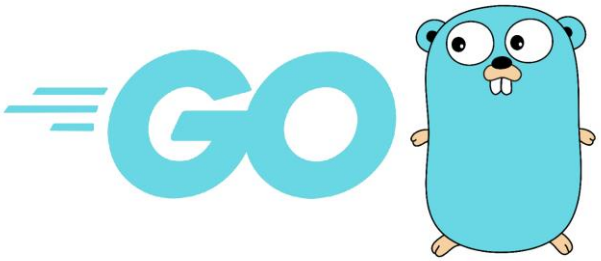
方法與介面



方法

方法(method)

- Go 語言不像 python 等程式有 classes，但是提供可以在某種型態上定義方法(method)，method其實是作用在接收者(receiver)上的一種函式，**接收者是某種型別的變數**，所以其實method也就是一種特殊型別的函式。
- 雖然go語言沒有類別，但方法也不是寫在結構內，而是寫在外面，並且會**透過接收者(receiver)來指定方法給一個結構型別**。下面先來看個範例，了解方法如何定義：

```go
package main

import (
    "fmt"
    "math"
)

type Vertex struct {
    X, Y float64
}

func (v Vertex) Abs() float64 {
    return math.Sqrt(v.X*v.X + v.Y*v.Y)
}

func main() {
    v := Vertex{3, 4}
    fmt.Println(v.Abs()) // 5
}
```

先宣告一個名為 Vertex 結構型態，裡面的屬性包含 X(float64) 和 Y(float64)，接著就是撰寫一個 method 了，這個 method 是以 Vertex 作為接收者，method 名稱為 Abs，最後回傳一個浮點數，接著 method 裡頭，即為對接收者的運算並回傳值。

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

方法

方法(method)

- 接收者(receiver)
  - Go語言的接收者像是其他語言中的 this 或 self，它就是一個連結呼叫物件的變數。會在 **func 和函式名稱之間**定義它，語法如下：

```go
func (p Point) AddX(val float64) {
    p.X = p.X + val
}

type Point struct {
    X, Y float64
}

func main() {
    var p = &Point{1.1, 2.5}
    p.AddX(1.2)
    fmt.Println(p)
}
// 執行結果: {1.1 2.5}
```

範例中發現一件事，明明就是宣告一個指標變數，怎麼執行方法卻沒有修改資料？

這是因為方法是用**傳值接受者**，當go看到指標變數呼叫的方法是傳值接受者，這時候go會**自動轉換**：

```go
p.AddX(1.2)       // 這是語法糖
(*p).AddX(1.2)    // 實際的語法
```

https://go.dev/play/p/HCFDUTIo5II

方法

方法(method)

- 接收者(receiver)
  - 因此，將上面的範例改成**傳指標接受者**，如下：

```go
func (p *Point) AddX(val float64) {   // <-- 這行 Point 加上 *
    p.X = p.X + val
}

type Point struct {
    X, Y float64
}

func main() {
    var p = &Point{1.1, 2.5}
    p.AddX(1.2)
    fmt.Println(p)
}
// 執行結果: {2.3 2.5}
```

這次資料就有被修改了，由於指標變數呼叫的方法是**傳指標接受者**，go也就不用轉換了。

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

方法

方法(method)

- 接收者(receiver)
  - 另外，這個情形也是可以倒過來，範例如下：

```go
func (p *Point) AddX(val float64) {
    p.X = p.X + val
}

type Point struct {
    X, Y float64
}

func main() {
    var p = Point{1.1, 2.5}  // <-- 這行刪除了 &
    p.AddX(1.2)
    fmt.Println(p)
}
// 執行結果: {2.3 2.5}
```

這個範例很神奇的是，宣告的 p 是一般變數，它不是指標。但是當它在呼叫方法後，竟然修改了資料。原因就和前一個範例剛好相反，**當go看到一般變數呼叫的方法是傳指標接收者**，這時候go也會**自動轉換**：

```go
p.AddX(1.2)        // 這是語法糖
(&p).AddX(1.2)  // 實際的語法
```

https://go.dev/play/p/jnqz-RsfFmx

方法

方法(method)

- 方法 vs 函式
  - 在前面有解釋到 method 即為特殊的函式，如何用函式來達到與 method 一樣的結果：

```go
package main

import (
    "fmt"
    "math"
)

type Vertex struct {
    X, Y float64
}

func Abs(v Vertex) float64 {
    return math.Sqrt(v.X*v.X + v.Y*v.Y)
}

func main() {
    v := Vertex{3, 4}
    fmt.Println(Abs(v))
}
```

https://go.dev/play/p/lprd2SVkqTy

方法

方法(method)

- 其他型別與方法
  - go語言中除了結構以外，其他的型別也是可以定義方法。但是go規定接收者只能是同個package裡的型別，**因此就必須使用型別別名**，如下：

```go
// 定義浮點數型別的別名
type MyFloat float64

func (a *MyFloat) Add(b float64) {
    *a = *a + MyFloat(b)
}

func main() {
    var a = MyFloat(2.2)
    a.Add(1.1)
    fmt.Println(a)
}
// 執行結果: 3.3000000000000003
```

https://go.dev/play/p/PY380h6YCpd

介面

介面(Interface)

- 目的是為了**在多種物件中找出共通性**，將這個共通性獨立出來。如果因為程式不大，可能會覺得interface這個方法多此一舉，但在實際上稍微有點規模的專案中，有效的運用interface可以讓整體管理更容易、程式碼更容易看懂等好處。
- 假設今天需要兩種形狀物件，圓形(circle)和正方形(square)
  - 圓形則是儲存半徑(radius)　　正方形是儲存邊長(side)

```
type circle {
    radius float64
}
```

```
type square struct {
    side float64
}
```

介面

介面(Interface)

- 這時候可能會想到求面積，但是圓形和正方形的面積求法一樣嗎？
  當然不同，這時候沒學過Interface可能會這樣寫：
- 到這邊有沒有發現，圓形和正方形都有形狀(shape)特徵，
  那不就剛好是Interface的使用最佳時機嗎？

```go
package main

import (
    "fmt"
    "math"
)

type circle struct {
    radius float64
}

type square struct {
    side float64
}

func (c circle) area() float64 {
    return math.Pi * c.radius * c.radius
}

func (s square) area() float64 {
    return s.side * s.side
}

func main() {
    c := circle{2}
    s := square{5}
    fmt.Println("Circle area: ", c.area())
    fmt.Println("Square area: ", s.area())
}
```

https://play.golang.org/p/yKKdo0gh9q

介面

介面(Interface)

- 其實和struct差不多，只要任何的struct有interface裡面的所有function，就算是實作了這個interface：

```
type shape interface {
    area() float64
}
```

- 只要struct實作area() float64就算是達成了shape，要一模一樣才算，不可以是area() int也不可以是area(x int) float64，這時候程式碼應該長這樣：

https://play.golang.org/p/uMIKyVX6SN

```
package main

import (
    "fmt"
    "math"
)

type circle struct {
    radius float64
}

type square struct {
    side float64
}

type shape interface {
    area() float64
}

func (c circle) area() float64 {
    return math.Pi * c.radius * c.radius
}

func (s square) area() float64 {
    return s.side * s.side
}

func main() {
    c := circle{2}
    s := square{5}
    fmt.Println("Circle area: ", c.area())
    fmt.Println("Square area: ", s.area())
}
```

介面

介面(Interface)

- Go 語言的介面(interface)是一組以方法簽名(method signatures)的組合，透過介面來定義物件的一組行為，它**將物件導向的內容組織**，實現得非常方便。
- Interface 定義了一組 method，Interface 裡的值可以保存實現這些方法的任何值。直接看以下的程式碼：
- Abser interface 實現了 Abs method，而 Abs method 同時定義在 Vertex struct 和 MyFloat 上。

https://go.dev/play/p/p8id5IqLPXQ

```go
package main

import (
    "fmt"
    "math"
)

type Abser interface {
    Abs() float64
}

func main() {
    var a Abser
    f := MyFloat(-math.Sqrt2)
    v := Vertex{3, 4}

    a = f  // a MyFloat implements Abser
    fmt.Println(a.Abs())

    a = &v // a *Vertex implements Abser
    fmt.Println(a.Abs())
}

type MyFloat float64

func (f MyFloat) Abs() float64 {
    if f < 0 {
        return float64(-f)
    }
    return float64(f)
}

type Vertex struct {
    X, Y float64
}

func (v *Vertex) Abs() float64 {
    return math.Sqrt(v.X*v.X + v.Y*v.Y)
}
```

介面

介面(Interface)

- Interface 值
  - Interface 裡到底能存什麼值呢？如果定義了一個 interface 的變數，那麼這個變數可以實現該 interface 裡所有的變數型態。
  - 例如：前面例子：Abser interface 最終實現 Vertex struct 和 MyFloat 兩種變數型態，而最後儲存 float64 這個變數型態。再來看一個例子：
    - Interface I 實現了 M method，而 M method 定義在 T struct 和 F float64上，**最後 M method 印出了兩個型態變數中帶的參數**

https://go.dev/play/p/O8zvddb29m7

```go
package main

import (
   "fmt"
   "math"
)

type I interface {
   M()
}

type T struct {
   S string
}

func (t *T) M() {
   fmt.Println(t.S)
}

type F float64

func (f F) M() {
   fmt.Println(f)
}

func main() {
   var i I

   i = &T{"Hello"}
   describe(i)
   i.M()

   i = F(math.Pi)
   describe(i)
   i.M()
}

func describe(i I) {
   fmt.Printf("(%v, %T)\n", i, i)
}
```

介面

介面(Interface)

- 當有了interface之後，就可以將interface當參數傳遞，很方便的！

  這時候只要是達成shape要求的都算是shape：

```
func info(x shape) {
    fmt.Println("This is a " + x)
    fmt.Println("The area are " + x.area())
}
```

https://play.golang.org/p/KhYN3P4iz6

```
package main

import (
    "fmt"
    "math"
)

type circle struct {
    radius float64
}

type square struct {
    side float64
}

type shape interface {
    area() float64
}

func (c circle) area() float64 {
    return math.Pi * c.radius * c.radius
}

func (s square) area() float64 {
    return s.side * s.side
}

func info(x shape) {
    fmt.Println(x.area())
}

func main() {
    c := circle{2}
    s := square{5}
    info(c)
    info(s)
}
```

介面

介面(Interface)

- 空 Interface (泛用型別)
  - **空 interface 不包含任何的 method，所以所有的型別都實現了空 interface**。空 interface 對於描述起不到任何的作用(因為它不包含任何的 method)，但是在需要儲存任意型別的數值時，空 interface 及發揮到他的作用，因為它**可以儲存任意變數型態**，來看一下下面的例子：

```go
func main() {
    var a interface{}
    a = 123
    fmt.Println(a, reflect.TypeOf(a))
    a = "hi"
    fmt.Println(a, reflect.TypeOf(a))
    a = true
    fmt.Println(a, reflect.TypeOf(a))
}
```

```
/* result:
123 int
hi string
true bool
*/
```

```go
package main

import (
   "fmt"
)

var a interface{}

func main() {
   // 定義 a 為空介面

   var i int = 5
   s := "Hello world"
   // a 可以儲存任意型別的數值
   a = i
   fmt.Println(a) // 5

   a = s
   fmt.Println(a) // Hello world
}
```

https://go.dev/play/p/c5twNB-Y-V8

介面

介面(Interface)

- Interface 值
  - 那interface到底是什麼型態？

```
func main() {
    var a interface{}
    fmt.Println(a, reflect.TypeOf(a))
}
/* result:
<nil> <nil>
 */
```

  - 居然是 nil，不過這也正常，因為什麼東西都還沒給，所以還沒實體化。**除了任意值之外，interface還有一個廣泛的用法，可以定義func的名字在interface之中。這樣可以方便容易地抽層，讓程式架構多一層，能讓程式更抽象化、更具有程式彈性！**

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

介面

介面(Interface)

- nil Interface 值
  - **一個 nil interface 值既不包含值也不包含具體類型。**
  - 在 nil interface 上使用方法是一個運行時錯誤，因為 interface 裡沒有任何類型定義在任何方法上。

```go
package main

import "fmt"

type I interface {
   M()
}

func main() {
   var i I
   describe(i)
   i.M()
}

func describe(i I) {
   fmt.Printf("(%v, %T)\n", i, i)
}
```

https://go.dev/play/p/Z1w0m0OqKvz

介面

介面(Interface)

- 使用Go提供的某些功能時必需先實作介面：以**排序**為例
  - 究竟什麼時候會需要實作介面呢？舉個例子來說好了，Go已經提供了一個可以幫忙排序的工具了，但是一些相關的參數必須自己決定，比如長度、大小和怎麼把兩個要排序的元素做交換，比如今天要請廠商做衣服，必需先提供圖案給廠商，廠商才能開始運作。
  - 要使用排序這個功能，得先使用 import 將該套件 sort 引入進來，因為同時會使用 fmt 所以直觀上是這種寫法：

```
import "fmt"
import "sort"
```

  - 但是如果要引用很多套件得一直重打 import 很麻煩所以可以簡單記為：

```
import(
    "fmt"
    "sort"
)
```

介面

介面(Interface)

- 使用Go提供的某些功能時必需先實作介面：以排序為例
  - STEP1：確認要使用的函式
    - 這次確定要使用 sort.Sort() 函式，但是裡頭的參數是 sort.Interface 型態。
    - Sort 會排序 data。 它會呼叫 data.Len 來決定 n，並呼叫 data.Less 和 data.Swap。

介面

介面(Interface)

- 使用Go提供的某些功能時必需先實作介面：以排序為例
  - STEP2: 尋找如何實作 Interface
    - 要使用 sort.Sort() 前必需先實作 sort.Interface

```
type Interface interface {
    // Len is the number of elements in the collection.
    Len() int
    // Less reports whether the element with
    // index i should sort before the element with index j.
    Less(i, j int) bool
    // Swap swaps the elements with indexes i and j.
    Swap(i, j int)
}
```

Len()：要排序的東西總數

Less(i, j int) bool：如果第 i 個要在第 j 個前面回傳 true, 否則回傳 false

swap(i, j int)：定義如果把第 i 個和第 j 個交換

```
imoprt(
    a "foo/aaa"
    b "bar/aaa"
)

func main(){
    a.F()
    b.F()
}
```

1. Less(i, j int)與Less(i int, j int) 意思相同
2. 使用引入的套件前面要加入套件的名稱，比如引入 sort 時若要使用 sort 裡的函式、常數、型態，必需在這些東西前方加上 sort。比如：sort.Sort(), sort.Interface, fmt.Println()
3. 如果引用的套件是encoding/json，這時套件名稱為json，總之就是挑「最後的那一個字串」。如果有兩個重複的套件名稱呢？比如：foo/aaa、 bar/aaa，這時只要加上別稱就行了(右上)：

介面

介面(Interface)

- 使用Go提供的某些功能時必需先實作介面：以排序為例
  - STEP3: 排序一個 int slice
    - 要由小到大排序：`list := []int{1, 4, 8, 3, 5, 7, 9, 6}`
    - 實作時馬上碰到問題，**因為無法對 []int 增加方法(無法對Golang原生型態新增方法)，試著使用自己的型態，對 []int 取別名**：

```
type myList []int

func main(){
    list := []int{1, 4, 8, 3, 5, 7, 9, 6}
    newList := myList(list) // 「轉型」成自定型態
```

介面

介面(Interface)

- 使用Go提供的某些功能時必需先實作介面：以排序為例
  - STEP3: 排序一個 int slice
    - 對自訂義的型態新增三個方法，來滿足 sort.Interface：

```
func (list myList) Len() int{
    return len(list)
}

func (list myList) Less(i, j int) bool{
    return list[i] < list[j]
    // 希望比較小的放前面 (升序)
}

func (list myList) Swap(i, j int){
    list[i], list[j] = list[j], list[i]
    // 注意如果要分開寫要寫成：
    // tmp := list[i]
    // list[i] = list[j]
    // list[j] = tmp
    // 因為 Golang 允許多對多賦值所以可以一行完成
    // 就不會使用傳統的交換方法了
}
```

    - 最後合起來：

https://go.dev/play/p/XP7kuD45EIN

```
package main
import (
    "fmt"
    "sort"
)

type myList []int

func (list myList) Len() int{
    return len(list)
}

func (list myList) Less(i, j int) bool{
    return list[i] < list[j]
}

func (list myList) Swap(i, j int){
    list[i], list[j] = list[j], list[i]
}

func main(){
    list := []int{1, 4, 8, 3, 5, 7, 9, 6}
    newList := myList(list) // 轉型
    fmt.Println(newList)
    sort.Sort(newList)
    fmt.Println(newList)
}
```

執行結果：

```
[1 4 8 3 5 7 9 6]
[1 3 4 5 6 7 8 9]
```

介面

介面(Interface)

- 使用Go提供的某些功能時必需先實作介面：以排序為例
  - STEP3: 排序一個 int slice
    - 如果想要從大到小排序呢？這時只需要更改 Less() 即可：

```
func (list myList) Less(i, j int) bool{
    return list[j] < list[i]
}
```

    - 替換後：

```
執行結果：
[14835796]
[98765431]
```