

CH7 介面

7-1 前言

- 假設今天你設計了三個函式，它們對資料處理的流程一模一樣，只有讀入資料的型別有所不同，那麼能不能只寫一個函式就好？你又該如何繞過不同資料型別帶來的限制，讓一個函式可以接收不同型別的值？
- 這個問題的答案就是使用介面(**interface**)，本章要來研究介面為何可以接收多重型別，並且探討介面如何引進鴨子定型(**duck typing**)和多型(**polymorphism**)的機制，最後會回顧第四章提過的空介面型別

7-2 介面(interface)

7-2-1 認識介面

- 在第四章提過，你能對自訂型別或結構綁定方法(method)，方法其實代表型別的行為
- 一個介面所做的事，就是描述了某事物應具備哪些行為，例如若“車輛”介面定義了一輛車必須有煞車的行為，那不管事“汽車”，“腳踏車”型別只要具備相同行為都會被當成“車輛”看待

- 更精確來說，**go**語言的介面型別會包含一系列函式或方法特徵，而其他型別只要定義了完全相同的方法，就等於實作(**implement**)了此介面，並可以被當作該型別看待：

```
type 型別名稱 interface {  
    方法 1 特徵  
    方法 2 特徵  
    ....  
}
```

下面是個例子，示範了一個介面**Speaker**應實作的方法，這些方法只有特徵(名稱/參數/回傳值)而不帶實作細節：

```
type Speaker interface {  
    Speak(message string) string //方法特徵  
    Greet() string //方法特徵  
}
```

7-2-2 定義型別介面

- 我們沿用前面的Speaker()介面範例，來看看定義介面的步驟：

```
type Speaker interface {  
    Speak(message string) string //方法特徵  
    Greet() string //方法特徵  
}
```

1. 首先是關鍵字type，接著是介面名稱，最後是另一個關鍵字interface
2. 慣例上介面名稱會拿其中一個方法的名稱加上er結尾
3. 在介面的大括號之間定義方法特徵

- 以下是前面提過的介面io.Reader, 定義於go語言io套件中：

```
type Reader interface {  
    Read(p []byte) (n int, err error)  
}
```

- 可以看到介面名稱是Reader, 它唯一的方法是Read(), 而Read()方法的參數與回傳值特徵是(p []byte) (n int, err error)

- 介面也可以有眾多方法，下面是另一個go語言套件的例子：

```
type FileInfo interface {  
    Name() string //base name of the file  
    Size() int64  //length in bytes for regular files; system-dependent for others  
    Mode() FileMode // file mode bits  
    IsDir() bool   //abbreviation for Mode().IsDir()  
    Sys() interface{} //underlying data source (can return nil)  
}
```

- 可見os.FileInfo介面有6個方法，而任何型別想符合這個介面，就要實作出以上這些方法(所有方法的特徵都得符合才行)

- 總結來說，介面是一種型別，但其內容就是方法特徵的集合
- 與其他程式語言類似的是，**go**語言的介面型別都不會明訂實作者要如何撰寫這些方法，畢竟實作細節並非介面定義的一部分
- 接下來就來介紹如何用**go**語言實作一個介面

7-2-3 實作一個介面

- 其他程式語言都必須以明確的方式實作介面；也就是你必須明白陳述一個物件是沿用哪個介面的規範
- 以Java為例：class Dog implements Pet
- 上面明確指出，Dog型別(class)實作了Pet介面，也就是說，現在Dog類別已表明要實作Pet介面，它就必須實作介面要求的方法，否則會產生錯誤

- 不過在go語言中，介面實作是隱性的，意思是，只要一個型別綁訂的方法特徵完全符合一個介面的規範，該型別就等於是自動實作了該介面
- 以下是個例子：

```
1 package main
2
3 import "fmt"
4
5 type Speaker interface { //Speaker介面
6     Speak() string
7 }
8
9 type cat struct { //cat結構
10 }
11
12 func main() {
13     c := cat{}
14     fmt.Println(c.Speak())
15     c.Greeting()
16 }
17
18 func (c cat) Speak() string { //cat的方法(使cat符合Speaker介面)
19     return "Purrr"
20 }
21
22 func (c cat) Greeting() { //cat的方法
23     fmt.Println("Meow")
24 }
```

執行結果：

```
Purrr
```

```
Meow
```

- 來分析一下剛剛的程式碼：

```
type Speaker interface { //Speaker介面  
    Speak() string  
}
```

- 這裡定義了一個Speaker介面，裡面列出一個方法特徵Speak() string
- 接著我們建立一個名為cat的空白結構型別，並給它一個特徵相同的方法Speak()：

```
type cat struct { //cat結構  
}  
  
func (c cat) Speak() string {  
    return "Purrr"  
}
```

- `cat` 的 `Speak()` 方法滿足了 `Speaker` 介面的規範，因此 `cat` 型別就可以被視為 `Speaker` 型別
- 注意到程式裡並沒有明確敘述指出 `cat` 是在實作 `Speaker` 介面；當你替 `cat` 定義 `Speak()` 方法時，隱性實作就發生了
- 另外在前面的程式碼中，`cat` 結構還有另一個方法 `Greeting`，這個方法並未定義在 `Speaker` 中，但既然 `cat` 型別已經滿足了 `Speaker` 型別，就不會影響 `cat` 對 `Speaker` 介面的實作
- 總結來說，一個型別只要擁有和一個介面完全相同的方法(實作該介面)，就會被視為該介面型別

7-2-4 隱性介面實作的優點

- 在其他程式語言中實作介面，你必須明確表明意圖，且若你修改了一個介面定義的方法，就要一一找出實作該介面的類別和修改，或將不符合資格的 `implements` 關鍵字移除
- 在 `go` 語言中，只要型別的行為滿足了某個介面，就會自動實作該介面，若你後來修改了介面的方法集合，不符合資格的實作也會自動失效(不會影響到該型別的其他方面)
- 另一個優點是，你可以讓型別實作其他套件內定義的介面，這麼一來就能將介面與其實作型別的定義分開，我們會在第8章討論如何運用自訂套件來將程式功能分類

- 我們在下面來看一個在主程式(main套件)中運用其他套件定義的介面的例子

- `fmt`套件的**Stringer**就是一個例子，不僅許多套件會用到，`fmt`套件本身也會用它來印出資料：

```
type Stringer interface{  
    String() string  
}
```

- 以**fmt**為例，若傳入的型別符合**Stringer**介面型別，那**Println()**就會呼叫其**String()**方法來取得字串
- 現在來修改前面的**cat**結構範例，給它加上兩個欄位，刪除**Greeting()**方法，換上一個新方法**String()**：

```
package main

import "fmt"

type Speaker interface {
    Speak() string
}

type cat struct { //加入欄位
    name string
    age  int
}

func (c cat) Speak() string {
    return "Purr Meow"
}

func (c cat) String() string { //String()方法
    return fmt.Sprintf("%v(%v years old)", c.name, c.age)
}

func main() {
    c := cat{name: "Oreo", age: 9}
    fmt.Println(c.Speak())
    fmt.Println(c) //用fmt套件直接印出cat
}
```

執行結果：

```
Purr Meow  
Oreo(9 years old)
```

- 現在`cat`結構型別同時實作了兩個介面，一個是在`main`內自訂的`Speaker`，另一個是來自`fmt`套件的`Stringer`
- 目前我們還沒有程式會用到`Speaker`介面，但可以發現用`fmt.Println()`印出`cat`結構變數`c`時，`Println()`自動呼叫了它的`String()`方法，這代表`c`符合並實作了`Stringer`介面，使它能夠被`Println()`接受和表現出特定行為

補充：實作介面時使用值接收器VS指標接收器

- 如果把前面程式碼中的**cat**方法改成如下：

```
14 func (c *cat) Speak() string { //用指標接收器來指向cat結構變數
15     return "Purr Meow"
16 }
17
18 func (c *cat) String() string { //用指標接收器來指向cat結構變數
19     return fmt.Sprintf("%v(%v years old)", c.name, c.age)
20 }
```

- 改成指標接收器後你會發現，`fmt.Println()`沒有呼叫`cat.String()`了，單純印出結構內容而已：

```
Purr Meow  
{Oreo 9}
```

- 這是因為加上指標接收器後，就變成是指標型別`*cat`而不是型別`cat`去實作`Stringer`介面，所以`cat`變數不被認為符合`stringer`介面
- 解決方法是把`cat`變數宣告成指標：

```
c := &cat{name: "Oreo", age: 9}
```

練習: 實作一個介面

- 在這個練習中，會先定義一個person結構型別，含有name, age, isMarried等欄位
- 它用有Speak()方法，隱含實作了自訂的Speaker()介面，此外它也有String()方法來隱含實作fmt套件的Stringer介面


```
1  package main
2
3  import (
4      |   "fmt"
5  )
6
7  type Speaker interface {
8      |   Speak() string
9  }
10
11 type person struct {
12     |   name      string
13     |   age       int
14     |   isMarried bool
15 }
16
17 func main() {
18     |   p := person{name: "Cailyn", age: 44, isMarried: false}
19     |   fmt.Println(p.Speak())
20     |   fmt.Println(p)
21 }
22
```

```
23  ✓ func (p person) String() string { //實作Stringer介面
24      |     return fmt.Sprintf("%v (%v 歲)\n已婚: %v ", p.name, p.age, p.isMarried)
25      | }
26
27  ✓ func (p person) Speak() string { //實作Speaker介面
28      |     return "各位好，我的名字是 " + p.name
29      | }
```

執行結果：

```
各位好，我的名字是 Cailyn  
Cailyn (44 歲)  
已婚: false
```

7-3 鴨子定型和多型

7-3-1 鴨子定型

- 前面實作**Speaker**和**Stringer**介面時所做的事，其實就是所謂的鴨子定型
- 鴨子定型是程式設計中的一種歸納推理：只要一個東西長得像鴨子，叫聲像鴨子，游泳像鴨子，那麼他就是鴨子
- 以**go**語言來說，任何型別只要符合某個介面的行為規範，那就能被當成該介面型別使用
- 意即，**go**語言的鴨子定型式根據型別方法來判斷型別符合介面，而不是明確的指定哪些型別能夠符合
- 下面來看一個例子：

```
1  package main
2
3  import "fmt"
4
5  type Speaker interface {
6      Speak() string
7  }
8
9  type cat struct {
10 }
11
12 func (c cat) Speak() string { //cat實作Speaker介面
13     return "Purr Meow"
14 }
15
16 func chatter(s Speaker) { //接收Speaker介面型別的引數
17     fmt.Println(s.Speak())
18 }
19
20 func main() {
21     c := cat{}
22     chatter(c)
23 }
```

執行結果：

Purr Meow

- 這次我們有一個函式`chatter()`，它接收的參數型別是`Speaker`介面
- 既然`cat`隱性實作了`Speaker`介面，因此他透過鴨子定型被視為`Speaker`型別，因此可以傳入`chatter()`的參數

7-3-2 多型

- 多型指的是一樣東西可以用多種形式呈現，舉例來說，一個形狀可以是正方形,長方形或其他任意形狀
- 在其他物件導向的程式語言中，子類別化意指讓一個類別繼承另一個型別的欄位和行為(例如, 圓形會從形狀繼承面積)
- 若你設計出多個子類別，每個子類別都經過修改而各有差異，這就是物件導向的多型
- 當然go語言並不是物件導向的語言，但仍可以透過內嵌結構和介面來實現類似子類別化的概念

- 在go語言是用多型的好處之一，是若你手上有已經寫好且經過測試的程式碼，你就可以重複利用它
- 你只要讓該函式接收介面型別參數，那麼任何符合介面型別規定的型別都可以傳入，而不僅限於int, float, bool等核心型別
- 你甚至不需要在函式中額外撰寫程式碼來應付每一種型別，因為只有正確實作介面的型別才能傳入這個函式

- 來看一個較為進階的例子，展示如何在go語言中運用多型：
 - 如我們前面看過的，任何實質型別都能實作一種以上的介面，反過來說，同一個介面也可以被多個型別實作，例如**Speaker**可以同時由**dog**, **cat**, **person**型別實作
 - 如果**cat**, **dog**和**person**都實作了**Speaker**介面，這代表他們一定都有**Speaker()**方法，且會回傳一個字串
 - 這表示我們可以撰寫一個共用函式，接收**Speaker**介面型別的參數，然後對任何傳入的值呼叫相同的行為：

```
1 package main
2
3 import "fmt"
4
5 type Speaker interface {
6     Speak() string
7 }
8
9 type cat struct {
10 }
11
12 type dog string
13
14 type person struct {
15     name string
16 }
17
18 func main() {
19     c := cat{}
20     d := dog("")
21     p := person{name: "Heather"}
22     thingSpeak(c)
23     thingSpeak(d)
24     thingSpeak(p)
25 }
26
```

```
27 func (c cat) Speak() string {
28     return "Purr Meow"
29 }
30
31 func (d dog) Speak() string {
32     return "Woof Woof"
33 }
34
35 func (p person) Speak() string {
36     return "Hi, my name is " + p.name + "."
37 }
38
39 func thingSpeak(s Speaker) {
40     fmt.Println(s.Speak())
41 }
```

執行結果：

```
Purr Meow  
Woof Woof  
Hi, my name is Heather.
```

- 下面來做一點變化，把thingSpeak()換稱接收數量不定的參數：

```
func thingSpeak(speakers ...Speaker) {  
    for _, s := range speakers {  
        fmt.Println(s.Speak())  
    }  
}
```

- 接著就能修改main()內呼叫thingSpeak()的方式如下：

```
thingSpeak(c, d, p)
```

- 執行的結果與之前一樣，不果程式碼變得比較簡潔

練習：使用多型別來計算不同形狀的面積

- 現在我們要寫一個程式，能印出圓形，正方形和三角形的名稱和面積
- 負責印出資訊的函式會接收**Shape**這個介面型別參數的數量不定參數，使任何滿足**Shape**規範的形狀都可以當成引數傳入


```
1  package main
2
3  import (
4      "fmt"
5      "math"
6  )
7
8  type shape interface {
9      Area() float64
10     Name() string
11 }
12
13 type circle struct {
14     radius float64
15 }
16
17 type square struct {
18     side float64
19 }
20
21 type triangle struct {
22     base    float64
23     height float64
24 }
25
```

```
26 func main() {
27     s := square{side: 10}
28     c := circle{radius: 6.4}
29     t := triangle{base: 15.5, height: 20.1}
30     printShapeDetails(s, c, t)
31 }
32
33 func printShapeDetails(shapes ...shape) {
34     for _, item := range shapes {
35         fmt.Printf("%s的面積: %.2f\n", item.Name(), item.Area())
36     }
37 }
38
```

```
39 func (c circle) Area() float64 {
40     return c.radius * c.radius * math.Pi
41 }
42
43 func (c circle) Name() string {
44     return "圓形"
45 }
46
47 func (s square) Area() float64 {
48     return s.side * s.side
49 }
50
51 func (s square) Name() string {
52     return "正方形"
53 }
54
55 func (t triangle) Area() float64 {
56     return (t.base * t.height) / 2
57 }
58
59 func (t triangle) Name() string {
60     return "三角形"
61 }
```

執行結果：

正方形的面積：100.00

圓形的面積：128.68

三角形的面積：155.78

7-4 在函式中活用介面

7-4-1 以介面為參數的函式

- 本章一開始提過`io.Reader`介面可以用來接收不同型別的值，現在來看看實際的運用效果
- 以下範例會寫出兩個任務相同的函式，用來解碼三筆JSON格式文字，但這兩個函式的參數型別不同，一個是字串，另一個是`io.Reader`介面
- 此外，兩筆JSON資料是字串，但第三筆資料儲存在專案目錄下的文字檔`data.json`，會被go讀取成`io.File`檔案物件：

```
{"Name": "John", "Age": 20}
```

- 我們會在第**11**章探討JSON資料的處理，並在第**12**章講到檔案讀寫
- 以下是範例主程式：

```
1  package main
2
3  √ import (
4      "encoding/json"
5      "fmt"
6      "io"
7      "os"
8      "strings"
9  )
10
11 √ type Person struct { //用於JSON資料的結構
12     Name string `json:"name"`
13     Age  int    `json:"age"`
14 }
15
```



```
16 func main() {
17     s := `{"Name":"Joe","Age":18}` //第一筆資料
18     s2 := `{"Name":"Jane","Age":21}` //第二筆資料
19
20     //第一筆資料 (字串)
21     p, err := loadPerson(s)
22     if err != nil {
23         fmt.Println(err)
24     }
25     fmt.Println(p)
26
27     //第二筆資料
28     //strings.NewReader()會回傳一個strings.Reader結構,符合io.Reader介面
29     p2, err := loadPerson2(strings.NewReader(s2))
30     if err != nil {
31         fmt.Println(err)
32     }
33     fmt.Println(p2)
34
35     //第三筆資料 (讀取檔案後傳回os.File結構,符合io.Reader介面)
36     f, err := os.Open("data.json") //開啟同資料夾下的文字檔
37     if err != nil {
38         fmt.Println(err)
39     }
40     p3, err := loadPerson2(f)
41     if err != nil {
42         fmt.Println(err)
43     }
44     fmt.Println(p3)
45 }
46
```

```
47 //第一個JSON解析函式,接收字串參數
48 func loadPerson(s string) (Person, error) {
49     var p Person
50     err := json.NewDecoder(strings.NewReader(s)).Decode(&p)
51     if err != nil {
52         return p, err
53     }
54     return p, nil
55 }
56
57 //第二個JSON解析函式,接收io.Reader介面參數
58 func loadPerson2(r io.Reader) (Person, error) {
59     var p Person
60     err := json.NewDecoder(r).Decode(&p)
61     if err != nil {
62         return p, err
63     }
64     return p, err
65 }
```

執行結果 (要記得切換到程式碼所在的目錄)

```
PS D:\git\Golang\ch7> go run "d:\git\Golang\ch7\7-4-1.go"  
{Joe 18}  
{Jane 21}  
{John 20}
```

- `json`套件的`NewDecoder()`函式能夠解析JSON資料，它實際上會接收一個`io.Reader`介面參數，並傳回解碼過的`Decoder`結構：

```
func NewDecoder(r io.Reader) *Decoder
```

- 然後程式會呼叫`Decoder`的`Decode()`方法，將資料寫入`Person`結構變數的欄位，細節留至第11章再討論
- 為了示範起見，函式`loadPerson()`會接收一個`string`型別引數，然後再呼叫`string.NewReader()`把字串轉成`strings.Reader`結構傳給`json.NewDecoder()`；`strings.Reader()`即是實作了`io.Reader`介面的結構型別
- 至於在功能完全相同的函式`loadPerson2()`中，我們就直接接收一個`io.Reader`介面參數，重複完全一樣的過程

- 現在再來看一下io.Reader介面定義：

```
type Reader interface {  
    Read(p []byte) (n int, err error)  
}
```

- 這顯示io.Reader介面要求型別有一個Read()方法，接收[]byte切片(這可以代表字串)並回傳一個int和error
- 如果看strings.Reader和os.File的定義，都會發現他們都實作了這個方法，這解釋了為何json.NewDecoder()能接收不同型別的值，並正確讀出JSON資料

- `io.Reader`是go語言標準函式庫中最常用的介面之一，這意味著你能解碼JSON資料的來源並不僅於此
- 當你再開發API時，使用介面型別作為參數，就意味著使用者傳入的資料不會受限於特定型別，可以更加的彈性

7-4-2 以介面為傳回值的函式

- 我們在第6章已經看過，任何型別只要實作**Error() string**方法就可以符合go語言的**error**介面，而事實上每個套件都會定義他們自己的**error**型別
- 下面就拿前一個範例做點修改，看看不同套件傳回的**error**值實際上是甚麼型別：

```
1  package main
2
3  import (
4      "encoding/json"
5      "fmt"
6      "os"
7  )
8
9  type Person struct {
10     Name string `json:"name"`
11     Age  string `json:"age"` //故意將欄位型別改錯
12 }
13
14 func main() {
15     p, err := loadPerson("data.json") //讀取同目錄下的文字檔
16     if err != nil {
17         //若有錯誤，印出其值和型別
18         fmt.Printf("%v", err)
19         fmt.Printf("%T", err)
20     }
21     fmt.Println(p)
22 }
23
```



```
24 func loadPerson(fname string) (Person, error) {
25     var p Person
26     f, err := os.Open(fname)
27     if err != nil {
28         return p, err //傳回檔案開啟錯誤
29     }
30     err = json.NewDecoder(f).Decode(&p)
31     if err != nil {
32         return p, err //傳回JSON解析錯誤
33     }
34     return p, err
35 }
```

- 現在loadPerson()會接收一個檔名，它同時會完成讀取檔案和解析JSON字串的任務
- 然而，程式中Person結構的Age欄位使用了錯誤的型別，因此會傳回JSON解析錯誤(*json.UnmarshalTypeError型別)：

執行結果

```
PS D:\git\Golang\ch7> go run "d:\git\Golang\ch7\7-4-2.go"  
json: cannot unmarshal number into Go struct field Person.age of type string*json.UnmarshalTypeError{John }
```

- 若把 `p, err := loadPerson("data.json)` 這行檔名也故意寫錯，然後執行程式：
- 結果：`open data1.json: The system cannot find the file specified.`
`*fs.PathError{ }`
- 可以看到這次傳回的 `err` 變數是 `*fs.PathError` 型別，這顯示了 `go` 語言可以透過 `error` 介面傳回不同型別的錯誤

- 究竟應不應該使用函式回傳介面，以下是簡單的方針：
 - 若無必要，不要讓函式回傳介面型別
 - 介面定義越精簡越好
 - 盡量在實值型別(需求)存在後才撰寫介面
 - 通常介面會定義在該型別的套件內，對外用不到的介面就不應該匯出

7-4-3 空介面 interface{}

- 沒有任何方法，也沒有定義任何行為的介面就是空介面，寫法如下：

```
interface{}
```

- 大家應該還有印象，go語言的介面實作是隱性的，那既然空介面未指定任何方法，這表示go語言的任何型別都會自動實作空介面，所以才會說任何型別都會自動符合空介面的規範
- 以下的程式碼中，會展示如何透過空介面，來接收任意型別的傳入值：

```
package main

import "fmt"

type cat struct {
    name string
}

func main() {
    i := 99           //整數形別
    b := false        //布林值
    str := "test"     //字串
    c := cat{name: "oreo"} //cat結構型別
    printDetails(i, b, str, c)
}

func printDetails(data ...interface{}) { //接收不定數量的空介面參數
    for _, i := range data {
        fmt.Printf("%v, %T\n", i, i) //印出值和型別
    }
}
```

執行結果：

```
99, int  
false, bool  
test, string  
{oreo}, main.cat
```


7-4-4 型別斷言與型別switch

- 型別斷言(**type assertion**)可以讓你檢查並取用介面背後的實值型別
- 剛剛說過**interface{}**可以接收任何型別的值，但在處理資料實必須知道介面底下的實值型別為何，才能做正確的處理
- 若試圖直接轉換空介面的型別，會發生甚麼是呢？下面試著用 **strconv.Atoi()**把底層值為字串的空介面轉成整數：

```
1  package main
2
3  import (
4      "fmt"
5      "strconv"
6  )
7
8  func main() {
9      var s interface{} = "42"
10     fmt.Println(strconv.Atoi(s))
11 }
```

執行結果：

```
# command-line-arguments
ch7\7-4-4.go:10:27: cannot use s (variable of type interface{}) as type string in argument to strconv.Atoi:
    need type assertion
```

錯誤訊息指出**strconv.Atoi**的參數不能接收空介面型別，必須做型別斷言才行

- 我們來回顧第4章提過的型別斷言語法：

$v := s.(T)$

- 上面的敘述是：用斷言主張介面值 s 底下的型別是 T ，若確實是就將 T 型別的值 s 賦予給 v ：

```
package main

import (
    "fmt"
    "strconv"
)

func main() {
    var s interface{} = "42"
    v := s.(string)           //用型別斷言把介面轉成字串
    fmt.Println(strconv.Atoi(v)) //將字串轉為整數
}
```

- 執行結果：

```
42 <nil>
```

- 第二個值是`strconv.Atoi()` 傳回的`error`, `nil`代表轉換成功
- 使用型別斷言，若值跟你想要轉換的型別一樣當然最好，但是萬一型別不符怎辦？讓我們先看看轉換失敗會發生甚麼事：

```
100, 415 別 | 1 author (100)
```

```
package main
```

```
import (
```

```
    "fmt"
```

```
    "strconv"
```

```
)
```

```
func main() {
```

```
    var i interface{} = 42 //換成整數
```

```
    s := i.(string)
```

```
    fmt.Println(strconv.Atoi(s))
```

```
}
```

- 我們試圖用型別斷言把介面轉成字串，但它底下的實質型別為int，因此引發了panic:

```
panic: interface conversion: interface {} is int, not string
```

- 幸好型別斷言會回傳第二個值(布林值), 告訴我們是否轉換成功:

```
func main() {  
    var str interface{} = 42  
    if s, ok := str.(string); ok { //做型別斷言，並檢查是否轉換成功  
        fmt.Println(strconv.Atoi(s))  
    } else {  
        fmt.Println("Type assertion failed")  
    }  
}
```

執行結果：

```
Type assertion failed
```


型別switch

- 你也可能遇上一種情況，就是空白型別背後的實質型別會有很多種，但你無法事先知道有甚麼
- 為了避免寫下多充型別斷言，型別**switch** 就派上用場了
- 同樣來回顧一下語法：

```
switch v:= i.(type) {  
case S:      //v是型別S時執行case底下的程式碼  
}
```

- 型別**switch**後面的語法和型別斷言十分類似，差別在把型別換成關鍵字**type**而已
- 型別斷言會比對每個**case**後面的型別，尋找吻合的對象：

```
package main

import "fmt"

type cat struct {
    name string
}

func main() {
    //建立一個空介面切片，放入不同型別的值
    i := []interface{}{42, "The book club", true, cat{name: "oreo"}}
    typeExample(i)
}

func typeExample(i []interface{}) {
    for _, x := range i {
        switch v := x.(type) { //對切片每個值做型別 switch
        case int:
            fmt.Printf("%v is int\n", v)
        case string:
            fmt.Printf("%v is a string\n", v)
        case bool:
            fmt.Printf("%v is a bool\n", v)
        default:
            fmt.Printf("%T is unknown type\n", v)
        }
    }
}
```

執行結果：

```
42 is int  
The book club is a string  
true is a bool  
main.cat is unknown type
```

練習：分析空介面的資料

- 現在我們會拿到一個map, 其索引鍵式字串, 對應值是interface{}空介面
- 我們的任務是找出每一個鍵對應元素的型別, 並把每個元素包裝成結構record
- record會用不同的欄位來記錄原始的鍵與值, 並有個字串藍未來紀錄該資料型別

```
1  package main
2
3  import (
4      "fmt"
5  )
6
7  type person struct { //自訂結構
8      lastName string
9      age      int
10     isMarried bool
11 }
12
13 type animal struct { //自訂結構
14     name      string
15     category string
16 }
17
18 type record struct { //用來整理map元素的結構
19     key      string
20     data      interface{}
21     valueType string
22 }
23
```

```
24  ✓ func main() {
25      m := make(map[string]interface{}) //建立並初始化一個map
26
27      //在map內加入不同型別的多筆資料
28      m["person"] = person{lastName: "Doe", isMarried: false, age: 19}
29      m["firstname"] = "Smith"
30      m["age"] = 54
31      m["isMarried"] = true
32      m["animal"] = animal{name: "oreo", category: "cat"}
33
34      //解析map的每個元素，轉換成record結構和范進一個切片
35      rs := []record{}
36  ✓  for k, v := range m {
37      |      rs = append(rs, newRecord(k, v))
38      |  }
39
40      //印出record結構切片的内容
41  ✓  for _, v := range rs {
42      |      fmt.Println("Key: ", v.key)
43      |      fmt.Println("Data: ", v.data)
44      |      fmt.Println("Type: ", v.valueType)
45      |      fmt.Println()
46      |  }
47  }
48
```

```
49 //處理map資料並輸出程結構
50 func newRecord(key string, i interface{}) record {
51     r := record{}
52     r.key = key
53     switch v := i.(type) {
54     case int:
55         r.valueType = "int"
56         r.data = v
57         return r
58     case bool:
59         r.valueType = "bool"
60         r.data = v
61         return r
62     case string:
63         r.valueType = "string"
64         r.data = v
65         return r
66     case person:
67         r.valueType = "person"
68         r.data = v
69         return r
70     default:
71         r.valueType = "unknown"
72         r.data = v
73         return r
74     }
75 }
```


執行結果：

```
Key:  person  
Data:  {Doe 19 false}  
Type:  person
```

```
Key:  firstname  
Data:  Smith  
Type:  string
```

```
Key:  age  
Data:  54  
Type:  int
```

```
Key:  isMarried  
Data:  true  
Type:  bool
```

```
Key:  animal  
Data:  {oreo cat}  
Type:  unknown
```

- 注意程式中的型別switch並沒有設立case來辨識animal結構型別，使得animal值在輸出之後的型別欄位為unknown
- 這裡的switch只是為了展示其用途，實際做的事情並不多
 - 若你是要單純用字串紀錄某個值的型別，也許可以這樣寫：

```
r.valueType = fmt.Sprintf("%T", i)
```

- `fmt.Sprintf()`的作用類似於`fmt.Printf()`，但不會印出字串到主控台，而是會傳回格式化後的字串

本章結束