

特殊語法與目錄結構



特殊語法與目錄結構

三個點的省略符號：刪節號...

- 除了人們會用...來代表很無言之外，在程式中也常會看到Golang很無言的...
- 常見的俗名或稱呼方式為：
 - 點點點 (Dot Dot Dot)
 - 三個點 (Three Dot 、Triple Dot)
 - 刪節號 (Ellipsis)

特殊語法與目錄結構

三個點的省略符號：刪節號...

- 在Golang中，刪節號 ... 共有四種用法：
 - 用法1 - 省略陣列長度
 - 在之前宣告Array的時候，也用過省略符號來省略長度。省略長度是要編譯器計算Array陣列長度、自動將長度填入，而不是沒有長度的Slice：

```
func main() {  
    var a = [...]int{1, 2, 3, 7, 8, 9}  
    fmt.Println(len(a))  
    fmt.Println(a)  
}  
  
/*  
6  
[1 2 3 7 8 9]  
*/
```

特殊語法與目錄結構

三個點的省略符號：刪節號...

- 在Golang中，刪節號 ... 共有四種用法：
 - 用法2 - 可變參數函式
 - 那麼這個刪節點、省略符號用在函式中，就稱作 可變函式(參數長度可變) Variadic Functions，意思即為，可以傳入多個參數(0~N個)，不傳入也行。在不知道參數到底有幾個的時候，可以透過這個方法來解決，但需為**同樣型別**。其實就有點像[] Slice 的意思。

先滿足必要的參數，再來填入剩下的剩餘參數

換角度思考一下，如果一開始在第一個元素上放省略符號

Golang編譯器要怎麼知道此人到底想省略幾個元素？

```
func test1(...int) {  
  
}  
  
// 省略符號只能放在最後的位置上  
func test2(int, ...string) {  
  
}  
  
// 這個會報錯 Can only use '...' as final argument in list  
func test3(... int, ... string) {  
  
}
```

特殊語法與目錄結構

三個點的省略符號：刪節號...

- 在Golang中，刪節號 ... 共有四種用法：
- 用法2 - 可變參數函式

```
func main() {  
    s1 := sum()  
    fmt.Println(s1)  
  
    s2 := sum(1, 5, 9)  
    fmt.Println(s2)  
}  
  
func sum(nums ...int) int {  
    var total int  
    for _, num := range nums {  
        total += num  
    }  
    return total  
}  
  
/*  
0  
15  
*/
```

在這個地方，省略長度的int nums，像極了Slice

```
func sum(nums ...int) int {  
    var total int  
    for _, num := range nums {  
        total += num  
    }  
    return total  
}
```

根本就是Slice

特殊語法與目錄結構

三個點的省略符號：刪節號...

- 在Golang中，刪節號 ... 共有四種用法：
 - 用法3 - 開箱、解包、解壓縮(Unpacking)
 - 在不同的地方上面可以代表不同的意涵，省略符號 ... 也可以使用在很暴力殘忍的地方上，例如：將Slice剝皮。
水能載舟亦能覆舟，才剛把他弄成Slice，現在又要將他剝皮。

```
func sumUnpacking(nums ...int) int { // 傳入int但不曉得參數長度為何
    var total int
    for _, num := range nums {
        total += num
    }
    return total
}

func main() {
    slice := []int{2, 3, 5}

    sum1 := sumUnpacking(slice...) // 把slice 解開、剝皮後傳入，同下
    fmt.Println(sum1)
}

/* result:
10
*/
```

那到底什麼時候是Slice、什麼時候是省略長度、什麼時候會被剝皮？

<https://go.dev/play/p/4-RXxIOe880>

特殊語法與目錄結構

三個點的省略符號：刪節號...

- 在Golang中，刪節號 ... 共有四種用法：
 - 用法3 - 開箱、解包、解壓縮(Unpacking)
 - 以下整理了三種不同的方式來比較、探討Slice與功能皆為實現一群數字的加總

```
func main() {
    slice := []int{2, 3, 5}

    sum1 := sumUnpacking(slice...) // 把slice 解開、剝皮後傳入，同下
    fmt.Println(sum1)

    sum2 := sumUnpacking(2, 3, 5) // 可變參數函式
    fmt.Println(sum2)

    sum3 := sumSlice(slice) // 不曉得int長度，也可以直接包成一個slice型別來傳遞
    fmt.Println(sum3)
}

func sumUnpacking(nums ...int) int { // 傳入int但不曉得參數長度為何
    var total int
    for _, num := range nums {
        total += num
    }
    return total
}

func sumSlice(nums []int) int { // 傳入slice
    var total int
    for _, num := range nums {
        total += num
    }
    return total
}

/* result:
10
10
10
*/
```

特殊語法與目錄結構

三個點的省略符號：刪節號...

- 在Golang中，刪節號 ... 共有四種用法：
 - 用法3 - 開箱、解包、解壓縮(Unpacking)
 - append 一群數字的合併：要怎麼使用 append 來合併兩個陣列？這邊整理了三種合併一群數字的方法。

```
func main() {  
    slice1 := []int{1, 2, 3, 4, 5}  
    slice2 := []int{6, 7, 8}  
  
    // 將兩 Slice 合併(append) 的方法 - 1  
    a1 := append(slice1, slice2[0], slice2[1], slice2[2]) // append用法，每個參數只能附上一個int  
    fmt.Println(a1)  
  
    // 將兩 Slice 合併(append) 的方法 - 2  
    a2 := slice1  
    for _, num := range slice2 { // 如方法1，只是這次透過for迴圈來迭代完成  
        a2 = append(a2, num)  
    }  
    fmt.Println(a2)  
  
    // 將兩 Slice 合併(append) 的方法 - 3  
    a3 := append(slice1, slice2...) // 直接將slice2 剝皮解壓縮後(Unpacking)再執行append，取代上面使用For迭代的方法  
    fmt.Println(a3)  
}  
  
/* result:  
[1 2 3 4 5 6 7 8]  
[1 2 3 4 5 6 7 8]  
[1 2 3 4 5 6 7 8]  
*/
```


特殊語法與目錄結構

三個點的省略符號：刪節號...

- 在Golang中，刪節號 ... 共有四種用法：

- **用法4 - go test 跑測試**

- 這不是程式內的用法，而是在編譯、跑測試時才會用到執行的指令

```
$ go test ./...
```

- 將自動運行所有這目錄底下的Package的測試程式，當然要將所有的測試檔案 `_test.go` file 先寫好，包括預設輸入及預期輸出。

特殊語法與目錄結構

目錄架構

- GOROOT 放的是安裝的go語言、**官方內建的函式庫**，例如寫程式時經常會用到的"fmt"就放在這底下：

```
import "fmt"
```

- GOPATH 放的是**別人開發的第三方套件以及自己的程式碼**：

```
import "github.com...(別人的repository)"
```

- 照慣例通常會把自己寫的專案及程式放入GOPATH底下的**src(source)**資料夾裡，如果有用Github會再多一層 github.com 及 account。
 - 以上的分法都通常是在較久遠的年代，透過**glide或dep**來管理。自從Go 1.11 推出了go mod後，就不一定要把專案放到GOPATH底下，也可以正常運行，但這邊還是偏向使用GOPATH及glide套件管理工具

特殊語法與目錄結構

目錄架構

- GOPATH目錄架構

- 使用者目錄 (C:\Users\USER 或 ~/)

```
- Go (GOPATH) 如果底下沒有bin/pkg/src目錄，別擔心，自己新增一個
- bin (放編譯過後的可執行檔案 二進制檔案)
- pkg (放編譯過後的library)
- src (放原始碼 Source Code)
  - github.com
    - Jack (使用者名稱)
      - HelloWorldProject (專案名稱)
        - main.go (主程式)
    - Tom
      - MyCoolProject
        - core (函式庫名稱)
          - core.go
          - xxxx.go
        - libs (函式庫名稱)
          - lib.go
          - oooo.go
        - main.go
  - gitlab.com
  - ...
```

套件與常用函式庫



套件與常用函式庫

套件(Package)

- 套件是 Go 語言程式碼的集合，是一種程式碼模組化與重用的方式，在開發時，很難不用到套件，像是大量使用 Go 語言內建函式庫中的 `fmt` 或是 `time` 套件。
- 在 `opensource` 裡面，可以提供自己寫好的 `package` 給別人用，也能引入別人所寫的 `package`。
- **import package**

```
package main

import (
    "fmt"
)

func main() {
    fmt.Println("Hello, playground")
}
```

這是一個 `golang` 標準的 `hello world` 寫法，大家應該有看到上面 `import` 這個 `keyword`，這句話意思就是載入 `package`，在這邊這個範例是載入 `fmt` 這個 `package`。

套件與常用函式庫

套件(Package)

- 在做下面這個範例前，請先在自己的 local 的開發環境下指令：

```
go get github.com/syhlion/simplenum
```

- 這是一個簡易的數字四捨五入套件。這句指令的意思就是把它 get 回來，這樣後續才能 import 跟 build。
- **import 外部 package**

```
package main

import (
    "github.com/syhlion/simplenum"
)

func main() {
    fmt.Println(simplenum.Round(1.1354, 2))
}
```

這個範例因為引入外部 package 所以 go playground 不能執行。

go 支援 git 的 namespace 協定，所以它 import 路徑也依照 git namespace 下去命名。這樣就完成套件的引入

套件與常用函式庫

套件(Package)

- **design package**

- 要開發一個套件，必須先命名：`package <packagename>`
- 上面的 `package` 宣告必須是 Go 檔案中的第一程式碼，而在同一個目錄下，同層檔案的都屬於同一個 `package`。
- 右上就完成一個非常基礎的 `math package`，有提供 `Add` 方法。只要推上 `github`、`gitlab`...，就能讓大家用 `go get` 引入 `package`
- `golang` 有一個很有趣的設計，它並沒有「顯示」的宣告 `public`、`private`，它的變數、`func`，可存取範圍是以 `package` 為單位，也就是同一個 `package` 下，並沒有區分 `public`、`private`，那不同 `package` 間要如何區分 `public`、`private`呢？用首字的大小寫來做辨識，只要首字大寫，`golang complie` 就會把它視為 `public`，外部 `package` 也都能存取，小寫就是反之(`private`)。
- 這樣來看，上面那個例子 `privateAdd`，在外部套件中是存取不到的，如果硬要存取，則會有 `undefined` 的錯誤，在這邊上，就可以藉由 `public`、`private`來做設計，保護自己的 `package` 一些比較危險的操作可以封裝起來。

```
package math

func privateAdd(a int, b int) (c int) {
    c = a + b
    return
}

func Add(a int, b int) (c int) {
    c = a + b
    retrun
}
```

套件與常用函式庫

常用函式庫

- strings

- 字串的處理，應該是基礎中的基礎，任何語言寫任何服務都可以用到。

```
package main

import (
    "fmt"
    "strings"
)

func main() {
    //字串轉換大小寫
    //https://golang.org/pkg/strings/#ToLower
    //https://golang.org/pkg/strings/#ToUpper

    upperString := "HELLO WORLD"
    lowerString := "hello world"
    fmt.Println("ToUpper:", strings.ToUpper(lowerString))
    fmt.Println("ToLower:", strings.ToLower(upperString))

    //字串切割，可以指定特定分隔符號轉換成 array
    //https://golang.org/pkg/strings/#Split
    splitString := "a,b,c,d,e,f"
    fmt.Println("Split:", strings.Split(splitString, ","))

    //字串陣列 依照特定分隔符，合併成一個字串
    //https://golang.org/pkg/strings/#Join
    joinArray := []string{"a", "b", "c", "d", "e", "f"}
    fmt.Println("Join:", strings.Join(joinArray, ","))
}
```


套件與常用函式庫

常用函式庫

- strconv
 - 字串數字的轉換，也是必備的工具之一。

```
package main

import (
    "fmt"
    "strconv"
)

func main() {
    var int1 int
    var string1 string
    int1 = 5
    string1 = "5"

    //字串轉數字
    ii, err := strconv.Atoi(string1)
    if err != nil {
        fmt.Println(err)
        return
    }
    fmt.Println("to int :& add one", ii+1)

    //數字轉字串
    fmt.Println("to string :", strconv.Itoa(int1))
}
```

套件與常用函式庫

常用函式庫

- `io/ioutil-ReadFile`
 - 讀檔功能，在這邊說明一下，`ioutil` 是 `golang` 包裝過後的簡易工具，所以基本上不需要處理『資源釋放』`Close` 的相關問題，它內部已經處理完了，如果是運用 `golang` 其他 `stdlib` 來讀檔，就要注意 `Close` 的相關問題。

```
package main

import (
    "fmt"
    "io/ioutil"
    "log"
)

func main() {
    content, err := ioutil.ReadFile("testdata/hello")
    if err != nil {
        log.Fatal(err)
    }

    fmt.Printf("File body: %s", content)
}
```

套件與常用函式庫

常用函式庫

- `io/ioutil-ReadDir`

- 這個函式很顯然，是讀取該檔案夾裡頭的檔案：

```
package main

import (
    "fmt"
    "io/ioutil"
)

func main() {
    datas, err := ioutil.ReadDir(".")
    if err != nil {
        fmt.Println(err)
    }

    for _, file := range datas {
        fmt.Println(file.Name())
    }
}
```

- 可以執行看看，會印出該檔案夾，所有檔案及資料夾。

套件與常用函式庫

常用函式庫

- `os`

- 如果想要開啟檔案後，做一些寫入的操作，可以使用 `os` 套件，它允許先開啟一個檔案，然後再執行一些對檔案的操作，如下列程式碼：

```
package main

import (
    "fmt"
    "os"
)

func main() {
    // 讀取檔案，若檔案不存在，則創建它
    f, err := os.OpenFile("log.csv", os.O_APPEND|os.O_CREATE|os.O_WRONLY, 0644)
    if err != nil {
        fmt.Println(err)
    }
    if _, err := f.Write([]byte("appended some log\n")); err != nil {
        f.Close()
        fmt.Println(err)
    }
    if err := f.Close(); err != nil {
        fmt.Println(err)
    }
}
```

- 上述程式碼，開啟一個 `csv` 檔，並執行一個寫入的動作。

套件與常用函式庫

常用函式庫

- `os`
 - `os.Mkdir` 是建立資料夾(名稱, 權限)
 - 如果要直接建立更多層的方式則是使用`os.MkdirAll`
 - `os.Remove`是刪除，但如果該資料夾內有東西，則不能使用`os.Remove`而是需要改用`os.RemoveAll`
 - 權限`777`，依序是指 `owner/group/others`這三者的權限，而數字代表的是要開放那些權限(`read`、`write`、`exe`)，`read = 4`、`write = 2`、`exe = 1`，所以`7` 等於全開，如果是`4`就代表只開`read`的權限。

```
package main

import (
    "fmt"
    "os"
)

func main() {
    os.Mkdir("demo", 0777)
    //os.MkdirAll("demo/test1/test2", 0777)
    err := os.Remove("demo ")
    if err != nil {
        fmt.Println(err)
    }
    //os.RemoveAll("demo")
}
```

套件與常用函式庫

常用函式庫

- OS

- 寫檔案：

```
package main

import (
    "fmt"
    "os"
)

func main() {

    userFile := "demo.txt"

    fout, err := os.Create(userFile)

    if err != nil {
        fmt.Println(userFile, err)
        return
    }
    defer fout.Close()

    for i := 0; i < 10; i++ {

        fout.WriteString("gogo test!\r\n")

    }

}
```

- 讀取檔案：

```
fl, err := os.Open(userFile)

if err != nil {

    fmt.Println(userFile, err)

    return

}

defer fl.Close()

buf := make([]byte, 1024)

for {

    n, _ := fl.Read(buf)

    if 0 == n {

        break

    }

    os.Stdout.Write(buf[:n])

}
```

<https://go.dev/play/p/2DzQ5VDzK01>

<https://go.dev/play/p/Uc07Ynu0sVg>

套件與常用函式庫

常用函式庫

- time

- golang的時間單位如下：

```
const (  
    Nanosecond Duration = 1  
    Microsecond      = 1000 * Nanosecond  
    Millisecond       = 1000 * Microsecond  
    Second            = 1000 * Millisecond  
    Minute            = 60 * Second  
    Hour              = 60 * Minute  
)
```

- 有了上面的單位換算，接下來來看下面的使用範例：

```
package main  
  
import (  
    "fmt"  
    "time"  
)  
  
func main() {  
    //unix time  
    fmt.Println("unix: ", time.Now().Unix())  
  
    //取得 nano second  
    fmt.Println("unix: ", time.Now().UnixNano())  
  
    //format成 正常格式化後的時間  
    fmt.Println("datetime: ", time.Now().Format("2006/01/02 15:04:05"))  
  
    //after  
    time.AfterFunc(3*time.Second, func() {  
        fmt.Println("hello world")  
    })  
  
    //這邊 Sleep 5s 是為了讓上面的 AfterFunc 會執行，不然就像有講到的 只要 main thread 結束，任何 sub thread 都會跟著一起結束  
    time.Sleep(5 * time.Second)  
    fmt.Println("end")  
}
```

套件與常用函式庫

常用函式庫

- time

- 在這邊要特別說明一下，這個 **Format** 語法，為什麼沒有像其他語言一樣用 %d、%M....，之類的符號呢？
- 為什麼會這樣呢？難道是 2006-01-02 15:04:05 發生了什麼事情？

裡面就有講到他的設定，就是照這幾個數字去 **parse**，記憶口訣是基於
01/02 03:04:05PM '06 -0700

有沒有注意到？剛好是 1234567 的順序，很難講這個有別於其他語言的設定，到底是好還是不好，不過要花點時間去適應就是了。

```
const (
    stdLongMonth      = iota
    stdMonth          = iota + stdNeedDate // "January"
    stdNumMonth       // "Jan"
    stdZeroMonth      // "1"
    stdLongWeekDay    // "01"
    stdWeekDay        // "Monday"
    stdDay            // "Mon"
    stdUnderDay       // "2"
    stdZeroDay        // "_2"
    stdHour           // "02"
    stdHour12         // "15"
    stdZeroHour12     // "3"
    stdMinute         // "03"
    stdZeroMinute     // "4"
    stdSecond         // "04"
    stdZeroSecond     // "5"
    stdLongYear       // "05"
    stdYear           // "2006"
    stdPM             // "06"
    stdpm             // "PM"
    stdTZ             // "pm"
    stdISO8601TZ      = iota // "MST"
    stdISO8601SecondsTZ // "Z0700" // prints Z for UTC
    stdISO8601ColonTZ  // "Z070000"
    stdISO8601ColonSecondsTZ // "Z07:00" // prints Z for UTC
    stdNumTZ            // "Z07:00:00"
    stdNumSecondsTZ    // "-0700" // always numeric
    stdNumShortTZ      // "-070000"
    stdNumColonTZ      // "-07" // always numeric
    stdNumColonSecondsTZ // "-07:00" // always numeric
    stdFracSecond0     // "-07:00:00"
    stdFracSecond9    // ".0", ".00", ... , trailing zeros included
                    // ".9", ".99", ... , trailing zeros omitted

    stdNeedDate = 1 << 8 // need month, day, year
    stdNeedClock = 2 << 8 // need hour, minute, second
    stdArgShift = 16 // extra argument in high bits, above low stdArgShift
    stdMask = 1<<stdArgShift - 1 // mask out argument
)
```


套件與常用函式庫

常用函式庫

- log
 - 寫任何服務、套件...，都免不了一定要有 log，golang 在這邊有提供最基礎的 log lib

```
package main

import (
    "log"
)

func main() {
    //最基礎的 印出 hello world
    log.Println("[standard] hello world")

    //設定輸出的格式
    log.SetFlags(log.Ltime)
    log.Println("[setflag] hello world")

    //設定前綴
    log.SetPrefix("[alvin]")
    log.Println("[prefix & setflag] hello world")
}
```

套件與常用函式庫

常用函式庫

- encoding/json

- 在這個萬物皆是 web 的時代，json 是一個不可或缺的格式，那就來看看怎麼使用 json：

- 這邊有用到一個 golang 很特殊的小功能叫做 struct tags，就是在右邊範例中 UserInfo 每個 field 後面的 json tag，需要更多的說明可以參閱[官網](#)。

- Json格式

- Json (JavaScript Object Notation) 是一種資料格式，由各種陣列(Array)以及鍵值(Key-Value)所組成的結構化格式。
 - [] 是陣列形式，裡面可以放各種物件。
 - {} 是鍵值形式，裡面可以放各種物件。
 - 兩種形式可以交錯使用。

```
package main

import (
    "encoding/json"
    "fmt"
)

type UserInfo struct {
    Name string `json:"name"`
    Age  int  `json:"age"`
}

func main() {
    var jsonString string
    jsonString = `{"name":"syhlion","age":5}`

    //把 json unmarshal 進去 struct
    u := &UserInfo{}
    err := json.Unmarshal([]byte(jsonString), u)
    if err != nil {
        fmt.Println(err)
        return
    }
    fmt.Printf("name:%s, age:%d\n", u.Name, u.Age)

    //把 struct 轉成 json 字串
    b, err := json.Marshal(u)
    if err != nil {
        fmt.Println(err)
        return
    }
    fmt.Println(string(b))
}
```

套件與常用函式庫

常用函式庫

- `regexp`
 - 正則表達式，也是各種應用中不可或缺的一個基礎功能。

```
package main

import (
    "fmt"
    "regexp"
)

func main() {
    re := regexp.MustCompile(`^[0-9]{3})(.*)`)
    m := re.MatchString("2345Abc")
    fmt.Println("2345Abc is match patter ? ", m)

    ss := re.FindStringSubmatch("2345Abc")
    fmt.Println("sub match array ", ss)
}
```

在這邊要注意一下，`golang` 的正則表達式是基於 `google` 開發的 `re2` 來實現的，因為這是為了解決目前現有的正則式演算法，時間複雜度並無固定的保證。

測試與驗證



測試

許多程式語言在主程式寫好後，還需要對程式進行測試，這時就需要寫測試的檔案。在 Go 語言中，測試框架已經有內建套件(testing)了，不需要在選擇框架或是安裝第三方套件。

go test

- 寫好了程式後，如果需要測試的檔案名稱為 `stat.go`，那麼需要新增一個檔案，這檔案名稱為檔案名後加上 `_test`，即為 `stat_test.go`。
- 先來看 `stat.go`：

```
package stat

func Mean(data []float64) (mean float64){
    var sum float64
    for _, v := range data{
        sum += v
    }
    mean = sum/float64(len(data))
    return
}
```

這是在計算一串數據的平均值，
接著就是要對這個函式做個測試了。

測試

go test

- 先新增 stat_test.go 這個檔案：

```
package stat

import "testing"

func TestMean1(t *testing.T) {
    if Mean([]float64{1,2,3}) != 2{
        t.Error("fail")
    }
}

func TestMean2(t *testing.T) {
    if Mean([]float64{1,9,5}) != 5{
        t.Error("fail")
    }
}

func TestMean3(t *testing.T) {
    if Mean([]float64{6,7,10}) != 23.0/3.0{
        t.Error("fail")
    }
}
```

一開始，需要載入套件 `testing`，接著必須寫一個 `function`，並且可能在裡頭運算要測試的函式，在看是否符合所想要的答案。

測試

go test

- 接著要在終端機運行下列指令：

```
$ go test -v -cover=true stat_test.go stat.go
=== RUN   TestMean1
--- PASS: TestMean1 (0.00s)
=== RUN   TestMean2
--- PASS: TestMean2 (0.00s)
=== RUN   TestMean3
--- PASS: TestMean3 (0.00s)
PASS
coverage: 100.0% of statements
ok      command-line-arguments 0.379s coverage: 100.0% of statements
```

- 而為什麼要把測試分成三個函式來寫呢？因為每個測試案例 (Test Case) 都應該是獨立的，每個測試函式都是一個測試案例，這樣比較能 check 是測試寫錯還是真的函式寫錯。

測試

go test

- 大致上需要注意以下的幾個規則
 1. 檔案名稱必須遵守`xxx_test.go`命名規則
 2. `function` 必須是`TestXxx`開頭
 3. `function` 參數必須 `† *testing.T`
 4. 執行檔案跟測試檔案必須是同一個`package`

驗證

Validator 資料驗證

- 如果有需要做資料或者數據相關的檢驗，可以考慮使用validator，跟先前一樣，需要先在終端機使用go get取得該套件，然後在import引用該套件。直接看範例程式碼：

- 執行起來會得到：

```
Key: 'User.Username' Error:Field validation for 'Username' failed on the 'min' tag
Key: 'User.Age' Error:Field validation for 'Age' failed on the 'lte' tag
Key: 'User.Sex' Error:Field validation for 'Sex' failed on the 'oneof' tag
```

- 從字面上來理解看看發生了什麼事情：

- User.Username 不符合min的規範
- User.Age 不符合lte的規範
- User.Sex 不符合oneof的規範

```
package main

import (
    "fmt"
    "github.com/go-playground/validator"
)

type User struct {
    Username string `validate:"min=4,max=10"`
    Age      uint8  `validate:"gte=1,lte=10"`
    Sex      string `validate:"oneof=female male"`
}

func main() {
    validate := validator.New()

    user1 := User{Username: "tom", Age: 11, Sex: "null"}
    err := validate.Struct(user1)
    if err != nil {
        fmt.Println(err)
    }

    user2 := User{Username: "Annabelle", Age: 8, Sex: "male"}
    err = validate.Struct(user2)
    if err != nil {
        fmt.Println(err)
    }
}
```

驗證

Validator 資料驗證

- 那可以設定那些規則呢?

- len : 等於(長度)
- max : 最大長度
- min : 最小長度
- eq : 等於(值)
- ne : 不等於該值
- gt : 大於該值
- gte : 大於等於該值
- lt : 小於該值
- lte : 小於等於該值
- onof : 必須是其中之一

- 所以看剛剛的code :

```
Username string `validate:"min=4,max=10"`  
Age        uint8  `validate:"gte=1,lte=10"`  
Sex        string `validate:"oneof=female male"`
```

Username 最小是4 最大是10

Age 必須大於等於1，小於等於10

Sex 必須是male和female其中一個