // receive from c

    fmt.Println(x)
}
```

https://go.dev/play/p/1XMUH55z2hj

併發

通道(Channel)

- 接收 channel 的值，不一定要真的有變數去接
  - 好比函式的回傳值不一定要有變數去接一樣，接收 channel 的值時不一定真的要有變數去接

```go
package main
import "fmt"

func foo(myChannel chan string){
    fmt.Println("呼叫 foo()")
    myChannel <- "封印解除！！"
}

func main(){
    myChannel := make(chan string)
    go foo(myChannel)
    // 等待接收 channel
    // 不一定要有變數去接球
    <- myChannel
}
```

執行結果：

呼叫 foo()

https://go.dev/play/p/Z8uHJPY39PR

併發

通道(Channel)

```
Variable := make(chan Type)

    c := make(chan int)
```

- 無緩衝的通道(unbuffered channel)
  - 通道(chan)製造出來之後，需要傳進要被併發出去的func，靠這個傳資料的。chan是有方向性的，要看箭頭<-的方向。

```
chan <- A `把 A這個東西 塞進chan`
B<- chan  `從chan 挖東西出來 到B`
```

  - 由以下兩個例子來說明接、收方向性：

```go
func main() {
    ch := make(chan int)
    go func1(ch)
    ch <- 100
}

func func1(ch chan int) {
    i := <-ch
    fmt.Println(i)
}
```

```go
func main() {
    ch := make(chan int)
    go func2(ch)
    got := <-ch
    fmt.Println(got)
}

func func2(ch chan int) {
    time.Sleep(time.Second * 2)
    ch <- 999
}
```

https://play.golang.org/p/KgZXzvNr4iP

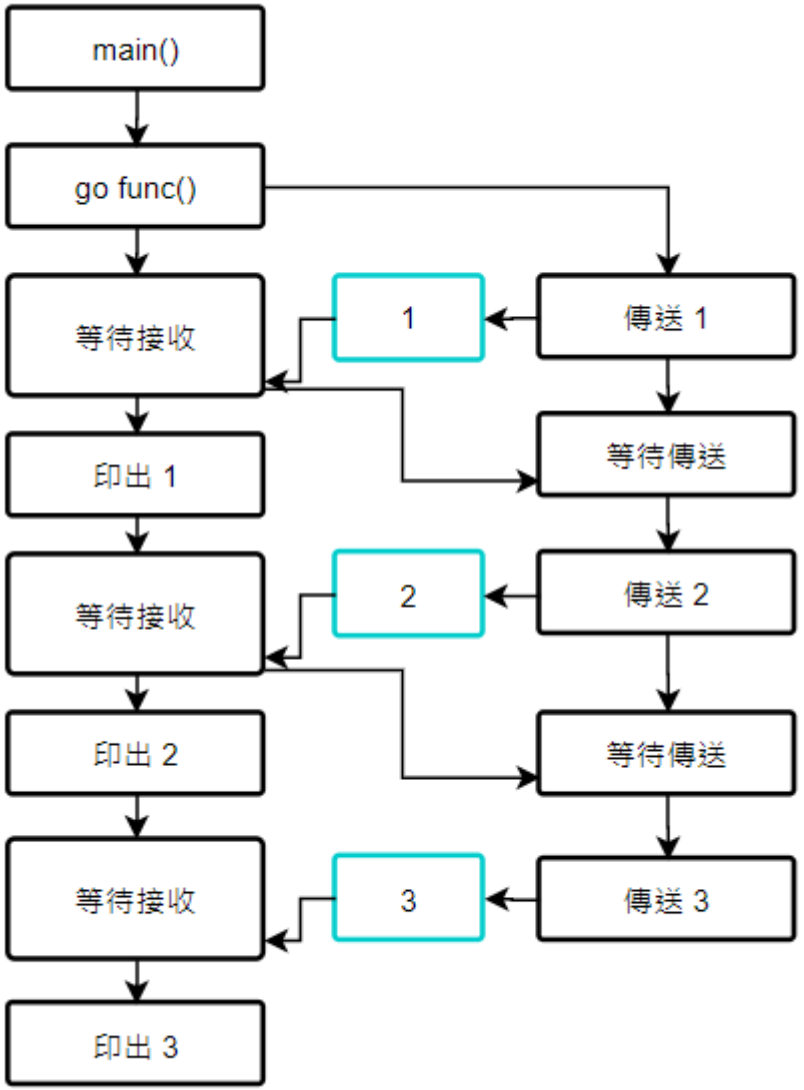
併發

通道(Channel)

- 無緩衝的通道(unbuffered channel)

```go
package main
import "fmt"

func main(){
    no_buffer_chan := make(chan int)
    go func(){
        fmt.Println("傳送1")
        no_buffer_chan <- 1
        fmt.Println("傳送2")
        no_buffer_chan <- 2
        fmt.Println("傳送3")
        no_buffer_chan <- 3
    }()
    fmt.Println("等待接收1")
    fmt.Println(<- no_buffer_chan)
    fmt.Println("等待接收2")
    fmt.Println(<- no_buffer_chan)
    fmt.Println("等待接收3")
    fmt.Println(<- no_buffer_chan)
}
```

執行結果：
等待接收1
傳送1
傳送2
等待接收2
等待接收3
傳送3



每次執行結果會不太一樣

因為對於一個沒有 buffer 的 channel ，channel 上面一次只能有一個值，

前一個傳完，才能傳下一個，不能允許一次上傳超過一個

https://go.dev/play/p/I5V1TK-1V0_e

併發

通道(Channel)

```
Variable := make(chan Type, Number)

c := make(chan int, 2)
```

- 有空間限制的通道(buffered channel)
  - 沒有 buffer 的 channel 是**一個傳送到達才能再送下一個**，如果要避免這種浪費時間的情況，可以選擇使用帶有 buffer 的 channel 傳送
  - 有限制儲存空間的通道，若限制放兩個，就只能有兩個。此時又塞第三個進去會**死結(deadlock)**。

```go
func main() {
    ch := make(chan int, 2)
    go func3(ch)
    ch <- 100
    ch <- 99

    ch <- 98 // 發生deadlock
}

func func3(ch chan int) {
}

/* result:
fatal error: all goroutines are asleep - deadlock!
*/
```

阻塞(Block) vs 死結(Deadlock)

通常chan塞不下時，只會發生Block(阻塞滯留)，而當Block永遠無法解開的情況發生，則是 Deadlock(死結)。會發生死結是因為不論等多久，都不會從Block的狀態中脫離。只要通道(Chan)塞不下，都會發生Block阻塞。

https://play.golang.org/p/AxO5Xd7tGrU

併發

通道(Channel)

- 有空間限制的通道(buffered channel)
  - 這次改以有 buffer 的 channel 實作。宣告有 buffer 的 channel 時，要**在第二個參數加上 buffer 的大小即可**

```go
package main
import "fmt"

func main(){
    buffer_chan := make(chan int, 3)
    go func(){
        fmt.Println("傳送1")
        buffer_chan <- 1
        fmt.Println("傳送2")
        buffer_chan <- 2
        fmt.Println("傳送3")
        buffer_chan <- 3
    }()
    fmt.Println("等待接收1")
    <- buffer_chan
    fmt.Println("等待接收2")
    <- buffer_chan
    fmt.Println("等待接收3")
    <- buffer_chan
}
```

執行結果

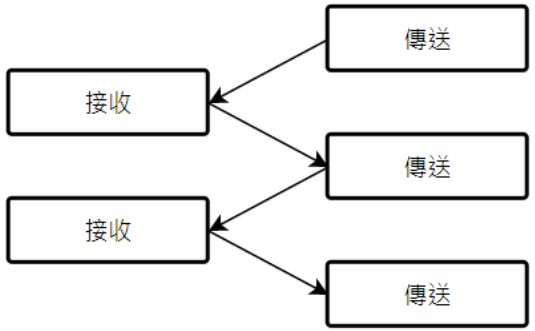
```
傳送1
傳送2
傳送3
等待接收1
等待接收2
等待接收3
```

https://go.dev/play/p/UvmWZvt-pxe

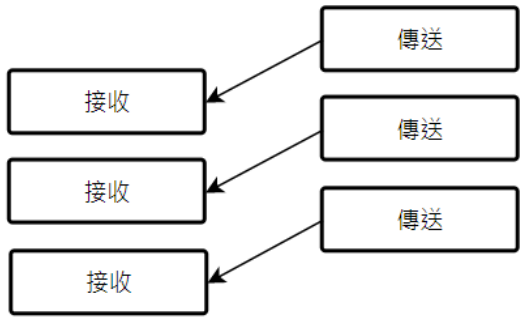
併發

通道(Channel)

- 沒有 buffer 的 channel



一次有 3 個 buffer 的 channel



- 如果是有buffer的channel那麼就不必在意對方是否接收到，只要在意是否仍有足夠的緩衝空間，如果沒有空間了才會進行等待。

併發

通道(Channel)

- **無緩衝通道不等於無限通道**
  - 剛開始時容易搞混，**無緩衝通道(Unbuffered) 並不等於 **無限制(Unlimited)**的通道**。
  - Buffer 是拿來緩衝用的，Unbuffered Channel則是0緩衝，就是沒有緩衝！Unbuffered 是**需要有同時有一頭寫入、另一頭讀出**才能動。那Golang有沒有無限制的通道(Unlimited)呢？答案是：沒有 。
  - 至於為什麼沒有？如果給1000個Byte緩衝的通道，代表程式執行時要預先挪空出1000個Byte個空間。那如果今天是1000000個呢、甚至無限個呢？就爆炸了。
  - 讀寫速率要控制在一定的範圍內，Channel中的緩衝區塊才能起到作用。若寫入通道中的速度永遠大於讀取速度，那麼給再多再大的倉庫放，永遠都會有不夠放的一天。但是若是實作上真的有需求，可以透過一些trick的手段達成、模擬無限通道這件事，**例如：使用 slice 來記錄通道中的東西**。

併發

通道(Channel)

- 效能分析(有緩衝及無緩衝)
  - 因為無緩衝的 channel 一次只能傳送一個值，所以會花很多時間做等待：

```go
package main
import (
    "fmt"
    "time"
)

func main(){
    // 無緩衝
    start := time.Now().UnixNano()
    channel := make(chan int)
    go func(){
        for i := 0; i < 10000; i = i+1{
            channel <- i
        }
        close(channel)
    }()
    for _ = range channel {
    }
    fmt.Println((time.Now().UnixNano() - start), "ns")

    ////////////////////////////////////

    // 有緩衝
    start = time.Now().UnixNano()
    buffer_channel := make(chan int, 100)
    go func(){
        for i := 0; i < 10000; i = i+1{
            buffer_channel <- i
        }
        close(buffer_channel)
    }()
    for _ = range buffer_channel {
    }
    fmt.Println((time.Now().UnixNano() - start), "ns")
}
```

執行結果：

3987500 ns

1093700 ns

https://go.dev/play/p/kwA1B2tYv6r

併發

通道(Channel)

- Block阻塞
  - 以下是通道Channel 阻塞Block的例子：

```go
func main() {
    ch := make(chan int, 2)
    go func4(ch)
    for i := 0; i < 10; i++ {
        ch <- i
        fmt.Println("main sent", i)
    }
    time.Sleep(time.Second)
}

func func4(ch chan int) {
    for {
        i := <-ch
        fmt.Println("func got", i)
        time.Sleep(time.Millisecond * 100)
    }
}
```

```
/* result:
func got 0
main sent 0
main sent 1
main sent 2
func got 1
...
...
*/
```

主程式不間斷地連續塞十次數字 送完休息1秒；

而func4每0.1秒處理一個數值。雖然慢，但程式不會打死結。

https://play.golang.org/p/9N-B6QFu1Q_v

併發

通道(Channel)

- Block阻塞
  - 如果把Buffer Size: 2換成5 會發生什麼事情？ `ch := make(chan int, 5)`
  - 同時間通道裡最多會有五個數字。
  - 寫入與讀取的先後順序，透過log.SetFlags(5)來看會比較清楚。

```go
package main

import (
	"log"
	"time"
)

func main() {
	log.SetFlags(5)
	ch := make(chan int, 5)
	go func4(ch)
	for i := 0; i < 10; i++ {
		ch <- i
		log.Println("main sent", i)
	}
	time.Sleep(time.Second)
}

func func4(ch chan int) {
	for {
		i := <-ch
		log.Println("func got", i)
		time.Sleep(time.Millisecond * 100)
	}
}
```

https://play.golang.org/p/MzL_-cj0N6h

併發

通道(Channel)

- 接收不確定個數的 channel
  - 因為有時候無法確定究竟有幾個值會傳送到 channel 上，所以接收端很難判定(可以用無緩衝channel實作，但很不直觀)。因此可以利用 for + range 來接收 channel 上的值，但是要注意一點，**傳送最後一個值後必需利用 close() 將 channel 關閉，golang 才知道這是最後一個值**：

```go
package main
import "fmt"

func main(){
    buffer_chan := make(chan int, 3)
    go func(){
        for i := 0; i < 10; i = i+1{
            buffer_chan <- i
        }
        close(buffer_chan)
    }()
    for k := range buffer_chan {
        fmt.Println(k)
    }
}
```

```
執行結果：
0
1
2
3
4
5
6
7
8
9
```

https://go.dev/play/p/L7NAA9WF2c9

併發

通道(Channel)

- 競爭危害(Race Condition)
  - 爭奪變數
    - 有了go這個好用的東西可以分成好幾個Thread同時進行，那就有可能會有同時搶奪資源的情況。比方說兩個Thread同時都要讀取並修改同一個變數，拿一個情境題來舉例：
      - 如果今天同一個銀行的網路銀行有提款功能，而有人同時在兩台電腦都登入了要提款，兩台電腦都送出了提領1000的請求會怎樣呢？
      - 到最後大家會發現1500提領了兩次1000還剩500？為甚麼呢？
      - 因為在第一次提領的時候，系統先讀取餘額為1500元，同時第二台電腦也登入了餘額也是顯示為1500元，這時候就是因為兩邊同時搶著讀取餘額的原因，所以第一次提領1000元時回報給系統"餘額剩500元"，第二台領了1000元也回報系統"餘額剩500元"

https://go.dev/play/p/2ph8O-axJ8q

```go
package main

import (
    "fmt"
    "sync"
    "time"
)

func withdraw() {
    balance := money
    time.Sleep(3000 * time.Millisecond)
    balance -= 1000
    money = balance
    fmt.Println("After withdrawing $1000, balace: ", money)
    wg.Done()
}

var wg sync.WaitGroup
var money int = 1500

func main() {
    fmt.Println("We have $1500")
    wg.Add(2)
    go withdraw() // first withdraw
    go withdraw() // second withdraw
    wg.Wait()
}
```

併發

通道(Channel)

- 競爭危害(Race Condition)
  - 爭奪變數
    - 在這個例子中使用了10000個被併發出去的func，每個func只做一件事：count++

```go
var count = 0

func main() {
    // runtime.GOMAXPROCS(1)  // 只讓一個線程運作就能解決問題。但...想要更快，就是要多核心嘛！單核心怎能呈爆？
    for i := 0; i < 10000; i++ {
        go race()
    }
    time.Sleep(time.Millisecond * 100)
    fmt.Println(count)
}

func race() {
    count++
}
```

```go
/* result:
9763
*/
```

輸出居然不是10000？

但如果數字小一點，例如10，就又恢復正常了？

這就是爭奪資源的關係。

- 不過Golang的編譯器可以檢查是不是有race condition，只要在平常執行go run後面加上參數-race即可：`go run -race main.go`

https://go.dev/play/p/EtJ6rge6dkS

併發

通道(Channel)

- 競爭危害(Race Condition)
  - 互斥鎖(Lock)-sync.Mutex
    - Mutex可以解決上面這樣的問題。會遇到上面問題，是由於同時對變數進行讀寫(Read/Write)的關係，

```go
var count = 0
var m sync.Mutex

func main() {
    for i := 0; i < 10000; i++ {
        go race()
    }
    time.Sleep(time.Millisecond * 100)
    fmt.Println(count)
}

func race() {
    m.Lock()
    count++
    m.Unlock()
}
```

```go
/* result:
10000
*/
```

只要在變數前上鎖(Lock)，在解鎖(Unlock)前 只有該線程能對其進行操作。

https://go.dev/play/p/bdF3hmg9JK7

併發

通道(Channel)

- 競爭危害(Race Condition)
  - 互斥鎖(Lock)-sync.Mutex
    - Lock() `mu.Lock()`
      - 當使用mu.Lock()的時候，之後所用到的變數就會上鎖，只有在使用中的Thread可以存取，其他都需要等到釋放後才能存取
    - Unlock() `mu.Unlock()`
      - 釋放lock的變數
    - 因為Lock()和Unlock()通常都會一起出現，所以有些人會這樣寫：

```
func withdraw() {
    {
        mu.Lock()
        balance := money
        time.Sleep(3000 * time.Millisecond)
        balance -= 1000
        money = balance
        mu.Unlock()
    }
    fmt.Println("After withdrawing $1000, balace: ", money)
    wg.Done()
}
```

https://play.golang.org/p/AXptlC-RXZ

併發

通道(Channel)

- 競爭危害(Race Condition)
  - 死結 (DeadLock)
    - channel如果沒有宣告buffer， sender端送進去的訊息，receiver 端沒收進來之前，sender端送不進去第二則訊息，此時，sender端會block，在這邊要注意控制這個問題，以避免造成 deadlock。

```go
package main

import (
    "fmt"
)

var message chan string

func Bot() {
    msg := <-message
    fmt.Printf("Bot Print:%s\n", msg)
}

func main() {

    message = make(chan string)

    go Bot()

    message <- "first message"
    fmt.Println("first message send finish")
    message <- "second message"
    fmt.Println("second message send finish")

    fmt.Println("end")
}
```

https://play.golang.org/p/O_DjamN8sss

併發

通道(Channel)

- 競爭危害(Race Condition)
  - 死結 (DeadLock)
    - 如何在沒宣告 buffer 的情況下，排除上面的 deadlock：

```go
package main

import (
    "fmt"
    "time"
)

var message chan string

func Bot() {
    //讓bot 不斷接收訊息，就不會造成 main thread 卡死
    for msg := range message {
        fmt.Printf("Bot Print:%s\n", msg)
    }
}

func main() {

    message = make(chan string)

    go Bot()

    message <- "first message"
    fmt.Println("first message send finish")
    message <- "second message"
    fmt.Println("second message send finish")

    //加入Sleep 是為了避免 main thread提前結束，而看不到 bot 後續印的值
    time.Sleep(1*time.Second)
    fmt.Println("end")
}
```

https://play.golang.org/p/OSzlVjXGMlj

併發

通道(Channel)

- 執行緒安全(Thread-Safe)
  - 所謂的「執行緒安全」就是指：某個函數、函數庫在多執行緒環境中被使用時，能夠正確地處理多個執行緒之間的共享變數。
  - Map 並不是執行緒安全，存取 Map 時僅限於同一個執行緒，若有若干個執行緒去存取 Map 則有機率會出錯！
  - 因為出現錯誤是有機率的，所以如果沒有出現錯誤，可以多試幾次。

```
執行結果：
map[小可:3 小櫻:1 小狼:0 桃矢:4 歌帆:5 知世:2]
fatal error: concurrent map writes

goroutine 16 [running]:
runtime.throw(0x4c78d0, 0x15)
C:/Go/src/runtime/panic.go:617 +0x79 fp=0xc00009ff30 sp=0xc00009ff00
pc=0x42b0b9
runtime.mapassign_faststr(0x4a9d80, 0xc00006e360, 0x4c4af6, 0x6, 0x0)
C:/Go/src/runtime/map_faststr.go:211 +0x431 fp=0xc00009ff98 sp=0xc00009ff30
pc=0x410541
main.test.func6(0xc00006e360, 0xc000018150)
C:/Users/liao2/OneDrive/go-tutorial/lesson18c.go:34 +0x53 fp=0xc00009ffd0
sp=0xc00009ff98 pc=0x491693
runtime.goexit()
C:/Go/src/runtime/asm_amd64.s:1337 +0x1 fp=0xc00009ffd8 sp=0xc00009ffd0
pc=0x451d21
created by main.test
C:/Users/liao2/OneDrive/go-tutorial/lesson18c.go:33 +0x15c

goroutine 1 [chan receive]:
main.test()
C:/Users/liao2/OneDrive/go-tutorial/lesson18c.go:40 +0x17c
main.main()
C:/Users/liao2/OneDrive/go-tutorial/lesson18c.go:48 +0x31
exit status 2
```

```go
package main
import "fmt"

func test(){
    m := make(map[string]int)
    done := make(chan bool, 6)

    go func(){
        m["小狼"] = 0
        done <- true
    }()

    go func(){
        m["小櫻"] = 1
        done <- true
    }()

    go func(){
        m["知世"] = 2
        done <- true
    }()

    go func(){
        m["小可"] = 3
        done <- true
    }()

    go func(){
        m["桃矢"] = 4
        done <- true
    }()

    go func(){
        m["歌帆"] = 5
        done <- true
    }()

    // 等待執行緒結束
    for i := 0; i < 6; i = i + 1{
        <- done
    }

    fmt.Println(m)
}

func main(){
    for i := 0; i < 10; i = i+1{
        test()
    }
}
```

併發

通道(Channel)

- 執行緒安全(Thread-Safe)
  - 使用執行緒安全的 Map：在使用同一個執行緒時搭配 hashmap 使用有較好的效能體驗，然後因為 hashmap 為執行緒不安全，所以如果要跨執行緒使用，則會選擇 hashtable：
  - 每次執行結果都稍有差異

```
執行結果：
map[歌帆:5 小狼:0 小櫻:1 知世:2 小可:3 桃矢:4 ]
map[小櫻:1 知世:2 小可:3 桃矢:4 小狼:0 歌帆:5 ]
map[小狼:0 小櫻:1 知世:2 小可:3 桃矢:4 歌帆:5 ]
map[歌帆:5 小狼:0 小櫻:1 知世:2 小可:3 桃矢:4 ]
map[小可:3 桃矢:4 歌帆:5 小狼:0 小櫻:1 知世:2 ]
map[歌帆:5 小狼:0 小櫻:1 知世:2 小可:3 桃矢:4 ]
map[小可:3 桃矢:4 歌帆:5 小狼:0 小櫻:1 知世:2 ]
map[知世:2 小可:3 桃矢:4 歌帆:5 小狼:0 小櫻:1 ]
map[歌帆:5 小狼:0 小櫻:1 知世:2 小可:3 桃矢:4 ]
map[歌帆:5 小狼:0 小櫻:1 桃矢:4 知世:2 小可:3 ]
```

```go
package main
import(
    "fmt"
    "sync"
)

func test(){
    m := new(sync.Map)
    done := make(chan bool, 6)

    go func(){
        m.LoadOrStore("小狼", 0)
        done <- true
    }()

    go func(){
        m.LoadOrStore("小櫻", 1)
        done <- true
    }()

    go func(){
        m.LoadOrStore("知世", 2)
        done <- true
    }()

    go func(){
        m.LoadOrStore("小可", 3)
        done <- true
    }()

    go func(){
        m.LoadOrStore("桃矢", 4)
        done <- true
    }()

    go func(){
        m.LoadOrStore("歌帆", 5)
        done <- true
    }()

    // 等待執行緒結束
    for i := 0; i < 6; i = i + 1{
        <- done
    }

    fmt.Print("map[")
    m.Range(func(key, value interface{}) bool{
        fmt.Printf("%s:%d ",key, value)
        return true
    })
    fmt.Println("]")
}

func main(){
    for i := 0; i < 10; i = i+1{
        test()
    }
}
```

併發

通道(Channel)

- select
  - select 在 golang 裡面是一個跟 switch 很像，但又只專屬用在 channel 的一個功能
  - **select 功能其實蠻簡單的，如果有多條 channel 要在一個線程，去控管它的輸出，這時侯就是運用 select 的好時機**
  - 要注意 select 到底會進入哪一條 case，外在是無法控制的，這是由 golang 內部演算法實作，所以如果 case 的情況多，的確有機會不會進到某 case 裡面。所以請斟酌使用方式。

https://play.golang.org/p/-scW4ZbLEa5

```go
package main

import (
    "fmt"
)

var sender1Channel chan string
var sender2Channel chan string

func Sender1() {
    sender1Channel <- "this is sender1"

}
func Sender2() {
    sender2Channel <- "this is sender2"
}

func main() {
    sender1Channel = make(chan string)
    sender2Channel = make(chan string)
    go Sender1()
    go Sender2()
    i := 0

    for {
        select {
            case msg := <-sender1Channel:
                fmt.Println(msg)
            case msg := <-sender2Channel:
                fmt.Println(msg)
        }

        //這邊因為只有兩個 sender 各送一條訊息,
        //所以這個loop 只要執行兩次就可以 break, 因為不會再有其他訊息進來

        i = i + 1
        if i >= 2 {
            break
        }

    }
    fmt.Println("end")
}
```

併發

通道(Channel)

- select
  - Select 算是監聽，左圖執行的時候，其實是會隨機亂跳的，如果把ch <- 1註解起來，則是會報錯"deadlock!"，這時候如果把程式碼修改為右圖：

```go
package main

import "fmt"

func main() {
    ch := make(chan int, 1)

    ch <- 1
    select {
        case <-ch:
            fmt.Println("random 01")
        case <-ch:
            fmt.Println("random 02")
    }
}
```

```go
package main

import "fmt"

func main() {
    ch := make(chan int, 1)

    select {
        case <-ch:
            fmt.Println("random 01")
        case <-ch:
            fmt.Println("random 02")
        default:
            fmt.Println("exit")
    }
}
```

  - 這樣就沒事情了，因為在有default的情況下會去執行
  - Select基本上只能應用於channel上，以範例來看是可以做channel的資料接收及溝通使用。**如果有滿足多個條件則是隨機選擇**，但在switch-case這點就比較不同，就跟其他語言雷同，是看順序的。

https://go.dev/play/p/j2i0TI3OoLO

https://go.dev/play/p/hFbCu0y-7CO

併發

通道(Channel)

- select：利用 select default 的特性來得知 buffered channel 是否已滿
  - buffered channel 在 channel 無空位的情況下也會做等待。利用這個等待的特性，搭配 default 可以用來檢查 buffered channel 是否已滿

```go
package main

import "fmt"

func main() {
    buffer_channel := make(chan bool, 1)

    // 占滿 buffer_channel
    buffer_channel <- true

    select {
        case buffer_channel <- true: // 嘗試傳值給 buffer_channel
            fmt.Println("buffer_channel 沒有滿")
        default:
            fmt.Println("buffer_channel 已滿")
    }
}
```

執行結果：

buffer_channel 已滿

https://go.dev/play/p/X49ytulw0U-

併發

通道(Channel)

- select：同時監聽多個 channel

```go
package main
import "fmt"
func main(){
    ch1 := make(chan int)
    ch2 := make(chan int)
    go func(){
        ch1 <- 1
    }()

    go func(){
        ch2 <- 2
    }()

    select{
    case <- ch1:
        fmt.Println("接收ch1")
    case <- ch2:
        fmt.Println("接收ch2")
    }
}
```

執行結果：
接收ch2

每次執行都不一樣，由作業系統決定

利用 select + case 可以同時監聽不同的 channel 與 switch 不一樣的地方

在於 switch 是由上到下有順序的比對，而 select 是同時比對、隨機的

https://go.dev/play/p/fIfM6NnSSkA

併發

通道(Channel)

- select：監聽同一個 channel

```go
package main
import "fmt"
func main(){
    for i:=0; i<10; i=i+1{
        ch := make(chan int)
        go func(){
            ch <- 1
        }()

        select{ // 三個 case 會同時收到 ch 上的訊息
        case <- ch:
            fmt.Println("1")
        case <- ch:
            fmt.Println("2")
        case <- ch:
            fmt.Println("3")
        }
    }
}
```

執行結果：

```
1
2
3
3
3
3
2
1
3
2
```

如果有多個 case 同時收到訊息，則 select 會隨機挑選 case。

https://go.dev/play/p/gKX9jBQEWuW

併發

通道(Channel)

- select：設置計時器，若超時就不繼續監聽
  - select 在實際使用上常常搭配計時器使用，當時間超過時就會結束監聽。另開一個通道做為計時器使用，時間一到就會發送訊息將 select 的等待打斷
  - 因為在收到 ch 訊息前就先收到 timeout 的訊息了，所以主執行緒離開 select，不會被擋住。這可以用來確保 select 最多只會給予 3 秒的等待。

```go
package main
import(
    "fmt"
    "time"
)
func main(){
    ch := make(chan int)
    timeout := make(chan bool)

    // 檢查有沒有超過 3 秒
    go func(){
        time.Sleep(3 * time.Second)
        timeout <- true
    }()

    // 模擬一個需要耗時五秒的執行緒
    go func(){
        // 可自行調整秒數試試
        time.Sleep(5 * time.Second)
        ch <- 10
    }()

    select{
    case num := <- ch:
        fmt.Println(num)
    case <- timeout:
        fmt.Println("Time out !!")
    }
}
```

執行結果：

Time out !!

https://go.dev/play/p/_i5GcKIXIT6

併發

通道(Channel)

- select：更簡單地設置計時器
  - 因為計時器常常使用，但使用時還要考慮計時的起點還有通道的使用等，還挺麻煩的，因此在 time package 中已經預設了一個函式，可以快速啟用計時器

```
func After(d Duration) <-chan Time
```

```go
package main
import(
    "fmt"
    "time"
)
func main(){
    ch := make(chan int)

    // 模擬一個需要耗時五秒的執行緒
    go func(){
        // 可自行調整秒數試試
        time.Sleep(5 * time.Second)
        ch <- 10
    }()

    select{
    case num := <- ch:
        fmt.Println(num)
    case timer := <- time.After(3 * time.Second):
        fmt.Println("Time out !!")
        fmt.Println(timer)
    }
}
```

執行結果

```
Time out !!
2020-09-19 18:17:13.878545989 +0000 UTC m=+3.000823994
```

https://go.dev/play/p/9AWCX8KwVe5

併發

通道(Channel)

- 單向通道
  - channel 通常都是可發送可接收，但如果今天設計了一款套件，該套件會在某些情況透過 channel 發送訊息。如果今天使用者反過來透來 channel 傳送訊息，會導致套件出錯。那麼要怎麼預防呢？其實這個就是 time.After()
  - 該函式在設定一段時間後，會透過通道傳回 Time 類型的 channel，回傳型態的左方多了 <- 。這個代表說該型態僅允許「接收」

```
func After(d Duration) <-chan Time
```

```go
package main
import "fmt"

func counter(start int, end int) <-chan int{
    channel := make(chan int)
    single := make(<-chan int)
    single = channel
    go func(){
        for start  <= end{
            channel <- start
            start = start + 1
        }
        close(channel)
    }()
    return single
}

func main(){
    // counter 回傳的 channel 僅用於接收
    // 若嘗試寫入將會出錯
    for k := range counter(0, 10){
        fmt.Println(k)
    }
}
```

```
執行結果：
0
1
2
3
4
5
6
7
8
9
10
```

併發

通道(Channel)

- 單向通道
  - 如果嘗試寫入則會在編譯時出錯：

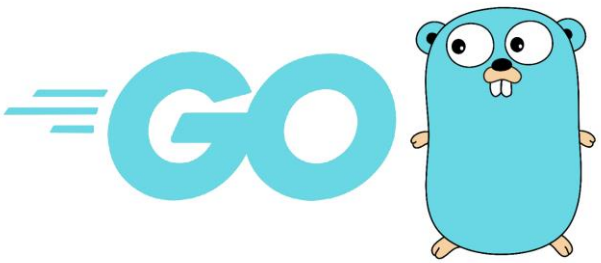
```go
package main
import "fmt"

func counter(start int, end int) <-chan int{
    channel := make(chan int)
    single := make(<-chan int)
    single = channel
    go func(){
        for start  <= end{
            channel <- start
            start = start + 1
        }
        close(channel)
    }()
    return single
}

func main(){
    ch := counter(0, 10)
    for k := range ch{
        fmt.Println(k)
        // 嘗試寫入回去
        ch <- k
    }
}
```

```
# command-line-arguments
./lesson20.go:23:12: invalid operation: ch <- k (send to receive-only type <-chan int)
```

僅能發送 `make(chan type <-)`

僅能接收 `make(<- chan type)`

錯誤處理



錯誤處理

錯誤處理(Error Handling)

- 在執行程式時，遇到系統錯誤或是網路異常是無可避免的，這時可能印出錯誤訊息並且讓程式中斷。當然也不只系統上的錯誤，有時會有參數帶入的錯誤，也必須印出錯誤訊息。
- **Error Interface**
  - 在 Go 語言中，將錯誤定義成一個 interface，而內建的 error interface 如下：

```
type error interface {
    Error() string
}
```

error 也是一個原生型態，是一個由 interface 所定義的型態，通常設為 nil 時當做沒有錯誤，反之如果為非 nil 則代表有錯誤發生，利用這個方法就能簡單地偵測是否有錯誤發生

  - **在許多套件中的 function，都會在回傳值中，帶一個 error 回傳**，例如 strconv.Atoi，看一下這個 function 的組成：

```
func strconv.Atoi(s string) (int, error)
```

錯誤處理

錯誤處理(Error Handling)

- Error Interface
  - 那直接來看如何運用這個 error 回傳值：

```go
package main

import (
    "fmt"
    "strconv"
)

func main() {
    i, err := strconv.Atoi("0.5")
    if err != nil {
        fmt.Println("couldn't convert number:", err)
        return
    }
    fmt.Println("Converted integer:", i)
}
```

先來講解 strconv.Atoi 的用途，帶入一個字串，但必須是整數字串，舉個例子，例如 "1"，將其帶入會輸出整數1，以及error，當然如果沒有錯誤，當然就是回傳空值。

那以左邊的例子，err 這個變數不為空值，這時試著只印出 err，會是 invalid syntax(無效的語法)，如果在一個較大的專案，哪會知道錯誤發生在哪段，所以盡可能印出更詳細的資訊，才比較能夠追蹤錯誤，
有時不見得是參數帶錯，而是程式和需求相左，所以這樣如此一來也方便debug。

https://go.dev/play/p/k62PaulF9Dm

錯誤處理

錯誤處理(Error Handling)

- Error Interface
  - 在這邊要特別宣導一下，只要有看到 func 的回傳值有 error ，請一定要接出來，不要忽略它。以 net.Listen 套件為例：

```
ln, err := net.Listen("tcp", ":8080")
if err != nil {
    error handler ...
}
conn, err := ln.Accept()
```

  - 如上，如果沒有讓它直接攔到，讓它持續往下執行，做了 ln.Accept()，這時侯 ln 是 nil ，所以會觸發 nil of pointer，所以請不要嫌麻煩，任何錯誤都要接起來並且處理，可以省掉很多 debug 時間。

錯誤處理

錯誤處理(Error Handling)

- **利用 errors.New() 自訂錯誤訊息**

1. 引入 "erros" 套件
2. 判斷錯誤條件，確認為錯誤情況時回傳 error.New("錯誤訊息") 否則回傳 nil

使用nil來判斷err

```go
package main

import (
    "fmt"
    "errors"
)

func Hello(name string) (string, error) {

    if name == "" {
        return "", errors.New("name is empty")
    }

    message := fmt.Sprintf("Hi, %v. Welcome!", name)
    return message, nil
}

func main() {

    message, err := Hello("")

    if err != nil {
         fmt.Println(err)
    }

    fmt.Println(message)
}
```

錯誤處理

錯誤處理(Error Handling)

- **利用 fmt.Errorf() 更快地使用自訂錯誤訊息**
  - 但是利用 errors.New() 還要引用 errors 套件，因此還可以使用 fmt.Errorf() 來實做，用法跟 fmt.Printf(), fmt.Sprintf() 很像，只是 Printf() 是直接印出，Sprintf() 是回傳 string，而 Errorf() 則是回傳 error

```go
package main
import "fmt"

func div(a, b int) (int, error){
    if b == 0 {
        // 使用上比 errors.New() 更好上手
        return 0, fmt.Errorf("%d / %d 除數不可為零", a, b)
    }
    return (a/b), nil
}

func main(){
    var a, b int
    fmt.Print("輸入兩個整數：")
    _, err := fmt.Scanf("%d %d", &a, &b)

    if err != nil{
        fmt.Println("格式錯誤")
        return  // 直接離開 main()
    }

    ans, err := div(a, b)
    if err != nil{
        fmt.Printf("%s", err)
    }else{
        fmt.Printf("%d / %d = %d",a ,b, ans)
    }
}
```

執行結果 (輸入 10 0)：
輸入兩個整數：10 0
10 / 0 除數不可為零

https://go.dev/play/p/Yxdke3bcW9P

錯誤處理

錯誤處理(Error Handling)

- **自己實作 error 的 interface{}**
  - error 是一個 interface{}，只是官方提供了幾個已經設計好的函式可以直接使用 errors.New() & fmt.Errorf，那麼想自己實作要怎麼實現呢？
  - 只要自己創立一個新的型態該型態滿足 Error() 方法，那麼也可以視為 error 來使用：

```
type myError struct{
    msg string
}

func (m *myError) Error() string{
    return m.msg
}
```

  - 如此一來 *myError 也符合 error 了

錯誤處理

錯誤處理(Error Handling)

- 自己實作 error 的 interface{}

```go
package main
import "fmt"

type myError struct{
    msg string
}

func (m *myError) Error() string{
    return m.msg
}

func div(a, b int) (int, error){
    if b == 0 {
        err := &myError{msg: "除數不可為零"}
        return 0, err
    }
    return (a/b), nil
}

func main(){
    var a, b int
    fmt.Print("輸入兩個整數：")
    _, err := fmt.Scanf("%d %d", &a, &b)

    if err != nil{
        fmt.Println("格式錯誤")
        return  // 直接離開 main()
    }

    ans, err := div(a, b)
    if err != nil{
        fmt.Printf("%s", err)
    }else{
        fmt.Printf("%d / %d = %d",a ,b, ans)
    }
}
```

執行結果 (輸入 10 0)：
輸入兩個整數：10 0
除數不可為零

https://go.dev/play/p/JcMipegm2op

錯誤處理

錯誤處理(Error Handling)

- Go 語言本身沒有例外處理機制，而是以 **panic**、**recover** 取而代之，用來滿足錯誤處理的需求。
- **panic(錯誤恐慌)**
  - 使用 panic 會導致系統崩潰和服務中斷，所以 panic 會使用在發生較重大錯誤的時候：

```go
package main
import (
  "fmt"
  "strconv"
)
func main() {
  i, err := strconv.Atoi("0.5")
  if err != nil {
      panic("crash")
      fmt.Println("couldn't convert number:", err)
  }
  fmt.Println("Converted integer:", i)
}
```

在執行後會產生裡面的錯誤訊息，且不會執行後面的程式碼，來看看輸出的結果：

```
panic: crash
```

其實這個例子只是簡單舉例 panic 的作用，在自己的服務中，什麼時候要使用到 panic，需要自己評估。

https://go.dev/play/p/wpHM-T966Cy

錯誤處理

錯誤處理(Error Handling)

- Go 語言本身沒有例外處理機制，而是以 **panic**、**recover** 取而代之，用來滿足錯誤處理的需求。
- **recover(捕獲錯誤)**
  - Go 語言中沒有try-catch這類的捕捉語法，但提供 recover 可以捕捉到 panic 拋擲錯誤，**以回復系統以避免程序崩潰**。
    因為 defer 在程式碼最後必執行的特性，recover 必須和 defer 配合使用，如下：

```go
package main

import (
   "fmt"
   "strconv"
)

func main() {
   i, err := strconv.Atoi("0.5")

   defer func() {
       if err := recover(); err != nil {
           fmt.Println("couldn't convert number:", err)
       }
   }()

   if err != nil {
       panic("crash")
   }

   fmt.Println("Converted integer:", i)
}
```

在執行 panic 後系統並無直接崩潰，而是繼續執行 recover 後面的程式碼，但一樣，在 panic 後的程式碼是不會被執行的。

https://go.dev/play/p/Kjl7MGl6n5V