

CH12 系統與檔案

12-1 前言

- 在前一章中學習了如何編碼及解碼JSON資料，而go語言實際上提供了豐富的檔案操作功能，支援開啟/建立/修改檔案等等的動作
- 除此之外，程式和系統的互動不侷限於檔案本身；程式可以接收來自使用者的命令列旗標(flag)，以便指定程式該做什麼事；或者，作業系統可能會對程式發出中斷訊息(例如使用者在程式執行時按下ctrl + c)，為了避免程式中斷時無法完成關閉檔案/清理快取等工作，你可以註冊中斷訊息和決定該對它們做什麼處理

12-2 命令列旗標及其引數

使用旗標

- 若使用過終端機及一些命令，你很可能會用到旗標和引數，例如：

```
go build -o .\bin\hello_world.exe main.go
```

- 這行指令使用go build將main.go編譯成\bin子目錄下的hello_world.exe可執行檔，而執行檔的路徑與名稱就是透過旗標-o來指定

- 在第四章中，我們曾使用過os.Args來接收使用者在執行程式時附加的引數
 - 旗標和os.Args引數的差別是，旗標有個名稱和對應值，而且可以是選擇性的，傳入順序也不用固定；換言之，旗標能當作程式行為的開關
- 對於旗標與其引數，go語言提供了flag套件來協助開發者處理他們
 - 第一步是得先定義自己的旗標
 - Flag套件提供了多種可定義旗標的函式，以下是幾種常用的：

```
func Bool(name string, value bool, usage string) *bool // 布林值
func Duration(name string, value time.Duration, usage string) *time.Duration // 時間長度
func Float64(name string, value float64, usage string) *float64 // float64 浮點數
func Int(name string, value int, usage string) *int // int 整數
func Int64(name string, value int64, usage string) *int64 // int64 整數
func String(name string, value string, usage string) *string // 字串
func Uint(name string, value uint, usage string) *uint // uint 正整數
func Uint64(name string, value uint64, usage string) *uint64 // uint64 正整數
```

- 從這些函式名稱與回傳值就能看出，他們的用途是接收特定型別的旗標引數；每個函式都有以下三個參數：
 - **name**：旗標的名稱，型別為字串
 - **value**：旗標的預設值
 - **usage**：說明旗標的用途，通常你設定旗標值錯誤時就會顯示這個內容
- 我們來看一個簡單的例子：

```
1  package main
2
3  import (
4      "flag"
5      "fmt"
6  )
7
8  func main() {
9      //定義一個旗標 -value, 接收整數, 預設值為 -1
10     v := flag.Int("value", -1, "Needs a value for the flag.")
11     flag.Parse()
12     fmt.Println(*v)
13 }
14
```

- `v := flag.int(...)`這行程式碼的意義是：

```
v := flag.Int("value", -1, "Needs a value for the flag.")
```

旗標名稱 預設值

旗標說明

- 這行程式碼搭配`flag.Parse()`，會解析使用者在命令列輸入的旗標－`value`，並將其值以指標整數的形式賦予給`v`，要是沒有這個旗標，`*v`預設為-1
- 當使用者執行程式時，便可用以下方式在主控台內加入旗標：


```
PS D:\git\Golang\ch12> go run 12-2.go
```

```
-1
```

```
PS D:\git\Golang\ch12> go run 12-2.go -value 1
```

```
1
```

```
PS D:\git\Golang\ch12> go run 12-2.go -value 10
```

```
10
```

```
PS D:\git\Golang\ch12> go run 12-2.go --value 50
```

```
50
```

- 現在來試試看將以上範例編譯成執行檔，然後對他套用旗標：

```
PS D:\git\Golang\ch12> go build -o 12-2.exe
PS D:\git\Golang\ch12> .\12-2.exe -value 100
100
```

- 若你在執行程式時加上-h旗標，或旗標的值型別不對，程式會列出可用的旗標/旗標值型別以及其說明，然後直接結束：

```
PS D:\git\Golang\ch12> .\12-2.exe -h
Usage of D:\git\Golang\ch12\12-2.exe:
    -value int
        Needs a value for the flag. (default -1)
PS D:\git\Golang\ch12> .\12-2.exe -value abc
invalid value "abc" for flag -value: parse error
Usage of D:\git\Golang\ch12\12-2.exe:
    -value int
        Needs a value for the flag. (default -1)
```

以旗標來決定程式的執行狀態

- 接下來看一個更複雜的例子：這次程式最多能接收3個旗標，分別是name, age, married
- 有時我們會希望某個旗標是執行應用程式時的必要參數，沒有這個旗標就要提醒使用者
- 這表示你需要謹慎決定旗標的預設值，因為你必須用這個預設值來判斷使用者是否有加上該旗標或給予正確的值：

```
1  package main
2
3  import (
4      "flag"
5      "fmt"
6      "os"
7  )
8
9  func main() {
10     n := flag.String("name", "", "your first name")
11     i := flag.Int("age", -1, "your age")
12     b := flag.Bool("married", false, "are you married?")
13     flag.Parse()
14
15     if *n == "" { //若名子旗標值為空字串，但表使用者沒有加上該旗標，會未給值
16         fmt.Println("Name is required.")
17         flag.PrintDefaults() //印出所有旗標的預設值
18         os.Exit(1)           //結束程式
19     }
20     fmt.Println("Name: ", *n)
21     fmt.Println("Age: ", *i)
22     fmt.Println("Married: ", *b)
23 }
24
```

測試結果：

```
PS D:\git\Golang> go run "d:\git\Golang\ch12\12-2(2)\12-2(2).go" -name John -age 43 -married
Name:  John
Age:  43
Married:  true
PS D:\git\Golang> go run "d:\git\Golang\ch12\12-2(2)\12-2(2).go" -age 43 -married=true
Name is required.
    -age int
        your age (default -1)
    -married
        are you married?
    -name string
        your first name
exit status 1
```

12=3 系統中斷訊號

- 在本章，訊號(signal)指的是作業系統傳給程式或程序的非同通知
- 當程式收到訊號時，他會停下手邊的任務並設法處理這個訊號
- 最常見的情形是，當使用者按下`ctrl + c`，系統會傳送名為**SIGINT**的中斷(interrupt)訊號給程式；或者作業系統要強制終止程式，會傳送**SIGTERM**訊號給他
- 程式收到這些訊號後會立即結束，以go程式來說就是執行`os.Exit(1)`

- 這樣的問題就在於，就算程式內有使用**defer**延遲執行的函式，他們也不會被執行，而這些延遲執行的函式有可能負責以下的善後功能：
 - 釋出資源
 - 關閉檔案
 - 結束資料庫連線
- 有鑑於此，我們可以在程式中註冊這些訊號，好在收到訊號時可以完成善後的工作，確保程式正常結束

接收中斷訊號通知

- 若希望程式能判斷何時收到特定的作業系統訊號，得使用signal套件的Notify()來註冊他：

`signal.Notify(通知通道, 訊號1, 訊號2...)`

- 當註冊的訊號發生時，他會被傳入通知通道 (通道式go語言專門用於非同步程式交換資料的管道，16章會再介紹)
- 你想接收的系統訊號，都定義在syscall套件的常數中，如syscall.SIGINT(中斷)和syscall.SIGTERM(終止)等

- 為了能註冊和收到系統訊號，你必須先建立一個通道和用make()初始化他，以便傳給signal.Notify()使用：

通道 := make(chan os.Signal, 1)

- chan(channel)關鍵字代表我們要建立通道，其內容為os.Signal
- 後面的1代表通道的緩衝區(buffer)大小，也就是做多可暫存一個訊號

- 建立好通道後，就可以用以下方式取一個值出來：

值 := <-通道

- 箭頭<-是受理算符(**receive operators**), 意義是從通道取出一個值，接著用短變數宣告將該值賦予給一個變數，以便拿來判斷內容

練習：接收中斷訊號並優雅的結束程式

- 以下練習模擬了一個作業程式，並得在使用者中斷程式時讓他“優雅”的結束
- 我們將攔截兩種最常見的訊號：`syscall.SIGINT`和`syscall.SIGTERM`
- 當使用者在主控制台按下`ctrl + c`時，作業系統會傳送`SIGINT`中斷訊號給應用程式
- 由於我們註冊了這個訊號，該訊號會被存入通道變數
- 當我們得知這個訊號存在，就可以自行控制程式要如何結束

```
1  package main
2
3  import (
4      "fmt"
5      "os"
6      "os/signal"
7      "syscall"
8      "time"
9  )
10
11 func main() {
12     //建立訊號通道
13     sigs := make(chan os.Signal, 1)
14     //註冊要通過通道接收的訊號
15     signal.Notify(sigs, syscall.SIGINT, syscall.SIGTERM)
16     defer cleanUp() //延後執行的清理作業
17     fmt.Println("程式執行中 (按下 Ctrl + C 來中斷)")
18 }
```

```
19 MainLoop: //一個標籤，用來代表以下這個無窮for迴圈
20     for {
21         s := <-sigs
22         switch s {
23         case syscall.SIGINT:
24             fmt.Println("程序中斷:", s)
25             break MainLoop
26         case syscall.SIGTERM:
27             fmt.Println("程序終止:", s)
28             break MainLoop
29         }
30     }
31
32     fmt.Println("程式結果")
33 }
34
35 //模擬程式終止後的清理作業
36 func cleanUp() {
37     fmt.Println("進行清理作業...")
38     for i := 0; i <= 10; i++ {
39         fmt.Printf("刪除檔案 %v...(僅模擬)\n", i)
40         time.Sleep(time.Millisecond * 100)
41     }
42 }
43
```

執行結果：

```
程式執行中 (按下 Ctrl + C 來中斷)
```

```
程序中斷: interrupt
```

```
程式結果
```

```
進行清理作業...
```

```
刪除檔案 0...(僅模擬)
```

```
刪除檔案 1...(僅模擬)
```

```
刪除檔案 2...(僅模擬)
```

```
刪除檔案 3...(僅模擬)
```

```
刪除檔案 4...(僅模擬)
```

```
刪除檔案 5...(僅模擬)
```

```
刪除檔案 6...(僅模擬)
```

```
刪除檔案 7...(僅模擬)
```

```
刪除檔案 8...(僅模擬)
```

```
刪除檔案 9...(僅模擬)
```

```
刪除檔案 10...(僅模擬)
```

- 以上練習中需要注意的是，由於尚未介紹go語言的非同步運算，所以使用一個無窮for迴圈來讓程式重複讀取訊號通道
- 此外，為了能從switch內直接脫離for迴圈，for迴圈本身也加上一個識別標籤MainLoop，這使得 break MainLoop 會打斷for迴圈，而不只是單純脫離switch敘述而已
- 正常來說，你應該將判斷訊號的程式碼放在一個非同步程序中，以免卡住其餘部分的程式 (這部分16章會再說明)

12-4 檔案存取權限

- 當你讀取檔案時，檔案權限是必須理解的重要議題
- **Go**語言沿用了**Unix**系統的檔案權限命名法，以符號或八進位數字來表示
- 檔案權限是指定給一個待操作的檔案，來決定你可以對它做什麼動作

- 權限一共有3種 : read(讀取) / write(寫入) / execute(執行)

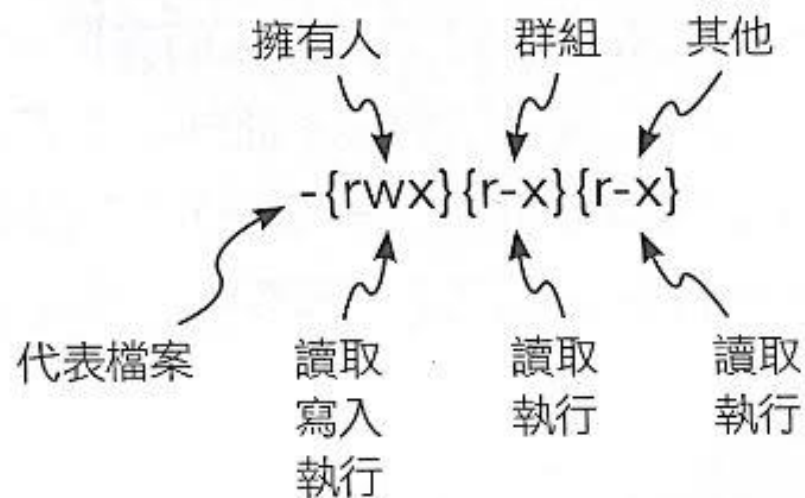
權限	符號	八進位值	說明
read	r	4	允許你開啟並閱讀檔案
write	w	2	允許修改檔案內容
execute	x	1	若是可執行檔或go原始檔, 允許執行
無權限	-	0	沒有給予任何權限

- 此外對於每一個檔案，會針對三組不同的個人或群組指定不同權限：

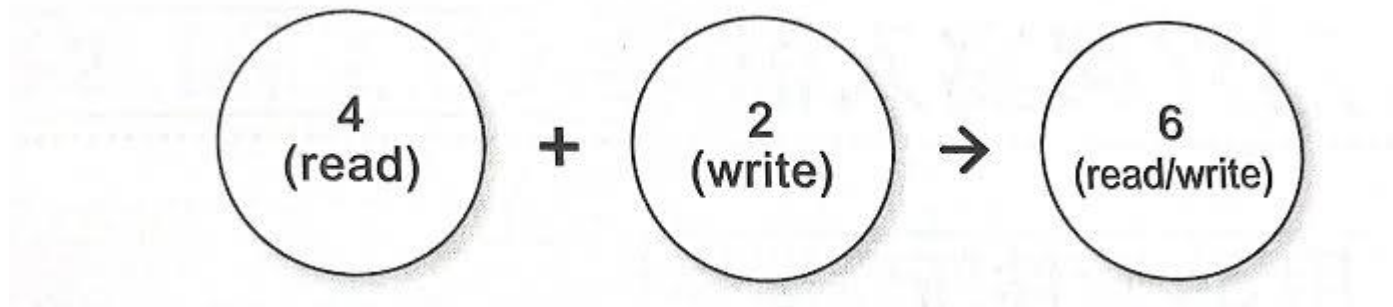
❑ 擁有人 (owner)：擁有檔案的個人，也稱為 root user。

❑ 群組 (group)：通常包括多個個人或其他群組。

❑ 其他 (others)：不屬於以上兩類的使用者或群組。



- 開頭的破折號-代表這是一個檔案，如果是d代表這是一個目錄或資料夾(directory)
- 每組權限表達方式都採“讀取寫入執行”的格式，且可以改寫成單一一個八進位數值
- 舉例來說，rw-代表擁有讀取和寫入權限沒有執行權限，並可以寫成8進位數字6：



右表即為單一組權限的所有可能組合，並以數字和符號兩種寫法來呈現：

權限	符號	八進位
無權限	---	0
可執行	--x	1
可寫入	-w-	2
可執行／寫入	-wx	3
可讀取	r--	4
可讀取／執行	r-x	5
可讀取／寫入	rw-	6
可讀取／寫入／執行	rwX	7

- 同理，三組權限也可以寫成如下表的3組8進位數字，而這也是go語言存取檔案會用到的代碼：

權限	符號	八進位
Owner：讀取	0444	-r--r--r--
Group：讀取		
Others：讀取		
Owner：寫入	0222	--w--w--w-
Group：寫入		
Others：寫入		
Owner：執行	0111	----x--x--x
Group：執行		
Others：執行		
Owner：讀／寫／執行	0763	-rwxrw--wx
Group：讀／寫		
Others：寫／執行		
Owner：讀／寫	0666	-rw-rw-rw-
Group：讀／寫		
Others：讀／寫		
Owner：讀／寫／執行	0777	-rwxrwxrwx
Group：讀／寫／執行		
Others：讀／寫／執行		

12-5 建立與寫入檔案

12-5-1 用os套件新增檔案

- os套件的Create()方法可以建立一個空白的新檔案，並賦予權限0666；若該檔已經存在則內容會被清空：

```
func Create(name string) (*File, error)
```

- 成功新增或清空檔案後，os.Create()會回傳一個*os.File結構
- 我們在第七章知道os.File結構實作了io.Reader介面；事實上也同時實作了io.Writer介面，稍後會解釋為何這點非常重要

- 以下程式會於程式目錄建立一個名叫test.txt的文字檔，並在程式結束時已File結構的Close()關閉它：

```
1  package main
2
3  import "os"
4
5  func main() {
6      f, err := os.Create("test.txt") //建立test.txt
7      if err != nil { //檢查建立檔案時是否遇到錯誤
8          panic(err)
9      }
10     defer f.Close() //確保在main()結束時關閉檔案
11 }
```

12-5-2 對檔案寫入字串

- 建立空檔很簡單，但還需要對它寫入資料，檔案才會有內容
- 這時可以運用os.File的兩個方法：

```
Write(b []byte) (n int, err error)
```

```
WriteString(s string) (n int, err error)
```

- `Write()`和`WriteString()`的功能是一樣的，只是接收的參數不同
- 傳回值`n`代表函式對檔案寫入了`n`個位元，並會在寫入失敗時回傳非`nil`的`error`
- 以下範例會在建立新檔後對該檔結構寫入一些字串：

```
1  package main
2
3  import "os"
4
5  func main() {
6      f, err := os.Create("test.txt")
7      if err != nil {
8          panic(err)
9      }
10     defer f.Close()
11     f.Write([]byte("使用Write()寫入\n"))
12     f.WriteString("使用WriteString()寫入\n")
13 }
14
```

- 執行後，同目錄底下應該會出現test.txt, 其內容如下：

```
ch12 > 12-5-2 > ≡ test.txt
1  使用Write()寫入
2  使用WriteString()寫入
3
```

12-5-3 一次完成建立檔案及寫入

- 我們也可以用一個指令建立新檔並完成寫入資料的動作，這需要使用os套件的WriteFile()函式：

```
func WriteFile(filename string, data []byte, perm os.FileMode) error
```

- filename參數是檔名，若不存在就新增檔案，已存在就清空內容
- data([]byte切片)是要寫入的字串
- perm是檔案權限，如0666或0763，這會設定新檔的權限，但若檔案已存在則不會修改權限

```
1  package main
2
3  import "os"
4
5  func main() {
6      message := "Hello Golang"
7      err := os.WriteFile("test.txt", []byte(message), 0644)
8      if err != nil {
9          panic(err)
10     }
11 }
12
```


執行結果：

```
ch12 > 12-5-3 > ≡ test.txt  
1      Hello Golang
```

12-5-4 檢查檔案是否存在

- 使用`os.Create`或`os.WriteFile`時若碰到已存在的檔案會直接清空，因此你或許會想在建立檔案前先檢查一下檔案是否存在？
- Go語言提供了檢查檔案存在與否的簡單機制，我們直接開程式碼：

```
1  package main
2
3  import (
4      "errors"
5      "fmt"
6      "os"
7  )
8
9  //檢查檔案是否存在的自訂函式
10 func checkFile(filename string) {
11     finfo, err := os.Stat(filename) //取得檔案描述資訊
12     if err != nil {
13         if errors.Is(err, os.ErrNotExist) { //若error中包含檔案不存在的錯誤
14             fmt.Printf("%v: 檔案不存在!\n\n", filename)
15             return //退出函式
16         }
17     }
18
19     //若檔案正確開啟，印出其檔案資訊
20     fmt.Printf("檔名: %s\n是目錄: %t\n修改時間: %v\n權限: %v\n大小: %d\n\n",
21         finfo.Name(), finfo.IsDir(), finfo.ModTime(), finfo.Mode(), finfo.Size())
22 }
23
24 func main() {
25     checkFile("junk.txt")
26     checkFile("test.txt")
27 }
28
```

- `os.Stat()`方法傳回的錯誤可能會包含多重error值，我們需要檢查當中是否包含`os.ErrNotExist`錯誤，是的話就代表此檔不存在
- 這個範例的執行結果如下：

```
PS D:\git\Golang\ch12\12-5-3> go run "d:\git\Golang\ch12\12-5-4\12-5-4.go"  
junk.txt: 檔案不存在!
```

```
檔名: test.txt  
是目錄: false  
修改時間: 2022-08-19 15:29:45.109344 +0800 CST  
權限: -rw-rw-rw-  
大小: 12
```

- `os.Stat`(以及`os.File`結構的`Stat()`方法)會回傳一個`os.fileStat`結構，它實作了`FileInfo`介面，這個介面的方法能查詢檔案的各種資訊：

```
type FileInfo interface {  
    Name() string          // 檔名  
    Size() int64           // 檔案大小（計算方式取決於系統）  
    Mode() FileMode        // 修改權限  
    ModTime() time.Time    // 修改時間  
    IsDir() bool           // 是否為目錄，相當於呼叫 Mode().IsDir()  
    Sys() interface{}      // 檔案資料來源（有可能傳回 nil）  
}
```

12-5-5 一次讀取整個檔案的內容

- 建立檔案後，自然會需要讀取它
- 若檔案不大，那可以用本小節的兩種方法一次讀取所有內容
- 但若用這些方法開啟過大的檔案，會耗費待量系統記憶體

使用os.ReadFile()

- 第一種讀取全部檔案內容的方法如下：

```
func ReadFile(filename string) ([]byte, error)
```

- `os.ReadFile`會開啟檔名參數指定的檔案並讀取其內容，成功的話以`[]byte`切片形式傳回，`err`也會回傳`nil`
- 前面的`os.File`結構在讀取內容時若碰到檔案結尾會傳回`io.EOF`(end of file)錯誤，但`ReadFile`不會回傳`EOF`

```
1  package main
2
3  import (
4      "fmt"
5      "os"
6  )
7
8  func main() {
9      //讀取整個檔案的內容
10     content, err := os.ReadFile("test.txt")
11     if err != nil {
12         fmt.Println(err)
13     }
14     fmt.Println("檔案內容:")
15     fmt.Println(string(content))
16 }
17
```


執行結果：

```
PS D:\git\Golang\ch12\12-5-5> go run .  
檔案內容：  
Golang programming language  
Go 技術者們
```

使用io.ReadAll()搭配os.Open

- 除了os.ReadFile(), 還可以用另一個函式io.ReadAll()讀取整個檔案的內容
- io.ReadAll()同樣會回傳一個[]byte切片, 不同之處在於ReadAll()接收的參數是io.Reader介面型別:

```
func ReadAll(r io.Reader) ([]byte, error)
```

- 這表示ReadAll()不僅能用來讀取os.File檔案, 也可以讀取符合io.Reader介面的任何物件, 比如strings.NewReader()或http.Request等
- 若要讀取檔案, 得先取得該檔案的os.File結構, 辦法是使用os.Open()函式:

```
func Open(name string) (*File, error)
```

```
1  package main
2
3  import (
4      "fmt"
5      "io"
6      "os"
7  )
8
9  func main() {
10     f, err := os.Open("test.txt") //開啟檔案
11     if err != nil {
12         panic(err)
13     }
14     defer f.Close()
15     content, err := io.ReadAll(f) //讀取檔案的整個內容
16     if err != nil {
17         fmt.Println(err)
18         os.Exit(1)
19     }
20     fmt.Println("檔案內容:")
21     fmt.Println(string(content))
22 }
23
```

執行結果：

```
檔案內容：  
Golang programming language  
Go 語言
```

12-5-6 一次讀取檔案中的一行字串

- 若文字檔很大的話，像前面那樣一次讀取會消耗不少記憶體
- 這時你便可以考慮一次讀取檔案的一行，為此可以使用**bufio**套件，也就是帶有緩衝區(**buffer**)的**io**套件
- 為了使用**bufio**，第一步是先將檔案結構換成**bufio.Reader**結構：

```
func NewReader(rd io.Reader) *Reader
```

```
func NewReaderSize(rd io.Reader, size int) *Reader
```

- 可以看到以上兩個函式都接收一個**io.Reader**介面型別，兩者的差異在於**NewReaderSize**有個**size**參數，這是你想使用的緩衝區大小
- 若設為小於**0**的值，那麼就會使用預設值**4096**
- 不管檔案有多大，同時讀進記憶體空間的就只有緩衝區能容納的字元樹而已，這樣就達到了節省空間的目的 (能指定的最大緩衝區是**64 * 1024**)
- 建立了**bufio.Reader**結構後，你就可以使用它新增的方法來讀取檔案內容，最常用的叫做**ReadString()**：

```
func (b *Reader) ReadString(delim byte) (string, error)
```

- 參數`delim`表示格符號(`delimiter`), 通常設為`\n`, `ReadString()`讀到該字元就會停下來, 將包含該字元的字串傳回
- 要是讀到該字元之前就碰到檔案結尾, 就會回傳檔案結尾前的內容以及`io.EOF`錯誤
- 來看以下範例:

```
1  package main
2
3  import (
4      "bufio"
5      "fmt"
6      "io"
7      "os"
8  )
9
10 func main() {
11     file, err := os.Open("test.txt") //開啟檔案取得os.File結構
12     if err != nil {
13         panic(err)
14     }
15     defer file.Close()
16
17     fmt.Println("檔案內容:")
18     //建立一個bufio.Reader結構，緩衝區大小10
19     reader := bufio.NewReaderSize(file, 10)
20     for {
21         //讀取reader直到碰到換行符號為止
22         line, err := reader.ReadString('\n')
23         fmt.Print(line)
24         if err == io.EOF { //若讀取到檔案結尾就結束
25             break
26         }
27     }
28 }
29
```


執行結果：

檔案內容：

Golang programming language

Go 技術者們

12-5-7 刪除檔案

- 最後，若想刪除檔案，可以使用`os.Remove()`函式：

```
func Remove(name string) error
```

- 刪除成功時會傳回值為`nil`的`error`
- 這裡就不先為這個函式舉例了，各位可以自行嘗試看看

12-6 最完整的檔案開啟與建立功能：os.OpenFile()

- 剛剛介紹的各種寫入/建立/開啟/讀取檔案的方法已經可以應付大部分的狀況；但若你希望在開啟檔案時能指定更特定的行為，例如限制它採用唯讀/唯寫模式，或是要附加還是先清空其內容等，就得使用os.OpenFile：

```
func OpenFile(name string, flag int, perm FileMode) (*File, error)
```

- 成功開啟檔案時，OpenFile()會回傳代表檔案結構的os.File結構
- 參數name是檔名，perm參數是指定給檔的權限(八進位數)，在檔案不存在時指定該權限
- 特別的是flag參數，他能決定檔案開啟後可進行的操作，os套件中定義了一系列相關的常數：

```
// https://golang.org/pkg/os/#pkg-constants
const (
    // 你必須指定 O_RDONLY, O_WRONLY 或 O_RDWR 其中之一
    O_RDONLY int = syscall.O_RDONLY // 將檔案開啟為唯讀模式
    O_WRONLY int = syscall.O_WRONLY // 將檔案開啟為唯寫模式
    O_RDWR  int = syscall.O_RDWR   // 將檔案開啟為可讀寫模式
    // 使用 | 算符來連接以下旗標
    O_APPEND int = syscall.O_APPEND // 將寫入資料附加到檔案尾端
    O_CREATE int = syscall.O_CREAT  // 檔案不存在時建立新檔案
    O_EXCL   int = syscall.O_EXCL   // 配合 O_CREATE 使用，確保檔案不存在
    O_SYNC   int = syscall.O_SYNC   // I/O 同步模式（等待儲存裝置寫入完成）
    O_TRUNC  int = syscall.O_TRUNC  // 在開啟檔案時清空內容
)
```

- 這些旗標可以串聯使用，改變檔案在不同情況下的操作行為
- 下面來看一個例子：

```
1  package main
2
3  import (
4      "os"
5      "time"
6  )
7
8  func main() {
9      //建立開啟檔案
10     f, err := os.OpenFile("junk.txt", os.O_CREATE|os.O_APPEND|os.O_WRONLY, 0644)
11     if err != nil {
12         panic(err)
13     }
14     defer f.Close()
15
16     f.Write([]byte(time.Now().String() + "\n"))
17 }
18
```

- 在以上範例, `OpenFile()` 函式使用了三個旗標: `os.O_CREATE`(新增), `os.O_APPEND`(附加), `os.O_WRONLY`(唯寫), 並用 `|` 算符串聯
- 下面展示了這些旗標的串聯效果:
 - `os.O_CREATE`: 使用預設的可讀寫模式, 若檔案不存在則建立新檔, 若已經存在沒有動作
 - `os.O_CREATE | os.O_WRONLY`: 同上, 但指定唯寫模式
 - `os.O_APPEND | os.O_WRONLY`: 指定唯寫模式, 並將寫入內容附加到檔案結尾, 若檔案不存在則回傳 `error`
 - `os.O_CREATE | os.O_APPEND | os.O_WRONLY`: 指定唯寫模式, 並將結尾附加到檔案結尾, 若檔案不存在則建立新檔
- 註: 若只是要開啟檔案使用 `os.Open()` 即可

- 現在來執行程式幾次：

```
PS D:\git\Golang\ch12\12-6> go run .  
PS D:\git\Golang\ch12\12-6> go run .
```

- 執行後該專案目錄下會新增一個junk.txt, 且會寫入目前時間的字串, 且只要多執行程式幾次, junk.txt的內容就會逐漸增加, 因為我們使用了O_APPEND旗標

```
ch12 > 12-6 > ≡ junk.txt  
1    2022-08-20 17:37:06.0973149 +0800 CST m=+0.002669701  
2    2022-08-20 17:37:09.3062328 +0800 CST m=+0.001578201
```

練習：檔案備份

- 在處理檔案時，時常需要備份檔案，甚至得紀錄過去修改的歷史版本
- 在這次練習中，我們要把一個既存的文字檔**note.txt**的內容拷貝到備份檔**backupFile.txt**，而且寫入的內容不能覆蓋既有資料，必須附加在檔案結尾
- 以下是本練習使用的**note.txt**內容：


```
ch12 > 12-6(2) > ≡ note.txt
```

```
1      1. Get better at coding.
```

```
2      note 1
```

```
3      note 2
```

```
4      note 3
```

```
5      note 4
```

```
6      note 5
```

```
7      note 6
```

```
8      note 7
```

```
9      note 8
```

```
10     note 9
```

```
11     note 10
```

```
12
```

- 此外使用`os.Open()`開啟`note.txt`時，假如檔案不存在，需要傳回一個自訂的`error`值
- 以下為程式碼：

```
1  package main
2
3  import (
4      "errors"
5      "fmt"
6      "io"
7      "os"
8      "time"
9  )
10
11  //自訂error
12  var ErrWorkingFileNotFound = errors.New("查無工作檔案")
13
14  func main() {
15      workFileName := "note.txt"
16      backupFileName := "backup.txt"
17      err := writeBackup(workFileName, backupFileName)
18      if err != nil {
19          panic(err)
20      }
21  }
22
```

```
23 //備分檔案的函式
24 func writeBackup(work, backup string) error {
25     workFile, err := os.Open(work) //開啟工作檔
26     if err != nil {
27         if errors.Is(err, os.ErrNotExist) {
28             return ErrWorkingFileNotFound //查無工作檔則傳回自訂error
29         }
30         return err
31     }
32     defer workFile.Close() //備份結束後關閉工作檔
33
34     //開啟備份檔，沒有就新增一個，資料附加到結尾
35     backFile, err := os.OpenFile(backup, os.O_CREATE|os.O_APPEND|os.O_WRONLY, 0644)
36     if err != nil {
37         return err
38     }
39     defer backFile.Close() //備分結束後關閉工作檔
40
41     content, err := io.ReadAll(workFile) //讀取工作檔內容
42     if err != nil {
43         return err
44     }
45 }
```

```
46      //把一行日期和工作檔內容寫入備分檔
47      _, err = backFile.WriteString(fmt.Sprintf("[%v]\n%v", time.Now().String(), string(content)))
48      if err != nil {
49          return err
50      }
51
52      return nil
53  }
54
```

- 執行程式後，檢視backup.txt的結果：

```
1    [2022-08-20 18:01:38.3126701 +0800 CST m=+0.002031501]
2    1. Get better at coding.
3    note 1
4    note 2
5    note 3
6    note 4
7    note 5
8    note 6
9    note 7
10   note 8
11   note 9
12   note 10
13
```

- 你可以試著修改**note.txt**的內容，然後重複執行以上程式，會發現程式將新版的工作檔內容複製到**backup.txt**的尾端，還記錄了備份時間

用log套件將日誌訊息寫入文字檔

- 第9章曾介紹過log套件，當時你提過你可以建立自己的logger日誌物件，甚至能將訊息輸出到檔案而不是主控台
- 下面就是一個簡單的範例：


```
1  package main
2
3  import (
4      "fmt"
5      "log"
6      "os"
7      "time"
8  )
9
10 func main() {
11     //建立或開啟一個日誌檔
12     //其名稱是log-年-月-日.txt，以當下時間為標準
13     logfile, err := os.OpenFile(
14         fmt.Sprintf("log-%v.txt", time.Now().Format("2006-01-02")),
15         os.O_CREATE|os.O_APPEND|os.O_WRONLY, 0644)
16     if err != nil {
17         panic(err)
18     }
19     defer logfile.Close()
20     //建立logger，寫入對象為前面開啟的檔案
21     logger := log.New(logfile, "log", log.Ldate|log.Lmicroseconds|log.Llongfile)
22     //將日誌輸出到檔案
23     logger.Println("log message")
24 }
25
```

- 這裡也運用了第10章的time套件，將執行當天的日期變成log檔的檔名，這樣當有大量日誌檔時就能依據日期進行排序
- 執行程式後，logger印出的任何東西便會寫入到你用os.OpenFile()開啟的檔案中，例如：

```
ch12 > 12-6(3) > ≡ log-2022-08-21.txt
1    log2022/08/21 16:29:47.161657 d:/git/Golang/ch12/12-6(3)/12-6(3).go:23: log message
2
```

12-7 處理CSV格式檔案

- 除了純文字檔和上一章介紹的JSON資料外，程式最常存取的檔案格式之一就是CSV(comma-separated value, 逗號分隔值)
- CSV本身是純文字，但使用逗號來分隔每一直行(column)或欄位的值，而每一橫列(row)則以每一行結尾的換行符號區隔
- 以下是一個典型的CSV格式資料：

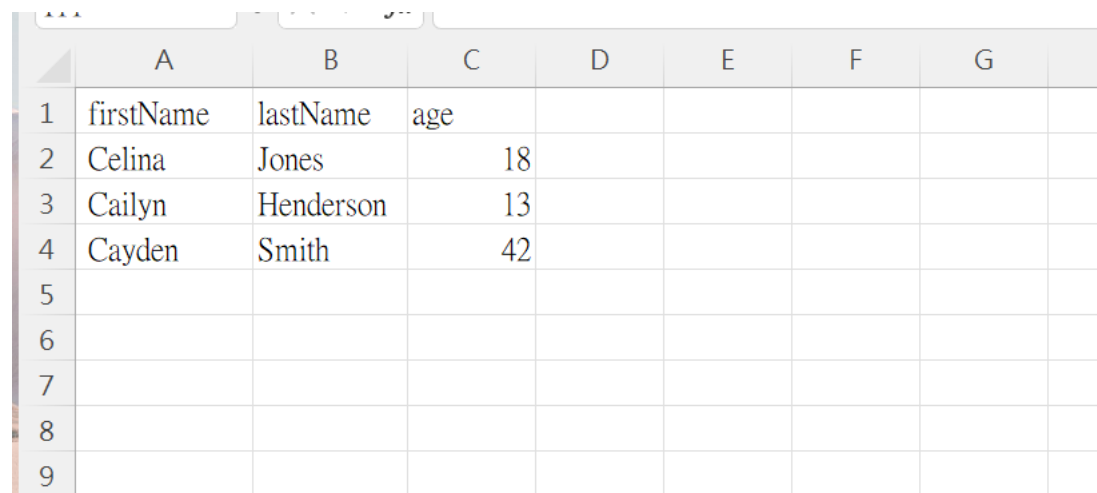
firstName,lastName,age

Celina,Jones,18

Cailyn,Henderson,13

Cayden,Smith,42

- 第一行是標頭(header), 及各行或欄位的名稱
- 若你將文字檔儲存成以.csv檔, 並使用試算表軟體開啟, 會看到如以下的呈現:



	A	B	C	D	E	F	G
1	firstName	lastName	age				
2	Celina	Jones	18				
3	Cailyn	Henderson	13				
4	Cayden	Smith	42				
5							
6							
7							
8							
9							

12-7-1 走訪CSV檔內容


- Go語言提供了標準函式庫`encoding/csv`, 可以用來解析CSV格式資料：

```
func NewReader(r io.Reader) *Reader
```

- 就像`bufio`套件一樣，`csv`套件的`NewReader()`接收一個`io.Reader`介面型別，然後傳回`csv.Reader`結構
- 此結構的`Read()`方法能用來讀取`csv`資料的一行內容，並將其轉成`[]string`切片：

```
func (r *Reader) Read() (record []string, err error)
```

- 不同於**bufio.Reader**的**ReadString()**方法，**csv**套件會自動判斷結尾的換行符號
- 但這行文字會以字串切片的形式回傳，下一小節我們再來看這點該如何利用
- 下面的範例要從一個名為**data.csv**的檔案讀取**CSV**資料，其內容如下：

```
ch12 > 12-7-1 >  data.csv  
1  firstName,lastName,age  
2  Celina,Jones,18  
3  Cailyn,Henderson,13  
4  Cayden,Smith,42
```

- 只要開啟這個檔案，就能將os.File結構傳給csv.NewReader()來建立我們需要的物件：

```
1  package main
2
3  import (
4      "encoding/csv"
5      "fmt"
6      "io"
7      "os"
8  )
9
10 func main() {
11     file, err := os.Open("data.csv") //開啟CSV檔案
12     if err != nil {
13         panic(err)
14     }
15
16     reader := csv.NewReader(file) //取得csv.Reader結構
17     for {
18         record, err := reader.Read() //從csv.Reader讀取一行資料
19         if err == io.EOF {           //遇到檔案結尾錯誤，就離開迴圈
20             break
21         }
22         if err != nil {
23             fmt.Println(err)
24             continue
25         }
26         fmt.Println(record) //印出該行結果
27     }
28 }
29
```

執行結果：

```
PS D:\git\Golang\ch12\12-7-1> go run .  
[firstName lastName age]  
[Celina Jones 18]  
[Cailyn Henderson 13]  
[Cayden Smith 42]
```


12-7-2 讀取每郎資料各欄位的值

- 這時你可能會想：既然可以讀出一行資料，那可以只抽出個別欄位的值嗎？可以跳過標頭嗎？這其實很簡單
- 如前所述，`csv.Reader`的`Read()`方法會回傳[]string切片，`csv`套件會以半形逗號為依據，將各欄位轉成字串切片的不同元素：索引0是從左邊數來第一個欄位，以此類推
- 所以只要事先知道CSV檔的組成，就很容易取出想要的東西了

- 至於跳過標頭，我們可以加入一個布林變數作為開關，在**for**迴圈第一次執行時不印出任何東西
- 以下我們來修改上一個範例的程式碼：

```
1  package main
2
3  import (
4      "encoding/csv"
5      "fmt"
6      "io"
7      "os"
8  )
9
10 const (
11     firstName = iota //CSV欄位索引
12     lastName
13     age
14 )
15
```

```
16 func main() {
17     file, err := os.Open("data.csv")
18     if err != nil {
19         panic(err)
20     }
21     defer file.Close()
22
23     header := true // 標頭開關
24     reader := csv.NewReader(file)
25     for {
26         record, err := reader.Read()
27         if err == io.EOF {
28             break
29         }
30         if err != nil {
31             fmt.Println(err)
32             continue
33         }
34         if header {
35             header = false
36             continue // 跳過第一行(標頭)
37         }
38         fmt.Println("-----")
39         fmt.Println("First name:", record[firstName])
40         fmt.Println("Last name :", record[lastName])
41         fmt.Println("Age      :", record[age])
42     }
43 }
44
```

執行結果：

```
-----  
First name: Celina  
Last name : Jones  
Age       : 18  
-----
```

```
First name: Cailyn  
Last name : Henderson  
Age       : 13  
-----
```

```
First name: Cayden  
Last name : Smith  
Age       : 42
```

補充

- `csv.Reader`結構的`Comma`屬性可以用來指定CSV資料的分隔符號，例如 `reader.Comma = "\t"`
- 該屬性為`rune`型別，因此可以是任何合法字元，但不可使用換行字元`\n`或回車字元`\r`

本章結束