CH11 編碼/解碼JSON資料

11-1 前言

- 在前面各章，我們學習了各種go語言的資料型別，但go語言要處理的資料不見得都來自程式本身，也有可能跟外界交換資料

- 最常見的資料交換格式之一，就是所謂的JSON

JSON格式的資料

- JSON(JavaScript Object Notation, JavaScript物件表示法) 儘管起源於JavaScript, 如今已經被許多程式語言用來儲存和交換資料，事實上很常用於HTTP伺服器和客戶端之間的通訊，也有靜態網站會拿JSON來產生網頁；或者像是NoSQL伺服器等技術也採用JSON作為儲存格式

- JSON是一種與任何程式語言無關的純文字格式，其設計宗旨為精簡至上，且`帶描述資訊，提高了JSON格式的可讀性並降低了撰寫難度

- JSON格式具備了以下特性：
  - 輕量 (lightweight)
  - 和程式語言無關 (programming language-agnostic)
  - 自我描述 (self-describing)
  - 使用鍵值對 (Key/value pairs)

- 諸如RESTful API這類的網路服務，之所以會用JSON而非XML作為資料交換格式，就是因為JSON較為簡潔明瞭，且容易閱讀：

- JSON:

```
{
   "firstname" : "Michael",
   "lastname" : "Jackson"
}
```

- XML:

```
<kingOfpop>
   <firstname>Michael</firstname>
   <lastname>Jackson</lastname>
</kingOfpop>
```

- 上面的JSON資料有兩對鍵與值；鍵一定是用雙引號括起來的字串，但值可以是多種資料型別

- 鍵與值之間以冒號連接，若鍵值對不只一組，各組間以逗號隔開

- JSON得值也可以是陣列，以中括號表示：

```
    }
        "phonenumbers": ["123-123-111", "123-123-2222"]
    }
```

- 以下是JSON可用的值型別 :
  - string , 例如 {“firstname”:”Michael”}
  - number, 例如 {“age”:”33”}
  - Boolean, 例如 {“ismarried”:true}
  - array, 例如 {“hobbies”:[“basketball”, “movie”]}
  - null(空值), 例如 {“middlename”:null}
  - object(JSON物件), 另外一筆JSON資料

11-2 解碼JSON為go結構

- 在這本書中，我們談到解碼JSON時，其實是指將JSON資料轉換成go的資料型別

- Go語言會自動將JSON值轉成對應的go語言型別，讓我們得以用go語言的方式處理

- 若我們事先知道JSON資料包含哪些鍵，就可以解析(unmarshal)它，再把結果存在一個對應的go結構中，而這得用到go標準函示庫的encoding/json的Unmarshal()函式

11-2-1 Unmarshal()

func Unmarshal(data []byte, v interface{}) error

- 參數data就是儲存JSON資料的字串，而v是用來儲存解析結果的變數，我們下面會傳入一個結構指標
- Unmarshal()會解析JSON字串，並試著將結果存到該結構中；v不能為nil, 否則會傳回錯誤
- 為了展示json.Unmarshal()如何將JSON轉成結構，來看一個例子：

```go
package main

import (
    "encoding/json"
    "fmt"
)

type greeting struct { //用來儲存JSON資料的結構
    Message string
}

func main() {
    //JSON資料
    data := []byte(`
{
    "message": "Greetings fellow gopher!"
}
`)
    var v greeting                    //建立一個空結構
    err := json.Unmarshal(data, &v) //解析JSON資料和寫入v
    if err != nil {
        fmt.Println(err)
    }
    fmt.Println(v)
}
```

執行結果：

```
{Greetings fellow gopher!}
```

- 可以發現 , Unmarshal()將JSON的鍵message對應到結構欄位 Message, 並將值放進去

- 注意 :
  - 結構欄位必須是可匯出的(exportable), 也就是開頭要用大寫 , 才能被 Unmarshal()使用

11-2-2 加上結構JSON標籤

- 若想更進一步，我們可以給結構欄位加上標籤(tag), 讓Unmarshal()知道欄位要怎麼用在JSON解碼

- 標籤必須用原始字串(用`括住)寫在欄位後面：

```
type person struct {
    LastName string `json:”lname”`
}
```

- 這個標籤json的值為“lname”, 意思是LastName欄位要對應到JSON資料的lname鍵，這樣我們就能隨意給結構欄位命名了，只要它有匯出就好

- Unmarshal()會根據以下規則決定要把JSON的鍵配對到哪個結構欄位：

  - 某個可匯出欄位的標籤值可以對應到JSON鍵

  - 某個可匯出欄位本身的名稱有對應到JSON鍵(大小寫可不同)

  - 若找不到符合的欄位該鍵就會被略過 (不會放進人和結構欄位)

- 現在借用標籤的特點修改前面的例子：

```go
package main

import (
    "encoding/json"
    "fmt"
    "os"
)

type greeting struct {
    SomeMessage string `json:"message"`
}

func main() {
    //JSON資料
    data := []byte(`
{
    "message": "Greetings fellow gopher!"
}
`)
```

```
20      if !json.Valid(data) {  //檢查JSON格式是否正確
21          fmt.Printf("JSON 格式無效: %s", data)
22          os.Exit(1)
23      }
24
25      v := greeting{}
26      err := json.Unmarshal(data, &v)
27      if err != nil {
28          fmt.Println(err)
29      }
30      fmt.Println(v)
31  }
```

- 注意到這次我們另外使用json.Valid()來檢查JSON是否有效，是的話傳回true, 反之傳回false

- 執行結果：

```
{Greetings fellow gopher!}
```

11-2-3 解碼JSON到複合結構

- 現在看看以下的JSON資料，你認為要用甚麼go結構才能儲存它？

```
{
    “lname”:”Smith”,
    “fname”:”John”,
    “address”:{
            “street”:”Sulphur  Springs  Rd”,
            “city”:”Park  City”
            “zipcode”:”12345”
    }
}
```

- 上面的JSON資料中，鍵address的值是另一個JSON物件，這表示我們得同樣使用雙層的go結構

- 以下程式碼會將以上的JSON物件解析成go語言結構：

```go
package main

import (
    "encoding/json"
    "fmt"
)

type person struct { //父結構
    Lastname  string  `json:"lname"`
    Firstname string  `json:"fname"`
    Address   address `json:"address"` //子結構欄位型別
}

type address struct { //子結構
    Street  string `json:"street"`
    City    string `json:"city"`
    State   string `json:"state"`
    ZipCode int    `json:"zipcode"`
}

```

```
21  func main() {
22      //JSON資料
23      data := []byte(`
24  {
25      "lname":"Smith",
26      "fname":"John",
27      "address":{
28          "street":"Sulphur Springs Rd",
29          "city":"Park City",
30          "state":"VA",
31          "zipcode":12345
32      }
33  }
34  `)
35      //解析JSON並將值存入結構
36      p := person{}
37      if err := json.Unmarshal(data, &p); err != nil {
38          fmt.Println(err)
39      }
40      fmt.Printf("%+v", p)
41  }
```

執行結果：

{Lastname:Smith Firstname:John Address:{Street:Sulphur Springs Rd City:Park City State:VA ZipCode:12345}}

練習：解碼學生課程JSON資料

- 現在我們有個大學選課網站，會接收JSON資料來取得學生提交的資訊和他們選擇的課程

- 一份典型的JSON資料如下：

```
{
        “id”:123,
        “lname”:”Smith”,
        “minitial”:”null”,
        “fname”:”John”,
        “enrolled”:true,
        “classes”:[
                {
                        “coursename”:”Intro to Golang”,
                        “coursenum”:101,
                        “coursehours”:4
                },
                {
                        “coursename”:”English Lit”,
                        “coursenum”:101,
                        “coursehours”:3
                },
```

```
                {
                        “coursename”:”World History”,
                        “coursenum”:101,
                        “coursehours”:3
                }
        ]
}
```

- 注意鍵classes下面是陣列, 當中每個元素都是JSON陣列
- 為了讓網站能進一步處理 , 這些JSON資料都須先轉成go結構
- 以下是完整程式碼 :

```go
package main

import (
    "encoding/json"
    "fmt"
)

type student struct {
    StudentId     int      `json:"id"`
    LastName      string   `json:"lname"`
    MiddleInitial string   `json:"minitial"`
    FirstName     string   `json:"fname"`
    IsEnrolled    bool     `json:"enrolled"`
    Courses       []course `json:"classes"`
}

type course struct {
    Name   string `json:"coursename"`
    Number int    `json:"coursenum"`
    Hours  int    `json:"coursehours"`
}

```

```
23  func main() {
24      data := []byte(`
25  {
26      "id":123,
27      "lname":"Smith",
28      "minitial":null,
29      "fname":"John",
30      "enrolled":true,
31      "classes":[
32          {
33              "coursename":"Intro to Golang",
34              "coursenum":101,
35              "coursehours":4
36          },
37          {
38              "coursename":"English Lit",
39              "coursenum":101,
40              "coursehours":3
41          },
42          {
43              "coursename":"World History",
44              "coursenum":101,
45              "coursehours":3
46          }
47      ]
48  }
49  `)
```

```
50      s := student{}
51      if err := json.Unmarshal(data, &s); err != nil {
52          fmt.Println(err)
53      }
54      fmt.Println(s)
55  }
```

執行結果：

{123 Smith John true [{Intro to Golang 101 4} {English Lit 101 3} {World History 101 3}]}

11-3 將go結構編碼為JSON

- 上面學到如何將JSON資料解碼為結構，現在則要反過來，將儲存在結構力的資料編碼成JSON格式

- 會這麼做的典型場合之一，是將資料從檔案 / 資料庫讀出來和轉成JSON格式，以便透過網路傳給請求者，或者為了將資料寫入NoSQL資料庫，得先將它轉為JSON才行

11-3-1 Marshal()

- 我們要用encoding/json套件的Marshal()函式來達到這個目的 :

```
func Marshal(v interface{}) ([]byte, error)
```

- 參數v是需要編碼成JSON格式的原始資料 , 通常是結構

- Marshal()會傳回JSON字串([]byte切片)以及error值 , 如果編碼失敗error就不會是 nil

- 我們來看簡單例子 , 你會發現Marshal()的運作方式其實就是Unmarshal()的相反

```go
package main

import (
    "encoding/json"
    "fmt"
)

type greeting struct {
    SomeMessage string
}

func main() {
    //包含原始資料的結構
    var v greeting
    v.SomeMessage = "Marshal me!"

    //編碼成JSON格式資料
    json, err := json.Marshal(v)
    if err != nil {
        fmt.Println(err)
    }
    fmt.Printf("%s", json)
}
```

執行結果：

```
{"SomeMessage":"Marshal me!"}
```

- 注意到Marshal()將欄位名稱轉成了JSON鍵，其值則是原本欄位內的字串

- Marshal()在解析結構時，會遵守以下規則來產生JSON鍵值對：

  - 只有可匯出欄位(大寫字母開頭)才能被加入為JSON鍵
  - 帶有JSON標籤的欄位才會被加入，其它的則忽略
  - 若結構只有一個欄位，不管有無JSON標籤都會被加入
  - 若結構有多重欄位，但都沒有JSON標籤，那會被全數忽略(不會產生錯誤)

11-3-2 將有多重欄位的結構轉為JSON

- 如上所述，若想轉換成JSON資料的結構包含多重欄位，那麼他們得加上JSON標籤才行

- 在下面的例子中，我們並沒有針對所有欄位賦值(保持為空值)，這樣會對產生出來的JSON字串造成甚麼影響?

```go
package main

import (
    "encoding/json"
    "fmt"
)

You, 28 秒前 | 1 author (You)
type book struct {
    ISBN          string `json:"isbn"`
    Title         string `json:"title"`
    YearPublished int    `json:"yearpub"`
    Author        string `json:"author"`
    CoAuthor      string `json:"coauthor"`
}

func main() {
    b := book{}
    b.ISBN = "9933HIST"
    b.Title = "Greatest of all Books"
    b.Author = "John Adams"        You, 16 小時前 • 22/8/14
    //沒有對YearPublished和CoAuthor賦值

    json, err := json.Marshal(b)
    if err != nil {
        fmt.Println(err)
    }
    fmt.Println(string(json))
}
```

執行結果：

{"isbn":"9933HIST","title":"Greatest of all Books","yearpub":0,"author":"John Adams","coauthor":""}

- 可以發現未賦值欄位仍會被轉成鍵值對放入JSON資料，其值維持go語言的零值

11-3-3 略過欄位

- 有時你可能希望某些結構的某些欄位不要被編碼進JSON格式中，這時可以在JSON標籤加入omitempty (略過零值)，讓零值欄位被Marshal()忽略

- 底下修改前面的範例：

```
 8  type book struct {
 9      ISBN          string `json:"isbn"`
10      Title         string `json:"title"`
11      YearPublished int    `json:"yearpub,omitempty"`
12      Author        string `json:"author"`
13      CoAuthor      string `json:"coauthor,omitempty"`
14  }
15
```

執行結果

{"Isbn":"9933HIST","title":"Greatest of all Books","author":"John Adams"}

注意：

- omitempty和前面的JSON鍵名稱以逗號格該，而且不能帶額外空格，要是你寫成這樣：

```
YearPublished int     `json:"yearpub, omitempty"`
```

執行時還是會以零值加入

其他標籤的效果

- 若把book結構的JSON標籤改成如下：

```
type book struct {
    ISBN          string `json:"-"` //短折線，代表直接略過
    Title         string `json:"title"`
    YearPublished int    `json:"yearpub,omitempty"`
    Author        string `json:""`            //沒有鍵名稱，代表加入欄位並沿用欄位名稱
    CoAuthor      string `json:",omitempty"` //沒有鍵名稱
}
```

- 這次main()中會賦予以下的值給該結構變數：

```go
func main() {
    b := book{}
    b.ISBN = "9933HIST" //由於已指定略過，不會出現在JSON中
    b.Title = "Greatest of all Books"
    b.YearPublished = 2020
    b.Author = "John Adams"
    //沒對CoAuthur賦值，所以會因為omitempty的原因被略過
```

- 執行結果：

{"title":"Greatest of all Books","yearpub":2020,"Author":"John Adams"}

11-3-4 有排版的JSON編碼效果

- 前面的範例中 , Marshal()產生的JSON字串通通擠在一起 , 不太好閱讀

- 這時可以使用MarshalIndent()函式 , 它的作用與Marshal()幾乎一樣 , 只差結果會縮排和換行 :

```
func MarshalIndent(v interface{}, prefix, indent string) ([]byte, error)
```

  - 參數prefix是要放在每一行開頭的前綴詞 , 這裡暫時不會使用
  - 參數indent則是縮排文字 , 例如幾個空格會其他字元

- 以下範例中，我們要從一個複合結構產生JSON格式字串，但輸出無排版和有排版的版本來比較

- 有排版的JSON會用\t (tab) 作為縮排文字：

```go
package main

import (
    "encoding/json"
    "fmt"
)

type person struct {
    LastName  string  `json:"lname"`
    FirstName string  `json:"fname"`
    Address   address `json:"address"`
}

type address struct {
    Street  string `json:"street"`
    City    string `json:"city"`
    State   string `json:"state"`
    ZipCode int    `json:"zipcode"`
}

```

```go
func main() {
    //建立用來編碼JSON的資料結構
    addr := address{
        Street:  "Galaxy Far Away",
        City:    "Dark Side",
        State:   "Tatooine",
        ZipCode: 12345,
    }
    p := person{
        LastName:  "Vader",
        FirstName: "Darth",
        Address:   addr,
    }

```

```
35      //不排版
36      noPrettyPrint, err := json.Marshal(p)
37      if err != nil {
38          fmt.Println(err)
39      }
40      fmt.Println(string(noPrettyPrint))
41      fmt.Println()
42
43      //有排版
44      prettyPrint, err := json.MarshalIndent(p, "", "\t")
45      if err != nil {
46          fmt.Println(err)
47      }
48      fmt.Println(string(prettyPrint))
49  }
50
```

執行結果：


```
{"lname":"Vader","fname":"Darth","address":{"street":"Galaxy Far Away","city":"Dark Side","state":"Tatooine","zipcod
e":12345}}

{
        "lname": "Vader",
        "fname": "Darth",
        "address": {
                "street": "Galaxy Far Away",
                "city": "Dark Side",
                "state": "Tatooine",
                "zipcode": 12345
        }
}
```

練習：產生學生課程的JSON資料

- 這次要做與 “練習：解碼學生課程JSON資料” 相反的事，也就是用go語言產生學生的選課資料，然後轉成JSON格式傳給學生

- 本練習會用MarshalIndent()產生兩位學生的JSON選課資料，以便展示各欄位在不同的標籤底下會被如何轉成JSON鍵與值：

```go
package main

import (
    "encoding/json"
    "fmt"
    "os"
)

type student struct {
    StudentId     int      `json:"id"`
    LastName      string   `json:"lname"`
    MiddleInitial string   `json:"mname,omitempty"`
    FirstName     string   `json:"fname"`
    IsEnrolled    bool     `json:"enrolled"`
    Courses       []course `json:"classes,omitempty"`
}

type course struct {
    Name   string `json:"coursename"`
    Number int    `json:"coursenum"`
    Hours  int    `json:"coursehours"`
}

```

```go
func main() {
    //第一位學生的資料
    s := student{
        StudentId:     1,
        LastName:      "Williams",
        MiddleInitial: "s",
        FirstName:     "Felicia",
        IsEnrolled:    false,
    }
    //這個學生沒有課程資料, Courses欄位會被略過

    //編碼成JSON時縮排4個空格
    student1, err := json.MarshalIndent(s, "", "    ")
    if err != nil {
        fmt.Println(err)
        os.Exit(1)
    }
    fmt.Println(string(student1))
    fmt.Println()

```

```
44      //第二位學生的資料
45      s2 := student{
46          StudentId:  2,
47          LastName:   "Washington",
48          FirstName:  "Bill",
49          IsEnrolled: true,
50      }
51
52      //第二位學生的選課資料
53      c := course{Name: "World Lit", Number: 101, Hours: 3}
54      s2.Courses = append(s2.Courses, c)
55      c = course{Name: "Biology", Number: 201, Hours: 4}
56      s2.Courses = append(s2.Courses, c)
57      c = course{Name: "Intro to Go", Number: 101, Hours: 4}
58      s2.Courses = append(s2.Courses, c)
59
60      student2, err := json.MarshalIndent(s2, "", "    ")
61      if err != nil {
62          fmt.Println(err)
63          os.Exit(1)
64      }
65      fmt.Println(string(student2))
66  }
```

執行結果：


```
{
    "id": 1,
    "lname": "Williams",
    "mname": "s",
    "fname": "Felicia",
    "enrolled": false
}

{
    "id": 2,
    "lname": "Washington",
    "fname": "Bill",
    "enrolled": true,
    "classes": [
        {
            "coursename": "World Lit",
            "coursenum": 101,
            "coursehours": 3
        },
        {
            "coursename": "Biology",
            "coursenum": 201,
            "coursehours": 4
        },
        {
            "coursename": "Intro to Go",
            "coursenum": 101,
            "coursehours": 4
        }
    ]
}
```

11-4 使用Decoder/Encoder處理JSON資料

- 在第七章談到介面時，就有範例使用json.NewDecoder()函式來解碼JSON資料
- 當時你可能有發現，NewDecoder()能接收幾種不同的資料來源，只要他們符合io.Reader介面的規範即可
- 事實上json套件還有NewEncoder(), 能將編碼好的JSON字串寫入符合io.Writer介面的物件
- 來看看這兩個函式的定義：

```
func NewDecoder(r io.Reader) *Decoder
func NewEncoder(r io.Writer) *Encoder
```

- 這兩個函式會分別傳回json.Decoder和json.Encoder結構指標，而這兩個指標結構則各自擁有用來解碼和編碼JSON的方法：

```
func  (dec  *Decoder)  Decode(v  interface{})  error //等同於Unmarshal()

func  (enc  *Encoder)   Encode(v  interface{})  error //等同於Marshal()
```

- 和Unmarshal() / Marshal() 不同的是，Decoder的資料來源是io.Reader介面物件，而Encoder會把編碼後的字串寫入io.Writer介面物件

- 下面就是一個簡單的例子：

```go
package main

import (
    "encoding/json"
    "fmt"
    "os"
    "strings"
)

type person struct {
    Lastname  string  `json:"lname"`
    Firstname string  `json:"fname"`
    Address   address `json:"address"`
}

type address struct {
    Street  string `json:"street"`
    City    string `json:"city"`
    State   string `json:"state"`
    ZipCode int    `json:"zipcode"`
}

```

```
23  func main() {
24      data := []byte(`
25  {
26      "lname":"Smith",
27      "fname":"John",
28      "address":{
29          "street":"Sulphur Springs Rd",
30          "city":"Park City",
31          "state":"VA",
32          "zipcode":12345
33      }
34  }
35  `)
36      dataStr := string(data)
37      p := person{}
38
39      //用strings.NewReader()從字串建立一個io.Reader
40      //並以此建立json.Decoder
41      decoder := json.NewDecoder(strings.NewReader(dataStr))
42      if err := decoder.Decode(&p); err != nil {
43          fmt.Println(err)
44          os.Exit(1)
45      }
46      fmt.Println(p)
47      fmt.Println()
48
```

```
49      //建立json.Encoder, 寫入對象是os.Stdout (主控台)
50      encoder := json.NewEncoder(os.Stdout)
51      //設定前綴詞和縮排文字
52      encoder.SetIndent("", "\t")
53      //將結構p編碼成JSON
54      if err := encoder.Encode(p); err != nil {
55          fmt.Println(err)
56          os.Exit(1)
57      }
58  }
59
```

執行結果：

```
{Smith John {Sulphur Springs Rd Park City VA 12345}}

{
        "lname": "Smith",
        "fname": "John",
        "address": {
                "street": "Sulphur Springs Rd",
                "city": "Park City",
                "state": "VA",
                "zipcode": 12345
        }
}
```

- 在go語言的標準函式庫中，經常可以看到io.Reader和io.Writer

- 比如你會看到以下物件，雖然用途各有不同，但是都可以搭配json套件的Decoder/Encoder：

類型	io.Reader	io.Writer
字串	strings.Reader	無
主控台	os.Stdin	os.Stdout
檔案	os.File	os.File
HTTP請求/回應	http.Request.Body	http.Response.Body

11-5 處理內容未知的JSON資料

11-5-1 將JSON格式解碼成map

- 若我們事先曉得JSON資料是怎樣組成的，就能定義對應的go結構以便解碼時承接各個鍵值對，或反過來用結構編碼成JSON格式

- 問題是，有時我們無法預知JSON的實際結果，比如某個網路API會產生動態的JSON回應，在不同情況下會有不同的鍵與值

- 幸好json.Unmarshal()不只能將JSON解碼到結構而已，他也能用map來儲存資料

- 更精確地說，這個map可定義成：

```
map[string]interface{}
```

JSON 鍵　　JSON值

- Unmarshal()會將JSON資料中的任何鍵轉成map鍵，並將值配對給對應的鍵

- JSON的鍵一定是字串，值則有可能是不同型別，所以要用空介面接收，如此一來不管JSON資料有甚麼東西，都可以放進map中

- 看看以下的範例：

```go
package main

import (
    "encoding/json"
    "fmt"
)

func main() {
    //原始資料
    jsonData := []byte(`{"checkNum":123,"amount":200,"category":["gift","clothing"]}`)
    //定義map
    var v map[string]interface{}

    //將JSON資料解碼到map
    json.Unmarshal(jsonData, &v)

    //印出map的內容
    fmt.Println(v)
    for key, value := range v {
        fmt.Println(key, "=", value)
    }
}

```

- 注意到我們並沒有初始化map變數v, 因為Unmarshal()自己會做初始化

- 若v在傳入Unmarshal()之前就已經有內容 , 那Unmarshal()會在v中新增其他鍵與值

- 以下是執行結果 : (for range走訪map時會照隨機順序走訪 , 所以你的輸出結果可能會不同)

```
map[amount:200 category:[gift clothing] checkNum:123]
amount = 200
category = [gift clothing]
checkNum = 123
```

練習：分析選課JSON資料內容

- 延續之前的練習，現在假設選課系統轉換一些舊版網站留下的JSON資料，但由於當時未留下說明文件，因此你並不清楚資料內容

- 我們得寫一支程式，分析未知的JSON資料和存入一個map, 接著走訪它，用型別斷言調查每個值的型別：

```go
package main

import (
    "encoding/json"
    "fmt"
    "os"
)

func main() {
    jsonData := []byte(`
{
    "id": 2,
    "lname": "Washington",
    "fname": "Bill",
    "IsEnrolled": true,
    "grades":[100,76,93,50],
    "class":
        {
            "coursename": "World Lit",
            "coursenum": 101,
            "coursehours": 3
        }
}
`)

    if !json.Valid(jsonData) { //先檢查資料是否符合JSON格式
        fmt.Println("JSON 格式不合法:", jsonData)
        os.Exit(1)
    }

```

```go
    //解碼JSON格式到map
    var v map[string]interface{}
    if err := json.Unmarshal(jsonData, &v); err != nil {
        fmt.Println(err)
        os.Exit(1)
    }

    //走訪map
    for key, value := range v {
        fmt.Printf("%s = %v (%s)\n", key, value, findTypeName(value))
    }
}

//用型別斷言來檢查值的函式
func findTypeName(i interface{}) string {
    switch i.(type) {
    case string:
        return "string"
    case int:
        return "int"
    case float64:
        return "float64"
    case bool:
        return "bool"
    default:
        return fmt.Sprintf("%T", i)
    }
}

```

執行結果：

```
id = 2 (float64)
lname = Washington (string)
fname = Bill (string)
IsEnrolled = true (bool)
grades = [100 76 93 50] ([]interface {})
class = map[coursehours:3 coursename:World Lit coursenum:101] (map[string]interface {})
```

11-5-2 將map編碼成JSON格式

- 同理，你也可以利用map來提供原始資料，讓json.Marshal()或MarshalIndent()產生JSON格式字串

- 這些函式會將map鍵轉成JSON鍵，並根據map的元素自動轉成適當的JSON值：

```go
package main

import (
    "encoding/json"
    "fmt"
)

func main() {
    v := make(map[string]interface{}) //初始化map
    v["checkNum"] = 123
    v["amount"] = 200
    v["category"] = []string{"gift", "clothing"}

    //將map編碼成JSON格式
    jsonData, err := json.MarshalIndent(v, "", "\t")
    if err != nil {
        fmt.Println(err)
    }
    fmt.Println(string(jsonData))
}

```

執行結果：


```
{
    "amount": 200,
    "category": [
        "gift",
        "clothing"
    ],
    "checkNum": 123
}
```

11-6 gob: GO自有的編碼格式

- 儘管JSON和XML之類的資料格式四海皆同，這些以純文字為基礎的格式在解讀上仍然偏慢，這對追求高效率的網路通訊來說是個問題

- 要是你的系統完全以go語言撰寫，那就可以使用go語言自己的二進位編碼格式 --- gob

- Go語言設計gob時, 是以高效率/簡易使用和完整為考量 , 不需要額外設定 , 甚至收發雙方使用的go結構也不見得需要相同

- 其實gob套件用起來就和json的Encoder/Decoder很像 :

```
func  NewEncoder(w  io.Writer)  *Encoder
func  NewDecoder(w io.Reader) *Decoder
func  (enc  *Encoder)  Encode(e   interface{})  error
func  (dec  *Decoder)  Decode(e  interface{})  error
```

練習：使用gob編碼和解碼資料

- 在此我們沿用前面各練習題的結構，只是改用gob來編碼和解碼：

```go
package main

import (
    "bytes"
    "encoding/gob"
    "fmt"
    "os"
)

type student struct {
    StudentId     int
    LastName      string
    MiddleInitial string
    FirstName     string
    IsEnrolled    bool
    Courses       []course
}

type course struct {
    Name   string
    Number int
    Hours  int
}

```

```go
func main() {
    s := student{
        StudentId:  2,
        LastName:   "Washington",
        FirstName:  "Bill",
        IsEnrolled: true,
        Courses: []course{
            {Name: "World Lit", Number: 101, Hours: 3},
            {Name: "Biology", Number: 201, Hours: 4},
            {Name: "Intro to Go", Number: 101, Hours: 4},
        },
    }

    var conn bytes.Buffer //模擬通訊用的io.Reader/io.Writer
    encoder := gob.NewEncoder(&conn)//產生encoder
    if err := encoder.Encode(&s); err != nil { //編碼gob
        fmt.Println("GOB 編碼錯誤:", err)
        os.Exit(1)
    }

    fmt.Printf("%x\n", conn.String()) //把conn的內容用16進位形式印出

```

```
47      s2 := student{} //接收解碼後資料的結構
48      decoder := gob.NewDecoder(&conn) //產生decoder
49      if err := decoder.Decode(&s2); err != nil { //解碼gob
50          fmt.Println("GOB 解碼錯誤:", err)
51          os.Exit(1)
52      }
53
54      fmt.Println(s2) //解碼後的資料
55  }
56
```

執行結果：

```
6cff81030101077374756465 6e7401ff8200010601095374756465 6e74496401040001084c6173744e616d65010c00010d4d6964646c65496e69
7469616c010c000109466972737 44e616d65010c00010a4973456e726f6c6c656401020001074 36f757273657301ff860000001cff850201010d
5b5d6d61696e2e636f7572736501ff860001ff84000032ff8303010106636f7572736501ff8400010301044e616d65010c0001064e756d626572
0104000105486f7572730104000000 4fff820104010a57617368696e67746f6e020442696c6c010101030109576f726c64204c697401ffca0106
00010742696f6c6f677901fe0192010800010b496e74726f20746f20476f01ffca01080000
{2 Washington  Bill true [{World Lit 101 3} {Biology 201 4} {Intro to Go 101 4}]}
```

- 在以上程式中，gob的Encoder與Decoder結構共用一個bytes.Buffer結構，後者能同時滿足io.Writer及io.Reader介面的定義

- 這樣做的意義是：若你把它轉換成其他結構，例如用於網路通訊的net.TCPConn, 就可以透過gob來在網路訊息交換中使用二進位編碼，進而提高通訊校略了，或透過io.File將訊息寫入檔案和讀出，比如在伺服器備份收到的訊息，以免系統重啟後遺失資料等等

本章結束