

# CH16 並行性運算

# 16-1 前言

- 到目前為止，你所學習的程式大多都是只給單一使用者操作的軟體，但其實有另一種軟體設計是給多人同時操作的
- 第15章的HTTP伺服器就是一個例子，他能同時提供網站或應用程式給許多人使用，或者你的程式可能需要同時讀取幾種不同的資料庫，在這種情況下，你可以試著用非同步的方式同時完成這些任務
- 打比方說，有幾個人要在牆壁上釘釘子，每個人分配到的釘子數都不同，但鐵鎚只有一把；這時與其讓每個人依序完成任務，倒不如規定每個人都只能用鐵鎚敲一下，然後就換下一個人，這樣釘子較少的人就會先完成任務，但起碼不用等釘子多的人先用完鐵鎚

- Go語言使用一種稱為Goroutine的輕量執行緒(thread)來實現並行性運算(concurrent computing), 也就是同一時間只進行一個任務, 但將每個任務切成小片段, 並輪流執行每個任務的片段, 直到所有任務完成
- Goroutine非常容易啟動, 而且已經運作在go語言的標準套件中; 比如前面提到的go語言伺服器就會自動使用goroutine, 讓客戶端能在任何時候連線, 不必等待伺服器處理完個別使用者的請求
- 不過, 這也意味著大多數的HTTP伺服器都是無狀態(stateless), 也就是每次的請求都是不相干的; 若想讓程式記住不同請求之間的狀態或不同使用者的資訊, 你就得讓這些goroutine共享資源, 但這也會導致所謂的記憶體資源競爭(race condition)問題, 在這一章中將會看到幾種對策

# Go語言的並行性運算/平行性運算

- 並行性運算(**concurrent computing**)是指在同一處理器上輪流切換不同的任務，而平行性運算(**parallel computing**)則是使用處理器的不同核心來同時進行幾個任務
- 儘管一般會用並行性運算來稱呼go語言的非同步運算功能，go語言其實會在幕後使用多個os執行緒來分攤goroutine的運行，也就是啟用了平行性運算，不過開發者不需擔心實作細節

- 從go 1.5 版之後，os執行緒的數量即系統最大核心數量，這會在安裝時就自動設定
- 若你想指定os執行緒數量，可用runtime套件的GOMAXPROCS()：

`runtime.GOMAXPROCS(N)` //指定N個核心

`numCPU := runtime.GOMAXPROCS(0)` //取回目前的OS執行緒數量

注意：若OS執行緒數量大於處理器核心數量，就會使他們競爭CPU資源

## 16-2 使用Go語言的並行性運算

## 16-2-1 Goroutine

- 每個goroutine就是一個非同步函式，通常用來做一件任務
- Go語言允許在同一時間執行多個goroutine，你甚至可以在一個goroutine內執行其他的goroutine，但這些函式彼此之間會是獨立的並行型運算任務
- Goroutine程序不會共用記憶體，有別於傳統的執行緒


- 但我們將看到，在程式中讓goroutine相互傳遞變數是相當容易的，不過沒有採取特別措施的話，還是會引發出乎預料的現象
- 撰寫goroutine的方式完全沒有任何特別之處，它本身就是一個普通的函式而已
- 任何函式都可以變成goroutine；只要在呼叫時前面加上go關鍵字即可，來看看下面這個hello()：



```
1  package main
2
3  import "fmt"
4
5  ✓ func hello() {
6      |   fmt.Println("Hello World")
7      | }
8
9  ✓ func main() {
10     |   fmt.Println("開始")
11     |   go hello() //產生一個goroutine
12     |   fmt.Println("結束")
13     | }
14
```

- 要注意的是，用**go**關鍵字呼叫的函式不能有回傳值，但後面會看到如何避開這個限制

- 執行結果：



開始  
結束

hello()沒有執行

- 程式會先印出開始，然後用**goroutine**的方式呼叫**hello()**
- 然而，**main**沒等**hello()**執行，就直接印出“結束”，因為**main()**和**hello()**變成兩個獨立執行的程序了

- 這裡的重點是：就算go語言程式沒有明確使用go關鍵字來呼叫任何函式，main()本身其實也是一個goroutine
- 也就是說，上面實際上在執行兩個goroutine，但main()一結束，所有的goroutine就會一並關閉
- 為了確保其他的goroutine有時間完成，可以用個粗糙卻有效的方法：讓main()等待片刻：

```
1  package main
2
3  import (
4      "fmt"
5      "time"
6  )
7
8  func hello() {
9      fmt.Println("Hello World")
10 }
11
12 func main() {
13     fmt.Println("開始")
14     go hello()
15     time.Sleep(time.Second) //等待一秒
16     fmt.Println("結束")
17 }
18
```

執行結果：

```
開始  
Hello World  
結束
```

# 練習：使用goroutine

- 這裡要進行兩個運算任務，一個是從1連加到10(等於55), 另一個是從1連加到100(等於5050)
- 為節省時間，我們會讓程式同時執行這些運算，並一起看到計算結果

```
1 package main
2
3 import (
4     "fmt"
5     "time"
6 )
7
8 func sum(from, to int) int {
9     res := 0
10    for i := from; i <= to; i++ {
11        res += i
12    }
13    return res
14 }
15
16 func main() {
17     var s1, s2 int
18
19     go func() { //執行匿名goroutine, 它會執行s1
20         s1 = sum(1, 100)
21     }()
22
23     s2 = sum(1, 10) //s2仍在main()內執行
24
25     time.Sleep(time.Second) //等待1秒
26     fmt.Println(s1, s2)
27 }
28
```

- 可以發現這兩個計算任務用goroutine的形式順利完成，也得到正確的結果：

```
5050 55
```



## 16-2-2 WaitGroup

- 上個練習用了一個粗暴的方式逼迫main()多等1秒, 好確保goroutine能執行完畢, 下面要來看第二種作法, 使使用sync套件的WaitGroup結構:

```
func main() {  
    wg := &sync.WaitGroup{} //建立新的WaitGroup結構  
    wg.Add(n) //在WaitGroup紀錄Goroutine數量  
    //執行n個goroutine  
    wg.wait() //等待所有goroutine執行完畢
```

- 在此我們建立了一個指標變數，指向一個WaitGroup結構，然後用其方法wg.Add()來告訴它我們準備加入n個goroutine
- WaitGroup其實就是個計數器，會記住目前有多少goroutine在跑
- 最後它會用wg.Wait()來等待所有goroutine結束
- 但WaitGroup要如何知道函式執行完畢呢？我們得讓每個goroutine在執行完畢時告訴WaitGroup：

wg.Done() ← 等於wg.Add(-1), 把目前正在跑的goroutine數量減1

# 練習：使用WaitGroup

- 這個練習一樣會執行sum函式兩次，其中一次是goroutine, 但這回改用WaitGroup來等待它結束：

```
1 package main
2
3 import (
4     "fmt"
5     "sync"
6 )
7
8 func sum(from, to int, wg *sync.WaitGroup, res *int) {
9     *res = 0
10    for i := from; i <= to; i++ {
11        *res += i
12    }
13    if wg != nil {
14        wg.Done() //回報goroutine結束
15    }
16 }
17
18 func main() {
19     var s1, s2 int
20
21     wg := &sync.WaitGroup{} //建立WaitGroup
22     wg.Add(1)                //要等待一個goroutine
23     go sum(1, 100, wg, &s1) //以goroutine形式執行sum()
24     sum(1, 10, nil, &s2)    //以正常方式執行sum()
25     wg.Wait()               //等待goroutine結束
26
27     fmt.Println(s1, s2)
28 }
29
```

- 在前一個練習中，我們把`sum()`包在一個匿名函式裡來取得傳回值，然後把該匿名函式當成`goroutine`執行，但這次用指標參數來取得結果，因為`goroutine`不能有回傳值
- `main` 執行完自己的計算後，`wg.Wait()`會等待所有的`goroutine`都執行完了才會印出結果：

```
PS D:\git\Golang> go run "d:\git\Golang\ch16\16-2-2\main.go"  
5050 55
```

## 16-3 解決記憶體資源競爭(race condition)

- 當使用並行性運算時，必須切記：我們無法擔保每個goroutine的程式碼會以怎樣的順序執行
- 雖然在很多情況下，這點並不是問題，因為每個函式都是獨立作業，但若這些goroutine必須公用同樣的資料，情況就不一樣了
- 因為我們不曉得goroutine的讀寫順序，可能某函式更新了變數卻被另一個函式蓋過，導致運算結果不正確

- 我們用一個例子來解釋這種情況：首先是用一個不使用goroutine的版本，每次呼叫函式會把一個指標變數遞增1：

```
package main

import "fmt"

func next(v *int) {
    (*v)++
}
```

You, 3 週前 • 22/8/16 ...

```
func main() {
    a := 0
    next(&a)
    next(&a)
    next(&a)
    fmt.Println(a)
}
```

- `main()`內的變數`a`會正常的遞增到3
- 但若把`man`的內容改成如下：

```
func main() {  
    a := 0  
    go next(&a)  
    go next(&a)  
    go next(&a)      You, 1  
    fmt.Println(a)  
}
```

- 執行的結果可能是0,1,2或3，這是因為現在每次呼叫的`next()`函式都變成獨立的goroutine，它們並不曉得其他goroutine在做甚麼，於是可能覆蓋了彼此寫入的變數值



- 以上情況就是所謂的記憶體資源競爭(**race condition**), 而若是我們沒有採用防範措施, 這種狀況就會一再發生
- 幸好, **go**語言給了幾種辦法, 確保一次只由一個來源能寫入變數

## 16-3-1 原子操作(atomic operation)

- 試想欲計算從1累加到某數，並將作業拆成兩個goroutine來處理
- 以下計算1累加到4，其總和為10：

s := 0

s += 1 (由goroutine1計算)

s += 2 (由goroutine1計算)

s += 3 (由goroutine2計算)

s += 4 (由goroutine2計算)

- 但兩個goroutine做累加的順序我們並不能保證，因此可能又發生記憶體資源競爭，也就是多重goroutine同時寫入同一個變數，覆蓋了彼此的值
- 幸好，go語言的sync/atomic套件讓我們能安全地在各goroutine之間修改數值
- 若你要在goroutine之間進行簡單的數值變數操作，這個套件就能確保該變數一次只能有一個來源寫入它，這種動作即稱為原子操作

- **atomic**套件提供了以下的原子操作函式：

```
func AddInt32(addr *int32, delta int32) (new int32)
```

```
func AddInt32(addr *int64, delta int64) (new int63)
```

```
func AddInt32(addr *unit32, delta unit32) (new unit32)
```

```
func AddInt32(addr *unit2, delta unit32) (new unit32)
```

以**atomic.AddInt32()**為例，它會接收一個指向**int32**型別的指標變數**addr**，將其值加上**delta**，並確保每一次都只能有一個來源修改**addr**

由上可見，**atomic**套件支援的操作型別只有**int32/64**和**unit32/64**

# 練習：使用原子操作

- 這個練習要計算1累加到100的值，但將計算過程拆成4個goroutine
- 為了確保變數寫入不會互相覆蓋，這裡使用原子操作
- 稍後還會寫一個單元測試，來展示使用原子操作與否有何差異

# 主程式

```
1  package main
2
3  import (
4      "log"
5      "sync"
6      "sync/atomic"
7  )
8
9  //計算累加的函式
10 func sum(from, to int, wg *sync.WaitGroup, res *int32) {
11     for i := from; i <= to; i++ {
12         atomic.AddInt32(res, int32(i))
13     }
14     wg.Done()
15 }
16
```

```
17 func main() {
18     s1 := int32(0)
19     wg := &sync.WaitGroup{}
20
21     wg.Add(4) //新增4個goroutine
22     go sum(1, 25, wg, &s1)
23     go sum(26, 50, wg, &s1)
24     go sum(51, 75, wg, &s1)
25     go sum(76, 100, wg, &s1)
26     wg.Wait() //等待所有goroutine結束
27
28     log.SetFlags(0) //設定log輸出實不帶其他資訊(時間/程式名稱等)
29     log.Println(s1)
30 }
31
```

- 這裡使用log套件輸出結果，是為了後面的單元測試中能讀取程式的輸出值
- 執行結果如下：

```
PS D:\git\Golang> go run "d:\git\Golang\ch16\16-3-1\main.go"  
5050
```



# 單元測試

- 現在我們在同一專案內撰寫單元測試`main_test.go`，它會執行上面的`main()`一萬次，並檢查主控台的印出結果是否符合預期：

```
package main
```

```
import (  
    "bytes"  
    "log"  
    "testing"  
)
```

```
run test | debug test
```

```
func Test_Main(t *testing.T) {  
    for i := 0; i < 10000; i++ {  
        var s bytes.Buffer //建立一個Buffer結構(符合io.Writer介面)  
        log.SetOutput(&s) //將log輸出結果寫道s  
        log.SetFlags(0) //設定log輸出時不帶其他資訊  
        main()  
  
        //只要log輸出內容不是字串5050\n就算失敗  
        if s.String() != "5050\n" {  
            t.Fail()  
        }  
    }  
}
```

- 如同第九章介紹的，使用go test來執行單元測試：

```
PS D:\git\Golang\ch16\16-3-1> go test -v
=== RUN   Test_Main
--- PASS: Test_Main (0.05s)
PASS
ok      ch12/ch16/16-3-1      0.399s
```

- 結果：全部正常

# 記憶體資源競爭測試

- 我們有時無法確定程式的計算結果是否正確，因為有可能計算結果正確，但檯面下卻發生記憶體資源競爭發生
- 這時可以欲`go test` 加上 `-race` 來測試
  - 不過要使用`-race`之前，需下載MinGW 64位元版，以下是教學網址

<https://www.twblogs.net/a/5cb2f0c9bd9eee480f079add>

- 加上-**race**標籤再跑一次單元測試：

```
PS D:\git\Golang\ch16\16-3-1> go test -v -race
=== RUN   Test_Main
--- PASS: Test_Main (1.59s)
PASS
ok      ch12/ch16/16-3-1      3.053s
```

- 若我們把main.go的原子操作功能拿掉，換成普通變數累加：

```
9  func sum(from, to int, wg *sync.WaitGroup, res *int32) {
10      for i := from; i <= to; i++ {
11          //atomic.AddInt32(res, int32(i))
12          *res += int32(i)
13      }
14      wg.Done()
15  }
16
```

- 這時再執行一次程式，結果可能是正確的，但執行多次後一定會出現錯誤
- 現在用go test 搭配 -race 再測試看看：

```
PS D:\git\Golang\ch16\16-3-1> go test -v -race
=== RUN    Test_Main
=====
WARNING: DATA RACE
Read at 0x00c00012025c by goroutine 9:
  ch12/ch16/16-3-1.sum()
    D:/git/Golang/ch16/16-3-1/main.go:12 +0x4d
ch12/ch16/16-3-1.main.func2()
    D:/git/Golang/ch16/16-3-1/main.go:23 +0x51

Previous write at 0x00c00012025c by goroutine 8:
  ch12/ch16/16-3-1.sum()
    D:/git/Golang/ch16/16-3-1/main.go:12 +0x5d
ch12/ch16/16-3-1.main.func1()
    D:/git/Golang/ch16/16-3-1/main.go:22 +0x51

Goroutine 9 (running) created at:
  ch12/ch16/16-3-1.main()
    D:/git/Golang/ch16/16-3-1/main.go:23 +0x1e4
ch12/ch16/16-3-1.Test_Main()
    D:/git/Golang/ch16/16-3-1/main_test.go:14 +0xe9
testing.tRunner()
    C:/Program Files/Go/src/testing/testing.go:1439 +0x213
testing.(*T).Run.func1()
    C:/Program Files/Go/src/testing/testing.go:1486 +0x47
```

- 訊息開頭的 **WARMING: DATA RACE** 就指出了 `mian()` 發生了記憶體資源競爭問題



## 16-3-2 互斥鎖(mutex)

- 除了原子操作，還有一個方法能讓你正常對共用變數寫入值，且榮維持並行安全性，那就是 **sync** 套件的 **mutex** 結構，這種方式適用於任何變數
- **mutex**是互斥鎖(**mutual exclusion**)的簡稱；當互斥鎖啟用時，它會停止所有的**goroutine**, 直到鎖被解除為止
- 因此某個**goroutine**需要操作資料時，可先要求上鎖，等到做完必要的任務後再解鎖；其他**goroutine**則需等待解鎖後才能接收互斥鎖

- `mutex` 和 `WaitGroup` 一樣定義在`sync`套件中，你首先得建立一個`sync.Mutex` 型別的互斥鎖結構：

```
mtx := sync.Mutex{}
```

- 但通常會將一個鎖傳給多個`goroutine`使用，因此會被宣告成指標變數：

```
mtx := &sync.Mutex{}
```

- 接著你就能在各個goroutine內使用互斥鎖上鎖和解鎖方法來包住共用變數的操作：

mtx.Lock()    上鎖  
(\*s)++        寫入變數  
mtx.Unlock() 解鎖

下面的程式碼修改了前一個練習，改用互斥鎖來進行數值累加作業：

```
1 package main
2
3 import (
4     "log"
5     "sync"
6 )
7
8 func sum(from, to int, wg *sync.WaitGroup, mtx *sync.Mutex, res *int32) {
9     for i := from; i <= to; i++ {
10         mtx.Lock()
11         *res += int32(i)
12         mtx.Unlock()
13     }
14     wg.Done()
15 }
16
17 func main() {
18     s1 := int32(0)
19     wg := &sync.WaitGroup{}
20     mtx := &sync.Mutex{}
21
22     wg.Add(4)
23     go sum(1, 25, wg, mtx, &s1)
24     go sum(26, 50, wg, mtx, &s1)
25     go sum(51, 75, wg, mtx, &s1)
26     go sum(76, 100, wg, mtx, &s1)
27     wg.Wait()
28
29     log.SetFlags(0)
30     log.Println(s1)
31 }
```

- 執行後結果一樣是5050
- 不過請記得，程式中上鎖/解鎖的動作應該越少，時間越短越好，才能減少各goroutine等待的時間，提升並行性運算的效能
- 因此，只有在真正會共用並寫入資料的資源才使用互斥鎖

## 16-4 通道(channel)

- 顧名思義，通道是傳遞訊息的管道，任何函式都能透過通道送出或接收訊息
- 通道的宣告和初始化方式跟切片很像：

```
var ch chan int    ← 建立名為ch的通道，型別為int  
ch = make(chan int, 10) ← 初始化通道，緩衝區大小為10
```

當然也可以直接建立它：

```
ch := make(chan int, 10)
```

- 通道可以是任何型別，諸如int/bool/float/自訂型別/結構/切片/指標，只是最後兩個比較少用
- 通道變數能當成參數傳給函式，不用宣告成指標就能讓goroutine分享資料，且他們已經具備並行性運算安全性，所以存取時不必動用互斥鎖之類的機制
- 先來看看如何傳遞訊息到一個通道：

```
ch <- 2 //將整數2傳入通道ch
```

- `<-` 就是所謂的受理算符(receive operators), 若試圖傳送不同型別的資料給ch通道就會產生錯誤
- 送出訊息後, 你也會想從通道中接收訊息:

`<- ch` //從通道中移除一筆訊息

`i := <- ch` //從通道中取出一筆訊息並存入變數i

`i, ok := <- ch` //取出一筆訊息存入變數i, 成功時回傳true

甚至可以像下面這樣, 直接從一個通道取值後放入另一個通道:

`out <- <- in`

- 註: go語言通道會遵守先進先出原則, 也就是先傳入的訊息會最早收到



## 16-4-1 使用通道傳遞訊息

```
1  package main
2
3  import "fmt"
4
5  func main() {
6      ch := make(chan int, 1)
7      defer close(ch)
8      ch <- 1    //在通道傳入1
9      i := <-ch //從通道取值並存入i
10     fmt.Println(i)
11 }
12
```

問題 11 輸出 偵錯主控台 GITLENS JUPYTER 終端機

Windows PowerShell

著作權 (C) Microsoft Corporation。保留擁有權利。

安裝最新的 PowerShell 以取得新功能和改進功能！<https://aka.ms/PSWindows>

PS D:\git\Golang> go run "d:\git\Golang\ch16\16-4-1\main.go"

1

- 上面的程式建立了緩衝區為1的通道，傳入整數1，再把他讀出來
- 現在我們移除通道緩衝區大小參數，也就是沒有緩衝區，會發生什麼呢？

6

```
ch := make(chan int)
```

- 執行結果：

```
fatal error: all goroutines are asleep - deadlock!  
  
goroutine 1 [chan send]:  
main.main()  
      d:/git/Golang/ch16/16-4-1/main.go:8 +0x6e  
exit status 2
```

- Go語言回報程式中所有的goroutine都陷入了死結(deadlock), 而錯誤發生在, 而錯誤發生在第8行(ch<-1)
- 由於這個通道ch沒有任何緩衝區, 當你傳一個值進去時就必須馬上讀出來, 問題是沒有任何函式能這麼做, 導致程式無止盡的等待下去

- 我們再來看通道的第二個特性，也就是通道是可以關閉的
- 當通道負責的任務結束時，你可以關閉它：

`close(ch)`

- 通道也可以用**defer**延後關閉，比如：

```
func main() {  
    ch := make(chan int, 1)  
    defer close(ch)  
    ch <- 1  
    i := <-ch  
    fmt.Println(i)  
}
```

- 但通道和檔案/HTTP連線不一樣，它並沒有一定要關閉，這麼做有其他的理由，稍後會再談到

# 練習：兩個goroutine透過通道交換訊息

- 下面練習中，會用一個goroutine傳送歡迎訊息，並在main()中接收它

```
1  package main
2
3  import "fmt"
4
5  func greet(ch chan string) {
6      |   ch <- "Hello" //對通道傳入訊息
7  }
8
9  func main() {
10     |   ch := make(chan string) //建立無緩衝區的字串通道
11     |   go greet(ch)           //將通道傳給goroutine
12     |   fmt.Println(<-ch)      //從通道接收訊息
13 }
14
```



- 執行結果：

```
PS D:\git\Golang> go run "d:\git\Golang\ch16\16-4-1(2)\main.go"  
Hello
```

- 以上練習中，可以看到如何用通道讓不同的goroutine彼此溝通
- 程式中使用的通道ch沒有緩衝區，但正因main()即時的讀出資料，所以沒有造成死結

# 練習：兩個goroutine用通道雙向交換訊息

- 現在我們想讓main()傳送一個訊息給另一個非同步函式，然後接收對方回傳的訊息

```
1  package main
2
3  import "fmt"
4
5  func greet(ch chan string) {
6      msg := <-ch //接收訊息1
7      ch <- fmt.Sprintf("收到訊息: %s", msg) //傳入訊息2(包含訊息1)
8      ch <- "Hello David" //傳入訊息3
9  }
10
11 func main() {
12     ch := make(chan string)
13     go greet(ch)
14
15     ch <- "Hello John" //傳入訊息1
16     fmt.Println(<-ch) //接收訊息2
17     fmt.Println(<-ch) //接收訊息2
18 }
19
```

- 本練習展示了goroutine如何利用通道來雙向溝通
- 注意到main()用了兩次<-ch, 因為你預期有兩條訊息傳回來
- 執行結果：

```
收到訊息: Hello John  
Hello David
```

## 16-4-2 從通道讀取多重來源資料

- 試想一個情況：你想加總一系列數字，但數字會由幾個不同的 `goroutine` 提供
- 事實上，我們不必知道實際處理的數字為何，反正全部加起來就對了

# 練習：用通道做數字加總

- 下面我們要改寫之前的數字加總練習，讓四個goroutine對main()傳送特定範圍的數字，並由main()負責加總

- 為了示範並行性運算的效果，這裡使用time模組在goroutine中加入一點點時間延遲，使每個goroutine在傳送訊息後稍等片刻，這樣比較容易看出各個goroutine輪流執行的效果：

```
PS D:\git\Golang> go run "d:\git\Golang\ch16\16-4-2\main.go"
76
26
1
51
77
2
27
52
28
78
53
3
79
```

中略

```
23
100
73
24
74
25
75
Result: 5050
```

- 根據提供的數字你能猜到它們是哪個goroutine提供的，這也展示了單一通道的資料可能來自多個來源
- 在真實世界中, 你可以用同樣的方式從多重資料庫讀取資料，並透過通道把他們通通傳給一個處理程序



# 練習：向goroutine請求資料

- 接下來的練習要解決的問題和前一個練習一樣，但方法稍微不同
- 與其一直接收goroutine傳回的數字，這回我們讓main()先對goroutine提出請求，後者收到要求後猜傳回一個數字

```
1 package main
2
3 import "fmt"
4
5 func push(from, to int, in chan bool, out chan int) {
6     for i := from; i <= to; i++ {
7         <-in //等待請求(值不重要)
8         out <- i //傳回一個值
9     }
10 }
11
12 func main() {
13     s1 := 0
14     out := make(chan int, 100) //用來接收值的通道
15     in := make(chan bool, 100) //用來送出請求的通道
16
17     go push(1, 25, in, out)
18     go push(26, 50, in, out)
19     go push(51, 75, in, out)
20     go push(76, 100, in, out)
21
22     for c := 0; c < 100; c++ {
23         in <- true //送出一個請求
24         i := <-out //接收一個數字
25         fmt.Println(i)
26         s1 += i
27     }
28
29     fmt.Println("Result:", s1)
30 }
31
```

# 執行結果：

前略

```
72  
48  
98  
23  
73  
74  
99  
49  
24  
100  
25  
Result: 5050
```

- 在這個練習中，`main()`的迴圈會先請求一個數字(在通道`in`放一個`true`)，然後等待通道`out`有值放入
- 對於任何`push()`函式來說，只要通道`in`內有值可取(有請求在排隊)，它就會提供一個數字到`out`通道；換言之，`main()`總共會送出100個請求，而四個`goroutine`總共會傳回100個數字

## 16-5 並行性運算的流程控制

- 在組織goroutine時，有幾種常見的模式：
  - 一種叫做管線(pipeline), 顧名思義就是將goroutine像生產線一樣串起來，資料會自來源輸入，藉由通道在各函式間傳遞，直到生產線盡頭為止
  - 另一種稱為扇出(fan out)/扇入(fan in), 也就是將資料分給多個goroutine處理，或者同時從多個goroutine接收訊息

- 但不管是什麼模式，基本都由以下部分組成：
  - 資料來源(管線模式的第一階段)
  - 從來源內部收集資料
  - 內部處理
  - Sink：將其他goroutine之結果合併的最終階段
- 在這些模式中，有些goroutine必須等待其他的goroutine處理完所有資料才能繼續，此外他們也不見得會像前面的範例那樣，明確知道有多少資料近來；要是等待的資料數量不對，goroutine很容易變成死結
- 幸好，go語言提供了內建的close()函式來關閉通道，讓我們決定goroutine該用到什麼時候

## 16-5-1 通道緩衝區與通道關閉:close()

# 緩衝區對讀寫的影響

- 前面已經看過，定義通道時可以指定緩衝區大小，也可以不指定：

```
ch1 := make(chan int)
```

```
ch2 := make(chan int, 10)
```

- 緩衝區(**buffer**)就像容器, 所以得先準備好(初始化)才能存放資料
- 而通道的操作是“阻斷式(**blocking**)”的，也就是當你嘗試從通道讀取資料時，其他存取通道的goroutine就會暫停執行和等待，與前面介紹的互斥鎖效果一樣



- 不過，這種阻斷性質會有些額外的影響，下面來看一個例子
- 在之前的練習中，goroutine可以對通道寫入值：

ch <- 值

- 若通道ch沒有緩衝區，又沒有其他goroutine能讀取值，這個寫入動作就會卡住
- 同樣的，若寫入值的次數超過ch的緩衝區長度，也會發生相同的事：  
(看下面的例子)

```
func main() {  
    ch := make(chan int, 2)  
    ch <- 1  
    ch <- 2  
    ch <- 3  
    fmt.Println(<-ch)  
    fmt.Println(<-ch)  
}
```

You, 現在

- 執行結果：

```
fatal error: all goroutines are asleep - deadlock!

goroutine 1 [chan send]:
main.main()
    d:/git/Golang/myproject/test/test.go:9 +0x5c
exit status 2
```

- 因為函式在寫入兩次值後，第三次就超過緩衝區的長度，使main()陷入了死結
- 若讀取通道的次數超過緩衝區長度，也會發生死結：

```
func main() {  
    ch := make(chan int, 2)  
    ch <- 1  
    ch <- 2  
    fmt.Println(<-ch)  
    fmt.Println(<-ch)  
    fmt.Println(<-ch)  
}
```

- 執行結果：

```
1  
2  
fatal error: all goroutines are asleep - deadlock!  
  
goroutine 1 [chan receive]:  
main.main()  
    d:/git/Golang/myproject/test/test.go:11 +0x125  
exit status 2
```

# 無緩衝區的通道

- 前面使用無緩衝區的通道時，只要寫入與讀出的動作同時存在就不會發生死結；但來看看底下的例子：

```
func readThem(ch chan int) {  
    for {  
        fmt.Println(<-ch) //不斷地取出值  
    }  
}  
  
func main() {  
    ch := make(chan int)  
    go readThem(ch)  
    ch <- 1  
    ch <- 2  
    ch <- 3  
}
```

- 理想上，執行結果應該是：

```
PS D:\git\Golang> go run "d:\git\Golang\myproject\test\test.go"
1
2
3
```

- 但執行多次之後，出現的數字有可能少於3個：

```
PS D:\git\Golang> go run "d:\git\Golang\myproject\test\test.go"
1
2
3
PS D:\git\Golang> go run "d:\git\Golang\myproject\test\test.go"
1
2
PS D:\git\Golang> 
```

- 當放入的數字越多，缺少數字的機率就越高，因為main()有可能在readThem()讀完所有數字前就結束了 (因為main一結束readThem也會一併關閉)
- 也就是說，雖然沒有緩衝區可以避免死結的發生，但可能會造成資料遺失的問題

# 用range讀取通道直到它被關閉

- 若不知道有多少值會傳入，也想確保goroutine能一直讀取該通道，可以使用for range迴圈：

```
for i := range ch
```

- 迴圈會不斷從通道ch取值和放進變數i，若無值可取就會等待，為了避免無限等待造成死結，必須用close()關閉通道，好讓for range迴圈知道該中斷了：

```
close(ch)
```



- 若通道被關閉，其實還是可以讀取值，所以若呼叫`close()`時`ch`仍有資料，`for range`仍會讀完所有值再結束
- 注意：通道不是檔案或HTTP物件，沒有一定要呼叫`close`，呼叫`close`的目的是通知其他goroutine這個通道不會再傳入新值；因此關閉通道的責任通常由傳值給通道的函式負責
- 下面來看完整範例，程式內會用一個無緩衝區通道，但藉由`WaitGroup`和通道來控制並行性運算的流程：

```
1 package main
2
3 import (
4     "fmt"
5     "sync"
6 )
7
8 func readThem(ch chan int, wg *sync.WaitGroup) {
9     defer wg.Done() //在結束時對WaitGroup回報
10    for i := range ch { //一直讀取ch, 直到他被關閉且無值可取
11        fmt.Println(i)
12    }
13 }
14
15 func main() {
16     wg := &sync.WaitGroup{}
17     wg.Add(1)
18     ch := make(chan int)
19     go readThem(ch, wg)
20     ch <- 1
21     ch <- 2
22     ch <- 3
23     ch <- 4
24     ch <- 5
25     close(ch) //值傳完就關閉通道
26     wg.Wait() //等待一個goroutine結束
27 }
28
```

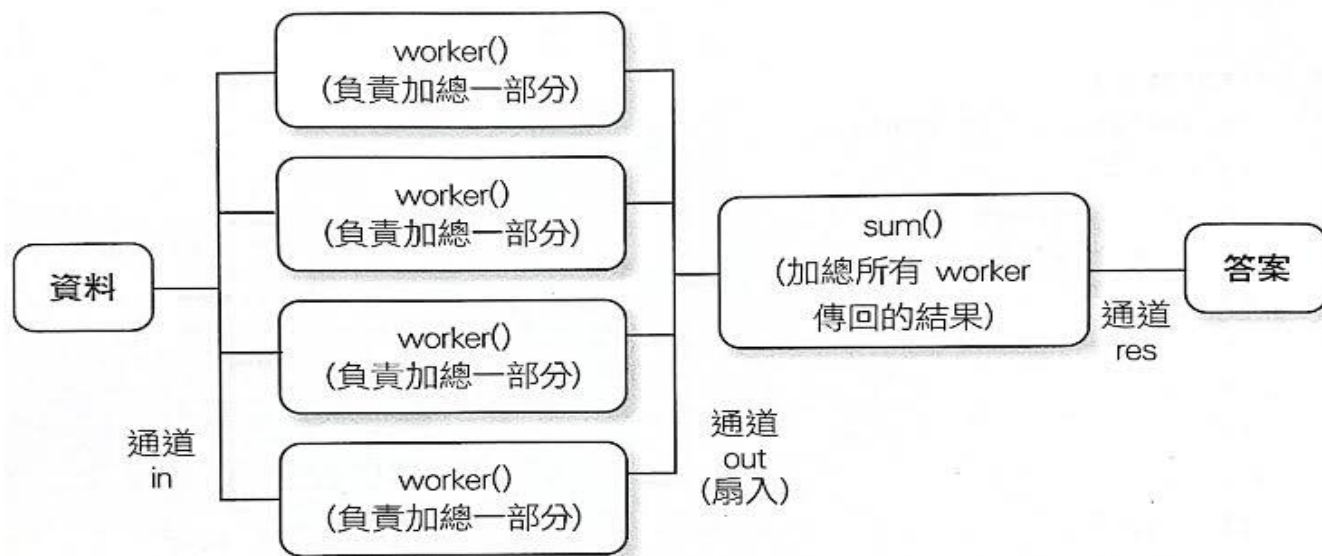
- 執行結果：

```
1
2
3
4
5
```

- `main()`函式會對通道提供5個值，然後關閉通道，這使得以goroutine形式啟動的`readThem`會讀出這些值然後結束
- 為了確保`readThem()`能讀完所有值，使用`WaitGroup`強迫`main()`等待`readThem()`

# 練習：在多個goroutine間分攤任務

- 這個練習中，要來看如何讓多個goroutine分攤數字加總的工作，再將所有數字交給單一的goroutine彙整，也就是前面說的“管線”和“扇入”模式：



- `woker()`和`sum()`都會用`for range`來讀取通道並等待，直到有人關閉他們讀取的通道為止
- 它們都不曉得自己會收到多少值，但起碼知道何時該結束，而透過這種方式，我們可以將任務分給多個`goroutine`，不必管誰該負責多少範圍

```
1  package main
2
3  import (
4      "fmt"
5      "sync"
6      "time"
7  )
8
9  func worker(in, out chan int, wg *sync.WaitGroup) {
10      defer wg.Done()
11      sum := 0
12      for i := range in { //讀取通道直到它被關閉
13          sum += i
14          time.Sleep(time.Millisecond) //模擬資料處理時間
15      }
16      out <- sum
17  }
18
19  //負責彙整woker()計算結果的goroutine
20  //注意sum()接收的in和out不會跟worker()一樣
21  func sum(in, out chan int) {
22      sum := 0
23      for i := range in { //讀取通道直到它被關閉
24          sum += i
25      }
26      out <- sum
27  }
28
```

```
29 func work(workers, from, to int) int {
30     wg := &sync.WaitGroup{}
31     wg.Add(workers)
32
33     in := make(chan int, (to-from)+1)
34     out := make(chan int, workers)
35     res := make(chan int, 1)
36
37     for i := 0; i < workers; i++ {
38         go worker(in, out, wg) //產生指定數量的worker()
39     }
40     go sum(out, res) //執行sum()
41
42     for i := from; i <= to; i++ {
43         in <- i //提供資料給各worker()
44     }
45     close(in) //關閉in(通知所有worker停止讀取)
46     wg.Wait() //等待所有worker結束
47     close(out) //關閉out(通知sum停止讀值)
48
49     return <-res //讀取並傳回最終加總
50 }
51
52 func main() {
53     res := work(4, 1, 100) //建立4個worker, 計算1~100加總
54     fmt.Println(res)
55 }
56
```

執行結果：

```
PS D:\git\Golang> go run "d:\git\Golang\ch16\16-5-1(2)\main.go"  
5050
```



## 1—5-2 使用通道訊息來等待goroutine結束

- 還有另一個方式能等待另一個goroutine結束，就是在通道中放入明確的通知
- 在以下練習中，我們要讓一個goroutine傳送訊息給另一個函式，並由後者印出它
- 不僅如此，我們希望讓傳送資料的goroutine知道對方什麼時候印完所有訊息 (不使用WaitGroup)

```
1  package main
2
3  import (
4      "fmt"
5      "strings"
6  )
7
8  func readThem(in chan string, done chan bool) {
9      for i := range in { //讀取通道in直到它關閉
10         |   fmt.Println(strings.ToUpper(i)) //把字母轉大寫印出
11     }
12     done <- true //傳送結束訊號(值不重要)
13 }
14
```

```
15 func main() {
16     strs := []string{"a", "b", "c", "d", "e", "f", "g", "h", "i", "j"}
17     workers := 4
18     in := make(chan string, len(strs))
19     done := make(chan bool, workers)
20
21     for i := 0; i < workers; i++ {
22         go readThem(in, done) //建立4個goroutine
23     }
24
25     for _, s := range strs { //將字母傳入通道in
26         in <- s
27     }
28     close(in) //關閉通道in
29
30     for i := 0; i < workers; i++ {
31         <-done //等待收到4個停止訊號
32     }
33 }
34
```

# 執行結果

```
C  
E  
F  
G  
H  
I  
J  
D  
A  
B
```

- 這個做法類似WaitGroup, 只不過使用的是另一個通道
- Main()會在做後讀取通道done, 且必須讀到特定的值才代表所有goroutine執行完畢

## 16-5-3 使用通道傳送取消訊號

- 有時候，你可能會讓一群goroutine進行作業，但其實你只需一個計算結果，也就是其中一個goroutine回傳結果後就取消其餘的任務，避免多餘的計算
- 這是另一種並行性運算的設計模式，稱為“明確的取消”(explicit concellation)
- 下面的範例要讓幾個goroutine產生0~99間的隨機整數，若外部函式收到大於90就回傳該數字，並取消所有goroutine
- 為了做到這點，可使用select...case敘述

# Select敘述：扮演取消信號的通道

- `selec...case`看起來很像`switch...case`, 但`selcet`是專門搭配通道的：

```
select {  
  case msg1 := <-c1 :  
    //可從通道msg1接收值  
  case msg2 := <-c2 :  
    //可從通道msg2接收值  
  case msg3 := <-c3 :  
    //可從通道msg3接收值  
  default :  
    //上述條件皆不成立時  
}
```

- **go**語言會尋找一個能夠執行動作的**case**, 執行對通道的讀取或寫入, 以及**case**下面的其他程式碼
- 假如同時有多個操作是允許的, 就會隨機選一個; 若沒有符合條件, 那麼它會等到任一**case**可執行為止
- 這跟傳送取消訊號有什麼關係呢? 通道有趣的一點就是: 當一個通道被關閉時, 它會進入可讀值狀態(**ready to receive**)
- 以上面的例子來說, 若**done**通道是無緩衝區的通道, 正常情況下它會被**select**敘述略過, 因為讀不到任何東西; 但一旦呼叫了**close(done)**, **done**變成可讀值狀態, **case<-done:**這一行就會成立, 使得它有機會被**select**執行, 這時就能用**return**或其他方式結束**goroutine**函式了

- 換言之，“關閉done通道”這件事就是取消訊號
- 你當然可以像前面一樣設立一個有緩衝區的通道，並傳送特定值給他們，不過這樣你就得事先知道有多少個goroutine在跑；若是使用關閉通道的方式，不管有多少goroutine使用這個通道都會收到信號
- 後面我們簡單談到context時還會看到類似的東西，但我們先看下面的範例：

在這裡，幾個worker()會不斷產生1~100的隨機數字，而work()只要從其中一個收到大於90的數字，就會把它回傳給main()，並用關閉通道的方式通知所有goroutine結束



```
1  package main
2
3  import (
4      "fmt"
5      "math/rand"
6      "time"
7  )
8
9  func worker(id int, out chan int, done chan bool) {
10     for {
11         n := rand.Intn(100) //產生0~99的隨機整數
12         select {
13             case out <- n: //把隨機整數傳給通道out
14                 fmt.Printf("ID %d 傳送 %d\n", id, n)
15             case <-done: //若通道done被關閉而變成可讀
16                 fmt.Printf("ID %d 結束\n", id)
17                 return //結束goroutine
18             }
19             time.Sleep(time.Millisecond) //模擬運算時間
20         }
21     }
22 }
```

```
23 func work(workers, from, to int) int {
24     out := make(chan int, workers)
25     done := make(chan bool)
26     defer close(done) //等work()函式結束時關閉done通道(送出取消訊號)
27
28     for i := 0; i < workers; i++ {
29         go worker(i, out, done) //建立若干goroutine
30     }
31
32     res := 0
33     for i := range out {
34         if i >= 90 { //若有goroutine回傳值大於90
35             res = i
36             break //結束work()
37         }
38     }
39     return res //回傳答案給main
40 }
41
42 func main() {
43     rand.Seed(time.Now().UnixNano())
44     res := work(4, 1, 100)
45     fmt.Println("答案:", res)
46     time.Sleep(time.Second) //等待1秒, 好看到所有goroutine跑完
47 }
48
```

- 我們在main()結尾加上一秒延遲，好觀察各個goroutine會在什麼時候結束(以免main()結束就關閉他們)
- 執行結果如下，可以看到各個goroutine確實收到了取消信號：

```
ID 3 傳送 57
ID 1 傳送 14
ID 2 傳送 89
ID 0 傳送 54
ID 1 傳送 18
ID 0 傳送 81
ID 3 傳送 62
ID 2 傳送 42
ID 2 傳送 0
ID 3 傳送 29
ID 0 傳送 90
ID 1 傳送 23
答案: 90
ID 3 結束
ID 0 結束
ID 2 結束
ID 1 傳送 11
ID 1 結束
```

← 第一個符合需求的答案

← 其他goroutine還在產生數字

← goroutine發現done通道關閉, 停止產生數字

← 有些goroutine結束的較慢, 所以繼續回傳數字

## 16-5-4 使用函式來產生通道

- 以上的練習，都是先建立通道，再用參數形式把他們傳給goroutine函式
- 其實，你也可以用函式來傳回通道，並在函式內產生goroutine
- 以下是一個例子：

```
1 package main
2
3 import "fmt"
4
5 func doSomething() (chan int, chan bool) {
6     //建立通道
7     in, out := make(chan int), make(chan bool)
8     //以匿名函式啟動goroutine
9     go func() {
10         for i := range in {
11             fmt.Println(i)
12         }
13         out <- true //通知作業結束
14     }()
15     return in, out //傳回通道
16 }
17
18 func main() {
19     in, out := doSomething() //從函式取得通道
20     in <- 1
21     in <- 2
22     in <- 3
23     close(in)
24     <-out //等待goroutine結束
25 }
26
```

- 這樣一來就不需要在`main()`自行呼叫`goroutine`; 而且對`doSomething()`內的匿名函式來說, 他存取的`in/out`通道就位於父函式的範圍中, 不必再用參數傳來傳去了
- 以下為執行結果:

```
1  
2  
3
```

## 16-5-5 限制通道的收發方向

- 預設上，通道是可以雙向操作的(寫入/讀出)，但你也許會想要限制通道只能單向傳送訊息，這時可以在通道的型別加上<-算符來限制其操作方向：

ch1 <-chan int 只能寫入整數的通道

ch2 chan<- int 只能讀取整數的通道



- 這種型別寫法適用於宣告通道變數，以及用在函式的參數/傳回值型別
- 若試圖對一個只讀取的通道寫入訊息，編譯時就會產生錯誤
- 若修改前面16-5-2的練習，可以改寫如下：

```
8 func readThem(in <-chan string, out chan<- string) {  
9     for i := range in {  
10         fmt.Println(strings.ToUpper(i)) //in只能用來讀取  
11     }  
12     out <- "done" //out只能用來寫入  
13 }  
14
```

## 15-5-6 將結構方法當成goroutine

- 在本章中，一般都用函式當成**goroutine**，但結構方法本身也是函式，只不過帶有接收器而已，你一樣能把它變成非同步執行的函式
- 尤其，你能把多個**goroutine**共用的資料/通道等都封裝在一個結構變數中，不必擔心要把它們傳來傳去 (若要打造像**HTTP**伺服器訪客計數器之類的東西，這樣就很有用)

# 練習：使用結構來執行goroutine

- 在下面的練習中，要定義一個**Workers**結構，它會啟動多個 `goroutine(worker)` 來做數字加總，並統計最終結果
- 這裡會使用通道和互斥鎖來確保資料安全

```
1  package main
2
3  import (
4      "fmt"
5      "sync"
6  )
7
8  type Workers struct { //Worker結構
9      in, out  chan int    //輸入和輸出通道
10     workerNum int        //最大goroutine數
11     mtx      sync.Mutex //互斥鎖(不需用指標)
12 }
13
14 //初始化Workers結構，建立通道及互斥鎖，啟動goroutine
15 func (w *Workers) init(maxWorkers, maxData int) {
16     //建立通道
17     w.in, w.out = make(chan int, maxData), make(chan int)
18     //建立互斥鎖
19     w.mtx = sync.Mutex{}
20     for i := 0; i < maxWorkers; i++ {
21         w.mtx.Lock()
22         w.workerNum++ //紀錄
23         w.mtx.Unlock()
24         go w.readThem() //啟動goroutine
25     }
26 }
27
```

```
28 //輸入資料
29 func (w *Workers) addData(data int) {
30     w.in <- data
31 }
32
33 //讀出資料
34 func (w *Workers) readThem() {
35     sum := 0
36     for i := range w.in { //讀取通道in直到關閉和無值
37         sum += i
38     }
39     w.out <- sum //將自己部份的加總值傳給通道out
40
41     //任務結束，減少goroutine的紀錄數量
42     w.mtx.Lock()
43     w.workerNum--
44     w.mtx.Unlock()
45     if w.workerNum <= 0 { //減到0時關閉通道out
46         close(w.out)
47     }
48 }
49
```

```
50 //取得結果
51 func (w *Workers) gatherResult() int {
52     close(w.in) //關閉通道in
53     total := 0
54     for i := range w.out { //讀取通道out直到關閉和無值
55         total += i
56     }
57     return total
58 }
59
60 func main() {
61     maxWorkers := 10
62     maxData := 100
63     workers := Workers{} //建立Workers節構
64     workers.init(maxWorkers, maxData) //初始化Workers
65
66     for i := 1; i <= maxData; i++ {
67         workers.addData(i) //新增資料
68     }
69     res := workers.gatherResult() //取得結果
70     fmt.Println(res)
71 }
72
```

執行結果：

```
PS D:\git\Golang> go run "d:\git\Golang\ch16\16-5-6\main.go"  
5050
```

## 16-6 context套件

- 以上我們看到了如何運用並行性運算，使用WaitGroup或通道關閉與否來等待運算結束
- 不過你或許在某些程式碼，尤其是HTTP遠端呼叫相關的程式，看過context套件有關的參數
- 比如，http.NewRequestWithContext()看起來跟NewRequest()很像，但多了一個context參數：

```
func NewRequestWithContext(ctx context, method, url string, body io.Reader)
(*Request, error)
```



- **context**是個結構變數，可以拿來在一系列**goroutine**的呼叫過程中傳遞，也和通道一樣具備並行性運算的安全性
- **context**裡面可能有值，也可能是空的；他雖是容器，但其用意並非在函式間傳遞資料(用通道即可)，而是讓在需要時對**goroutine**送出訊號，停止他們的運作
- 其實**context**的使用方式跟前面用通道送出取消訊號很像，等一下就會看到用法；**context**也用在**go**語言的**http**套件中，可以用來取消客戶端送出的請求(比如回應時間過長被視為逾時的時候)

# 練習：用context取消goroutine執行

```
1  package main
2
3  import (
4      "context"
5      "fmt"
6      "time"
7  )
8
9  func countNumbers(c context.Context, out chan int) {
10     i := 0
11     for { //無窮迴圈
12         select {
13             case <-c.Done(): //收到取消信號，船值給out
14                 out <- i
15                 return
16             default: //正常情況下，每100毫秒讓計數器+1
17                 time.Sleep(time.Millisecond * 100)
18                 i++
19         }
20     }
21 }
22
```

```
23 func main() {
24     out := make(chan int)
25
26     //建立一個空context結構
27     c := context.TODO()
28     //延伸一個可取消的context並取得取消函式
29     cl, cancel := context.WithCancel(c)
30
31     go countNumbers(cl, out) //將context傳給goroutine
32
33     time.Sleep(time.Millisecond * 100 * 5) //等待500毫秒
34     cancel()                               //呼叫context提供的取消函式
35
36     fmt.Println(<-out) //印出out內的值
37 }
38
```

- `contextTODO()`會傳回一個空值，不為`nil`的`context`結構，然後我們可以使用`WithCancel()`函式把它轉成一個帶有取消信號功能的`context`：

```
func WithCancel(parent Context) (ctx Context, cancel CancelFunc)
```

- 這個`context`結構的方法`Done()`會傳回一個通道，此通道的作用跟16-5-3範例中的通道`done`一樣
- 注意`WithCancel()`的第二個參數為一個函式：只要呼叫這個函式，`context`的`Done()`通道便會關閉，使`countNumbers()`內的`select...case`會執行`<-c.Done()`的區塊

- 在本練習中，`countNumbers()`每100毫秒會自行讓計數器加1，但我們在500毫秒後就打斷他，因此最終印出結果為 5
- 由此可見，任何有接收同一個context結構的goroutine, 可以透過context來一併關閉，不論層級或數量多寡

# 使用context.WithTimeout

- 若要在指定的時間後結束所有goroutine, 可以使用context.WithTimeout()來產生context結構：

```
func main() {  
    out := make(chan int)  
  
    c := context.TODO()  
    c1, cancel := context.WithTimeout(c, time.Millisecond*500)  
    defer cancel()  
  
    go countNumbers(c1, out)  
  
    fmt.Println(<-out) //印出out內的值  
}
```

- `cl`會在指定時間後自動關閉其通道`Done()`, 使`countNumbers()`中止
- 注意這裡仍須用`defer`延後呼叫`cancel()`, 這樣才能釋放`context`占用的相關資源

本章結束