CH6 錯誤處理

6-1 前言

- 前一章中，我們學習了如何建立函式，而我們也提過go函式在回傳值時，最後一個值應該要是error, 這個錯誤值也應該得到妥善處理

- 在本章中，我們要來檢視何謂程式錯誤，go語言中的錯誤又是長甚麼樣，更重要的是，在go語言中要如何處理錯誤

6-2 程式錯誤的類型

- 你會遇到的錯誤包含以下三種類型：

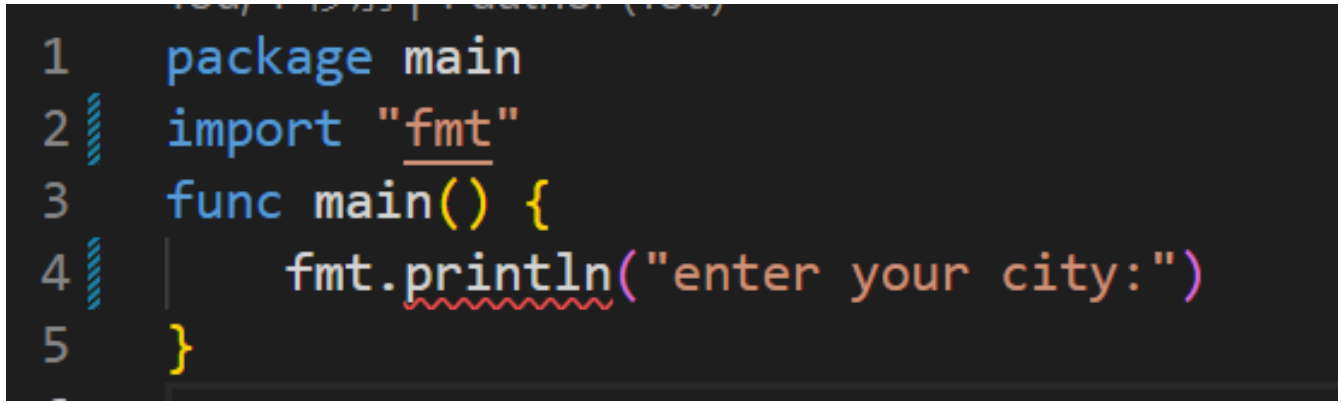
1. 語法錯誤(syntax errors)
2. 值行期間錯誤(runtime errors)
3. 邏輯錯誤(loginc error)或語意錯誤(semantic errors)

6-2-1 語法錯誤

- 此錯誤來自對於程式語言的運用不當，通常是對程式語言不夠熟悉，或者出於疏失而打字錯誤或寫錯語法所致
- 隨著你對程式語言越來越熟練，這種錯誤也會漸漸減少
- 如今大部分編輯器都有能力用視覺化的方式標示出語法錯誤，甚至能在編譯或執行前就抓出一些錯誤

- 常見的語法錯誤像是：
  - 迴圈語法不正確
  - 各種括號放錯地方
  - 寫錯函式或套件名稱
  - 將型別不符的引數傳給函式參數

- 以下就是一個經典的語法錯誤(寫錯套件名稱)



6-2-2 執行期間錯誤

- 這種錯誤來自於程式被要求進行它做不到的動作
- 與語法錯誤不同的是，這種錯誤要等到實際執行時才會出現
- 以下是常見的執行期間錯誤：
  - 連接一個不存在的資料庫
  - 開啟不存在的檔案
  - 以迴圈走訪切片和陣列，但是迴圈索引卻超過集合中的索引範圍
  - 變數的值計算後超過範圍，發生越界繞回
  - 不當的數學運算，例如以0作為分母

練習：加總數字時的執行期間錯誤

```go
package main

import "fmt"

func main() {
    nums := []int{2, 4, 6, 8}
    total := 0
    for i := 0; i <= 10; i++ {
        total += nums[i] //將發生錯誤
    }
    fmt.Println("總和 :", total)
}
```

執行結果：

```
panic: runtime error: index out of range [4] with length 4

goroutine 1 [running]:
main.main()
        d:/git/go lan/ch6/6-2-2.go:9 +0xdf
exit status 2
```

- 程式出錯的原因是發生了index out of range錯誤

- 因為當for迴圈走訪切片時，索引 i 遞增到4時超出了切片的最大所引3,於是引發了panic

- 解法之一是用range來走訪切片：

```go
func main() {
    nums := []int{2, 4, 6, 8}
    total := 0
    for i := range nums {
        total += nums[i]
    }
    fmt.Println("總和 :", total)
}
```

6-2-3 邏輯錯誤/語意錯誤

- 邏輯錯誤(或稱為語意錯誤)是最難被找出來的，也就是程式對資料做了不正確的判斷，有時在第一時間極為難以察覺

- 發生這種錯誤的原因可能包括：

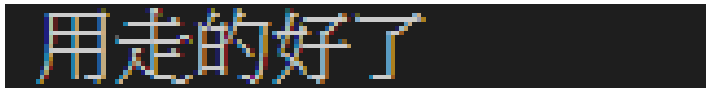
  - 錯誤的計算方式
  - 存取錯誤的資源(檔案/資料庫/伺服器/變數....)
  - 變數邏輯判斷不當

練習：評估步行距離的邏輯錯誤

- 現在要寫一支程式，判斷要步行還是搭車前往目的地
- 若距離目的有2公里以上(包含兩公里)就搭車，否則就走路

```go
package main

import "fmt"

func main() {
    km := 2 //目的地距離
    if km > 2 {
        fmt.Println("搭車吧")
    } else {
        fmt.Println("用走的好了")
    }
}
```

- 執行結果：



- 應該要顯示的是搭車，不過修正方式也很簡單，修改邏輯算符就好:

```go
func main() {
    km := 2 //目的地距離
    if km >= 2 {
        fmt.Println("搭車吧")
    } else {
        fmt.Println("用走的好了")
    }
}
```

6-3 其他程式語言的錯誤處理方式

- Go語言處理錯誤的方式和Java, Python, C#和Ruby都不一樣，這些語言做的是所謂的例外處理

- 以下的程式碼片段示範了其他程式語言例外處理的方式應付錯誤：

Java, C#

```
try {
    // 可能產生錯誤的程式碼
} catch (exception e) {
    // 處理錯誤的程式碼
} finally {
    // 處理完後的程式碼
}
```

Python

```
try:
    # 可能產生錯誤的程式碼
except:
    # 處理錯誤的程式碼
else:
    # 沒有發生錯誤時的程式碼
finally:
    # 處理完後的程式碼
```

Ruby

```
begin
    # 可能產生錯誤的程式碼
rescue =>
    # 處理錯誤的程式碼
else
    # 沒有發生錯誤時的程式碼
ensure
    # 處理完後的程式碼
end
```

- 在大部分程式語言，例外處理是隱性的：任何程式都可能出錯和拋出意外，但無法事先知道哪裡會出問題，只能用try....error來攔截它們，若沒有處裡例外，該函式會導致整個程式當掉

- 然而在go語言，錯誤處理是顯性的

- 許多函式會很明確的回傳一個你無法拒絕的錯誤值，但在該函式成功執行時會是nil

- 就算真的回傳非nil錯誤，函式也不見得會讓程式當掉，但你有責任處理錯誤值當它不為nil時

- 簡單來說，在大部分程式語言中，若某些功能有發生錯誤的可能，就必須用try…catch敘述來包住它

- 但在go語言裡，你會很明確地收到error值，然後自己判斷該它做甚麼事:

```
val, err := someFunc() //呼叫函式, 接收回傳值(包括 error)
If err != nil {
    //若有錯誤存在，做些處理然後繼續往外傳
    return err
}
return nil //沒有錯誤，對上一層回傳nil
```

- Go語言藉由這種方式要求開發者承擔處理錯誤的責任，不僅簡化錯誤檢查的流程，也能讓你對程式碼付出更多關注

6-4 error介面

6-4-1 go語言的error值

- 在go語言中，一個error是一個值

- 既然error是一個值，它就可以當作引數傳給函式/被函式傳回，並且能像任何值一樣被讀取和做比較

- 事實上，go語言的error值都必須實作來符合error介面的定義(第7章會介紹何謂介面實作)，下面是go語言中宣告的error介面型別：

```
type error interface {
    Error() string
}
```

- 在第四章曾說過，一個型別只要符合介面的規範(擁有一樣的方法函式)，就會被視為符合該型別

- 也就是說，任何型別只要符合error介面的要求，就能被當作error型別：
  - 型別得擁有一個函式(或方法)叫Error()
  - Error()得回傳一個string型別的值

- 在go語言標準函式庫中，各套件可能會定義自己的error型別，包含不同的欄位或方法，但只要這些型別具備Error() string方法，就可以用error型別介面的形式建立error值，而且能被許多跟錯誤處理有關的功能共用

- 我們要用以下這段程式碼當作起點，以程式內的錯誤處理部分來說明go語言如何處理錯誤：

```
func main() {
    v := 10
    if err != nil {
        fmt.Println(err)
    }
    fmt.Printf("%T, %v\n", s, s)

    v = "s2"
    s2, err := strconv.Atoi(v)
    if err != nil {
        fmt.Println(err)
    }
    fmt.Printf("%T, %v\n", s2, s)
}
```

- 在第5章時，提過函式可以回傳多個值，這是go語言的優勢之一，這點對於錯誤處理更是深具好處；從上面可以看到，go語言標準函式庫 strconv.Atoi() 函式會接收一個字串，並遵守go語言設計慣例回傳一個int以及一個error值

- 任何函式若要傳回錯誤，error必須是最後一個回傳值

- 若函式傳回error卻置之不理是很糟糕的習慣，因為事後可能要花大量精力除錯

- Error值若是nil時代表沒有錯誤，但若不是nil便代表有錯誤發生，依據不同場合，我們的因應可能如下：

- 將error傳給函式呼叫者
- 用log記錄錯誤並繼續執行
- 停止程式執行
- 忽略error(不建議)
- 引發panic(稍後會提到)

error型別定義

- 我們繼續來研究go語言標準套件中的error型別，首先先從函式庫中的errors.go著手：

```
type errorString struct {
    s  string
}
```

- errorString這個結構型別位於error套件中，有一個字串型別欄位s來儲存錯誤內容

- 注意errorString型別和其欄位s都是英文小寫開頭(第8章會進一步說明)，這代表它們是不可匯出或公開的，亦即不能在外部程式直接使用它們：

- 這樣看起來errorString似乎毫無用處，不過先別急著跳過，errors.go中還有這麼一段：

```
func (e *errorString) Error() string {
    return e.s
}
```

- Error()函式透過指標接收器變成errorString結構的方法，這樣errorString結構的定義就符合error介面的要求，使errorString結構可以被視為error型別

- 若你要查看錯誤內容，只要呼叫其Error()方法和取得回傳字串即可

- 這表示在go語言中，你可以用各種方式定義error值，而它的型別一旦符合error介面，就能用完全一樣的方式操作它們

6-4-3建立error值

- 在error.go裡有一個函式，可以用來建立你自己的error值：

```
func New(text string) error {
    return &errorString{text}
}
```

- New()會接收一個字串引數，並以此產生新的指標變數結構變數*errors.errString, 再以error介面形別傳回

- 雖然說傳回值是error介面，但實際傳回的其實是*errors.errString型別，以下程式碼可以證明這一點：

```go
package main

import (
    "errors"
    "fmt"
)

func main() {
    ErrBadData := errors.New("some bad data")
    fmt.Printf("ErrBadData Type : %T", ErrBadData)
}
```

- 執行結果：

```
ErrBadData Type : *errors.errorString
```

- error.New()非常有用，它能讓你快速產生包含自訂訊息的error值無須自己另外定義一個符合error介面的型別

- 在go語言中，error值的名稱習慣以Err開頭，並採用駝峰式命名

練習：建立一個周薪資計算程式

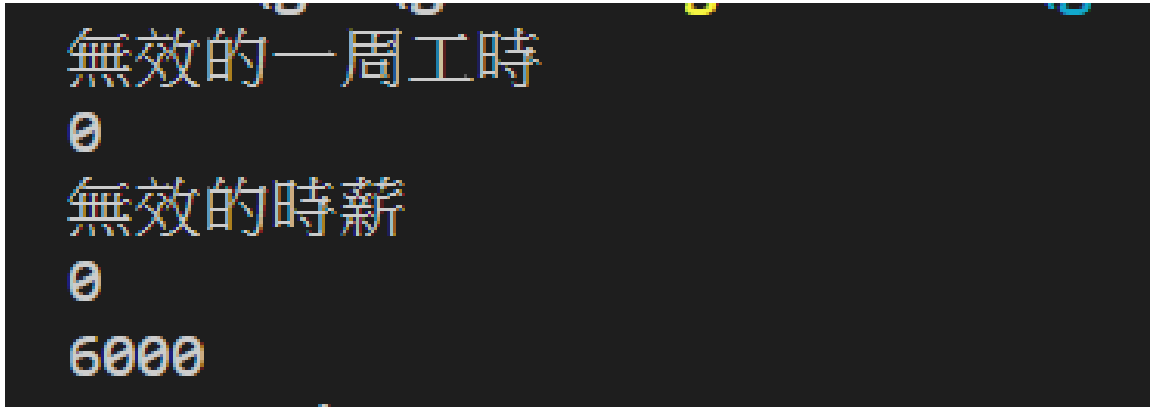
- 這個練習要建立一個函式來計算周薪
- 此函式會接收兩個引數，一個是工作時數，另一個是時薪
- 函式要檢查這兩個參數是否有效，並且得計算加班費：
  - 時薪必須介於10~75美金
  - 一周工時必須介於0~80小時
  - 若工作超過40小時，額外工時的時薪乘以2
  - 若時薪或工時有誤，周薪回傳0，並傳回對應的error值，若沒有錯誤，傳回計算後的周薪，錯誤回傳nil

```go
package main

import (
    "errors"
    "fmt"
)

var ( //自己定義error值
    ErrHourlyRate  = errors.New("無效的時薪")
    ErrHoursWorked = errors.New("無效的一周工時")
)

```

```go
func main() {
    pay, err := payDay(81, 50)
    if err != nil { //若payDay傳回錯誤就印出
        fmt.Println(err)
    }
    fmt.Println(pay)

    pay, err = payDay(80, 5)
    if err != nil { //同上
        fmt.Println(err)
    }
    fmt.Println(pay)

    pay, err = payDay(80, 50)
    if err != nil { //同上
        fmt.Println(err)
    }
    fmt.Println(pay)
}

```

```go
func payDay(hoursWorked, hourlyRate int) (int, error) {
    if hourlyRate < 10 || hourlyRate > 75 {
        return 0, ErrHourlyRate //若時薪不對 , 傳回時薪的error
    }
    if hoursWorked < 0 || hoursWorked > 80 {
        return 0, ErrHoursWorked //若工時不對 , 傳回工時的e的error
    }
    //計算加班費
    if hoursWorked > 40 {
        hoursOver := hoursWorked - 40
        hoursRegular := hoursWorked - hoursOver
        return hoursRegular*hourlyRate + hoursOver*hourlyRate*2, nil
    }
    //沒有錯誤就傳回計算的周薪(含加班費), 錯誤就傳回nil
    return hoursWorked * hourlyRate, nil
}
```

執行結果



6-4-4 使用fmt.Errorf()建立error值

- 前面的練習是先用error.New()建立error值，還有另一個方式是使用fmt.Errorf(), 讓你建立格式化的錯誤訊息：

```
func payDay(hoursWorked, hourlyRate int) (int, error) {
    if hourlyRate < 10 || hourlyRate > 75 {
        return 0, fmt.Errorf("無效的時薪: %d", hourlyRate)
    }
    if hoursWorked < 0 || hoursWorked > 80 {
        return 0, fmt.Errorf("無效的一周工時: %d", hoursWorked)
    }
```

- 這表示我們更可以把其他error值的內容讀出來，連同其他訊息合併成一個新的error值：

```
func payDay(hoursWorked, hourlyRate int) (int, error) {
    if hourlyRate < 10 || hourlyRate > 75 {
        //用error值得Error()取得內容字串 ， 產生新的error值後回傳
        return 0, fmt.Errorf("payDay錯誤: %s", ErrHourlyRate.Error())
    }
    if hoursWorked < 0 || hoursWorked > 80 {
        return 0, fmt.Errorf("payDay錯誤: %s", ErrHoursWorked.Error())
    }
```

- 但這種合併法意味著舊的error值會被新型別蓋掉，而特定的error值可能是有特殊意義的

- 此外，原本的error值也可能擁有額外的欄位 / 方法等等，而這些資訊都會在合併的過程中消失

- 因此fmt.Errorf()提供了另一種結合error值的做法----error值可以包覆其他的error值，辦法是在格式化字串中使用%w來對應到要被包覆的error:

```go
func payDay(hoursWorked, hourlyRate int) (int, error) {
    if hourlyRate < 10 || hourlyRate > 75 {
        //用error值得Error()取得內容字串 , 產生新的error值後回傳
        return 0, fmt.Errorf("無效的時薪: %w",ErrHourlyRate)
    }
    if hoursWorked < 0 || hoursWorked > 80 {
        return 0, fmt.Errorf("無效的一周工時: %w", ErrHoursWorked)
    }
```

- 若仔細觀察error套件的原始碼，會發現fmt.Errof()傳回的error型別實際上為wrapError結構，它有個error型別欄位能記住它包覆的錯誤值，並有額外的方法可以讀取該欄位：

```
type  wrapError struct {
      msg string
      err error
}
// wrapError有實作Error()因此符合error介面型別
func (e *wrapError) Error() string {
      return e.meg
}

func (e *wrapError) Unwrap() error {
      return e.err
}
```

- 當你在fmt.Errof()使用%w來包覆其他error值時，後者會被儲存到新error值(wrapError型別)的err欄位中

- 這也意味當函式接收了error值並往外傳時，可以將錯誤值層層包覆起來，行稱所謂的錯誤鏈

- go語言也提供了兩個新方法，error.Is()和error.As(), 使你能夠檢查錯誤鏈中是否存在某個特定的error型別或某個error值 (有興趣可上網查閱)

6-5 panic

6-5-1 何謂panic

- 如前面所提，很多程式語言會用例外處理錯誤，但go語言會以error值的形式傳回，通常也不會影響程式運作
- 但若遇到真正嚴重的情況，go語言會引發所謂的恐慌(panic)
- 當panic發生時，你會在錯誤訊息看到像是 Goroutine running 的字樣，這是因為main()自己也是一個Goroutine(第16章會再探討)

- 如果發生了panic，代表程式遭遇了完全不正常的狀況

- Go 語言執行環境或開發者之所以引發panic，通常是為了保護程式的完整性，藉由中斷程式來避免造成其他影響

6-5-2 panic()函式

- panic也可以由開發者在程式執行期間觸發，辦法是使用panic()函

- 它接受一個空介面型別的參數，這表示panic()可以接收任何型別的資料，但大部分情況，你應該傳入一個error介面型別的值，這也是go語言的既定慣例

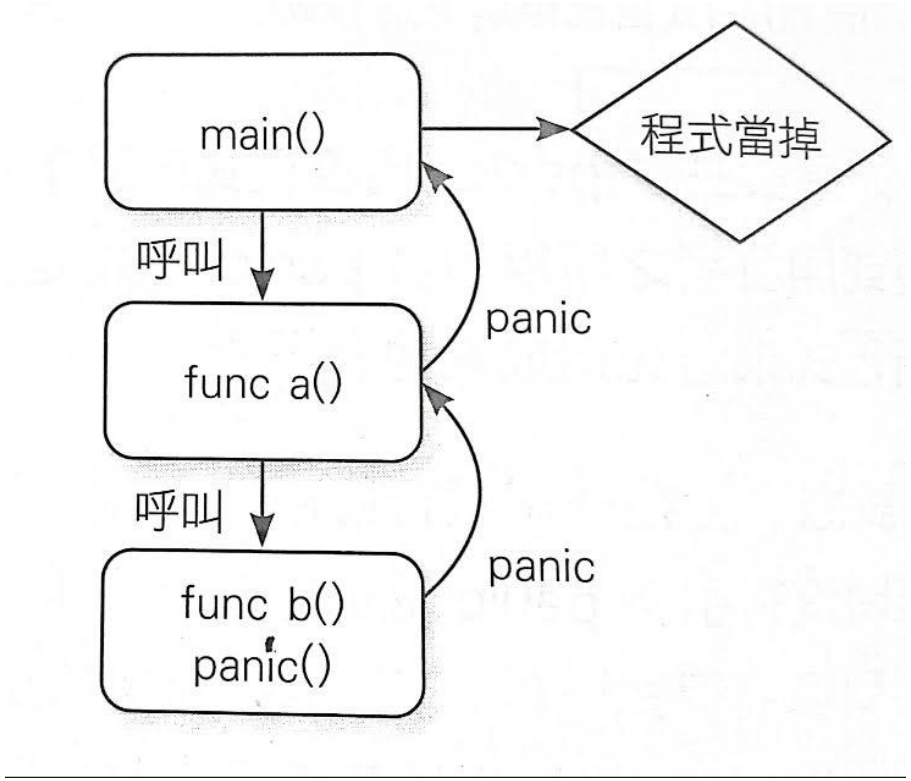
- 對於函式的使用者來說，要是能得知panic為何發生，自然會比較應對，因此我們後面會介紹如何從panic狀況恢復，以及如果panic()收到error值，有哪些不同的救援選項

- 當panic發生時，一般會伴隨以下動作：

1. 停止程式執行
2. 發生panic的函式中若有延後執行的函式(deffered)，它們會被呼叫
3. 發生panic的函式的上層函式中，若有deffered函式會被呼叫
4. 沿著函式堆疊一路往上走，最後抵達main
5. 發生panic的函式之後所有的敘述都不會執行
6. 程式當掉

- panic的運作流程如下：

1. main()呼叫 func a()
2. func a () 呼叫func b()
3. 結果func b()發生panic
4. panic 回依序往上傳，但這些上層函式都沒有處理panic
5. 最終程式當掉



手動引發panic

- 現在回頭看6-2-2練習的錯誤內容：

```go
package main

import "fmt"

func main() {
    nums := []int{2, 4, 6, 8}
    total := 0
    for i := 0; i <= 10; i++ {
        total += nums[i]
    }
    fmt.Println("總和 :", total)
}
```

- 引發的錯誤訊息：

```
panic: runtime error: index out of range [4] with length 4

goroutine 1 [running]:
main.main()
        d:/git/go lan/ch6/6-2-2.go:9 +0xdf
exit status 2
```

- Go語言之所以將這種情況視為panic，是因為我們企圖走訪一個切片，索引範圍卻超過了元素數量，go認為這使程式進入不正常狀況，因而觸發了panic

- 下面的程式碼示範了如何主動使用panic()來讓程式當掉

```go
package main

import (
    "errors"
    "fmt"
)

func main() {
    msg := "good-bye"
    message(msg)
    fmt.Println("這行不會印出")
}

func message(msg string) {
    if msg == "good-bye" {
        panic(errors.New("出事了"))
    }
}
```

執行結果：

```
panic: 出事了

goroutine 1 [running]:
main.message(...)
        d:/git/go lan/ch6/6-5-2.go:16
main.main()
        d:/git/go lan/ch6/6-5-2.go:10 +0x49
exit status 2
```

panic時defer的執行效果

- 在以上程式中，發生了panic之後程式碼就不會執行了
- 不過，我們來看看若在panic後面的函式加上defer敘述會如何：

```go
package main

import (
    "errors"
    "fmt"
)

func main() {
    defer fmt.Println("在 main() 使用 defer")
    test()
    fmt.Println("這一行不會印出")
}

func test() {
    defer fmt.Println("在 test() 使用 defer")
    msg := "good-bye"
    message(msg)
}

func message(msg string) {
    defer fmt.Println("在 message() 使用 defer")
    if msg == "good-bye" {
        panic(errors.New("出事了"))
    }
}
```

執行結果：

```
在 message() 使用 defer
在 test() 使用 defer
在 main() 使用 defer
panic: 出事了

goroutine 1 [running]:
main.message({0xb04dbb?, 0x0?})
        d:/git/go lan/ch6/6-5-2(2).go:23 +0xce
main.test()
        d:/git/go lan/ch6/6-5-2(2).go:17 +0x7c
main.main()
        d:/git/go lan/ch6/6-5-2(2).go:10 +0x70
exit status 2
```

- 我們逐步剖析這段程式碼：

1. 由於panic發生在message(), 該函式內defer 會先被執行
2. 程式沿著函式呼叫堆疊往上 , 上一層函式是test(), 它當中的defer也會被執行
3. 接著來到main()函式 , 其中的defer也會被呼叫
4. 最終程式因為panic中斷

練習：利用panic()讓程式在發生錯誤時當掉

- 現在我們要改寫6-4-3的周薪計算應用程式
- 這回需求有所變更：有人抱怨薪資支票有誤，我們認為是使用者呼叫payDay()時忽略了error所致
- 於是新版的payDay()函式只需要回傳薪資，不必回傳error
- 若傳給函式的引數無效，函式就直接引發panic讓程式當掉

```go
package main

import (
    "errors"
    "fmt"
)

var (
    ErrHourlyRate  = errors.New("無效的時薪")
    ErrHoursWorked = errors.New("無效的一周工時")
)

func main() {
    pay := payDay(81, 50)
    fmt.Println(pay)
}

func payDay(hoursWorked, hourlyRate int) int { //不回傳error
    //不管有沒有引發panic, 在payDay()結束時印出工時與薪資
    report := func() {
        fmt.Printf("工時: %d\n時薪: %d\n", hoursWorked, hourlyRate)
    }
    defer report()

    if hourlyRate < 10 || hourlyRate > 75 {
        panic(ErrHourlyRate)
    }
    if hoursWorked < 0 || hoursWorked > 80 {
        panic(ErrHoursWorked)
    }
    return hoursWorked * hourlyRate
}
```

執行結果：

```
工時: 81
時薪: 50
panic: 無效的一周工時

goroutine 1 [running]:
main.payDay(0x9852f9?, 0x60?)
        d:/git/go lan/ch6/6-5-2(3).go:29 +0xbc
main.main()
        d:/git/go lan/ch6/6-5-2(3).go:14 +0x25
exit status 2
```

6-6 recover (復原)

- panic狀況其實也非法補救 , go語言提供了recover()函式 , 可以在某個Gorutine發生panic後取回控制權

- recover()函式的定義如下 :

```
func  recover() interface{}
```

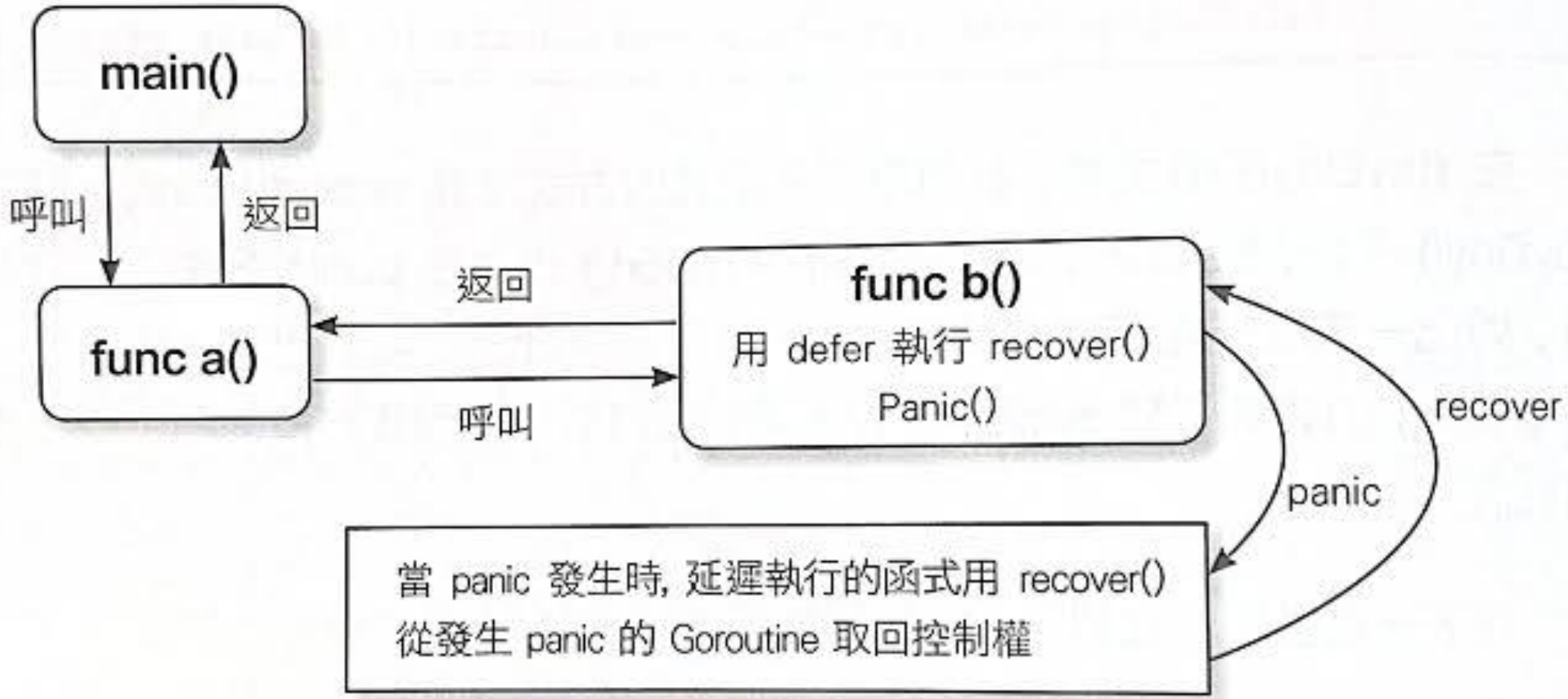
- recover()沒有參數, 回傳值是一個空介面 , 這表示傳回資料可以是任意型別

- 事實上 , recover()傳回的是你一開始傳給panic()函式的值

- recover()只有在使用defer延遲執行的函式中才有作用

- 若你在延遲函式中呼叫recover(), 就可以恢復正常執行並停止panic , 但如果你在延遲函式以外的地方呼叫 , 就無法阻止panic

- 下圖展示了一支程式在使用panic(), recover()和defer時回經歷的過程


main()
呼叫
返回
func a()
返回
呼叫
func b()
用 defer 執行 recover()
Panic()
recover
panic
當 panic 發生時, 延遲執行的函式用 recover()
從發生 panic 的 Goroutine 取回控制權

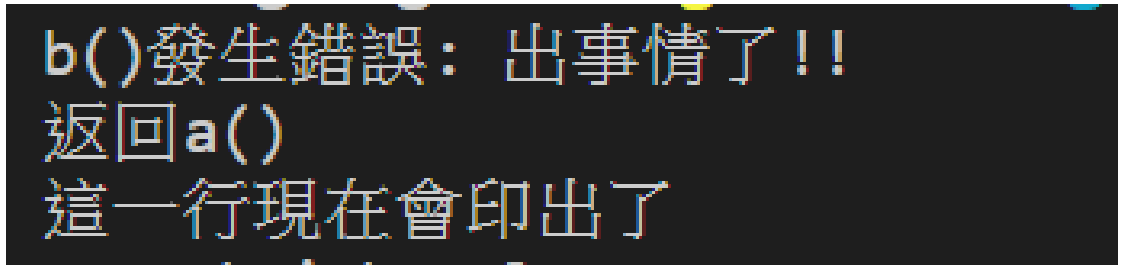
- 上圖流程的說明如下：

1. main呼叫a(), a()再呼叫b()
2. 在b()當中發生panic
3. b()當中一個defer函式在b()結束前被呼叫，並執行了recover()
4. recover()阻止了panic, 使程式流程正常地回到a(), 然後再回到main()

- 下面的程式碼模擬了以上流程：

```go
package main

import (
    "errors"
    "fmt"
)

func main() {
    a()
    fmt.Println("這一行現在會印出了")
}

func a() {
    b("good-bye")
    fmt.Println("返回a()")
}

func b(mesg string) {
    defer func() { //用defer來確保匿名函式在panic發生後執行
        //若有panic發生 , 用recover()救回程式
        if r := recover(); r != nil {
            fmt.Println("b()發生錯誤:", r) //印出error
        }
    }()
    if mesg == "good-bye" {
        panic(errors.New("出事情了!!")) //引發panic
    }
    fmt.Print(mesg)
}
```

執行結果：



- 不管b()有沒有發生panic, 都會呼叫延遲執行的匿名函式 , 而匿名函式會呼叫recover() , 若它傳回的error值是nil , 就代表b()沒有發生panic ; 若不是nil , 則會印出error的值(也就是傳入panic()的錯誤內容)

- 現在recover()就會阻止b()內發生的panic, 因此panic不會往上傳

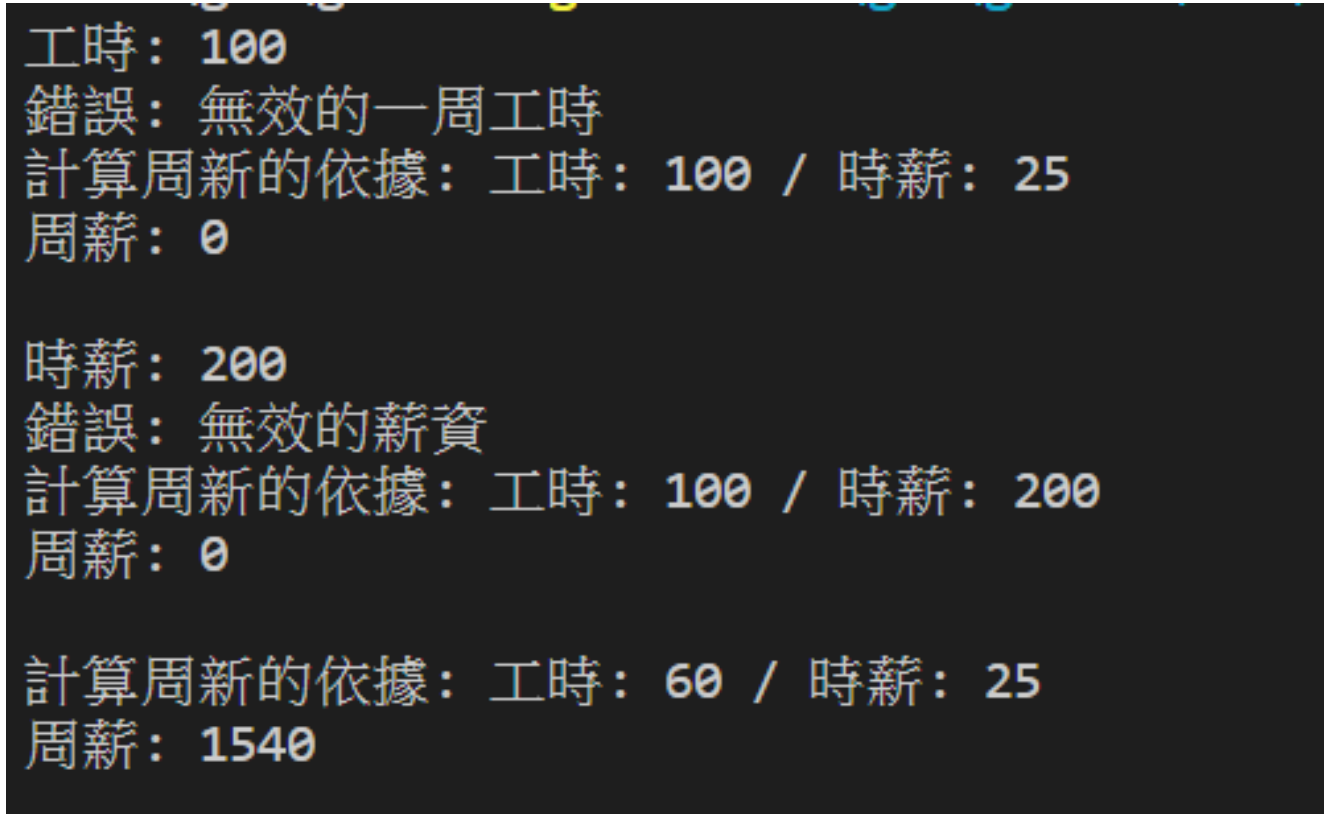
練習：從panic中復原

- 這個練習要繼續改良payDay()函式，讓它從panic中復原，同時我們也要調查造成panic的錯誤內容是甚麼

```go
package main

import (
    "errors"
    "fmt"
)

var (
    ErrHourlyRate   = errors.New("無效的薪資")
    ErrHourlyWorked = errors.New("無效的一周工時")
)

func main() {
    pay := payDay(100, 25)
    fmt.Printf("周薪: %d\n\n", pay)

    pay = payDay(100, 200)
    fmt.Printf("周薪: %d\n\n", pay)

    pay = payDay(60, 25)
    fmt.Printf("周薪: %d\n\n", pay)
}

```

```go
func payDay(hoursWorked, hourlyRate int) int {
    defer func() {
        if r := recover(); r != nil {
            if r == ErrHourlyRate {
                fmt.Printf("時薪: %d\n錯誤: %v\n", hourlyRate, r) //若panic是隨ErrHourlyRate錯誤發生
            }
            if r == ErrHourlyWorked { //若panic是隨ErrHoursworked錯誤發生
                fmt.Printf("工時: %d\n錯誤: %v\n", hoursWorked, r)
            }
        }
        fmt.Printf("計算周新的依據: 工時: %d / 時薪: %d\n", hoursWorked, hourlyRate)
    }()
    if hourlyRate < 10 || hourlyRate > 75 {
        panic(ErrHourlyRate)
    }

    if hoursWorked < 0 || hoursWorked > 80 {
        panic(ErrHourlyWorked)
    }

    if hoursWorked > 40 {
        hoursOver := hoursWorked - 40
        overTime := hoursOver * 2
        regularPay := hoursWorked * hourlyRate
        return regularPay + overTime
    }
    return hoursWorked * hourlyRate
}
```

執行結果：


工時: 100
錯誤: 無效的一周工時
計算周新的依據: 工時: 100 / 時薪: 25
周薪: 0
時薪: 200
錯誤: 無效的薪資
計算周新的依據: 工時: 100 / 時薪: 200
周薪: 0
計算周新的依據: 工時: 60 / 時薪: 25
周薪: 1540

6-7 處理error與panic的指導方針

- 指導方針僅供參考，並非金科玉律：

1. 宣告自訂的err值時，變數命名應以Err開頭，並遵照駝峰式命名
2. error的字串若是英文，應以小寫開頭，結尾也沒有標點符號，這樣做的原因之一是可以讓error跟其他相關資訊合併
3. 若函式會傳回error，呼叫者應檢查其內容是否為nil
4. 使用panic()時，請傳入一個error值作為引數，不要只傳入空介面或空自串
5. 不要拿error中的字串內容來做運算或比對
6. 盡量少用panic()

本章結束