CH10 時間處理

10-1 前言

- 前一章介紹了go語言的基本除錯, 其中在除錯輸出的log訊息中，通常會包含時間資料，但你或許會想用自己的格式輸出時間; 或者會想衡量一段程式碼的平均時間,以判斷效能

- 而這一切都牽涉到時間處理，在go語言中，time套件就負責主要的時間處理功能

10-2 建立時間資料

10-2-1 取得系統時間

- 在go語言中，時間資料會是time.Time結構型別
- 你能建立一個變數，紀錄特定的時間，並使用該結構的各種方法來抽出時間時間資料的不同部分
- 在所有時間資料中，最容易取得的就是系統當前時間：

```go
package main

import (
    "fmt"
    "time"
)

func main() {
    start := time.Now()
    fmt.Println("程式開始時間:", start)
    fmt.Println("資料處理中...")
    time.Sleep(2 * time.Second)
    end := time.Now()
    fmt.Println("程式結束時間:", end)
}
```

執行結果：

```
PS D:\git\Golang> go run "d:\git\Golang\ch10\10-2-1.go"
程式開始時間: 2022-08-09 14:44:41.9272346 +0800 CST m=+0.002127701
資料處理中...
程式結束時間: 2022-08-09 14:44:43.9555453 +0800 CST m=+2.030438401
```

- 在以上時間值中，包含以下兩個部分：

    1. Wall clock : 2022-08-09 14:44:41.9272346 +0800 CST
    2. Monotonic clock : m=+0.002127701

- Wall clock 就是電腦系統時間，會透過NTP(network time protocol,網路鞋時間協定)來同步

- Monotonic clock是程序啟動後經過的時間(單位為奈秒), 它不會跟外界同步，其為一功用就是拿來比較時間

- Go語言的時間值一定包含wall clock, 但不一定有monotonic clock值

10-2-2 取得時間資料中的特定項目

- 接著來思考以下情境：你的上司交代你一項任務，用go語言寫一支小程式來設式公司的網頁應用程式

- 這支小程式平常只會做幾分鐘的簡易測試，對系統影像很小，不過伺服器在每周一凌晨~3點會停機，以便上線新版本程式；上線過程約1小時，而完全測試也需要1小時左右，因此你只能在凌晨0~2點之間執行全功能測試

- 可想而知，你的程式必須判斷現在的時間是否允許進行全功能測試，以免對系統造成負擔：

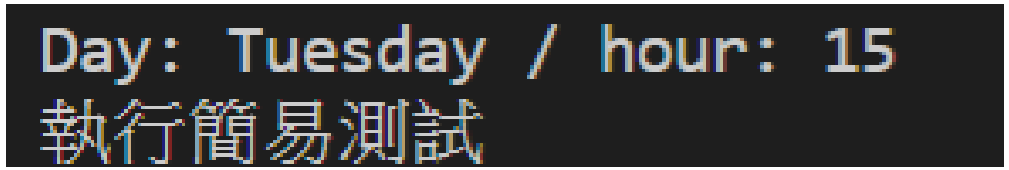
```go
package main

import (
    "fmt"
    "time"
)

func main() {
    now := time.Now()     //取得當下時間
    day := now.Weekday() //取得星期幾
    // day = time.Monday
    hour := now.Hour() //取得小時
    // hour = 1
    fmt.Println("Day:", day, "/ hour:", hour)
    if day.String() == "Monday" && (hour >= 0 && hour < 2) {
        fmt.Println("執行全功能測試")
    } else {
        fmt.Println("執行簡易測試")
    }
}
```

- 程式先取得當下的執行時間，然後用now(time.Time 結構)的 Weekday()方法取得今天是星期幾, 用Hour()方法取得小時

- 因此不管你何時執行程式，他都會根據當下的時間判斷現在該做哪種測試：



- 下面是time.Time結構的常用方法與其傳回值：

Date()	傳回年 (int)、月 (Month 型別)、日 (int) (Month 型別本質為 int, 1~12 代表 1~12 月)
Clock()	傳回時 (int)、分 (int)、秒 (int)
YearDay()	傳回是該年第幾天 (int)
Year()	傳回年 (int)
Month()	傳回月 (Month 型別)
Month().String()	傳回月名稱 (string)
Day()	傳回日 (int)
Weekday()	傳回星期幾 (Weekday 型別, 本質為 int, 0~6 代表星期日~星期六)
Weekday().String()	傳回星期幾名稱 (string)
Hour()	傳回小時 (int)
Minute()	傳回分鐘 (int)
Second()	傳回秒 (int)
Nanosecond()	傳回奈秒 (int)
Unix()	傳回 Unix epoch 時間 (int64), 即從 1970 年 1 月 1 日 0 時到這個時間經過的總秒數
UnixNano()	同上, 但傳回奈秒數 (1 秒 = 10 億奈秒)
String()	將整個時間轉成字串

轉換時間資料為字串

- 再看一個例子，現在我們想用go語言建立特定的log檔名，好讓日誌能反映應用程式名稱/程式所做的事，以及內容涵蓋的日期：

  程式名稱_行為_年_月_日.log

- 在前一個範例中，時間資料的星期即可以用Weekday().String()轉成字串，然而並不是time.Time的所有方法都能這樣做

- Time結構帶多數方法傳回的資料就是int型別，而go語言中若想將int轉為string, 你得使用strconv套件提供的轉換功能：

```go
package main

import (
    "fmt"
    "strconv"
    "time"
)

func main() {
    appName := "HTTPCHECKER"
    action := "BASIC"
    date := time.Now()
    logFileName := appName + "_" + action + "_" + strconv.Itoa(date.Year()) + "_" + date.
        Month().String() + "_" + strconv.Itoa(date.Day()) + ".log"
    fmt.Println("log 檔名稱:", logFileName)
}
```

- strconv.Itoa()函式其實也會傳回error值，但既然時間資料的特定部分已知一定是int型別，我們就不必特地檢查轉換是否會失敗

- 輸出會像這樣：

```
log 檔名稱: HTTPCHECKER_BASIC_2022_August_9.log
```

10-3 時間值的格式化

10-3-1 將時間轉成指定格式的字串

- time.Time結構的Format()方法可以將時間轉成特定格式的字串：

```
func (t time) Format(layout string) string
```

- 參數layout為時間格式字串

- 在time套件對此定義了一系列常數：

```
const (
    ANSIC       ="Mon Jan _2 15:04:05 2006"
    UnixDate    ="Mon Jan _2 15:04:05 MST 2006"
    RubyDate    ="Mon Jan 02 15:04:05 -0700 2006"
    RFC822      ="02 Jan 06 15:04 MST"
    RFC822Z     ="02 Jan 06 15:04 -0700" // RFC822 with numeric zone
    RFC850      ="Monday, 02-Jan-06 15:04:05 MST"
    RFC1123     ="Mon, 02 Jan 2006 15:04:05 MST"
    RFC1123Z    ="Mon, 02 Jan 2006 15:04:05 -0700"// RFC1123 with numeric zone
    RFC3339     ="2006-01-02T15:04:05Z07:00"
    RFC3339Nano ="2006-01-02T15:04:05.99999999Z07:00"
    Kitchen     ="3:04PM"
    // Handy time stamps.
    Stamp      ="Jan _2 15:04:05"
    StampMilli ="Jan _2 15:04:05.000"
    StampMicro ="Jan _2 15:04:05.000000"
    StampNano  ="Jan _2 15:04:05.000000000"
)
```

後面我們會再解釋go語言是如何解讀這些時間格式的

練習：用不同格式輸出時間字串

- 在練習中，我們要試著將time.Now()傳回的系統時間轉成不同格式並印出，分別是ANSIC(美國國家標準時間)格式/Unix系統格式以及網路上常見的RFC3339格式，甚至，我們也要來嘗試自訂時間格式

```go
package main

import (
    "fmt"
    "time"
)

func main() {
    fmt.Println(time.Now().Format(time.ANSIC))
    fmt.Println(time.Now().Format(time.UnixDate))
    fmt.Println(time.Now().Format(time.RFC3339))
    fmt.Println(time.Now().Format("2006/1/2 3:4:5")) //自定格式
}
```

執行結果：

```
Wed Aug 10 14:05:41 2022
Wed Aug 10 14:05:41 CST 2022
2022-08-10T14:05:41+08:00
2022/8/10 2:5:41
```

自訂時間格式

- Go語言使用一個叫做 “魔法參照時間(magical reference date)”的字串來讓你自訂時間格式

- 可能你已經發現，前面的時間常數都有特定的值，例如 Jan 2 15:04:05 2006 -0700 :

月	日	時	分	秒	年	時區
Jan	2	15	04	05	2006	-0700
1	2	3	4	5	6	7

- 也就是說，在這個時間字串中，每個值都會照順序對應到1~7
- Go語言會用這些值來判定個值的位置，使你能用來自訂想要的格式

- 其他一些格式規則包括：
  - 星期必須寫Mon(輸出縮寫)或Monday(完整名稱)
  - 月份寫Jan代表用英文簡寫，Juanuary則用完整名稱，寫1則是以數字顯示
  - 小時寫15代表24時制，寫3代表12時制，可用AM來代表要顯示AM/PM
  - 月/時/分/秒的數字前加上0代表補0，底線則是補空白(不一定有用)
  - 日一定是2位數，年一定是4位數
  - 時區可以不寫，或寫-07, -0700或Z0700，這樣寫並不是設定時區，而是沿用時間資料中的既有時區，你也可寫MST來表示要輸出時區的簡寫

- 下面是一些範例，格式化目標為2021/04/22下午4:44:5 ：

時間格式	輸出
"Mon, 02 Jan 2006 15:04:05 -0700"	Thu, 22 Apr 2021 16:44:05 +0800
"2006 年 1 月 2 日 3 時 4 分 5 秒"	2021 年 4 月 22 日 16 時 44 分 5 秒
"今天是 Mon 2006-01-02"	今天是 Thu 2021-04-22
"現在時間: Monday 15:04:05 MST!"	現在時間: Thursday 16:44:05 CST!
"PM 03:04:05, January/2, Mon 2006 (GMT-0700)"	PM 04:44:05, April/22, Thu 2021 (GMT+0800)

10-3-2 將特定格式的時間字串轉換成時間值

- go語言也允許你將符合特定格式的時間字串轉成time.Time結構 :
- Parse()會嘗試以layout參數指定的格式轉換value中的日期時間
- 若格式不符則轉換失敗 , Parse()會傳回一個存有時間零值的Time結構 , 以及不為nil的error值
- 補充 : 時間零值是January 1, year 1, 00:00:00 UTC, 可以用IsZero()方法來檢驗是否Time結構為此零值

練習：將時間字串轉成time.Time結構

- 在這個練習中，會將前一個練習的結果重新轉回time.Time節構

```go
package main

import (
    "fmt"
    "time"
)

func main() {
    t1, err := time.Parse(time.ANSIC, "Thu Apr 22 16:44:05 2021")
    //美國國家時間格式
    if err != nil {
        fmt.Println(err)
    }
    fmt.Println("from ANSIC :", t1)

    t2, err := time.Parse(time.UnixDate, "Thu Apr 22 16:44:05 CST 2021")
    //Unix系統格式
    if err != nil {
        fmt.Println(err)
    }
    fmt.Println("from UnixDate  :", t2)

```

```
23      t3, err := time.Parse(time.RFC3339, "2021-04-22T16:44:05+08:00")
24      //RFC3339格式
25      if err != nil {
26          fmt.Println(err)
27      }
28      fmt.Println("from RFC3339   :", t3)
29
30      t4, err := time.Parse("2006/1/2 3:4:5", "2021/4/22 4:44:5")
31      //自訂格式
32      if err != nil {
33          fmt.Println(err)
34      }
35      fmt.Println("from custom    :", t4)
36  }
```

執行結果：

```
from ANSIC       : 2021-04-22 16:44:05 +0000 UTC
from UnixDate    : 2021-04-22 16:44:05 +0800 CST
from RFC3339     : 2021-04-22 16:44:05 +0800 CST
from custom      : 2021-04-22 04:44:05 +0000 UTC
```

- 各位會發現轉換出來的Time結構內容，似乎和前一個練習的結果相同，但仍有些差異

- 這是因為Unix系統格式時間和RFC3339格式含有時區資訊，但ANSIC時間沒有，因此被time套件認定為UTC標準時間

- 接著，我們自訂的時間格式用了12時制，但time套件將它解讀成24時制，解決方式是以24時制表達時間，或用PM標示時間是上午會下午：

```
t4, err := time.Parse("2006/1/2 PM 3:4:5", "2021/4/22 PM 4:44:5")
//自訂格式
```

10-4 時間值的管理

10-4-1 建立和增減時間值

- 除了用time.Now()取得系統當下的時間，go語言允許你建立代表特定時間的time.Time結構：

```
func Date(year int, month Month, day, hour, min, sec, nsec int, loc *location) Time
```

- 這個方式是依次傳入年/月/日/時/分/秒/奈秒，最後一個則是時區(time.Location型別)，然後回傳Time結構

- 在建立Time結構後，你就可以用它的AddDate()方法來增減日期：

```
func (t Time) AddDate(years int, months int, days int) Time
```

建立並改變時間值

```go
package main

import (
    "fmt"
    "time"
)

func main() {
    date1 := time.Date(2021, 4, 22, 16, 44, 05, 324359102, time.UTC)
    fmt.Println(date1)
    date2 := time.Date(2021, 4, 22, 16, 44, 05, 324359102, time.Local)
    fmt.Println(date2)
    date3 := date2.AddDate(-1, 3, 5) //減1年, 加3個月又5天
    fmt.Println(date3)
}
```

執行結果

```
2021-04-22 16:44:05.324359102 +0000 UTC
2021-04-22 16:44:05.324359102 +0800 CST
2020-07-27 16:44:05.324359102 +0800 CST
```

10-4-2 設定時區來取得新時間值

- 除了使用New()建立Time結構時設定時區，你也可以用Time結構本身的In()來指定時區，並傳回一個新時間值：

```
func (t Time) In(loc *Location) Time
```

- 如前面看過的，time套件中的時區是time.Location結構型別，之前我們使用的time.UTC和time.Local都屬於這種型別

- 若想使用特定的時區，甚至建立自訂時區，有以下兩種方式：
  - `func LoadLocation(name string) (*Location, error)`
  - `func FixedZone(name string, offset int) *Location`
- LoadLocation()要傳入一個現有的IANA時區名稱，以此來建立時區結構(失敗時傳回不為nil的error)
- FixedZone()則以UTC時區為準，加減offset填入的秒數後，傳回一個以name參數為名稱的自訂時區

練習：設定不同時區

```go
package main

import (
    "fmt"
    "time"
)

func displayTimeZone(t time.Time) {
    fmt.Print("Time: ", t, "\nTimezone: ", t.Location(), "\n\n")
}

```

```go
func main() {
    //本地時間
    date := time.Date(2021, 4, 22, 16, 44, 05, 324359102, time.Local)
    //設為美國紐約時區
    timeZone1, _ := time.LoadLocation("America/New_York")
    //美國紐約時區
    remoteTime1 := date.In(timeZone1)
    //設為澳洲雪梨時區
    timeZone2, _ := time.LoadLocation("Australia/Sydney")
    //澳洲雪梨時區
    remoteTime2 := date.In(timeZone2)
    //自訂時區
    timeZone3 := time.FixedZone("My TimeZone", -1*60*60)
    //自訂時區, 即UTC時區減1小時
    remoteTime3 := date.In(timeZone3)

    displayTimeZone(date)
    displayTimeZone(remoteTime1)
    displayTimeZone(remoteTime2)
    displayTimeZone(remoteTime3)
}
```

執行結果：

```
Time: 2021-04-22 16:44:05.324359102 +0800 CST
Timezone: Local

Time: 2021-04-22 04:44:05.324359102 -0400 EDT
Timezone: America/New_York

Time: 2021-04-22 18:44:05.324359102 +1000 AEST
Timezone: Australia/Sydney

Time: 2021-04-22 07:44:05.324359102 -0100 My TimeZone
Timezone: My TimeZone
```

10-5 時間值的比較與長度處理

10-5-1 比較時間

- 有的時候，你或許需要確保你的go程式必須在特定時間執行特定任務

- 與其像本章開頭的例子個別比較時間值的各個部分，time套件提供了更容易的方式來判斷兩個時間的先後順序

練習：比較時間順序

- 下面我們要建立一個時間值當作門檻，並使用Time結構的Equal(), Before(), After() 方法來檢視現在的系統時間是否等於/小於/大於這個門檻

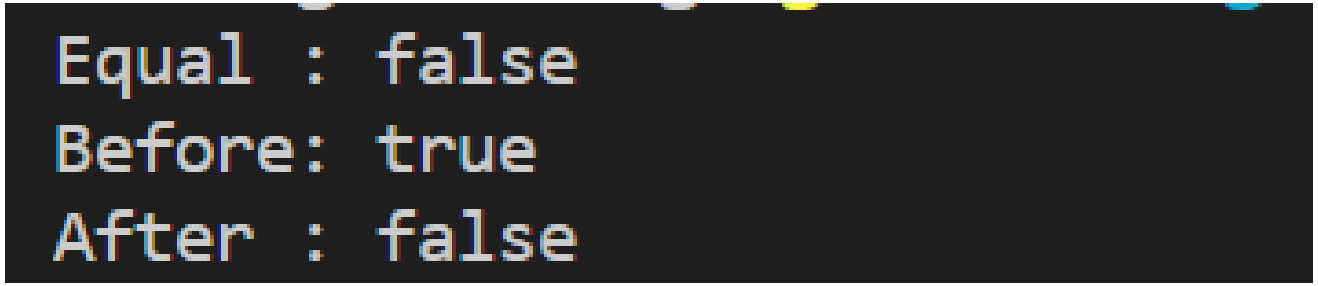
```go
package main

import (
    "fmt"
    "time"
)

func main() {
    date := time.Date(2050, 12, 31, 0, 0, 0, 0, time.Local)

    fmt.Println("Equal :", time.Now().Equal(date))
    fmt.Println("Before:", time.Now().Before(date))
    fmt.Println("After :", time.Now().After(date))
}
```

執行結果：



10-5-2 用時間長度來改變時間

- 前一節中，AddDate()只能用來更動日期

- 若要做出時/分/秒，甚至小於1秒的改變，必須使用時間長度值(time.Duration結構)來搭配時間值的Add()方法：

```
func (t Time) Add(d Duration) Time
```

- time.Duration是自訂型別，代表時間的變量，或者說兩個時間值之間的差異，其底下的型別是int64

- 時間長度常數：
  - time套件內定義了以下常數，讓使用者能更輕易地建立想要的時間長度:

```
const (
    Nanosecond  Duration = 1    //奈秒
    Microsecond          = 1000 * Nanosecond  //微秒
    Millisecond          = 1000 * Microsend  //毫秒
    Second               = 1000 * Millisecond  //秒
    Minute               = 60 * Second  //分
    Hour                 = 60 * Minute  //時
)
```

- Duration型別也有以下方法，可以將時間長度值轉換成特定格是：

  - Hours() : 以小時為單位呈現
  - Minute() : 以分為單位呈現
  - Seconds() : 以秒為單位呈現
  - Milliseconds() : 以毫秒為單位呈現
  - Microseconds() : 以微秒為單位呈現
  - Nanoseconds() : 以奈秒為單位呈現
  - String() : 轉為字串(單位為毫秒)

練習：使用Duration改變時間

```go
package main

import (
    "fmt"
    "time"
)

func main() {
    now := time.Now()
    //時間長度1 (360秒, 等於6分鐘)
    duration1 := time.Duration(time.Second * 360)
    //時間長度2 (1小時又30分鐘)
    duration2 := time.Duration(time.Hour*1 + time.Minute*30)
    //顯示時間長度值(以奈秒為單位)
    fmt.Println("Dur1 :", duration1.Nanoseconds(), "ns")
    fmt.Println("DUr2 :", duration2.Nanoseconds(), "ns")

    //取得加上時間長度後的新時間
    date1 := now.Add(duration1)
    date2 := now.Add(duration2)
    fmt.Println("Now  :", now)
    fmt.Println("Date1:", date1)
    fmt.Println("Date2:", date2)
}
```

執行結果：

```
Dur1 : 360000000000 ns
DUr2 : 5400000000000 ns
Now  : 2022-08-12 15:23:50.5245867 +0800 CST m=+0.002686501
Date1: 2022-08-12 15:29:50.5245867 +0800 CST m=+360.002686501
Date2: 2022-08-12 16:53:50.5245867 +0800 CST m=+5400.002686501
```

10-5-3 測量時間長度

- 在現實世界的應用程式中，你可能會需要計算程式執行所耗費的時間

- 想測量程式的執行時間，只需要在程式頭位各取一次當下的系統時間，然後用Sub()方法相減：

```
func (t Time) Sub(u Time) Duration
```

- 若想計算當下到未來某時還有多久可用time.Until(時間值)

練習：測量程式執行時間

- 這個練習中，我們將使用time.Sleep()方法 (讓程式停頓指定時間，接收一個Duration值) 模擬程式運算時間，然後衡量其長度

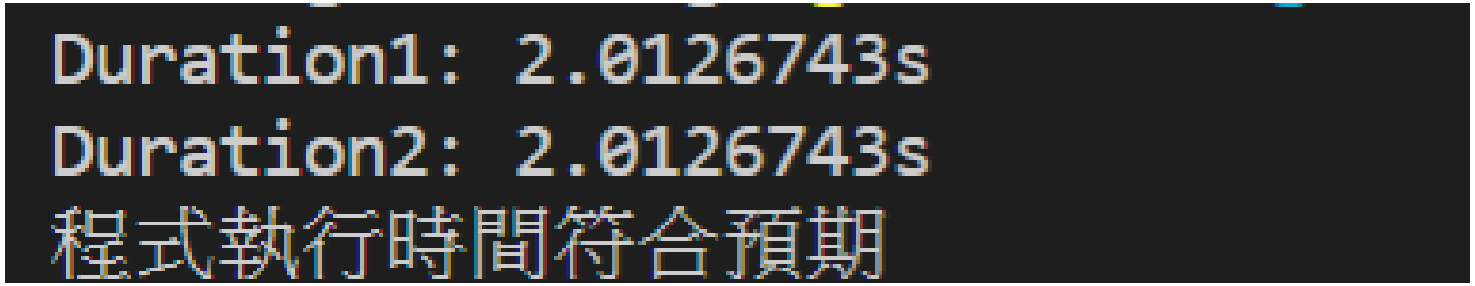
```go
package main

import (
    "fmt"
    "time"
)

func main() {
    start := time.Now()             //第一次取得系統時間
    time.Sleep(time.Second * 2)     //等待2秒
    end := time.Now()               //第二次取得系統時間
    duration1 := end.Sub(start)     //  計算兩指之間的長度
    duration2 := time.Since(start) //計算start到time.Now()的時間長度

    fmt.Println("Duration1:", duration1)
    fmt.Println("Duration2:", duration2)
    if duration1 < time.Duration(time.Millisecond*2500) {
        fmt.Println("程式執行時間符合預期")
    } else {
        fmt.Println("程式執行時間超出預期")
    }
}
```

執行結果



本章結束