

Swift 起步走

從 Swift 基礎入門到實戰 iPhone App

◆ 適合閱讀對象 ◆

- ✓ 零程式語言設計基礎的初心者
- ✓ 想要加大守備範圍的 Objective-C 老手
- ✓ 想要增進更多技能的其他程式語言高手

本書內容包含：

- Swift 語法的詳細介紹
- UIKit 常用元件的詳細介紹
- 35 個範例 App
- 3 個完整實戰 App

使用 Swift 2.2 版本

馮信喬 編著

目錄

Introduction	1.1
Swift 基礎	1.2
基本知識	1.2.1
基本型別	1.2.2
基本運算子	1.2.3
字串及字元	1.2.4
集合型別	1.2.5
控制流程	1.2.6
函式	1.2.7
閉包	1.2.8
Swift 進階	1.3
列舉	1.3.1
類別及結構	1.3.2
屬性	1.3.3
方法	1.3.4
下標	1.3.5
繼承	1.3.6
建構過程及解構過程	1.3.7
自動參考計數	1.3.8
可選鍵	1.3.9
錯誤處理	1.3.10
型別轉換	1.3.11
巢狀型別	1.3.12
擴展	1.3.13
協定	1.3.14
泛型	1.3.15
存取控制	1.3.16

UIKit	1.4
UIKit 初探	1.4.1
文字標籤 UILabel	1.4.2
文字輸入 UITextField	1.4.3
輸入多行文字 UITextView	1.4.4
按鈕 UIButton	1.4.5
提示框 UIAlertController	1.4.6
圖片 UIImageView	1.4.7
選取日期時間 UIDatePicker	1.4.8
選擇器 UIPickerView	1.4.9
開關 UISwitch	1.4.10
分段控制 UISegmentedControl	1.4.11
進度條 UIProgressView	1.4.12
滑桿 UISlider	1.4.13
步進器 UIStepper	1.4.14
網頁 UIWebView	1.4.15
表格 UITableView	1.4.16
網格 UICollectionView	1.4.17
搜尋 UISearchController	1.4.18
滑動視圖 UIScrollView	1.4.19
多頁面	1.4.20
導覽控制器 UINavigationController	1.4.21
標籤列控制器 UITabBarController	1.4.22
手勢 UIGestureRecognizer	1.4.23
簡單動畫 Animations	1.4.24
儲存資訊 UserDefaults	1.4.25
資料庫	1.5
SQLite	1.5.1
Core Data	1.5.2
iPhone Apps	1.6

待辦事項	1.6.1
規劃與實作	1.6.1.1
程式之外的設定	1.6.1.2
播放音效	1.6.1.3
UITableView 的編輯模式	1.6.1.4
遊玩臺北	1.6.2
規劃與實作	1.6.2.1
取得遠端 API 資料並儲存	1.6.2.2
地圖與定位	1.6.2.3
記帳	1.6.3
規劃與實作	1.6.3.1
補充	1.7
Xcode 介紹	1.7.1
安裝 Xcode	1.7.1.1
開啟 playground	1.7.1.2
開啟專案	1.7.1.3
介面簡介	1.7.1.4
新增檔案	1.7.1.5
加入檔案	1.7.1.6
刪除檔案	1.7.1.7
新增 Framework	1.7.1.8
使用模擬器及實機測試	1.7.1.9
熱鍵	1.7.1.10
系統關鍵字	1.7.2
駝峰式命名法	1.7.3
關於本書	1.8

Swift 起步走

本站電子書以 Swift 2.2 為基礎，如果需要以 Swift 3.0 為基礎的最新書籍內容，請前往下列電子書店取得：

- [Readmoo](#)
- [Leanpub](#)

v1.0.2

本書內容包含：

- Swift 語法的詳細介紹
- UIKit 常用元件的詳細介紹
- 35 個範例 App
- 3 個完整實戰 App

使用 Swift 2.2 版本，對應 iOS 9 作業系統。

需要工具

目前 Swift 主要是用來開發 iPhone 應用程式(另外還可以開發 iPad 、 Mac 及 Apple Watch 的應用程式)，所以你必須先有一台 Mac (MacBook 、 Mac mini 或是 Mac Pro)，且必須安裝足夠新的作業系統版本，才能夠安裝開發軟體 Xcode 。(像是 Xcode 7.3 只能安裝在 OS X 10.11 以上的作業系統。)

如果你只是想嘗鮮 Swift 的語法而還沒有 Mac ，你也可以選擇線上即時編譯的工具來寫寫 Swift ， IBM 提供了一個線上環境 [IBM Swift Sandbox](#) ，讓你可以直接在瀏覽器上測試 Swift 語法。

課程內容

以下是本書會介紹的內容，如果時間允許，建議依照下列章節依序閱讀，以達較佳學習效果。另外如果是第一次接觸 Xcode ，可以先看看 [Xcode 介紹](#) 章節，來熟悉軟體操作。

- [Swift 基礎](#)

- 基本知識
- 基本型別
- 基本運算子
- 字串及字元
- 集合型別
- 控制流程
- 函式
- 閉包
- Swift 進階
 - 列舉
 - 類別及結構
 - 屬性
 - 方法
 - 下標
 - 繼承
 - 建構過程及解構過程
 - 自動參考計數
 - 可選鍊
 - 錯誤處理
 - 型別轉換
 - 巢狀型別
 - 擴展
 - 協定
 - 泛型
 - 存取控制
- UIKit
 - UIKit 初探
 - 文字標籤 UILabel
 - 文字輸入 UITextField
 - 輸入多行文字 UITextView
 - 按鈕 UIButton
 - 提示框 UIAlertController
 - 圖片 UIImageView
 - 選取日期時間 UIDatePicker
 - 選擇器 UIPickerView
 - 開關 UISwitch

- 分段控制 UISegmentedControl
 - 進度條 UIProgressView
 - 滑桿 UISlider
 - 步進器 UIStepper
 - 網頁 UIWebView
 - 表格 UITableView
 - 網格 UICollectionView
 - 搜尋 UISearchController
 - 滑動視圖 UIScrollView
 - 多頁面
 - 導覽控制器 UINavigationController
 - 標籤列控制器 UITabBarController
 - 手勢 UIGestureRecognizer
 - 簡單動畫 Animations
 - 儲存資訊 UserDefaults
- 資料庫
 - SQLite
 - Core Data
 - iPhone Apps
 - 待辦事項
 - 規劃與實作
 - 程式之外的設定
 - 播放音效
 - UITableView 的編輯模式
 - 遊玩臺北
 - 規劃與實作
 - 取得遠端 API 資料並儲存
 - 地圖與定位
 - 記帳
 - 規劃與實作

範例

本書範例程式碼放在 https://github.com/itisjoe/swiftgo_files，請在閱讀本書內容時，交互搭配範例程式碼。強烈建議先將範例程式碼下載回本地隨時閱讀，以簡省時間。

下載方式請點擊該頁面右上方 `Clone or download` ，接著點擊 `Download ZIP` ，如下圖：



Swift 基礎

在課程的一開始，會先介紹下列程式語言的基礎部分，這些除了在 Swift 中屬於基本知識，也可以在其他程式語言中看到類似內容。

Swift 基礎這章所有的內容都是在 playground 中進行，所以在每節開始前，請先行[建立一個 playground 檔案](#)，或是直接開啓範例程式(每節最後有提供範例檔案位置)。

如果是初次接觸程式語言，務必要將這些東西搞清楚，第一次閱讀時可能在部分地方會有疑問，請交互搭配文字說明與範例程式碼，親自動手寫寫看，了解其中的意思。未來如果有機會接觸其他程式語言，就會發現這些東西都只是寫法上的差異，邏輯都是差不多的。

而如果已經有程式語言經驗，可以大略翻閱以下內容，看看在 Swift 中，這些語法是如何運作。其中 Swift 的[可選型別\(optional type\)](#)特性需要了解一下，後續學習中會經常使用到。

- [基本知識](#)
- [基本型別](#)
- [基本運算子](#)
- [字串及字元](#)
- [集合型別](#)
- [控制流程](#)
- [函式](#)
- [閉包](#)

基本知識

- 印出文字
- 變數及常數
- 註解

印出文字

一開始的一開始，我們先介紹如何將文字印出，在之後的學習中會頻繁的使用到這個函式 `print()`，將要印出的文字擺在小括號 () 中就可以將文字印出，如下：

```
// 印出：Hello World !
print("Hello, World !")
```

Hint

- 函式簡單來說就是一個獨立的功能，可以丟給它參數讓它進行這個功能的處理，後面章節會詳細介紹 [函式](#)。
- 將文字用一對雙引號 " 包起來，即代表一個字串(也就是一段文字)。

變數及常數

程式碼中，會使用變數及常數來暫時保存資料，以供後續程式碼存取、操作或儲存。

兩者的差別在於，常數是宣告後就不能改變的值，像是用來表示一個人的名字或是身分證號碼。變數則是宣告後，還可以依照需求更改這個值，像是要計算累計的購買價錢或計算人數。

使用 `let` 來宣告常數，使用 `var` 宣告變數。

```
// 使用 let 告知常數 name  
let name = "Joe"  
  
// 使用 var 告知變數 price  
var price = 300  
  
// 接著就可以使用這個宣告過的常數 用來印出名字  
print(name)
```

Hint

- 這邊的 `=` 代表的是，將右邊的值指派給左邊的值，如第一行寫的即是將 "Joe" 指派給 `name`，之後程式需要用到的時候就直接使用 `name`，他就會將 Joe 拿出來使用。

命名規則

Swift 支援 Unicode 編碼，所以除了系統保留字、數學符號、箭頭以外，幾乎都可以用來命名。

習慣上通常會以英文單字或是單字組合(小駝峰式命名法)來命名。還有一點，不能以數字為開頭，但變數的其他部分則可以使用數字。

```
let score = 100  
let myFullName = "Kevin Chen"  
//var 123myName = "Jess" // 不能以數字開頭 所以這行會報錯誤  
var myClass55 = "101" // 在其他部分使用數字則是沒問題
```

註解

有時候會需要說明程式內容或是注意事項，就要用到註解，註解是給人看的，在其中的內容都不會被程式所執行。有兩種註解，可以單行使用，或是內容多一點的時候可以使用多行。

```
// 這是單行註解，以兩個斜線開始，因為單行即結束，所以不用特別語法結尾  
let anotherName = "Kevin" // 也可寫在程式的後面，但後面就不能再寫程式了
```

```
/*
```

這是多行註解，以一個斜線跟星號開始，再以一個星號跟斜線做結尾。
這裡面的內容

都會被程式忽略，不會被執行。

```
*/
```

```
/*
```

比較特別的一點是
Swift 的多行註解支援巢狀註解

```
/*
```

也就是多行註解裡
再包著多行註解
*/

仍然是可以執行的

這點在需要暫時將一大段包含註解的程式碼一次註解起來時很好用

```
*/
```

範例

本節範例程式碼放在 [ch1/basics.playground](#)

基本型別

- 型別標註
- 整數
- `Float, Double` 浮點數
- 整數和浮點數轉換
- 布林值
- 字元及字串
- 元組
- 型別別名
- 可選型別
- 強制解析
- 隱式解析可選型別

依據不同需求，變數或常數會需要不同的型別來執行動作，像是身高體重需要有小數點的數字，計算人數需要整數，姓名、名稱需要文字字串。

型別標註

宣告變數或常數時，可以加上型別標註(`type annotation`)，說明這個值的型別(像是整數、字串)。使用方法是在值的後面加上冒號 `:` 接著加上型別名稱，如下：

```
// 宣告一個整數變數
var number: Int

// 宣告一個字串常數
let str: String = "It is a string ."
```

通常很少需要寫型別標註，如果在宣告時給了一個初始值，Swift 則會自動推斷出型別。

整數

整數(`Int`)指的是沒有小數點的數字，可以有符號(正、負、零)或是無符號(正、零)。

Swift 提供 8 、 16 、 32 和 64 位元的有符號的整數型別，依序為 `Int8` 、 `Int16` 、 `Int32` 和 `Int64` ，以及無符號的整數型別，依序是 `UInt8` 、 `UInt16` 、 `UInt32` 和 `UInt64` 。

另外也提供一個特殊的整數型別 `Int` ，這個型別的長度與目前平台的原生字長相同，在 32 位元平台與 `Int32` 相同，在 64 位元平台則與 `Int64` 相同。

所以通常我們使用整數型別時，使用 `Int` 即可。

```
let oneNumber = 12
var anotherNumber = -240
```

Hint

- 還有一個特殊的無符號整數型別為 `UInt` ，這個型別的長度與目前平台的原生字長相同。但基於程式碼的可重複性，避免不同型別數字之間的轉換，以及數字的型別推斷，大部分情況都建議只使用 `Int` 即可。

浮點數

浮點數(`Float` 、 `Double`)指的是有包含小數點的數字， `Float` 跟 `Double` 的差別在於精確度， `Float` 有 6 位數，而 `Double` 可以達到 15 位數，選擇使用哪一個則是看你程式需要處理值的範圍而定，如下：

```
let piValue = 3.1415926
var height = 178.25

// 宣告浮點數時 如果沒有型別標註 通常會將他判斷為 Double
let oneHeight = 165.25 // 型別為 Double
let anotherHeight: Float = 175.5 // 除非型別標註填寫為 Float
```

整數和浮點數轉換

整數和浮點數的轉換必須指定型別。例子如下：

```
// 型別為整數 Int  
let number1 = 3  
  
// 型別為浮點數 Double  
let number2 = 0.1415926  
  
// 相加前 需要將 Int 轉換成 Double 否則會報錯誤  
let pi = Double(number1) + number2  
  
// 這個值的型別也就是 Double  
// 印出：3.1415926  
print(pi)
```

相反來說也行，可以將浮點數轉換成整數，但小數點後的數字就會被截斷。如下：

```
let integerPi = Int(pi)  
  
// 型別為 Int 小數點後的數字被截斷  
// 所以只會印出：3  
print(integerPi)
```

布林值

布林值(`bool`)指的是邏輯上的值，只能為真或假。在後續的學習中會使用到這個特性，依據一個條件式(像是數字比大小或兩數是否相等)會返回真或假，進而使用不同的程式碼。Swift 有兩個布林常數：`true` 跟 `false`，如下：

```
let storeOpen = true  
let forFree = false
```

字元及字串

字元(`character`)指的是依照編碼格式的一個位元組(簡單來說就是一個英文字母、數字或符號)，而字串(`string`)是有序的字元集合(簡單說就是一段文字)，皆是以一對雙引號 " " 前後包起來，如下：

```
let firstString = "Nice to meet you."  
let secondString = "Nice to meet you,too."  
  
// 宣告字串時 不論字數多少 都會判斷為 String  
let str1 = "It is a string ." // 型別為 String  
let str2 = "b" // 型別仍然是 String  
let str3: Character = "c" // 除非型別標註填寫為 Character
```

如果要在字串中加入其他變數或常數，要使用 `\()` 這個方式，如下：

```
let score = 80  
let string = "My score is \(score) ."  
// 印出：My score is 80 .  
print(string)
```

後面章節會介紹更多字串的操作。

元組

元組(`tuple`)是將多個值組合成一個複合值，其內的型別可以不同，以一對小括號 `()` 前後包起來，每個值以逗號 `,` 分隔，如下：

```
// 宣告一個元組並填值進去 依序是字串、整數、浮點數  
let myInfo = ("Kevin Chang", 25, 178.25)
```

要使用其中一個值，可以依照順序的索引值取得(這裡的順序從 0 開始算起，接著依序 1,2,3 ...)，如下：

```
// 取得前面宣告的 myInfo 的第三個值 因為是從 0 開始算 所以是 2  
let myHeight = myInfo.2  
  
// 印出：My height is 178.25  
print("My height is \(myHeight)")
```

也可將一個元組分解成單獨的常數或變數，如下：

```
// 將前面宣告的 myInfo 分解成三個常數
let (myName, myAge, myRealHeight) = myInfo

// 印出：My name is Kevin Chang .
print("My name is \(myName) .")

// 印出：I am 25 years old .
print("I am \(myAge) years old .")
```

如果只需要其中某些值時，分解時可以把不需要的用底線 `_` 標記，如下：

```
let (_, _, myTrueHeight) = myInfo

// 印出：My height is 178.25 .
print("My height is \(myTrueHeight) .")
```

或是在宣告元組時就個別給裡面的值一個名稱也可以，如下：

```
let herInfo = (name:"Jess", age:24, height:160.5)

// 除了用順序取得外 如果有設定名稱 也可以直接取用
// 印出：Her name is Jess .
print("Her name is \(herInfo.name) .")
```

型別別名

型別別名(`type aliases`)就是給已存在的型別定義另一個名字，必須使用關鍵字 `typealias` 來定義型別別名。當你想要給已存在的型別命名一個更有意義的名字時很有用，底下是一個例子：

```
// 將 Int 型別定義一個新的名字 MyType
typealias MyType = Int

// 這時就可以宣告一個 MyType 變數 其實也就是 Int 變數
var someNumber: MyType = 123
```

可選型別

這是 Swift 的一個特性，讓變數或常數可以有沒有值的情況，這與零 `0` 或是空字串 `""` 不同，當沒有值時，變數或常數會返回 `nil`。而 `nil` 代表的就是沒有值，任何型別只要有加上可選型別(`optional type`)都可以設置成 `nil`。使用方法為在型別標註後面加上一個問號 `?`，如下：

```
// 在宣告變數時 型別標註後面加上一個問號 ?
var someScore: Int?
// 因為目前尚未指派 所以目前 score 會被設置成 nil
// 也就是沒有值的狀態

// 設值為 100
someScore = 100
// 再將變數設為 nil 目前又是沒有值的狀態
someScore = nil

// 但如果沒有加上 ? 則是尚未指派的狀態 這時如果直接使用會報錯誤
var totalScore: Int
// 也不能設成 nil 這行同樣也會報錯誤
//totalScore = nil

// 宣告常數也是一樣 在型別標註後面加上一個問號 ?
let someName: String?
```

以下是一個例子，來說明可能會遇到的可選型別的情況：

```
// 宣告一個字串常數
let numberValue = "5566"

// 嘗試將這個字串轉換成整數
let newNumber = Int(numberValue)
```

這時如果原字串內容不是單純的數字，轉換後則是會返回 `nil`，來避免發生錯誤的情況。也就是說返回的是一個可選型別 `Int?`，他可能會是整數，也可能不是 `nil`。

強制解析

當你確認一個可選型別一定有值，則可以在這個變數後面加上一個驚嘆號！，表示這個可選型別有值，請使用它，稱為強制解析(forced unwrapping)，例子如下：

```
// 宣告一個整數常數 並賦值
let number3: Int? = 500
// 以這個例子來說 常數確實有值
// 所以加上驚嘆號 表示這個可選型別有值 可以直接使用
print(number3!)

// 尚未賦值 所以目前是 nil
var number4: Int?
// 仍然要使用的話 下面這行則會報錯誤
//print(number4!)
```

隱式解析可選型別

當可選型別第一次被指派值後，如果可以確定他之後都會有值，這時可以將其改為隱式解析可選型別(implicitly unwrapped optional)，這樣便不需要每次都判斷及解析，作法則是將可選型別的問號？改成驚嘆號！，如下說明：

```
// 可選型別
let oneString: String? = "Good morning ."
// 需要驚嘆號來取值
let anotherString: String = oneString!

// 如果改成隱式解析可選型別
let twoString: String! = "Good night ."
// 則可以直接使用 不用加上驚嘆號
let finalString: String = twoString
```

範例

本節範例程式碼放在 [ch1/types.playground](#)

基本運算子

- 指派運算子
- 數值運算子
- 複合指派運算子
- 比較運算子
- 三元運算子
- 空值聚合運算子
- 區間運算子
- 邏輯運算子
- 括號優先

運算子是檢查、改變、合併值的特殊符號或語句。像是加號 `+` 就是將兩個數相加
`(let number = 1 + 2)`。

指派運算子

`a = b` 表示將右邊的 `b` 指派給左邊的 `a`，如下：

```
let b = 10
var a = 5
// 將 b 指派給左邊的 a
a = b
// 現在 a 等於 10

// 你也可以直接指派元組 會直接分解為多個常數或變數
let (x, y) = (1, 2)
// 現在 x 為 1, y 為 2
```

數值運算子

Swift 中所有數值型別都支援基本的四則運算，加 `+`、減 `-`、乘 `*`、除 `/`。

```
var c = 1 + 2 // c 等於 3
var d = 7 - 2 // d 等於 5
var e = 3 * 2 // e 等於 6
var f = 10.0 / 2.5 // f 等於 4.0
```

加法運算子也可以用於字串的合併：

```
let firstString = "Hello, "
let secondString = "world."
let finalString = firstString + secondString

// 印出：Hello, world.
print(finalString)
```

後面章節會介紹更多字串的操作。

餘數運算

餘數運算(`a % b`)是計算`b`的多少倍剛剛好可以容入`a`，返回多出來的那部分，也就是餘數。

`a = (b * 倍數) + 餘數`

以下的例子為`9 = (4 * 2) + 1`，`9`等於`4`乘上倍數`2`再加上餘數`1`。返回的值就是餘數，也就是`1`。

```
var oneNumber = 9 % 4
print(oneNumber) // 餘數等於 1
```

Swift 比較特別的一點是，浮點數也可以取餘數：

```
var anotherNumber = 8.0 % 2.5 // 8.0 = (2.5 * 3.0) + 0.5
print(anotherNumber) // 餘數等於 0.5
```

一元負號

數值的正負號可以使用前綴`-`(即一元負號)來切換：

```
let number1 = 3
var anotherNumber1 = -number1 // 為 -3
var finalNumber1 = -anotherNumber1 // 為 3
```

一元正號

一元正號 `+` 則是不做任何改變地回傳數值。

```
let number2 = -6
var anotherNumber2 = +number2 // 為 -6
```

複合指派運算子

Swift 提供一個簡潔的方式，將數值運算與指派運算合併，像是 `+=`，很多時候可以簡化程式，如下：

```
var n = 3
n += 2 // 這行等同於 n = n + 2 的簡寫
print(n) // 現在 n 等於 5

// 其他數值運算子也是一樣
n -= 4 // n = n - 4，現在 n 等於 1
n *= 10 // n = n * 10，現在 n 等於 10
n /= 2 // n = n / 2，現在 n 等於 5
n %= 2 // n = n % 2，現在 n 等於 1
```

比較運算子

將兩個數值作比較，並返回這個比較是否成立的布林值，即返回 `true` 或是 `false`，以下是常用的比較運算。

- 等於 (`a == b`)
- 不等於 (`a != b`)
- 大於 (`a > b`)
- 小於 (`a < b`)
- 大於等於 (`a >= b`)

- 小於等於 (`a <= b`)

```
1 == 1 // 返回 true 因為 1 等於 1
2 != 1 // 返回 true 因為 2 不等於 1
2 > 1 // 返回 true 因為 2 大於 1
1 < 2 // 返回 true 因為 1 小於 2
1 >= 1 // 返回 true 因為 1 大於等於 1
2 <= 1 // 返回 false 因為 2 不小於等於 1
```

常用於條件語句，像是 `if` 條件：

```
var i = 1
if i == 1 {
    print("Yes, it is 1 .")
} else {
    print("No, it is not 1 .")
}

// 因為 i 等於 1, 返回 true, 所以會印出：Yes, it is 1 .
```

後面章節會正式介紹 `if` 的使用方法。

當元組中的值可以比較時，你也可以用比較運算子來比較它們的大小。像是 `Int` 和 `String` 型別的值可以比較，所以元組 (`Int, String`) 也可以被比較。而 `Bool` 不能比較，所以內含布林值型別的元組不能被比較。

元組比較大小會依序由左到右逐個比較，直到有兩個值不相等為止。而如果所有值都相等，則會將這一對元組稱為相等的。例子如下：

```
// true 因為 1 小於 2
(1, "zebra") < (2, "apple")

// true 因為 3 等於 3, 但是 apple 小於 bird
(3, "apple") < (3, "bird")

// true 因為 4 等於 4, dog 等於 dog
(4, "dog") == (4, "dog")
```

Hint

- Swift 在比較元組的成員時，限制最多只能比較六個成員，如果有七個或七個以上成員則無法比較。

三元運算子

這是一個簡潔的條件式運算： 問題 ? 答案1 : 答案2 。 問題 需要返回一個是否成立的布林值，表示 true 或 false ，如果為 true ，則是返回 答案1 ，反之如果為 false ，則是返回 答案2 。也就是下列寫法的簡寫：

```
if 問題 {  
    答案1  
} else {  
    答案2  
}
```

以下是個例子：

```
var score = 25  
if score < 60 {  
    score = score + 50  
} else {  
    score = score + 20  
}  
  
print(score) // 現在 score 等於 75
```

後面章節會正式介紹 if 的使用方法。

使用三元運算子可以簡化成下面這樣：

```
var newScore = 25  
newScore = newScore + (newScore < 60 ? 50 : 20)
```

空值聚合運算子

前面章節介紹過的可選型別，Swift 提供一個簡潔的使用方法：`a ?? b`。先判斷 `a` 是否為 `nil`，如果 `a` 有值，不是 `nil`，就會解析 `a` 並返回，但如果 `a` 為 `nil`，則返回預設值 `b`。也就是下面這個寫法的簡寫：

```
a != nil ? a! : b
```

這裡用到前面提到的三元運算子，如果 `a` 不等於 `nil`，則強制解析 `a` 並返回，否則就返回 `b`。以下為一個例子：

```
let defaultColor = "red"
var userDefinedColor: String? // 未指派值 所以預設為 nil
var colorToUse = userDefinedColor ?? defaultColor
// 未指派值給 userDefinedColor，所以會返回 defaultColor
// 這邊即印出：red
print(colorToUse)

// 反之如果有指派值
var userAnotherDefinedColor: String? = "green"
var anotherColorToUse = userAnotherDefinedColor ?? defaultColor
// 這邊即印出：green
print(anotherColorToUse)
```

區間運算子

Swift 提供兩個方便表達一個區間的值的運算子。

閉區間運算子

表示方式為：`a...b`，定義一個包含從 `a` 到 `b` (包括 `a` 和 `b`)的所有值的區間。`b` 必須大於等於 `a`。

```
// 1...5 代表的就是 1,2,3,4,5 這五個數字
for index in 1...5 {
    print("\(index) * 5 = \(index * 5)")
}
// 依序印出
// 1 * 5 = 5
// 2 * 5 = 10
// 3 * 5 = 15
// 4 * 5 = 20
// 5 * 5 = 25
```

後面章節會正式介紹 **for-in** 的使用方法，這邊先理解為是一個會依照規則依序執行動作的語法。

半開區間運算子

表示方式為：`a..`，定義一個從 `a` 到 `b` 但不包括 `b` 的區間。`b` 必須大於等於 `a`，但當 `a` 等於 `b` 時，則不會有值。

```
// 1..5 代表的就是 1,2,3,4 這四個數字 不包括 5
for index in 1..5 {
    print("\(index) * 5 = \(index * 5)")
}
// 依序印出
// 1 * 5 = 5
// 2 * 5 = 10
// 3 * 5 = 15
// 4 * 5 = 20
```

邏輯運算子

Swift 支援三個標準邏輯運算，常與條件式合用。

- 邏輯非 (`!a`)
- 邏輯且 (`a && b`)
- 邏輯或 (`a || b`)

`a` 及 `b` 都是邏輯布林值，且皆會返回一個邏輯布林值，即 `true` 或是 `false`。

邏輯非

`!a` 對一個布林值取相反值，即將 `true` 變 `false`，或是將 `false` 變 `true`。這是一個前綴運算子，且不加空格，例子如下：

```
let isOn = false
if !isOn {
    print("It is on .")
}
```

邏輯且

`a && b` 表示只有當 `a` 跟 `b` 都為 `true` 時，才會返回 `true`，否則如果 `a` 或 `b` 其中一個為 `false`，就會返回 `false`，如下：

```
let isOpen = true
let isMorning = false
if isOpen && isMorning {
    print("Success !")
} else {
    print("Failure !")
}
// 因為其中一個為 false 所以會返回 false
// 即印出：Failure !
```

邏輯或

`a || b` 表示只要 `a` 跟 `b` 其中一個值為 `true` 時，就會返回 `true`，除非 `a` 和 `b` 皆為 `false`，才會返回 `false`，如下：

```
let isSunday = true
let isWeekday = false
if isSunday || isWeekday {
    print("Success !")
} else {
    print("Failure !")
}

// 因為其中一個為 true 就會返回 true
// 即印出：Success !
```

括號優先

以上介紹很多運算子，當一個運算式太複雜時，可以使用括號 () 來標示清楚，同時也用來表明優先級(如同傳統學習數學計算一樣，括號括起來的部份要優先計算)。例子如下：

```
// 數值運算
// 先乘除後加減 所以 number 等於 13
var number = 3 + 2 * 5
// 括號括起來的優先 所以 someNumber 等於 25
var someNumber = (3 + 2) * 5

// 邏輯運算
let isOpen = false
let isWeekend = true
let isMonday = true

// 由左至右依序判斷
if isOpen && isWeekend || isMonday {
    print("Success !")
} else {
    print("Failure !")
}
// 先作"邏輯且"判斷 isOpen && isWeekend 會返回 false
// 再與後面的 isMonday 作"邏輯或"的判斷 會返回 true
// 所以這邊會印出：Success !

// 括號有優先權
if isOpen && (isWeekend || isMonday) {
    print("Success !")
} else {
    print("Failure !")
}
// 括號優先 所以先做"邏輯或"判斷 isWeekend || isMonday 會返回 true
// 再與前面的 isOpen 作"邏輯且"的判斷 會返回 false
// 所以這邊會印出：Failure !
```

範例

本節範例程式碼放在 [ch1/basic_operators.playground](#)

字串及字元

- 字串字面量
- 初始化空字串
- 字串可變性
- 使用字元
- 連接字元及字串
- 字串插值
- 特殊符號
- 計算字串中的字元數量
- 比較字串

字元指的是依照編碼格式的一個位元組(簡單來說就是一個英文字母、數字或符號)，而字串是有序的字元集合(簡單說就是一段文字)，皆是以一對雙引號 " 前後包起來。

字串字面量

在程式碼中包含一段預先定義的字串值作為字串字面量(`string literal`)。字串字面量是由一對雙引號 "" 包著的具有固定順序的文字字元集合，可以為常數和變數提供初始值。

```
// 將一個字串字面量指派給一個常數
let someString = "Some string literal value"
```

初始化空字串

將空的字串字面量指派給變數，或是也可以初始化一個新的 `String` 變數：

```
// 這兩個是一樣的意思
var emptyString = ""
var anotherEmptyString = String()
```

字串可變性

將一個特定的字串指派給一個變數，之後還可以對其修改。而字串指派給一個常數，則無法再做修改，例子如下：

```
var variableString = "Cat"  
variableString = "Book"  
// variableString 現在為 Book  
  
let constantString = "Sun"  
//constantString = "Moon" // 這行會報錯誤 因為常數不能被修改
```

使用字元

字串是有序的字元集合，所以可以使用 `for-in` 迴圈來遍歷字串中的每一個字元：

```
for character in "Dog!".characters {  
    print(character)  
}  
// 依序印出  
// D  
// o  
// g  
// !
```

後面章節會正式介紹 `for-in` 的使用方法。

連接字元及字串

可以簡單的使用加號 `+` 將兩個字串連結在一起，加號指派運算 `+=` 同樣也可以使用。字元也是一樣的使用方式。

```
let str = "Hello"
let secondStr = ", world ."
var anotherStr = str + secondStr
// 印出:Hello, world .
print(anotherStr)

anotherStr += " Have a nice day ."
// 印出:Hello, world . Have a nice day .
print(anotherStr)
```

字串插值

可以使用反斜線 \ 接著小括號 () : \(變數、常數或表達式) 來將其內的值插入到一個字串中。

```
let str1 = "Sunday"
var anotherStr1 = "It is \(str1) ."
// 印出:It is Sunday .
print(anotherStr1)

// 表達式也可以
// 印出:I have 13 cars .
print("I have \(1 + 2 * 6) cars .")
```

特殊符號

字串中可以使用下面這些特殊符號：

- 跳脫字元： \0 (空字元)、 \\ (反斜線)、 \t (水平 tab)、 \n (換行)、 \r (回車)、 \" (雙引號)、 \' (單引號)。
- Unicode 純量：寫成\u{n}(u為小寫)，其中 n 為任意一到八位十六進制數且可用的 Unicode 位碼。

```
// 印出 "Imagination is more important than knowledge" - Einstein
let wiseWords = "\"Imagination is more important than knowledge\" - Einstein"

let dollarSign = "\u{24}"           // $, Unicode 純量 U+0024
let blackHeart = "\u{2665}"         // ❤, Unicode 純量 U+2665
```

計算字串中的字元數量

```
let str2 = "What a lovely day !"

// 印出字元數量：19
print(str2.characters.count)
```

比較字串

有三種方式來比較字串：

- 字串相同或不同 `==` 、 `!=`
- 前綴相同 `hasPrefix`
- 後綴相同 `hasSuffix`

```

let str3 = "It is Sunday ."
let str4 = "It is Sunday ."
let str5 = "It is Saturday ."

// 兩個字串相同 所以成立
if str3 == str4 {
    print("Success")
}

// 印出：Success

// str4 有前綴字串 It is 所以成立
if str4.hasPrefix("It is") {
    print("Success")
}

// 印出：Success

// str5 沒有後綴字串 Sunday . 所以不成立
if str5.hasSuffix("Sunday .") {
    print("Success")
} else {
    print("Failure")
}

// 印出：Failure

```

可以看到有 `str.characters` 、 `str.characters.count` 或是 `str.hasPrefix()` 這種以小數點 . 連接的表示方式，代表的是這個變數的屬性或是方法。

使用方法會依照其設定的規則表示，像是 `str.characters` 就是這個字串的字元集合，`str.characters.count` 是字元集合的字元數量。而 `str.hasPrefix()` 則是會對變數作處理後再返回。往後會很常見到這種用法。

範例

本節範例程式碼放在 [ch1/strings_characters.playground](#)

集合型別

- 陣列
- Sets 集合
- 字典

程式中處理資料時，有時會需要將類似或同質性高的資料集中一起處理(像是購物清單的每項商品或是班級裡每個人的姓名)，這時會需要有不同的資料結構來因應不同的處理情況。

Swift 提供三種基本的集合型別：`Array`、`Set`、`Dictionary` 來儲存集合資料。儲存的資料型別必須明確，且都只能儲存同一種型別的資料。

- `Array` 陣列：按順序儲存資料。
- `Set` 集合：沒有順序、不能重複儲存資料。
- `Dictionary` 字典：沒有順序，鍵值對 `key : value`，也就是可以經由唯一的識別鍵找到需要的值。

陣列

陣列(`array`)使用有序列表儲存同一型別的多個值。相同的值可以多次出現在一個陣列的不同位置中。

宣告陣列變數或常數時的型別，有 `Array<Element>` 及 `[Element]` 兩種方式 (`Element` 是需要明確表示的型別，如 `Int`、`String`、`Double` 等等)，如下：

```
// 宣告儲存 Int 型別的陣列
var arr: Array<Int>
var arr2: [Int]
// 兩種陣列型別表示方式 在功能上是一樣的 所以用第二種就好
```

創建一個空陣列

```
// 宣告一個型別為 Int 的空陣列
var arr3 = [Int]()

// 為這個陣列加上一個值
arr3.append(12)

// 這時如果又要再將這個陣列指派成空陣列
// 因為前面宣告時已經定義好型別
// 所以可以很簡單的使用 [] 來指派成空陣列
arr3 = []

// 或是首次宣告變數時 有明確定義好型別 也可以使用 []
var anotherArr: [Int] = []
```

創建一個帶有預設值的陣列

陣列型別還提供一個可以創建特定大小並且所有資料都被預設的建構方法。將準備加入新陣列的資料項數量 `count` 和適當型別的初始值 `repeatedValue` 傳入陣列建構函式：

```
var threeInts = [Int](count: 3, repeatedValue:0)
// threeInts 是一種 [Int] 陣列，等於 [0, 0, 0]
```

合併兩個陣列

陣列可以使用加法運算子 `+` 來合併兩個陣列，如下：

```
// 首先創建一個 [0,0,0] 的陣列
var secondThreeInts = [Int](count: 3, repeatedValue:0)

// 接著再創建一個 [2,2,2] 的陣列
var anotherThreeInts = [Int](count: 3, repeatedValue:2)

// 最後將兩個陣列合併
var SixInts = secondThreeInts + anotherThreeInts
// 會變成 [0,0,0,2,2,2]
```

直接填入值來創建陣列

可以直接填入值來創建陣列，每個值以逗號 , 分隔，如下：

```
var shoppingList: [String] = ["Eggs", "Milk"]
// 即創建了一個型別為 [String] 且包含兩個值的陣列

// 因為 Swift 會自動的型別推斷
// 所以陣列中如果明確的表示了是什麼型別的值 便不用再標註型別
var anotherList = ["Rice", "Apples"]
// 因為陣列包含著型別為 String 的值
// Swift 可以推斷這個陣列的型別為 [String]
```

存取與修改陣列

一開始說過陣列是有序的集合，所以可以依照索引值(依照順序的排列序號)來取得陣列內的值，如下：

Hint

- 陣列的索引值是從 0 開始算起。

```
var arr4 = ["Apples", "Eggs", "Milk", "Rice", "Water"]

// 陣列的索引值是由 0 開始計算 所以 arr4[2] 指的是第三個
// 印出：Milk
print(arr4[2])

// 要修改一個索引值對應的值 直接將其指派新的值就可以了
arr4[2] = "Oranges"
// 這時已將 Milk 改成 Oranges 所以會印出：Oranges
print(arr4[2])

// 如果要修改一個區間內的值 可以使用區間寫法修改
arr4[1...4] = ["Milk"]
// 這樣會將原本索引值 1 到 4 的值修改成 Milk
// 所以現在 arr4 會變成 ["Apples", "Milk"]
print(arr4)
```

可以使用屬性 `count` 來表示陣列內值的數量，會返回一個非負的整數。另外也可以使用屬性 `isEmpty` 來檢查陣列內是否有值，會返回一個布林值。例子如下：

```
var arr5 = ["Apples", "Eggs", "Milk"]

// 印出陣列中的個數：3
print(arr5.count)

// 將陣列指派為空陣列
arr5 = []

if arr5.isEmpty {
    print("Empty !")
} else {
    print("Not Empty !")
}

// 因為為空陣列 內部沒有值 所以會印出：Empty !
```

加入或移除陣列內的值：

```
// 使用 append(_:) 方法來加入新的值
var arr6 = ["Apples", "Eggs"]
arr6.append("Milk") // 加入新的值 順序會在最後一個
// 現在 arr6 會變成 ["Apples", "Eggs", "Milk"]

// 如果要選擇加入的索引值的位置 使用 insert(_:atIndex:) 方法
// arr6.insert(要加入的值, atIndex:要加入的索引值位置)
arr6.insert("Rice" ,atIndex:0 )
// 現在 arr6 會變成 ["Rice" , "Apples", "Eggs", "Milk"]
// 所有索引值在後面的都會順延往後一個位置

// 移除一個索引位置的值 使用 removeAtIndex(_) 方法
arr6.removeAtIndex(1) // 將排在第二個的 Apples 移除
// 所有後面的值都會遞補向前一個位置
// 現在 arr6 會變成 ["Rice", "Eggs", "Milk"]

// 或者是移除最後一個值 使用 removeLast() 方法
arr6.removeLast()
// 現在 arr6 會變成 ["Rice", "Eggs"]
```

使用 `for-in` 遍歷陣列中的所有值：

```
var arr7 = ["Rice", "Apples", "Eggs", "Milk"]
for item in arr7 {
    print(item)
}
// 會依序印出：
// Rice
// Apples
// Eggs
// Milk

// 當你同時也需要獲得陣列值時 可以使用 enumerate() 方法
for (index, value) in arr7.enumerate() {
    print("Item \(index + 1): \(value)")
}
// 會依序印出：
// Item 1: Rice
// Item 2: Apples
// Item 3: Eggs
// Item 4: Milk
```

後面章節會正式介紹 [for-in](#) 的使用方法。

Sets 集合

集合(Set)用來儲存相同型別且沒有順序、沒有重複的值，當順序不重要或是需要每個值只能出現一次時，可以選擇使用 Set 。

宣告 Set 型別時，使用 `Set<Element>` 這個方式，這裡的 Element 表示 Set 中儲存的型別，如下：

```
// 宣告一個 Set 型別
var someSet: Set<String>
```

與陣列使用上有點類似，基本使用方法如下：

```
// 創建一個空的 Set
var mySet = Set<Int>()

// 可以在宣告時直接填入值
var anotherSet: Set<String> = ["Rock", "Classical", "Hip hop"]

// 指派為一個空 Set， 雖然長得跟陣列使用方法一樣
// 但因為前面已經有明確宣告是 Set， 所以這仍然是 Set<String> 型別
anotherSet = []

// Set 所含的值的數量
// 因為目前是空 Set，印出：0
print(anotherSet.count)

// 使用 isEmpty 檢查 Set 內是否有值
if anotherSet.isEmpty {
    print("Empty !")
} else {
    print("Not empty !")
}
// 印出：Empty !
```

加入或移除 Set 內的值：

```
var mySet2: Set<String> = ["Rock", "Classical"]
// 使用 insert(_:) 來加入新的值
mySet2.insert("Hip hop")
// 目前為 ["Rock", "Classical", "Hip hop"] (無順序)

// 使用 remove(_:) 來移除一個值
// 如果這個值是 mySet2 裡的一個值，會移除掉這個值並返回此值
// 反之 不存在裡面的話 則是返回 nil
mySet2.remove("Rock")
// 目前為 ["Classical", "Hip hop"] (無順序)

// 使用 contains(_:) 來檢查 Set 裡是否包含一個特定的值
if mySet2.contains("Classical") {
    print("Classical is here!")
} else {
    print("Not Here!")
}
// 印出：Classical is here！

// 使用 removeAll() 來移除其內所有的值
mySet2.removeAll()
```

使用 `for-in` 遍歷 `Set` 中的所有值：

```

var mySet3: Set<String> = ["Rice" , "Apples", "Eggs"]
for item in mySet3 {
    print(item)
}
// 會印出：(順序不一定)
// Rice
// Apples
// Eggs

// 因為 Set 沒有順序，可以使用 sort() 方法來返回一個有序的陣列
for item in mySet3.sort() {
    print(item)
}
// 會印出：
// Apples
// Eggs
// Rice

```

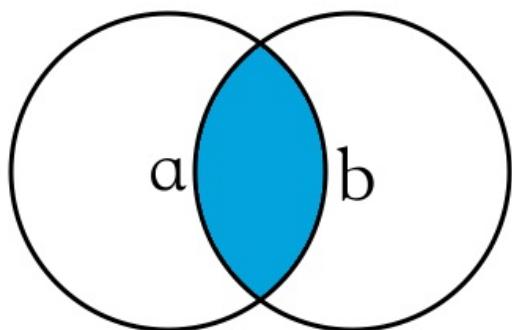
集合(Sets)操作

因為 `Set` 的特性(用來儲存相同型別且沒有順序、沒有重複的值)，Swift 提供以下幾個方法，依據兩個 `Set` 之間交集(有相同的值)與否的關係來創建新的 `Set`：

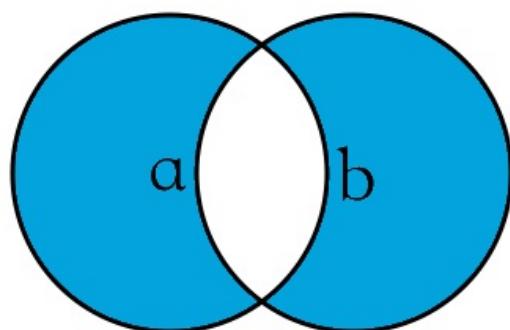
- `intersect(_:)` 創建一個新的 `Set`，其內的值為兩個 `Set` 中個別都包含的值。
- `exclusiveOr(_:)` 創建一個新的 `Set`，其內的值為兩個 `Set` 中只單獨存在其中一個 `Set` 的值。
- `union(_:)` 創建一個新的 `Set`，其內的值包含兩個 `Set` 中所有的值。
- `subtract(_:)` 創建一個新的 `Set`，其內的值在一個 `Set` 內，但不在另一個 `Set` 內。

或參考下圖依據兩個 `Set` 之間關係表現的圖：

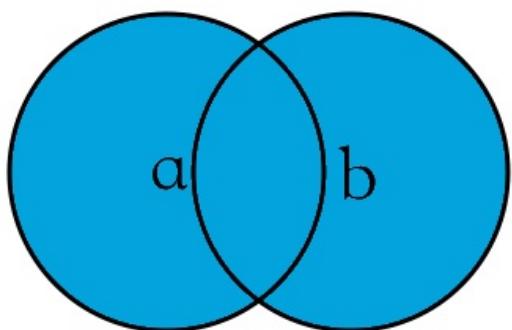
a.intersect(b)



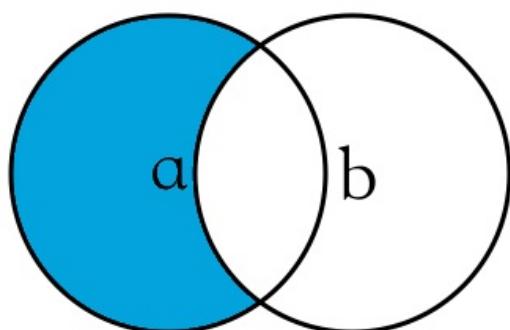
a.exclusiveOr(b)



a.union(b)



a.subtract(b)



例子如下：

```

let oddDigits: Set = [1, 3, 5, 7, 9]
let evenDigits: Set = [0, 2, 4, 6, 8]
let singleDigitPrimeNumbers: Set = [2, 3, 5, 7]

oddDigits.intersect(evenDigits).sort()
// [] 空 Set 因為兩個 Set 沒有交集

oddDigits.exclusiveOr(singleDigitPrimeNumbers).sort()
// [1, 2, 9] 因為兩個 Set 都有 3,5,7
// 所以返回兩個 Set 中 除了這三個值以外的值

oddDigits.union(evenDigits).sort()
// [0, 1, 2, 3, 4, 5, 6, 7, 8, 9] 包含兩個 Set 中所有的值

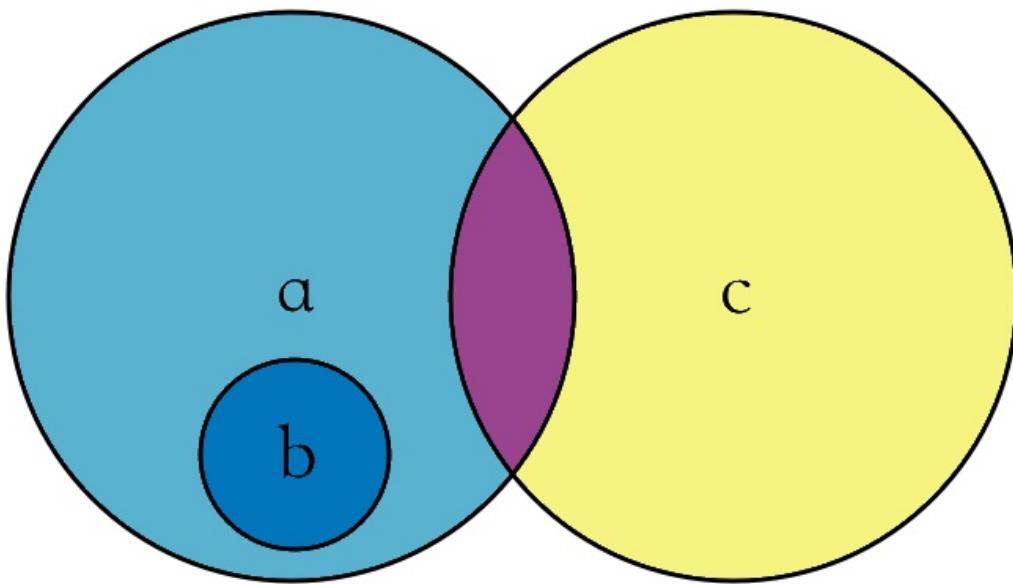
oddDigits.subtract(singleDigitPrimeNumbers).sort()
// [1, 9] 因為 3,5,7 在 singleDigitPrimeNumbers 內
// 所以返回 oddDigits 中 除了這三個值之外的值

```

Swift 提供幾個方法來比對兩個 Set 的關係，皆是返回一個布林值：

- `isSubsetOf(_:)` 判斷一個 Set 是否被包含在另一個 Set 中。
- `isSupersetOf(_:)` 判斷一個 Set 是否包含另一個 Set 所有的值。
- `isStrictSubsetOf(_:)` 判斷一個 Set 是否被包含在另一個 Set 中，且兩個 Set 不相等。
- `isStrictSupersetOf(_:)` 判斷一個 Set 是否包含另一個 Set 所有的值，且兩個 Set 不相等。
- `isDisjointWith(_:)` 判斷兩個集合是否不含有相同的值(是否沒有交集)。

或參考下圖依據兩個 Set 之間的關係表示的圖：



例子如下：

```

let houseAnimals: Set = ["狗", "貓"]
let farmAnimals: Set = ["牛", "雞", "羊", "狗", "貓"]
let cityAnimals: Set = ["鳥", "鼠"]

houseAnimals.isSubsetOf(farmAnimals)
// 返回 true 因為 farmAnimals 包含 houseAnimals 內所有的值

farmAnimals.isSupersetOf(houseAnimals)
// 與上一行意思一樣 只是效果相反 所以也是返回 true

farmAnimals.isDisjointWith(cityAnimals)
// 返回 true 因為 farmAnimals 跟 cityAnimals 沒有交集

```

字典

字典(`dictionary`)用來儲存多個相同型別的值。每個值(`value`)都屬於一個唯一的鍵(`key`)，鍵作為字典中這個值的識別符號，所有鍵的型別也必須相同(鍵與值的型別不一定要相同)。

因為字典內的值沒有順序，所以需要根據這個鍵(`key`)來找到需要的值(`value`)。宣告字典型別時，使用 `Dictionary<Key, Value>` 這個方式，這裡的 `Key` 表示字典裡鍵的型別，`Value` 表示字典裡儲存的型別，如下：

```
// 宣告一個字典型別  
var someDict: Dictionary<String, String>  
  
// 或是這樣也可以  
var someAnotherDict: [String: String]
```

創建一個空字典。如果已經有型別標註了，則可以使用 `[:]` 設為空字典，如下：

```
// 宣告一個空字典 鍵的型別是 String , 值的型別是 Int  
var myDict = [String: Int]()  
  
// 在字典中新增一個值  
myDict["one"] = 1  
  
// 再將字典設為空字典 因為前面已經有型別標註過了 所以使用 [:]  
myDict = [:]
```

一個 `key` 跟一個 `value` 組成一組鍵值對，一個字典以中括號 `[]` 前後包起來，可以包含多組鍵值對，每組以逗號 `,` 分隔，格式如下：

```
[key1:value1, key2:value2, key3:value3]
```

存取與修改字典

使用下標(`subscript`)來存取或是修改字典的值。其餘使用方法與陣列類似，例子如下：

Hint

- 下標(`subscript`)的表示方法為字典變數後加一個中括號 `[]`，中括號裡面填入鍵(`key`)，以取得對應的值(`value`)。

```
// 宣告一個型別為 [String: String] 的字典
var myDict2 = ["TYO": "Tokyo", "DUB": "Dublin"]

// 字典裡值的數量
// 印出：2
print(myDict2.count)

// 檢查字典裡是否有值
if myDict2.isEmpty {
    print("Empty !")
} else {
    print("Not empty !")
}
// 印出：Not empty !

// 如果這個 key 沒有對應到字典裡的值，就新增一個值
myDict2["LHR"] = "London"

// 如果 key 有對應到字典裡的值，則是修改這個值
myDict2["LHR"] = "London Heathrow"

// 如果要移除這個值 則是將其設為 nil
myDict2["LHR"] = nil
```

除了使用下標語法，字典也提供方法可以修改或刪除，如下：

```
var myDict3 = ["LHR": "London", "DUB": "Dublin"]

// 使用 updateValue(_:forKey:) 更新一個值
// 返回一個對應值的型別的可選值（這邊就是返回一個 String? ）
myDict3.updateValue("London Heathrow", forKey: "LHR")
// 印出：London Heathrow
print(myDict3["LHR"])

// 使用 removeValue(forKey:) 移除一個值
// 返回被刪除的值，如果沒有對應的值的話則會返回 nil
myDict3.removeValue(forKey: "DUB")
```

只需要字典中的鍵或值時，可以使用 `keys` 或 `values` 屬性。這時會建立一個鍵或值的新陣列，例子如下：

```
var myDict4 = ["LHR": "London", "DUB": "Dublin"]

// 只需要鍵時，使用 keys 屬性，取得一個只有鍵的陣列
var someArr1 = [String](myDict4.keys)
// someArr1 為 ["LHR", "DUB"]

// 只需要值時，使用 values 屬性，取得一個只有值的陣列
var someArr2 = [String](myDict4.values)
// someArr2 為 ["London", "Dublin"]

// 如果需要固定順序的話 可以加上 sort() 方法
someArr1.sort()
someArr2.sort()
```

使用 `for-in` 遍歷字典中的所有值：

```
var myDict5 = ["LHR": "London", "DUB": "Dublin"]

for (code, n) in myDict5 {
    print("\(code): \(n)")
}

// 印出：
// LHR: London
// DUB: Dublin

// 如果只需要鍵或值時，使用 keys 或 values 屬性
for code in myDict5.keys {
    print(code)
}

// 印出：
// LHR
// DUB

for n in myDict5.values {
    print(n)
}

// 印出：
// London
// Dublin
```

範例

本節範例程式碼放在 [ch1/collection_types.playground](#)

控制流程

- For-in 循環
- While 循環
- 條件語句
- 可選綁定
- Switch
- 控制轉移語句
- 提前退出
- 斷言
- 檢測 API 可用性

Swift 提供了可以循環執行任務的 `for-in` 和 `while` 迴圈，以及根據條件選擇執行不同程式碼分支的 `if` 和 `switch` 語句，還有控制流程跳轉到其他程式碼的 `break` 和 `continue` 語句。

For-in 循環

使用 `for-in` 遍歷一個集合內的所有元素，像是一個數字區間、陣列、字典中的值或是字串內的字元，例子如下：

```
// 一個一到三的閉區間
for index in 1...3 {
    print(index)
}
// 會依序印出：
// 1
// 2
// 3
```

上述程式內的 `index` 不用做宣告的動作，在每次遍歷開始時會被自動指派值，預設是一個常數。

另外如果只是要純粹的循環，不需要用到區間內的每一個值，可以用下底線 `_` 來代替。

```

let base = 2
var total = 1
for _ in 1...3 {
    total *= base
}

// 因為循環了三次 所以乘了三次 印出：8
print(total)

```

使用 `for-in` 遍歷一個陣列(`array`)或是字典(`dictionary`)。遍歷字典時，會使用元組(`tuple`)來分別表示鍵與值，例子如下：

```

let arr = ["Apple", "Book", "Cat"]
for n in arr {
    print(n)
}

// 依序印出：
// Apple
// Book
// Cat

let dict = ["Apple":12, "Book":3, "Cat":5]
for (key, values) in dict {
    print("\(key) : \(values)")
}

// 印出：(因為字典沒有順序 所以不一定是這樣的順序)
// Apple : 12
// Book : 3
// Cat : 5

```

While 循環

Swift 提供兩種 `while` 循環方式：`while` 及 `repeat-while`，兩者都是循環地執行程式直到條件表達式返回 `false`，差別在於，後者一開始在檢查條件表達式之前，一定會先執行一次內部程式。

While

`while` 循環格式如下：

```
while 條件表達式 {  
    每次循環執行的程式  
}
```

`while` 會循環地執行程式直到條件表達式返回 `false`，例子如下：

```
var n = 2  
while n < 20 {  
    n = n * 2  
}  
  
// 印出：32  
print(n)
```

Repeat-while

`repeat-while` 循環格式如下：

```
repeat {  
    每次循環執行的程式  
} while 條件表達式
```

`repeat-while` 會先執行一次程式，再檢查條件表達式，接著循環地執行程式直到條件表達式返回 `false`。

```
var m = 512  
repeat {  
    m = m * 2  
} while m < 100  
  
// 印出：1024  
print(m)  
// 因為不論如何 都會先執行一次程式 所以 m 會先乘一次 2 為 1024  
// 接著檢查條件表達式 會返回 false 即結束這個循環
```

條件語句

條件語句為根據不同特定條件執行不同的程式。Swift 提供兩種條件語句：`if` 與 `switch`，如果需要判斷的條件較單純或可能的情況較少時，可以使用 `if`，反之使用 `switch`。

If

最簡單的形式為只有一個條件表達式，而只有當這個條件表達式返回 `true`，才執行其內的程式，例子如下：

```
let someNumber = 2
if someNumber == 2 {
    print("It is 2 .")
}
```

或是當條件表達式返回 `false` 時，需要執行另一段程式，則使用 `else`，如下：

```
let number = 10
if number > 20 {
    print ("Number is larger than 20 .")
} else {
    print ("Number is smaller than 20 or equal to 20 .")
}
// 印出：Number is smaller than 20 or equal to 20 .
```

又或是有多個條件需要判斷，可以將 `else` 跟 `if` 連在一起，最後一個 `else` 會在所有條件都不成立(返回 `false`)時被執行，如下：

```

let number2 = 100
if number2 < 20 {
    print ("數字小於 20")
} else if number2 < 200 {
    print ("數字不小於 20，但小於 200")
} else if number2 < 1000 {
    print ("數字不小於 200，但小於 1000")
} else {
    print ("數字不小於 1000")
}
// 印出：數字不小於 20，但小於 200

```

上述程式中最後一個 `else` 不是一定要有，也可以省略，但就可能會沒有返回 `true` 的條件表達式，例子如下：

```

let number3 = 10
if number3 > 50 {
    print("number3 > 50")
} else if number3 > 200 {
    print("number3 > 200")
}
// 這時候不會有東西被印出來

```

可選綁定

使用可選綁定(`optional binding`)來判斷可選型別是否有值，如果有值的話就指派給一個臨時常數或臨時變數，並執行其內部的程式，這個臨時常數或變數只能在其內部使用。 `if` 跟 `while` 語句都可以使用可選綁定。格式如下：

```

if let 臨時常數 = 可選型別 {
    執行的程式
} else {
    可選型別沒有值 也就是 nil 時 執行的程式
}

```

以下是一個例子：

```
// 字串內容是純整數 所以經過轉換後會是一個整數
let str = "123"
if let number = Int(str) {
    print("字串 \"\\"(str)\\" 轉換成一個整數 \"(number)\")"
} else {
    print("字串 \"\\"(str)\\" 不是一個整數")
}

// 如果字串內容不是整數 轉換後會返回 nil
let str2 = "It is a string ."
if let number = Int(str2) {
    print("字串 \"\\"(str2)\\" 轉換成一個整數 \"(number)\")"
} else {
    print("字串 \"\\"(str2)\\" 不是一個整數")
}
```

Switch

`switch` 會將一個值與多個情況作比對，根據第一個比對成功的情況，會執行相對應的程式。所有 `case` 必須涵蓋全部可能的情況，如果有未涵蓋的部份，最後需要補上 `default`。最簡單的格式如下：

```
switch 值 {
case 比對情況1:
    相對應情況1 執行的程式
case 比對情況2, 比對情況3: // 多個情況可以用逗號 , 隔開
    相對應情況2或情況3 執行的程式
default:
    以上情況比對都不成功時 執行的程式
}
```

每個 `case` 中，都一定要有相對應執行的程式，如果沒有的話會報錯誤，例子如下：

```

let number4 = 2
switch number4 {
case 1:
case 2:
    print("It is 2 .")
case 3:
default:
    print("沒有比對到")
}
// case 1 以及 case 3 內部都沒有執行的程式 所以這邊會報錯誤

```

與其他程式語言不一樣的是，有些程式語言的 `switch` 需要在每個 `case` 內的程式最後一行加上 `break` 來結束，否則會繼續執行底下其他 `case` 中的程式。

`Swift` 在遇到第一個情況 `case` 比對成功後，即會結束 `switch` 這部份的動作，繼續執行 `}` 之後的程式。(除非使用 `fallthrough`，稍後即會提到。)

區間匹配

`case` 中比對的情況也可以是一個區間：

```

let number5 = 120
var str3: String
switch number5 {
case 0...10:
    str3 = "幾"
case 11...100:
    str3 = "很多"
case 101...1000:
    str3 = "非常多"
default:
    str3 = "超級多"
}

// 因為 120 在 101...1000 這個區間內
// 印出：我有非常多顆蘋果
print("我有\($str3)顆蘋果")

```

元組

`switch` 可以使用元組(tuple)來一次比對多個值，元組內可以是值也可以是區間，如果要忽略比對其中一項的話，可以填入下底線 `_`：

```
let somePoint = (1, 1)
switch somePoint {
    case (0, 0):
        print("(0, 0) 在原點")
    case (_, 0):
        print("\((somePoint.0), 0) 在 X 軸上")
    case (0, _):
        print("(0, \((somePoint.1))) 在 Y 軸上")
    case (-2...2, -2...2):
        print("\((somePoint.0), \((somePoint.1))) 在方形內")
    default:
        print("\((somePoint.0), \((somePoint.1))) 在方形外")
}

// 印出：(1, 1) 在方形內
```

值綁定

`case` 可以將比對的值綁定(value binding)到一個臨時的常數或變數，以便在其內的程式中使用：

```
let onePoint = (2, 0)
switch onePoint {
    case (let x, 0):
        print("在 X 軸上, x 的值為 \((x))")
    case (0, let y):
        print("在 Y 軸上, y 的值為 \((y))")
    case let (x, y):
        print("\((x), \((y))) 不在 X 軸也不在 Y 軸上")
}

// 印出：在 X 軸上, x 的值為 2
```

另外可以使用 `where` 來判斷額外的條件：

```
let number6 = 20
switch number6 {
    case 1...100 where number6 == 50:
        print("在 1...100 區間內 且值為 50")
    case 1...100 where number6 == 20:
        print("在 1...100 區間內 且值為 20")
    default:
        print("沒有比對到")
}
```

控制轉移語句

控制轉移語句(`control transfer statement`)可以改變程式的執行順序，或是跳轉執行程式。

Continue

`continue` 表示在循環流程中，立即停止本次循環，重新開始此流程的下個循環。
以下是個例子：

```
for n in 1...10 {
    // n 除以 2 的餘數為零時(即 n 為偶數時)
    if n % 2 == 0 {
        // 停止本次循環 重新開始此流程的下個循環
        continue
    }
    print(n)
}
// 因為當 n 為偶數時 都會走到 continue 即立即停止本次循環
// 所以會被依序印出的只有 1, 3, 5, 7, 9
```

Break

`break` 會立即停止這個循環流程，接著繼續執行之後的程式。如果要提早結束 `switch` 時也可以使用。以下是個例子：

```

for n in 1...10 {
    // n 大於 2 時
    if n > 2 {
        // 立即停止這個循環流程
        break
    }
    print(n)
}
// 當迴圈進行到第三圈時 因為 n 為 3 已經大於 2 了
// 即會進到 break 同時立即停止這個循環流程
// 所以這邊只會印出 1, 2

```

Fallthrough

Swift 的 `switch` 中，只要比對到一個 `case` 即會執行其內的程式，並結束這整個 `switch` 的動作，如果在特殊情況下需要執行緊接著的下一個 `case` 內的程式，就要用到 `fallthrough`。

```

let number7 = 5
var str4 = ""
switch number7 {
    case 2, 3, 5, 7, 11, 13, 17, 19:
        str4 += "It is a prime number. "
        fallthrough
    case 100, 200:
        str4 += "Fallthrough once. "
        fallthrough
    default:
        str4 += "Fallthrough twice."
}
// 印出：
// It is a prime number. Fallthrough once. Fallthrough twice.
print(str4)
// 雖然只比對到第一個 case 但兩個 case 都有使用 fallthrough
// 所以最後 str4 是將所有字串相加

```

Hint

- 加上 `fallthrough` 後進入到的下一個 `case`，不會對其條件做比對，而是直接執行其內的程式。

帶標籤的語句

有時會需要較複雜的混用多個 `switch`、`for` 或是 `while`，當需要對其中一個循環流程使用 `continue` 或 `break` 來跳轉或停止時，可以額外地對 `continue` 或 `break` 指名是屬於哪個循環流程。格式如下：

```
標籤名稱: while 條件 {  
    循環執行的程式  
}
```

以下是個例子：

```
var number8 = 1  
gameLoop: while number8 < 10 {  
    switch number8 {  
        case 1...4:  
            number8 += 1  
        case 5:  
            number8 *= 10  
            // break 標註為 gameLoop 的 while 迴圈  
            break gameLoop  
        default:  
            number8 += 1  
    }  
    // 印出: 50  
    print(number8)  
    // 在 1...4 區間內時 會將 number8 加 1  
    // 直到 n==5 時 會乘以 10 並結束 while 循環  
    // 因為有將 while 加上名為 gameLoop 的標籤  
    // 所以可以很明白的了解 case 5 中的 break 是要結束 while  
    // 因此這個 while 實際只循環 5 次即結束
```

提前退出

有點類似 `if` 的用法，`guard` 同樣會有一個條件表達式且會返回一個布林值，不同的地方在於，`guard` 後面一定要接一個 `else`，如果條件表達式返回 `false` 時，會執行 `{}` 內的程式。

```
guard 條件表達式 else {  
    // 條件表達式返回 false 時 執行的程式  
    // 可以配合 continue, break 或 return 來控制轉移流程  
}  
  
// 條件表達式返回 true 時 則繼續執行接下來的程式
```

以下是一個例子：

```

// 建立一個名叫 post() 的函式
// 需要傳入一個型別為 [String: String] 的字典
func post(article: [String: String]) {
    // 首先取得傳入字典中 鍵為 title 的值 並指派給一個常數
    guard let insideTitle = article["title"] else {
        // 如果沒有鍵為 title 的值 這裡面的程式就會被執行
        // 函式中的 return 表示會直接結束這個函式
        return
    }

    // 上面的 insideTitle 如果有正確的被指派值 則會繼續進行到這裡
    print("標題是 \(insideTitle) ,")

    // 接著取得傳入字典中 鍵為 content 的值 並指派給一個常數
    guard let insideContent = article["content"] else {
        // 如果沒有鍵為 content 的值 這裡面的程式就會被執行
        print("但是沒有內容。")
        return
    }

    // 上面的 insideContent 如果有正確的被指派值 則會繼續進行到這裡
    print("內容為 \(insideContent)。")
}

post(["title": "Article_1"])
// 印出：標題是 Article_1 ,
// 印出：但是沒有內容。

post(["title": "Article_2", "content": "Article_2_full_content"])
// 印出：標題是 Article_2 ,
// 印出：內容為 Article_2_full_content。

```

Hint

- `guard` 條件表達式中，使用可選綁定而被指派的常數或變數，可以在函式 {} 範圍裡接下來的程式中使用。

雖然 `guard` 用法與 `if` 類似，但使用 `guard` 可以一次專注於一種條件的情況，提高程式的可靠性，也可以讓函式內容更清晰好讀。

後面章節會正式介紹函式。

斷言

斷言(`assertion`)會在執行時判斷一個邏輯條件是否為 `true`，而如果為 `false` 的話，程式則會被中止，使用函式 `assert()`，格式如下：

```
assert(邏輯條件, 返回 false 時會顯示的訊息)
```

```
// 或是省略成只有判斷邏輯  
assert(邏輯條件)
```

底下是一個例子：

```
var age = -25  
assert(age > 0, "年齡必須大於零")
```

斷言通常使用在測試階段的程式，你可以使用斷言來保證在執行其他程式碼之前，某些重要的條件已經被滿足。可能適用的情況如下：

- 數字可能過大或過小。
- 需要給函式傳入一個值，但是非法的值可能導致函式不能正常執行。
- 一個可選值為 `nil`，但後面需要這個值為一個非 `nil` 值。

檢測 API 可用性

Swift 可以讓你檢查 API 的預設支持，可以防止不小心使用到了對於當前部屬目標不可用的 API。以下是使用方式：

```
if #available(平台名稱 版本號, ..., *) {  
    // 在某個平台或版本下使用特別的 API  
} else {  
    // 而其他的平台或版本則使用其他的 API  
}
```

平台名稱可以是 `iOS` 、 `OSX` 或 `watchOS` 。版本號可以是大版本號像是 `iOS 9` ，或是較小的版本像是 `iOS 8.3` 或 `OSX 10.10.3` ，以下是一個例子：

```
if #available(iOS 9, OSX 10.10.3, *) {  
    // 在 iOS 使用 iOS 9 的 API  
    // 在 OSX 使用 OSX 10.10.3 的 API  
} else {  
    // 使用先前版本的 iOS 和 OSX 的 API  
}
```

範例

本節範例程式碼放在 [ch1/control_flow.playground](#)

函式

- 函式定義
- 函式參數
- 函式返回值
- 函式型別
- 巢狀函式

函式定義

函式(`function`)是一個獨立的程式碼區塊，用來完成特定任務。當你需要一個會多次使用到的功能(例如計算總和)時，將這個功能寫成一個函式可以簡化程式碼。

函式命名的方式與常數變數一樣，但名稱後面需要加上小括號 `()`，像是前面章節很常使用的 `print()`，就是一個函式。建立一個函式要使用 `func` 關鍵字，函式格式如下：

```
func 函式名稱( 參數: 參數型別 ) -> 返回值型別 {  
    內部執行的程式  
    return 返回值  
}
```

最簡單的函式格式如下：

```
func 函式名稱() {  
    內部執行的程式  
}
```

沒有參數也沒有返回值，以下是一個例子：

```
// 建立一個函式
func simpleOne() {
    print("It is a simple function .")
}

// 呼叫函式
simpleOne()
// 這樣就會執行函式內的程式 這邊是一個簡單的功能
// 印出：It is a simple function .
```

函式參數

函式可以傳入參數(parameter)，參數需要明確標註型別，會將參數指派給一個常數，格式如下：

```
func 函式名稱( 參數指派給的常數：型別標註 ) {
    內部執行的程式
}
```

以下是一個例子：

```
// 建立函式 有一個型別為 Int 的參數
// 函式的功能是將帶入的參數加一 並印出來
func addOne(number: Int) {
    // number 即為被指派參數的常數 只能在函式內部範圍內使用
    print(number + 1)
}

// 呼叫函式 傳入整數 12
// 印出：13
addOne(12)
```

多重參數函式

函式如果有超過一個參數時，要依序將參數填入，並以逗號 , 隔開，以下是一個例子：

```
// 建立一個有兩個參數的函式 參數的型別分別為 String 及 Int
func hello(name: String, age: Int) {
    print("\(name) is \(age) years old .")
}

// 呼叫函式
hello("Jack", age: 25)
```

上述程式中可以看到呼叫函式時，第二個參數前需要標註其對應的名稱。

Hint

- 如果函式有更多參數，除了第一個參數不用之外，第二個以及之後的參數都需要標註其對應的名稱。

函式參數名稱

前面提到呼叫函式時，第二個以及之後的參數都需要標註其對應的名稱，其實可以為函式特別設定這個名稱，稱作外部參數名稱(external parameter name)，而原先設定的名稱為內部參數名稱(local parameter name)。

- 外部參數名稱用於呼叫函式時，標註給函式的參數。
- 內部參數名稱用於函式內部操作。

以下為一個有外部參數名稱及內部參數名稱的函式格式：

```
func 函式名稱(外部參數名稱1 內部參數名稱1: 型別1,
    外部參數名稱2 內部參數名稱2: 型別2) {
    // 內部執行的程式
}
```

以下為一個例子：

```
func hello2(name n: String, age a: Int) {  
    // n 跟 a 為內部參數名稱 在函式內部使用  
    print("\(n) is \(a) years old .")  
}  
  
// name 跟 age 為外部參數名稱 呼叫函式時要標註在參數之前  
hello2(name: "Jack", age: 25)
```

前面提到第一個參數前可以不用寫外部參數名稱，如果其後的參數也不想寫外部參數名稱時，只要使用下底線 `_` 即可，以下為一個例子：

```
// 第二個參數的外部參數名稱 使用下底線 _ 替換  
func hello3(name: String, _ age: Int) {  
    print("\(name) is \(age) years old .")  
}  
  
// 呼叫函式時 第二個參數就可以不用寫外部參數名稱  
hello3("Jack", 33)
```

Swift 中，函式第一個參數是預設不需要加上外部參數名稱，呼叫函式時也不需要寫。而其後的參數，可以使用其內部參數名稱作為外部參數名稱，就如同前面一開始介紹多重參數函式時的方式，如下：

```
// 第一個參數不需要外部參數名稱  
// 第二個及之後的參數 可以使用其內部參數名稱作為外部參數名稱 這邊就是 age  
func hello4(name: String, age: Int) {  
    print("\(name) is \(age) years old .")  
}  
  
// 呼叫函式時 第二個參數的外部參數名稱是 age 而內部參數名稱也是 age  
hello4("Jack", age: 25)
```

Swift 會這樣稍微複雜的設計函式參數名稱，是為了讓呼叫函式時，語意可以更明顯，以下是一個例子：

```
func sayHello(to name: String, and name2: String) {  
    print("Hello \(name) and \(name2) !")  
}  
  
// 外部參數名稱設為 to 跟 and  
// 這行看起來就像個完整的英文句子 可以明顯的知道這個函式要幹嘛  
sayHello(to: "Joe", and: "Amy")
```

預設參數值

可以對函式的參數設定一個預設值，如果未傳入值時，則內部就使用這個預設值。如下：

```
func someFunction(number: Int = 12) {  
    print(number)  
}  
  
// 印出：6  
someFunction(6)  
  
// 沒有傳入值 則會使用預設值 印出：12  
someFunction()
```

Hint

- 有多重參數時，最好將有預設值的參數放在參數列表的最後，這樣呼叫函式時，可以保證其他無預設值參數的順序是一致的。

可變數量參數

函式的可變數量參數(variadic parameter)可以接受零個或多個值，呼叫函式時，使用可變數量參數來傳入數量不確定的參數，使用方法是在參數的型別後面加上三個點(...)，以下是個例子：

這個函式有一個型別為 Double... 的可變數量參數，在函數內會轉成一個型別為 [Double] 的陣列常數。

```

func arithmeticMean(numbers: Double...) -> Double {
    var total: Double = 0
    for number in numbers {
        total += number
    }
    return total / Double(numbers.count)
}
arithmeticMean(1, 2, 3, 4, 5) // 返回 3.0
arithmeticMean(3, 8, 19) // 返回 10.0

```

Hint

- 一個函式只能有一個可變數量參數。
- 函式如果有預設值參數，也有可變數量參數時，必須把可變數量參數放在參數列表的最後。

輸入輸出參數

普通函式的參數使用範圍都只在函式程式體內，如果想要一個可以修改參數的函式，並在呼叫函式結束後，這個修改仍然存在，則必須將參數設定為輸入輸出參數 (In-Out Parameters)。使用方式為：

- 定義函式時，在參數名稱前加上 `inout`。此參數不能有預設值，也不能再設為可變數量參數。
- 當呼叫函式，傳入的參數作為輸入輸出參數時，需要在參數前加上 `&`。這個參數只能是一個變數，不能是常數、字面量(單純的數值或字符串)。

以下是個例子：

函式

```
// 定義一個[有一個輸入輸出參數]的函式 參數前要加上 inout
func newNumber(inout number: Int) {
    number *= 2
}

var n = 10
print(n) // 這時 n 為 10

// 傳入的參數在函式結束後 變更仍然存在
newNumber(&n)

print(n) // 所以這時再印出 就會是 20
```

函式返回值

函式除了可以傳入參數，同時也可以返回值，使用方式是在函式 () 後面接著一個減號跟大於組成的箭頭 -> ，然後再接著寫返回值的型別標註，函式內部要使用 return 語句後接著返回值來返回。使用 return 後會隨即終止函式的動作，函式內部其後的程式都不會繼續執行。以下是個例子：

```
// 定義函式 有一個型別為 Int 的參數以及一個型別為 Int 的返回值
// 函式的功能是將帶入的參數加十 並返回
func addTen(number: Int) -> Int {
    let n = number + 10
    // 使用 return 來返回值 這個返回值的型別要與上面標註的相同
    return n
}

// 呼叫函式 傳入整數 12 會返回 22
let newNumber = addTen(12)

// 印出：22
print(newNumber)
```

多重返回值函式

函式

函式可以有不只一個的返回值，返回值超過一個時以一個元組(Tuple)返回，元組內包含著要回傳的每個值的型別標註。在建立函式時也可以給返回元組的每個值加上名稱。以下是一個例子：

```
// 定義函式 有一個型別為 Int 的參數，返回兩個型別為 Int 的值
func findNumbers(number: Int) -> (Int, Int) {
    let n = number + 10
    // 返回一個元組
    return (number, n)

    // 合併成一行直接返回也是可以
    // return (number, number + 10)
}

// 呼叫函式 傳入整數 12 會返回 (12, 22)
let numbers = findNumbers(12)
// numbers 為一個元組 可以自 0 開始算 依序取得其內的值
// 印出：12 and 22
print("\(numbers.0) and \(numbers.1)")

// 定義另一個函式 將上面函式中返回的元組內的值加上名稱
func findNumbers2(number: Int) -> (oldNumber: Int, newNumber: Int)
{
    let n = number + 10
    return (number, n)
}

// 呼叫函式 傳入整數 24 會返回 (24, 34)
let numbers2 = findNumbers2(24)
// 這邊即可使用定義函式時 返回元組內的值設定的名稱
// 印出：24 and 34
print("\(numbers2.oldNumber) and \(numbers2.newNumber)")
```

可選元組返回型別

如果整個返回的元組可能會沒有值(`nil`)，可以將此返回元組設為可選元組型別，使用方式為在括號 `()` 後面接著一個問號 `?`，像是 `(Int, String)?` 或是 `(String, Int, Int)?`。以下是個例子：

```
// 定義函式 參數為一個型別為 [Int] 的陣列
// 返回值為 包含兩個型別為 Int 的元組 或是 nil
func findNumbers3(arr: [Int]) -> (Int, Int)? {
    // 檢查傳入的陣列 如果其內沒有值的話 就直接返回 nil
    if arr.isEmpty {
        return nil
    }
    let n = arr[0] + 10
    let n2 = arr[0] + 100
    return (n, n2)
}

// 呼叫函式
// 因為返回的是一個可選型別 這邊先做可選綁定的動作 確定有值後再印出來
if let numbers = findNumbers3([11, 22, 33]) {
    print("\(numbers.0) and \(numbers.1)")
}
// 如果傳入的陣列 內部沒有值
if let numbers = findNumbers3([]) {
    print("這裡不會被印出來")
}
```

函式型別

函式也是一種型別，由函式的參數型別和返回值型別組成：

```
func someFunction(a: String, b: Int) -> Int {
    // 函式內部程式
}
```

上述程式中的函式，這個函式型別就標註為 `(String, Int) -> Int`，意思就是有兩個型別依序為 `String` 跟 `Int` 的參數，會返回一個型別為 `Int` 的值。

函式

以下是另一個例子，如果沒有參數也沒有返回值時，函式型別標註為 `() -> Void`。

```
func hello5() {
    print("Hello !")
}
```

這個函式型別也可以標註為 `() -> ()`。實際上，沒有返回值的函式，會返回一個特殊的值，叫做 `void`，也就是一個空的元組(tuple)，沒有任何元素，可以寫成 `()`。

使用函式型別

前面剛提過函式是一種型別，所以也可以將變數或常數宣告為一個函式，然後可以指派一個適當的函式。

以下是一個例子，宣告一個變數 `mathFunction`，是一個型別為 `(Int) -> Int` 的函式，最後指派一個已經存在且型別相同的函式：

```
var mathFunction: (Int) -> Int = addTen
```

函式型別作為參數型別

函式可以作為另一個函式的參數，以下是一個例子：

```
// 定義一個將兩個整數相加的函式
func addTwoInts(number1: Int, number2: Int) -> Int {
    return number1 + number2
}

// 定義另一個函式，有三個參數依序為
// 型別為 (Int, Int) -> Int 的函式, Int, Int
func printMathResult(
    mathFunction: (Int, Int) -> Int, _ a: Int, _ b: Int) {
    print("Result: \(mathFunction(a, b))")
}

// 將一個函式 addTwoInts 傳入函式 printMathResult
printMathResult(addTwoInts, 3, 5)
```

上述程式中，作為參數的函式只需要型別正確，不用了解其內的程式如何運作，這表示可以將 `printMathResult` 函式一部分的操作交給呼叫函式的人來實作，也是以一種型別安全(`type-safe`)的方式來保證傳入函式的呼叫是正確的。

函式型別作為返回型別

函式可以作為另一個函式的返回值，使用方式為在返回箭頭(`->`)後寫一個完整的函式型別。以下是個例子：

```
// 定義一個將傳入的參數加一的函式
func stepForward(input: Int) -> Int {
    return input + 1
}

// 定義一個將傳入的參數減一的函式
func stepBackward(input: Int) -> Int {
    return input - 1
}

// 建立一個參數為布林值的函式 會返回一個函式
// 根據布林值返回上述兩個函式的其中一個
func chooseStepFunction(backwards: Bool) -> (Int) -> Int {
    return backwards ? stepBackward : stepForward
}

// 宣告一個整數常數
let number = 3
// 宣告一個函式常數
let someFunction2 = chooseStepFunction(number > 0)
// 根據 chooseStepFunction 函式內容
// 傳入 true 時 會返回 stepBackward 函式
// 所以 someFunction2 會被指派為 stepBackward

someFunction2(10) // 返回 9
```

Hint

- 這邊宣告函式常數時沒有標註型別，因為就如同其他變數或常數宣告時，Swift 會根據指派的函式自動判斷出型別。

巢狀函式

到目前為止所使用的函式都叫全域函式(`global functions`)，定義在全區域中，每個地方都可以使用。

如果將函式建立在另一個函式中，稱作巢狀函式(nested function)，被建立在其內的函式只能在裡面使用，也可以當做返回值返回以讓其他地方也可以使用，以下為一個例子：

```
// 改寫前面的內容 將兩個函式建立在這個函式內
// 同樣是依據傳入的布林值 返回不同的函式
func anotherChooseStepFunction(backwards: Bool) -> (Int) -> Int {
    func stepForward(input: Int) -> Int {
        return input + 1
    }

    func stepBackward(input: Int) -> Int {
        return input - 1
    }

    return backwards ? stepBackward : stepForward
}

let number2 = -35
let someFunction3 = anotherChooseStepFunction(number > 0)

someFunction3(10) // 返回 11
```

範例

本節範例程式碼放在 [ch1/functions.playground](#)

閉包

- 閉包表達式
- 尾隨閉包
- 捕獲值
- 閉包是參考型別
- 非逃逸閉包
- 自動閉包

閉包(closure)簡單來講就是一個匿名函式，同樣是一個獨立的程式區塊。像是前面章節提到的巢狀函式(nested function)就是一種閉包，可以在程式中被傳遞和使用。

閉包有三種表現方式：

- 函式就是一種有名稱的閉包。
- 巢狀函式就是一種有名稱且被包含在另一個函式中的閉包。
- 閉包表達式就是使用簡潔語法來描述的一種沒有名稱的函式，可以在程式中被傳遞和使用。

閉包表達式

閉包表達式(closure expression)是一種利用簡潔語法建立匿名函式的方式。同時也提供了一些優化語法，可以使得程式碼變得更好懂及直覺。閉包表達式的格式如下：

```
{ (參數) -> 返回值型別 in
    內部執行的程式
}
```

上述程式中可以看到，與函式相同是以大括號 {} 將程式包起來，但省略了名稱，包著參數的小括號 () 擺到 {} 裡並接著箭頭 -> 及返回值型別。然後使用 in 分隔內部執行的程式。

閉包裹達式可以使用常數、變數和 `inout` 型別作為參數，但不能有預設值。也可以在參數列表的最後使用可變數量參數(`variadic parameter`)。元組也可以作為參數和返回值。

下面是一個例子，從將一個函式當做另一個函式的參數開始進行，原始程式碼如下：

```
// 這是一個要當做參數的函式 功能為將兩個傳入的參數整數相加並返回
func addTwoInts(number1: Int, number2: Int) -> Int {
    return number1 + number2
}

// 建立另一個函式，有三個參數依序為
// 型別為 (Int, Int) -> Int 的函式, Int, Int
func printMathResult(
    mathFunction: (Int, Int) -> Int, _ a: Int, _ b: Int) {
    print("Result: \(mathFunction(a, b))")
}
```

這邊將函式 `addTwoInts()` 修改成一個匿名函式(即閉包)傳入，程式如下：

```
printMathResult({(number1: Int, number2: Int) -> Int in
    return number1 + number2
}, 12, 85)
// 印出：97

/* 第一個參數為一個匿名函式(閉包) 如下
{(number1: Int, number2: Int) -> Int in
    return number1 + number2
}
*/
```

上述程式中可以看到函式 `printMathResult()` 依舊為三個參數，第一個參數原本應該是一個函式，但可以簡化成一個閉包並直接傳入，其後為兩個 `Int` 的參數。

接下來會使用上面這個例子，介紹幾種優化語法的方式，越後面介紹的程式語法會越簡潔，但使用上功能是一樣的。

根據上下文推斷型別

因為兩數相加閉包是作為函式 `printMathResult()` 的參數傳入的，Swift 可以自動推斷其參數及返回值的型別(根據建立函式 `printMathResult()` 時的參數型別 `(Int, Int) -> Int`)，因此這個閉包的參數及返回值的型別都可以省略，同時包著參數的小括號 `()` 及返回箭頭 `->` 也可以省略，修改如下：

```
printMathResult(  
    {number1, number2 in return number1 + number2}, 12, 85)  
// 印出：97  
  
/* 第一個參數修改成如下  
{number1, number2 in return number1 + number2}  
*/
```

單表達式閉包隱式回傳

單行表達式閉包可以通過隱藏 `return` 來隱式回傳單行表達式的結果，修改如下：

```
printMathResult({number1, number2 in number1 + number2}, 12, 85)  
// 印出：97  
  
/* 第一個參數修改成如下  
{number1, number2 in number1 + number2}  
*/
```

參數名稱縮寫

Swift 自動為閉包提供參數名稱縮寫功能，可以直接以 `$0`，`$1`，`$2` 這種方式來依序呼叫閉包的參數。

如果使用了參數名稱縮寫，就可以省略在閉包參數列表中對其的定義，且對應參數名稱縮寫的型別會通過函式型別自動進行推斷，所以同時 `in` 也可以被省略，修改如下：

```
printMathResult({$0 + $1}, 12, 85)
// 印出：97

/* 第一個參數修改成如下
{$0 + $1}
*/
```

運算子函式

實際上還有一種更簡潔的語法。Swift 的 `String` 型別定義了關於加號 `+` 的字串實作，其作為一個函式接受兩個數值，並返回這兩個數值相加的值。而這正好與最開始的函式 `addTwoInts()` 相同，因此你可以簡單地傳入一個加號 `+`，Swift 會自動推斷出加號的字串函式實作，修改如下：

```
printMathResult(+, 12, 85)
// 印出：97

// 第一個參數修改成： +
```

以上介紹了四種優化閉包的語法，使用上功能都與最開始的閉包相同，所以可以依照需求以及合適性，使用不同的優化語法，來讓你的程式更簡潔與直覺，當然都使用完整寫法的閉包也是可以的。

尾隨閉包

如果需要將一個很長的閉包裹達式作為最後一個參數傳遞給函式，可以使用尾隨閉包(`trailing closure`)來增強函式的可讀性。尾隨閉包是一個書寫在函式括號`()`之後的閉包裹達式，函式支援將其作為最後一個參數呼叫。以下是一個例子：

```
// 這是一個參數為閉包的函式
func someFunction(closure: () -> ()) {
    // 內部執行的程式
}

// 內部參數名稱為 closure
// 閉包的型別為 () -> () 沒有參數也沒有返回值

// 不使用尾隨閉包進行函式呼叫
someFunction({
    // 閉包內的程式
})

// 可以看到這個閉包作為參數 是放在 () 裡面

// 使用尾隨閉包進行函式呼叫
someFunction() {
    // 閉包內的程式
}

// 可以看到這個閉包作為參數 位置在 () 後空一格接著寫下去
```

如果函式只有閉包這一個參數時，甚或是可以將函式的 () 省略，修改如下：

```
// 使用尾隨閉包進行函式呼叫 省略函式的 ()
someFunction {
    // 閉包內的程式
}
```

捕獲值

閉包可以在其定義的上下文中捕獲(capture)常數或變數，即使定義這些常數或變數的原使用區域已經不存在，閉包仍可以在閉包函式體內參考或修改這些值。

Swift 中，可以捕獲值的閉包的最簡單形式是巢狀函式，也就是定義在其他函式內的函式。巢狀函式可以捕獲並存取外部函式(把它定義在其中的函式)內所有的參數以及定義的常數與變數，即使這個巢狀函式已經回傳，導致常數或變數的作用範圍不存在，閉包仍能對這些已經捕獲的值做操作。

以下是一個例子：

```
// 定義一個函式 參數是一個整數 回傳是一個型別為 () -> Int 的閉包
func makeIncrementer(forIncrement amount: Int) -> () -> Int {
    // 用來儲存計數總數的變數
    var runningTotal = 0

    // 巢狀函式 簡單的將參數的數字加進計數並返回
    // runningTotal 和 amount 都被捕獲了
    func incrementer() -> Int {
        runningTotal += amount
        return runningTotal
    }

    // 返回捕獲變數參考的巢狀函式
    return incrementer
}
```

上述程式中可以看到，巢狀函式內部存取了的 `runningTotal` 及 `amount` 變數，是因為它從外部函式捕獲了這兩個變數的參考。而這個捕獲參考會讓 `runningTotal` 與 `amount` 在呼叫完 `makeIncrementer` 函式後不會消失，並且下次呼叫 `incrementer` 函式時，`runningTotal` 仍會存在。

以下是呼叫這個函式的例子：

```
// 宣告一個常數
// 會被指派為一個每次呼叫就會將 runningTotal 加 10 的函式 incrementer
let incrementByTen = makeIncrementer(forIncrement: 10)
// 呼叫多次 可以觀察到每次返回值都是累加上去
incrementByTen() // 10
incrementByTen() // 20
incrementByTen() // 30

// 如果另外再宣告一個常數
// 會有屬於它自己的一個全新獨立的 runningTotal 變數參考
// 與上面的常數無關
let incrementBySix = makeIncrementer(forIncrement: 6)
incrementBySix() // 6

// 第一個常數仍然是對它自己捕獲的變數做操作
incrementByTen() // 40
```

閉包是參考型別

前面的例子中，`incrementByTen` 和 `incrementBySix` 是常數，但這些常數指向的閉包仍可以增加其捕獲的變數值，這是因為函式與閉包都是參考型別。

參考型別就是無論將函式(或閉包)指派給一個常數或變數，實際上都是將常數或變數的值設置為對應這個函式(或閉包)的參考(參考其在記憶體空間內配置的位置)。

所以當你將閉包指派給了兩個不同的常數或變數，這兩個值都會指向同一個閉包(的參考)，如下：

```
// 指派給另一個常數
let alsoIncrementByTen = incrementByTen

// 仍然是對原本的 runningTotal 操作
alsoIncrementByTen() // 50
```

後面章節會正式介紹[值型別與參考型別](#)的不同。

非逃逸閉包

當一個閉包被當做參數傳入一個函式中，但是這個閉包在函式返回後才被執行(例如像是閉包被當做函式的返回值，然後接著被拿去做別的操作)，這樣稱作閉包從函式中逃逸(`escape`)。而如果要明確表示一個當做參數的閉包不能從函式中逃逸，要在參數前標註 `@noescape`，來表明這個閉包的生命週期只在這個函式體內。例子如下：

```
// 參數為一個閉包的函式 參數前面標註 @noescape
func someFunctionWithNoescapeClosure(
    @noescape closure: () -> Void) {
    // 而這個閉包的生命週期只在這個函式內
    closure()
}
```

而下面這個例子是說明一個逃逸的閉包：

```
// 宣告一個函式之外的變數 是一個陣列 陣列成員的型別為閉包 () -> Void
var completionHandlers: [() -> Void] = []

// 接著定義一個函式 參數為一個閉包 型別與上面陣列成員的型別一樣
func someFunctionWithEscapingClosure(
    completionHandler: () -> Void) {
    // 這個函式將閉包加入一個函式之外的陣列變數中
    completionHandlers.append(completionHandler)
}

// 如果將這個函式的參數前面加上 @noescape 的話會報錯誤 因為閉包逃逸了
```

另外還有一點，將閉包標註為 `@noescape`，可以讓你在閉包中隱式的參考 `self` (例如原本應該寫 `self.x` 的，可以簡化寫成 `x`，因為可以隱式參考 `self`，會自動推斷為 `self` 的 `x` 屬性)，以下利用到前面定義的兩個示範函式做例子：

```
// 定義一個類別
class SomeClass {
    var x = 10
    func doSomething() {
        // 使用到前面定義的兩個函式 都使用了尾隨閉包來讓語法更為簡潔
        // 傳入當參數的閉包 內部都是將實體的屬性指派為新的值
        someFunctionWithEscapingClosure { self.x = 100 }
        // 可以看到這個標註 @noescape 的參數的閉包
        // 其內可以隱式的參考 self
        someFunctionWithNoescapeClosure { x = 200 }
    }
}

// 生成一個實體
let instance = SomeClass()

// 呼叫其內的方法
instance.doSomething()
// 接著那兩個前面定義的函式都會被呼叫到 所以最後實體的屬性 x 為 200
print(instance.x)

// 接著呼叫陣列中的第一個成員
// 也就是示範逃逸閉包的函式中 會將閉包加入陣列的這個動作
// 而這個第一個成員就是 { self.x = 100 }
completionHandlers.first?()
// 所以這時實體的屬性 x 便為 100
print(instance.x)
```

後面章節會正式介紹類別。

自動閉包

自動閉包(autoclosure)是一種自動被建立的閉包，用於包裝後傳遞給函式作為參數的表達式。這種閉包沒有參數，而當被使用時，會返回被包裝在其內的表達式的值。

也就是說，自動閉包是一種簡化的語法，讓你可以用一個普通的表達式代替顯式的閉包，進而省略了閉包的大括號 {} 。

自動閉包讓你可以延遲求值，因為這個閉包會直到被你呼叫時才會執行其內的程式，以下先示範一個普通的閉包如何延遲求值：

```
// 首先宣告一個有五個成員的陣列
var customersInLine = ["Albee", "Alex", "Eddie", "Zack", "Kevin"]

// 印出：5
print(customersInLine.count)

// 接著宣告一個閉包 會移除掉陣列的第一個成員
let customerProvider = { customersInLine.removeAtIndex(0) }

// 這時仍然是印出：5
print(customersInLine.count)

// 直到這個閉包被呼叫時 才會執行
// 印出：開始移除 Albee !
print("開始移除 \(customerProvider() ) !")

// 這時就只剩下 4 個成員了
print(customersInLine.count)
```

上述程式可以看到閉包直到被呼叫時，才會移除成員，所以如果不呼叫閉包的話，則陣列成員都不會被移除。另外要注意一點，這個閉包 `customerProvider` 的型別為 `() -> String`，而不是 `String`。

將閉包作為參數傳遞給函式時，一樣可以延遲求值，如下：

```
// 這時 customersInLine 為 ["Alex", "Eddie", "Zack", "Kevin"]

// 定義一個閉包作為參數的函式
func serveCustomer(customerProvider: () -> String) {
    // 函式內部會呼叫這個閉包
    print("開始移除 \(customerProvider()) !")
}

// 呼叫函式時 [移除陣列第一個成員] 這個動作被當做閉包的內容
// 閉包被當做參數傳入函式
// 這時才會移除陣列第一個成員
serveCustomer( { customersInLine.removeAtIndex(0) } )
```

接著則介紹如何使用自動閉包完成上述一樣的動作。你必須在參數前面標註 `@autoclosure`，以表示這個參數可以是一個自動閉包的簡化寫法，這時就可以將該函式當做接受 `String` 型別參數的函式來呼叫。這個前面標註 `@autoclosure` 的參數會將自己轉換成一個閉包，如下：

```
// 這時 customersInLine 為 ["Eddie", "Zack", "Kevin"]

// 這個函式的參數前面標註了 @autoclosure
// 表示這參數可以是一個自動閉包的簡化寫法
func serveCustomer(@autoclosure customerProvider: () -> String)
{
    print("開始移除 \(customerProvider()) !")
}

// 因為函式的參數有標註 @autoclosure 這個參數可以不用大括號 {}
// 而僅僅只需要[移除第一個成員]這個表達式
// 而這個表達式會返回[被移除的成員的值]
serveCustomer(customersInLine.removeAtIndex(0))
```

自動閉包含有非逃逸閉包的特性，所以你如果想讓這個閉包可以逃逸，則必須標註為 `@autoclosure(escaping)`，以下是一個例子：

```

// 這時 customersInLine 為 ["Zack", "Kevin"]

// 宣告另一個變數 為一個陣列 其內成員的型別為 () -> String
var customerProviders: [() -> String] = []

// 定義一個函式 參數標註 @autoclosure(escaping)
// 表示參數是一個可逃逸自動閉包
func collectCustomerProviders(
    @autoclosure(escaping) customerProvider: () -> String) {
    // 函式內部的動作是將當做參數的這個閉包 再加入新的陣列中
    // 因為可逃逸 所以不會出錯
    customerProviders.append(customerProvider)
}

// 呼叫兩次函式
// 會將 customersInLine 剩餘的兩個成員都移除並轉加入新的陣列中
collectCustomerProviders(customersInLine.removeAtIndex(0))
collectCustomerProviders(customersInLine.removeAtIndex(0))

// 印出：獲得了 2 個成員
print("獲得了 \(customerProviders.count) 個成員")

// 最後將這兩個成員也從新陣列中移除
for customerProvider in customerProviders {
    print("開始移除 \(customerProvider()) !")
}
// 依序印出：
// 開始移除 Zack !
// 開始移除 Kevin !

```

範例

本節範例程式碼放在 [ch1/closures.playground](#)

Swift 進階

Swift 進階這章所有的內容都是在 playground 中進行，所以在每節開始前，請先行[建立一個 playground 檔案](#)，或是直接開啓範例程式(每節最後有提供範例檔案位置)。

這章開始進入到物件導向的內容，以下會依序介紹屬性、方法、繼承等等重要特性：

- [列舉](#)
- [類別及結構](#)
- [屬性](#)
- [方法](#)
- [下標](#)
- [繼承](#)
- [建構過程及解構過程](#)

接著下列是屬於 Swift 的特性，初次閱讀可能會不知道實際應用的地方，在往後的學習中遇到時可以再回來複習此特性的使用方式：

- [自動參考計數](#)
- [可選鍊](#)
- [錯誤處理](#)
- [型別轉換](#)
- [巢狀型別](#)
- [擴展](#)
- [協定](#)
- [泛型](#)
- [存取控制](#)

列舉

- 列舉語法
- 使用 Switch 語句匹配列舉值
- 相關值
- 原始值
- 遍迴列舉

列舉(enumeration)是可以讓你自定義一個型別的一組相關的值(例如表示學校有哪些教學科目或是購買過程中會出現的錯誤狀況)，使你可以在程式碼中以型別安全(type-safe)的方式來使用這些值。

列舉支援很多特性，例如計算型屬性(computed property)、實體方法(instance method)、定義建構器(initializer)、擴展(extension)及協定(protocol)，後面章節會正式介紹這些內容。

列舉語法

列舉使用 `enum` 關鍵字建立，並將列舉定義放在一組大括號 `{}` 內，格式如下：

```
enum 列舉的自定義型別 {  
    各列舉定義  
}
```

列舉使用 `case` 關鍵字定義成員值，例子如下：

```
//這是一個定義指南針四個方位的列舉
enum CompassPoint {
    case North
    case South
    case East
    case West
}

// 多個成員值可以寫在同一行 以逗號 , 隔開
// 這是一個定義太陽系八大行星的列舉
enum Planet {
    case Mercury, Venus, Earth, Mars, Jupiter, Saturn, Uranus, Neptune
}
```

每個列舉都定義了一個全新的型別。像 Swift 中其他型別一樣，列舉的名稱(如上述程式中的 `CompassPoint` 及 `Planet`)應該以一個大寫字母開頭且為單數。

定義完列舉後，接著將其指派給一個變數，與一般指派方式相同：

```
// 這邊使用上面定義過的指南針方位的列舉

// 型別為 CompassPoint 的一個變數 值為其列舉內的 West
var directionToHead = CompassPoint.West

// 這時已經可以自動推斷這個變數的型別為 CompassPoint
// 如果要再指派新的值 可以省略列舉的型別名稱
directionToHead = .North
```

使用 **Switch** 語句匹配列舉值

這邊同樣使用上面定義過的指南針方位的列舉。使用 `switch` 語句匹配單個列舉值：

```

directionToHead = .South
switch directionToHead {
    case .North:
        print("Lots of planets have a north")
    case .South:
        print("Watch out for penguins") // 這行會被印出
    case .East:
        print("Where the sun rises")
    case .West:
        print("Where the skies are blue")
}

```

相關值

列舉中的每個成員值，視需求可以在需要的時候，一併儲存自定義的一個或以上其他型別的相關值(`associated value`)。使用方法為在成員值後面加上小括號 `()`，並將相關值型別放在小括號內(就像使用元組 `tuple` 一樣)。往後在程式中將該列舉成員值指派給變數或常數時，這個(或這些)相關值才會被設置，且可以是不同的。

以下是一個例子，假設建立一個庫存追蹤系統，商品條碼可能會有 `UPC-A` 格式的一維碼，每一個 `UPC-A` 條碼是一組四個正整數的值，或是使用 `QR Code` 格式的二維碼，每一個 `QR Code` 條碼是一個最多為 2,953 字元的字串，依據這個條件建立的列舉如下：

```

enum Barcode {
    case UPCA(Int, Int, Int, Int)
    case QRCode(String)
}

```

上述程式的意思是：定義一個名稱為 `Barcode` 的列舉型別，有兩個成員值，一個成員值為 `UPCA`，夾帶著 `(Int, Int, Int, Int)` 型別的相關值，另一個成員值為 `QRCode`，夾帶著 `String` 型別的相關值。

這個定義沒有提供任何 `Int` 或 `String` 的實際的值，它只是定義了：當一個型別為 `Barcode` 的變數或常數等於 `Barcode.UPCA` 或 `Barcode.QRCode` 時，可以儲存的相關值的型別。

這時可以指派一個型別為 `Barcode` 的變數，例子如下：

```
// 指派 Barcode 型別 成員值為 UPCA
// 相關值為 (8, 85909, 51226, 3)
var productBarcode = Barcode.UPCA(8, 85909, 51226, 3)

// 如果要修改為儲存 QR Code 條碼
productBarcode = .QRCode("ABCDEFG")

// 這時 .UPCA(8, 85909, 51226, 3) 會被 .QRCode("ABCDEFG") 所取代
// 一個變數 同一時間只能儲存一個列舉的成員值(及其相關值)
```

在使用 `switch` 語句匹配列舉值時，可以把相關值取出作為常數(`let`)或變數(`var`)使用，例子如下：

```
switch productBarcode {
    case .UPCA(
        let numberSystem, let manufacturer, let product, let check):
        print("UPC-A: \(numberSystem), \(manufacturer),
              \(product), \(check).")
    case .QRCode(let productCode):
        print("QR Code: \(productCode).") // 會印出這行
}
```

如果相關值同樣都被取出作為常數(或變數)，可以改成只在成員名稱前加上 `let` (或 `var`)，來使程式更為簡潔，如下：

```
switch productBarcode {
    case let .UPCA(numberSystem, manufacturer, product, check):
        print("UPC-A: \(numberSystem), \(manufacturer), \(product),
              \(check).")
    case let .QRCode(productCode):
        print("QR Code: \(productCode).")
}
```

原始值

除了使用相關值的列舉，其內的成員值可以儲存不同型別的相關值。Swift 也提供列舉先設置原始值(`raw value`)來代替相關值，這些原始值的型別必須相同。使用方法為在列舉名稱後加上冒號 : 並接著原始值型別，例子如下：

```
enum WeekDay: Int {  
    case Monday = 1  
    case Tuesday = 2  
    case Wednesday = 3  
    case Thursday = 4  
    case Friday = 5  
    case Saturday = 6  
    case Sunday = 7  
  
}  
  
let today = WeekDay.Friday  
// 使用 rawValue 屬性來取得原始值  
// 印出：5  
print(today.rawValue)
```

Hint

- 原始值可以是字串、字元或者任何整數值或浮點數值。
- 每個原始值在它的列舉宣告中必須是唯一的。

原始值(`raw value`)跟相關值(`associated value`)是不同的。原始值在定義列舉時即被設置，對於一個特定的列舉成員，它的原始值始終是相同的。而相關值是在列舉成員被指派為一個變數(或常數)時才一併設置的值，列舉成員的相關值是可以不同的。

原始值的隱式指派

在使用原始值為整數型別的列舉時，可以不需要為每個成員設置原始值，Swift 會將每個成員的原始值依次遞增 1，這個特性稱為原始值的隱式指派(`implicitly assigned raw value`)。成員都沒有原始值時，則會將第一個成員的原始值設置為 0，再依序遞增 1，例子如下：

```
// 第一個成員有設置原始值 1，接著下去成員的原始值就是 2, 3, 4 這樣遞增下去
enum SomePlanet: Int {
    case Mercury=1, Venus, Earth, Mars, Jupiter, Saturn, Uranus, Neptune
}

let ourPlanet = SomePlanet.Earth

// 印出：3
print(ourPlanet.rawValue)
```

在使用原始值為字串型別的列舉時，可以不需要為每個成員設置原始值，將會直接將該成員值設置為原始值。例子如下：

```
enum AnotherCompassPoint: String {
    case North, South, East, West
}

let directionPoint = AnotherCompassPoint.East

// 印出：East
print(directionPoint.rawValue)
```

使用原始值初始化列舉實體

在定義列舉時，如果使用了原始值，則這個列舉會有一個初始化方法(`method`)，這個方法有一個名稱為 `rawValue` 的參數，其參數型別就是列舉原始值的型別，返回值為列舉成員或 `nil`。例子如下：

```
// 一個使用原始值的列舉 原始值依序是 1,2,3,4,5,6,7,8
enum OtherPlanet: Int {
    case Mercury=1, Venus, Earth, Mars, Jupiter, Saturn, Uranus, Neptune
}

let possiblePlanet = OtherPlanet(rawValue: 7)
// possiblePlanet 型別為 OtherPlanet? 值為 OtherPlanet.Uranus
```

不過不是所有傳入的 `Int` 參數都會有返回值，所以其實他是返回一個 `OtherPlanet?` 型別，也就是可選的 `OtherPlanet`。以下是一個返回 `nil` 的例子：

```
let positionToFind = 9
if let targetPlanet = OtherPlanet(rawValue: positionToFind) {
    switch targetPlanet {
        case .Earth:
            print("We are here !")
        default:
            print("Not Safe !")
    }
} else {
    print("No planet at position \(positionToFind)")
}
// 印出：No planet at position 9
```

上述程式先使用了一個可選綁定(optional binding)，使用原始值 `9` 來尋找是否有星球，但可以看到列舉 `OtherPlanet` 中沒有原始值為 `9` 的成員，所以會返回一個 `nil`，接著則執行 `else` 內部程式。

遞迴列舉

遞迴列舉(recursive enumeration)是一種列舉型別，它會有一個或多個列舉成員使用該列舉型別的實體作為相關值。如果要表示一個列舉成員可以遞迴，必須在成員前面加上 `indirect`，例子如下：

```
// 定義一個列舉
enum ArithmeticExpression {
    // 一個純數字成員
    case Number(Int)

    // 兩個成員 表示為加法及乘法運算 各自有兩個[列舉的實體]相關值
    indirect case Addition(ArithmeticExpression,
                           ArithmeticExpression)
    indirect case Multiplication(ArithmeticExpression,
                                  ArithmeticExpression)
}

// 或是你也可以把 indirect 加在 enum 前面
// 表示整個列舉都是可以遞迴的
indirect enum ArithmeticExpression {
    case Number(Int)
    case Addition(ArithmeticExpression, ArithmeticExpression)
    case Multiplication(ArithmeticExpression,
                         ArithmeticExpression)
}
```

接者使用一個遞迴函式來示範這個遞迴列舉：

```
func evaluate(expression: ArithmeticExpression) -> Int {  
    switch expression {  
        case .Number(let value):  
            return value  
        case .Addition(let left, let right):  
            return evaluate(left) + evaluate(right)  
        case .Multiplication(let left, let right):  
            return evaluate(left) * evaluate(right)  
    }  
}  
  
// 計算 (5 + 4) * 2  
let five = ArithmeticExpression.Number(5)  
let four = ArithmeticExpression.Number(4)  
let sum = ArithmeticExpression.Addition(five, four)  
let product = ArithmeticExpression.Multiplication(  
    sum, ArithmeticExpression.Number(2))  
  
// 印出：18  
print(evaluate(product))
```

上述程式可以看到，當函式的參數為純數字，則直接返回該數字的值。而如果是加法或乘法運算，則是分別計算兩個表達式的值後，再相加或相乘。

範例

本節範例程式碼放在 [ch2/enumerations.playground](#)

類別及結構

- 類別與結構的比較
- 類別與結構實體
- 取得屬性
- 結構型別的成員逐一建構器
- 值型別與參考型別
- 恒等運算子
- 選擇使用類別或結構

在前面章節介紹了函式，它是一段執行特定任務的獨立程式碼區塊，而更進階地，Swift 提供了兩個型別語法：類別(`class`)及結構(`structure`)，可以讓你將多個相關的函式及值儲存在內，以及更多的特性。

類別與結構的比較

類別及結構有很多相同的地方，如下：

- 屬性(`property`)：用於儲存值
- 方法(`method`)：用於提供功能
- 下標(`subscript`)：用於存取值
- 建構器(`initializer`)：用於生成初始化值
- 擴展(`extension`)：增加預設實作的功能
- 協定(`protocol`)：對某類別提供標準功能

與結構相比，類別還有以下的其他功能：

- 繼承(`inherit`)：類別可以繼承另一個類別的內容
- 解構器(`deinitializer`)允許一個類別實體釋放任何其所被分配的資源
- 型別轉換允許在執行時檢查和轉換一個類別實體的型別
- 參考計數允許對一個類別實體的多次參考

以上這些特性，會在後面章節陸續介紹。

定義一個類別及結構分別要使用 `class` 及 `struct` 關鍵字，並接著一組大括號 `{}`，格式如下：

```
class 類別名稱 {
    類別內的屬性、方法及其他可以定義在內的特性
}

struct 結構名稱 {
    結構內的屬性、方法及其他可以定義在內的特性
}
```

Hint

- 類別或結構內的變數或常數，會稱作屬性(`property`)。而類別或結構內的函式，會稱作方法(`method`)。
- 每次定義一個新的類別或結構時，實際上你是定義了一個新的 Swift 型別，所以在習慣上會以[大駝峰式命名法](#)來為類別與結構命名，以符合標準 Swift 型別的大寫命名風格(像是 `String` 、 `Int`)。相對地，使用[小駝峰式命名法](#)為屬性與方法命名(與常數、變數及函式相同)，以便與類別區分。

以下是定義結構與類別的例子：

```
struct CharacterStats {
    var hp = 0.0
    var mp = 0.0
}

class GameCharacter {
    var stats = CharacterStats()
    var attackSpeed = 1.0
    var name: String?
}
```

上述程式中，定義了一個名為 `CharacterStats` 的結構來描述一個遊戲角色的狀態，這個結構包含了兩個屬性(屬性即為儲存在類別或結構內的常數或變數)：`hp` 跟 `mp`，分別表示角色血量與法力的最大值。當這兩個屬性被初始化為 `0.0` 時，會被自動推斷為 `Double` 型別。

接著定義了一個名為 `GameCharacter` 的類別，描述一個遊戲角色的基本資訊，這個類別包含了三個屬性，第一個為角色的狀態，會被初始化為一個 `CharacterStats` 結構的實體，其後兩個屬性為攻速及名字，攻速有預設

值 `1.0`，名字則為一個可選型別 `String?`，會被自動指派為一個預設值 `nil`，表示一開始沒有 `name` 值。

類別與結構實體

定義類別與結構，就像是給了一個固定的規格，譬如說明一個遊戲角色會有血量、法力、攻速、名字這些資訊，但使用上，必須依照這個規格(即類別或結構)來實際建立一個角色，這個動作就是生成類別或結構的實體(**instance**)。

Hint

- 實體有些地方會翻譯成實例，同樣都是指 `instance`。

類別與結構都使用建構器語法來生成實體，建構器最簡單的形式就是在類別或結構名稱後面加上一個小括號 `()` 語法，這種方式建立的實體，其內的屬性都會被初始化為預設值，如下：

```
let someStats = CharacterStats()  
let someGameCharacter = GameCharacter()
```

上述程式中，`someStats` 就是一個 `CharacterStats` 結構的實體，`someGameCharacter` 則為一個 `GameCharacter` 類別的實體。

後面章節會正式介紹建構器。

可以看到生成實體的方式與函式相似，所以習慣上兩種命名方式會不一樣，一個使用大駝峰式命名法，另一個使用小駝峰式命名法，來作區別。

取得屬性

使用點語法(`dot syntax`)可以取得實體的屬性(`property`)。規則為實體名稱後加上一個點號 `.`，然後緊接著屬性名稱，如下：

```
// 這邊使用前面定義的 CharacterStats 結構 及生成的實體 someStats  
// 因為沒有初始值 所以會使用預設值 這邊會印出：someStats 血量最大值為0.0  
print("someStats 血量最大值為\("someStats.hp")")
```

也可以直接取得子屬性，像是 `someGameCharacter` 中 `stats` 屬性的 `hp` 屬性，如下：

```
// 這邊使用前面定義的 GameCharacter 類別  
// 及其生成的實體 someGameCharacter  
// 印出：血量最大值為0.0  
print("血量最大值為\(someGameCharacter.stats.hp)")
```

也可以將新的值指派給實體的屬性，如下：

```
// 將 someGameCharacter 的血量最大值指派為 500  
someGameCharacter.stats.hp = 500  
// 再次印出 即會變成：500.0  
print(someGameCharacter.stats.hp)
```

結構型別的成員逐一建構器

所有結構都有一個自動生成的成員逐一建構器(`memberwise initializer`)，當要生成一個結構的實體時，用來初始化實體內的屬性，如下：

```
// 這邊使用前面定義的 CharacterStats 結構 生成新的實體 someoneStats  
let someoneStats = CharacterStats(hp: 120, mp: 100)  
  
// 印出：120.0  
print(someoneStats.hp)
```

Hint

- 類別實體沒有成員逐一建構器這個功能。

值型別與參考型別

Swift 中以記憶體配置的方式不同來說，可以分為值型別(`value type`)與參考型別(`reference type`)。值型別會儲存實際的值，而參考型別只是儲存其在記憶體空間中配置的位置。

結構和列舉是值型別

值型別(`value type`)在被指派給一個變數、常數或被傳遞給一個函式時，實際上操作的是其拷貝(`copy`)。指派或傳遞後，兩者的值即各自獨立，不會互相影響。

在前面章節中，其實已經大量使用了值型別。實際上，在 `Swift` 中，所有的基本型別：整數、浮點數、布林值、字串、陣列和字典，都是值型別，並且背後都是以結構的形式實作。

結構和列舉也都是值型別，代表它們的實體及其內任何值型別的屬性，在被指派或傳遞時都會被複製。例子如下：

```
// 這邊使用前面定義的 CharacterStats 結構
var oneStats = CharacterStats(hp: 120, mp: 100)
var anotherStats = oneStats

// 這時修改 anotherStats 的 hp 屬性
anotherStats.hp = 300
// 可以看出來已經改變了 印出：300.0
print(anotherStats.hp)

// 但 oneStats 的屬性不會改變
// 仍然是被生成實體時的初始值 印出：120.0
print(oneStats.hp)
```

在 `oneStats` 被指派給 `anotherStats` 時，其實是將 `oneStats` 內的值進行拷貝(`copy`)，接著將拷貝的資料儲存給新的實體 `anotherStats`，兩者是完全獨立的實體，彼此不會互相影響。

Hint

- 雖然值型別的指派或傳遞是一個「拷貝」的動作，在程式碼中會大量的出現，但 `Swift` 在處理時只有在確實必要時才會執行實際的拷貝，系統會管理這部份的最優化性能，所以沒有必要為了性能特地避免指派值。

類別是參考型別

與值型別不同，參考型別(`reference type`)在被指派給一個變數、常數或被傳遞給一個函式時，操作的不是其拷貝，而是已存在的實體本身。例子如下：

```
// 這邊使用前面定義的 GameCharacter 類別
let archer = GameCharacter()
archer.attackSpeed = 1.5
archer.name = "弓箭手"

// 指派給一個新的常數
let superArcher = archer
// 並修改這個新實體的屬性 attackSpeed
superArcher.attackSpeed = 1.8

// 可以看到這邊印出的都為：1.8
print("archer 的攻速為 \(archer.attackSpeed)")
print("superArcher 的攻速為 \(superArcher.attackSpeed)")
```

從上述程式可以知道，`archer` 跟 `superArcher` 實際上是對同一個實體做操作。也就是同一個實體的兩個不同名稱。

有一點要注意的是，可以看到 `archer` 跟 `superArcher` 都被宣告為常數(使用 `let` 告訴)，但依然可以改變其內的屬性 `archer.attackSpeed` 跟 `superArcher.attackSpeed`，因為這兩個常數儲存的是一個 `GameCharacter` 類別實體的參考(參考其在記憶體空間內配置的位置)，而不是儲存這個實體，所以這個實體的 `attackSpeed` 屬性可以被修改，但不會修改到常數的值。

恆等運算子

因為類別是參考型別，可能有多個常數和變數在後台同時參考某一個類別實體，所以 Swift 提供了兩個恆等運算子，能夠判斷兩個變數或常數是不是參考同一個類別實體：

- 等價於 `==`
- 不等價於 `!=`

以下是一個例子：

```
// 使用前面宣告的兩個常數 archer 與 superArcher
if archer === superArcher {
    print("沒錯！！！是同一個類別實體")
}
```

Hint

- 請注意 `==` 與 `===` 的不同
 - 等價於(`==`)為判斷兩個變數或常數是不是參考同一個類別實體
 - 等於(`==`)為判斷兩個值是否相等

選擇使用類別或結構

類別與結構有很多相似的地方，要如何決定每個資料建構要定義為類別還是結構，可以參考以下原則：

- 需要封裝的資料量較少且較簡單。
- 在指派或傳遞這個實體時，有特別需求這個資料是會被拷貝而不是參考。
- 不需要去繼承另一個已存在型別的屬性或行為。

如果符合上述一條或以上原則，則建議定義為結構，其餘則是定義為類別。

所以實際上，大部分的自定義資料建構都應該使用類別。

範例

本節範例程式碼放在 [ch2/classes_structures.playground](#)

屬性

- 儲存屬性
- 計算屬性
- 屬性觀察器
- 型別屬性

屬性(`property`)為特定型別(類別、結構或列舉)的值，有以下幾種使用方式：

- 儲存屬性(`stored property`): 在實體內儲存常數或變數，可以用於類別及結構。
- 計算屬性(`computed property`): 在實體內計算一個值，可以用於類別、結構及列舉。
- 型別屬性(`type property`): 與前兩個不同，這是屬於型別本身的屬性。
- 屬性觀察器(`property observer`): 用來觀察屬性值的變化，並以此觸發一個自定義的操作。

儲存屬性

儲存屬性(`stored property`)就是一個儲存在特定型別(類別或結構)的常數或變數。可以在定義儲存屬性時指定預設值，也可以在建構過程中設置或修改儲存屬性的值，以下是個例子：

```
// 定義一個遊戲角色的血量與法力最大值
struct CharacterStats {
    // 指定一個預設值
    var hpValueMax: Double = 300
    let mpValueMax: Double
}

// 或是在建構時設置屬性的值
var oneStats = CharacterStats(hpValueMax: 500, mpValueMax: 120)

// 生成實體後也可以再修改屬性的值
oneStats.hpValueMax = 200

// 但因為 mpValueMax 為一個結構裡的常數屬性 所以不能修改常數
// 下面這行會報錯誤
oneStats.mpValueMax = 200
```

常數結構的儲存屬性

如果生成一個結構的實體並指派給一個常數，則無法修改這個實體的任何屬性，就算是結構裡的儲存屬性為變數也無法，例子如下：

```
// 這邊使用前面定義的 CharacterStats 結構
// 生成一個 CharacterStats 結構的實體 並指派給一個常數 someStats
let someStats = CharacterStats(hpValueMax: 900, mpValueMax: 150)

// 這個實體 someStats 為一個常數 所以即使 hpValue 為一個變數屬性
// 仍然不能修改這個值 這行會報錯誤
someStats.hpValue = 1200
```

前面章節有提到過，結構(`struct`)是屬於 **值型別**(`value type`)，所以當實體宣告為常數時，其內所有屬性也就都是常數而無法修改了。

而相對地，類別(`class`)是屬於 **參考型別**(`reference type`)，一個類別實體的常數，仍可以修改其內的屬性，因為這時候這個常數儲存的是參考(參考其在記憶體空間內配置的位置)，而不是儲存這個實體。

延遲儲存屬性

延遲儲存屬性(`lazy stored property`)是指當第一次被呼叫的時候才會計算其初始值的屬性。在屬性宣告前使用 `lazy` 來表示一個延遲儲存屬性。

Hint

- 延遲儲存屬性只能使用在變數，因為屬性的值在實體建構完成之前可能無法得到，而常數屬性在建構完成之前必須要有初始值。

使用延遲儲存屬性可以讓類別中如果需要大量計算才能初始化的屬性，在需要的時候才真的初始化它，以下是個例子：

```
// 首先定義一個類別 DataImporter
// 這個類別會導入外部檔案並執行一些操作 初始化可能會花費不少時間
class DataImporter {
    // 這邊簡化成一個檔案名稱 實際上可能會有很多操作
    var fileName = "data.txt"
}

// 接著定義另一個類別 DataManager
class DataManager {
    // 延遲儲存屬性
    lazy var importer = DataImporter()
    // 操作時需要用到的資料
    var data = [String]()

    // 簡化內部內容 可能還有許多操作資料的動作
}

// 生成一個類別 DataManager 的實體常數
let manager = DataManager()

// 添加一些資料
manager.data.append("Some data")
manager.data.append("Some more data")

// 到目前為止 manager 的 importer 都尚未被初始化
// 直到第一次使用這個屬性 才會被創建並初始化
print(manager.importer.fileName)
```

計算屬性

除了儲存屬性外，類別、結構和列舉還可以定義計算屬性(`computed property`)，計算屬性不直接儲存值，而是提供一個 `getter` (使用關鍵字 `get`)來存取值，及一個可選的 `setter` (使用關鍵字 `set`)來間接設置其他屬性的值。以下是個例子：

```
// 定義一個遊戲角色的狀態
class GameCharacter {
    // 血量初始值
    var hpValue: Double = 100

    // 防禦力初始值
    var defenceValue: Double = 300

    // 總防禦力的 getter 跟 setter
    var totalDefence: Double {
        get {
            // 總防禦力的算法是 防禦力加上 10% 血量
            return (defenceValue + 0.1 * hpValue)
        }
        set(levelUp) {
            // 升級時 會將血量及防禦力乘上一個倍數
            hpValue = hpValue * (1 + levelUp)
            defenceValue = defenceValue * (1 + levelUp)
        }
    }
}

// 生成一個類別 GameCharacter 的實體常數 oneChar
let oneChar = GameCharacter();

// 取得目前角色的總防禦力
// 印出：310.0
print(oneChar.totalDefence)

// 升級時 角色狀態各數值會乘上的倍數 0.05
oneChar.totalDefence = 0.05

// 則現在角色的血量與防禦力會變成 105 跟 315
// 印出：血量：105.0, 防禦力：315.0
print("血量：\$(oneChar.hpValue), 防禦力：\$(oneChar.defenceValue)")
```

簡化 **setter**

可以將 `setter` 簡化，省略掉傳入的參數時，Swift 會提供一個內建的參數名稱 `newValue`。以下將前面定義的類別 `GameCharacter` 中的 `setter` 簡化：

```
// 將原先的 levelUp 參數移除 這時會提供一個內建的參數名稱 newValue
set {
    // 升級時 會將血量及防禦力乘上一個倍數
    hpValue = hpValue * (1 + newValue)
    defenceValue = defenceValue * (1 + newValue)
}
```

唯讀計算屬性

計算屬性的 `setter` 是可選的，所以依照需求可以只寫 `getter`，這時可以將計算屬性簡化，以下修改自前面定義的類別 `GameCharacter`：

```
// 定義一個遊戲角色的狀態
class AnotherGameCharacter {
    // 血量初始值
    var hpValue: Double = 100

    // 防禦力初始值
    var defenceValue: Double = 300

    // 總防禦力只有 getter
    var totalDefence: Double {
        // 總防禦力的算法是 防禦力加上 10% 血量
        return (defenceValue + 0.1 * hpValue)
    }

}
```

上述程式中，因為計算屬性只有 `getter`，所以 `getter` 可以省略掉關鍵字 `get` 及大括號 `{}`。

屬性觀察器

屬性觀察器(`property observer`)會監控和回應屬性值的變化，每次屬性被設置新的值都會呼叫屬性觀察器。以下為兩個可以使用的屬性觀察器：

- `willSet`：在設置新的值之前呼叫，會將這個新的值當做一個常數參數傳入，如果不命名這個參數名稱時，會有一個內建的參數名稱 `newValue`。
- `didSet`：在新的值被設置之後立即呼叫，會將舊的屬性值當做參數傳入，這個參數可以自己命名，或直接使用內建的參數名稱 `oldValue`。

以下是一個例子：

```
// 定義一個遊戲角色的狀態
class SomeGameCharacter {
    // 血量初始值
    var hpValue: Double = 100 {
        willSet (hpChange) {
            // 改變血量前
            print("新的血量為\($0)")
        }
        didSet {
            // 改變血量後
            if oldValue > hpValue {
                // 原血量較高 所以是受到攻擊 損血
                print("我損血了！哦阿！")
            } else {
                print("我補血了！耶！")
            }
        }
    }
}

// 生成一個類別 SomeGameCharacter 的實體常數 oneChar
let anotherChar = SomeGameCharacter();

// 角色受到攻擊 血量降低
// 因為有 willSet 所以會印出：新的血量為90.0
oneChar.hpValue = 90

// 設置完新的血量後 因為有 didSet 所以會印出：我損血了！哦阿！
```

上述程式中的 `willSet` 有命名參數名稱 `hpChange`，所以其內是使用 `hpChange`，而 `didSet` 沒有命名參數名稱，所以其內是使用內建的參數名稱 `oldValue`。

Hint

- 如果屬性經由輸入輸出參數(`inout`)方式傳入函式，`willSet` 和 `didSet` 也一樣會被觸發。

型別屬性

型別屬性(`type property`)是屬於這個型別(類別、結構或列舉)的屬性，無論生成了多少這個型別的實體，型別屬性都只有唯一一份。

型別屬性使用於定義所有從這個型別生成的實體共享的資料。

Hint

- 儲存型的型別屬性一定要有預設值，因為型別本身沒有建構器，無法在初始化過程中設值給型別屬性。
- 儲存型的型別屬性是延遲初始化的，只有在第一次被呼叫時才會被初始化，所以不需要對其使用 `lazy`。

型別屬性是使用 `static` 關鍵字作宣告變數或常數。

在為類別宣告計算型的型別屬性時，依照需求可以改用關鍵字 `class` 來支持子類別對父類別的實作覆寫(`override`)，也就是在將一個類別**A**的計算型的型別屬性以 `class` 宣告後，之後新的類別**B**繼承這個類別**A**時，可以覆寫這個型別屬性。

設置儲存型跟計算型的型別屬性如下：

```

struct SomeStructure {
    static var storedTypeProperty = "Some value in structure."
    static var computedTypeProperty: Int {
        return 1
    }
}

enum SomeEnumeration {
    static var storedTypeProperty = "Some value in enumeration."
    static var computedTypeProperty: Int {
        return 6
    }
}

class SomeClass {
    static var storedTypeProperty = "Some value in class."
    static var computedTypeProperty: Int {
        return 27
    }
    class var overrideableComputedTypeProperty: Int {
        return 107
    }
}

```

上述程式中，計算型的型別屬性只有 `getter`，實際上可以定義有 `getter` 跟 `setter` 的計算型的型別屬性，使用方法與先前介紹的計算型屬性相同。

後面章節會正式介紹繼承(子類別與父類別的關係)。

存取或設置型別屬性的值

與實體的屬性一樣，型別屬性的存取也是使用點語法(`dot syntax`)，但是型別屬性是向型別本身存取和設置，而不是向實體，例子如下：

```
// 這邊使用前面定義的
// 結構 SomeStructure, 列舉 SomeEnumeration, 類別 SomeClass

// 印出: Some value in structure.
print(SomeStructure.storedTypeProperty)

// 設置一個型別屬性
SomeStructure.storedTypeProperty = "Another value."
// 印出: Another value.
print(SomeStructure.storedTypeProperty)

// 印出: 6
print(SomeEnumeration.computedTypeProperty)

// 印出: 27
print(SomeClass.computedTypeProperty)
```

上述程式可以看到，要使用一個型別屬性時，是不需要生成任何的實體就可以使用的。

範例

本節範例程式碼放在 [ch2/properties.playground](#)

方法

- 實體方法
- 型別方法

方法(`method`)為特定型別(類別、結構或列舉)的函式，可以分為兩種：

- 實體方法(`instance method`)：先需要生成一個特定型別(類別、結構或列舉)的實體，才能使用這個實體裡的方法。
- 型別方法(`type method`)：屬於特定型別(類別、結構或列舉)本身的方法。

實體方法

實體方法(`instance method`)是屬於一個特定型別(類別、結構或列舉)的實體，可以用來存取和設置實體屬性或是提供實體需要的功能。

實體方法的語法跟使用方式與前面章節提到的 **函式** 一樣，使用 `func` 關鍵字建立，並需要放在這個定義的特定型別(類別、結構或列舉)的大括號 `{}` 內，例子如下：

```
// 定義一個類別 Counter
class Counter {

    // 定義一個變數屬性 預設值為零
    var count = 0

    // 定義一個實體方法 會將變數屬性加一
    func increment() {
        count += 1
    }

    // 定義一個實體方法 會將變數屬性加上一個傳入的數字
    func incrementBy(amount: Int) {
        count += amount
    }

    // 定義一個實體方法 會將變數屬性歸零
    func reset() {
        count = 0
    }
}
```

上述程式中，定義了一個簡單的計數器的類別，如果要呼叫方法的話，也是像屬性一樣使用點語法(dot syntax)，如下：

```
// 生成一個實體 counter 其內的計數值屬性初始化為 0
let counter = Counter()

// 呼叫其內的一個實體方法 現在計數值為 1
counter.increment()

// 呼叫其內的一個實體方法， 傳入一個參數 9， 現在計數值為 10
counter.incrementBy(9)

// 呼叫其內的一個實體方法， 將計數值歸零， 現在計數值為 0
counter.reset()
```

內建屬性 **self**

每一個實體都有一個內建的隱藏屬性 `self`，來代表這個實體本身。可以在實體方法中使用 `self` 來代表這個實體，如下：

```
// 將前面定義的類別 Counter 裡的 increment 方法作修改
func increment() {
    self.count += 1
}
```

不過實際上，`self` 不會常使用到，因為在一個方法中使用一個已知的屬性或方法時，Swift 都會將其當做目前這個實體的屬性或方法。

而有一個情況是，當一個實體方法的參數名稱與實體的一個屬性名稱相同時(像是方法的參數名稱為 `count`，而屬性也命名為 `count`)，這時在這個方法中使用 `count` 的話，都會將其當做參數，除非加上 `self` 為 `self.count`，才會當做是實體的屬性。

在實體方法中修改值型別

一般情況下，一個值型別(結構或列舉)實體的屬性，不能在它的實體方法中被修改。但如果有特殊需求需要修改屬性，可以使用變異(`mutating`)這個方法。要使用變異方法，將關鍵字 `mutating` 放在方法的 `func` 之前就可以了，如下：

```
// 定義一個結構 Point
struct Point {
    // 兩個變數屬性 可以代表一個二維圖上的一個點
    var x = 0.0, y = 0.0

    // 一個變異方法 會將兩個屬性各別加上一個值
    mutating func moveByX(deltaX: Double, y deltaY: Double) {
        x += deltaX
        y += deltaY
    }
}

// 生成一個結構的實體的變數 並給初始值
var somePoint = Point(x: 1.0, y: 1.0)

// 修改其內的屬性值
somePoint.moveByX(2.0, y: 3.0)

// 現在兩個屬性已經被改變了
// 印出：x: 3.0, y: 4.0
print("x: \(somePoint.x), y: \(somePoint.y)")
```

要注意一點，上述程式中如果生成實體的是一個常數，則無法使用變異方法來改變屬性，即使屬性是變數也不行。

使用變異方法指派給 **self** 值

除了修改屬性，變異方法還可以指派 `self` 屬性一個全新的實體，方法結束後，會將原先實體替換成這個新的實體。如下：

```
// 將前面定義的結構 Point 改寫成這樣
struct AnotherPoint {
    var x = 0.0, y = 0.0
    mutating func moveByX(deltaX: Double, y deltaY: Double) {
        self = AnotherPoint(x: x + deltaX, y: y + deltaY)
    }
}
```

另外在列舉中使用變異方法，可以把 `self` 設置為相同的列舉型別中不同的成員，如下：

```
// 定義一個三態開關的列舉
enum TriStateSwitch {
    // 列舉的三個成員
    case Off, Low, High

    // 變異方法 會在三個成員中依序切換
    mutating func next() {
        switch self {
        case Off:
            self = Low
        case Low:
            self = High
        case High:
            self = Off
        }
    }
}

// 宣告一個列舉的變數 且目前 ovenLight 為 TriStateSwitch.Low
var ovenLight = TriStateSwitch.Low

// 每次呼叫這個變異方法 都會在三個成員中依序切換
ovenLight.next()
// 現在 ovenLight 為 .High
ovenLight.next()
// 現在 ovenLight 為 .Off
```

型別方法

型別方法(`type method`)為定義在特定型別(類別、結構與列舉)上的方法。不是屬於實體，而是屬於這個型別(類別、結構或列舉)本身的方法，與實體方法一樣使用點語法(`dot syntax`)呼叫。

宣告型別方法時需要在 `func` 前加上關鍵字 `static` 。而以類別來說，還可以將 `static` 替換成 `class` 來允許子類別覆寫(`override`)父類別的類別方法。

例子如下：

```
// 定義一個類別
class SomeClass {
    // 定義一個型別方法
    class func someTypeMethod() {
        print("型別方法")
    }
}

// 呼叫一個型別方法
SomeClass.someTypeMethod()
```

上述程式中可以看到，不用生成實體就可以直接呼叫型別方法，因為型別方法是屬於一個特定型別(類別、結構與列舉)而不是一個實體。

後面章節會正式介紹繼承(子類別與父類別的關係)。

範例

本節範例程式碼放在 [ch2/methods.playground](#)

下標

- 下標語法
- 下標用法

下標(`subscript`)是一個可以快速存取及設置值的方式，單看中文字面上可能不太清楚什麼意思，其實在前面章節介紹陣列(`array`)及字典(`dictionary`)時已經有接觸過，在陣列或字典名稱後面緊接著中括號`[]`，括號內填入陣列的索引值(`index`)或是字典的鍵(`key`)，即可存取或設置值，以下是個例子：

```
// 宣告一個 [Int] 型別的陣列
var arr = [1, 2, 3, 4, 5, 6, 7]

// 印出其內第三個元素(請記得 陣列的索引值是從 0 開始算起)
print(arr[2])

// 修改第四個元素為 12
arr[3] = 12

// 宣告一個 [String:String] 型別的字典
var dict = ["name": "Kevin", "city": "Taipei"]

// 印出鍵為 name 的值
print(dict["name"])

// 修改鍵為 city 的值為 New York
dict["city"] = "New York"
```

下標語法

下標也可以定義在類別(`class`)、結構(`structure`)及列舉(`enumeration`)中，在其內的定義方式如下：

```
subscript(索引值: 索引值型別) -> 返回值型別 {
    get {
        // 存取值的程式操作
        // 必須返回一個前面定義的型別的值
    }
    set(新設置值的名稱) {
        // 設置值的程式操作
    }
}
```

與前面章節介紹的計算型屬性相似，可以定義為讀寫(`getter` 及 `setter`)，也可以定義為唯讀(`getter`)。 `setter` 傳入的設置值名稱也可以省略，省略時會有一個內建預設的名稱 `newValue` 。這兩種方式定義如下：

```
// 當下標為唯讀時 可以將 get 關鍵字及大括號 {} 省略掉 如下
subscript(索引值: 索引值型別) -> 返回值型別 {
    // 存取值的程式操作
    // 必須返回一個前面定義的型別的值
}

// 省略 setter 傳入的設置值名稱 會有一個內建預設名稱 newValue
subscript(索引值: 索引值型別) -> 返回值型別 {
    get {
    }
    set{
        // 設置值的程式操作
        // 可使用 newValue 來操作
    }
}
```

下標用法

以下為在一個類別中定義下標的例子：

```
// 定義一個簡單數學算式功能的類別
class SimpleMath {
```

```

// 一個數字屬性預設值
var num = 500

// 定義一個下標 為乘法
subscript(times: String) -> Int {
    get {
        // 其內可以使用這個索引值
        print(times)

        // 會返回 數字屬性乘以 2 的數值
        return num * 2
    }
    set {
        // 將數字屬性乘以新的倍數
        num *= newValue
    }
}

// 定義另一個下標 為除法
subscript(divided: Int) -> Int {
    return num / divided
}

// 宣告一個類別 SimpleMath 的常數
let oneMath = SimpleMath()

// 印出：1000
print(oneMath["simple"])

// 傳入值為 3， 會將數字屬性乘以 3
oneMath["star"] = 3

// 這個下標例子中的字串索引值沒有被使用到
// 實際是可以依照傳入的索引值 來做不同的需求及返回值

// 印出：1500
print(oneMath.num)

// 使用到另一個下標 索引值型別為 Int

```

```
// 印出：15  
print(oneMath[100])
```

下標可以定義多個索引值，可以是任意型別的參數或可變數量參數，返回值也可以是任意型別，但不能使用輸入輸出參數(`inout`)，也不能給參數設置預設值。

如上述程式，類別或結構可以定義多個下標，使用下標時會依據索引值數量及型別，自動推斷使用合適的下標。

範例

本節範例程式碼放在 [ch2/subscripts.playground](#)

繼承

- 基礎類別
- 生成子類別
- 覆寫
- 防止覆寫

繼承(`inherit`)這個概念主要是類別(`class`)在使用，是物件導向的一個重要特性。一個類別可以繼承另一個類別的屬性(`property`)、方法(`method`)及其他特性。

一個類別繼承其他類別時，這個類別會被稱為子類別(`subclass`)。被繼承的類別則是稱為父類別(`superclass`)，直翻會是超類別，但還是以父類別為主，以與子類別相對應)。

基礎類別

基礎類別(`base class`)就是不繼承於其它類別的類別。

Swift 中沒有一個通用的基礎類別，只要一個類別沒有繼承於其他類別，這個類別即為一個基礎類別。

以下先定義一個基礎類別，其實就是一個普通的類別：

```
// 定義一個遊戲角色職業通用的類別
class GameCharacter {
    // 攻擊速度
    var attackSpeed = 1.5

    // 這個職業的敘述
    var description: String {
        return "職業敘述"
    }

    // 執行攻擊的動作
    func attack() {
        // 無任何動作 有待子類別實作
    }
}

// 生成一個類別 GameCharacter 的實體
let oneChar = GameCharacter()

// 印出：職業敘述
print(oneChar.description)
```

上述程式為一個基礎的遊戲角色職業類別，僅是定義了一些通用的內容，接著需要再定義子類別來完備。

生成子類別

生成子類別(subclassing)指的是基於一個基礎類別來定義一個新的類別，子類別會繼承父類別所有的特性，且還可以增加新的特性。

使用方式為在類別名稱後面加上冒號 : ，接著寫上父類別名稱，如下：

```
class 子類別: 父類別 {
    子類別的定義內容
}
```

以下是個例子，這邊定義一個繼承自類別 GameCharacter 的新類別 Archer :

```

class Archer: GameCharacter {
    // 新增一個屬性 攻擊範圍
    var attackRange = 2.5
}

// 父類別所有的特性都一併繼承下來
let oneArcher = Archer()
// 可以直接以點語法來存取或設置一個父類別中定義過的屬性
oneArcher.attackSpeed = 1.8

```

一個子類別仍然可以再被其他類別繼承，以下再定義一個類別 Hunter，繼承自類別 Archer：

```

class Hunter: Archer {
    // 新增一個方法 必殺技攻擊
    func fatalBlow() {
        print("施放必殺技攻擊！")
    }
}

let oneHunter = Hunter()
// 這個類別一樣可以使用 Archer 及 GameCharacter 定義過的屬性及方法
print("攻擊速度為 \(oneHunter.attackSpeed)")
print("攻擊範圍為 \(oneHunter.attackRange)")
// 當然自己新增的方法也可以使用
oneHunter.fatalBlow()

```

覆寫

類別繼承的同時，子類別可以重新定義父類別中定義過的特性，如實體方法(`instance method`)、型別方法(`type method`)、實體屬性(`instance property`)、型別屬性(`type property`)或下標(`subscript`)，這種行為即是覆寫(`overriding`)。

使用關鍵字 `override` 來表示你要覆寫這個特性(即方法、屬性或下標)。

覆寫方法

以下為一個覆寫方法的例子：

```
// 使用並改寫前面定義的類別 Hunter
class OtherHunter: Archer {
    // 覆寫父類別的實體方法
    override func attack() {
        print("攻擊！這是獵人的攻擊！")
    }

    // 省略其他內容
}

let otherHunter = OtherHunter()
otherHunter.attack()
// 即會印出覆寫後的內容：攻擊！這是獵人的攻擊！
```

覆寫屬性

覆寫屬性時，需要使用 `getter` (以及有時可省略的 `setter`)來覆寫繼承來的屬性，且一定要寫上屬性的名稱及型別，這樣才能確定是從哪一個屬性繼承而來的。

Hint

- 可以將一個繼承來的唯讀屬性覆寫為一個讀寫屬性，但不行將一個讀寫屬性覆寫為唯獨屬性。即原本有 `setter` 的話，覆寫時就一定要有 `setter`。

以下是一個例子：

```
// 使用並改寫前面定義的類別 Hunter
class AnotherHunter: Archer {
    // 覆寫父類別的屬性 重新實作 getter 跟 setter
    override var attackSpeed: Double {
        get {
            return 2.4
        }
        set {
            print(newValue)
        }
    }

    // 省略其他內容
}
```

覆寫屬性觀察器

覆寫屬性時，通常可以加上屬性觀察器(`property observer`)，但以下兩點要注意：

- 當繼承的屬性為常數儲存型屬性或唯讀計算型屬性時，不能加上屬性觀察器，因為這兩者的屬性無法再被設置，所以 `willSet` 跟 `didSet` 對它們沒有意義。
- 覆寫時不能同時有 `setter` 跟屬性觀察器(`willSet` 跟 `didSet`)，因為 `setter` 中即可做到屬性觀察器的功能要求。

雖然說是覆寫，但如果覆寫的父類別屬性也有屬性觀察器，其實子類別跟父類別兩者的屬性觀察器都會被執行，例子如下：

```

// 使用並改寫前面定義的類別 Archer
class OtherArcher: GameCharacter {
    // 覆寫一個屬性 重新實作 getter 跟 setter
    override var attackSpeed: Double {
        willSet {
            print("OtherArcher willSet")
        }
        didSet {
            print("OtherArcher didSet")
        }
    }
}

// 省略其他內容
}

// 使用並改寫前面定義的類別 Hunter
class SomeHunter: OtherArcher {
    // 覆寫一個屬性 重新實作 getter 跟 setter
    override var attackSpeed: Double {
        willSet {
            print("SomeHunter willSet")
        }
        didSet {
            print("SomeHunter didSet")
        }
    }
}

// 省略其他內容
}

let someHunter = SomeHunter()
// 設置新的值 會觸發 willSet 跟 didSet
someHunter.attackSpeed = 1.8
// 依序會印出：
// SomeHunter willSet
// OtherArcher willSet
// OtherArcher didSet
// SomeHunter didSet

```

上述程式中可以知道，`willSet` 觸發時，會先執行子類別的再來才是父類別的，而`didSet` 則是相反，先執行父類別的再來才是子類別的。

存取父類別的屬性、方法及下標

在前面章節介紹類別的時候，提到類別有一個隱藏的內建屬性`self`，可以在方法中代表類別本身。而當繼承自另一個類別時，可以使用`super` 屬性來存取父類別的屬性、方法或下標。

- 方法`someMethod()` 的覆寫實作裡，使用`super.someMethod()` 來呼叫父類別的`someMethod()` 方法。
- 屬性`someProperty` 的`getter` 及`setter` 覆寫實作裡，使用`super.someProperty` 來存取父類別的`someProperty` 屬性。
- 下標的覆寫實作裡，使用`super[someIndex]` 來存取父類別的下標。

以下為一個例子：

```
// 使用並改寫前面定義的類別 Hunter
class GoodHunter: Archer {
    // 覆寫一個屬性 並使用父類別原先的屬性值
    override var description: String {
        return super.description + " 精通箭術的獵人"
    }

    // 省略其他內容
}

let goodHunter = GoodHunter()

// 印出：職業敘述 精通箭術的獵人
print(goodHunter.description)
```

防止覆寫

可以在類別的方法、屬性或下標前面加上`final`，來防止它們被覆寫，使用方式為`final var`、`final func` 或是`final class func` 這樣。

甚至也可以在整個類別的關鍵字 `class` 前面加上 `final`，這樣整個類別都不能再被繼承。

範例

本節範例程式碼放在 [ch2/inheritance.playground](#)

建構過程及解構過程

- 建構器
- 設置儲存屬性的初始值
- 為建構器提供參數
- 結構的成員逐一建構器
- 值型別的建構器委任
- 類別的繼承與建構過程
- 可失敗的建構器
- 必要建構器
- 解構器

建構過程(`initialization`)就是要生成一個類別、結構或列舉的實體時進行初始化的過程，這個過程必須為實體中每個屬性設置初始值及其他視需求執行的程式。

與此相對的，解構過程(`deinitialization`)則是在類別實體被釋放前，執行特定的清除工作。

建構器

建構過程則是透過建構器(`initializer`)實作，也就是 `init()` 方法，最簡單的一個形式如下：

```
init() {  
    執行的建構過程  
}
```

以下是一個例子：

```
// 定義一個類別 SomeClass 並在建構器中指派初始值給屬性 number
class SomeClass {
    let number: Int

    init() {
        number = 12
    }
}
```

建構器不是一定要寫，如果屬性都已經有預設值，且沒有任何自定義的建構器，Swift 會自動提供一個預設建構器(default initializer)，在建構過程中，這個預設建構器會簡單地生成一個所有屬性都設置為預設值的實體。

設置儲存屬性的初始值

在建構過程結束前，這個類別的儲存屬性都必須被指派一個明確的值，可以是定義時直接指派，或是在建構器中為屬性指派。如下：

```
class SomeClass2 {
    let number: Int = 20
    let anotherNumber: Int

    init() {
        anotherNumber = 12
    }
}
```

上述程式可以看到，有一個屬性是定義時即指派值，另一個屬性則是在建構器中被指派。

可選型別的屬性

如果一個儲存屬性依照需求或規劃，在建構過程中沒有辦法被指派或是需要在之後可以被設置為 nil，可以將其定義為可選型別(optional type)，這樣它會被初始化為 nil，則建構過程中可以不用被指派值，例子如下：

```
class OneQuestion {  
    var question: String  
  
    // 可選型別 即可在建構過程時不用指派值  
    var answer: String?  
  
    init() {  
        // 僅指派一個型別為 String 的屬性  
        question = "問題的題目"  
    }  
}  
  
let someQuestion = OneQuestion()  
// 這時才將 answer 指派值  
someQuestion.answer = "答案隨後跟上"
```

使用閉包或函式設置屬性的預設值

可以使用閉包或全域函式來為屬性提供預設值，以下是一個閉包的例子：

```
class SomeClass {  
    let numbers: [Int] = {  
        var temporaryNumbers = [Int]()  
        var isBlack = false  
        for i in 1...10 {  
            temporaryNumbers.append(i)  
        }  
        return temporaryNumbers  
    }()  
}
```

上述程式可以看到閉包後緊接著一對小括號 ()，表示要立即執行這個閉包並返回閉包的值。

為建構器提供參數

建構器可以加入參數，使用方式與函式跟方法類似，例子如下：

```
struct SimpleMath {  
    var number: Double  
    init(huge n: Double) {  
        number = n * 100  
    }  
    init(tiny n: Double) {  
        number = n / 10  
    }  
}  
  
let oneSimpleMath = SimpleMath(huge: 30.0)  
// 印出 3000.0  
print(oneSimpleMath.number)  
  
let anotherSimpleMath = SimpleMath(tiny: 10.0)  
// 印出 1.0  
print(anotherSimpleMath.number)
```

上述程式為一個結構，可以看到建構器可以不只有一個，由參數的不同來辨別要使用哪一個建構器。

與函式跟方法不一樣的是，因為建構器都叫做 `init()`，為了辨識，所以不管有幾個參數，外部參數名稱預設都是必須的(函式跟方法的第一個參數預設是省略外部參數名稱)。

內部參數名稱及外部參數名稱

與函式跟方法一樣，可以使用其內部參數名稱作為外部參數名稱，如下：

```
// 定義一個結構 有兩個建構器
struct Color {
    let red, green, blue: Double

    // 這個建構器有寫外部參數名稱跟內部參數名稱
    init(red r: Double, green g : Double, blue b: Double) {
        self.red    = r
        self.green = g
        self.blue   = b
    }

    // 這個建構器則是合併成一個參數名稱 外部跟內部參數名稱相同
    init(white: Double) {
        red    = white
        green = white
        blue   = white
    }
}

var oneColor = Color(red: 0.9, green: 0.5, blue: 0.5)
var anotherColor = Color(white: 1.0)
```

省略外部參數名稱

與函式跟方法一樣，可以省略其外部參數名稱，使用下底線 `_` 替代外部參數名稱，如下：

```
struct SomeNumbers {
    let number: Int
    // 使用下底線 _ 表示要省略外部參數名稱
    init(_ n: Int) {
        number = n
    }
}

// 生成一個實體時 參數前就不需要有外部參數名稱
var oneNumbers = SomeNumbers(9)
```

結構的成員逐一建構器

前面章節有提到，當結構沒有自定義的建構器時，會自動生成一個成員逐一建構器，以下是一個例子：

```
struct CharacterStats {  
    var hp = 0.0  
    var mp = 0.0  
}  
  
let someoneStats = CharacterStats(hp: 120, mp: 100)
```

值型別的建構器委任

建構器委任(`initializer delegation`)指的是建構器可以呼叫其他建構器來完成生成實體時的部份建構過程，可以整合及減少多個建構器間的程式碼重複。

值型別(結構及列舉)沒有繼承這個特性，所以建構器委任相對簡單，值型別(結構及列舉)只能委任本身的其他建構器。

在自定義的建構器中使用 `self.init` 來委任(也就是呼叫)其他建構器，需要注意只能在建構器內使用 `self.init`，以下是個例子：

```
// 定義兩個示範需要用到的結構
struct Size {
    var width = 0.0, height = 0.0
}
struct Point {
    var x = 0.0, y = 0.0
}

// 定義一個方形的結構 Rect
struct Rect {
    // 使用上面兩個定義的結構來儲存這個方形的原點及尺寸
    var origin = Point()
    var size = Size()

    // 三個建構器
    init() {}
    init(origin: Point, size: Size) {
        self.origin = origin
        self.size = size
    }
    init(center: Point, size: Size) {
        let originX = center.x - (size.width / 2)
        let originY = center.y - (size.height / 2)
        self.init(
            origin: Point(x: originX, y: originY), size: size)
    }
}
```

上述程式中可以看到結構 `Rect` 有三個建構器，以下會依序生成由這三個建構器初始化的實體。

使用第一個建構器 `init()`，這是預設建構器，其內沒有設置任何屬性，所以生成的這個實體的屬性皆是使用預設值，如下：

```
let basicRect = Rect()
// basicRect 內的屬性的值分別為
// origin 為 (0.0, 0.0)
// size 為 (0.0, 0.0)
```

使用第二個建構器 `init(origin:size:)`，簡單的將屬性指派為新的值，如下：

```
let originRect = Rect(origin: Point(x: 2.0, y: 2.0),
    size: Size(width: 5.0, height: 5.0))
// originRect 內的屬性的值分別為
// origin 為 (2.0, 2.0)
// size 為 (5.0, 5.0)
```

使用第三個建構器 `init(center:size:)`，第一個參數是這個方形的中心點座標，這個建構器會先利用中心點與尺寸的長寬來算出原點的位置，再委任(也就是呼叫)另一個建構器 `init(origin:size:)` 來為屬性指派新的值，如下：

```
let centerRect = Rect(center: Point(x: 4.0, y: 4.0),
    size: Size(width: 3.0, height: 3.0))
// centerRect 內的屬性的值分別為
// origin 為 (2.5, 2.5)
// size 為 (3.0, 3.0)
```

類別的繼承與建構過程

類別可以繼承其他的類別(當然也包含屬性)，為了確保在類別的建構過程中，儲存屬性(包含本身的及繼承自父類別的)都設置了初始值，Swift 提供了兩種建構器，分別是指定建構器和便利建構器。

指定建構器與便利建構器

指定建構器(**designated initializer**)是類別中最主要的建構器，負責在初始化時給所有無預設值的屬性指派一個值，還需要負責委任(也就是呼叫)父類別的建構器來完成父類別的初始化，每個類別至少要有一個指定建構器。

便利建構器(**convenience initializer**)是輔助型的建構器，可以委任類別本身其他的建構器，最後必須以委任一個指定建構器結束。便利建構器不是一定需要，你可以依需求定義便利建構器，來使得生成實體時可以更明確或更快速的知道這個建構器的目的。

指定建構器的格式如下：

```
init(參數) {  
    執行的建構過程  
}
```

便利建構器的格式是在 `init` 前面加上 `convenience` 關鍵字，如下：

```
convenience init(參數) {  
    執行的建構過程  
}
```

類別的建構器委任

類別的指定建構器與便利建構器委任關係規則如下：

1. 便利建構器的建構過程中，必須委任類別本身中的另一個建構器(可以是指定建構器或便利建構器)。
2. 便利建構器可以一直委任另一個便利建構器(一個接著一個)，但最後必須要委任一個指定建構器。
3. 指定建構器必須要委任其父類別的指定建構器(如果有父類別的話)。

一個簡單的記憶方法為：

- 便利建構器必須横向委任。
- 指定建構器必須向上委任。

建構器的繼承與覆寫

`Swift` 的類別預設不會繼承父類別的建構器，在有需求時可以手動覆寫。

- 覆寫父類別的指定建構器時，必須在建構器(不論覆寫成爲指定建構器或便利建構器)前面加上關鍵字 `override`。
- 覆寫父類別的便利建構器時，前面則不需要加上 `override`，直接重新定義即可。

建構器的自動繼承

前面提到類別預設不會繼承父類別的建構器，但在以下兩個規則且此類別的屬性都有預設值時，建構器會自動繼承：

- 子類別沒有定義任何指定建構器，則會自動繼承父類別所有的指定建構器。
- 子類別實作了父類別所有的指定建構器(不論是上述規則 1 來的，或是自己定義實作的)，則會自動繼承父類別所有的便利建構器。

指定建構器與便利建構器的示範

以下例子會依序定義三個類別 `GameCharacter` 、 `Archer` 跟 `Hunter` ，用來示範指定建構器、便利建構器與建構器的自動繼承。

首先定義一個基礎類別 `GameCharacter` ，如下：

```
class GameCharacter {
    var name: String
    init(name: String) {
        self.name = name
    }
    convenience init() {
        self.init(name: "[未命名]")
    }
}
```

上述程式中，類別 `GameCharacter` 有兩個建構器：

- `init(name: String)` 為一個指定建構器，確保所有儲存屬性都設置到值。
- `init()` 為一個沒有參數的便利建構器，但建構過程中會委任類別中另一個指定建構器 `init(name: String)` ，並將一個值作為參數傳入。

以下為使用不同建構器生成的實體：

```
// 使用指定建構器 生成實體後的屬性 name 為: Kevin
let oneChar = GameCharacter(name:"Kevin")

// 使用便利建構器 生成實體後的屬性 name 為: [未命名]
let anotherChar = GameCharacter()
```

接著定義一個繼承自 `GameCharacter` 的類別 `Archer`：

```

class Archer: GameCharacter {
    var attackRange: Double
    init(name: String, attackRange: Double) {
        self.attackRange = attackRange
        super.init(name: name)
    }
    override convenience init(name: String) {
        self.init(name: name, attackRange: 1)
    }
}

```

上述程式中，類別 `Archer` 新增了一個屬性 `attackRange`，且有兩個建構器：

- `init(name: String, attackRange: Double)` 為一個指定建構器。
 - 先將傳入的 `attackRange` 指派給新增的屬性，接著會向上委任父類別的建構器 `init(name: String)`。
 - 這邊要注意，類別本身的屬性都有設置初始值之後，才能向上委任父類別的建構器，讓父類別繼續進行它自己屬性的設置初始值。
- `init(name: String)` 為一個便利建構器。
 - 簡單的傳入一個參數 `name`，並設置一個固定的值給 `attackRange`，最後委任本身的指定建構器 `init(name: String, attackRange: Double)`，完成指派值給屬性的工作。
 - 可以觀察得到，這是覆寫父類別的指定建構器，所以前面必須加上 `override` 關鍵字。
 - 這個便利建構器的定義可以讓生成實體更為簡潔，當需要生成多個實體時可以避免程式碼的冗餘。

因為類別 `Archer` 實作了其父類別 `GameCharacter` 所有的指定建構器，所以自動繼承了 `GameCharacter` 所有的便利建構器。以下為使用不同建構器(也包含繼承自父類別的建構器)生成的實體：

```
// 繼承自父類別的建構器
let oneArcher = Archer()

// 覆寫自父類別並重新定義的建構器
let secondArcher = Archer(name: "Joe")

// 類別本身自己定義的建構器
let anotherArcher = Archer(name: "Adam", attackRange: 2.4)
```

最後定義一個繼承自 `Archer` 的類別 `Hunter`，新增了兩個屬性 `hp` 跟 `description`：

```
class Hunter: Archer {
    var hp = 100
    var description: String {
        return "\(name)，基礎血量為 \(hp)"
    }
}
```

上述程式可以看到，類別 `Hunter` 新增的兩個屬性都有預設值，且自己沒有定義任何建構器，所以它會自動繼承父類別的所有指定建構器跟便利建構器。可以使用所有繼承來的建構器來生成實體：

```
let oneHunter = Hunter()
let secondHunter = Hunter(name: "Black")
let anotherHunter = Hunter(name: "Dwight", attackRange: 3)
```

可失敗的建構器

類別、結構或是列舉在建構過程中可能失敗，這個失敗可能是傳入無效的參數、缺少某種外部需要的資源或是沒有滿足某種必要條件。

為了處理這種可能失敗的情況，可以定義一個可失敗建構器(`failable initializer`)，使用方式為在 `init` 後面加上一個問號 `? : init?`。

Hint

- 可失敗建構器的參數名稱及型別，不能與其他非可失敗建構器相同。
- 嚴格來說，建構器都沒有返回值，但當必須表示一個建構器在建構過程中失敗時，會以 `return nil` 來表示。

以下是一個例子：

```
// 定義一個結構 Animal 當傳入的參數為空字串時 建構過程會失敗
struct Animal {
    let name: String
    init?(name: String) {
        if name.isEmpty { return nil }
        self.name = name
    }
}

// 傳入 Lion 當參數
var oneAnimal = Animal(name: "Lion")
if let one = oneAnimal {
    print("動物的名字為 \(one.name)")
}

// 傳入一個空字串當參數 (請注意 空字串與 nil 完全不一樣)
var anotherAnimal = Animal(name: "")
if anotherAnimal == nil {
    print("沒有傳入名字 所以建構過程中失敗了")
}
```

列舉型別的可失敗建構器

可以定義一個帶一個或多個參數的可失敗建構器，來取得列舉型別中特定的成員，當參數無法匹配任何成員時，則為建構失敗。以下是個例子：

```

enum TemperatureUnit {
    case Kelvin, Celsius, Fahrenheit
    init?(symbol: Character) {
        switch symbol {
        case "K":
            self = .Kelvin
        case "C":
            self = .Celsius
        case "F":
            self = .Fahrenheit
        default:
            return nil
        }
    }
}

```

上述程式是定義一個溫度單位的列舉，當失敗建構器的參數為 K 、 C 或 F 時，可以匹配到成員，則建構成功。相反地，輸入其他參數則不會匹配成功，則稱為建構失敗，則會返回 nil ，如下：

```

let oneUnit = TemperatureUnit(symbol: "F")
if oneUnit != nil {
    print("這是一個溫度單位")
}

let anotherUnit = TemperatureUnit(symbol: "X")
if anotherUnit == nil {
    print("這不是一個溫度單位")
}

```

如果不為列舉定義一個可失敗建構器，其本身會自動建立一個帶有參數的可失敗建構器 init?(rawValue:) ，這個參數名稱 rawValue 是固定的，其型別與列舉成員原始值的型別相同。所以可以此來簡化上面定義的 TemperatureUnit 列舉，如下：

```
enum AnotherTemperatureUnit: Character {  
    case Kelvin = "K", Celsius = "C", Fahrenheit = "F"  
}  
  
// 可以匹配到成員的原始值 所以建構成功  
let oneUnit2 = AnotherTemperatureUnit(rawValue: "F")  
  
// 無法匹配到成員的原始值 所以建構失敗  
let anotherUnit2 = AnotherTemperatureUnit(rawValue: "X")
```

建構失敗的傳遞

可失敗建構器的委任關係及規則如下：

- 類別、結構或列舉的可失敗建構器可以橫向委任到同一個類別、結構或列舉裡的另一個可失敗建構器。
- 類別的可失敗建構器可以向上委任到父類別的可失敗建構器。
- 可失敗建構器可以委任到其他非可失敗建構器。可用來為已定義好的建構過程，增加可失敗的條件。
- 只要委任傳遞過程中，遇到一個建構器失敗時，則整個建構過程會立即返回失敗，之後的程式碼都不會再執行。

以下是一個例子：

```
// 定義一個類別 AnotherGameCharacter 有一個可失敗建構器
// 當名稱參數為空字串時會建構失敗
class AnotherGameCharacter {
    let name: String
    init?(name: String) {
        if name.isEmpty { return nil }
        self.name = name
    }
}

// 定義一個繼承自 AnotherGameCharacter 的類別 AnotherArcher
// 有一個可失敗建構器 當攻速參數小於 1 時會建構失敗
class AnotherArcher: AnotherGameCharacter {
    let attackSpeed: Int
    init?(name: String, attackSpeed: Int) {
        if attackSpeed < 1 { return nil }
        self.attackSpeed = attackSpeed
        super.init(name: name)
    }
}

// 作為參數的名稱跟攻速都符合規則 建構成功
// 會生成一個 AnotherArcher 的實體
let oneArcher2 = AnotherArcher(name: "Jim", attackSpeed: 2)

// 作為參數的攻速為 0 會建構失敗
// 在 AnotherArcher 中即返回 nil
// 不會再向上傳遞至父類別 AnotherGameCharacter
let anotherArcher2 = AnotherArcher(name: "Zack", attackSpeed: 0)

// 作為參數的名稱為空字串 會建構失敗
// 建構過程一直到父類別 AnotherGameCharacter 的建構器 才會失敗
let finalArcher = AnotherArcher(name: "", attackSpeed: 1)
```

覆寫一個可失敗建構器

你可以在類別中覆寫父類別的可失敗建構器，可覆寫成為可失敗建構器或非可失敗建構器。以下是個例子：

Hint

- 不能將一個父類別的非可失敗建構器，覆寫成爲可失敗建構器。

```
//定義一個類別 Document
class Document {
    // 可選型別的屬性
    var name: String?
    // 使用這個建構器 會生成一個屬性 name 為 nil 的實體
    init() {}
    // 使用這個建構器 會生成一個屬性 name 不爲空字串的實體
    // 或是建構失敗 返回 nil
    init?(name: String) {
        self.name = name
        if name.isEmpty { return nil }
    }
}

// 定義一個繼承自 Document 的類別 AutomaticallyNamedDocument
class AutomaticallyNamedDocument: Document {
    // 覆寫父類別的建構器 會指派值給屬性
    override init() {
        super.init()
        self.name = "[未命名]"
    }
    // 覆寫父類別的可失敗建構器 成爲 非可失敗建構器
    // 可以看到他將條件修改成爲 不會有失敗的狀況發生
    override init(name: String) {
        super.init()
        if name.isEmpty {
            self.name = "[未命名]"
        } else {
            self.name = name
        }
    }
}
```

當你用非可失敗建構器覆寫一個父類別的可失敗建構器，又需要向上委任到父類別的這個可失敗建構器時，必須強制解析這個父類別建構器，在 `super.init()` 後面加上一個驚嘆號！，表示這時建構過程一定會成功，不會返回 `nil`，如下：

```
// 定義一個繼承自 Document 的類別 UntitledDocument
class UntitledDocument: Document {
    // 覆寫一個父類別的可失敗建構器 並向上委任到這個建構器
    override init() {
        // 這時必須強制解析這個父類別的建構器
        // 表示不會有失敗的狀況
        super.init(name: "[未命名]")!
    }
}
```

就如同前面章節提過的，變數或常數的可選型別(`optional type`)及隱式解析可選型別(`implicitly unwrapped optional`)的關係，可失敗建構器也可以將問號？改為驚嘆號！，定義成 `init!`，可以生成一個隱式解析可選型別的實體。

必要建構器

在類別的建構器前加上 `required`，表示所有繼承這個類別的子類別，都必須實作這個建構器：

```
class SomeClass4 {
    required init() {
        // 建構器執行程式的實作
    }
}
```

繼承這個類別的子類別，定義這個建構器時，前面同樣需要加上 `required`(不需要 `override`)：

```
class SomeSubclass: SomeClass4 {  
    required init() {  
        // 必要建構器執行程式的實作  
    }  
}
```

解構器

與建構過程相對的，在一個類別的實體不再被需要使用時，Swift 會自動將其釋放掉，在釋放前會先進行解構過程，使用解構器(`deinitializer`)來實作，也就是 `deinit` 方法。

每個定義的類別中，只能有一個解構器，解構器沒有任何參數，也不需要寫小括號 `()`，格式如下：

```
deinit {  
    執行的解構過程  
}
```

Swift 有一個自動參考計數(ARC)的機制，會處理實體的記憶體管理，所以大部分的情況下，不需要手動清除，交給 Swift 來自動處理就好。

後面章節會正式介紹[自動參考計數\(ARC\)](#)

範例

本節範例程式碼放在 [ch2initialization_deinitialization.playground](#)

自動參考計數

- 自動參考計數的運作方式
- 類別實體間的強參考循環
- 解決實體間的強參考循環
- 閉包的強參考循環
- 解決閉包的強參考循環

Swift 使用自動參考計數(ARC , Automatic Reference Counting)機制來追蹤與管理記憶體使用狀況，所以大部分情況下，你不需要自己管理，Swift 會自動釋放掉不需要的記憶體。

Hint

- 參考計數只應用在類別(也就是參考型別)的實體。結構與列舉為值型別，也不是通過參考的方式儲存與傳遞。

自動參考計數的運作方式

當一個類別實體被指派值(給一個屬性、常數或變數)的時候，會建立一個該實體的強參考(strong reference)，同時會將參考計數(reference counting)加 1 ，強參考表示會將這個實體保留下，只要強參考還在(也就是參考計數不為 0)，儲存這個實體的記憶體就不會被釋放掉。

以下簡單介紹一下 **ARC** 運作的方式：

```
// 定義一個類別 SomePerson
class SomePerson {
    let name: String
    init (name: String) {
        self.name = name
    }
}

// 先宣告三個可選 SomePerson 的變數 會被自動初始化為 nil
// 這三個變數目前都尚未有實體的參考
var reference1: SomePerson?
```

```

var reference2: SomePerson?
var reference3: SomePerson?

// 先生成一個實體 並指派給其中一個變數 reference1
reference1 = SomePerson(name: "Jess")

// 目前這個實體就有了一個強參考 參考計數為 1
// 所以 ARC 會保留下這個實體使用的記憶體

// 接著再指派給另外兩個變數
reference2 = reference1
reference3 = reference1

// 這時這個實體多了 2 個強參考 總共為 3 個強參考
// 也就是目前的參考計數為 3

// 接著將其中兩個變數指派為 nil 斷開他們的強參考
reference1 = nil
reference2 = nil

// 目前還有 1 個強參考 參考計數為 1
// 所以 ARC 仍然會保留下記憶體

// 最後將第三個變數也指派為 nil 斷開強參考
reference3 = nil

// 這時這個實體已經沒有強參考了 參考計數為 0
// ARC 就會將記憶體釋放掉

```

類別實體間的強參考循環

ARC 在大部分時間都可以運作順利，但在有些情況下會造成強參考永遠不會歸零，進而發生記憶體洩漏(memory leak)的問題。

以下是一個例子，兩個類別彼此都擁有對方強參考的屬性，一個實體要釋放記憶體前，必須先釋放對方強參考，而對方要釋放前也是要原本實體先釋放，進而產生強參考循環：

```
// 定義一個類別 Person
// 有一個屬性為可選 Apartment 型別 因為人不一定住在公寓內
class Person {
    let name: String
    init(name: String) { self.name = name }
    var apartment: Apartment?
}

// 定義一個類別 Apartment
// 有一個屬性為可選 Person 型別 因為公寓不一定有住戶
class Apartment {
    let unit: String
    init(unit: String) { self.unit = unit }
    var tenant: Person?
}

// 宣告一個變數為可選 Person 型別 並生成一個實體
var joe: Person? = Person(name: "Joe")
// 生成實體後 其內的 apartment 屬性沒有指派值 初始值為 nil
// 目前這個實體的強參考有 1 個 參考計數為 1

// 宣告一個變數為可選 Apartment 型別 並生成一個實體
var oneUnit: Apartment? = Apartment(unit: "5A")
// 生成實體後 其內的 tenant 屬性沒有指派值 初始值為 nil
// 目前這個實體的強參考有 1 個 參考計數為 1
```

接著把這兩個不同的實體聯繫起來，如下：

```
joe!.apartment = oneUnit
oneUnit!.tenant = joe
// 這時這兩個實體 各別都有 2 個強參考

//如果此時將 2 個變數斷開強參考
joe = nil
oneUnit = nil

// 這時這 2 個實體 各別仍還是有 1 個指向對方的強參考
// 也就造成記憶體無法釋放
```

Hint

- 前面章節提過，上述程式中，在實體後的驚嘆號(!)指的是將一個可選型別強制解析，也就是隱式解析可選型別。

解決實體間的強參考循環

Swift 提供了兩種辦法來解決強參考循環，分別是弱參考(weak reference)及無主參考(unowned reference)。

這兩種參考也能參考實體，但因為不是強參考，所以不會保留下實體的參考(也就是這個實體的參考計數不會增加)。

而兩者的差別在於，如果一個參考這個實體的變數在生命週期中，可能會為 nil 時，就使用弱參考，而在初始化之後不會再變為 nil 的則是使用無主參考。

弱參考

弱參考(weak reference)也能參考實體，但不會保留下參考的實體(所以這個實體的參考計數不會增加)。而一個參考這個實體的變數在生命週期中可能沒有值(為 nil)時，就使用弱參考。

弱參考必須宣告為變數，表示可以被修改，同時也必須是可選型別(optional)，因為可能沒有值(為 nil)。

弱參考使用 weak 關鍵字來定義，以下將前面強參考循環的例子改寫，將類別 Apartment 內的屬性 tenant 改為弱參考(因為公寓可能有時沒有住戶，即有時會沒有值，適合使用弱參考)：

```

class AnotherPerson {
    let name: String
    init(name: String) { self.name = name }
    var apartment: AnotherApartment?
}

class AnotherApartment {
    let unit: String
    init(unit: String) { self.unit = unit }

    // 將這個屬性定義為弱參考 使用 weak 關鍵字
    weak var tenant: AnotherPerson?
}

var joe2: AnotherPerson? = AnotherPerson(name: "Joe")
var oneUnit2: AnotherApartment? = AnotherApartment(unit: "5A")
joe2!.apartment = oneUnit2

// 因為是弱參考
// 所以這個指派為實體的屬性 不會增加 joe2 參考的實體的參考計數
oneUnit2!.tenant = joe2

// 當斷開這個變數的強參考時 目前該實體的參考計數會減為 0
// 所以會將這個實體釋放
// 而所有指向這個實體的弱參考 都會被設為 nil
joe2 = nil

// 隨著上面的 joe2 被釋放
// 目前 oneUnit2 參考的實體的參考計數減為 1
// 以下再將原本的強參考斷開 參考計數減為 0 則也會將此實體釋放
oneUnit2 = nil

```

無主參考

與弱參考一樣，無主參考(unowned reference)不會保留下參考的實體(所以這個實體的參考計數不會增加)，但不同的是，無主參考會被視為永遠有值，所以需要被定義為非可選型別，而因此可以直接存取，不需要強制解析(即加上驚嘆號 !)。

無主參考使用 `unowned` 關鍵字來定義，以下例子將介紹一個使用者類別 `Customer` 與信用卡類別 `CreditCard` 之間的關係，使用者不一定有信用卡，但當產生出信用卡時，這信用卡一定屬於某個使用者：

```
// 定義一個類別 Customer
class Customer {
    let name: String
    var card: CreditCard?
    init(name: String) {
        self.name = name
    }
}

// 定義一個類別 CreditCard
class CreditCard {
    let number: Int

    // 定義一個無主參考 非可選型別 因為一定會有使用者(一定有值)
    unowned let customer: Customer

    init(number: Int, customer: Customer) {
        self.number = number
        self.customer = customer
    }
}

// 宣告一個可選 Customer 的變數
var jess: Customer? = Customer(name: "Jess")

// 接著生成一個 CreditCard 實體並指派給 jess 的 card 屬性
jess!.card = CreditCard(number: 123456789, customer: jess!)
// 這個 CreditCard 實體的 customer 屬性 則使用無主參考指向 jess

// 現在 jess 指向的實體 參考計數為 1 (即 jess 這個變數強參考指向的)
// jess 內的屬性 card 指向的實體 參考計數也為 1
// (即這個 card 屬性強參考指向的)

// 而 CreditCard 實體的 customer 屬性 因為是無主參考指向 jess
// 所以不會增加參考計數
```

```
// 這時將 jess 指向的實體強參考斷開  
jess = nil  
// 這時這個實體的參考計數為 0 則實體會被釋放  
// 指向 CreditCard 實體的強參考也會隨之斷開  
// 因此也就被釋放了
```

無主參考和隱式解析可選型別

除了上述兩種情況，還有一種情況為，兩個互相參考實體的屬性都必須有值，且初始化後永遠不為 `nil`，這時要在其中一個類別使用無主參考，另一個類別使用隱式解析可選型別。

當初始化完成後，這兩個屬性都能被直接存取(不需要強制解析，即不用加上驚嘆號`!`)，且也避免了強參考循環。

以下的例子分別定義了類別 `Country` 及類別 `City`，每個類別都有一個儲存對方類別實體的屬性，也就是每個國家(`Country`)都有一個首都(`capitalCity`)，而一個城市(`City`)必須屬於一個國家(`country`)：

```

class Country {
    let name: String

    // 定義為 隱式解析可選型別
    var capitalCity: City!

    init(name: String, capitalName: String) {
        self.name = name
        self.capitalCity=City(name:capitalName,country:self)
    }
}

class City {
    let name: String

    // 定義為 無主參考
    unowned let country: Country

    init(name: String, country: Country) {
        self.name = name
        self.country = country
    }
}

var country = Country(name: "Japan", capitalName: "Tokyo")

```

上述程式中，為了建立兩個類別的依賴關係，`City` 的建構函式有一個 `Country` 實體的參數，並儲存為 `country` 屬性。`Country` 的建構函式則呼叫了 `City` 的建構函式。

以 `Country` 來說：

1. 在建構器中要使用 `self` 代表自己本身，必須要在自己初始化完成後才行。而 `Country` 的建構函式中，在 `name` 設置完值後就完成了初始化，所以可以將 `self` (即本身)當做參數傳給 `City` 的建構函式。
2. 而因為 `Country` 將屬性 `capitalCity` 定義為隱式解析可選型別，所以不需要使用可選綁定(`optional binding`)或強制解析(即在後面加驚嘆號`!`)去參考 `City`，可以直接存取。

以 `City` 來說：

1. `City` 在 `country` 屬性使用了無主參考，因為一個城市一定屬於一個國家，所以一定有值，且不會增加 `Country` 實體的參考計數。(其實就如同前面例子 `Customer` 之於 `CreditCard` 的關係)

以上的意義在於可以通過一條語句同時生成 `Country` 跟 `City` 的實體，而不會產生強參考循環，並且 `capitalCity` 屬性可以被直接存取，不需要被強制解析(即加上驚嘆號 !)。

閉包的強參考循環

除了前面提到的類別實體之間可能產生強參考循環，當將一個閉包(`closure`，也就是匿名函式)設置給一個類別實體的屬性時，這個閉包函式內存取了這個實體的某個屬性，或是呼叫了實體的一個方法，都會導致閉包捕獲(`capture`)了 `self`，進而產生了強參考循環。

Hint

- 閉包所捕獲的參考會被自動視為強參考。

這個強參考循環的產生，是因為閉包也是參考型別，當把閉包設置給一個屬性時，實際上是設置了閉包的參考。

以下是一個閉包與類別實體的強參考循環的例子：

```

// 定義一個代表 HTML 元素的類別 HTMLElement
class HTMLElement {
    let name: String
    let text: String?

    // 定義為 lazy 屬性 表示只有當初始化完成以及 self 實際存在後
    // 才能存取這個屬性
    lazy var asHTML: Void -> String = {
        if let text = self.text {
            return "<\(self.name)>\(text)</\(self.name)>"
        } else {
            return "<\(self.name) />"
        }
    }

    init(name: String, text: String? = nil) {
        self.name = name
        self.text = text
    }
}

// 告知為可選 HTMLElement 型別 以便後面設為 nil
var paragraph: HTMLElement?
= HTMLElement(name: "p", text: "Hello, world")

// 初始化完成後 就可以存取這個屬性
print(paragraph!.asHTML())

// 這時 paragraph 指向的實體的參考計數為 2
// 一個是自己 一個是閉包

// 而這個實體也有一個強參考指向閉包

// 這時將變數指向的強參考斷開 參考計數減為 1
// 參考仍然不會被釋放 造成強參考循環
paragraph = nil

```

Hint

- 閉包中雖然多次使用了 `self`，但只捕獲了 1 個強參考(也就是參考計數只算 1 次)

解決閉包的強參考循環

在定義閉包時，同時定義捕獲列表(`capture list`)作為閉包的一部分，通過這種方式可以解決閉包和類別實體之間的強參考循環。捕獲列表中必須定義每個閉包中捕獲的參考為弱參考或無主參考(依其相互關係來決定)。

定義捕獲列表

捕獲列表(`capture list`)中每一項都以 `weak` 或 `unowned` 關鍵字與類別實體的參考(如 `self`)或初始化過的變數(如 `delegate = self.delegate!`)成對組成，每一項以逗號 , 隔開，並寫在中括號 [] 內。

以下為捕獲列表的格式：

```
// 如果閉包有參數及返回型別 則將捕獲列表寫在他們前面
lazy var someClosure: (Int, String) -> String = {
    [unowned self, weak delegate = self.delegate!]
    (index: Int, stringToProcess: String) -> String in
    // 閉包內執行的程式
}

// 或是省略閉包定義的參數或返回型別 讓他們可以通過上下文自動推斷
// 這時將捕獲列表放在關鍵字 in 的前面
lazy var someClosure: Void -> String = {
    [unowned self, weak delegate = self.delegate!] in
    // 閉包內執行的程式
}
```

以下則是將前面的例子 `HTMLElement` 中的閉包加上捕獲列表，便可以避免強參考循環：

```
class NewHTMLElement {
    let name: String
    let text: String?

    lazy var asHTML: Void -> String = {
        // 這邊使用無主參考 unowned
        [unowned self] in
        if let text = self.text {
            return "<\(self.name)>\(text)</\(self.name)>"
        } else {
            return "<\(self.name) />"
        }
    }

    init(name: String, text: String? = nil) {
        self.name = name
        self.text = text
    }
}
```

範例

本節範例程式碼放在 [ch2/arc.playground](#)

可選鏈

- 以可選鏈替代強制解析
- 通過可選鏈存取屬性, 下標及呼叫方法

可選鏈(optional chaining)是一個可以存取或呼叫屬性(property)、方法(method)及下標(subscript)的過程。

稱其為可選(optional)是因為當前存取或呼叫的目標可能為空(nil)，而多次存取或呼叫可以用點語法(dot syntax 即 .)將其全部鏈結在一起，所以稱為鏈(chaining)。

可選鏈中，只要其中一個節點為空(nil)，則會立即返回 nil 。相對地，如果存取或呼叫至最後一個節點都有值，則會返回一個可選型別的值。

以可選鏈替代強制解析

可選鏈中，如果其中一個節點為可選值時，必須在這個值後面加上問號(?)，來表示這是一個可選值，也就是當這個值為空(nil)時，會立即返回 nil 。

與強制解析(forced unwrapping)在值後面加上驚嘆號(!)不同，強制解析的值如果為空(nil)時，會觸發程式運行錯誤。而可選鏈中的可選值(加上問號 ?)如果為空(nil)，則只會單純的返回 nil 。

不論可選鏈最後返回的屬性、方法或下標是不是可選型別的值，都是返回一個可選值，所以可以用來判斷這個可選鏈是否成功，有返回值是成功，返回 nil 則是失敗。像是原本返回應該是一個 Int 型別的值，經由可選鏈後返回會是一個 Int? 可選型別的值。以下是一個例子：

```
// 定義一個類別 Person
class Person {
    // residence 屬性為可選的 Residence 型別
    var residence: Residence?
}

// 定義一個類別 Residence
class Residence {
    // numberOfRooms 屬性為 Int 型別
    var numberOfRooms = 1
}

// 首先生成一個實體
let joe = Person()

// 這時 joe 實體內的可選屬性 residence 沒有設置值 所以會初始化為 nil
// 如果強制解析的話 會發生錯誤 如以下這行
//let roomCount = joe.residence!.numberOfRooms

// 所以這時可使用可選鍊來呼叫
if let roomCount = joe.residence?.numberOfRooms {
    print("Joe 有 \(roomCount) 間房間")
} else {
    print("無法取得房間數量")
}

// 上述 if 語句目前會印出：無法取得房間數量
// 接著為 joe.residence 設置一個 Residence 實體
joe.residence = Residence()

// 這時就會返回 1
// 記得這是返回一個可選型別 Int?
print(joe.residence?.numberOfRooms)
```

通過可選鍊存取屬性, 下標及呼叫方法

這邊定義三個類別以供後續示範，沿用先前的類別 `Person`，並新增一個類別 `Room`，最後將 `Residence` 重新定義，如下：

```

// 與先前的類別 Person 相同
class NewPerson {
    var residence: NewResidence?
}

// 定義一個類別 Room
class Room {
    // name 屬性為 String 型別
    let name: String

    // 生成實體時同時設置屬性 name 的值
    init(name: String) { self.name = name }
}

// 重新定義類別 Residence
class NewResidence {
    // rooms 屬性為一個陣列 型別為 [Room] 預設值是空陣列
    var rooms = [Room]()

    // 將 numberOfRooms 改為一個計算屬性 並返回 rooms 的數量
    var numberOfRooms: Int {
        return rooms.count
    }

    // 新增一個下標 可以依索引值取得 rooms 陣列內的值
    // 或是依索引值修改 rooms 陣列內的值
    subscript(i: Int) -> Room {
        get {
            return rooms[i]
        }
        set {
            rooms[i] = newValue
        }
    }

    // 新增一個方法 簡單的印出房間數量
    func printNumberOfRooms() {
        print("有 \(numberOfRooms) 間房間")
    }
}

```

```
}
```

存取屬性

前面的例子就是示範存取屬性，還要提的一點是，你也可以嘗試使用可選鏈來給屬性指派值，如下：

```
// 使用上面生成的實體 joe 設置新的值
joe.residence?.numberOfRooms = 12
```

何以說嘗試指派值，因為如果 `joe.residence` 尚未有值(為 `nil`)時就將值指派給他，這時不會發生任何事，因為 `joe.residence` 仍為 `nil`，必須直到他有一個 `Residence` 的實體後，才能正確指派值。

呼叫方法

前面定義的類別中可以看到 `printNumberOfRooms()` 方法沒有返回值，前面章節提過，其實是會[返回一個 Void](#)(即返回一個 `()` 的值或稱為空的元組 `tuple`)。

這時在可選鏈中呼叫這個方法的話，則是會返回 `Void?`，而不是 `Void`(因為可選鏈一定會返回可選值)，所以仍然可以依照是否返回 `nil` 來判斷這個可選鏈是否成功，如下：

```
let kevin = NewPerson()
kevin.residence = NewResidence()

// 可以依據是否返回 nil 來判斷可選鏈是否成功
if kevin.residence?.printNumberOfRooms() != nil {
    kevin.residence?.printNumberOfRooms()
} else {
    print("無法呼叫函式")
}

// 這時因為都有值且有預設值 所以會印出：有 0 間房間
```

存取下標

與屬性相同，你可以使用可選鍊存取下標，也可以指派新的值給下標。底下是一個例子：

```
// 先生成一個實體
let jack = NewPerson()

// 這邊嘗試存取放在陣列中第一個房間的名稱
if let firstRoomName = jack.residence?[0].name {
    print("第一個房間的名稱為 \(firstRoomName).")
} else {
    print("無法取得第一個房間的名稱")
}

// 這邊嘗試用下標指派一個值
jack.residence?[0] = Room(name: "臥房")

// 上面兩個返回的都是 nil 因為目前 jack.residence 尚未有值

// 先生成一個 NewResidence 實體
let jackHouse = NewResidence()

// 接著為其內型別為 [Room] 的陣列屬性加上值
jackHouse.rooms.append(Room(name: "廚房"))
jackHouse.rooms.append(Room(name: "浴室"))

// 最後將其設置為 jack.residence
jack.residence = jackHouse

// 這時再存取 即會有值 這邊會印出：第一個房間名稱為 廚房
if let firstRoomName = jack.residence?[0].name {
    print("第一個房間名稱為 \(firstRoomName).")
} else {
    print("無法取得第一個房間的名稱")
}
```

Hint

- 如果可選鍊中的下標的值為一個可選值，記得要將問號 ? 放在中括號 [] 的前面而不是後面。可選鍊的問號一定是緊接在可選表達式的後面。

存取可選型別的下標

這邊以一個字典 Dictionary 型別的變數做示範，可選鍊的問號 ? 必須放在下標的中括號 [] 後面來鏈接可選返回值，如下：

```
// 宣告一個字典的變數
var testScores = [
    "Dave": [86, 82, 84],
    "Bev": [79, 94, 81]
]

// 為下標設置新的值
testScores["Dave"]?[0] = 91
testScores["Bev"]?[0] += 1

// 因為沒有 Brian 這個鍵 會是 nil
// 所以指派值會失敗 底下這行不會有任何事情發生
testScores["Brian"]?[0] = 72
```

可選鍊的方法返回一個可選值

前面的例子說明可以使用可選鍊來存取可選型別的值，同樣地，也可以使用可選鍊來呼叫一個會返回可選型別的值的方法，如果需要的話仍然可以對該返回值繼續進行鏈接，如下：

```
john.someProperty?.someMethod()?.hasPrefix("The")
```

Hint

- 可選鍊的問號 ? 是放在方法的小括號 () 後面，因為可選值是 someMethod() 方法的返回值，而不是 someMethod() 方法本身。

範例

本節範例程式碼放在 [ch2/optional-chaining.playground](#)

錯誤處理

- 錯誤的描述與拋出
- 使用拋出函式傳遞錯誤
- 錯誤的捕獲及處理
- 轉換錯誤為可選值
- 禁用錯誤傳遞
- 必定執行的程式區塊

程式運行中，有時會遇到錯誤需要處理，像是需要讀取一個檔案，但檔案可能不存在或是沒有讀取權限，還有像是一個購物車需要進行業務邏輯上的判斷，結帳前要檢查是否有商品或是超過數量限制等等。對此 Swift 提供了完整的對於錯誤的拋出、捕獲、傳遞及處理的支持。

錯誤的描述與拋出

首先我們必須定義一組錯誤描述，來讓程式中遇到錯誤時，可以清楚知道當前是遇到了什麼錯誤，以及各自匹配後續的處理。Swift 中通常是使用一個遵循 `ErrorType` 協定(`protocol`)的列舉來表示一組錯誤描述(`ErrorType` 是一個空的協定，只是為了告訴 Swift 這個列舉是用來表示錯誤描述)。

後面章節會正式介紹 [協定](#)。

下面例子為一組自動販賣機錯誤描述的列舉，成員依序為：無此商品、金額不足(有一個參數為還需要補足多少錢幣)及商品已賣光：

```
enum VendingMachineError: ErrorType {
    case InvalidSelection
    case InsufficientFunds(coinsNeeded: Int)
    case OutOfStock
}
```

當遇到一個錯誤的時候，我們會表示這邊有一個錯誤發生，然後會將這個錯誤拋出，後續再由自己定義處理方式或交由 Swift 自動處理。Swift 中使用 `throw` 語句來拋出一個錯誤。下面例子表示拋出一個自動販賣機還需要補足 3 個錢幣的錯誤：

```
throw VendingMachineError.InsufficientFunds(coinsNeeded: 3)
```

使用拋出函式傳遞錯誤

前面說明了如何拋出錯誤，接著我們必須設計一個函式，其內部會經過邏輯判斷是否發生異常或錯誤，當發生時便會拋出錯誤。Swift 使用 `throws` 關鍵字來標記函式，稱其為拋出函式(`throwing function`)，格式如下：

```
func 函式名稱() throws -> 返回值型別 {
    內部執行的程式
}
```

Hint

- 拋出函式中的拋出(`throw`)有點類似 `return`，因為拋出一個錯誤表示一個異常或錯誤發生了，所以正常的執行流程會立即中止，其後的程式都不會繼續執行，會直接傳遞至處理錯誤的地方繼續。
- 只有拋出函式可以傳遞錯誤。任何在一個非拋出函式中拋出的錯誤都必須在該函式內部處理。

以下定義一個自動販賣機的類別：

```
// 先定義一個結構來表示一個商品的內容 分別為商品的價錢及數量
struct Item {
    var price: Int
    var count: Int
}

// 定義一個自動販賣機的類別
class VendingMachine {
    // 自動販賣機內的商品
    var inventory = [
        "可樂": Item(price: 25, count: 4),
        "洋芋片": Item(price: 20, count: 7),
        "巧克力": Item(price: 35, count: 11)
    ]
}
```

```
// 目前已投入了多少錢幣 預設值為 0
var coinsDeposited = 0

// 所有判斷錯誤的邏輯都通過後 確定購買商品的動作
func dispenseSnack(snack: String) {
    print("Dispensing \(snack)")
}

// 販售的動作 確定售出前會做些判斷
// 這是一個拋出函式 所以函式名稱需要加上 throws
func vend(itemNamed name: String) throws {
    // 檢查是否有這個商品 沒有的話會拋出錯誤
    guard var item = inventory[name] else {
        throw VendingMachineError.InvalidSelection
    }

    // 檢查這個商品是否還有剩 已賣光的話會拋出錯誤
    guard item.count > 0 else {
        throw VendingMachineError.OutOfStock
    }

    // 檢查目前投入的錢幣夠不夠 不夠的話會拋出錯誤
    guard item.price <= coinsDeposited else {
        // 參數為還需要補足多少錢幣 所以是商品價錢減掉已投入錢幣
        throw VendingMachineError.InsufficientFunds(
            coinsNeeded: item.price - coinsDeposited)
    }

    // 所有判斷都通過後 才確定會售出
    coinsDeposited -= item.price
    item.count -= 1
    inventory[name] = item
    dispenseSnack(name)
}

}
```

錯誤的捕獲及處理

前面定義的類別中有一個拋出函式，如果遇到錯誤時會將錯誤拋出並傳遞至錯誤處理的地方，目前尚未定義怎麼處理錯誤，所以這時 Swift 會自動處理，不過這可能會導致程式中止，所以我們還是自行定義錯誤處理的方式。

Swift 使用 `do-catch` 語句來定義錯誤的捕獲(`catch`)及處理，每一個 `catch` 表示可以捕獲到一個錯誤拋出的處理方式，以下是格式：

```
do {
    try 拋出函式
    其他執行的程式
} catch 錯誤1 {
    處理錯誤1
} catch 錯誤2 {
    處理錯誤2
}
```

Hint

- 如果要呼叫拋出函式，必須在函式前加上 `try` 關鍵字。
- 如果要捕獲拋出的錯誤，必須將拋出函式(或拋出錯誤的程式)寫在關鍵字 `do` 包含的大括號 `{}` 內。
- 使用關鍵字 `catch` 來匹配要捕獲的每個錯誤。

底下是一個例子：

```
// 生成一個自動販賣機類別的實體 並設置已投入 8 個錢幣
var vendingMachine = VendingMachine()
vendingMachine.coinsDeposited = 8

// 進行錯誤的拋出、捕獲及處理
do {
    // 呼叫拋出函式 我要購買可樂這個商品
    try vendingMachine.vend(itemNamed: "可樂")

    // 其他可能需要執行的程式 這邊先省略

    // 以下每個 catch 為各自匹配錯誤的處理
} catch VendingMachineError.InvalidSelection {
    print("無此商品")
} catch VendingMachineError.OutOfStock {
    print("商品已賣光")
} catch VendingMachineError.InsufficientFunds(let coinsNeeded) {
    print("金額不足，還差 \(coinsNeeded) 個錢幣")
}
```

上述程式中可以看到 `vendingMachine.vend(itemNamed:)` 函式內，每個會拋出的錯誤都可以在 `do-catch` 中列出的 `catch` 語句中匹配到。

而上述這個例子依序判斷錯誤到最後，會因為錢幣不足而拋出 `VendingMachineError.InsufficientFunds` 這個錯誤，並在外面的 `do-catch` 中被捕獲到，最後會印出：金額不足，還差 **17** 個錢幣。當然因為已經拋出錯誤，其後的程式都不會繼續執行。

轉換錯誤為可選值

前面提到可以使用一個拋出函式來拋出錯誤，並使用 `do-catch` 來捕獲並處理錯誤。而如果只是要簡單讓錯誤發生時，返回為一個 `nil`，像是如下的表示：

```
// 定義一個拋出函式 會返回一個 Int
func someThrowingFunction() throws -> Int {
    // 內部執行的程式
}

// 告知一個可選型別 Int? 的常數 x
let x: Int?
do {
    // 呼叫拋出函式 會返回一個 Int
    x = try someThrowingFunction()
} catch {
    // 錯誤發生而被拋出 進而捕獲時 將其設為 nil
    x = nil
}
```

如上述程式的功能，我們可以簡單的將 `try` 改成使用 `try?`，這樣當錯誤發生要被拋出時，會簡單的返回一個 `nil`。如下：

```
let y = try? someThrowingFunction()
```

Hint

- 不論原本拋出函式返回的是不是可選值，使用 `try?` 呼叫的拋出函式，都會返回可選值。

禁用錯誤傳遞

當你知道一個拋出函式確定不會在執行時拋出錯誤，這時可以使用 `try!` 來呼叫拋出函式，這樣會將錯誤傳遞禁用，但當錯誤真的被拋出時，會發生程式運行時錯誤。

也就是說，使用 `try!` 呼叫拋出函式來告訴 Swift 確定這個呼叫不會發生異常或錯誤。還有一點，使用 `try!` 呼叫拋出函式，可以不用放在 `do` 的大括號 `{}` 內。例子如下：

```
let z = try! someThrowingFunction()
```

必定執行的程式區塊

我們可以使用 `defer` 定義一個程式區塊，當在無論是拋出(`throw`)錯誤，或是使用 `return`、`break` 結束這個函式後，都必定會執行這個程式區塊。

當在需要做清理工作或是釋放記憶體之類的程式時很好用，像是一個開啓檔案的函式，如下：

```
func someMethod() throws {
    // 打開一個資源 像是開啓一個檔案

    defer {
        // 釋放資源記憶體或清理工作
        // 像是關閉一個開啓的檔案
    }

    // 錯誤處理 像是檔案不存在或沒有讀取權限

    // 及其他要執行的程式
}
```

依照上述程式，這樣不論在正常執行程式到最後或是因為發生錯誤拋出而中止，最後都會執行 `defer` 內的程式，保證清理工作一定會執行。

Hint

- 如果定義多個 `defer`，會先執行最後一個定義的 `defer`，再依序往前執行到第一個。
- `defer` 不是一定要與錯誤處理一起使用，普通的函式內也可以使用。

範例

本節範例程式碼放在 [ch2/error-handling.playground](#)

型別轉換

- 型別檢查
- 向下型別轉換
- Any 及 AnyObject 的型別轉換

Swift 提供兩個運算子：`is` 跟 `as` 來檢查一個值的型別以及將一個值轉換成另一種型別。

基於類別繼承的特性，一個子類別繼承於另一個父類別時，這個子類別的型別除了可以是自己之外，也可以是父類別型別。下面例子以三個類別(一個基礎類別及兩個繼承自它的子類別)示範 `is` 及 `as` 的使用方法：

```
// 首先定義一個類別 GameCharacter
class GameCharacter {
    var name: String
    init(name: String) {
        self.name = name
    }
}

// 接著定義兩個繼承自 GameCharacter 的類別
class Archer: GameCharacter {
    var intro: String
    init(name: String, intro: String) {
        self.intro = intro
        super.init(name: name)
    }
}

class Warrior: GameCharacter {
    var description: String
    init(name: String, description: String) {
        self.description = description
        super.init(name: name)
    }
}
```

根據上面建立的類別，接著宣告一個陣列常數，包含 2 個 Archer 實體及 1 個 Warrior 實體，而初始化時會自動推斷出 Archer 跟 Warrior 有一個共同的父類別 GameCharacter，所以這個陣列會自動推斷型別為 [GameCharacter]，如下：

```
// gameTeam 的型別被自動推斷為 [GameCharacter]
let gameTeam = [
    Archer(name: "one", intro: "super power"),
    Warrior(name: "two", description: "good fighter"),
    Archer(name: "three", intro: "not bad")
]
```

型別檢查

使用型別檢查運算子 `is` 來檢查一個實體是否屬於一個類別。檢查後會返回一個布林值，是這個類別會返回 `true`，反之則返回 `false`。以下使用 `for-in` 來遍歷前面宣告的陣列 `gameTeam`：

```
// 用來計算弓箭手跟戰士的數量
var archerCount = 0
var warriorCount = 0

for character in gameTeam {
    // 檢查是否是弓箭手或戰士
    if character is Archer {
        archerCount += 1
    } else if character is Warrior {
        warriorCount += 1
    }
}

// 最後印出：有 2 個弓箭手跟 1 個戰士。
print("有 \(archerCount) 個弓箭手跟 \(warriorCount) 個戰士。")
```

你也可以用 `is` 來檢查一個類別是否遵循了某個協定(`protocol`)。

後面章節會正式介紹[協定](#)。

向下型別轉換

一個類別的常數或變數，可能實際上是屬於一個子類別(像是前面提的例子)。當遇到這種情況，可以嘗試使用型別轉換運算子 `as?` 或 `as!` 將其向下型別轉換至子類別型別。

如果不確定向下型別轉換是否能夠成功，則使用 `as?` 來轉換，這會返回一個可選值，如果向下型別轉換失敗會返回一個 `nil`，這樣的特性可以讓你檢查向下型別轉換是否成功。

如果確定向下型別轉換一定成功時，則可以使用強制性的 `as!` 來轉換。但如果轉換至一個錯誤的型別，則會觸發程式運行時錯誤。

以下仍用前面宣告的陣列 `gameTeam` 來做例子：

```
for character in gameTeam {
    if let oneChar = character as? Archer {
        print("弓箭手的名字：\\" + (oneChar.name) + "\\")
        print("介紹：" + (oneChar.intro))
    } else if let anotherChar = character as? Warrior {
        print("戰士的名字：" + (anotherChar.name))
        print("描述：" + (anotherChar.description))
    }
}

// 使用可選綁定來檢查是否轉換成功 會依序印出：
// 弓箭手的名字：one
// 介紹：super power
// 戰士的名字：two
// 描述：good fighter
// 弓箭手的名字：three
// 介紹：not bad
```

Any 及 AnyObject 的型別轉換

Swift 為不確定的型別提供了兩種特殊型別別名：

- `AnyObject`：可以表示任何類別型別的實體。
- `Any`：可以表示為任何型別。

Hint

- 為了型別安全，我們應該明確地指定值或實體的型別，除非是真的必要或確切需要才使用 `Any` 跟 `AnyObject`。

AnyObject

這邊使用前面定義的類別 `Archer` 做例子：

```
let someObjects: [AnyObject] = [
    Archer(name: "one", intro: "super power"),
    Archer(name: "two", intro: "not bad")
]

// 我們明確的知道這個陣列只包含著 Archer 實體
// 所以可以使用強制性的 as! 來向下轉換至 Archer 型別
for object in someObjects {
    let oneChar = object as! Archer
    print("弓箭手的名字：\u{00d7}(oneChar.name)，介紹：\u{00d7}(oneChar.intro)")
}

// 依序會印出：
// 弓箭手的名字：one，介紹：super power
// 弓箭手的名字：two，介紹：not bad
```

以這個例子來說，可以直接將陣列強制向下轉換(`as!`)為 `[Archer]` 型別，這樣可以讓程式碼更為簡潔，如下：

```
for object in someObjects as! [Archer] {
    print("弓箭手的名字：\u{00d7}(object.name)，介紹：\u{00d7}(object.intro)")
}
```

Any

以下宣告一個 `[Any]` 型別的陣列，所以其內可以放進各種型別的值，如下：

```

var things = [Any]()

// 依序加入 浮點數, 字串, 元組, Archer 實體, 閉包
things.append(3.1415926)
things.append("Hello, world")
things.append((3.0, 5.0))
things.append(Archer(name: "one", intro: "super power"))
things.append({ (name: String) -> String in "Hello, \(name)" })

// 再以 for-in 遍歷陣列 使用 switch 配對每一個值
for thing in things {
    switch thing {
        case let someDouble as Double where someDouble > 0:
            print("浮點數為 \(someDouble)")
        case let someString as String:
            print("字串為 \"\(someString)\"")
        case let (x, y) as (Double, Double):
            print("元組為 \(x), \(y)")
        case let oneChar as Archer:
            print("弓箭手的名字 : \(oneChar.name)")
            print("介紹 : \(oneChar.intro)")
        case let stringConverter as String -> String:
            print(stringConverter("Jess"))
        default:
            print("沒有配對到的值")
    }
}

// 依序印出：
// 浮點數為 3.1415926
// 字串為 "Hello, world"
// 元組為 3.0, 5.0
// 弓箭手的名字 : one
// 介紹 : super power
// Hello, Jess

```

範例

本節範例程式碼放在 [ch2/type-casting.playground](#)

巢狀型別

巢狀型別(`nested types`)表示在一個列舉、結構或類別中，還可以依照需求在其內，再定義列舉、結構或類別，以下是定義一個撲克牌結構，並內含列舉的例子：

```
struct Poker {

    enum Suit: String {
        case Spades = "黑桃", Hearts = "紅心"
        case Diamonds = "方塊", Clubs = "梅花"
    }

    enum Rank: Int {
        case Two = 2, Three, Four, Five
        case Six, Seven, Eight, Nine, Ten
        case Jack, Queen, King, Ace
    }

    let rank: Rank, suit: Suit

    func description () {
        print("這張牌的花色是:\(suit.rawValue)")
        print("點數為:\(rank.rawValue)")
    }

}

let poker = Poker(rank: .King, suit: .Hearts)

// 印出：這張牌的花色是：紅心，點數為：13
poker.description()
```

如果你要在外部使用一個巢狀型別內部的列舉、結構或類別時，可以在其前面直接加上這個巢狀型別的名稱即可，如下：

```
let diamondsName = Poker.Suit.Diamonds  
  
// 印出：方塊  
print(diamondsName.rawValue)
```

範例

本節範例程式碼放在 [ch2/nested-types.playground](#)

擴展

- 擴展語法
- 計算屬性
- 方法
- 建構器
- 下標
- 巢狀型別

擴展(`extension`)是 Swift 一個重要的特性，它可以為已存在的列舉、結構、類別和協定添加新功能，而且不需要修改該型別原本定義的程式碼。擴展也可以使用在內建的型別上，像是 `Int` 、 `Double` 或 `String` 等等。

Swift 的擴展可以：

- 新增計算屬性(包含實體屬性和型別屬性)。
- 定義實體方法和型別方法(不能覆寫已存在的方法)。
- 提供新的建構器。
- 定義下標。
- 定義和使用新的巢狀型別。
- 讓一個已存在的型別遵循某個協定。

擴展語法

使用 `extension` 關鍵字來定義一個擴展，格式如下：

```
extension 某個型別 {  
    新增的程式內容  
}
```

當你對一個已存在的型別新增一個擴展之後，擴展的新功能可以立即給該型別的所有實體使用，即使這個實體在定義擴展前就已經生成了也是可以。

另外，擴展也可以讓一個已有的型別遵循一個或多個協定，格式就如同結構及類別一樣：

```
extension 某個型別: 協定, 另一個協定, 又另一個協定 {
    新增的程式內容
}
```

後面章節會正式介紹[協定](#)。

計算屬性

擴展可以對內建的型別增加計算實體屬性與計算型別屬性。下面例子為內建的 `Double` 型別增加了 3 個計算實體屬性，用來表示常見的距離單位：

```
extension Double {
    var km: Double { return self * 1_000.0 }
    var m: Double { return self }
    var cm: Double { return self / 100.0 }
}
```

定義好新增的擴展之後，就可以直接使用，使用方法就如同普通的屬性一樣使用點語法再緊接著屬性名稱，如下：

```
// 直接對型別 Double 的值取得屬性
let aMarathon = 42.km + 195.m

// 印出：馬拉松的距離全長為 42195.0 公尺
print("馬拉松的距離全長為 \(aMarathon) 公尺")
```

Hint

- 擴展不能新增儲存屬性，也不能為已有的屬性添加屬性觀察器(`property observer`)。

方法

擴展可以為已有的型別新增實體方法與型別方法。以下例子為內建的 `Int` 型別新增一個實體方法：

```
// 新增一個實體方法 有一個參數 型別為 () -> Void 的閉包
// 這個新增的實體方法會執行這個閉包
// 執行次數為：這個整數本身代表數字
extension Int {
    func repetitions(task: () -> Void) {
        for _ in 0..

```

變異實體方法

擴展也可以新增變異實體方法，與一般變異方法一樣在前面加上 `mutating` 關鍵字，下面例子為內建的 `Int` 型別新增一個變異實體方法：

```
// 為內建的 Int 型別新增一個變異實體方法：取得這個整數的平方數
extension Int {
    mutating func square() {
        self = self * self
    }
}

// 先宣告一個整數
var oneInt = 5

// 接著呼叫方法 這裡會得到 25
oneInt.square()
```

建構器

擴展能為類別新增便利建構器(`convenience initializer`)，但不能新增指定建構器(`designated initializer`)跟解構器(`deinitializer`)。

以下例子為一個結構新增一個建構器。在介紹結構時有提過，如果沒有為結構定義建構器時，結構會有一個自動生成的成員逐一建構器(`memberwise initializer`)，而這邊因為是使用擴展為結構新增建構器，所以原本的成員逐一建構器仍然可以使用：

```
// 定義一個結構 會有一個自動生成的成員逐一建構器
struct GameCharacter {
    var hp = 100, mp = 100, name = ""
}

// 為結構 GameCharacter 定義一個建構器的擴展
extension GameCharacter {
    init(name:String) {
        self.name = name
        print("新名字為 \(name)")
    }
}

// 使用擴展後定義的建構器
let oneChar = GameCharacter(name: "弓箭手")

// 原本的成員逐一建構器仍然可以使用
let twoChar = GameCharacter(hp: 200, mp: 50, name: "戰士")
```

Hint

- 使用擴展新增一個新的建構器時，仍然需要確保建構過程中的每一個實體的完全初始化。

下標

擴展可以為已有的型別新增下標。下面例子為內建的 `Int` 型別增加下標：

```
// 定義下標 取得一個整數從個位數算起第幾個數字
// 索引值：0 為取得個位數， 1 為取得十位數， 2為取得百位數 依此類推
extension Int {
    subscript(digitIndex: Int) -> Int {
        var decimalBase = 1
        for _ in 0..
```

巢狀型別

擴展可以為已有的列舉、結構和類別新增巢狀型別。以下為內建的 Int 型別內新增一個列舉的擴展：

```
// 為內建的 Int 型別內新增一個列舉的擴展
// 用來表示這個整數是負數、零還是正數
extension Int {
    enum Kind {
        case Negative, Zero, Positive
    }

    // 另外還新增一個計算屬性 用來返回列舉情況
    var kind: Kind {
        switch self {
        case 0:
            return .Zero
        case let x where x > 0:
            return .Positive
        default:
            return .Negative
        }
    }
}

// 依序會印出：Positive、Negative、Zero
for number in [3, -12, 0] {
    print(number.kind)
}
```

範例

本節範例程式碼放在 [ch2/extensions.playground](#)

協定

- 協定語法
- 屬性的規則
- 方法的規則
- 建構器的規則
- 協定為一種型別
- 委任模式
- 為擴展添加協定
- 協定型別的集合
- 協定的繼承
- 協定合成
- 檢查協定
- 可選協定的規則
- 協定擴展

協定(protocol)是 Swift 一個重要的特性，它會定義出為了完成某項任務或功能所需的方法、屬性，協定本身不會實作這些任務跟功能，而僅僅只是表達出該任務或功能的名稱。這些功能則都交由遵循協定的型別來實作，列舉、結構及類別都可以遵循協定，遵循協定表示這個型別必須實作出協定定義的方法、屬性或其他功能。

有點像是協定定義出一個 To Do List ，而所有遵循協定的型別都必須照表操課，將需要的功能都實作出來。

協定語法

使用 protocol 關鍵字來定義一個協定，格式如下：

```
protocol 協定名稱 {
    協定定義的內容
}
```

要讓自定義的型別遵循協定，寫法有點像繼承，一樣是把協定名稱寫在冒號 : 後面，而要遵循多個協定時，則是以逗號 , 分隔每個協定，格式如下：

```
struct 自定義的結構名稱: 協定, 另一個協定 {
    結構的內容
}
```

同時要繼承父類別跟遵循協定時，應該將父類別名稱寫在第一個，其後才是接著協定名稱，同樣都是以逗號 , 分隔，格式如下：

```
class 類別名稱: 父類別, 協定, 另一個協定 {
    類別的內容
}
```

屬性的規則

協定不能定義一個屬性是儲存屬性或計算屬性，而只是定義屬性的名稱及是實體屬性或型別屬性。此外還可以定義屬性是唯讀或是可讀寫的。

- 協定定義屬性是可讀寫時，則遵循協定的型別定義的屬性不能是常數屬性或唯讀的計算屬性。
- 協定定義屬性是唯讀時，則遵循協定的型別定義的屬性可以是唯讀或依照需求改定義為可讀寫。

協定使用 var 關鍵字來定義變數屬性，在型別標註後加上 { get set } 來表示是可讀寫的，唯讀則是使用 { get } 來表示，如下：

```
protocol 協定 {
    var 可讀寫變數: 型別 { get set }
    var 唯讀變數: 型別 { get }
}
```

在協定中定義型別屬性時，必須在前面加上 static 關鍵字。而當一個類別遵循這個協定時，除了 static 還可以使用 class 關鍵字來定義類別的這個型別屬性：

```
protocol 協定 {
    static var 型別屬性: 屬性型別 { get set }
}
```

底下是一個例子：

```
// 定義一個協定 包含一個唯讀的字串屬性
protocol FullyNamed {
    var fullName: String { get }
}

// 定義一個類別 遵循協定 FullyNamed
struct Person: FullyNamed {
    // 因為遵循協定 FullyNamed
    // fullName 這個屬性一定要定義才行 否則會報錯誤
    var fullName: String
}

let joe = Person(fullName: "Joe Black")
print("名字為 \(joe.fullName)")
// 印出：名字為 Joe Black
```

方法的規則

協定可以定義實體方法或型別方法以供遵循，而這些方法不需要大括號 {} 以及其內的內容(即不需要實作)，而實作則是交給遵循協定的型別來做。

- 協定可以定義含有可變數量參數(variadic parameter)的方法。
- 協定不能為方法的參數提供預設值。

與屬性的規則一樣，協定中要定義型別方法時，必須在前面加上 static 關鍵字。而當一個類別遵循這個協定時，除了 static 還可以使用 class 關鍵字來定義類別的這個型別方法。

協定定義方法的格式如下：

```
protocol SomeProtocol {  
    // 定義一個型別方法  
    static func someTypeMethod()  
  
    // 定義一個實體方法  
    func instanceMethod() -> Double  
  
    // 協定定義方法皆不需要大括號 {} 及其內內容  
}
```

底下是一個例子：

```
protocol SomeProtocol {  
    // 定義一個實體方法 返回一個整數  
    func instanceMethod() -> Int  
}  
  
// 定義一個類別 遵循協定 SomeProtocol  
class MyClass: SomeProtocol {  
    // 因為遵循協定 SomeProtocol  
    // instanceMethod() 這個方法一定要定義才行 否則會報錯誤  
    func instanceMethod() -> Int {  
        return 300  
    }  
}
```

變異方法的規則

使用 `mutating` 關鍵字放在 `func` 關鍵字前來定義變異方法(變異方法表示可以在方法中修改它所屬的實體以及實體的屬性的值)。

遵循一個包含變異方法的協定時，列舉跟結構定義時必須加上 `mutating` 關鍵字，而類別定義時則不用加上。

底下是一個例子：

```
// 定義一個包含變異方法的協定
protocol Toggable {
    // 只需標明方法名稱 不用實作
    mutating func toggle()
}

// 定義一個開關切換的列舉
enum OnOffSwitch: Toggable {
    case Off, On

    // 實作這個遵循協定後需要定義的變異方法
    mutating func toggle() {
        // 會在 On, Off 兩者中切換
        switch self {
            case Off:
                self = On
            case On:
                self = Off
        }
    }
}

var lightSwitch = OnOffSwitch.Off
lightSwitch.toggle()
// lightSwitch 現在切換為 .On
```

建構器的規則

協定可以定義一個建構器，與定義方法一樣不需要寫大括號 {} 及其內的內容，格式如下：

```
protocol OtherProtocol {
    init(someParameter: Int)
}
```

類別中實作協定的建構器

如果是一個類別遵循一個含有建構器的協定時，無論是指定建構器或便利建構器，都必須為類別的建構器加上 `required` 修飾符，以確保所有子類別也必須定義這個建構器，從而符合協定(如果類別已被加上 `final`，則不需要為其內的建構器加上 `required`，因為 `final` 類別不能再被子類別繼承)，如下：

```
class OtherClass: OtherProtocol {
    required init(someParameter: Int) {
        // 建構器的內容
    }
}
```

如果一個子類別覆寫了父類別的指定建構器，且此建構器滿足了某個協定的要求，則該建構器必須同時加上 `required` 和 `override`，如下：

```
// 定義一個協定
protocol AnontherProtocol {
    init()
}

// 定義一個類別
class AnontherSuperClass {
    init() {
        // 建構器的內容
    }
}

// 定義一個繼承 AnontherSuperClass 的類別
// 同時還遵循了協定 AnontherProtocol
class SomeSubClass: AnontherSuperClass, AnontherProtocol {
    // 必須同時加上 required 和 override
    required override init() {
        // 建構器的內容
    }
}
```

可失敗建構器的規則

協定可以定義可失敗建構器：

- 一個協定包含可失敗建構器時，遵循這個協定的型別，可以使用可失敗建構器(`init?`)或非可失敗建構器(`init`)來定義這個建構器。
- 一個協定包含非可失敗建構器時，遵循這個協定的型別，可以使用隱式解析可失敗建構器(`init!`)或非可失敗建構器(`init`)來定義這個建構器。

協定為一種型別

雖然協定本身沒有實作任何功能，但協定仍可以被當做一種型別來使用。使用情況就如同一般型別一樣，如下：

- 作為函式、方法或建構器的參數型別或返回值型別。
- 作為常數、變數或屬性的型別。
- 作為陣列、字典或其他集合中的元素型別。

協定習慣的命名方式就如同一般型別一樣使用大寫字母開頭的[大駝峰式命名法](#)。

底下為一個例子：

```
// 定義一個協定
protocol SomeOtherProtocol {
    func method() -> Int
}

// 定義一個類別 遵循協定 SomeOtherProtocol
class OneClass: SomeOtherProtocol {
    func method() -> Int {
        return 5566
    }
}

// 定義另一個類別 有一個型別為 SomeOtherProtocol 的常數
class AnotherClass {
    // 常數屬性 型別為[協定 SomeOtherProtocol]
    let oneMember: SomeOtherProtocol

    // 建構器有個參數 member 型別為 SomeOtherProtocol
    init(member: SomeOtherProtocol) {
        self.oneMember = member
    }
}

// 先宣告一個類別 OneClass 的實體
let oneInstance = OneClass();

// 任何遵循[協定 SomeOtherProtocol]的實體
// 都可以被當做[協定 SomeOtherProtocol]型別
// 所以上面宣告的 oneInstance 可以被當做參數傳入
let twoInstance = AnotherClass(member: oneInstance)

// 印出：5566
print(twoInstance.oneMember.method())
```

由上述程式可以知道，任何遵循一個協定的實體，都可以被當做這個協定型別的值。

委任模式

委任(delegation)是一種設計模式，它允許類別或結構將一些需要它們負責的功能委任給其他型別的實體。

委任模式的實作就是定義協定來封裝那些需要被委任的功能，而遵循這個協定的型別就能提供這些功能。委任模式可以用來回應特定的動作或是接收外部資料，而不需要知道外部資料的型別。

以下是一個例子：

```
// 定義一個協定 遵循這個協定的類別都要實作 attack() 方法
protocol GameCharacterProtocol {
    func attack()
}

// 定義一個委任協定 將一些其他功能委任給別的實體實作
protocol GameCharacterProtocolDelegate {
    // 這邊是定義一個在攻擊後需要做的整理工作
    func didAttackDelegate()
}

// 定義一個類別 表示一個遊戲角色
class GameCharacter: GameCharacterProtocol {
    // 首先定義一個變數屬性 delegate
    // 型別為 GameCharacterProtocolDelegate
    // 定義為可選型別 會先初始化為 nil 之後
    // 再將其設置為負責其他動作的另一個型別的實體
    var delegate: GameCharacterProtocolDelegate?

    // 因為遵循[協定 GameCharacterProtocol]
    // 所以需要實作 attack() 這個方法
    func attack() {
        print("攻擊！")
    }

    // 最後將其他動作委任給另一個型別的實體實作
    delegate?.didAttackDelegate()
}
}
```

```
// 定義一個類別 遵循[協定 GameCharacterProtocolDelegate]  
// 這個類別生成的實體會被委任其他動作  
class GameCharacterDelegate: GameCharacterProtocolDelegate {  
    // 必須實作這個方法  
    func didAttackDelegate() {  
        print("攻擊後的整理工作")  
    }  
}  
  
// 首先生成一個遊戲角色的實體  
let oneChar = GameCharacter()  
  
// 接著生成一個委任類別的實體 要負責其他的動作  
let charDelegate = GameCharacterDelegate()  
  
// 將遊戲角色的 delegate 屬性設為委任的實體  
oneChar.delegate = charDelegate  
  
// 接著呼叫攻擊方法  
oneChar.attack()  
// 會依序印出：  
// 攻擊！  
// 攻擊後的整理工作
```

為擴展添加協定

你也可以讓擴展遵循協定，這樣就可以在不修改原始程式碼的情況下，讓已存在的型別經由擴展來遵循一個協定。當已存在型別經由擴展遵循協定時，這個型別的所有實體也會隨之獲得協定中定義的功能。

```
// 定義另一個協定 增加一個防禦方法 defend
protocol GameCharacterDefend {
    func defend()
}

// 定義一個擴展 會遵循新定義的協定 GameCharacterDefend
extension GameCharacter: GameCharacterDefend {
    // 必須實作這個方法
    func defend() {
        print("防禦！")
    }
}

// 使用前面生成的實體 oneChar
// 這樣這個被擴展的類別生成的實體 也隨即可以使用這個方法
oneChar.defend()
// 印出：防禦！
```

經由擴展遵循協定

當一個型別已經符合某個協定的所有要求，但卻沒有在型別的定義中宣告時，可以經由一個空的擴展來遵循這個協定，以下是個例子：

```
// 定義一個協定
protocol NewProtocol {
    var name: String { get set }
}

// 定義一個類別 滿足了[協定 NewProtocol]的要求 但尚未遵循它
class NewClass {
    var name = "good day"
}

// 這時可以使用擴展來遵循
extension NewClass: NewProtocol {}
```

Hint

- 即使滿足了協定的所有要求，型別也不會自動遵循協定，必須顯式地為它加上遵循協定才行。

協定型別的集合

協定型別也可以作為陣列或字典內成員的型別，以下是個例子：

```
// 生成另外兩個實體
let twoChar = GameCharacter()
let threeChar = GameCharacter()

// 宣告一個型別為 [GameCharacterProtocol] 的陣列
let team:[GameCharacterProtocol]=[oneChar,twoChar,threeChar]

// 因為都遵循這個協定 所以這個 attack() 方法一定存在可以呼叫
for member in team {
    member.attack()
}
```

協定的繼承

協定也可以像類別一樣繼承另外一個或多個協定，可以在繼承的協定的基礎上增加新的功能，繼承的多個協定一樣是使用逗號 , 隔開，如下：

```
protocol 新協定: 繼承的協定, 繼承的另一個協定 {
    協定新增加的功能
}
```

只用於類別的協定

你可以在協定的繼承列表中，增加關鍵字 `class` 來限制這個協定只能被類別型別遵循，而列舉跟結構不能遵循這個協定。`class` 必須擺在繼承列表的第一個，在其他要繼承的協定之前，如下：

```
protocol 只用於類別的協定: class, 其他要遵循的協定 {  
    只用於類別的協定的功能  
}
```

協定合成

如果你需要一個型別同時遵循多個協定，你可以將多個協定使用 `protocol<SomeProtocol, AnotherProtocol>` 這種格式組合起來，這種方式稱為協定合成(`protocol composition`)，在 `<>` 中可以填入多個需要遵循的協定，並以逗號 , 隔開，如下：

```
// 定義一個協定 有一個 name 屬性
protocol Named {
    var name: String { get }
}

// 定義另一個協定 有一個 age 屬性
protocol Aged {
    var age: Int { get }
}

// 定義一個結構 遵循上面兩個定義的協定
struct OnePerson: Named, Aged {
    var name: String
    var age: Int
}

// 定義一個函式 有一個參數 定義為遵循這兩個協定的型別
// 所以寫成 protocol<Named, Aged> 格式
func wishHappyBirthday(celebrator: protocol<Named, Aged>) {
    print("生日快樂！ \(celebrator.name)")
    print("\\(celebrator.age) 歲囉！")
}

let birthdayPerson = OnePerson(name: "Brian", age: 25)
wishHappyBirthday(birthdayPerson)
// 印出：生日快樂！ Brian
// 25 歲囉！
```

檢查協定

你可以使用前面章節提過的 `is` 與 `as` 來檢查是否符合某協定或是轉換到指定的協定型別，使用方式與 [型別檢查與轉換](#) 一樣：

- `is` 用來檢查實體是否符合某協定，符合會返回 `true`，反之則返回 `false`。
- `as?` 返回一個可選值。當實體符合某協定時，會返回協定型別的可選值，反之則返回 `nil`。
- `as!` 將實體強制向下轉換到某協定型別，如果失敗則會引發運行時錯誤。

以下是一個例子：

```
// 定義一個協定 有一個 area 屬性 表示面積
protocol HasArea {
    var area: Double { get }
}

// 定義一個圓的類別 遵循[協定 HasArea] 所以會有 area 屬性
class Circle: HasArea {
    var area: Double
    init(radius: Double) { self.area = 3.14 * radius * radius }
}

// 定義一個國家的類別 遵循[協定 HasArea] 所以會有 area 屬性
class Country: HasArea {
    var area: Double
    init(area: Double) { self.area = area }
}

// 定義一個動物的類別 沒有面積
class Animal {
    var legs: Int
    init(legs: Int) { self.legs = legs }
}

// 以上三個類別的實體都可以作為 [AnyObject] 陣列的成員
let objects: [AnyObject] = [
    Country(area: 243610),
    Circle(radius: 2.0),
    Animal(legs: 4)
]

// 遍歷這個陣列
for object in objects {
    // 使用可選綁定來將成員綁定為 HasArea 的實體
    if let objectWithArea = object as? HasArea {
        // 符合協定 就會綁定成功 也就可以取得 area 屬性
        print("面積為 \(objectWithArea.area)")
    } else {
        // 不符合協定 則是返回 nil
    }
}
```

```

        print("沒有面積!")
    }

}

// 依序印出:
// 面積為 243610.0
// 面積為 12.56
// 沒有面積!

```

可選協定的規則

你可以在 `protocol` 前面加上 `@objc` 特性來讓協定可以定義它的功能(像是屬性或方法)為可選。要定義一個可選功能，必須在前面加上 `optional` 關鍵字。

如果將功能變為可選後，它們的型別會自動變成可選的。像是一個型別為 `(Int) -> String` 的方法會變成 `((Int) -> String)?`，是方法的型別為可選，不是方法返回值的型別。

而這些定義為可選的功能，可以使用可選鍊來呼叫。

以下是一個例子：

```

// 要加上 @objc 必須引入 Foundation
import Foundation
// 這邊不詳細說明 因為可選協定與 Objective-C 程式語言有關係
// 而 Objective-C 大量使用到 Foundation 的功能 所以需要引入

// 定義一個可選協定 用於計數 分別有兩種不同的增量值
@objc protocol CounterDataSource {
    // 定義一個可選方法 可以傳入一個要增加的整數
    optional func incrementForCount(count: Int) -> Int

    // 定義一個可選屬性 為一個固定增加的整數
    optional var fixedIncrement: Int { get }
}

// 定義一個遵循可選協定的類別 計數用
class CounterSource: CounterDataSource {
    // 一個經由遵循協定而擁有的可選屬性 設值為 3
}

```

```

// 前面必須加上 @objc
@objc let fixedIncrement = 3

// 不過因為是可選的 所以另一個可選方法可以不用實作 這邊將其註解起來
/*
@objc func incrementForCount(count: Int) -> Int {
    return count
}
*/
}

// 用來計數的變數
var count = 0

// 生成一個型別為[可選協定 CounterDataSource]的實體
// 因為類別 CounterSource 有遵循這個協定 所以可以指派為這個類別的實體
var dataSource: CounterDataSource = CounterSource()

// 迴圈跑 4 次
for _ in 1...4 {
    // 使用可選綁定
    // 首先呼叫 incrementForCount() 方法
    // 但因為這是個可選方法 所以需要加上 ?
    // 而目前這個 incrementForCount 沒有實作這個方法
    // 所以會返回 nil 也就不會執行 if 內的程式
    if let amount = dataSource.incrementForCount?(count){
        count += amount
    }
    // 接著依舊使用可選綁定 取得屬性 fixedIncrement
    // 因為有設置這個屬性 所以會有值 流程則會進入此 else if 內的程式
    else if let amount = dataSource.fixedIncrement{
        count += amount
    }
}

// 因為迴圈跑了 4 次，每次都是加上 3，所以最後計為 12
// 印出：最後計數為 12
print("最後計數為 \(count)")

```

協定擴展

你也可以擴展一個協定來為遵循這個協定的型別新增屬性、方法或下標，而不需要在每個遵循這個協定的型別內實作一樣的功能。以下是個例子：

```
// 擴展前面定義的協定 GameCharacterProtocol
// 此協定原本只有定義一個 attack() 方法
// 這邊增加一個新的方法

extension GameCharacterProtocol {
    func superAttack() {
        print("額外的攻擊！")
        attack()
    }
}

// 生成一個遊戲角色的實體
let member = GameCharacter()
member.delegate = GameCharacterDelegate()

// 可以直接呼叫擴展協定後新增的方法
member.superAttack()
// 依序印出：
// 額外的攻擊！
// 攻擊！
// 攻擊後的整理工作
```

由上述程式可知，經由擴展一個協定，可以直接為屬性、方法及下標建立預設的實作功能，而這些遵循協定的型別如果自己又另外實作的話，則這些自定義的實作會替代擴展中的預設實作功能。

為協定擴展添加限制條件

在擴展一個協定時，可以指定一些限制條件，當遵循協定的型別滿足這些限制條件時，才能獲得這個擴展的協定提供的預設實作。使用方式為在協定名稱後面加上 `where` 語句並接著限制條件。例子如下：

```
// 先為[協定 GameCharacterProtocol]經由擴展增加一個新的屬性
extension GameCharacterProtocol {
    var description: String {
        return "成員"
    }
}

// 接著擴展[集合型別的協定 CollectionType]
// 且其內成員必須遵循[協定 GameCharacterProtocol]
extension CollectionType where
    Generator.Element: GameCharacterProtocol {
    var allDescription: String {
        let itemsAsText = self.map { $0.description }
        return "[" + itemsAsText.joinWithSeparator(", ") + "]"
    }
}

// 生成三個實體並放入一個陣列中
let oneMember = GameCharacter()
let twoMember = GameCharacter()
let threeMember = GameCharacter()
let myTeam = [oneMember, twoMember, threeMember]

// 因為陣列定義時有遵循[協定 CollectionType]
// 且其內成員都遵循[協定 GameCharacterProtocol]
// 所以這個 allDescription 屬性會自動獲得
// 印出：[成員, 成員, 成員]
print(myTeam.allDescription)
```

範例

本節範例程式碼放在 [ch2/protocols.playground](#)

泛型

- 泛型能解決的問題
- 泛型函式
- 型別參數
- 泛型型別
- 型別約束
- 關聯型別

泛型(generic)是 Swift 一個重要的特性，可以讓你自定義出一個適用任意型別的函式及型別。可以避免重複的程式碼且清楚的表達程式碼的目的。

許多 Swift 標準函式庫就是經由泛型程式碼建構出來的，像是陣列(Array)和字典(Dictionary)都是泛型的，你可以宣告一個 [Int] 陣列，也可以宣告一個 [String] 陣列。同樣地，你也可以宣告任意指定型別的字典。

你可以將泛型使用在函式、列舉、結構及類別上。

泛型能解決的問題

以下是一個可以利用泛型來簡化程式碼的例子：

```
// 定義一個將兩個整數變數的值互換的函式
func swapTwoInts(inout a: Int, inout _ b: Int) {
    let temporaryA = a
    a = b
    b = temporaryA
}

// 宣告兩個整數變數 並當做參數傳入函式
var oneInt = 12
var anotherInt = 500
swapTwoInts(&oneInt, &anotherInt)

// 印出：互換後的 oneInt 為 500，anotherInt 為 12
print("互換後的 oneInt 為 \(oneInt)，anotherInt 為 \(anotherInt)")

// 與上面定義的函式功能相同 只是這時互換的變數型別為字串
func swapTwoStrings(inout a: String, inout _ b: String) {
    let temporaryA = a
    a = b
    b = temporaryA
}
```

由上述程式可以看到，兩個函式的功能完全一樣，唯一不同的只有傳入參數的型別，這種情況便可以使用泛型來簡化。

泛型函式

根據前面提到的兩個功能完全一樣的函式，以下使用泛型來定義一個適用任意型別的函式：

```
func swapTwoValues<T>(inout a: T, inout _ b: T) {
    let temporaryA = a
    a = b
    b = temporaryA
}
```

上述程式中的函式使用了佔位型別名稱(placeholder type name)，習慣以字母 T 來表示)來代替實際型別名稱(像是 Int 、 Double 或 String)。

可以注意到函式名稱後面緊接著一組角括號 <> ，且包著 T 。這代表角括號內的 T 是函式定義的一個佔位型別名稱，因此 Swift 不會去查找名稱為 T 的實際型別。

定義佔位型別名稱時不會明確表示 T 是什麼型別，但參數 a 與 b 都必須是這個 T 型別。而只有當這個函式被呼叫時，才會根據傳入參數的實際型別，來決定 T 所代表的型別。

這時便可以使用這個泛型函式，如下：

```
// 首先是兩個整數
var oneInt2 = 12
var anotherInt2 = 320
swapTwoValues(&oneInt2, &anotherInt2)

// 再來是兩個字串
var oneString = "Hello"
var anotherString = "world"
swapTwoValues(&oneString, &anotherString)
```

型別參數

前面提到的 swapTwoValues(_:_:) 中，佔位型別名稱 T 是型別參數的一個例子。

型別參數會指定並命名一個佔位型別，且會緊跟在函式名稱後面使用一組角括號 <> 包起來。當一個型別參數被指定後，就可以用來定義一個函式的參數型別、函式的返回值型別或是函式內的型別標註。

型別參數可以指定一個或一個以上，使用多個時以逗號 , 隔開。

命名型別參數

在一般情況下，型別參數會指定為一個有描述性的名字，像是 Dictionary<Key, Value> 中的 Key 和 Value ，或是 Array<Element> 中的 Element ，用來明顯表示這些型別參數與泛型函式之間的關係。而當無法有意義的描述型別參數時，

通常會使用單一字母來命名，像是 T 、 U 或 V 。

Hint

- 通常會使用大駝峰式命名法(像是 T 或 MyTypeParameter)來為型別參數命名，以表示他們是佔位型別，而不是一個值。

泛型型別

除了泛型函式，你也可以定義一個泛型型別。你可以定義在列舉、結構或類別上，類似陣列(Array)和字典(Dictionary)。

以下會定義一個堆疊(Stack)的泛型集合型別來當做一個例子。堆疊的運作方式有點像陣列，可以增加(push)一個元素到陣列最後一員，也可以從陣列中取出(pop)最後一個元素。

```
// 定義一個泛型結構 Stack 其佔位型別參數命名為 Element
struct Stack<Element> {
    // 將型別參數用於型別標註 設置一個型別為 [Element] 的空陣列
    var items = [Element]()

    // 型別參數用於方法的參數型別 方法功能是增加一個元素到陣列最後一員
    mutating func push(item: Element) {
        items.append(item)
    }

    // 型別參數用於方法的返回值型別 方法功能是移除陣列的最後一個元素
    mutating func pop() -> Element {
        return items.removeLast()
    }
}
```

上述定義的結構中可以看到，指定 Element 為佔位型別參數後，便可在結構中作為型別標註、方法的參數型別及方法的返回值型別，而因為必須修改結構的內容，所以方法都必須加上 mutating 。

接著就可以使用這個剛定義好的 Stack 型別，如下：

```
// 先宣告一個空的 Stack 這時才決定其內元素的型別為 String
var stackOfStrings = Stack<String>()

// 依序放入三個字串
stackOfStrings.push("one")
stackOfStrings.push("two")
stackOfStrings.push("three")

// 然後移除掉最後一個元素 即字串 "three"
stackOfStrings.pop()

// 現在這個 Stack 還有兩個元素 分別為 one 及 two
```

擴展一個泛型型別

當你擴展一個泛型型別時，不需要在擴展的定義中提供型別參數列表，原型別已經定義的型別參數列表(如前面提到的 Stack 定義的 Element)可以直接在擴展中使用。

以下為堆疊(Stack)擴展一個名稱為 topItem 的唯讀計算屬性，它會返回這個堆疊的最後一個元素，且不會將其移除：

```
extension Stack {
    var topItem: Element? {
        return items.isEmpty ? nil : items[items.count - 1]
    }
}
```

上述程式可以看到，擴展中可以直接使用 Element 。而返回值為一個可選值，所以底下使用可選綁定來取得最後一個元素：

```
if let topItem = stackOfStrings.topItem {
    // 印出：最後一個元素為 two
    print("最後一個元素為 \(topItem)")
}
```

型別約束

有時在定義一個泛型函式或泛型型別時，會需要為這個泛型型別參數增加一些限制，可能是指定型別參數必須繼承自指定的類別，或是符合一個特定的協定。

像是 Swift 內建的字典(`Dictionary`)便對字典的鍵的型別作了些限制。字典的鍵的型別必須是可雜湊的(`hashable`)，也就是必須只有唯一一種方式可以表示這個鍵。

而實際上為了實現這個限制，字典的鍵的型別符合了 `Hashable` 協定。`Hashable` 是 Swift 標準函式庫中定義的一個特定協定，所有 Swift 的基本型別(像是 `Int` 、 `Double` 、 `Bool` 和 `String`)預設都是可雜湊的(`hashable`)。

型別約束語法

你可以在一個型別參數名稱後面加上冒號 : 並緊接著一個類別或是協定來做為型別約束，它們會成為型別參數列表的一部分，例子如下(泛型型別也是一樣方式)：

```
func 泛型函式名稱<T: 某個類別, U: 某個協定>(參數: T, 另一個參數: U) {
    函式內部的程式
}
```

上述定義中可以看到， `T` 型別參數必須繼承自某個類別， `U` 型別參數則必須遵循某個協定。

使用型別約束

以下會定義一個函式，兩個參數分別為一個陣列及一個值，函式的功能是尋找第一個參數陣列中是否有另一個參數值，如果有就返回這個值在陣列中的索引值，找不到則返回 `nil` 。

這個函式的型別約束會使用到另一個 Swift 標準函式庫中的 `Equatable` 協定，這個協定要求任何遵循該協定的型別必須實作 `==` 及 `!=` ，進而可以對該型別的任意兩個值進行比較。(所有的 Swift 標準型別預設都符合 `Equatable` 協定。)

```

func findIndex<T: Equatable>(
    array: [T], _ valueToFind: T) -> Int? {
    for (index, value) in array.enumerate() {
        if value == valueToFind {
            return index
        }
    }
    return nil
}

// 首先找看看 [Double] 陣列的值
let doubleIndex = findIndex([689, 5566, 10.05], 9.2)
// 因為 9.2 不在陣列中 所以返回 nil

// 接著找 [String] 陣列的值
let stringIndex = findIndex(["Adam", "Kevin", "Jess"], "Kevin")
// Kevin 為陣列中第 2 個值 所以會返回 1

```

關聯型別

關聯型別(associated type)表示會為協定中的某個型別提供一個佔位名稱(placeholder name)，其代表的實際型別會在協定被遵循時才會被指定。使用 `associatedtype` 關鍵字來指定一個關聯型別。

底下是一個例子，定義一個協定 `Container`，協定中定義了一個關聯型別 `ItemType`：

```

protocol Container {
    associatedtype ItemType
    mutating func append(item: ItemType)
    var count: Int { get }
    subscript(i: Int) -> ItemType { get }
}

```

上述程式中可以看到，協定定義的方法 `append()` 參數的型別及下標的返回值型別都是 `ItemType`，目前仍是佔位名稱，實際型別要等到這個協定被遵循後才會被指定。

接著我們將前面定義的堆疊(Stack)遵循這個協定 Container，在實作協定 Container 的全部功能後，Swift 會自動推斷 ItemType 的型別就是 Element，如下：

```
struct NewStack<Element>: Container {
    // Stack<Element> 實作的內容
    var items = [Element]()
    mutating func push(item: Element) {
        items.append(item)
    }
    mutating func pop() -> Element {
        return items.removeLast()
    }

    // 原本應該要寫 typealias
    // 但因為 Swift 會自動推斷型別 所以下面這行可以省略
    // typealias ItemType = Element

    // 協定 Container 實作的內容
    mutating func append(item: Element) {
        self.push(item)
    }
    var count: Int {
        return items.count
    }
    subscript(i: Int) -> Element {
        return items[i]
    }
}
```

經由擴展一個已存在的型別來設置關聯型別

前面章節有提過，可以利用擴展來讓一個已存在的型別符合協定，使用了關聯型別的協定也一樣可以。

Swift 內建的陣列(Array)型別恰恰好已經有前面提過的協定 Container 需要實作的功能(分別是方法 append()、屬性 count 及下標返回一個依索引值取得的元素)。所以現在可以很簡單的利用一個空的擴展來讓 Array 遵循這個協定，如下：

```
extension Array: Container {}
```

Where 語句

有時候你也可能需要對關聯型別定義更多的限制，這時可以經由在參數列表加上一個 `where` 語句，並緊接著限制條件來定義。你可以限制一個關聯型別要遵循某個協定，或是某個型別參數和關聯型別必須相同型別。

底下定義一個泛型函式 `allItemsMatch()`，功能為檢查兩個容器是否包含相同順序的相同元素，如果條件都符合會返回 `true`，否則返回 `false`：

```
func allItemsMatch<  
    C1: Container, C2: Container  
    where C1.ItemType == C2.ItemType, C1.ItemType: Equatable>  
    (someContainer: C1, _ anotherContainer: C2) -> Bool {  
  
    // 檢查兩個容器含有相同數量的元素  
    if someContainer.count != anotherContainer.count {  
        return false  
    }  
  
    // 檢查每一對元素是否相等  
    for i in 0..        if someContainer[i] != anotherContainer[i] {  
            return false  
        }  
    }  
  
    // 所有條件都符合 返回 true  
    return true  
}
```

從上述定義可以看到，這個函式的型別參數列表還定義了對兩個型別參數的要求：

- `C1` 必須符合協定 `Container` (即 `C1: Container`)。
- `C2` 必須符合協定 `Container` (即 `C2: Container`)。
- `C1` 的 `ItemType` 必須與 `C2` 的 `ItemType` 型別相同(即 `C1.ItemType == C2.ItemType`)。

- C1 的 ItemType 必須符合協定 Equatable (即 C1.ItemType: Equatable)。

接著可以實際使用這個函式，如下：

```
// 宣告一個型別為 NewStack 的變數 並依序放入三個字串
var newStackOfStrings = NewStack<String>()
newStackOfStrings.push("one")
newStackOfStrings.push("two")
newStackOfStrings.push("three")

// 宣告一個陣列 也放置了三個字串
var arrayOfStrings = ["one", "two", "three"]

// 雖然 Stack 跟 Array 不是相同型別
// 但先前已將兩者都遵循了協定 Container
// 且都包含相同型別的值
// 所以可以把這兩個容器當做參數傳入函式
if allItemsMatch(newStackOfStrings, arrayOfStrings) {
    print("所有元素都符合")
} else {
    print("不符合")
}
// 印出：所有元素都符合
```

範例

本節範例程式碼放在 [ch2/generics.playground](#)

存取控制

- 存取層級
- 自定義型別
- 子類別
- 常數, 變數, 屬性及下標
- 建構器
- 協定
- 擴展
- 泛型
- 型別別名

Swift 提供存取控制(access control)的特性，讓你可以為程式碼或模組設置存取權限，決定哪些部分可以開放給外部程式碼使用。

Swift 中，可以為列舉、結構或類別設置存取權限，這些型別內部的屬性、方法、下標或建構器也可以設置存取權限。而協定中的全域變數、常數或函式也可以設置存取權限。

如果你只是單純的在開發一個獨立的應用程式，而不是開發一個模組(像是可供他人使用的工具)，其實可以不用顯式地設置存取權限，Swift 已為大部分的情況提供預設存取權限。

存取層級

Swift 提供了 3 種不同的存取層級，分別如下：

- **public**：公開存取的層級，同模組中的任何程式及外部引用這個模組的程式都可以使用。通常要將一個模組的介面(與模組以外程式互動交流的部分)設為公開時，會將其設為 **public**。
- **internal**：內部存取的層級，同模組中的程式可以使用，但外部引用的程式不能使用。當定義為只供應用程式或模組內部使用時，可以將其設為 **internal**。
- **private**：私有存取的層級，只能在原始定義的程式碼中使用。使用 **private** 可以用來隱藏特定功能的實作。

使用方式為在變數、函式或類別的前面加上這 3 種不同的關鍵字來宣告它們的存取層級，如下：

```
public class SomePublicClass {}
internal class SomeInternalClass {}
private class SomePrivateClass {}

public var somePublicVariable = 0
internal let someInternalConstant = 0
private func somePrivateFunction() {}
```

存取層級基本原則

Swift 存取層級的基本原則：不可以在一個實體中定義存取層級限制更嚴格的實體。如以下的例子：

- 一個 `public` 的變數，不能將它的型別定義為 `internal` 或 `private`。因為當變數可以被公開存取，但定義它的型別不行，這樣會出現錯誤。
- 一個函式的參數、返回型別存取層級限制不能比函式本身的更嚴格。因為當函式存取層級設為 `public`，可以被公開存取時，但參數或返回型別存取層級設為 `internal` 或 `private`，無法被公開存取，這樣會出現錯誤。

預設存取層級

如果沒有顯性宣告存取層級時，`internal` 就是預設的存取層級。

自定義型別

你也可以為自定義的型別定義存取層級，當然你必須確認這個型別的作用範圍與存取層級的限制相符。

類別的存取層級會影響其內部成員的存取層級，像是定義一個 `private` 的類別，則其內部成員的存取層級都會是 `private`。而定義一個 `public` 或 `internal` 的類別，其內部成員都會是 `internal`。

Hint

- 一個 `public` 類別，其內成員預設存取層級為 `internal` 而不是 `public`，

如果要為某個成員設為 `public`，必須顯示定義。這樣在你定義一個模組的公開介面時，可以明確的選擇哪些介面是公開的，避免不小心將內部使用的介面公開。

以下是幾個類別與其內部成員存取層級的例子：

```
// 顯示指定為 public 類別
public class SomePublicClass {
    // 顯示指定為 public 成員
    public var somePublicProperty = 0

    // 隱式推斷為 internal 成員
    var someInternalProperty = 0

    // 顯示指定為 private 成員
    private func somePrivateMethod() {}

}

// 隱式推斷為 internal 類別
class SomeInternalClass {
    // 隱式推斷為 internal 成員
    var someInternalProperty = 0

    // 顯示指定為 private 成員
    private func somePrivateMethod() {}

}

// 顯示指定為 private 類別
private class SomePrivateClass {
    // 隱式推斷為 private 成員
    var somePrivateProperty = 0

    // 隱式推斷為 private 成員
    func somePrivateMethod() {}

}
```

元組型別

元組不能明確指定存取層級，而是在使用時被自動推斷的。元組的存取層級會根據其內成員存取層級最嚴格的一個為準，例子如下：

```
private var description = "Sunny day !"
internal var number = 300
public var name = "Joe Black"

// 這時這個元組的存取層級會根據最嚴格的 private 為準
let someTuple = (description, number, name)
```

函式型別

函式的存取層級是根據存取層級最嚴格的參數或返回值型別來決定。經由此規則得到的存取層級如果與預設存取層級不一樣，則必須明確的指定函式的存取層級，如下：

```
// 定義一個用來當做[函式返回值型別]的類別
private class SomeClass {}

// 定義一個函式
// 返回值為 SomeClass 存取層級為 private
// 則這個函式的存取層級也為 private
// 這時與預設的 internal 不一樣 所以必須明確指定函式為 private
// 如果將函式前面的 private 拿掉 會報錯誤
private func someFunction() -> SomeClass {
    return SomeClass()
}
```

列舉型別

列舉成員的存取層級與列舉相同，無法為列舉成員單獨指定不同的存取層級，如下：

```
// 定義列舉的存取層級為 public
public enum CompassPoint {
    // 則列舉成員都為 public
    case North
    case South
    case East
    case West
}
```

而列舉的原始值(raw value)與相關值(associated value)的存取層級限制不能比列舉的存取層級嚴格。例如，你不能在一個 internal 的列舉中定義 private 的原始值。

巢狀型別

以巢狀型別來說，定義在 private 型別中的巢狀型別，會自動指定為 private 。而在 public 或 internal 型別中，巢狀型別則自動指定為 internal ，這時如果要指定巢狀型別為 public ，則必須明確指定為 public 。

子類別

子類別的存取層級限制不能比父類別更為寬鬆，例如父類別為 internal 時，子類別就不能是 public 。

在符合當前存取層級限制的條件下：

- 子類別可以覆寫父類別任意的成員(方法、屬性、建構器或下標等)，用來提供限制較寬鬆的存取層級。
- 子類別成員可以存取限制更嚴格的父類別成員(因為子類別與父類別定義為同一個原始程式碼中)。

以下是一個例子：

```
// 定義一個 public 的類別 A
public class A {
    // 定義一個 private 的方法
    private func someMethod() {}

}

// 繼承自 A 的類別 B 其存取層級為 internal
// 符合 子類別的存取層級限制不能比父類別更為寬鬆
internal class B: A {
    // 可以覆寫父類別的方法 更新為較寬鬆的存取層級
    // (當然必須符合自身的存取層級)
    override internal func someMethod() {
        // 可以呼叫 存取層級限制更嚴格的父類別成員
        super.someMethod()
    }
}
```

常數, 變數, 屬性及下標

常數、變數及屬性不能擁有比它們的型別限制更為寬鬆的存取層級。例如，不能定義一個 `public` 的屬性，但它的型別卻是 `private`。而下標也不能擁有比它們的索引值或返回值型別限制更為寬鬆的存取層級。以下是一個例子：

```
// 定義一個 private 的類別
private class SomeClass {}

// 這時變數的型別為 private 則變數必須顯式的定義為 private
// 將 private 拿掉會報錯誤 因為會變成預設的 internal 則與規則不符
private var someInstance = SomeClass()
```

Getter 與 Setter

常數、變數、下標與屬性的 `Getter` 和 `Setter` 的存取層級與他們所屬型別的存取層級相同。

Setter 的存取層級可以比對應的 Getter 存取層級限制更為嚴格，可以用來控制其讀寫權限。使用方式為在 var 或 subscript 關鍵字前，加上 private(set) 或 internal(set) 來指定限制更為嚴格的存取層級。以下是一個例子：

```
// 定義一個結構 預設存取層級為 internal
struct TrackedString {
    // 將變數的 Setter 存取層級指定為 private
    private(set) var numberofEdits = 0
    var value: String = "" {
        didSet {
            // 所以在結構內部 是可以讀寫的
            numberofEdits += 1
        }
    }
}

// 宣告一個結構的變數
var stringToEdit = TrackedString()

// 每修改一次 會經由屬性觀察器來將內部變數屬性加一
stringToEdit.value = "字串修改次數會被記錄"
stringToEdit.value += "每修改一次, numberofEdits 數字會加一"
stringToEdit.value += "這行修改也會加一"

// 印出：已修改了 3 次
print("已修改了 \(stringToEdit.numberofEdits) 次")
```

你也可以在必要時為 Getter 與 Setter 顯式指定存取層級，例子如下：

```
// 顯式指定這個結構的存取層級為 public
public struct TrackedString {
    // 結合 public 與 private(set)
    // 所以這時這個變數屬性的 Setter 為 private
    // Getter 為 public
    public private(set) var numberOfEdits = 0
    public var value: String = "" {
        didSet {
            numberOfEdits += 1
        }
    }
    public init() {}
}
```

建構器

自定義建構器的存取層級限制不能比其所屬型別的存取層級寬鬆，像是一個 `internal` 的類別，不能設置一個 `public` 的建構器。而唯一例外是，當建構器為必要建構器(`required initializer`)時，其存取層級必須與所屬型別相同。

與函式或方法一樣，建構器參數的存取層級限制也不能比建構器本身嚴格。像是一個 `internal` 的建構器，不能設置一個 `private` 的參數。

預設建構器

預設建構器的存取層級與所屬型別的存取層級相同。除非當型別的存取層級為 `public`，則預設建構器會被設置為 `internal`，如果需要一個 `public` 的建構器，必須自己定義一個，如下：

```
public class SomeClass {
    public init() {}
}
```

結構的成員逐一建構器

如果結構中任意儲存屬性的存取層級為 `private`，那麼這個結構預設的成員逐一建構器的存取層級就是 `private`，否則就為 `internal`。

如果需要在其他模組也可以使用這個結構的成員逐一建構器，則必須自行定義一個 `public` 的成員逐一建構器。

協定

如果想為一個協定明確的指定存取層級，必須在定義此協定時指定。這樣可以確保這個協定只能在適當的存取層級範圍內被遵循。

協定中的每個功能都與該協定的存取層級相同，這樣才能確保協定所有功能都可以被遵循此協定的型別存取。

協定繼承

從已存在的協定繼承了一個新的協定時，這個新協定的存取層級不能比已存在協定的寬鬆。例如，定義一個 `public` 的協定時，不能繼承自一個 `internal` 的協定。

協定一致性

一個型別可以遵循一個存取層級限制更為嚴格的協定，例如，你可以定義一個 `public` 的型別，並遵循一個 `internal` 的協定。

遵循了協定的型別的存取層級，會以型別本身與遵循的協定限制較嚴格的存取層級為準，如果一個 `public` 的型別，遵循了一個 `internal` 的協定，則在此型別作為符合協定的型別時，其存取層級也是 `internal`。

當你讓一個型別遵循某個協定並滿足其所有要求後，你必須確保所有這些實作協定的部分，其存取層級不能比協定更為嚴格。例如一個 `public` 的型別，遵循了 `internal` 的協定，則實作協定的部份最嚴格只能到 `internal` 存取層級。

擴展

你可以在符合存取層級的情況下，擴展一個列舉、結構或類別，這個擴展會與擴展的對象擁有一樣的存取層級。例如，你擴展了一個 `public` 或 `internal` 型別，擴展中的成員則預設為 `internal`，與原始型別中的成員一樣。而當擴展了一

個 `private` 型別時，擴展成員則預設為 `private`。

或者，你也可以明確的指定擴展的存取層級，來讓其內成員都預設成一樣的存取層級。這個預設的存取層級仍可被個別成員所指定的存取層級覆蓋。

經由擴展來遵循協定

如果你經由擴展來遵循協定，那你就不能顯式的指定這個擴展的存取層級了。而這個協定本身的存取層級會變成預設的存取層級，且擴展中每個協定功能的實作也是一樣的預設的存取層級。

泛型

泛型型別或泛型函式的存取層級由泛型型別或泛型函式本身與泛型的型別約束參數中限制最嚴格的來確定。

型別別名

自定義的任何型別別名都會被當做不同的型別來做存取控制。型別別名不能擁有比原始型別限制更為寬鬆的存取層級。

- 一個 `public` 的型別，可以宣告 `private`、`internal` 或 `public` 的型別別名。
- 一個 `private` 的型別，僅能宣告 `private` 的型別別名，不能宣告為 `internal` 或 `public` 的型別別名。

UIKit

UIKit 是 Apple 官方提供的一個框架，用來建構常見的 UI 功能，再搭配上官方建議的介面規則，這樣層層規範下，你可以發現 App Store 上大部分應用程式的介面或是操作模式都很一致，使用久了可以很直覺的知道常見的功能如何使用。(像是回上頁很常擺在左上角。)

這一章會介紹常見的 UIKit 元件，如果你已經有使用 iPhone 的經驗，你大概可以發現幾乎都是 iPhone 上看過的東西。

如果是第一次接觸，相當建議你依序閱讀每一節的內容，可能這時候尚未有實際使用的機會，別擔心，未來在需要的時候再回來複習就可以快速上手了。

- [UIKit 初探](#)
- [文字標籤 UILabel](#)
- [文字輸入 UITextField](#)
- [輸入多行文字 UITextView](#)
- [按鈕 UIButton](#)
- [提示框 UIAlertController](#)
- [圖片 UIImageView](#)
- [選取日期時間 UIDatePicker](#)
- [選擇器 UIPickerView](#)
- [開關 UISwitch](#)
- [分段控制 UISegmentedControl](#)
- [進度條 UIProgressView](#)
- [滑桿 UISlider](#)
- [步進器 UIStepper](#)
- [網頁 UIWebView](#)
- [表格 UITableView](#)
- [網格 UICollectionView](#)
- [搜尋 UISearchController](#)
- [滑動視圖 UIScrollView](#)
- [多頁面](#)
- [導覽控制器 UINavigationController](#)
- [標籤列控制器 UITabBarController](#)
- [手勢 UIGestureRecognizer](#)

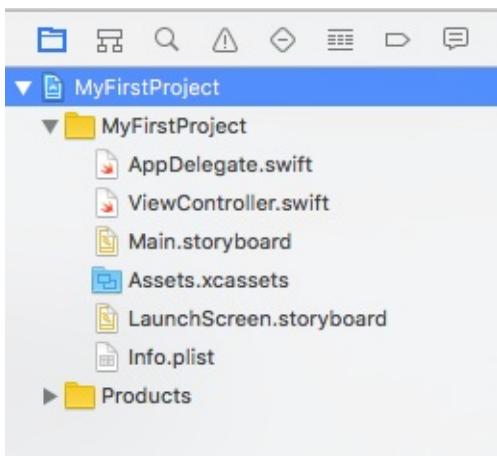
- 簡單動畫 Animations
- 儲存資訊 UserDefaults

UIKit 初探

從這節起，我們會開始介紹如何使用 UIKit 框架來建構一個應用程式。

首先在 Xcode 裡，新建一個 **Single View Application** 類型的專案，取名為 MyFirstProject 。

▼ 已經有預先建立一些檔案在專案裡面，如下圖：



首先看到有兩隻 .swift 檔

案：`AppDelegate.swift` 及 `ViewController.swift`，程式主要都是寫在這兩隻檔案裡面。

應用程式開啓時，會自 `AppDelegate.swift` 開始，這隻檔案負責應用程式的生命週期，像是啟動、閒置、進入後台、返回前台或是退出時要執行的動作。

接著看到 `ViewController.swift`，是應用程式預設的主要視圖(`View`)控制器(`Controller`)，所有需要的 UI 功能(像是按鈕、文字或圖案等等)，都必須在這個 `ViewController` 裡面建立，通常會寫在 `viewDidLoad()` 這個方法裡面。

要如何建立 UI (User Interface 使用者介面)呢？就是要用到內建的許多 UIKit 元件。

Hint

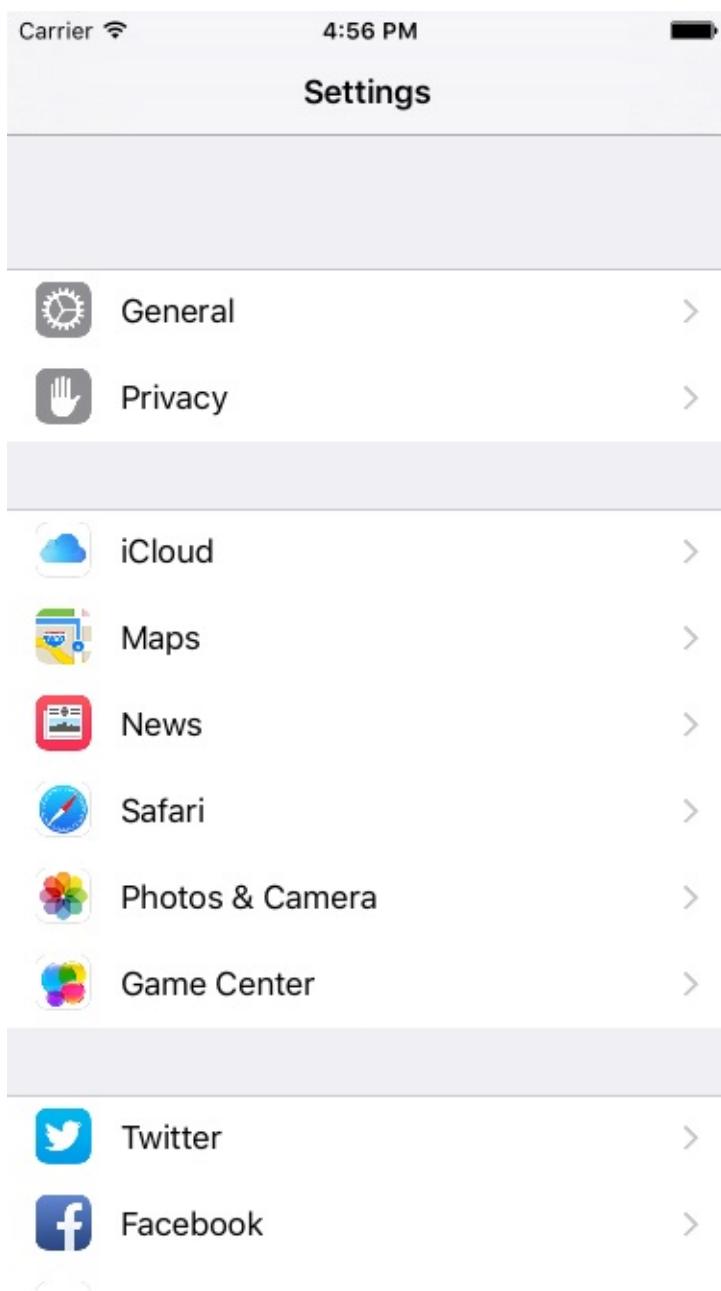
- 視圖(`View`)指的是在裝置螢幕上會呈現出來的東西，像是文字、圖片、元件(按鈕、選單、列表之類)等等。
- 視圖控制器(`ViewController`)則是負責資料的處理與編排設置視圖要如何呈現。

- 一個 `UIViewController` 負責一個畫面(即呈現視圖 `View`)的功能，未來如果有許多個畫面時，就需要各自建立一個不同的 `UIViewController`。

隨處可見的 UIKit

在使用 iPhone 的經驗裡，你可能會發現很多應用程式的樣貌及使用方式都差不多，除了 Apple 官方提供了一些 UI 規則給開發者遵循之外，其實是因為大多數功能都是使用內建的 UIKit 元件就可以完成的。

▼ iPhone 的設定 App 就是由許多的 UIKit 元件所組成，像是 `UINavigationController` 、 `UITableView` 及 `UIImageView` 等等，如下圖：



你也許可以觀察得到，這些元件都以 `UI` 為開頭，這是在 `UIKit` 設計時便特地命名的(也可以說是一種習慣)，讓你可以很輕易、清楚的明白你現在使用的東西的主要目的：這些元件都是用來建構 `UI` 的。

Hint

- 在往後的學習中，也可能會遇到一些有相同縮寫字母開頭的函式或類別，這都是在建立這些功能時特地命名的，所以你可以很清楚的知道哪些函式是用於同一種類型的功能。

建立第一個 `UIKit` 元件

最開頭的一開始，我們先介紹最基礎的一個元件：`UIView`，所有 `UIKit` 的元件(像是按鈕 `UIButton`、文字 `UITextView` 等等)都是繼承自 `UIView`，所以 `UIView` 實際很單純，只是有著每個元件都需要的、最基礎的屬性及方法。

要將元件放進畫面(即視圖 `View`)中，需要了解有三個條件：原點、尺寸以及要疊加上去的父視圖。

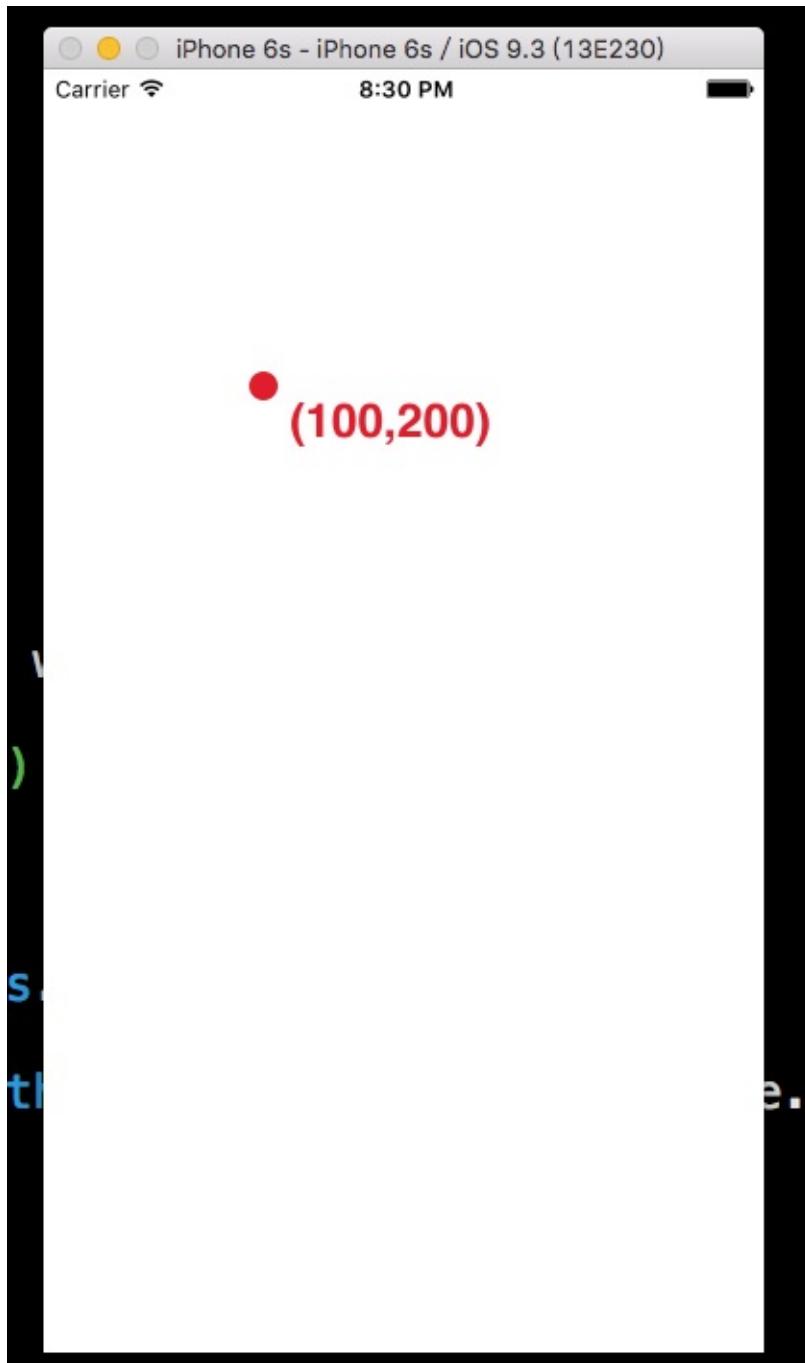
原點

`iPhone` 的原點是以左上角為 $(0, 0)$ 開始向右(`x`軸)跟向下(`y`軸)，如下圖以 `iPhone 6s` 為例：



上圖表示整個畫面中四個角各自 (x, y) 的數值。

而每個元件的原點，都是以相對於父視圖的 $(0,0)$ 為準，如果一個元件的原點為 $(100, 200)$ ，表示 x 軸相對於父視圖的 $(0,0)$ 還要向右 100， y 軸相對於父視圖的 $(0,0)$ 還要向下 200，如下圖：



尺寸

每個元件都可以設定自己的寬(`Width`)與長(`Height`)，而當有了原點及長寬之後，就可以構成一個矩形，也就是組合成了這個元件的形狀跟位置。

父視圖

建構一個應用程式的畫面其實就像疊積木一樣，最底下的就是基底，將元件一個一個疊在另一個元件上來完成。每個元件都必須疊加在一個父視圖上才能顯示出來，而最基底的視圖則是 `UIViewController` 的一個屬性，在其內的方法都可以

用 `self.view` 來表示這個基底視圖。

以程式碼來完成

綜合上述元件的三個條件，可以用程式寫成如下：

```
// 定義一個 UIView 的常數 名稱為 firstView
let firstView = UIView(frame:
    CGRect(x: 0, y: 0, width: 100, height: 100))

// 將 firstView 加入到 self.view
self.view.addSubview(firstView)
```

上述程式可以看到，定義一個 `UIView` 可以使用 `UIView(frame:)` 這個函式，函式的參數是一個 `CGRect`，`CGRect` 就是由一個原點 (`x, y`) 及一個尺寸 (`width, height`) 組成，這行即定義了這個元件的原點與尺寸。

接著可以看到在基底視圖 `self.view` 使用 `addSubview()` 方法來加入前面定義的這個 `UIView`，這樣就完成了將這個元件加進畫面的動作。

這個 `UIView` 對於其父視圖(也就是 `self.view`)的原點距離為 `(0, 0)` (其實也就是同一點)，而尺寸為 `(100, 100)` (長寬皆為 100)。

以相對尺寸來設置元件

接著介紹另一個工具：`UIScreen`，這主要是用來代表螢幕的資訊，通常是用來取得整個螢幕的尺寸，如下：

```
// 取得螢幕的尺寸
let fullScreenSize = UIScreen.mainScreen().bounds.size
```

上述程式中，`UIScreen.mainScreen()` 表示的是主畫面的資訊，其內有一個屬性為 `bounds`，`bounds` 又包含了兩個主要屬性：`origin` 及 `size`，分別是主畫面的原點及尺寸。

- `origin`：有兩個屬性 `x` 及 `y`，也就是原點。通常會使用 `CGPoint(x:,y:)` 來設置一個點。
- `size`：有兩個屬性 `width` 及 `height`，也就是尺寸，通常會使

用 `CGSize(width:, height:)` 來設置一個尺寸。

而往後都會利用這個主畫面的尺寸來為每個元件設置相對的位置。

Hint

- 因為本書所有畫面都是用純程式碼構成，為了可以較為彈性的適用各尺寸 iPhone，所以大多會以相對於整個螢幕畫面尺寸來設置 UIKit 元件的大小及位置。

在獲得了螢幕的尺寸以後，我們來將前面定義的 `UIView` 設置一個新的位置：

```
// 設置 UIView 的位置到畫面的中心
firstView.center = CGPointMake(
    x: fullScreenSize.width * 0.5,
    y: fullScreenSize.height * 0.5)
```

上述程式可以看到，使用了 `UIView` 另一個屬性 `center`，這代表著這個元件的中心點位置，也是以 `(x, y)` 來表示，所以可以用 `CGPoint(x:,y:)` 來設置一個新的點。這邊便以前面所取得的螢幕長與寬的一半數值來設置，這樣這個 `UIView` 便會被放置在畫面的正中央。

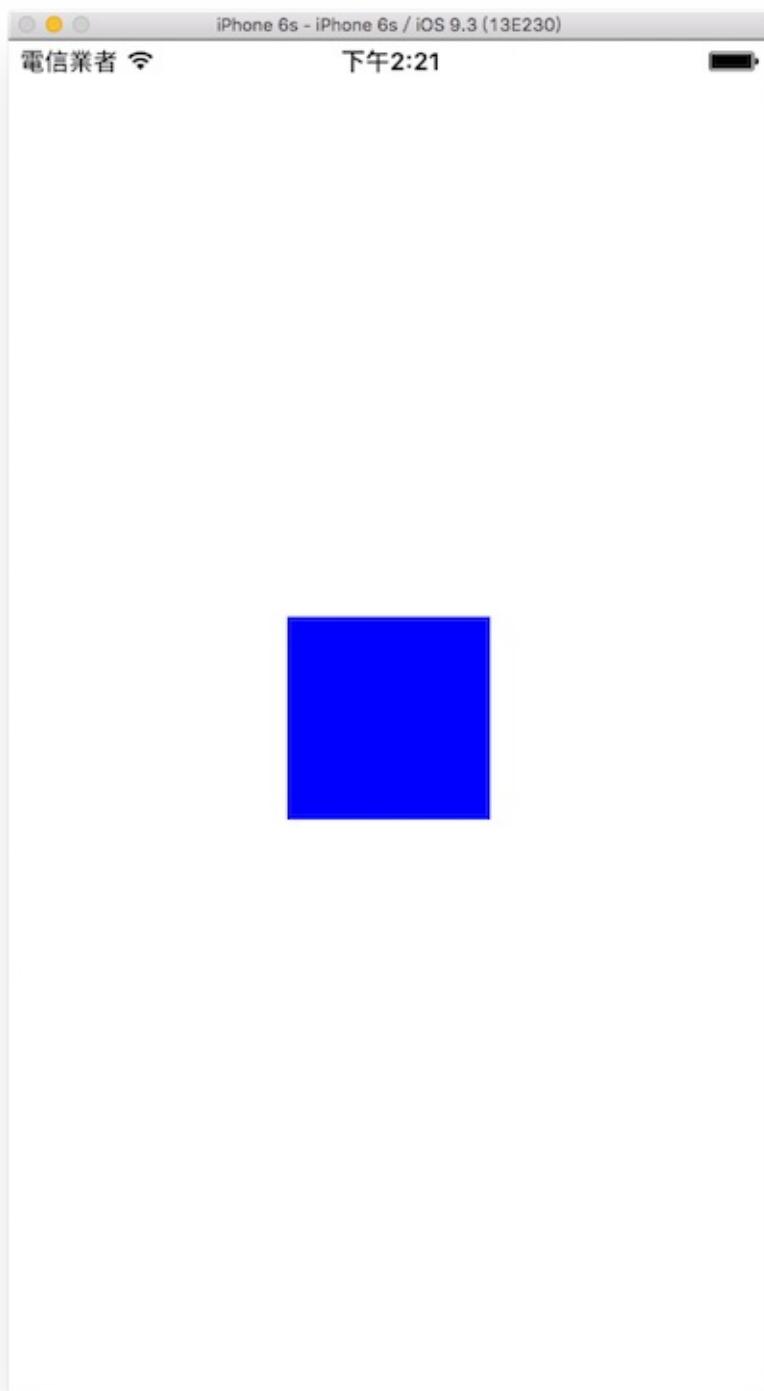
最後為了讓這個 `UIView` 可以明顯表示出來(不然原本都是白色，看不出來在哪)，為它加上一個底色：

```
// 將 UIView 的底色設置為藍色
firstView.backgroundColor = UIColor.blueColor()
```

這邊提到了 `UIView` 的一個屬性 `backgroundColor`，是用來表示這個元件的底色。

而要設置成什麼顏色，就要用到另一個工具 `UIColor`，用來產生顏色。`UIColor` 已經內建許多直接產生顏色的方法，像是上面寫的 `blueColor()` 就是藍色，或是可以使
用 `init(red:,green:,blue:,alpha:)` 來產生一個 RGB 顏色。

以上便完成了這節所要說明的內容，最後使用模擬器來看看成果，如下：



自動完成功能

在這節的學習中，你可能會發現當打字打到一半時，會出現下面這樣的提示：

```

15
16 // 定義一個 UIView 的常數 名稱為 firstView
17 let firstView = UIView
    C UIView UIView
    C UIViewController UIViewController
    C UIVisualEffect UIVisualEffect
    C UVibrancyEffect UVibrancyEffect
    C UIViewContentMode UIViewContentMode
    S UIViewAutoresizing UIViewAutoresizing
    C UIVisualEffectView UIVisualEffectView
    C UIViewAnimationCurve UIViewAnimationCurve

```

UITabBarController manages a button bar and transition view, for an application with multiple top-level modes.

這是 Xcode 內建的一個方便功能，你可以在尚未打完一個單字時，它會自動顯示這個列表，讓你使用上下鍵來選擇並完成這個單字(像是元件、屬性或方法)。

如果選擇的是方法或函式時，它完成後同時會提示你每個參數的型別為何，相當方便，如下圖：

```
// 定義一個 UIView 的常數 名稱為 firstView
let firstView = UIView(frame: CGRect(x: Double, y: Double, width: Double, height: Double))
```

有時候它不一定會自動出現，這時你可以按下 `esc` 鍵來顯示這個自動完成功能：

```

16 // 定義一個 UIView 的常數 名稱為 firstView
17 let firstView = UIView
    C UIView UIView
    C UIViewController UIViewController
    C UIViewContentMode UIViewContentMode
    S UIViewAutoresizing UIViewAutoresizing
    C UIViewAnimationCurve UIViewAnimationCurve
    C UIViewPrintFormatter UIViewPrintFormatter
    S UIViewAnimationOptions UIViewAnimationOptions
    V CGFloat UIViewNoIntrinsicMetric

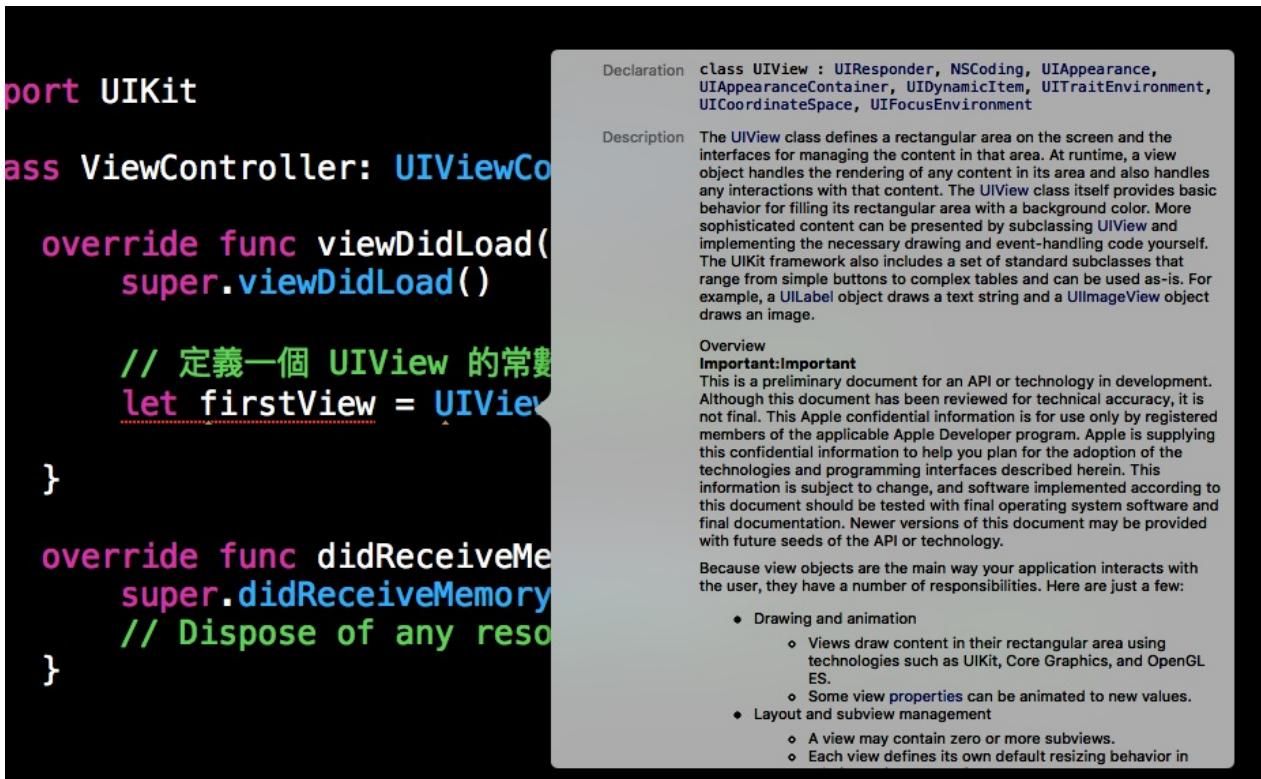
```

UITabBarController manages a button bar and transition view, for an application with multiple top-level modes.

如上圖在單字打到一半時，不會自動出現，便可以自己按 `esc` 鍵來顯示(你甚至可以都打小寫英文，也是可以找到)。

查找文件

如果對於一個元件、屬性或方法不熟悉，可以按著 option 鍵(游標會變成一個問號)再用滑鼠點選你有疑問的方法或屬性(像是按住 option 鍵再點 UIView 或 bounds 等等)，會列出更詳細的說明，如下：



或是你也可以開啟右側側邊欄，點選右邊的頁籤，當你游標指到一個元件、屬性或方法時，這邊便會顯示詳細說明，如下：

```

MyFirstProject | Build MyFirstProject: Succeeded | 2016/5/11 at 下午11:28
MyFirstProject > MyFirstProject > ViewController.swift > viewDidLoad()
Quick Help

Declaration class UIView : UIResponder, NSCoding, UIAppearance, UIAppearanceContainer, UIDynamicItem, UITraitEnvironment, UICoordinateSpace, UIFocusEnvironment

Description The UIView class defines a rectangular area on the screen and the interfaces for managing the content in that area. At runtime, a view object handles the rendering of any content in its area and also handles any interactions with that content. The UIView class itself provides basic behavior for filling its rectangular area with a background color. More sophisticated content can be presented by subclassing UIView and implementing the necessary drawing and event-handling code yourself. The UIKit framework also includes a set of standard subclasses that range from simple buttons to complex tables and can be used as-is. For example, a UILabel object draws a text string and a UIImageView object draws an image.

Overview
Important:Important
This is a preliminary document for an API or technology in development. Although this document has been reviewed for technical accuracy, it is not final. This Apple confidential information is for use only by registered members of the applicable Apple Developer Program.

class ViewController: UIViewController {
    override func viewDidLoad() {
        super.viewDidLoad()

        // 定義一個 UIView 的常數 名稱為 firstView
        let firstView = UIView(frame: CGRect(x: 0, y: 0, width: 100, height: 100))

        // 將 firstView 加入到 self.view
        self.view.addSubview(firstView)

        // 取得螢幕的尺寸
        let fullScreenSize = UIScreen.mainScreen().bounds.size

        // 設置 UIView 的位置到畫面的中心
        firstView.center = CGPoint(x: fullScreenSize.width * 0.5, y: fullScreenSize.height * 0.5)

        // 將 UIView 的底色設置為藍色
        firstView.backgroundColor = UIColor.blueColor()
    }
}

```

Hint

- 如果將 iPhone 6s 模擬器整個螢幕的 width 及 height 用 print() 印出來，你會發現寬與長分別為 375 與 667，但它實際的尺寸其實是 750 x 1334，這是因為單位不同。在程式中使用的單位是點(Point)，而實際尺寸是像素(Pixel)，剛好是兩倍，也就是以 iPhone 6s 來說，一個點會包含 2x2 個像素。

範例

本節範例程式碼放在 [uikit/uikit_intro](#)

文字標籤 **UILabel**

UILabel 是用來顯示文字的元件，如果你需要單純顯示像是標題、人名、數字或是
一段文字，就很適合使用 **UILabel**。下圖為本節內容的目標：



建立一個 **UILabel**

首先在 Xcode 裡，新建一個 **Single View Application** 類型的專案，取名為 ExUILabel 。

使用 `UILabel(frame:)` 來建立一個 **UILabel** :

```
// 使用 UILabel(frame:) 建立一個 UILabel
let myLabel = UILabel(frame: CGRect(x: 0, y: 0, width: 300, height: 80))
```

使用 `CGRect(x:,y:,width:,height:)` 來設定原點及尺寸。

設定 **UILabel** 文字屬性

UILabel 的文字可以設定很多屬性，以下是常用到的：

```
// 文字內容  
myLabel.text = "Swift 起步走"  
  
// 文字顏色  
myLabel.textColor = UIColor.redColor()  
  
// 文字的字型與大小  
myLabel.font = UIFont(name: "Helvetica-Light", size: 20)  
  
// 可以再修改文字的大小  
myLabel.font = myLabel.font.fontWithSize(24)  
  
// 或是可以使用系統預設字型 並設定文字大小  
myLabel.font = UIFont.systemFontOfSize(36)  
  
// 設定文字位置 置左、置中或置右等等  
myLabel.textAlignment = NSTextAlignment.Right  
  
// 也可以簡寫成這樣  
myLabel.textAlignment = .Center  
  
// 文字行數  
myLabel.numberOfLines = 1  
  
// 文字過多時 過濾的方式  
myLabel.lineBreakMode = NSLineBreakMode.ByTruncatingTail  
  
// 陰影的顏色 如不設定則預設為沒有陰影  
myLabel.shadowColor = UIColor.blackColor()  
  
// 陰影的偏移量 需先設定陰影的顏色  
myLabel.shadowOffset = CGSize(width: 2, height: 2)
```

上述內容在第一次看到時可能覺得那麼多東西怎麼記得著哪個要用哪個，但請記得常使用 `esc` 鍵來顯示自動完成功能，看看這個元件有什麼屬性或方法可以使用，以及按住 `option` 鍵再用滑鼠點選你有疑問的方法或屬性(像是按住 `option` 鍵再點 `textColor` 或 `font` 等等)，會列出更詳細的說明。

另外注意到 `UILabel` 的 `textAlignment` 及 `lineBreakMode` 這兩個屬性的值，使用到 [Swift 列舉](#)的特性，所以可以簡寫成 `.Center` 這種方式，如果忘記了可以回去該章節再看看。

還有關於 `lineBreakMode` 這個屬性，指的是文字過多時，要如何處理，像是 `.ByTruncatingTail` 就是保留開頭的文字，而將後面超出的文字以 `...` 代替。

通用屬性

前一節提到大多數的元件都是繼承自 `UIView`，所以很多屬性都是通用的，如下：

```
// 可以單獨設置新的 x 或 y  
myLabel.bounds.origin.x = 50  
myLabel.bounds.origin.y = 100  
// 或是使用 CGPoint(x:,y:) 設置新的原點  
myLabel.bounds.origin = CGPointMake(x: 60, y: 120)  
  
// 可以單獨設置新的 width 或 height  
myLabel.bounds.size.width = 200  
myLabel.bounds.size.height = 100  
// 或是使用 CGSize(width:,height:) 設置新的尺寸  
myLabel.bounds.size = CGSizeMake(width: 250, height: 80)  
  
// 或是也可以一起設置新的原點及尺寸  
myLabel.bounds = CGRectMake(  
    x: 60, y: 120, width: 250, height: 80)  
  
// 取得螢幕的尺寸  
let screenSize = UIScreen.mainScreen().bounds.size  
  
// 設置於畫面的中心點  
myLabel.center = CGPointMake(  
    x: screenSize.width * 0.5,  
    y: screenSize.height * 0.5)  
  
// UILabel 的背景顏色  
myLabel.backgroundColor = UIColor.orangeColor()  
  
// 加入到畫面中  
self.view.addSubview(myLabel)
```

範例

本節範例程式碼放在 [uikit/uilabel](#)

文字輸入 UITextField

應用程式很常遇到需要輸入文字的時候，像是登入時的帳號或密碼，這時會需要用到 UITextField，本節的目標如下：



建立一個 UITextField

首先在 Xcode 裡，新建一個 **Single View Application** 類型的專案，取名為 ExUITextField。

以下使用 `UITextField(frame:)` 建立一個 `UITextField`，並介紹其常用的屬性：

```
// 使用 UITextField(frame:) 建立一個 UITextField
let myTextField = UITextField(frame: CGRect(x: 0, y: 0, width: 200, height: 50))

// 尚未輸入時的預設顯示提示文字
myTextField.placeholder = "請輸入文字"

// 輸入框的樣式 這邊選擇圓角樣式
myTextField.borderStyle = .RoundedRect

// 輸入框右邊顯示清除按鈕時機 這邊選擇當編輯時顯示
myTextField.clearButtonMode = .WhileEditing

// 輸入框適用的鍵盤 這邊選擇 適用輸入 Email 的鍵盤(會有 @ 跟 . 可供輸入)
myTextField.keyboardType = .EmailAddress

// 鍵盤上的 return 鍵樣式 這邊選擇 Done
myTextField.returnKeyType = .Done

// 輸入文字的顏色
myTextField.textColor = UIColor.whiteColor()

// UITextField 的背景顏色
myTextField.backgroundColor = UIColor.lightGrayColor()
```

上述程式中

的 `borderStyle` 、 `clearButtonMode` 、 `keyboardType` 和 `returnKeyType` 屬性，都各自有多種選擇可供使用，你可以在輸入 `.` 之後按下 `esc` 鍵，會列出來所有可以選擇的項目，其他的項目就交由你自己玩玩看。

UITextField 也可以使用多種通用屬性，上述程式只設置了 `textColor` 及 `backgroundColor`，但其實像是 `font` 或 `textAlignment`，它也可以設置。

委任模式

基於 Swift 協定的特性，這邊會開始介紹一個極為重要且在後續學習中會大量見到的設計模式：委任模式(Delegation)。如果覺得不太熟悉的話，可以先回到前面的章節看看。

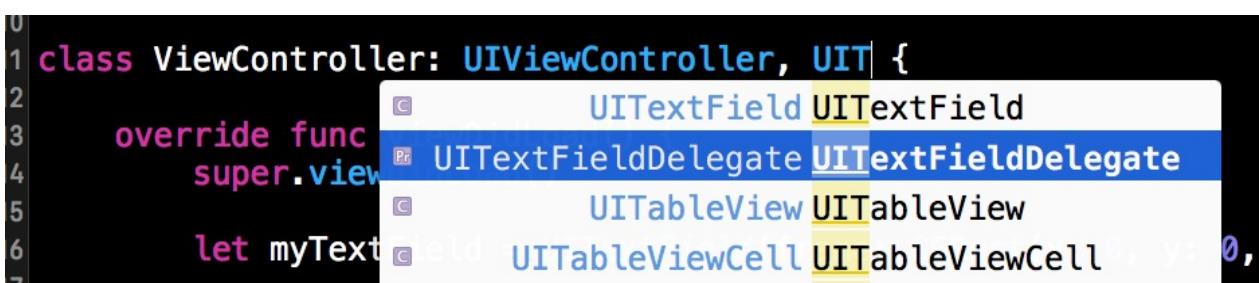
委任模式的意思是，在設計元件或功能時，會定義數個不等的方法(method)，但只會定義出這些方法的名稱，而這些方法要做什麼事，則是要交由委任(**delegate**)的對象來實作。

這邊使用 UITextField 來示範你的第一個使用委任模式的應用程式。例如在設計一個 UITextField (輸入框) 時，通常用來輸入內容的鍵盤上會有一個完成的按鈕 (通常位於鍵盤的右下角 `Return` 鍵)。原始設計上都不會實際把按下完成按鈕會做什麼事寫死在元件裡，而是設計成委任模式，表示我這邊有一個事件 (即按下完成後要幹嘛) 待完成，我會交給委任的對象來實作。

而交付委任對象的方式，就是要設置 `delegate` 屬性，如下：

```
// 委任模式( Delegation )所要實作方法的對象
// 這邊就是交由 self 也就是 ViewController 本身
myTextField.delegate = self
```

以上設置完成後，你會得到一個紅色的錯誤，先別緊張，因為這是提示你為 UITextField 設置了一個委任的對象，但這個對象尚未遵循這個委任，所以要為 ViewController 加上協定 `UITextFieldDelegate`，如下圖：



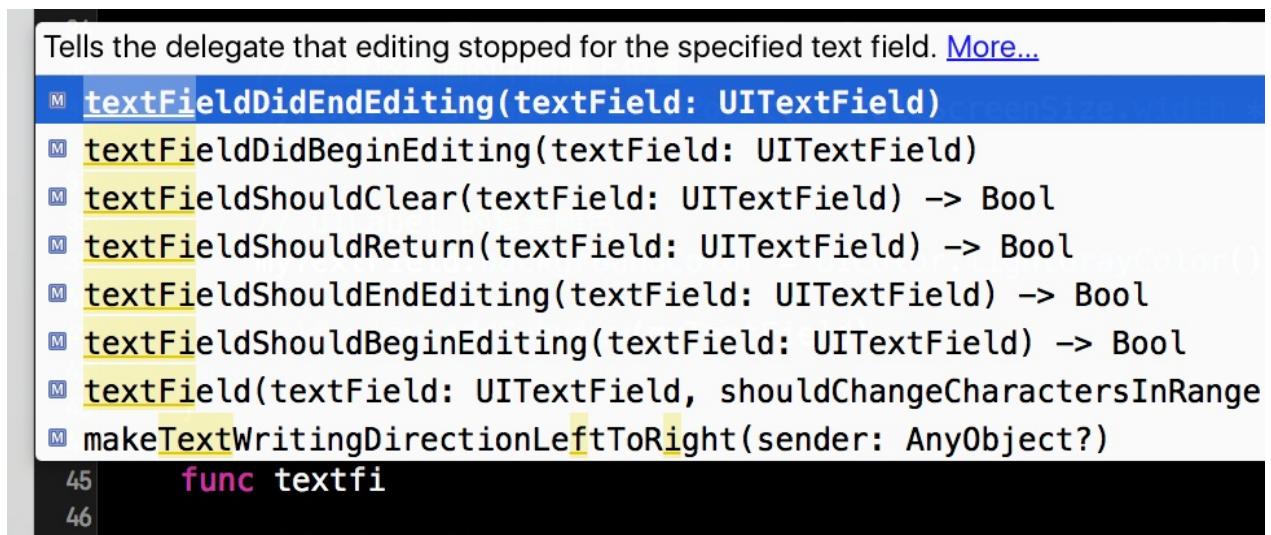
設置完成的 ViewController 會長得如下：

```
class ViewController: UIViewController, UITextFieldDelegate {  
    // 省略內容  
}
```

Hint

- 如果看不太懂，請往前參考 [Swift 協定](#) 章節。

這時你就可以依據委任的協定來建立應該要實作的方法，如下圖：



上圖可以看到，為 ViewController 建立一個委任模式需要的新方法，輸入 `func textfi` 這樣到一半時，會自動顯示多個方法可供實作，以 UITextField 這邊來說就是輸入框各個動作階段，像是圖中前三個方法就是結束輸入時、開始輸入時及按下清除按鈕時。

以下為實作按下 `return` 按鈕時的方法：

```
func textFieldShouldReturn(textField: UITextField) -> Bool {  
    // 結束編輯 把鍵盤隱藏起來  
    self.view.endEditing(true)  
  
    return true  
}
```

上述程式就是當按下 `return` 按鈕時，會將鍵盤隱藏起來。

Hint

- 委任模式會將需要實作的方法設置為 必須 或 可選，如果設置為必須的方法，則委任對象一定要將這個方法實作出來，不然會直接吐錯誤給你，叫你一定要實作，而如果設置為可選的方法，則是依照你的需求，看要實作哪些方法，才實際實作出來，像是上面示範的 UITextField 委任的方法就都是可選的方法，不用每一個都實作出來。

將 UITextField 放入畫面

最後就是將 UITextField 放入畫面中，結束這節的內容：

```
// 取得螢幕的尺寸
let fullScreenSize = UIScreen.mainScreen().bounds.size

// 設置於畫面的中間偏上位置
myTextField.center = CGPointMake(x: fullScreenSize.width * 0.5, y: fullScreenSize.height * 0.3)

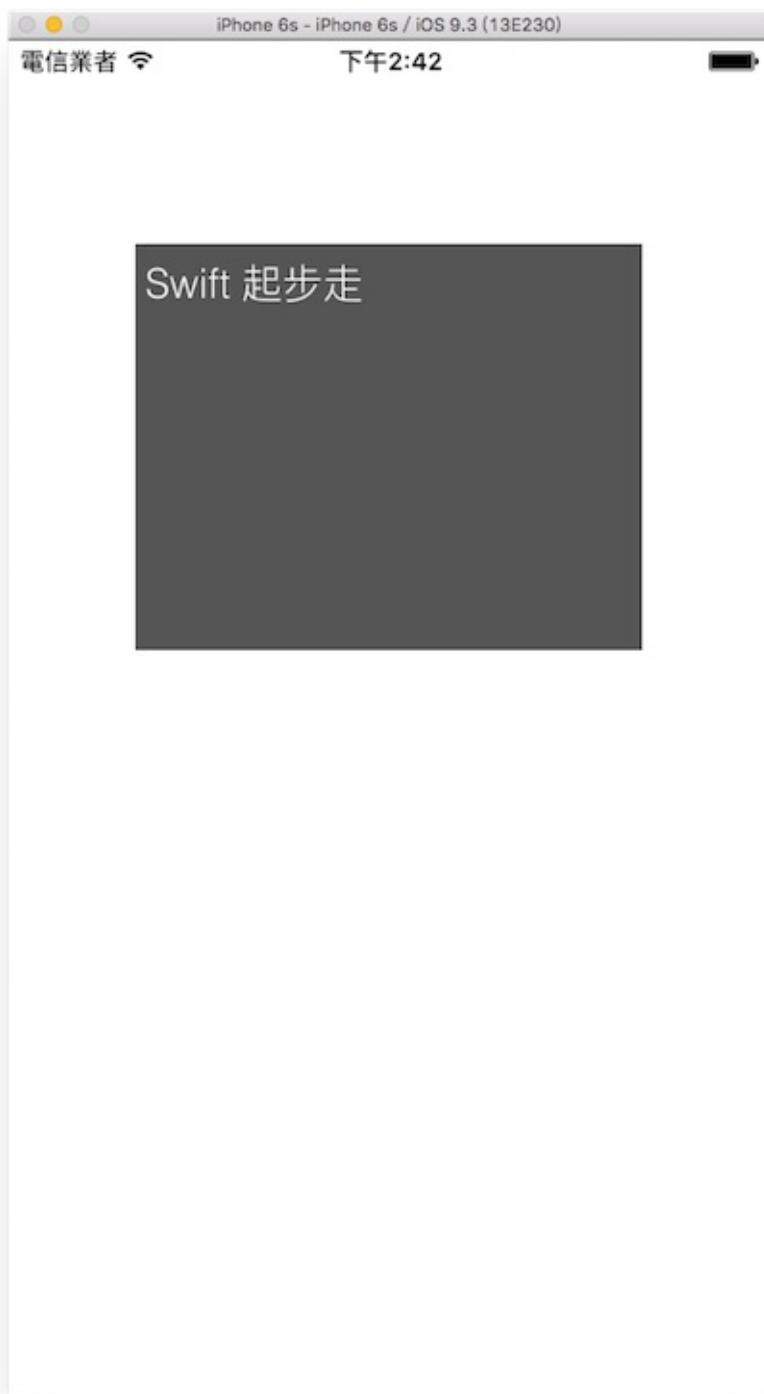
// 加入畫面
self.view.addSubview(myTextField)
```

範例

本節範例程式碼放在 [uikit/uitextfield](#)

輸入多行文字 UITextView

UITextView 與 UITextField 有點類似，時常用在輸入或顯示多行文字，很常見的應用就是通訊軟體的文字輸入框，你可以輸入多行文字一次送出。本節的目標如下：



建立一個 UITextView

首先在 Xcode 裡，新建一個 **Single View Application** 類型的專案，取名為 ExUITextView。

這節建立 UITextView 時，我們會為 ViewController 建立一個 UITextView 屬性，以因應後續的使用，如下：

```
class ViewController: UIViewController {  
    // 建立一個 UITextView 的屬性  
    var myTextView: UITextView!  
  
    override func viewDidLoad() {  
        // 省略  
    }  
}
```

接著便在 viewDidLoad() 中建立 UITextView，可以看到這邊是使用屬性 myTextView 來生成，如下：

```
// 使用 UITextView(frame:) 建立一個 UITextView  
myTextView = UITextView(  
    frame: CGRect(x: 0, y: 0, width: 250, height: 200))
```

與 UITextField 類似，有許多一樣的屬性可以設定，如下：

```
// 背景顏色  
myTextView.backgroundColor = UIColor.darkGrayColor()  
  
// 文字顏色  
myTextView.textColor = UIColor.whiteColor()  
  
// 文字字型及大小  
myTextView.font = UIFont(name: "Helvetica-Light", size: 20)  
  
// 文字向左對齊  
myTextView.textAlignment = .Left  
  
// 預設文字內容  
myTextView.text = "Swift 起步走"  
  
// 適用的鍵盤樣式 這邊選擇預設的  
myTextView.keyboardType = .Default  
  
// 鍵盤上的 return 鍵樣式 這邊選擇預設的  
myTextView.returnKeyType = .Default
```

另外常用的屬性還有如下：

```
// 文字是否可以被編輯  
myTextView.editable = true  
  
// 文字是否可以被選取  
myTextView.selectable = true
```

最後就是將元件加入畫面中：

```
// 取得螢幕的尺寸
let fullScreenSize = UIScreen.mainScreen().bounds.size

// 設置於畫面的中間偏上位置
myTextView.center = CGPointMake(
    x: fullScreenSize.width * 0.5,
    y: fullScreenSize.height * 0.3)

// 加入畫面
self.view.addSubview(myTextView)
```

自訂選取內容後的選項

當顯示一段文字時，可以在選取文字後，對這段文字進行操作，像是全選、複製、貼上等等。除此之外，我們還可以自己增加需要的功能選項，如下：

```
// 建立兩個新的選項
let mail = UIMenuItem(
    title: "寄送",
    action: #selector(ViewController.sendMail))
let facebook = UIMenuItem(
    title: "FB",
    action: #selector(ViewController.sendFB))

// 建立選單
let menu = UIMenuController.sharedMenuController()

// 將新的選項加入選單
menu.menuItems = [mail,facebook]
```

上述程式中，建立選項時可以看到 `UIMenuItem()` 的其中一個參數為 `action`，這代表按下這個選項後要執行的動作，而這個動作會寫在 `#selector(ViewController.sendMail)`，也就是 `ViewController` 的 `sendMail` 方法中，所以我們還需要為這兩個新的選項加上新的 `ViewController` 的方法，用來處理按下後的動作，如下：

```
func sendMail() {  
    print("sendMail")  
}  
  
func sendFB() {  
    print("sendFB")  
}
```

選項建立好後，就可以看到下面這樣的結果，有額外的選項可供選擇：



Hint

- 如果要為不同輸入框元件設置不同的選取內容後的選單，可以將上述建立選單的程式，改放到元件取得焦點時的方法。以 UITextView 來說，就可以寫在 `textViewDidBeginEditing(textView: UITextView)` 這個委任函式中(這節範例沒有設置 `UITextViewDelegate` 委任模式，必須再自己加上)。而有不同元件(像是 UITextField)同時存在時，就是在各自取得焦點(也就是 `becomeFirstResponder`)時設置不同選單。

按空白處隱藏鍵盤

介紹前一節的 UITextField 時，是使用鍵盤上的 `return` 鍵來隱藏鍵盤，而使用 UITextView 時，有時候會需要換行就沒辦法使用 `return` 鍵來結束編輯，這時就需要使用別的方式來替換。這邊介紹一個常用的方式，以按輸入框之外其他空白處的方式來隱藏鍵盤。

```
// 增加一個觸控事件
let tap = UITapGestureRecognizer(
    target: self,
    action: #selector(ViewController.hideKeyboard(_:)))

tap.cancelsTouchesInView = false

// 加在最基底的 self.view 上
self.view.addGestureRecognizer(tap)
```

上述程式可以看到同樣需要一個 `action` 來處理這個觸控事件，所以需要為 `ViewController` 加上一個新的方法 `hideKeyboard(_:)`，如下：

```
func hideKeyboard(tapG:UITapGestureRecognizer){
    // 除了使用 self.view.endEditing(true)
    // 也可以用 resignFirstResponder()
    // 來針對一個元件隱藏鍵盤
    self.myTextView.resignFirstResponder()
}
```

要隱藏鍵盤，除了前一節介紹的 `self.view.endEditing(true)` 之外，也可以對元件使用 `resignFirstResponder()` 方法。

`First Responder` 代表的是目前畫面中，處於焦點狀態的元件，而當輸入文字時，這個輸入框就是 `First Responder`，所以如果要隱藏鍵盤，當然就是將 `First Responder` 取消，也就是使用 `resignFirstResponder()` 方法。

Hint

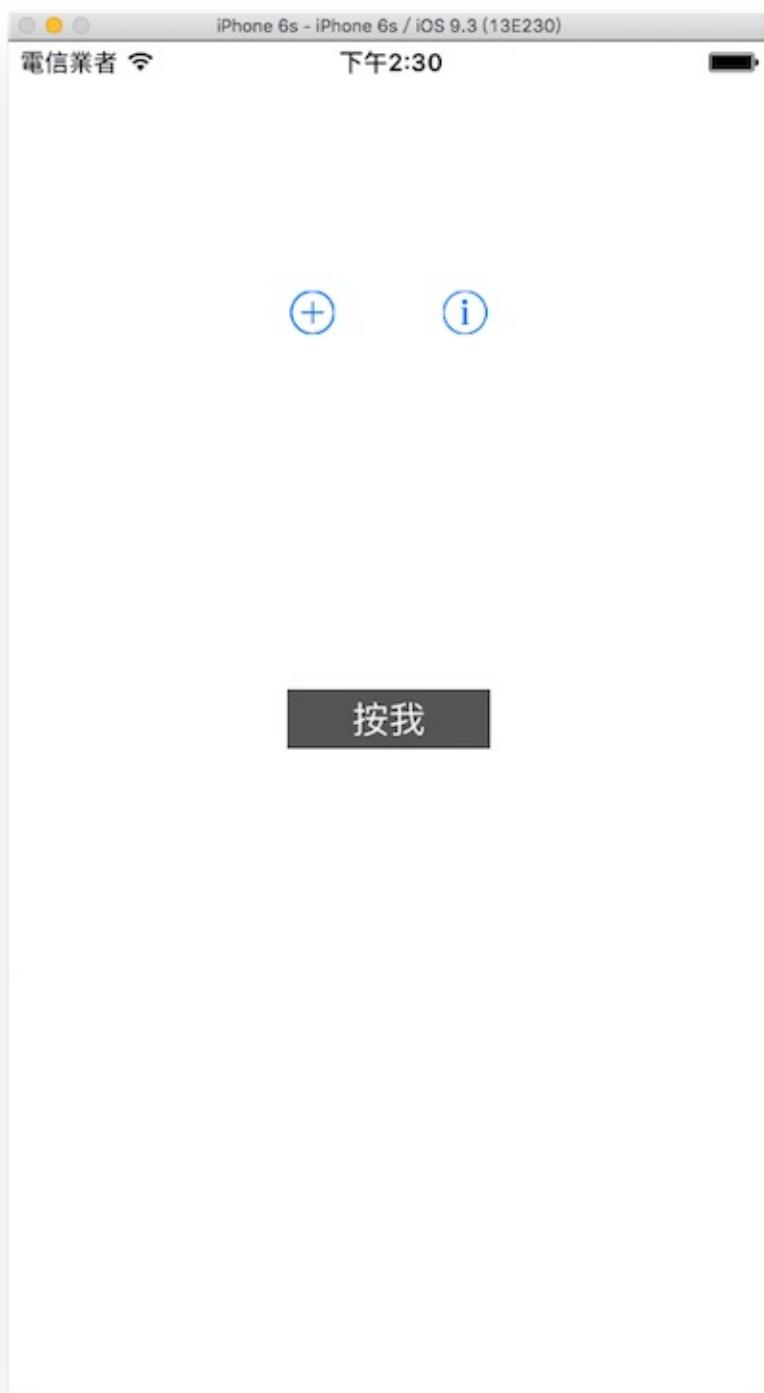
- 實際上，觸控事件是由 `UIControl` 所負責的動作。`UIControl` 是繼承自 `UIView` 的子類別，而這些需要觸控事件的元件則是再繼承自 `UIControl`，像是往後會學習到的 `UIButton`、`UISwitch`、`UISlider` 都是。

範例

本節範例程式碼放在 [uikit/uitextview](#)

按鈕 UIButton

按鈕是最常見的控制元件，按下按鈕後可以執行許多動作，例如送出表單、切換頁面等等，應用程式的互動幾乎都與 UIButton 有關，是不可缺少的元件。本節的目標如下：



首先在 Xcode 裡，新建一個 **Single View Application** 類型的專案，取名為 ExUIButton。

一開始先在 `viewDidLoad()` 中取得螢幕尺寸，以及設置 `self.view` 的底色，以供後續使用，如下：

```
// 取得螢幕的尺寸
let fullScreenSize = UIScreen.mainScreen().bounds.size

// 為基底的 self.view 設置底色
self.view.backgroundColor = UIColor.whiteColor()
```

建立預設樣式的按鈕

UIKit 提供幾個內建預設樣式的按鈕，都是系統中可以看得到的樣式，如果你有一樣的需求，可以直接使用這樣的方式建立按鈕，如下：

```
// 預設的按鈕樣式
var myButton = UIButton(type: .ContactAdd)
myButton.center = CGPointMake(
    x: fullScreenSize.width * 0.4,
    y: fullScreenSize.height * 0.2)
self.view.addSubview(myButton)

myButton = UIButton(type: .InfoLight)
myButton.center = CGPointMake(
    x: fullScreenSize.width * 0.6,
    y: fullScreenSize.height * 0.2)
self.view.addSubview(myButton)
```

上述程式分別建立了加號 + 及驚嘆號 ! 的按鈕(請參考上圖)，這邊可以注意到，這兩個按鈕用的是同一個變數 `myButton`，因為各別都使用了 `UIButton(type:)` 來初始化一個按鈕，所以可以這樣使用。當然你也可以分別建立兩個常數(像是 `let myButton1, myButton2` 這樣)。

建立自定義的按鈕

如果要建立自定義的按鈕，則是使用 `UIButton(frame:)`，如下：

```
// 使用 UIButton(frame:) 建立一個 UIButton
myButton = UIButton(
    frame: CGRect(x: 0, y: 0, width: 100, height: 30))

// 按鈕文字
myButton.setTitle("按我", forState: .Normal)

// 按鈕文字顏色
myButton.setTitleColor(
    UIColor.whiteColor(),
    forState: .Normal)

// 按鈕是否可以使用
myButton.enabled = true

// 按鈕背景顏色
myButton.backgroundColor = UIColor.darkGrayColor()

// 按鈕按下後的動作
myButton.addTarget(
    self,
    action: #selector(ViewController.clickButton),
    forControlEvents: .TouchUpInside)

// 設置位置並加入畫面
myButton.center = CGPoint(
    x: screenSize.width * 0.5,
    y: screenSize.height * 0.5)
self.view.addSubview(myButton)
```

上述程式要注意到的是 `addTarget(target:, action:, forControlEvents:)` 這個方法，用來負責按下按鈕後的動作，這是由 `UIControl` 所提供的方法，再繼承給 `UIButton`。這個方法的參數說明如下：

- **target**：當事件發生時，要呼叫哪一個物件。
- **action**：呼叫的物件要執行的方法，以 `#selector()` 來指定，`ViewController.clickButton` 指的就是 `ViewController` 類別

的 `clickButton` 方法。將 `ViewController` 省略掉，只填入 `clickButton` 也可以。

- `forControlEvents`：觸發的事件。(這裡則是按下事件。)

Hint

- 觸控事件是由 `UIControl` 所負責的動作。`UIControl` 是繼承自 `UIView` 的子類別，而這些需要觸控事件的元件則是再繼承自 `UIControl`，除了 `UIButton` 之外，還有像是往後會學習到的 `UISwitch`、`UISlider` 都是。
- `#selector()` 也可以填入需要參數的方法，像是 `clickButton(_:)` 或是 `clickButton(_:otherSender:)`，請再另外定義這兩個不同的方法。請記得方法(以及函式)就算名稱一樣，參數不同的話也是代表不同的方法。更多函式說明請參考[函式](#)章節。

接著則是為 `ViewController` 新增按下按鈕需要的方法 `clickButton()`，如下：

```
func clickButton() {  
    // 為基底的 self.view 的底色在黑色與白色兩者間切換  
    if self.view.backgroundColor!.isEqual(  
        UIColor.whiteColor()) {  
        self.view.backgroundColor =  
            UIColor.blackColor()  
    } else {  
        self.view.backgroundColor =  
            UIColor.whiteColor()  
    }  
}
```

Hint

- 如果要為按鈕設置圖片，請參考[設置使用圖片的按鈕](#)。

範例

本節範例程式碼放在 [uikit/uibutton](#)

提示框 **UIAlertController**

在一些情況下，你可能會需要向使用者確認動作是否執行，或是提示一些內容，這時可以使用 `UIAlertController`。不過因為這是一個強制將畫面焦點集中於這個提示的動作，所以要確認真的需要才使用，如果過於濫用會帶來不佳的使用者體驗。

本節的目標會建立五個按鈕，按下後會分別示範不同功能的提示框，如下：



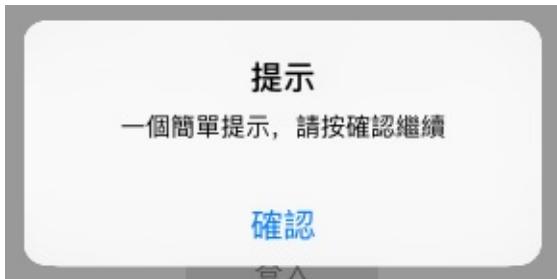
首先在 Xcode 裡，新建一個 **Single View Application** 類型的專案，取名為 **ExUIAlert**。

一開始先在 `viewDidLoad()` 中取得螢幕尺寸，以供後續使用，如下：

```
// 取得螢幕的尺寸  
let screenSize = UIScreen.mainScreen().bounds.size
```

簡單的提示

首先先建立一個簡單的提示，可以用來說明一些簡單的內容，如下：



先建立一個按鈕：

```
// 設置一個按下會顯示簡單提示的按鈕
var myButton = UIButton(type: .System)
myButton.frame = CGRect(
    x: 0, y: 0, width: 100, height: 30)
myButton.setTitle("簡單提示", forState: .Normal)
myButton.backgroundColor = UIColor.init(
    red: 0.9, green: 0.9, blue: 0.9, alpha: 1)
myButton.addTarget(
    nil,
    action: #selector(ViewController.simpleHint),
    forControlEvents: .TouchUpInside)
myButton.center = CGPoint(
    x: screenSize.width * 0.5,
    y: screenSize.height * 0.15)
self.view.addSubview(myButton)
```

接著在 `ViewController` 新增一個按下按鈕後要執行動作的方法：

```

func simpleHint() {
    // 建立一個提示框
    let alertController = UIAlertController(
        title: "提示",
        message: "一個簡單提示，請按確認繼續",
        preferredStyle: .Alert)

    // 建立[確認]按鈕
    let okAction = UIAlertAction(
        title: "確認",
        style: .Default,
        handler: {
            (action: UIAlertAction!) -> Void in
            print("按下確認後，閉包裡的動作")
        })
    alertController.addAction(okAction)

    // 顯示提示框
    self.presentViewController(
        alertController,
        animated: true,
        completion: nil)
}

```

上述程式中，首先看到提示框控制器：`UIAlertController(title:, message:, preferredStyle:)`，是用來建立一個提示框的函式，參數如下：

- `title`：提示的標題，會以粗體顯示。
- `message`：提示的內容。
- `preferredStyle`：提示框的類型，這邊填寫 `.Alert`，會顯示在畫面中間，另外如果填寫 `.ActionSheet` 則是顯示在畫面底部。

接著可以看到用來建立提示框按鈕的 `UIAlertAction(title:, style:, handler:)`，參數如下：

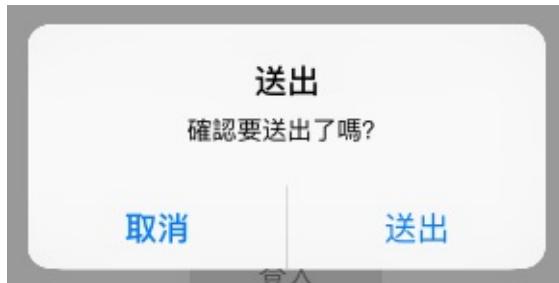
- `title`：按鈕的文字。
- `style`：按鈕的樣式，可選擇 `.Cancel`、`.Default`、`.Destructive` 等等。
- `handler`：按下按鈕後要執行的動作，是一個型別為 `(action:`

`UIAlertAction!) -> Void` 的閉包，如果不要有動作則是填入 `nil`。

按鈕設置好之後，再使用 `alertController.addAction()` 方法，將按鈕加入到提示框控制器中。

最後則是對 `self` 使用 `presentViewController()` 方法來顯示提示框。

有確認與取消的提示框



先建立一個按鈕：

```
// 設置一個按下會顯示確認及取消提示的按鈕
myButton = UIButton(type: .System)
myButton.frame = CGRectMake(
    x: 0, y: 0, width: 100, height: 30)
myButton.setTitle("送出", forState: .Normal)
myButton.backgroundColor = UIColor.init(
    red: 0.9, green: 0.9, blue: 0.9, alpha: 1)
myButton.addTarget(
    nil,
    action: #selector(ViewController.confirm),
    forControlEvents: .TouchUpInside)
myButton.center = CGPointMake(
    x: screenSize.width * 0.5,
    y: screenSize.height * 0.3)
self.view.addSubview(myButton)
```

接著在 `ViewController` 新增一個按下按鈕後要執行動作的方法：

```
func confirm() {
    // 建立一個提示框
    let alertController = UIAlertController(
        title: "送出",
        message: "確認要送出了嗎？",
        preferredStyle: .Alert)

    // 建立[取消]按鈕
    let cancelAction = UIAlertAction(
        title: "取消",
        style: .Cancel,
        handler: nil)
    alertController.addAction(cancelAction)

    // 建立[送出]按鈕
    let okAction = UIAlertAction(
        title: "送出",
        style: .Default,
        handler: nil)
    alertController.addAction(okAction)

    // 顯示提示框
    self.presentViewController(
        alertController,
        animated: true,
        completion: nil)
}
```

Hint

- 按鈕的順序會如同使用 `addAction()` 方法加入的順序。
- 習慣上會將 `Cancel` 按鈕放在左邊。
- `Cancel` 按鈕(也就是 `style` 設置為 `.Cancel` 的 `UIAlertAction()`)在一個提示框控制器中只能有一個。

警示提示框



先建立一個按鈕：

```
// 設置一個按下會提示刪除的按鈕
myButton = UIButton(type: .System)
myButton.frame = CGRect(
    x: 0, y: 0, width: 100, height: 30)
myButton.setTitle("刪除", forState: .Normal)
myButton.backgroundColor = UIColor.init(
    red: 0.9, green: 0.9, blue: 0.9, alpha: 1)
myButton.addTarget(
    nil,
    action: #selector(ViewController.deleteSomething),
    forControlEvents: .TouchUpInside)
myButton.center = CGPoint(
    x: fullScreenSize.width * 0.5,
    y: fullScreenSize.height * 0.45)
self.view.addSubview(myButton)
```

接著在 `ViewController` 新增一個按下按鈕後要執行動作的方法：

```
func deleteSomething() {
    // 建立一個提示框
    let alertController = UIAlertController(
        title: "刪除",
        message: "刪除字樣會變紅色的",
        preferredStyle: .Alert)

    // 建立[取消]按鈕
    let cancelAction = UIAlertAction(
        title: "取消",
        style: .Cancel,
        handler: nil)
    alertController.addAction(cancelAction)

    // 建立[刪除]按鈕
    let okAction = UIAlertAction(
        title: "刪除",
        style: .Destructive,
        handler: nil)
    alertController.addAction(okAction)

    // 顯示提示框
    self.presentViewController(
        alertController,
        animated: true,
        completion: nil)
}
```

上述程式可以看到建立刪除按鈕的 `UIAlertAction()` 的參數 `style` 設置為 `.Destructive`，這會將按鈕文字改為紅色的，表示這個按鈕選項是警示用的，一般用在可能會改變或刪除資料的動作上。

有輸入框的提示框



先建立一個按鈕：

```
// 設置一個按下會提示登入的按鈕
myButton = UIButton(type: .System)
myButton.frame = CGRect(
    x: 0, y: 0, width: 100, height: 30)
myButton.setTitle("登入", forState: .Normal)
myButton.backgroundColor = UIColor.init(
    red: 0.9, green: 0.9, blue: 0.9, alpha: 1)
myButton.addTarget(
    nil,
    action: #selector(ViewController.login),
    forControlEvents: .TouchUpInside)
myButton.center = CGPoint(
    x: fullScreenSize.width * 0.5,
    y: fullScreenSize.height * 0.6)
self.view.addSubview(myButton)
```

接著在 `ViewController` 新增一個按下按鈕後要執行動作的方法：

```
func login() {
    // 建立一個提示框
    let alertController = UIAlertController(
        title: "登入",
        message: "請輸入帳號與密碼",
        preferredStyle: .Alert)

    // 建立兩個輸入框
    alertController.addTextFieldWithConfigurationHandler {
        (textField: UITextField!) -> Void in
```

```
        textField.placeholder = "帳號"
    }
alertController.addTextFieldWithConfigurationHandler {
    (textField: UITextField!) -> Void in
    textField.placeholder = "密碼"

    // 如果要輸入密碼 這個屬性要設定為 true
    textField.secureTextEntry = true
}

// 建立[取消]按鈕
let cancelAction = UIAlertAction(
    title: "取消",
    style: .Cancel,
    handler: nil)
alertController.addAction(cancelAction)

// 建立[登入]按鈕
let okAction = UIAlertAction(
    title: "登入",
    style: UIAlertActionStyle.Default) {
    (action: UIAlertAction!) -> Void in
    let acc =
        ( alertController.textFields?.first)!
            as UITextField
    let password =
        ( alertController.textFields?.last)!
            as UITextField

    print("輸入的帳號為 : \(acc.text)")
    print("輸入的密碼為 : \(password.text)")
}
alertController.addAction(okAction)

// 顯示提示框
self.presentViewController(
    alertController,
    animated: true,
    completion: nil)
}
```

要在提示框中加入輸入框，要使用提示框控制器 `UIAlertController` 的 `addTextFieldWithConfigurationHandler()` 方法加入，參數為一個型別為 `(textField: UITextField!) -> Void` 的閉包，可以在這個閉包內為輸入框作額外的設定。

最後在按下登入按鈕函式的閉包中，獲得輸入的帳號與密碼，並使用 `print()` 印出來。

Hint

- `as` 的用法請參考[向下型別轉換](#)。
- 這部份示範的閉包有使用尾隨閉包的特性，請參考[尾隨閉包](#)。

從底部彈出的提示框



先建立一個按鈕：

```
// 設置一個按下會從底部彈出提示的按鈕  
myButton = UIButton(type: .System)  
myButton.frame = CGRect(  
    x: 0, y: 0, width: 100, height: 30)  
myButton.setTitle("底部提示", forState: .Normal)  
myButton.backgroundColor = UIColor.init(  
    red: 0.9, green: 0.9, blue: 0.9, alpha: 1)  
myButton.addTarget(  
    nil,  
    action: #selector(ViewController.bottomAlert),  
    forControlEvents: .TouchUpInside)  
myButton.center = CGPoint(  
    x: fullScreenSize.width * 0.5,  
    y: fullScreenSize.height * 0.75)  
self.view.addSubview(myButton)
```

接著在 `ViewController` 新增一個按下按鈕後要執行動作的方法：

```
func bottomAlert() {
    // 建立一個提示框
    let alertController = UIAlertController(
        title: "底部提示",
        message: "這個提示會從底部彈出",
        preferredStyle: .ActionSheet)

    // 建立[取消]按鈕
    let cancelAction = UIAlertAction(
        title: "取消",
        style: .Cancel,
        handler: nil)
    alertController.addAction(cancelAction)

    // 建立[確認]按鈕
    let okAction = UIAlertAction(
        title: "確認",
        style: .Default,
        handler: nil)
    alertController.addAction(okAction)

    // 顯示提示框
    self.presentViewController(
        alertController,
        animated: true,
        completion: nil)
}
```

上述程式中可以看到 `UIAlertController()` 的參數 `preferredStyle` 設置為 `.ActionSheet`，這便會讓提示框從畫面底部彈出來，要使用哪種方式則是看你自己需求。

Hint

- 底部提示框不能加入輸入框。
- 如果有加入 `Cancel` 按鈕到底部提示框，它會永遠在最底下一個。

範例

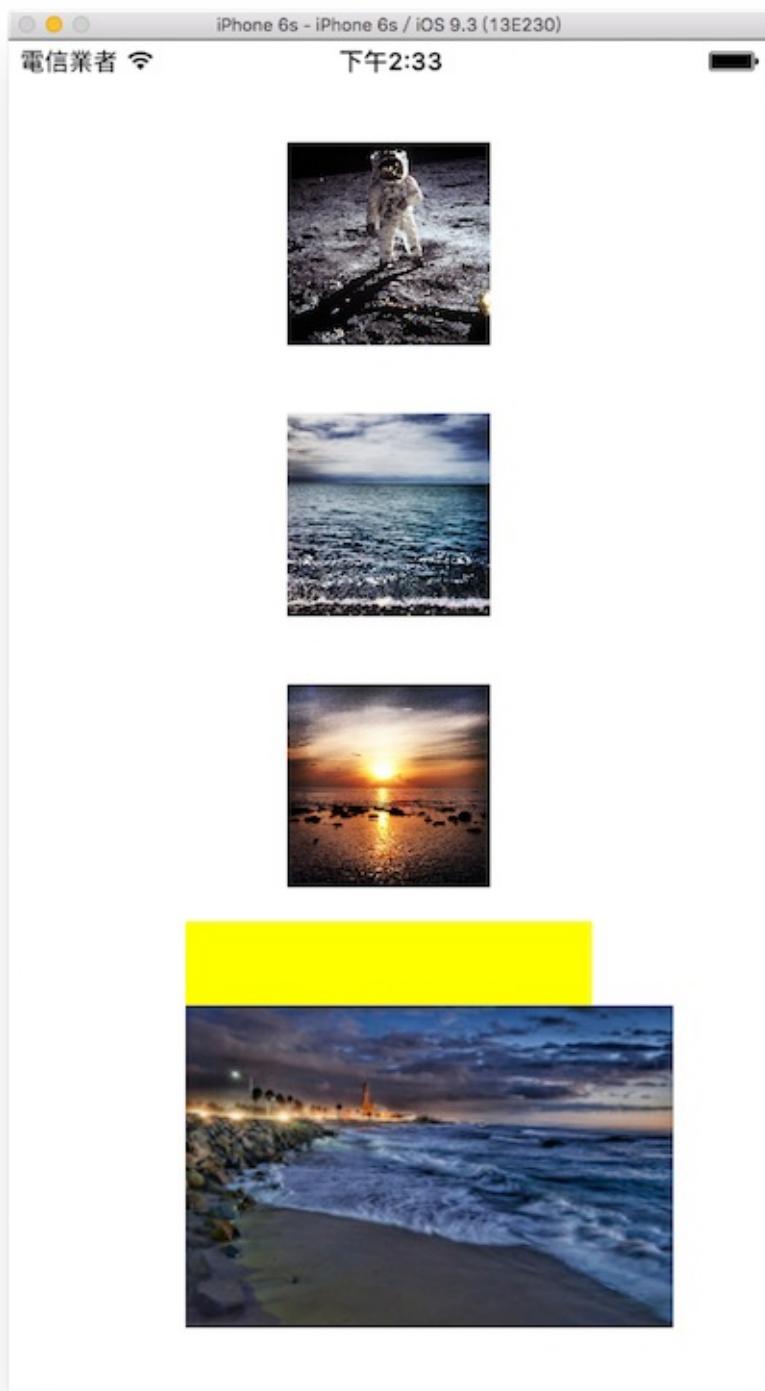
本節範例程式碼放在 [uikit/uialert](#)

圖片 UIImageView

顯示圖片主要是使用 UIImageView 及 UIImage 來完成，本節會介紹兩個範例，第一個範例會示範三種建立 UIImageView 的方式以及顯示模式，第二個範例則會示範一個自動輪播圖片的功能。

建立 UIImageView

這個範例的目標如下圖：



首先在 Xcode 裡，新建一個 **Single View Application** 類型的專案，取名為 **ExUIImageView**。

一開始先以加入檔案的方式加入四張圖片，並在 `viewDidLoad()` 中取得螢幕尺寸，以供後續使用，如下：

```
// 取得螢幕的尺寸  
let fullScreenSize = UIScreen.mainScreen().bounds.size
```

▼ 第一個建立 UIImageView 的方式為使用 `UIImageView(frame:)` :

```
// 使用 UIImageView(frame:) 建立一個 UIImageView  
var myImageView = UIImageView(  
    frame: CGRect(  
        x: 0, y: 0, width: 100, height: 100))  
  
// 使用 UIImage(named:) 放置圖片檔案  
myImageView.image = UIImage(named: "01.jpg")  
  
// 設置新的位置並放入畫面中  
myImageView.center = CGPointMake(  
    x: fullScreenSize.width * 0.5,  
    y: fullScreenSize.height * 0.15)  
self.view.addSubview(myImageView)
```

上述程式可以看到，`UIImageView` 不是直接放置圖片檔案名稱，而是要藉由 `UIImage(named:)` 建立，再設置給 `UIImageView` 的屬性 `image`，未來其他元件需要使用圖片時，也都是使用 `UIImage(named:)` 設置圖片檔案。

▼ 第二個建立 UIImageView 的方式為使用 `UIImageView(image:)`，在初始化時直接給一個 `UIImage`，後續再設定尺寸：

```
// 使用 UIImageView(image:) 建立一個 UIImageView
myImageView = UIImageView(
    image: UIImage(named: "02.jpg"))

// 設置原點及尺寸
myImageView.frame = CGRect(
    x: 0, y: 0, width: 100, height: 100)

// 設置新的位置並放入畫面中
myImageView.center = CGPoint(
    x: fullScreenSize.width * 0.5,
    y: fullScreenSize.height * 0.35)
self.view.addSubview(myImageView)
```

▼ 第三個建立 UIImageView 的方式為使用 `UIImageView(image:, highlightedImage:)`，在初始化時除了直接設置一個 `UIImage` 外，還設置了一個 `highlighted` 狀態時的 `UIImage`：

```
// 使用 UIImageView(image:, highlightedImage:)
// 建立一個 UIImageView
myImageView = UIImageView(
    image: UIImage(named: "02.jpg"),
    highlightedImage: UIImage(named: "03.jpg"))

// 設置原點及尺寸
myImageView.frame = CGRect(
    x: 0, y: 0, width: 100, height: 100)

// 設置圖片 highlighted 狀態
myImageView.highlighted = true

// 設置新的位置並放入畫面中
myImageView.center = CGPoint(
    x: fullScreenSize.width * 0.5,
    y: fullScreenSize.height * 0.55)
self.view.addSubview(myImageView)
```

上面三個 UIImageView 的圖片剛好都為正方形，所以可以剛好適合視圖的尺寸，而如果當圖片尺寸與 UIImageView 的尺寸不一樣時，就需要使用 UIImageView 的一個屬性 `contentMode` 來設置顯示模式：

```
// 建立一個 UIImageView
myImageView = UIImageView(
    image: UIImage(named: "04.jpg"))
myImageView.frame = CGRect(
    x: 0, y: 0, width: 200, height: 200)

// 設置背景顏色
myImageView.backgroundColor = UIColor.yellowColor()

// 設置圖片顯示模式
myImageView.contentMode = .BottomLeft

// 設置新的位置並放入畫面中
myImageView.center = CGPoint(
    x: fullScreenSize.width * 0.5,
    y: fullScreenSize.height * 0.8)
self.view.addSubview(myImageView)
```

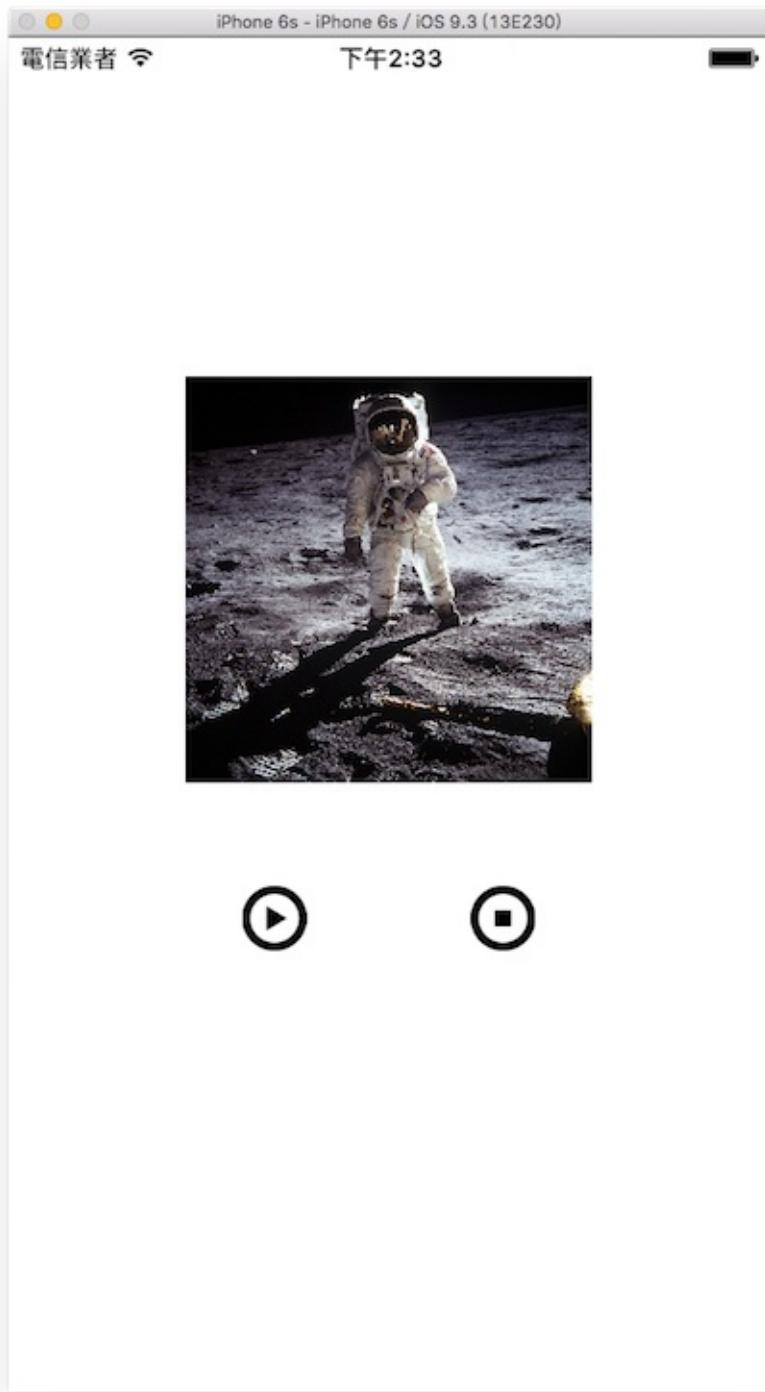
上述程式可以看到 UIImageView 的尺寸為 200x200，而圖片的尺寸為 240x159，所以沒有辦法剛好放進去，這時就使用 `contentMode` 這個屬性來設定顯示模式，這邊示範設置為 `.BottomLeft`，也就是以左下角為準。將 UIImageView 底色設為黃色，所以你可以看到最後會超出原本 UIImageView 設置的尺寸，並在上面露出了一些底色。

Hint

- 預設 `contentMode` 屬性的值為 `ScaleToFill`，會自動縮放圖片以填滿 UIImageView 的尺寸。
- `contentMode` 屬性其他可以設定的模式，還有以長或寬為準的縮放、重新裁切或是以八個方向或是中心為準等等，你可以自己測試看看。

自動輪播圖片

這個範例的目標如下圖，圖片會自動輪播，且有兩個按鈕可以播放及停止輪播：



首先在 Xcode 裡，新建一個 **Single View Application** 類型的專案，取名為 ExAutoPlayImage。

一開始先以加入檔案的方式加入三張輪播的圖片及兩張按鈕的圖片。

首先為 `ViewController` 建立一個 `myImageView` 屬性：

```
class ViewController: UIViewController {  
    var myImageView: UIImageView!  
  
    // 省略  
}
```

以及在 `viewDidLoad()` 中取得螢幕尺寸，以供後續使用，如下：

```
// 取得螢幕的尺寸  
let fullScreenSize = UIScreen.mainScreen().bounds.size
```

設置輪播圖片的資訊

```
myImageView = UIImageView(  
    frame: CGRect(x: 0, y: 0, width: 200, height: 200))  
  
// 建立一個陣列 用來放置要輪播的圖片  
let imgArr = [  
    UIImage(named:"01.jpg")!,  
    UIImage(named:"02.jpg")!,  
    UIImage(named:"03.jpg")!]  
  
// 設置要輪播的圖片陣列  
myImageView.animationImages = imgArr  
  
// 輪播一次的總秒數  
myImageView.animationDuration = 6  
  
// 要輪播幾次 (設置 0 則為無限次)  
myImageView.animationRepeatCount = 0  
  
// 開始輪播  
myImageView.startAnimating()  
  
// 設置位置及放入畫面中  
myImageView.center = CGPoint(  
    x: fullScreenSize.width * 0.5,  
    y: fullScreenSize.height * 0.4)  
self.view.addSubview(myImageView)
```

建立圖片輪播一樣是使用 UIImageView，以下是需要的屬性說明：

- **animationImages**：為一個型別為 [UIImage] 的陣列，其內使用 `UIImage(named:)` 生成的圖片因為是可選的(optional)，所以必須將其解析(unwrap，即加上驚嘆號！)，當然你必須確定這些圖片檔案都存在，不然可能會導致程式錯誤並中止。
- **animationDuration**：輪播一次的總秒數，如果一張圖片要顯示 2 秒，則乘上圖片張數 3 張，所以這裡設置為 6 秒。
- **animationRepeatCount**：要輪播的次數，如果設置 0 則是無限次。

另外還有兩個關於輪播的方法：

- `startAnimating()`：開始圖片輪播。
- `stopAnimating()`：停止圖片輪播。

設置使用圖片的按鈕

```
// 建立一個播放按鈕
let playButton = UIButton(
    frame: CGRect(x: 0, y: 0, width: 64, height: 64))

// 設置播放按鈕的圖片
playButton.setImage(
    UIImage(named: "play"), forState: .Normal)

// 設置按下播放按鈕的動作的方法
playButton.addTarget(
    self,
    action: #selector(ViewController.play),
    forControlEvents: .TouchUpInside)

// 設置位置及放入畫面中
playButton.center = CGPoint(
    x: fullScreenSize.width * 0.35,
    y: fullScreenSize.height * 0.65)
self.view.addSubview(playButton)

// 建立一個停止按鈕
let stopButton = UIButton(
    frame: CGRect(x: 0, y: 0, width: 64, height: 64))

// 設置停止按鈕的圖片
stopButton.setImage(
    UIImage(named: "stop"), forState: .Normal)

// 設置按下停止按鈕的動作的方法
stopButton.addTarget(
    self,
    action: #selector(ViewController.stop),
    forControlEvents: .TouchUpInside)
```

```
// 設置位置及放入畫面中
stopButton.center = CGPoint(
    x: fullScreenSize.width * 0.65,
    y: fullScreenSize.height * 0.65)
self.view.addSubview(stopButton)
```

按鈕除了可以使用文字之外，也可以設置成圖片，利用 `UIButton` 的方法 `setImage()`，設置一個 `UIImage` 給它即可。`UIButton` 的詳細說明請參考前節說明。

接著 `ViewController` 新增兩個按下按鈕後執行動作的方法：

```
func play() {
    print("play images auto play")
    myImageView.startAnimating()
}

func stop() {
    print("stop images auto play")
    myImageView.stopAnimating()
}
```

Hint

- 圖片如果為 `png` 檔案類型，使用 `UIImage(named:)` 生成元件時，可以不用寫副檔名 `png`，如上面程式中的 `UIImage(named:"play")`，程式會自己找到 `play.png` 的圖片檔案。
- 另外你可能會發現範例檔案內，播放按鈕的圖片除了 `play.png` 外，還有 `play@2x.png` 跟 `play@3x.png` 檔案，這是為了因應不同解析度的 iPhone，可以設置不同解析度的圖片，像是 iPhone 5、iPhone 6 及 iPhone 6s 使用 `@2x` 的圖片，而 iPhone 6 Plus 及 iPhone 6s Plus 則是使用 `@3x` 的圖片。只要將檔案名稱設定好，程式就會自己找到檔案，當然如果沒有 `@2x` 及 `@3x` 的圖片檔案，就會一律使用同樣尺寸的圖片檔案。

圖片來源

- <https://www.flickr.com/photos/134525588@N04/20344150965>

- <https://www.flickr.com/photos/voilaquoi/18911800758>
- <https://www.flickr.com/photos/nasacommons/5136519916/>
- <https://www.flickr.com/photos/136245400@N03/24183123165/>
- https://www.iconfinder.com/icons/106223/play_icon
- https://www.iconfinder.com/icons/106221/stop_icon

範例

本節範例程式碼放在 [uikit/uiimageview](#)

選取日期時間 UIDatePicker

UIDatePicker 可以用來選取日期或時間，以下為本節的目標，設置一個 UIDatePicker，並在改變日期時間時同步更新 UILabel 的內容：



首先在 Xcode 裡，新建一個 **Single View Application** 類型的專案，取名為 ExUIDatePicker。

先為 `ViewController` 建立兩個屬性，如下：

```
class ViewController: UIViewController {  
    var myDatePicker: UIDatePicker!  
    var myLabel: UILabel!  
  
    // 省略  
}
```

以及在 `viewDidLoad()` 中取得螢幕尺寸，以供後續使用，如下：

```
// 取得螢幕的尺寸  
let fullScreenSize = UIScreen.mainScreen().bounds.size
```

建立 UIDatePicker

```
// 使用 UIDatePicker(frame:) 建立一個 UIDatePicker  
myDatePicker = UIDatePicker(frame: CGRect(  
    x: 0, y: 0,  
    width: fullScreenSize.width, height: 100))  
  
// 設置 UIDatePicker 格式  
myDatePicker.datePickerMode = .DateAndTime  
  
// 選取時間時的分鐘間隔 這邊以 15 分鐘為一個間隔  
myDatePicker.minuteInterval = 15  
  
// 設置預設時間為現在時間  
myDatePicker.date = NSDate()  
  
// 設置 NSDate 的格式  
let formatter = NSDateFormatter()  
  
// 設置時間顯示的格式  
formatter.dateFormat = "yyyy-MM-dd HH:mm"
```

```
// 可以選擇的最早日期時間
let fromDateTime = formatter.dateFromString("2016-01-02 18:08")

// 設置可以選擇的最早日期時間
myDatePicker.minimumDate = fromDateTime

// 可以選擇的最晚日期時間
let endDateTime = formatter.dateFromString("2017-12-25 10:45")

// 設置可以選擇的最晚日期時間
myDatePicker.maximumDate = endDateTime

// 設置顯示的語言環境
myDatePicker.locale = NSLocale(
    localeIdentifier: "zh_TW")

// 設置改變日期時間時會執行動作的方法
myDatePicker.addTarget(self,
    action:
        #selector(ViewController.datePickerChanged),
    forControlEvents: .ValueChanged)

// 設置位置並加入到畫面中
myDatePicker.center = CGPointMake(
    x: screenSize.width * 0.5,
    y: screenSize.height * 0.4)
self.view.addSubview(myDatePicker)
```

UIDatePicker 常用的屬性如下：

- **datePickerMode**：可以顯示的模式，有只顯示時間、只顯示日期、顯示日期與時間以及倒數計時器。
- **minuteInterval**：時間選取時的分鐘間隔，單位是分鐘。
- **date**：設置預設顯示的日期時間。
- **minimumDate**：可以選擇的最早日期時間。
- **maximumDate**：可以選擇的最晚日期時間。
- **locale**：顯示的語言環境。

NSDate 與 NSDateFormatter 會在稍後介紹。

接著建立一個 UILabel，在 UIDatePicker 變更時可以同步更新內容：

```
// 建立一個 UILabel 來顯示改變日期時間後的結果
myLabel = UILabel(frame: CGRect(
    x: 0, y: 0,
    width: fullScreenSize.width, height: 50))
myLabel.backgroundColor = UIColor.lightGrayColor()
myLabel.textAlignment = .Center
myLabel.textColor = UIColor.blackColor()
myLabel.center = CGPoint(
    x: fullScreenSize.width * 0.5,
    y: fullScreenSize.height * 0.15)
self.view.addSubview(myLabel)
```

最後在 ViewController 中新建一個改變 UIDatePicker 時會執行動作的方法：

```
func datePickerChanged(datePicker:UIDatePicker) {
    // 設置要顯示在 UILabel 的日期時間格式
    let formatter = DateFormatter()
    formatter.dateFormat = "yyyy-MM-dd HH:mm"

    // 更新 UILabel 的內容
    myLabel.text = formatter.stringFromDate(
        datePicker.date)
}
```

NSDate 與 NSDateFormatter

在 Swift 中要處理有關於日期或時間的功能時，都是使用 NSDate 類別，而要怎麼呈現日期時間的顯示格式，則是需要搭配 NSDateFormatter 類別來使用：

```

// 簡單的建立了一個目前時間的常數
// 型別為 NSDate
let nowDate = NSDate()

// 你可以使用型別為 NSDate 的常數來對日期時間作處理
// 例如將現在時間加上一天
// dateByAddingTimeInterval 的單位是秒
let oneDayAfter = nowDate.dateByAddingTimeInterval(
    60 * 60 * 24)

// 現在處理完後
// 如果要將這個 NSDate 常數轉換成字串
// 則是要利用 NSDateFormatter
let formatter = NSDateFormatter()

// 先設置日期時間顯示的格式
formatter.dateFormat = "yyyy-MM-dd HH:mm"

// 再利用 NSDateFormatter 的 stringFromDate 方法
// 將 oneDayAfter 轉換成字串
let oneDayAfterToString =
    formatter.stringFromDate(oneDayAfter)

// 印出：明天這一刻的日期時間
print(oneDayAfterToString)

// 或是你從一個顯示日期時間的字串 轉換成 NSDate
let xmasDate = formatter.dateFromString(
    "2016-12-25 21:30")
// 這時便可以再對這個 NSDate 常數作處理

```

UIDatePicker 的 `date` 、 `minimumDate` 及 `maximumDate` 屬性，型別都是 `NSDate`，所以如果你要設置或修改這些屬性時，都必須使用 `NSDate` 及 `NSDateFormatter` 類別來做處理。

以下列出常使用到的日期時間顯示的格式，你可以用在 `NSDateFormatter` 的屬性 `dateFormat`，依照需求來組合你要的格式：

- `yyyy`：西元年份，像是 2015、1998。

- yy：西元年份末兩位數，像是 15、95。
- MMMM：月份，像是 December、January。
- MMM：月份簡寫，像是 Oct、Feb。
- MM：以數字代表月份，像是 08、12。
- dd：日期，像是 07、31。
- EEEE：星期幾，像是 Saturday、Monday。
- EEE：星期幾的簡寫，像是 Sun、Wed。
- HH：24 小時制的時。
- hh：12 小時制的時。
- mm：分。
- ss：秒。

前面範例設置的格式為 `formatter.dateFormat = "yyyy-MM-dd HH:mm"`，你也可以設置為年月日，像是 `formatter.dateFormat = "yyyy 年 MM 月 dd 日"`。

Hint

- NSDate 與 NSDateFormatter 都是屬於 Foundation 框架的類別，未來也會使用到一些以 NS 開頭的類別，都是用來執行一些基礎功能的工具。

範例

本節範例程式碼放在 [uikit/uidepicker](#)

選擇器 **UIPickerView**

UIPickerView 與 UIDatePicker 有點類似，但彈性更大，當你需要有一個或多個選擇項目時，可以設定自定義的選項及數目。

本節會介紹兩個範例，第一個會介紹如何使用 UIPickerView，接著則會介紹 UIPickerView 與 UIDatePicker 如何跟 UITextField 結合在一起應用。

建立 UIPickerView

這個範例會建立一個含有兩個選項的 UIPickerView，可以分別選擇自己的選項，選完後可以再執行自定義的動作，目標如下：



首先在 Xcode 裡，新建一個 **Single View Application** 類型的專案，取名為 **ExUIPickerView**。

一開始先以新增檔案的方式加入一個繼承自 **UIViewController** 的檔案，命名為 **MyViewController**，這要用來實作 **UIPickerView** 委任模式的方法。

先看到 **ViewController**，我們在 **viewDidLoad()** 裡建立一個 **UIPickerView**，設定好它的位置、尺寸及委任對象，並加入到畫面中：

```
// 取得螢幕的尺寸
let fullScreenSize = UIScreen.mainScreen().bounds.size

// 建立 UIPickerView 設置位置及尺寸
let myPickerView = UIPickerView(frame: CGRect(
    x: 0, y: fullScreenSize.height * 0.3,
    width: fullScreenSize.width, height: 150))

// 新增另一個 UIViewController
// 用來實作委任模式的方法
let myViewController = MyViewController()

// 必須將這個 UIViewController 加入
self.addChildViewController(myViewController)

// 設定 UIPickerView 的 delegate 及 dataSource
myPickerView.delegate = myViewController
myPickerView.dataSource = myViewController

// 加入到畫面
self.view.addSubview(myPickerView)
```

上述程式可以看到，利用稍前新建的 `MyViewController.swift` 檔案來做為 `UIPickerView` 的委任對象，在前面章節 [以 UITextField 為例的委任模式](#) 中，是將委任對象設置為 `self`，也就是本身所在的 `ViewController`，而這邊則是使用另一個方式，將委任對象設置為另一個檔案，交由它來實作。

兩個方式都可以，端看你如何設計，像如果需要設置委任對象的元件太多時，你就可以分別為元件各自建立所屬的檔案來實作委任模式的方法，未來如果有需要這個獨立檔案也可以很快速的拿過去使用。

委任模式

接著我們看到新建的 `MyViewController`，首先為它加上委任模式需要的協定 `UIPickerViewDelegate` 與 `UIPickerViewDataSource`，如下：

```

class MyViewController: UIViewController,
    UIPickerViewDelegate, UIPickerViewDataSource {
    // 省略
}

```

加上協定後你會看到出現如下圖的錯誤，因為這個委任模式有設置必須實作的方法：

```

10
11 class MyViewController: UIViewController, UIPickerViewDelegate, UIPickerViewDataSource {
12     // Type 'MyViewController' does not conform to protocol 'UIPickerViewDataSource'
13
14
15
16
17
18
19
20

```

所以你必須實作它設置為必須實作的方法，這邊必須實作的方法有兩個，如下：

```

11 class MyViewController: UIViewController, UIPickerViewDelegate, UIPickerViewDataSource {
12
13     func numberOfComponentsInPickerView(pickerView: UIPickerView) -> Int {
14         return 1
15     }
16
17     func pickerView(pickerView: UIPickerView, numberOfRowsInComponent component: Int) -> Int {
18         return 1
19     }
20

```

上圖中返回的數字 1 是暫時的，稍後會完成這兩個方法的內容。

再來先為 `MyViewController` 加上四個屬性，稍後會使用於示範中：

```

class MyViewController: UIViewController,
    UIPickerViewDelegate, UIPickerViewDataSource {

    let week = [
        "星期日", "星期一", "星期二", "星期三",
        "星期四", "星期五", "星期六"]
    let meals = ["早餐", "午餐", "晚餐", "宵夜"]
    var whatDay = "星期日"
    var whatMeal = "早餐"

    // 省略
}

```

接著在 `MyViewController` 內實作其餘需要的委任的方法，以完善 `UIPickerView` 的內容，如下：

```
// UIPickerViewDataSource 必須實作的方法：
```

```
// UIPickerView 有幾列可以選擇
func numberOfComponentsInPickerView(
    pickerView: UIPickerView) -> Int {
    return 2
}

// UIPickerViewDataSource 必須實作的方法：
// UIPickerView 各列有多少行資料
func pickerView(pickerView: UIPickerView,
    numberOfRowsInComponent component: Int) -> Int {
    // 設置第一列時
    if component == 0 {
        // 返回陣列 week 的成員數量
        return week.count
    }

    // 否則就是設置第二列
    // 返回陣列 meals 的成員數量
    return meals.count
}

// UIPickerView 每個選項顯示的資料
func pickerView(pickerView: UIPickerView,
    titleForRow row: Int, forComponent component: Int)
-> String? {
    // 設置第一列時
    if component == 0 {
        // 設置為陣列 week 的第 row 項資料
        return week[row]
    }

    // 否則就是設置第二列
    // 設置為陣列 meals 的第 row 項資料
    return meals[row]
}

// UIPickerView 改變選擇後執行的動作
func pickerView(pickerView: UIPickerView,
    didSelectRow row: Int, inComponent component: Int) {
    // 改變第一列時
```

```
if component == 0 {  
    // whatDay 設置為陣列 week 的第 row 項資料  
    whatDay = week[row]  
} else {  
    // 否則就是改變第二列  
    // whatMeal 設置為陣列 meals 的第 row 項資料  
    whatMeal = meals[row]  
}  
  
// 將改變的結果印出來  
print("選擇的是 \(whatDay) , \(whatMeal)")  
}
```

上述程式的方法中，可以看到出現的參數 `component` 及 `row`，分別代表著哪一列以及這列的哪一行資料，例如第二列的第三行資料，就是 `component` 為 1 以及 `row` 為 2。

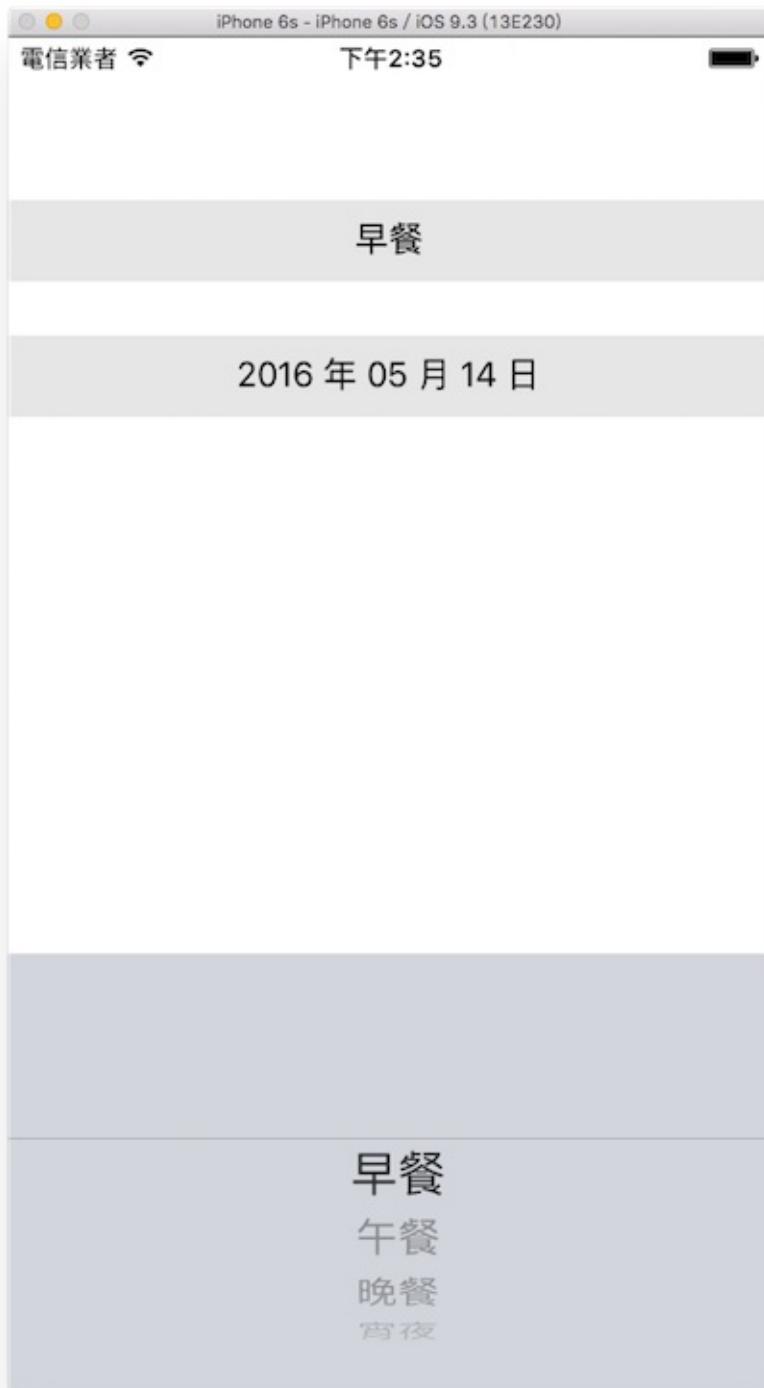
請注意，這兩個參數 `component` 及 `row` 都是從 0 開始算起，剛好與陣列從 0 開始算起相同，所以可以很方便的使用 `week[row]` 這種方式來找到需要的資料。

以上建立 `UIPickerView` 並實作好它需要的委任方法，即完成這個範例的程式內容。

與 `UITextField` 的綜合應用

以實務經驗來說，`UIPickerView` 大多會與 `UITextField` 結合應用，將選擇器嵌入在輸入框原先鍵盤的視圖位置，這樣就只會在需要的時候才顯示整個選擇器的畫面。

這個範例會放入兩個輸入框 `UITextField`，並分別使用 `UIPickerView` 及 `UIDatePicker` 來當做輸入框的輸入工具，以下是目標：



首先在 Xcode 裡，新建一個 **Single View Application** 類型的專案，取名為 `ExPickerInUITextField`。

先為 `ViewController` 建立兩個屬性：

```
class ViewController: UIViewController {  
    let meals = ["早餐", "午餐", "晚餐", "宵夜"]  
    var formatter: NSDateFormatter! = nil  
  
    // 省略  
}
```

以及在 `viewDidLoad()` 中取得螢幕尺寸，以供後續使用，如下：

```
// 取得螢幕的尺寸  
let fullScreenSize = UIScreen.mainScreen().bounds.size
```

UIPickerView 與 UITextField

一開始先在 `ViewController` 內建立一個與 `UIPickerView` 結合應用的 `UITextField`，如下：

```

// 建立一個 UITextField
var myTextField = UITextField(frame: CGRect(
    x: 0, y: 0,
    width: fullScreenSize.width, height: 40))

// 建立 UIPickerView
let myPickerView = UIPickerView()

// 設定 UIPickerView 的 delegate 及 dataSource
myPickerView.delegate = self
myPickerView.dataSource = self

// 將 UITextField 原先鍵盤的視圖更換成 UIPickerView
myTextField.inputView = myPickerView

// 設置 UITextField 預設的內容
myTextField.text = meals[0]

// 設置 UITextField 的 tag 以利後續使用
myTextField.tag = 100

// 設置 UITextField 其他資訊並放入畫面中
myTextField.backgroundColor = UIColor.init(
    red: 0.9, green: 0.9, blue: 0.9, alpha: 1)
myTextField.textAlignment = .Center
myTextField.center = CGPoint(
    x: fullScreenSize.width * 0.5,
    y: fullScreenSize.height * 0.15)
self.view.addSubview(myTextField)

```

上述程式可以看到 UITextField 的屬性 `inputView`，預設為一個鍵盤的視圖，而將它設置為 UIPickerView，便可以將其鍵盤取代。

另外還有一個屬性 `tag`，所有 UIKIT 元件都有 `tag` 這個屬性，你可以為它設置一個整數數字，在之後要使用時，可以用父視圖的方法 `viewWithTag()` 來取得。像是上述的 UITextField 元件的 `tag` 設為 100 且加入到 `self.view`，那稍後要取回這個 UITextField 時，就是使用 `self.view?.viewWithTag(100) as? UITextField`。

接著如同前面一個範例檔案一樣，要設置委任對象，這次是將委任對象設為 `self`，所以以下為 `ViewController` 加上需要的協定：

```
class ViewController: UIViewController,  
UIPickerViewDelegate, UIPickerViewDataSource {  
    // 省略  
}
```

再為 `ViewController` 加上委任需要的方法：

```

// UIPickerViewDataSource 必須實作的方法：
// UIPickerView 有幾列可以選擇
func numberOfComponentsInPickerView(
    pickerView: UIPickerView) -> Int {
    return 1
}

// UIPickerViewDataSource 必須實作的方法：
// UIPickerView 各列有多少行資料
func pickerView(
    pickerView: UIPickerView,
    numberOfRowsInComponent component: Int) -> Int {
    // 返回陣列 meals 的成員數量
    return meals.count
}

// UIPickerView 每個選項顯示的資料
func pickerView(pickerView: UIPickerView,
    titleForRow row: Int,
    forComponent component: Int) -> String? {
    // 設置為陣列 meals 的第 row 項資料
    return meals[row]
}

// UIPickerView 改變選擇後執行的動作
func pickerView(pickerView: UIPickerView,
    didSelectRow row: Int, inComponent component: Int) {
    // 依據元件的 tag 取得 UITextField
    let myTextField =
        self.view?.viewWithTag(100) as? UITextField

    // 將 UITextField 的值更新為陣列 meals 的第 row 項資料
    myTextField?.text = meals[row]
}

```

Hint

- `as?` 的用法請參考[向下型別轉換](#)。

UIDatePicker 與 UITextField

再來在 ViewController 內建立一個與 UIDatePicker 結合應用的 UITextField，如下：

```
// 建立另一個 UITextField
myTextField = UITextField(frame: CGRect(
    x: 0, y: 0,
    width: fullScreenSize.width, height: 40))

// 初始化 formatter 並設置日期顯示的格式
formatter = NSDateFormatter()
formatter.dateFormat = "yyyy 年 MM 月 dd 日"

// 建立一個 UIDatePicker
let myDatePicker = UIDatePicker()

// 設置 UIDatePicker 格式
myDatePicker.datePickerMode = .Date

// 設置 UIDatePicker 顯示的語言環境
myDatePicker.locale =
    NSLocale(localeIdentifier: "zh_TW")

// 設置 UIDatePicker 預設日期為現在日期
myDatePicker.date = NSDate()

// 設置 UIDatePicker 改變日期時會執行動作的方法
myDatePicker.addTarget(
    self,
    action:
        #selector(ViewController.datePickerChanged),
    forControlEvents: .ValueChanged)

// 將 UITextField 原先鍵盤的視圖更換成 UIDatePicker
myTextField.inputView = myDatePicker

// 設置 UITextField 預設的內容
myTextField.text =
    formatter.stringFromDate(myDatePicker.date)
```

```
// 設置 UITextField 的 tag 以利後續使用  
myTextField.tag = 200  
  
// 設置 UITextField 其他資訊並放入畫面中  
myTextField.backgroundColor = UIColor.init(  
    red: 0.9, green: 0.9, blue: 0.9, alpha: 1)  
myTextField.textAlignment = .Center  
myTextField.center = CGPoint(  
    x: fullScreenSize.width * 0.5,  
    y: fullScreenSize.height * 0.25)  
self.view.addSubview(myTextField)
```

接著為 ViewController 加上變換日期後執行動作的方法：

```
// UIDatePicker 變更選擇時執行的動作  
func datePickerChanged(datePicker:UIDatePicker) {  
    // 依據元件的 tag 取得 UITextField  
    let myTextField =  
        self.view?.viewWithTag(200) as? UITextField  
  
    // 將 UITextField 的值更新為新的日期  
    myTextField?.text =  
        formatter.stringFromDate(datePicker.date)  
}
```

隱藏編輯狀態

最後為了選擇完後可以隱藏 UIPickerView (或 UIDatePicker)，加上點擊空白處可以隱藏編輯狀態的功能，先在 ViewController 中加上觸控事件：

```
// 增加一個觸控事件
let tap = UITapGestureRecognizer(
    target: self,
    action:
        #selector(ViewController.hideKeyboard(_:)))

tap.cancelsTouchesInView = false

// 加在最基底的 self.view 上
self.view.addGestureRecognizer(tap)
```

再為 `ViewController` 加上觸控事件的方法：

```
// 按空白處會隱藏編輯狀態
func hideKeyboard(tapG:UITapGestureRecognizer){
    self.view.endEditing(true)
}
```

以上便為這個範例的程式說明。

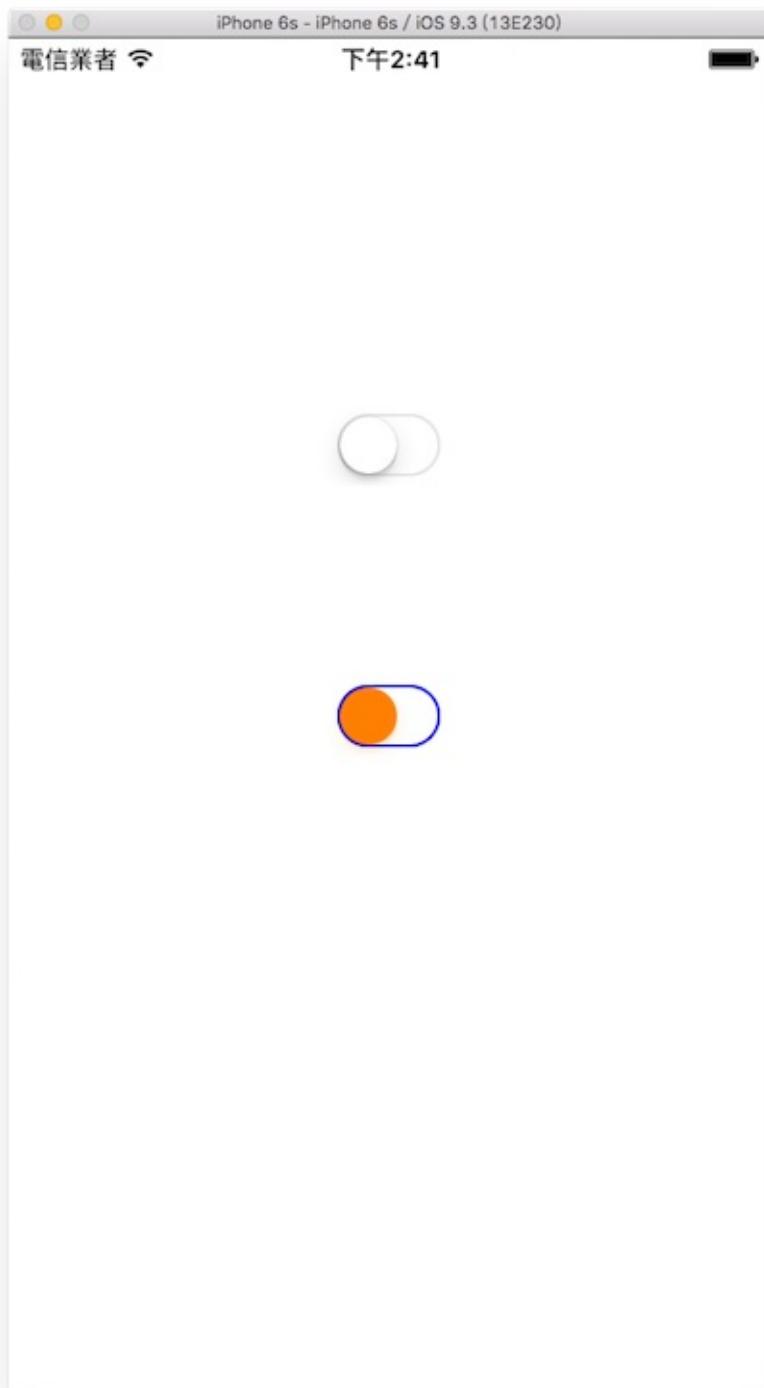
範例

本節範例程式碼放在 [uikit/uipickerview](#)

開關 UISwitch

UISwitch 適用於設定一個功能要開啓或關閉，是一個很常見的元件。

本節的目標如下，建立一個預設顏色的 UISwitch，以及一個自定義顏色的 UISwitch，並設定切換開關時變換底色：



首先在 Xcode 裡，新建一個 **Single View Application** 類型的專案，取名為 ExUISwitch。

先在 `viewDidLoad()` 中取得螢幕尺寸，以供後續使用，如下：

```
// 取得螢幕的尺寸  
let screenSize = UIScreen.mainScreen().bounds.size
```

一開始在 `viewDidLoad` 中建立一個預設狀態的 UISwitch：

```
// 建立一個 UISwitch  
var mySwitch = UISwitch()  
  
// 設置位置並放入畫面中  
mySwitch.center = CGPointMake(  
    x: screenSize.width * 0.5,  
    y: screenSize.height * 0.3)  
self.view.addSubview(mySwitch)
```

接著在 `viewDidLoad` 中建立一個自定義顏色的 UISwitch，並設定切換開關時變換底色：

```
// 建立另一個 UISwitch  
mySwitch = UISwitch()  
  
// 設置滑桿鈕的顏色  
mySwitch.thumbTintColor = UIColor.orangeColor()  
  
// 設置未選取時( off )的外觀顏色  
mySwitch.tintColor = UIColor.blueColor()  
  
// 設置選取時( on )的外觀顏色  
mySwitch.onTintColor = UIColor.brownColor()  
  
// 設置切換 UISwitch 時執行的動作  
mySwitch.addTarget(  
    self,  
    action:  
        #selector(ViewController.onChange),  
    forControlEvents: .ValueChanged)  
  
// 設置位置並放入畫面中  
mySwitch.center = CGPoint(  
    x: screenSize.width * 0.5,  
    y: screenSize.height * 0.5)  
self.view.addSubview(mySwitch)
```

最後為 ViewController 新增一個切換開關時執行動作的方法：

```
// UISwitch 切換時 執行動作的方法
func onChange(sender: AnyObject) {
    // 取得這個 UISwtich 元件
    let tempSwitch = sender as! UISwitch

    // 依據屬性 on 來為底色變色
    if tempSwitch.on {
        self.view.backgroundColor =
            UIColor.blackColor()
    } else {
        self.view.backgroundColor =
            UIColor.whiteColor()
    }
}
```

Hint

- 因為 UISwitch 元件的尺寸是固定的，所以範例便沒有為它設定新的尺寸，實際上如果設置新的尺寸也不會改變它的大小。

以上便為本節範例的內容。

範例

本節範例程式碼放在 [uikit/uiswitch](#)

分段控制 **UISegmentedControl**

UISegmentedControl 最常見到的地方就是 iPhone 螢幕上緣下拉後出現的通知中心，你可以看到有 今天 與 通知 兩個選項(iOS 9)，來切換底下不同的內容。

以下是本節的目標，分成四段選項，切換選項時會印出選到的值：



首先在 Xcode 裡，新建一個 **Single View Application** 類型的專案，取名為 ExUISegmentedControl 。

一開始先在 `viewDidLoad()` 中取得螢幕尺寸，以供後續使用，如下：

```
// 取得螢幕的尺寸  
let fullScreenSize = UIScreen.mainScreen().bounds.size
```

接著在 `viewDidLoad()` 中建立 UISegmentedControl：

```
// 使用 UISegmentedControl(items:) 建立 UISegmentedControl  
// 參數 items 是一個陣列 會依據這個陣列顯示選項  
// 除了文字 也可以擺放圖片 像是 [UIImage(named:"play")!, "晚餐"]  
let mySegmentedControl = UISegmentedControl(  
    items: ["早餐", "午餐", "晚餐", "宵夜"])  
  
// 設置外觀顏色 預設為藍色  
mySegmentedControl.tintColor = UIColor.blackColor()  
  
// 設置底色 沒有預設的顏色  
mySegmentedControl.backgroundColor =  
    UIColor.lightGrayColor()  
  
// 設置預設選擇的選項  
// 從 0 開始算起 所以這邊設置為第一個選項  
mySegmentedControl.selectedSegmentIndex = 0  
  
// 設置切換選項時執行的動作  
mySegmentedControl.addTarget(  
    self,  
    action:  
        #selector(ViewController.onChange),  
    forControlEvents: .ValueChanged)  
  
// 設置尺寸及位置並放入畫面中  
mySegmentedControl.frame.size = CGSize(  
    width: screenSize.width * 0.8, height: 30)  
mySegmentedControl.center = CGPoint(  
    x: screenSize.width * 0.5,  
    y: screenSize.height * 0.25)  
self.view.addSubview(mySegmentedControl)
```

最後為 ViewController 新增切換選項時執行動作的方法：

```
// 切換選項時執行動作的方法
func onChange(sender: UISegmentedControl) {
    // 印出選到哪個選項 從 0 開始算起
    print(sender.selectedSegmentIndex)

    // 印出這個選項的文字
    print(
        sender.titleForSegmentAtIndex(
            sender.selectedSegmentIndex))
}
```

以上即為本節範例的內容。

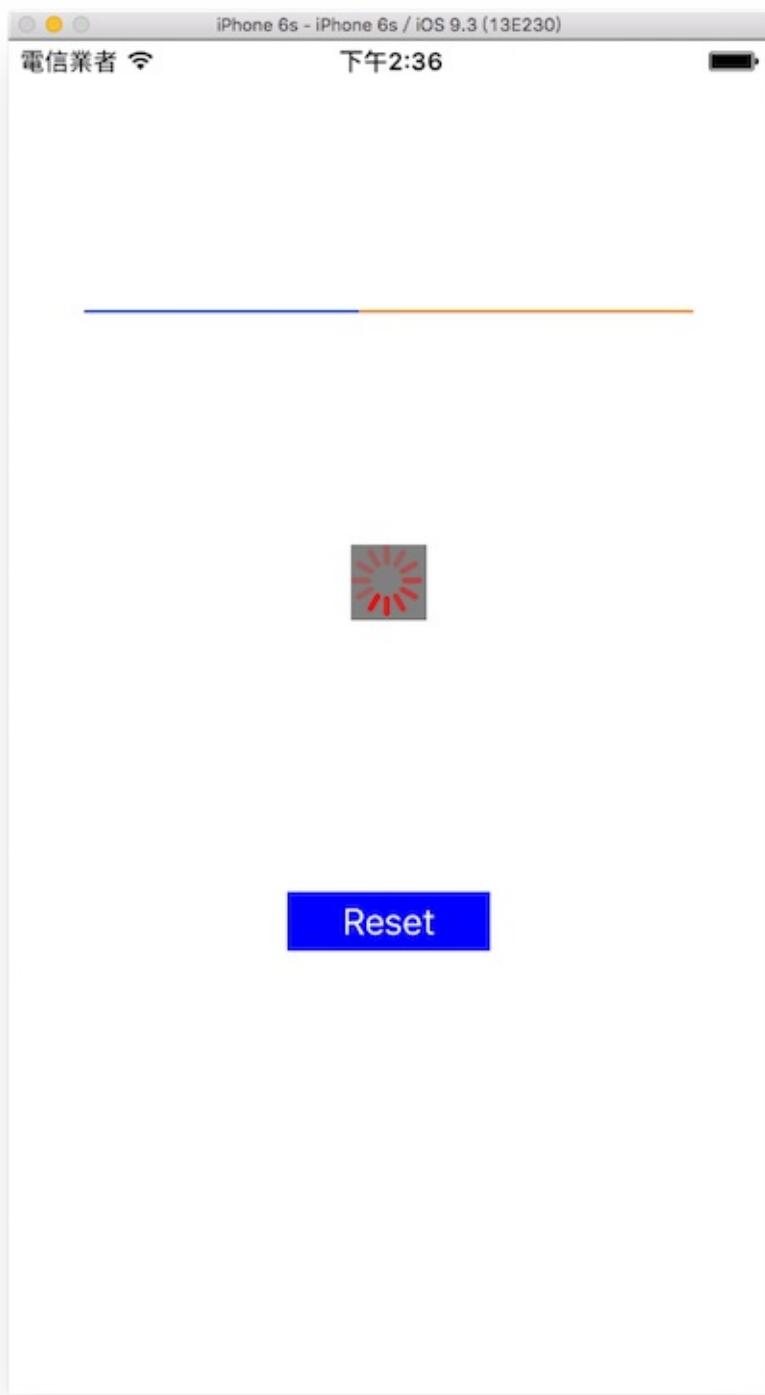
範例

本節範例程式碼放在 [uikit/uisegmentedcontrol](#)

進度條 **UIProgressView**

當應用程式要執行一個需要花點時間的工作時，通常會使用進度條來告訴使用者，目前正在執行程式中，請稍後。**UIKit** 提供了兩種可以使用的進度條，一個是長條進度條 **UIProgressView** 以及環狀進度條 **UIActivityIndicatorView** 。

這節會一起示範這兩種進度條，以下是本節目標，分別建立兩種進度條，並放置一個按鈕用來重新執行進度條：



首先在 Xcode 裡，新建一個 **Single View Application** 類型的專案，取名為 **ExUIProgressView**。

一開始先為 **ViewController** 建立六個屬性：

```
class ViewController: UIViewController {
    var myProgressView:UIProgressView!
    var myActivityIndicator:UIActivityIndicatorView!
    var myTimer:NSTimer?
    var myButton:UIButton!
    var count = 0
    let complete = 100

    // 省略
}
```

以及在 `viewDidLoad()` 中取得螢幕尺寸，以供後續使用，如下：

```
// 取得螢幕的尺寸
let fullScreenSize = UIScreen.mainScreen().bounds.size
```

建立 UIProgressView

先在 `viewDidLoad` 裡建立一個長條進度條 `UIProgressView`，如下：

```
// 建立一個 UIProgressView
myProgressView = UIProgressView(
    progressViewStyle : .Default)

// UIProgressView 的進度條顏色
myProgressView.progressTintColor=UIColor.blueColor()

// UIProgressView 進度條尚未填滿時底下的顏色
myProgressView.trackTintColor=UIColor.orangeColor()

// 設置尺寸與位置並放入畫面中
myProgressView.frame=CGRectMake(
    0, 0, fullScreenSize.width * 0.8, 50)
myProgressView.center = CGPoint(
    x: fullScreenSize.width * 0.5,
    y: fullScreenSize.height * 0.2)
self.view.addSubview(myProgressView)
```

上述程式中可以看到，使用 `UIProgressView(progressViewStyle:)` 建立，有兩種樣式可以選擇，分別為 `.Default` 及 `.Bar`。要讓進度可以推進則是使用到 `progress` 屬性，稍後會繼續介紹。

請注意 `UIProgressView` 只可以設定寬度(`width`)，而高度(`height`)是固定的。

建立 UIActivityIndicatorView

接著在 `viewDidLoad` 裡建立一個環狀進度條 `UIActivityIndicatorView`，如下：

```
// 建立一個 UIActivityIndicatorView
myActivityIndicator = UIActivityIndicatorView(
    activityIndicatorStyle:.WhiteLarge)

// 環狀進度條的顏色
myActivityIndicator.color = UIColor.redColor()

// 底色
myActivityIndicator.backgroundColor =
    UIColor.grayColor()

// 設置位置並放入畫面中
myActivityIndicator.center = CGPoint(
    x: fullScreenSize.width * 0.5,
    y: fullScreenSize.height * 0.4)
self.view.addSubview(myActivityIndicator);
```

上述程式中可以看到，使

用 `UIActivityIndicatorView(activityIndicatorStyle:)` 建立，有三種樣式可以選擇，分別為 `.Gray`、`.White` 及 `.WhiteLarge`。要啓動與停止則是使用到 `startAnimating()` 及 `stopAnimating()` 方法，稍後會繼續介紹。

請注意 `UIActivityIndicatorView` 設定的尺寸只會影響底色部分的大小，環狀進度條的尺寸部份則是固定無法變動。

模擬進度推進

在 `viewDidLoad` 裡建立一個按鈕用來重設進度以測試，並在 `viewDidLoad` 的最後先執行一次進度條的動作：

```
// 建立一個 UIButton
myButton = UIButton(frame: CGRect(
    x: 0, y: 0, width: 100, height: 30))
myButton.setTitle("Reset", forState: .Normal)
myButton.backgroundColor = UIColor.blueColor()
myButton.addTarget(
    nil,
    action:
        #selector(ViewController.clickButton),
    forControlEvents: .TouchUpInside)
myButton.center = CGPoint(
    x: fullScreenSize.width * 0.5,
    y: fullScreenSize.height * 0.65)
self.view.addSubview(myButton)

// 先執行一次進度條的動作
self.clickButton()
```

接著是按下按鈕後執行動作的方法：

```
func clickButton() {  
    // 進度推進時讓按鈕無法作用  
    myButton.enabled = false  
  
    // 分別重設兩個進度條  
    myProgressView.progress = 0  
    myActivityIndicator.startAnimating()  
  
    // 建立一個 NSTimer  
    myTimer = NSTimer.scheduledTimerWithTimeInterval(  
        0.2,  
        target: self,  
        selector:  
            #selector(ViewController.showProgress),  
        userInfo: ["test":"for userInfo test"],  
        repeats: true)  
}
```

上述程式可以看到利用 `NSTimer` 來設定一個計時器，以模擬進度推進。

`NSTimer` 是一個用來定時執行動作的類別，以下介紹 `scheduledTimerWithTimeInterval()` 方法的各參數：

- `ti`：第一個參數是間隔多久執行一次動作，單位是秒。
- `target`：執行動作的對象，通常是 `self`。
- `selector`：定時執行的方法。
- `userInfo`：可以把需要的參數帶入方法中，沒有的話就填 `nil`。
- `repeats`：是否可以重複執行，如果填 `false` 則是執行一次即不再動作。

在按下按鈕後，就會啓動一個 `NSTimer`，以下則是這個計時器定時執行的方法：

```
func showProgress(sender: NSTimer) {
    // 以一個計數器模擬背景處理的動作
    count += 5

    // 每次都為進度條增加進度
    myProgressView.progress =
        Float(count) / Float(complete)

    // 進度完成時
    if count >= complete {
        // 示範 userInfo 傳入的參數
        var info =
            sender.userInfo as?
            Dictionary<String, AnyObject>
        print(info?["test"])

        // 重設計數器及 NSTimer 供下次按下按鈕測試
        count = 0
        myTimer?.invalidate()
        myTimer = nil

        // 隱藏環狀進度條
        myActivityIndicator.stopAnimating()

        // 將按鈕功能啓動
        myButton.enabled = true
    }
}
```

上述程式可以看到，使用屬性 `count` 每 0.2 秒 +5，來模擬進度推進的效果，當 `count` 加到 100 時則是完成進度，會重設各設定，以利下次按下按鈕繼續測試。

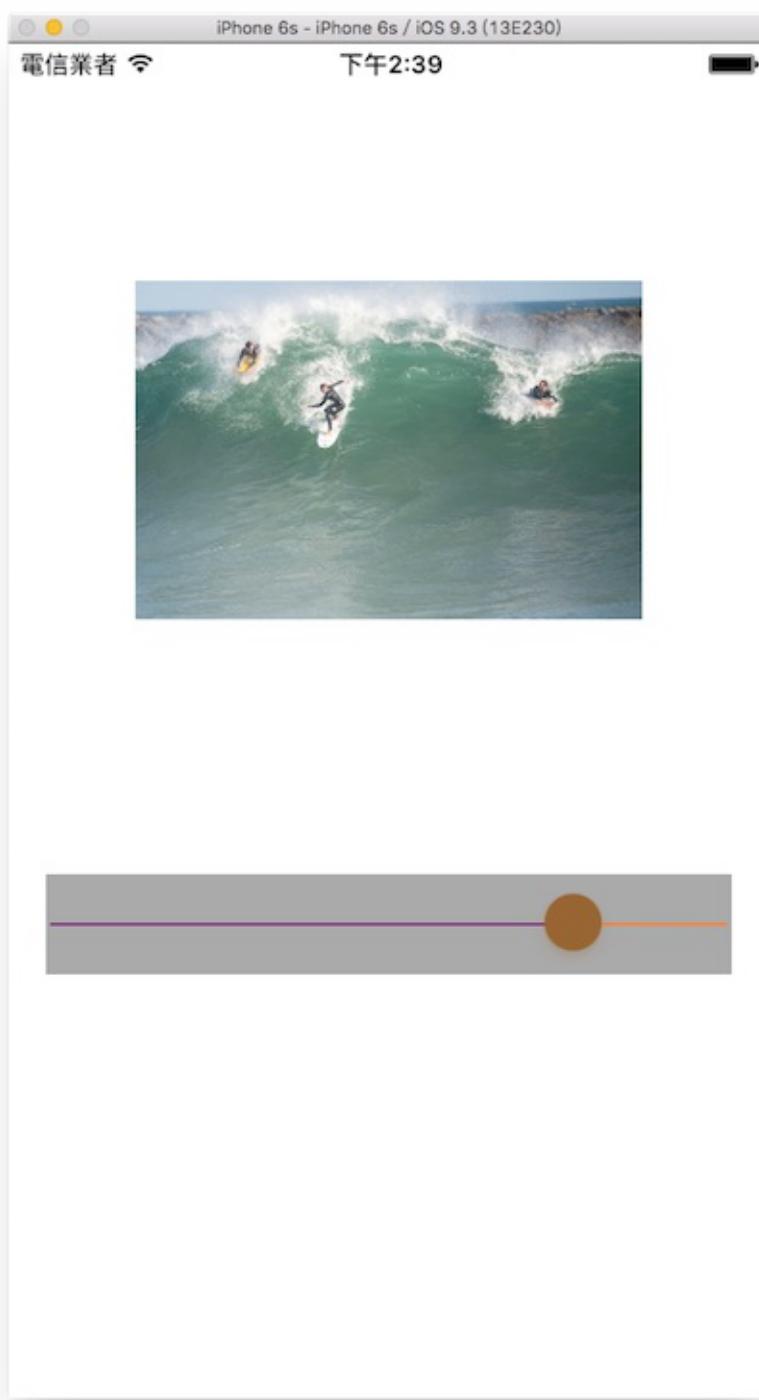
以上就是本節範例的介紹。請注意，本節範例只是示範用，實際使用進度條的情況應該是，在執行一個需要花點時間的工作時啓動，在這個工作完成時結束。

範例

本節範例程式碼放在 [uikit/uiprogressview](#)

滑桿 UISlider

UISlider 常用在控制音量或影片播放進度。以下是本節的目標，建立一張圖片，並使用 UISlider 來控制圖片的透明度：



首先在 Xcode 裡，新建一個 **Single View Application** 類型的專案，取名為 ExUISlider。

一開始先為 `ViewController` 建立兩個屬性：

```
class ViewController: UIViewController {  
    var imageView :UIImageView!  
    var mySlider :UISlider!  
  
    // 省略  
}
```

以及在 `viewDidLoad()` 中取得螢幕尺寸，以供後續使用，如下：

```
// 取得螢幕的尺寸  
let fullScreenSize = UIScreen.mainScreen().bounds.size
```

先在 `viewDidLoad()` 中建立一張圖片，以供後續示範使用：

```
// 建立一個 UIImageView  
imageView = UIImageView(  
    image: UIImage(named: "01.jpg"))  
imageView.contentMode = .ScaleAspectFit  
imageView.frame = CGRect(  
    x: 0, y: 0, width: 250, height: 250)  
imageView.center = CGPoint(  
    x: fullScreenSize.width * 0.5,  
    y: fullScreenSize.height * 0.3)  
self.view.addSubview(imageView)
```

建立 UISlider

接著在 `viewDidLoad()` 中建立一個 `UISlider`：

```
// 建立一個 UISlider  
mySlider=UISlider(frame: CGRect(  
    x: 0, y: 0, width:
```

```
fullScreenSize.width * 0.9, height: 50))

// UISlider 底色
mySlider.backgroundColor = UIColor.lightGrayColor()

// UISlider 滑桿按鈕右邊 尚未填滿的顏色
mySlider.maximumTrackTintColor = UIColor.orangeColor()

// UISlider 滑桿按鈕左邊 已填滿的顏色
mySlider.minimumTrackTintColor = UIColor.purpleColor()

// UISlider 滑桿按鈕的顏色
mySlider.thumbTintColor = UIColor.brownColor()

// UISlider 的最小值
mySlider.minimumValue = 0

// UISlider 的最大值
mySlider.maximumValue = 100

// UISlider 預設值
mySlider.value = 100

// UISlider 是否可以在變動時同步執行動作
// 設定 false 時 則是滑動完後才會執行動作
mySlider.continuous = true

// UISlider 滑動滑桿時執行的動作
mySlider.addTarget(
    self,
    action:
        #selector(ViewController.onSliderChange),
    forControlEvents: UIControlEvents.ValueChanged)

// 設置位置並放入畫面中
mySlider.center = CGPointMake(
    x: fullScreenSize.width * 0.5,
    y: fullScreenSize.height * 0.65)
self.view.addSubview(mySlider)
```

請注意 `UISlider` 可以設定寬度(`width`)，而高度(`height`)則是固定無法變動。

接著在 `ViewController` 中加入滑動滑桿時執行動作的方法：

```
func onSliderChange() {  
    // 設置圖片的透明度  
    imageView.alpha = CGFloat(  
        mySlider.value / mySlider.maximumValue)  
}
```

上述程式可以看到利用 `UISlider` 目前的值與最大值相比，來設定圖片的透明度。一個元件的 `alpha` 屬性是用來表示此元件的透明度，範圍是 0 到 1，0 為完全透明，一直到 1 為完全顯示。

以上即為本節範例的內容。

圖片來源

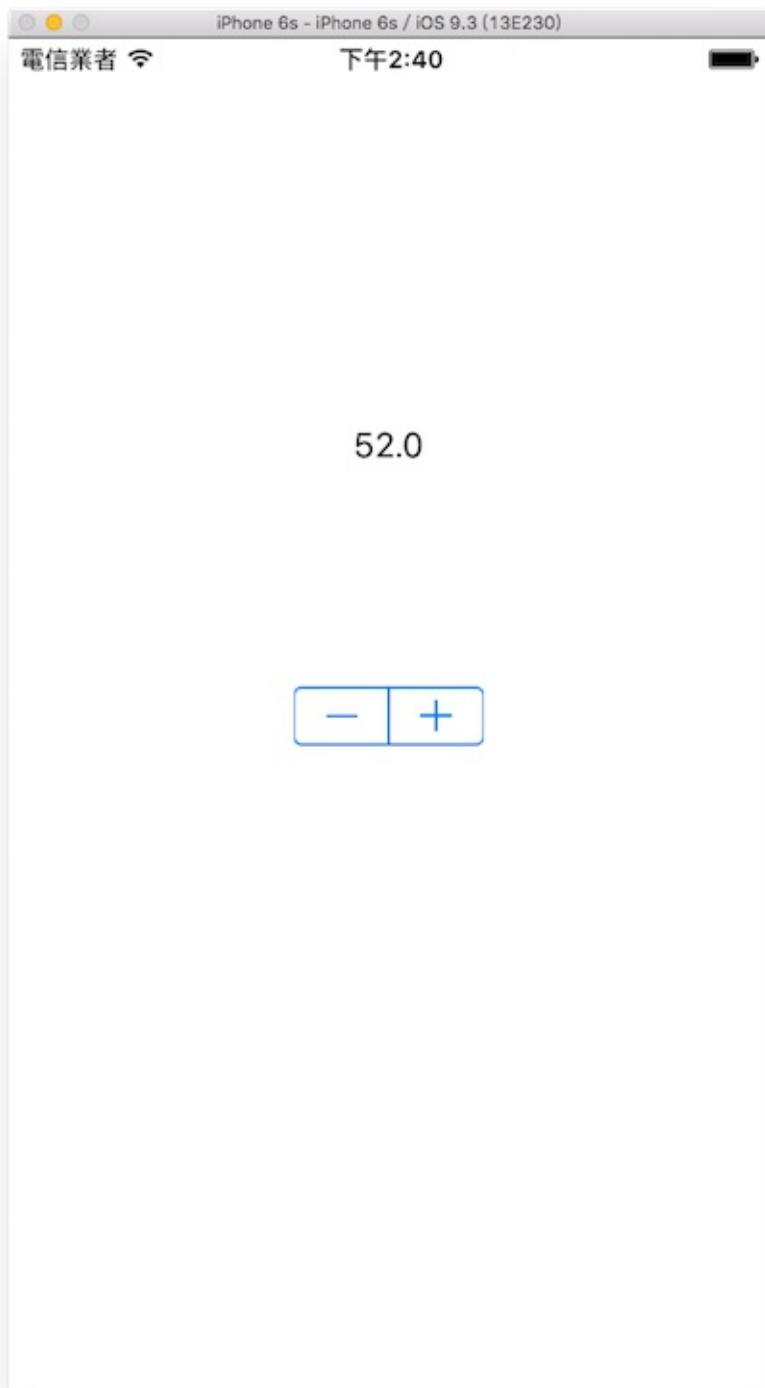
- <https://www.flickr.com/photos/66603656@N05/17349463116/>

範例

本節範例程式碼放在 [uikit/uislider](#)

步進器 UIStepper

UIStepper 可以用來為一個數值作增減的動作，這個元件已經設定好增加及減少的按鈕，可以快速的建立起來。以下是本節的目標，使用 UIStepper 來增減數值並顯示在一個 UILabel 上：



首先在 Xcode 裡，新建一個 **Single View Application** 類型的專案，取名為 ExUIStepper。

一開始先為 `ViewController` 建立兩個屬性：

```
class ViewController: UIViewController {  
    var myLabel :UILabel!  
    var myStepper :UIStepper!  
  
    // 省略  
}
```

以及在 `viewDidLoad()` 中取得螢幕尺寸，以供後續使用，如下：

```
// 取得螢幕的尺寸  
let fullScreenSize = UIScreen.mainScreen().bounds.size
```

先在 `viewDidLoad()` 中建立一個 `UILabel`，以供後續示範使用：

```
// 建立一個 UILabel  
myLabel = UILabel(frame: CGRectMake(  
    x: 0, y: 0, width: 100, height: 40))  
myLabel.text = "0"  
myLabel.textColor = UIColor.blackColor()  
myLabel.textAlignment = .Center  
myLabel.center = CGPointMake(  
    x: fullScreenSize.width * 0.5,  
    y: fullScreenSize.height * 0.3)  
self.view.addSubview(myLabel)
```

建立 **UIStepper**

接著在 `viewDidLoad()` 中建立一個 `UIStepper`：

```
// 建立一個 UIStepper  
myStepper = UIStepper()  
  
// UIStepper 預設值  
myStepper.value = 0  
  
// UIStepper 最小值  
myStepper.minimumValue = 0  
  
// UIStepper 最大值  
myStepper.maximumValue = 100  
  
// UIStepper 每按一次按鈕 增減的數值  
myStepper.stepValue = 2  
  
// UIStepper 是否可接著增減按鈕持續變化數值  
myStepper.autorepeat = true  
  
// UIStepper 是否可以在變動時同步執行動作  
// 設定 false 時 則是放開按鈕後才會執行動作  
myStepper.continuous = true  
  
// UIStepper 數值是否可以循環  
// 例如填 true 時 如果值已達到最大值  
// 再按一次 + 會循環到最小值繼續加  
myStepper.wraps = true  
  
// UIStepper 按下增減按鈕後 執行的動作  
myStepper.addTarget(  
    self,  
    action:  
        #selector(ViewController.onStepperChange),  
    forControlEvents: .ValueChanged)  
  
// 設置 UIStepper 位置並放入畫面中  
myStepper.center = CGPoint(  
    x: screenSize.width * 0.5,  
    y: screenSize.height * 0.5)  
self.view.addSubview(myStepper)
```

請注意 `UIStepper` 的尺寸長寬是固定的，無法變動。

接著為 `ViewController` 加入 `UIStepper` 按下按鈕後執行動作的方法：

```
func onStepperChange() {
    // 將 UILabel 的值設置為 UIStepper 目前的值
    myLabel.text = "\(myStepper.value)"
}
```

以上即為本節範例的內容。

範例

本節範例程式碼放在 [uikit/uistepper](#)

網頁 **UIWebView**

應用程式中有時如果需要載入一些外部網站時，可以使用 **UIWebView**，可以很快速的建立起來。以下是本節的目標，一個超輕量瀏覽器，上方有五個按鈕，功能分別為上一頁、下一頁、重新讀取、取消讀取及前往網址，以及一個網址列與 **UIWebView**：



首先在 Xcode 裡，新建一個 **Single View Application** 類型的專案，取名為 `ExUIWebView`。再以 [加入檔案](#) 的方式加入四張按鈕的圖片。

先為 `ViewController` 建立三個屬性：

```

class ViewController: UIViewController {
    var myTextField :UITextField!
    var myWebView :UIWebView!
    var myActivityIndicator:UIActivityIndicatorView!

    // 省略
}

```

以及在 `viewDidLoad()` 中取得螢幕尺寸跟預設尺寸，以供後續使用，如下：

```

// 取得螢幕的尺寸
let fullScreenSize = UIScreen.mainScreen().bounds.size

// 預設尺寸
let goWidth = 100.0
let actionWidth =
    ( Double(fullScreenSize.width) - goWidth ) / 4

```

前置作業

一開始先在 `viewDidLoad()` 裡建立五個 `UIButton` 及一個用來輸入網址的 `UITextField`：

```

// 建立五個 UIButton
var myButton = UIButton(frame: CGRect(
    x: 0, y: 20,
    width: actionWidth, height: actionWidth))
myButton.setImage(
    UIImage(named: "back")!, forState: .Normal)
myButton.addTarget(
    self,
    action:
        #selector(ViewController.back),
    forControlEvents: .TouchUpInside)
self.view.addSubview(myButton)

myButton = UIButton(frame: CGRect(

```

```
x: actionWidth, y: 20,  
    width: actionWidth, height: actionWidth))  
myButton.setImage(  
    UIImage(named: "forward")!, forState: .Normal)  
myButton.addTarget(  
    self,  
    action:  
        #selector(ViewController.forward),  
    forControlEventss: .TouchUpInside)  
self.view.addSubview(myButton)  
  
myButton = UIButton(frame: CGRect(  
    x: actionWidth * 2, y: 20,  
    width: actionWidth, height: actionWidth))  
myButton.setImage(  
    UIImage(named: "refresh")!, forState: .Normal)  
myButton.addTarget(  
    self,  
    action:  
        #selector(ViewController.reload),  
    forControlEventss: .TouchUpInside)  
self.view.addSubview(myButton)  
  
myButton = UIButton(frame: CGRect(  
    x: actionWidth * 3, y: 20,  
    width: actionWidth, height: actionWidth))  
myButton.setImage(  
    UIImage(named: "stop")!, forState: .Normal)  
myButton.addTarget(  
    self,  
    action:  
        #selector(ViewController.stop),  
    forControlEventss: .TouchUpInside)  
self.view.addSubview(myButton)  
  
myButton = UIButton(frame: CGRect(  
    x: Double(fullScreenSize.width) - goWidth, y: 20,  
    width: goWidth, height: actionWidth))  
myButton.setTitle("前往", forState: .Normal)  
myButton.setTitleColor(
```

```
UIColor.blackColor(), forState: .Normal)
myButton.addTarget(
    self,
    action:
        #selector(ViewController.go),
    forControlEvents: .TouchUpInside)
self.view.addSubview(myButton)

// 建立一個 UITextField 用來輸入網址
myTextField = UITextField(frame: CGRect(
    x: 0, y: 20.0 + CGFloat(actionWidth),
    width: fullScreenSize.width, height: 40))
myTextField.text = "https://www.google.com"
myTextField.backgroundColor = UIColor.init(
    red: 0.95, green: 0.95, blue: 0.95, alpha: 1)
myTextField.clearButtonMode = .WhileEditing
myTextField.returnKeyType = .Go
myTextField.delegate = self
self.view.addSubview(myTextField)
```

接著設置 UITextField 的委任需要的協定：

```
class ViewController:
    UIViewController, UITextFieldDelegate {
    // 省略
}
```

再於 ViewController 中，將按下按鈕及鍵盤執行動作的方法加入：

```
func back() {
    // 上一頁
    myWebView.goBack()
}

func forward() {
    // 下一頁
    myWebView.goForward()
}

func reload() {
    // 重新讀取
    myWebView.reload()
}

func stop() {
    // 取消讀取
    myWebView.stopLoading()

    // 隱藏環狀進度條
    myActivityIndicator.stopAnimating()
}

func go() {
    // 隱藏鍵盤
    self.view.endEditing(true)

    // 前往網址
    let url = NSURL(string:myTextField.text!)
    let urlRequest = URLRequest(URL: url!)
    myWebView.loadRequest(urlRequest)
}

func textFieldShouldReturn(textField: UITextField) -> Bool {
    self.go()

    return true
}
```

建立 UIWebView

在 `viewDidLoad()` 裡建立 `UIWebView` 以及讀取網頁時顯示的環狀進度條。
在 `viewDidLoad()` 的最後先讀取一次網址：

```
// 建立 UIWebView
myWebView = UIWebView(frame: CGRect(
    x: 0, y: 60.0 + CGFloat(actionWidth),
    width: fullScreenSize.width,
    height:
        fullScreenSize.height - 60
    - CGFloat(actionWidth)))

// 設置委任對象
myWebView.delegate = self

// 加入到畫面中
self.view.addSubview(myWebView)

// 建立環狀進度條
myActivityIndicator = UIActivityIndicatorView(
    activityIndicatorStyle:.Gray)
myActivityIndicator.center = CGPoint(
    x: fullScreenSize.width * 0.5,
    y: fullScreenSize.height * 0.5)
self.view.addSubview(myActivityIndicator);

// 先讀取一次網址
self.go()
```

為了讓進度條可以顯示及隱藏，必須設定 `UIWebView` 的委任並實作兩個方法，首先加上委任需要的協定：

```
class ViewController: UIViewController,
    UITextFieldDelegate, UIWebViewDelegate {
    // 省略
}
```

兩個委任需要實作的方法，分別是開始讀取網址時執行動作，以及讀取網址完成時執行動作：

```
func webViewDidStartLoad(webView: UIWebView) {
    // 顯示進度條
    myActivityIndicator.startAnimating()
}

func webViewDidFinishLoad(webView: UIWebView) {
    // 隱藏進度條
    myActivityIndicator.stopAnimating()

    // 更新網址列的內容
    if let currentURL = myWebView.request?.URL!.absoluteString {
        myTextField.text = currentURL
    }
}
```

以上便為本節範例的內容。

讀取靜態 HTML 原始碼

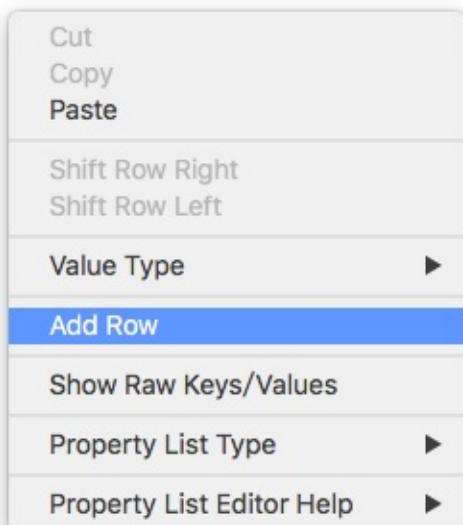
UIWebView 還有另一個功能，可以讓你讀取 HTML 原始碼並顯示出來，使用 `loadHTMLString()` 方法，如下：

```
// 你也可以設置 HTML 內容到一個常數
// 用來載入一個靜態的網頁內容
let content =
    "<html><body><h1>Hello World !</h1></body></html>"
myWebView.loadHTMLString(content, baseURL: nil)
```

無法載入 http 的網址

在 iOS 9 之後，應用程式內的 UIWebView 預設為只能載入 https 的網頁(也就是加密過的)，如果是普通 http 的網頁會無法開啟，這邊介紹如何設定成可開啟 http 網頁。

首先在 Xcode 左側專案的檔案列表中，找到 info.plist 並點開，在其內空白處點右鍵，接著按下 Add Row ，如下圖：



接著他會多出一列，並要你填寫，如下圖：

Key	Type	Value
▼ Information Property List	Dictionary	(16 items)
Application Category	String	
Application Category	Dictionary	(1 item)
Application does not run in bac...	Boolean	YES
Application fonts resource path	String	en

先填入 NSAppTransportSecurity ，再對剛新增的這列按右鍵點選 Add Row ，填入 NSAllowsArbitraryLoads 並設為 YES ，最後會變成如下圖所示(填入的值會與最後顯示的字不同，請和下圖裡文字比對是否設定正確)：

Key	Type	Value
▼ Information Property List	Dictionary	(15 items)
▼ App Transport Security Settings	Dictionary	(1 item)
Allow Arbitrary Loads	Boolean	YES

這樣便設定完成，你的 UIWebView 就可以載入 http 網頁了。不過這只是暫緩之計，以長遠來看，還是將所有讀取的網頁都設定為 https 還是比較安全。

圖片來源

- https://www.iconfinder.com/icons/211686/arrow_back_icon
- https://www.iconfinder.com/icons/211688/arrow_forward_icon

- https://www.iconfinder.com/icons/293657/x_icon
- https://www.iconfinder.com/icons/293697/refresh_icon

範例

本節範例程式碼放在 [uikit/uiwebview](#)

表格 UITableView

UITableView 是一個很常見的元件，當你需要將一批資料逐列顯示時常會使用到，每一個儲存格稱作一個 cell，每個 cell 除了可以顯示文字外，還可以放置多個不同的元件。本節的目標如下：



首先在 Xcode 裡，新建一個 **Single View Application** 類型的專案，取名為 ExUITableViewController。

一開始先為 `ViewController` 建立一個測試用的陣列屬性：

```
class ViewController: UIViewController {  
    var info = [  
        ["林書豪", "陳信安"],  
        ["陳偉殷", "王建民", "陳金鋒", "林智勝"]  
    ]  
  
    // 省略  
}
```

以及在 `viewDidLoad()` 中取得螢幕尺寸，以供後續使用，如下：

```
// 取得螢幕的尺寸  
let fullScreenSize = UIScreen.mainScreen().bounds.size
```

建立 UITableView

先在 `viewDidLoad()` 中建立 `UITableView`：

```
// 建立 UITableView 並設置原點及尺寸
let myTableView = UITableView(frame: CGRect(
    x: 0, y: 20,
    width: fullScreenSize.width,
    height: fullScreenSize.height - 20),
    style: .Grouped)

// 註冊 cell
myTableView.registerClass(
    UITableViewCell.self, forCellReuseIdentifier: "Cell")

// 設置委任對象
myTableView.delegate = self
myTableView.dataSource = self

// 分隔線的樣式
myTableView.separatorStyle = .SingleLine

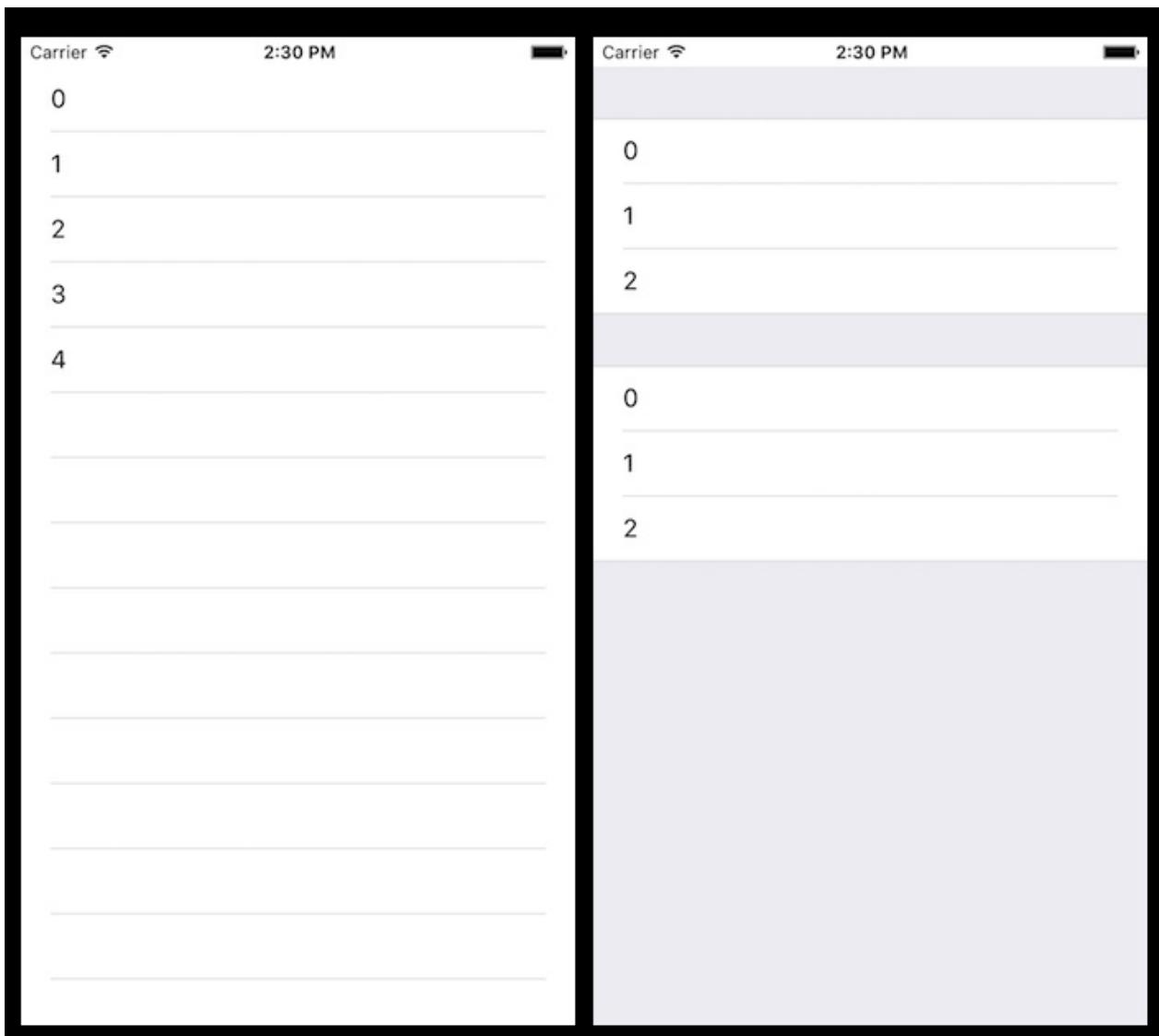
// 分隔線的間距 四個數值分別代表 上、左、下、右 的間距
myTableView.separatorInset =
    UIEdgeInsetsMake(0, 20, 0, 20)

// 是否可以點選 cell
myTableView.allowsSelection = true

// 是否可以多選 cell
myTableView.allowsMultipleSelection = false

// 加入到畫面中
self.view.addSubview(myTableView)
```

UITableView 可以設置為兩種樣式，如下圖，左邊為 .Plain 及右邊的 .Grouped：



可依照需求設置不同的樣式，本節範例會以 `.Grouped` 作為示範。接著看
到 `registerClass()` 這個註冊 cell 的方法。要先了解到當 cell 數量超過一個畫面
可顯示時，目前存在的 cell 只有畫面上的這些(否則資料有成千上萬時，誰受得了)，當上下滑動時，會隨顯示畫面的不同同時移出並加入 cell，這個動作不是一直
建立新的 cell 而是會重複使用(reuse)，所以必須先註冊這個 reuse 的 cell，辨識
名稱設為 "Cell"，來讓後續顯示時可以使用。

委任模式

UITableView 必須設置委任模式的對象來完善這個表格的內容，先
為 ViewController 加上委任需要的協定：

```
class ViewController: UIViewController,  
    UITableViewDelegate, UITableViewDataSource {  
    // 省略  
}
```

以及兩個必須實作的方法：

表格 UITableView

```
// 必須實作的方法：每一組有幾個 cell
func tableView(tableView: UITableView,
    numberOfRowsInSection section: Int) -> Int {
    return info[section].count
}

// 必須實作的方法：每個 cell 要顯示的內容
func tableView(tableView: UITableView,
    cellForRowAtIndexPath indexPath: NSIndexPath)
-> UITableViewCell {
    // 取得 tableView 目前使用的 cell
    let cell =
        tableView.dequeueReusableCellWithIdentifier(
            "Cell", forIndexPath: indexPath) as
        UITableViewCell

    // 設置 Accessory 按鈕樣式
    if indexPath.section == 1 {
        if indexPath.row == 0 {
            cell.accessoryType = .Checkmark
        } else if indexPath.row == 1 {
            cell.accessoryType = .DetailButton
        } else if indexPath.row == 2 {
            cell.accessoryType =
                .DetailDisclosureButton
        } else if indexPath.row == 3 {
            cell.accessoryType = .DisclosureIndicator
        }
    }

    // 顯示的內容
    if let myLabel = cell.textLabel {
        myLabel.text =
            "\u{1d3c}(info[indexPath.section][indexPath.row])"
    }

    return cell
}
```

UITableView 委任的方法大多會有 `indexPath` 參數，這個參數有兩個屬性分別為 `section` 及 `row`，是用来表示目前要設置的 cell 是屬於那一組(`section`)的那一列(`row`)，型別都為 `Int` 且都是由 0 開始算起。

主要注意到第二個方法每個 `cell` 要顯示的內容。一開始可以看到需要使用稍前註冊的 `cell` 使用的辨識名稱 "Cell"，這邊就是告訴程式要使用哪一個 `cell` 來重複使用。

接著看到設置 Accessory 按鈕樣式，顯示在每個 `cell` 右邊的按鈕，可依照需求設置不同的樣式，如果不設置就是留空。

最後可以看到 `cell.textLabel`，型別為 `UILabel?`，分別依照不同組的不同列來設置顯示的文字。

委任對象除了必須實作的方法之外，還有許多可以額外設置的方法，如下：

```
// 點選 cell 後執行的動作
func tableView(tableView: UITableView,
    didSelectRowAtIndexPath indexPath: NSIndexPath) {
    // 取消 cell 的選取狀態
    tableView.deselectRowAtIndexPath(
        indexPath, animated: true)

    let name = info[indexPath.section][indexPath.row]
    print("選擇的是 \(name)")
}

// 點選 Accessory 按鈕後執行的動作
// 必須設置 cell 的 accessoryType
// 設置為 .DisclosureIndicator (向右箭頭)之外都會觸發
func tableView(tableView: UITableView,
    accessoryButtonTappedForRowWithIndexPath
    indexPath: NSIndexPath) {
    let name = info[indexPath.section][indexPath.row]
    print("按下的是 \(name) 的 detail")
}

// 有幾組 section
func numberOfSectionsInTableView(
    tableView: UITableView) -> Int {
    return info.count
}

// 每個 section 的標題
func tableView(tableView: UITableView,
    titleForHeaderInSection section: Int) -> String? {
    let title = section == 0 ? "籃球" : "棒球"
    return title
}
```

以上便為本節範例的內容。

更多委任對象的方法

UITableView 提供給委任對象可以實作的方法很多，除了本節範例中示範的之外，以下再列出常使用到的方法，可依照需求彈性使用：

```
// 設置每個 section 的 title 為一個 UIView  
// 如果實作了這個方法 會蓋過單純設置文字的 section title  
func tableView(tableView: UITableView,  
    viewForHeaderInSection section: Int) -> UIView? {  
    return UIView()  
}  
  
// 設置 section header 的高度  
func tableView(tableView: UITableView,  
    heightForHeaderInSection section: Int) -> CGFloat {  
    return 80  
}  
  
// 每個 section 的 footer  
func tableView(tableView: UITableView,  
    titleForFooterInSection section: Int) -> String? {  
    return "footer"  
}  
  
// 設置每個 section 的 footer 為一個 UIView  
// 如果實作了這個方法 會蓋過單純設置文字的 section footer  
func tableView(tableView: UITableView,  
    viewForFooterInSection section: Int) -> UIView? {  
    return UIView()  
}  
  
// 設置 section footer 的高度  
func tableView(tableView: UITableView,  
    heightForFooterInSection section: Int) -> CGFloat {  
    return 80  
}  
  
// 設置 cell 的高度  
func tableView(tableView: UITableView,  
    heightForRowAtIndexPath indexPath: NSIndexPath)  
-> CGFloat {  
    return 80  
}
```

更多內容請參考 [UITableView 的編輯模式](#)。

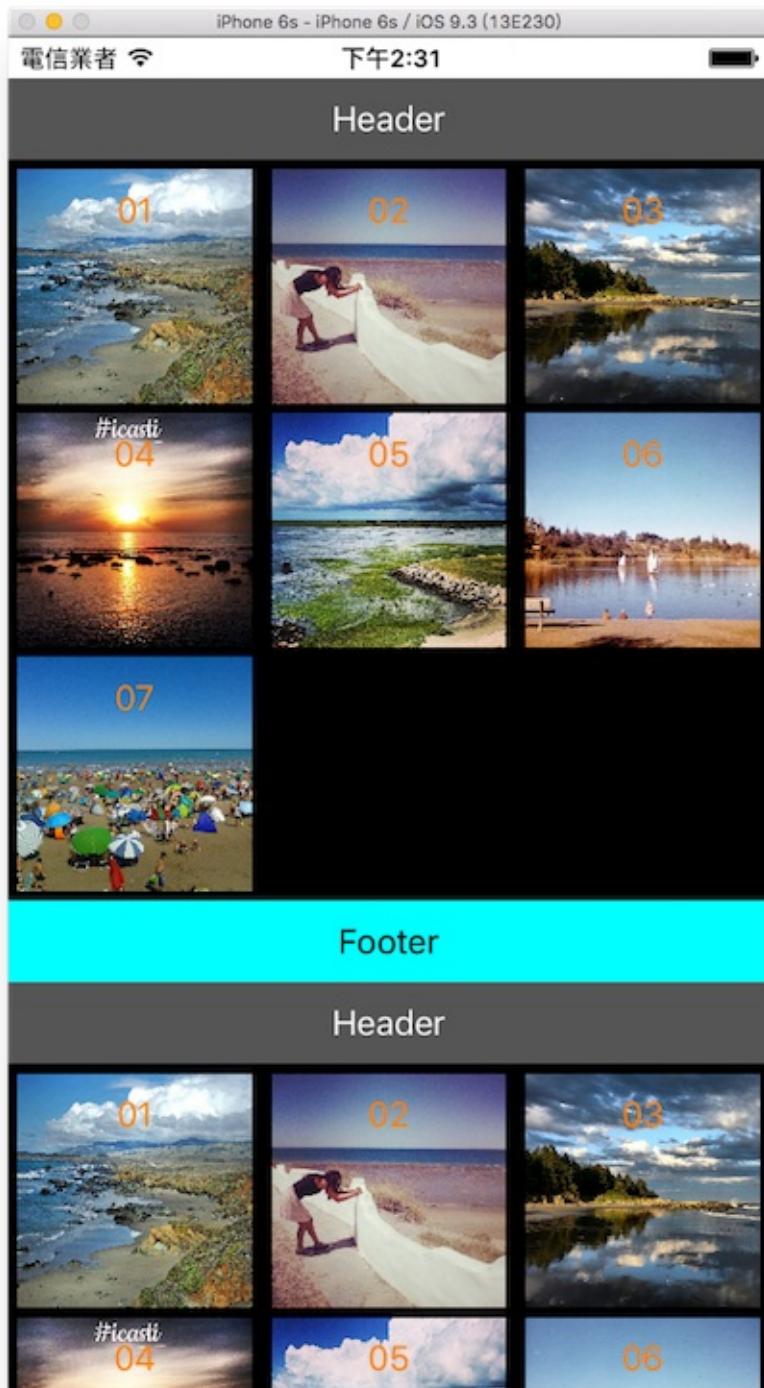
範例

本節範例程式碼放在 [uikit/uitableview](#)

網格 **UICollectionView**

UICollectionView 可以用來表現網格這樣多行多列元件的樣式，iPhone 內建的 照片 **App** 就是用 **UICollectionView** 為主要呈現方式。

本節的目標如下，分成兩組照片依序往下排列，每組有七張照片，一行有三張照片，點擊照片可以有自定義執行的動作：



首先在 Xcode 裡，新建一個 **Single View Application** 類型的專案，取名為 **ExUICollectionView**。

一開始先以加入檔案的方式加入七張示範用的圖片。

首先為 **ViewController** 建立一個屬性，以及在 **viewDidLoad()** 中取得螢幕尺寸跟設置底色，以供後續使用，如下：

```
class ViewController: UIViewController {
    var fullScreenSize :CGSize!

    override func viewDidLoad() {
        super.viewDidLoad()

        // 取得螢幕的尺寸
        fullScreenSize =
            UIScreen.mainScreen().bounds.size

        // 設置底色
        self.view.backgroundColor =
            UIColor.whiteColor()

        // 省略
    }

    // 省略
}
```

UICollectionView 有些地方與 UITableView 類似，它的每個儲存格也稱為 cell，委任對象實作的方法也差不多。而 UICollectionView 有更多需要設置的地方，所以本節範例步驟較多，這邊先簡單講解一下：

1. 一開始需要先設置 UICollectionViewFlowLayout，用來自定義呈現的樣式，再交給建立 UICollectionView 元件時的函式使用。
2. 接著建立 UICollectionView 元件，除了要註冊 cell 之外，如果要自定義每個 section 的 header 或 footer 時，也必須註冊 header 或 footer，以供後續重複使用。（註冊 cell 原因請[參考前節說明](#)）
3. UICollectionView 的 cell 需要自定義一個繼承自 UICollectionViewCell 的類別，用來加上需要的元件，這邊會加上一張圖片（UIImageView）及一行字（UILabel）。
4. 最後要實作委任對象需要的方法。

建立 UICollectionViewFlowLayout

UICollectionView 大部分的建立步驟與 UITableView 相同，
UICollectionViewFlowLayout 則是需要額外建立的部份，用來自定義呈現的樣式，
首先在 viewDidLoad() 裡建立 UICollectionViewFlowLayout：

```
// 建立 UICollectionViewFlowLayout
let layout = UICollectionViewFlowLayout()

// 設置 section 的間距 四個數值分別代表 上、左、下、右 的間距
layout.sectionInset = UIEdgeInsetsMake(5, 5, 5, 5);

// 設置每一行的間距
layout.minimumLineSpacing = 5

// 設置每個 cell 的尺寸
layout.itemSize = CGSizeMake(
    CGFloat(fullScreenSize.width)/3 - 10.0,
    CGFloat(fullScreenSize.width)/3 - 10.0)

// 設置 header 及 footer 的尺寸
layout.headerReferenceSize = CGSizeMake(
    width: fullScreenSize.width, height: 40)
layout.footerReferenceSize = CGSizeMake(
    width: fullScreenSize.width, height: 40)
```

建立 UICollectionView

緊接著在 viewDidLoad() 裡建立 UICollectionView，這邊會使用到前面建立的
UICollectionViewFlowLayout (常數 layout)：

```

// 建立 UICollectionView
let myCollectionView = UICollectionView(frame: CGRect(
    x: 0, y: 20,
    width: fullScreenSize.width,
    height: fullScreenSize.height - 20),
    collectionViewLayout: layout)

// 註冊 cell 以供後續重複使用
myCollectionView.registerClass(
    MyCollectionViewCell.self,
    forCellWithReuseIdentifier: "Cell")

// 註冊 section 的 header 跟 footer 以供後續重複使用
myCollectionView.registerClass(
    UICollectionViewReusableView.self,
    forSupplementaryViewOfKind:
        UICollectionViewElementKindSectionHeader,
    withReuseIdentifier: "Header")
myCollectionView.registerClass(
    UICollectionViewReusableView.self,
    forSupplementaryViewOfKind:
        UICollectionViewElementKindSectionFooter,
    withReuseIdentifier: "Footer")

// 設置委任對象
myCollectionView.delegate = self
myCollectionView.dataSource = self

// 加入畫面中
self.view.addSubview(myCollectionView)

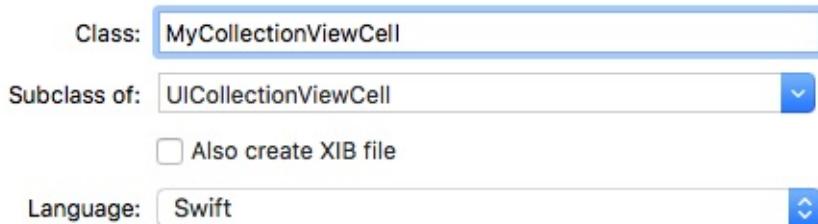
```

section 的 header 與 footer 都是屬於 UICollectionViewReusableView 類別，所以註冊時就是使用這個類別註冊。

自定義的 UICollectionViewCell 類別

前面註冊 cell 時用到的就是自定義的 UICollectionViewCell 類別 **MyCollectionViewCell**，這邊會介紹如何建立。

先以[新增檔案](#)的方式增加一個新的檔案，其中要注意的是，因為是要建立繼承自 `UICollectionViewCell` 的類別，所以進行到下圖這個步驟時， `Subclass of:` 要填寫 `UICollectionViewCell`，並將檔案名稱設為 `MyCollectionViewCell`：



建立完成後，接著進到 `MyCollectionViewCell.swift` 這隻檔案中，加入需要顯示的元件，一張圖片(`UIImageView`)及一行字(`UILabel`)：

```
class MyCollectionViewCell: UICollectionViewCell {
    var imageView: UIImageView!
    var titleLabel: UILabel!

    override init(frame: CGRect) {
        super.init(frame: frame)

        // 取得螢幕寬度
        let w = Double(
            UIScreen.mainScreen().bounds.size.width)

        // 建立一個 UIImageView
        imageView = UIImageView(frame: CGRect(
            x: 0, y: 0,
            width: w/3 - 10.0, height: w/3 - 10.0))
        self.addSubview(imageView)

        // 建立一個 UILabel
        titleLabel = UILabel(frame: CGRect(
            x: 0, y: 0, width: w/3 - 10.0, height: 40))
        titleLabel.textAlignment = .Center
        titleLabel.textColor = UIColor.orangeColor()
        self.addSubview(titleLabel)
    }

    required init?(coder aDecoder: NSCoder) {
        fatalError("init(coder:) has not been implemented")
    }
}
```

委任模式

接著再回到 `ViewController`，先加上委任需要的協定：

```
class ViewController: UIViewController,  
    UICollectionViewDelegate,  
    UICollectionViewDataSource {  
    // 省略  
}
```

以及必須實作的方法：

```
// 必須實作的方法：每一組有幾個 cell  
func collectionView(collectionView: UICollectionView,  
    numberOfItemsInSection section: Int) -> Int {  
    return 7  
}  
  
// 必須實作的方法：每個 cell 要顯示的內容  
func collectionView(collectionView: UICollectionView,  
    cellForItemAtIndexPath indexPath: NSIndexPath)  
-> UICollectionViewCell {  
    // 依據前面註冊設置的識別名稱 "Cell" 取得目前使用的 cell  
    let cell =  
collectionView.dequeueReusableCellWithIdentifier(  
    "Cell", forIndexPath: indexPath)  
as! MyCollectionViewCell  
  
    // 設置 cell 內容 (即自定義元件裡 增加的圖片與文字元件)  
    cell.imageView.image =  
        UIImage(named: "0\(indexPath.item + 1).jpg")  
    cell.titleLabel.text = "0\(indexPath.item + 1)"  
  
    return cell  
}
```

最後則是其餘委任對象需要的方法，以完善整個 UICollectionView 元件：

```
// 有幾個 section  
func numberOfSectionsInCollectionView(  
    collectionView: UICollectionView) -> Int {  
    return 2
```

```
}

// 點選 cell 後執行的動作
func collectionView(collectionView: UICollectionView,
didSelectItemAtIndexPath indexPath: NSIndexPath) {
    print("你選擇了第 \(indexPath.section + 1) 組的")
    print("第 \(indexPath.item + 1) 張圖片")
}

// 設置 reuse 的 section 的 header 或 footer
func collectionView(collectionView: UICollectionView,
viewForSupplementaryElementOfKind kind: String,
atIndexPath indexPath: NSIndexPath)
-> UICollectionViewReusableView {
    // 建立 UICollectionViewReusableView
    var reusableView = UICollectionViewReusableView()

    // 顯示文字
    let label = UILabel(frame: CGRect(
        x: 0, y: 0,
        width: fullScreenSize.width, height: 40))
    label.textAlignment = .Center

    // header
    if kind == UICollectionElementKindSectionHeader {
        // 依據前面註冊設置的識別名稱 "Header" 取得目前使用的 header
        reusableView =
            collectionView.dequeueReusableCellWithReuseIdentifier(
                UICollectionElementKindSectionHeader,
                forIndexPath: indexPath)
        // 設置 header 的內容
        reusableView.backgroundColor =
            UIColor.darkGrayColor()
        label.text = "Header";
        label.textColor = UIColor.whiteColor()

    } else if kind ==
        UICollectionElementKindSectionFooter {
        // 依據前面註冊設置的識別名稱 "Footer" 取得目前使用的 footer
    }
}
```

```

        reusableView =
collectionView.dequeueReusableCellSupplementaryViewOfKind(
    UICollectionViewElementKindSectionFooter,
    withReuseIdentifier: "Footer",
    forIndexPath: indexPath)
// 設置 footer 的內容
reusableView.backgroundColor =
    UIColor.cyanColor()
label.text = "Footer";
label.textColor = UIColor.blackColor()

}

reusableView.addSubview(label)
return reusableView
}

```

上述程式中需要注意的是最後一個方法，用來顯示每個 section 的 header 與 footer (如果你前面有設置的話)，因為 header 與 footer 是共用一個方法，且都是屬於 `UICollectionViewReusableView` 元件，所以需要先建立一個 `UICollectionViewReusableView` 元件，並使用 `kind` 參數來分辨這時是要設置 header 或 footer。

接著就與 cell 類似，依據註冊時使用的識別名稱，取得重複使用的元件，再設置其內的內容，這邊範例是單純的加上一個 `UILabel`。

以上即為本節範例的內容。

圖片來源

- <https://www.flickr.com/photos/sloalan/15367448967/>
- <https://www.flickr.com/photos/14076637@N04/15097401627/>
- <https://www.flickr.com/photos/126692641@N04/16389075922/>
- <https://www.flickr.com/photos/134525588@N04/20333376532/>
- https://www.flickr.com/photos/jane_nospecial/15076368915/
- <https://www.flickr.com/photos/sdasmarchives/8871276526/>
- <https://www.flickr.com/photos/14076637@N04/16178682209/>

範例

本節範例程式碼放在 [uikit/uicollectionview](#)

搜尋 UISearchController

當有一大筆資訊時，通常可以加上搜尋功能，用來篩選出需要的資訊，這節使用 UITableView 配合 UISearchController 來示範一個搜尋框功能，以下是本節目標：



首先在 Xcode 裡，新建一個 **Single View Application** 類型的專案，取名為 ExUISearchController。

一開始先為 `ViewController` 建立四個屬性：

```
class ViewController: UIViewController {
    var tableView: UITableView!
    var searchController: UISearchController!

    let cities = [
        "臺北市", "新北市", "桃園市", "臺中市", "臺南市",
        "高雄市", "基隆市", "新竹市", "嘉義市", "新竹縣",
        "苗栗縣", "彰化縣", "南投縣", "雲林縣", "嘉義縣",
        "屏東縣", "宜蘭縣", "花蓮縣", "臺東縣", "澎湖縣", ]

    var searchArr: [String] = [String]()
    didSet {
        // 重設 searchArr 後重整 tableView
        self.tableView.reloadData()
    }
}

// 省略
```

以及在 `viewDidLoad()` 中取得螢幕尺寸，以供後續使用，如下：

```
// 取得螢幕的尺寸
let fullScreenSize = UIScreen.mainScreen().bounds.size
```

上述設置中，`cities` 是全部原始的資料，而 `searchArr` 則是隨搜尋文字不同，篩選出來的資料，`didSet` 的使用方法請參考[屬性觀察器](#)。`UITableView` 的方法 `reloadData()` 則是當資料有更新時重整表格用的。

建立 UISearchController

範例使用 `UITableView` 來配合搜尋結果，所以要先在 `viewDidLoad()` 中建立一個 `UITableView`：

```
// 建立 UITableView
self.tableView = UITableView(frame: CGRect(
    x: 0, y: 20,
    width: fullScreenSize.width,
    height: fullScreenSize.height - 20),
    style: .Plain)
self.tableView.registerClass(UITableViewCell.self,
    forCellReuseIdentifier: "Cell")
self.tableView.delegate = self
self.tableView.dataSource = self
self.view.addSubview(self.tableView)
```

詳細使用方式請參考[表格 UITableView](#)。

接著在 `viewDidLoad()` 中建立 `UISearchController`：

```

// 建立 UISearchController 並設置搜尋控制器為 nil
self.searchController =
    UISearchController(searchResultsController: nil)

// 將更新搜尋結果的對象設為 self
self.searchController.searchResultsUpdater = self

// 搜尋時是否隱藏 NavigationBar
// 這個範例沒有使用 NavigationBar 所以設置什麼沒有影響
self.searchController
    .hidesNavigationBarDuringPresentation = false

// 搜尋時是否使用燈箱效果 (會將畫面變暗以集中搜尋焦點)
self.searchController
    .dimsBackgroundDuringPresentation = false

// 搜尋框的樣式
self.searchController.searchBar.searchBarStyle =
    .Prominent

// 設置搜尋框的尺寸為自適應
// 因為會擺在 tableView 的 header
// 所以尺寸會與 tableView 的 header 一樣
self.searchController.searchBar.sizeToFit()

// 將搜尋框擺在 tableView 的 header
self.tableView.tableHeaderView =
    self.searchController.searchBar

```

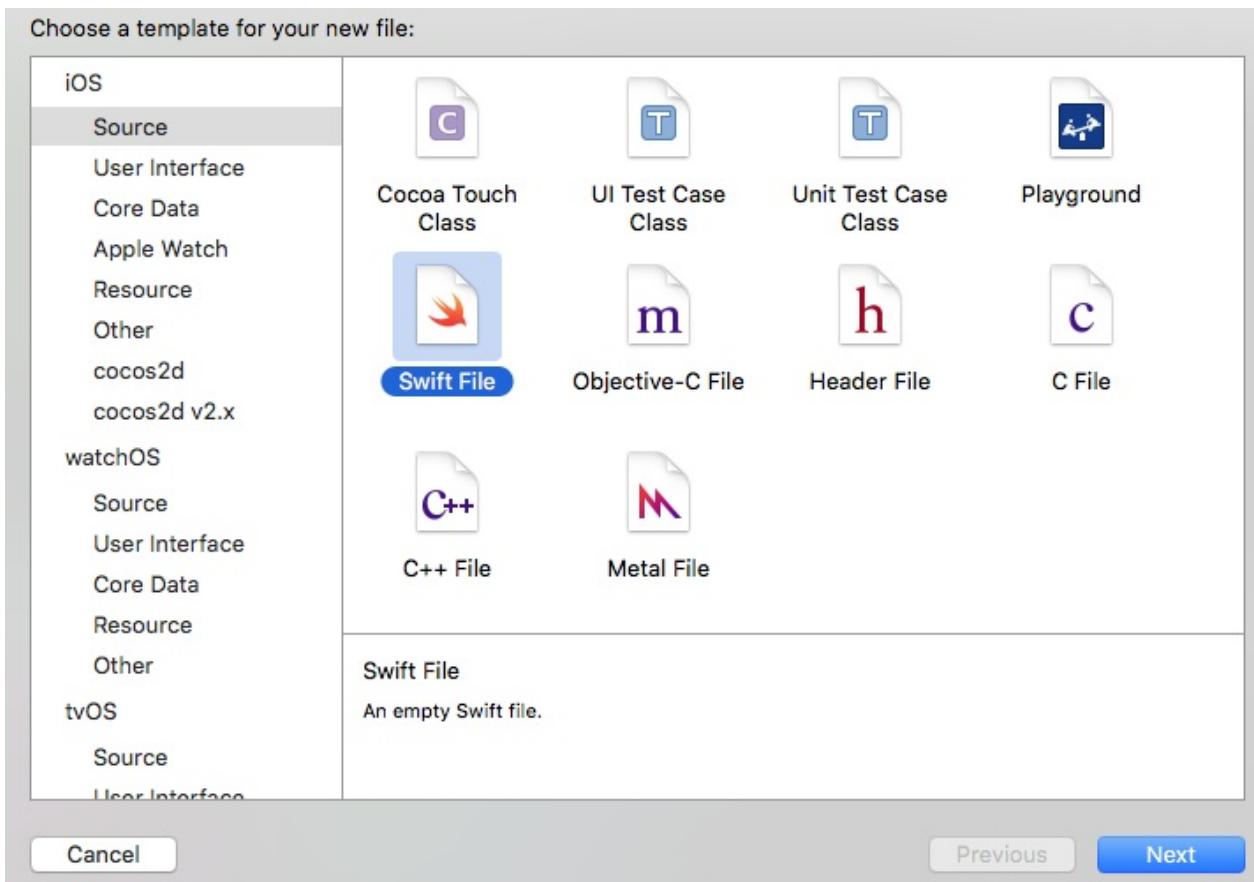
委任模式

除了 UITableView 有設置委任對象，UISearchController 也同樣設置了更新搜尋結果的對象，在前面的範例中已經示範過了在 `self` (也就是 ViewController 本身) 以及 [另外建立一個獨立檔案來實作委任對象的方法](#)。這邊則介紹第三種方式，以擴展來實作委任的方法。

實際上就與在 `self` 裡實作一樣，但基於 **Swift** 擴展 (**extension**) 的特性，可以對 ViewController 新增一個擴展，來將委任方式寫在另外一個擴展檔案中。

擴展的詳細說明請參考前面章節介紹的內容。

請先以新增檔案的方式新增一個檔案，命名為 ViewControllerExtensions，不同的地方在於，請選擇 iOS > Source > Swift File，如下圖：



建立好檔案後，開啟 ViewControllerExtensions.swift，先將原本的 import Foundation 改為 import UIKit，並在其中以擴展 ViewController 的方式，加上需要實作的委任方法：

```
extension ViewController: UITableViewDataSource {
    func tableView(tableView: UITableView,
                  numberOfRowsInSection section: Int) -> Int {
        if (self.searchController.active) {
            return self.searchArr.count
        } else {
            return self.cities.count
        }
    }

    func tableView(tableView: UITableView,
                  cellForRowAtIndexPath indexPath: NSIndexPath)
```

```

-> UITableViewCell {
    let cell =
        tableView.dequeueReusableCell(withIdentifier(
            "Cell", forIndexPath: indexPath)

    if (self.searchController.active) {
        cell.textLabel?.text =
            self.searchArr[indexPath.row]
        return cell
    } else {
        cell.textLabel?.text =
            self.cities[indexPath.row]
        return cell
    }
}

extension ViewController: UITableViewDelegate {
    func tableView(tableView: UITableView,
                  didSelectRowAtIndexPath indexPath: NSIndexPath){
        tableView.deselectRowAtIndexPath(
            indexPath, animated: true)
        if (self.searchController.active) {
            print(
                "你選擇的是 \(self.searchArr[indexPath.row])")
        } else {
            print(
                "你選擇的是 \(self.cities[indexPath.row])")
        }
    }
}

extension ViewController: UISearchResultsUpdating {
    func updateSearchResultsForSearchController(searchController
        : UISearchController){
        // 取得搜尋文字
        guard let searchText =
            searchController.searchBar.text else {
                return
            }
}

```

```
// 使用陣列的 filter() 方法篩選資料
self.searchArr = self.cities.filter(
    { (city) -> Bool in
        // 將文字轉成 NSString 型別
        let cityText:NSString = city

        // 比對這筆資訊有沒有包含要搜尋的文字
        return (cityText.rangeOfString(
            searchText, options:
            NSStringCompareOptions.CaseInsensitiveSearch).location)
            != NSNotFound
    }
)
}
```

上述程式中，前兩個擴展用來實作 UITableView 委任的方法，需要注意的是 UISearchController 有一個屬性 active，用來表示目前是否為搜尋狀態，當搜尋狀態時就是顯示搜尋後的結果 searchArr，而非搜尋狀態時則是顯示所有資訊 cities。

最後一個擴展則是用來實作更新搜尋結果的方法，首先是取得搜尋的文字，接著則是對原始資料的陣列使用 filter() 方法篩選資訊，以比對文字的方式來搜尋。

guard 的使用方式請參考[提前退出](#)。

以上即為本節範例的內容。

範例

本節範例程式碼放在 [uikit/uiseachcontroller](#)

滑動視圖 UIScrollView

當一個元件的實際視圖範圍比可見視圖範圍大時，會加上滑動條(scroll bar)讓使用者自由滑動，UIScrollView 就是有這一特性的最基本元件。實際上，前面章節介紹過的 UITextView 、 UITableView 及 UICollectionView 都是繼承自 UIScrollView ，所以他們也都繼承了 UIScrollView 的特性。

本節會先介紹如何建立一個基本的 UIScrollView 並介紹常用的屬性及方法，接著會配合 UIPageControl 元件建立像是新手導覽左右滑動的分頁功能。

基本的 UIScrollView

首先在 Xcode 裡，新建一個 Single View Application 類型的專案，取名為 ExUIScrollView 。

一開始先為 ViewController 建立兩個屬性：

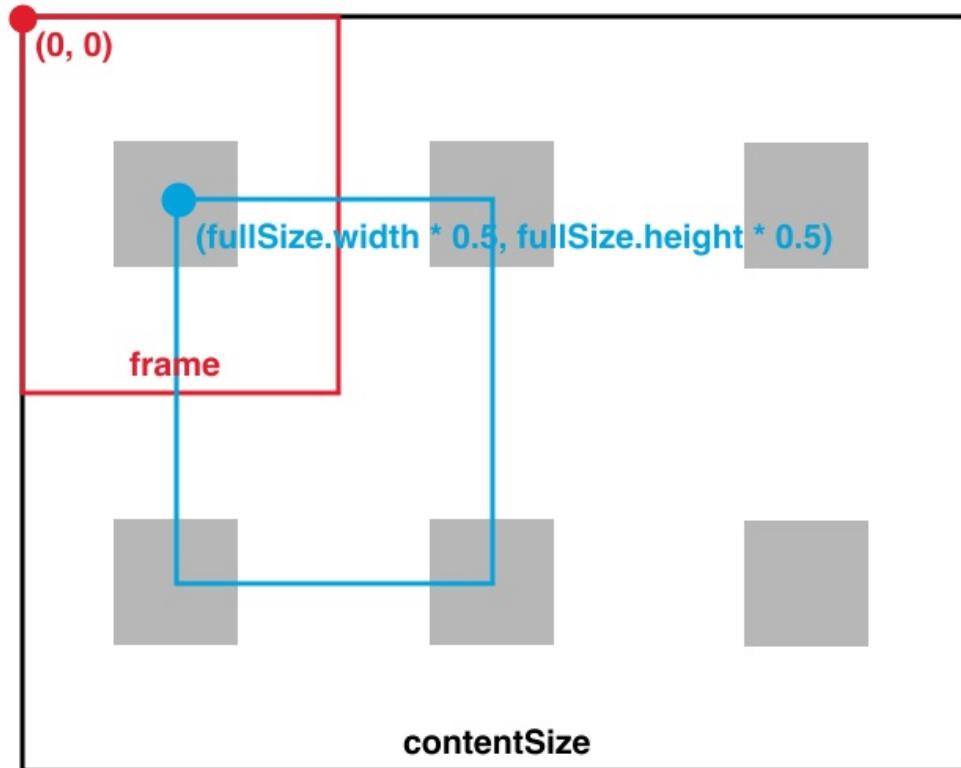
```
class ViewController: UIViewController {
    var myScrollView: UIScrollView!
    var fullSize :CGSize!

    // 省略
}
```

以及在 viewDidLoad() 中取得螢幕尺寸，以供後續使用，如下：

```
// 取得螢幕的尺寸
fullSize = UIScreen.mainScreen().bounds.size
```

下圖是這個範例的示意圖，紅色矩形為屬性 frame 的尺寸，也就是可見視圖範圍。黑色矩形為屬性 contentSize 的尺寸，也就是實際視圖範圍。藍色點點則是屬性 contentOffset 設置的點(CGPoint)，表示起始畫面偏移量，如果不設定則是預設為紅色點點 (0, 0) ：



接著使用程式碼在 `viewDidLoad()` 中建立一個 UIScrollView :

```
// 建立 UIScrollView
myScrollView = UIScrollView()

// 設置尺寸 也就是可見視圖範圍
myScrollView.frame = CGRect(
    x: 0, y: 20, width: fullSize.width,
    height: fullSize.height - 20)

// 實際視圖範圍 為 3*2 個螢幕大小
myScrollView.contentSize = CGSize(
    width: fullSize.width * 3,
    height: fullSize.height * 2)

// 是否顯示水平的滑動條
myScrollView.showsHorizontalScrollIndicator = true

// 是否顯示垂直的滑動條
myScrollView.showsVerticalScrollIndicator = true
```

```
// 滑動條的樣式  
myScrollView.indicatorStyle = .Black  
  
// 是否可以滑動  
myScrollView.scrollEnabled = true  
  
// 是否可以按狀態列回到最上方  
myScrollView.scrollsToTop = false  
  
// 是否限制滑動時只能單個方向 垂直或水平滑動  
myScrollView.directionalLockEnabled = false  
  
// 滑動超過範圍時是否使用彈回效果  
myScrollView.bounces = true  
  
// 縮放元件的預設縮放大小  
myScrollView.zoomScale = 1.0  
  
// 縮放元件可縮小到的最小倍數  
myScrollView.minimumZoomScale = 0.5  
  
// 縮放元件可放大到的最大倍數  
myScrollView.maximumZoomScale = 2.0  
  
// 縮放元件縮放時是否在超過縮放倍數後使用彈回效果  
myScrollView.bouncesZoom = true  
  
// 設置委任對象  
myScrollView.delegate = self  
  
// 起始的可見視圖偏移量 預設為 (0, 0)  
// 設定這個值後 就會將原點滑動至這個點起始  
myScrollView.contentOffset = CGPoint(  
    x: fullSize.width * 0.5, y: fullSize.height * 0.5)  
  
// 以一頁為單位滑動  
myScrollView.pagingEnabled = false  
  
// 加入到畫面中
```

```
self.view.addSubview(myScrollView)
```

以及在 `viewDidLoad()` 中建立用來凸顯滑動效果的六個 `UIView`：

```
// 建立六個 UIView 來顯示出滑動的路徑
var myUIView = UIView()
for i in 0...2 {
    for j in 0...1 {
        myUIView = UIView(frame: CGRect(
            x: 0, y: 0, width: 100, height: 100))
        myUIView.tag = i * 10 + j + 1
        myUIView.center = CGPoint(
            x: fullSize.width * (0.5 + CGFloat(i)),
            y: fullSize.height * (0.5 + CGFloat(j)))
        let color =
            ((CGFloat(i) + 1) * (CGFloat(j) + 1)) / 12.0
        myUIView.backgroundColor = UIColor.init(
            red: color, green: color, blue: color, alpha: 1)
        myScrollView.addSubview(myUIView)
    }
}
```

可以縮放的元件還需要由委任對象實作的方法來設置，所以緊接著再設置委任需要的協定：

```
class ViewController: UIViewController, UIScrollViewDelegate {
    // 省略
}
```

以及委任模式可以實作的方法。UIScrollView 提供給委任對象可以實作的方法很多，你可以在滑動或是縮放的各個階段(開始、進行中與結束等等)設置自定義的動作，以下示範幾個常用的：

```
// 開始滑動時
func scrollViewWillBeginDragging(scrollView: UIScrollView) {
    print("scrollViewWillBeginDragging")
}
```

```
// 滑動時
func scrollViewDidScroll(scrollView: UIScrollView) {
    //print("scrollViewDidScroll")
}

// 結束滑動時
func scrollViewDidEndDragging(scrollView: UIScrollView,
    willDecelerate decelerate: Bool) {
    print("scrollViewDidEndDragging")
}

// 縮放的元件
func viewForZoomingInScrollView(scrollView: UIScrollView)
-> UIView? {
    // 這邊用來示範縮放的元件是 tag 為 1 的 UIView
    // 也就是左上角那個 UIView
    return self.view.viewWithTag(1)
}

// 開始縮放時
func scrollViewWillBeginZooming(scrollView: UIScrollView,
    withView view: UIView?) {
    print("scrollViewWillBeginZooming")
}

// 縮放時
func scrollViewDidZoom(scrollView: UIScrollView) {
    //print("scrollViewDidZoom")
}

// 結束縮放時
func scrollViewDidEndZooming(scrollView: UIScrollView,
    withView view: UIView?, atScale scale: CGFloat) {
    print("scrollViewDidEndZooming")

    // 縮放元件時 會將 contentSize 設為這個元件的尺寸
    // 會導致 contentSize 過小而無法滑動
    // 所以縮放完後再將 contentSize 設回原本大小
    myScrollView.contentSize = CGSize(
```

```
width: fullSize.width * 3, height: fullSize.height * 2)  
}
```

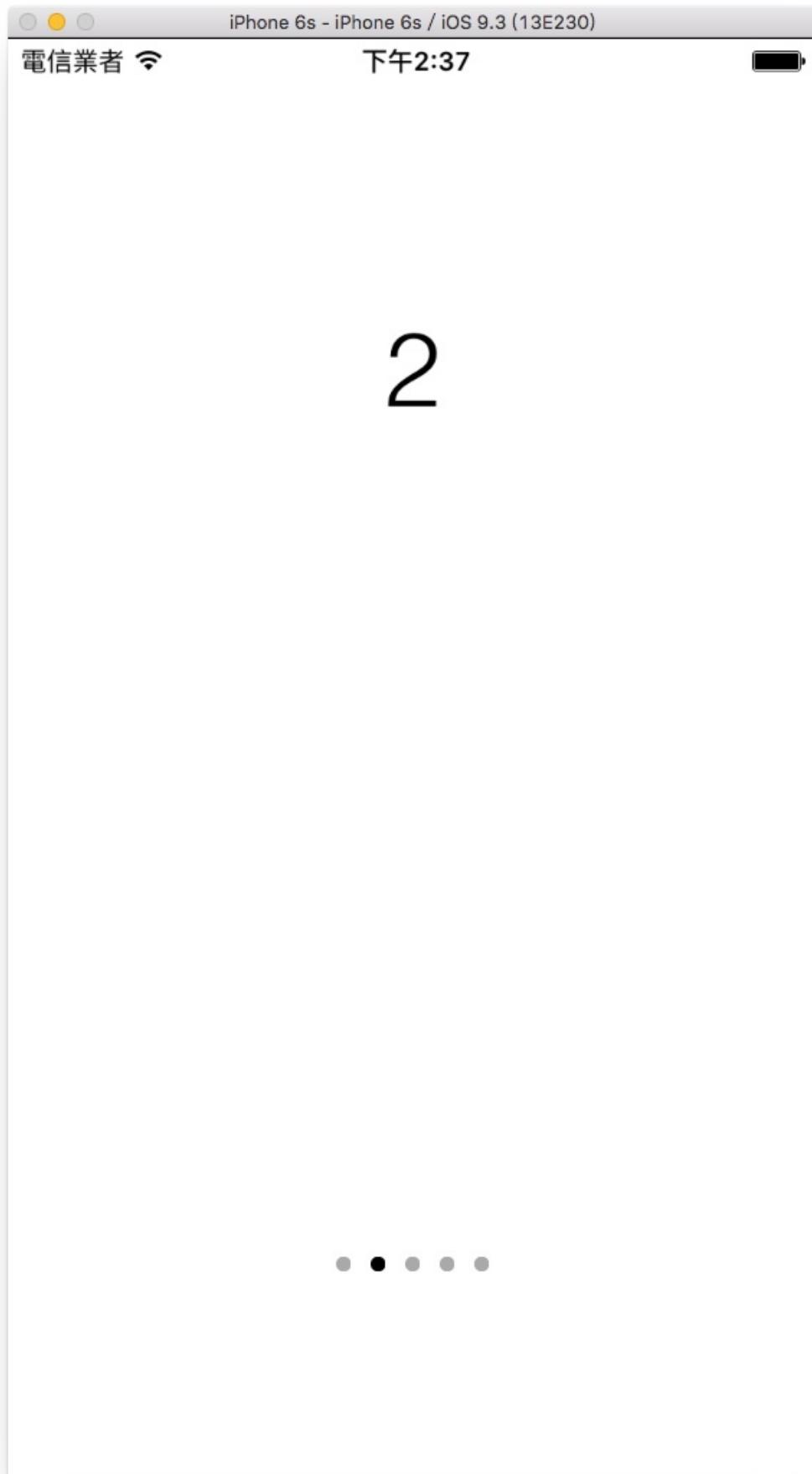
如果要在模擬器上使用縮放手勢，請按著 `option`，模擬器畫面上會出現兩個灰色的圓圈，就可以開始做縮放的動作。

以上就是這個範例的內容。

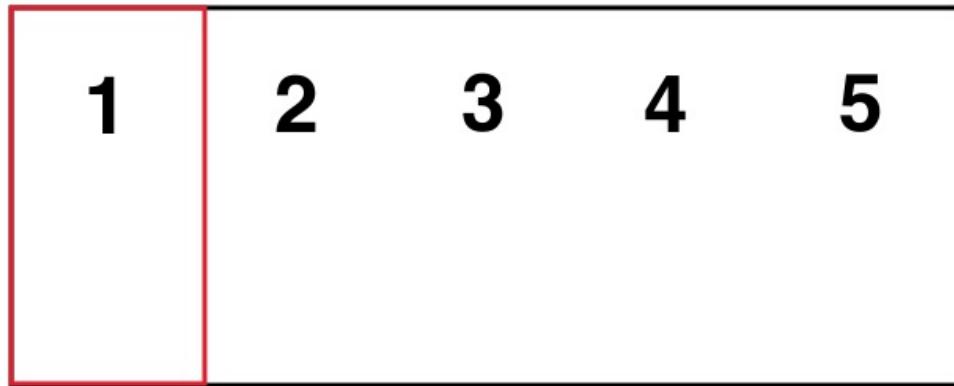
與 **UIPageControl** 的綜合應用

UIPageControl 其實就是很常見放在畫面下方的一排圓點點，用來表示目前所在頁數，這個範例會示範一個常用於首次打開應用程式時，新手導覽的功能，分為若干頁，你可以左右滑動來觀看導覽步驟。

以下為這個範例的目標，分為五頁，各放置一個 **UILabel** 來表示每頁的內容，並使用 **UIPageControl** 表示目前所在頁數：



先看一下這個範例的示意圖，整個頁面為 5 個螢幕尺寸大小，可以左右滑動換頁：



首先在 Xcode 裡，新建一個 **Single View Application** 類型的專案，取名為 ExIntroStepByStep。

一開始先為 `ViewController` 建立三個屬性：

```
class ViewController: UIViewController {  
    var myScrollView: UIScrollView!  
    var pageControl: UIPageControl!  
    var fullSize :CGSize!  
  
    // 省略  
}
```

以及在 `viewDidLoad()` 中取得螢幕尺寸，以供後續使用，如下：

```
// 取得螢幕的尺寸  
fullSize = UIScreen.mainScreen().bounds.size
```

首先在 `viewDidLoad()` 中建立一個 `UIScrollView`：

```
// 建立 UIScrollView  
myScrollView = UIScrollView()  
  
// 設置尺寸 也就是可見視圖範圍  
myScrollView.frame = CGRect(  
    x: 0, y: 20,  
    width: fullSize.width, height: fullSize.height - 20)  
  
// 實際視圖範圍  
myScrollView.contentSize = CGSize(  
    width: fullSize.width * 5, height: fullSize.height - 20)  
  
// 是否顯示滑動條  
myScrollView.showsHorizontalScrollIndicator = false  
myScrollView.showsVerticalScrollIndicator = false  
  
// 滑動超過範圍時是否使用彈回效果  
myScrollView.bounces = true  
  
// 設置委任對象  
myScrollView.delegate = self  
  
// 以一頁為單位滑動  
myScrollView.pagingEnabled = true  
  
// 加入到畫面中  
self.view.addSubview(myScrollView)
```

接著在 `viewDidLoad()` 中建立用來顯示頁數的 `UIPageControl`：

```
// 建立 UIPageControl 設置位置及尺寸
pageControl = UIPageControl(frame: CGRect(
    x: 0, y: 0, width: fullSize.width * 0.85, height: 50))
pageControl.center = CGPoint(
    x: fullSize.width * 0.5, y: fullSize.height * 0.85)

// 有幾頁 就是有幾個點點
pageControl.numberOfPages = 5

// 起始預設的頁數
pageControl.currentPage = 0

// 目前所在頁數的點點顏色
pageControl.currentPageIndicatorTintColor =
    UIColor.blackColor()

// 其餘頁數的點點顏色
pageControl.pageIndicatorTintColor = UIColor.lightGrayColor()

// 增加一個值改變時的事件
pageControl.addTarget(
    self,
    action: #selector(ViewController.pageChanged),
    forControlEvents: .ValueChanged)

// 加入到基底的視圖中 (不是加到 UIScrollView 裡)
// 因為比較後面加入 所以會蓋在 UIScrollView 上面
self.view.addSubview(pageControl)
```

最後在 `viewDidLoad()` 中加入五個 `UILabel` 用來代表每個頁面的內容：

```
// 建立 5 個 UILabel 來顯示每個頁面內容
var myLabel = UILabel()
for i in 0...4 {
    myLabel = UILabel(frame: CGRect(
        x: 0, y: 0, width: fullSize.width, height: 40))
    myLabel.center = CGPoint(
        x: fullSize.width * (0.5 + CGFloat(i)),
        y: fullSize.height * 0.2)
    myLabel.font = UIFont(name: "Helvetica-Light", size: 48.0)
    myLabel.textAlignment = .Center
    myLabel.text = "\(i + 1)"
    myScrollView.addSubview(myLabel)
}
```

因為會用到 UIScrollView 委任模式的方法，所以先為 ViewController 加上委任需要的協定：

```
class ViewController: UIViewController, UIScrollViewDelegate {
    // 省略
}
```

再在 ViewController 中實作需要的委任方法：

```
// 滑動結束時
func scrollViewDidEndDecelerating(scrollView: UIScrollView) {
    // 左右滑動到新頁時 更新 UIPageControl 顯示的頁數
    let page = Int(scrollView.contentOffset.x / scrollView.frame.size.width)
    pageControl.currentPage = page
}
```

最後在 ViewController 中加上點擊 UIPageControl 的點點時執行動作的方法：

```
// 點擊點點換頁
func pageChanged(sender: UIPageControl) {
    // 依照目前圓點在的頁數算出位置
    var frame = myScrollView.frame
    frame.origin.x =
        frame.size.width * CGFloat(sender.currentPage)
    frame.origin.y = 0

    // 再將 UIScrollView 滑動到該點
    myScrollView.scrollRectToVisible(frame, animated:true)
}
```

以上即為這個範例的內容。

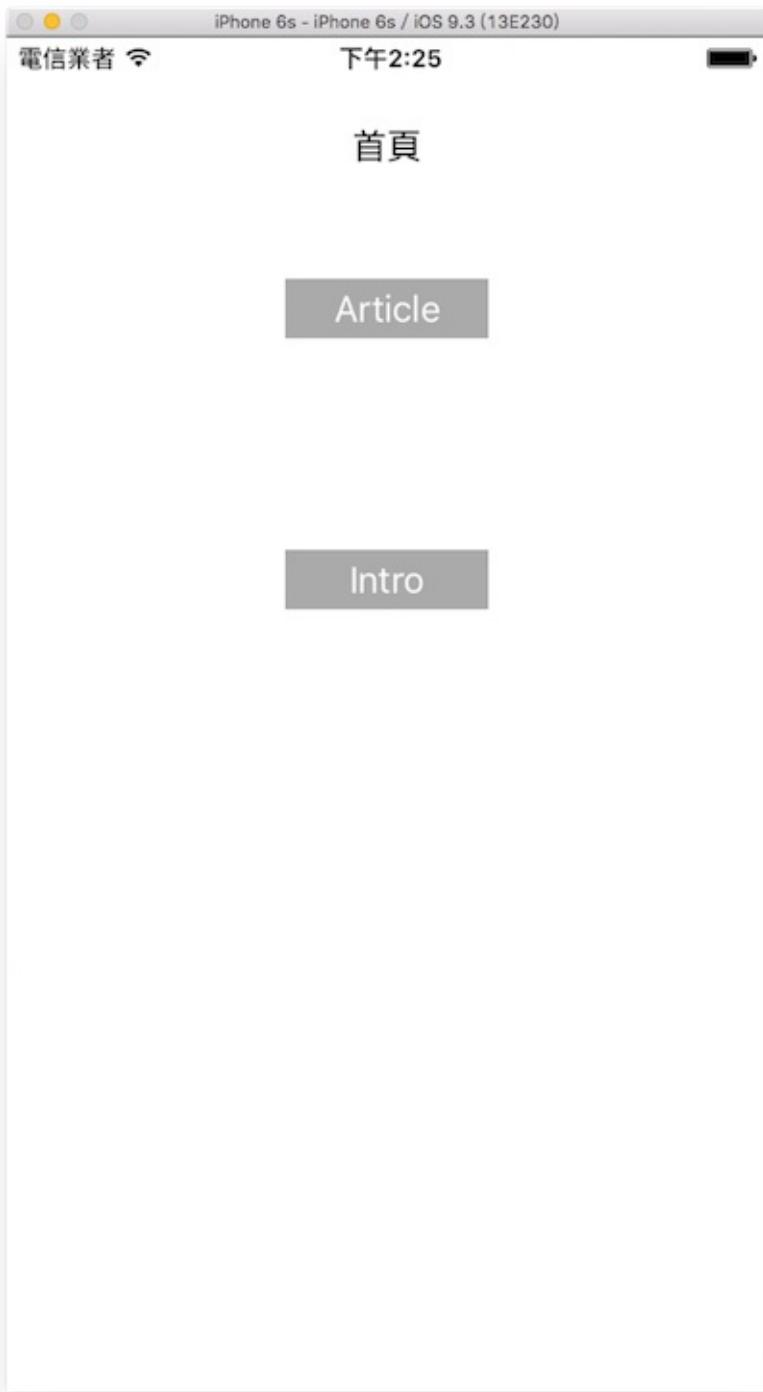
範例

本節範例程式碼放在 [uikit/uiscrollview](#)

多頁面

前面章節介紹了許多常用的 UIKit 元件，但都是運作在單一個頁面中(也就是單一個 UIViewController)，前面提到過一個 UIViewController 負責了一個頁面的運作。)，這節會介紹如何建立多個頁面，並在這些頁面中切換。

本節的目標如下，建立兩個按鈕，分別會進入不同頁面，兩個頁面都會有回上頁的按鈕， Article 頁裡還會有一個按鈕，可以再進入 Article Detail 頁面：



首先在 Xcode 裡，新建一個 **Single View Application** 類型的專案，取名為 ExMultiPages。

一開始先以新增檔案的方式加入三個繼承自 `UIViewController` 的 `.swift` 檔案，分別命名

為 `IntroViewController` 、`ArticleViewController` 及 `ArticleDetailViewController`。

將上述三個 UIViewController 與 ViewController 都分別在各自的 viewDidLoad() 中取得螢幕尺寸及設置底色，以供後續使用，如下：

```
// 取得螢幕的尺寸  
let fullSize = UIScreen.mainScreen().bounds.size  
  
// 設置底色  
self.view.backgroundColor = UIColor.whiteColor()
```

切換頁面

首先在 ViewController 的 viewDidLoad() 裡建立一個作為標題的 UILabel 與兩個切換頁面用的 UIButton：

```
// 頁面標題
let myLabel = UILabel(frame: CGRect(
    x: 0, y: 0, width: fullSize.width, height: 40))
myLabel.center = CGPoint(
    x: fullSize.width * 0.5, y: fullSize.height * 0.08)
myLabel.textAlignment = .Center
myLabel.text = "首頁"
self.view.addSubview(myLabel)

// 建立前往 Article 頁面的 UIButton
var myButton = UIButton(frame: CGRect(
    x: 0, y: 0, width: 100, height: 30))
myButton.setTitle("Article", forState: .Normal)
myButton.backgroundColor = UIColor.lightGrayColor()
myButton.addTarget(
    nil, action: #selector(ViewController.goArticle),
    forControlEventss: .TouchUpInside)
myButton.center = CGPoint(
    x: fullSize.width * 0.5, y: fullSize.height * 0.2)
self.view.addSubview(myButton)

// 建立前往 Intro 頁面的 UIButton
myButton = UIButton(frame: CGRect(
    x: 0, y: 0, width: 100, height: 30))
myButton.setTitle("Intro", forState: .Normal)
myButton.backgroundColor = UIColor.lightGrayColor()
myButton.addTarget(nil,
    action: #selector(ViewController.goIntro),
    forControlEventss: .TouchUpInside)
myButton.center = CGPoint(
    x: fullSize.width * 0.5, y: fullSize.height * 0.4)
self.view.addSubview(myButton)
```

以及在 ViewController 加上按下按鈕執行動作的方法：

```

func goArticle() {
    self.presentViewController(ArticleViewController(), animated:
: true, completion: nil)
}

func goIntro() {
    self.presentViewController(IntroViewController(), animated:
true, completion: nil)
}

```

上述程式可以看到，切換頁面主要是使用 `self` 的方

法 `presentViewController(viewControllerToPresent:, animated:, completion:)`，參數分別為：

- `viewControllerToPresent`：要切換前往的頁面，也就是要交棒給予的 `UIViewController`，所以上面兩個按鈕按下後就分別前往 `ArticleViewController()` 及 `IntroViewController()`。
- `animated`：換頁動作是否加上過場動畫，填 `true` 的話會由下向上滑出新的頁面，填 `false` 則是直接替換畫面。
- `completion`：切換頁面完成後執行的動作，這邊是一個閉包(`closure`)，你可以在裡面加上自定義的程式。

退出頁面

接著看到 `IntroViewController.swift`，一樣是加上一個 `UILabel` 及一個 `UIButton`，這個按鈕則是用來退出頁面，程式碼與前述類似，這邊就略過。主要看到按下退出頁面按鈕後執行的動作，在 `IntroViewController` 內加上方法，如下：

```

func goBack() {
    self.dismissViewControllerAnimated(true, completion:nil)
}

```

上述程式可以看到要退出頁面，是使用 `self` 的方

法 `dismissViewControllerAnimated(flag:, completion:)`，參數分別如下：

- `flag`：換頁動作是否加上過場動畫，填 `true` 的話會由上向下退出頁面，

填 `false` 則是直接替換畫面。

- **completion**：切換頁面完成後執行的動作，這邊是一個閉包(**closure**)，你可以在裡面加上自定義的程式。

Article 要進到 Article Detail 也是如前述一樣的方式，這邊就省略，完整內容請參考範例程式碼。

頁面開啓退出的過程

這節還要介紹一個功能，就是在頁面開啓及退出的各個階段，你都可以加入自定義的程式，在首頁 `ViewController` 裡面你可以看到如下幾個方法：

```
override func viewDidAppear(animated: Bool) {
    super.viewDidAppear(animated)

    print("viewDidAppear")
}

override func viewDidDisappear(animated: Bool) {
    super.viewDidDisappear(animated)

    print("viewDidDisappear")
}

override func viewWillDisappear(animated: Bool) {
    super.viewWillDisappear(animated)

    print("viewWillDisappear")
}

override func viewDidLoad() {
    super.viewDidLoad()

    print("viewDidLoad")
}
```

這些方法加上一開始的 `viewDidLoad()` 分別會在不同階段執行，執行的順序如下：

- `viewDidLoad()`：最先被執行，時間點在 **View** 被載入時，不論切換退出這個頁面幾次，一個頁面只會執行一次 `viewDidLoad()`。
- `viewWillAppear()`：在 `viewDidLoad()` 之後被執行，時間點在 **View** 要被呈現前，每次切換到這個頁面時都會執行。
- `viewDidAppear()`：在 `viewWillAppear()` 之後被執行，時間點在 **View** 呈現後，每次切換到這個頁面時都會執行。
- `viewWillDisappear()`：執行的時間點在 **View** 要結束前，每次要切換到別頁或是退出這個頁面時都會執行。
- `viewDidDisappear()`：執行的時間點在 **View** 完全結束後，每次要切換到別頁或是退出這個頁面時都會執行。

以上方法可以依照需求寫入不同的程式。例如，如果每次進入一個頁面都需要更新資訊，那這個更新資訊的動作就應該放在 `viewWillAppear()` 裡而不是 `viewDidLoad()` 內。

以上即為本節範例的內容。

範例

本節範例程式碼放在 [uikit/multipages](#)

導覽控制器 UINavigationController

前一個章節介紹了如何在多個頁面間切換，這節接著要介紹一個相當常見的元件：導覽控制器 UINavigationController，可以更為方便的掌控頁面切換。

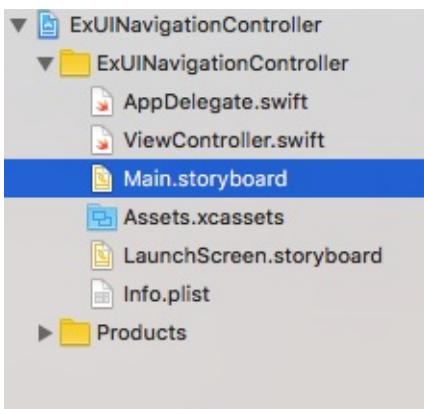
導覽控制器就像是一個容器，裡面可以用來放置及疊放各個頁面，畫面上方預設會有一個導覽列(Navigation Bar)，其中可以放置標題及按鈕，來切換或退出頁面，像是內建的 設定 App 就是一個例子，如下：



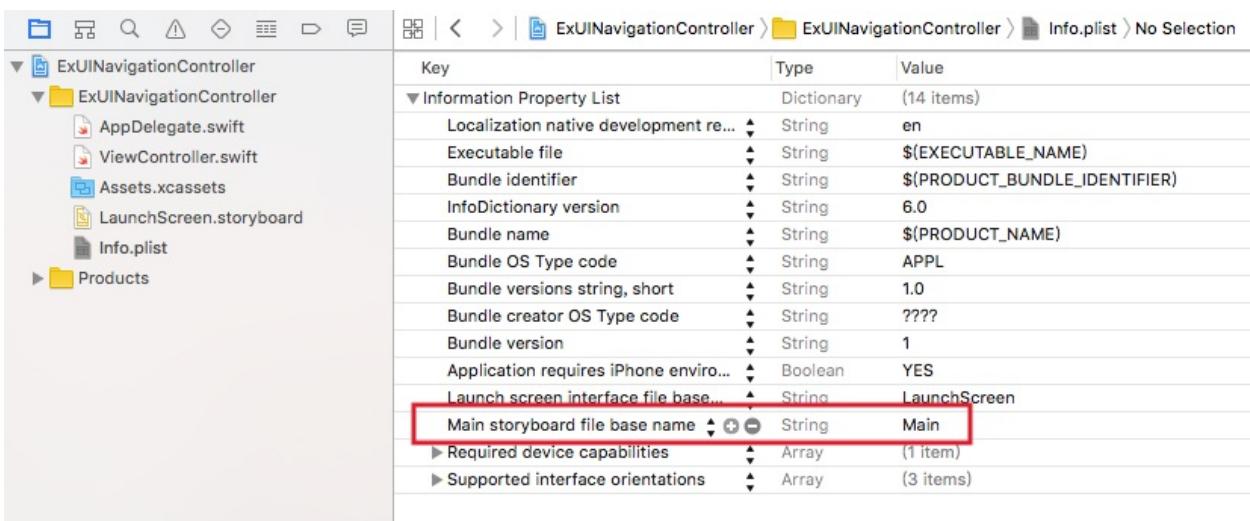
手動建立頁面

在講導覽控制器之前，要先介紹如何手動建立頁面。最一開始有提過這本書的內容都是以純程式碼為主，但實際上每次新建一個 **Single View Application** 類型的專案時，這個專案已經都內建了一個 Storyboard 以及相關的設定，這小節會介紹如何移除掉內建的 Storyboard 並手動建立頁面。

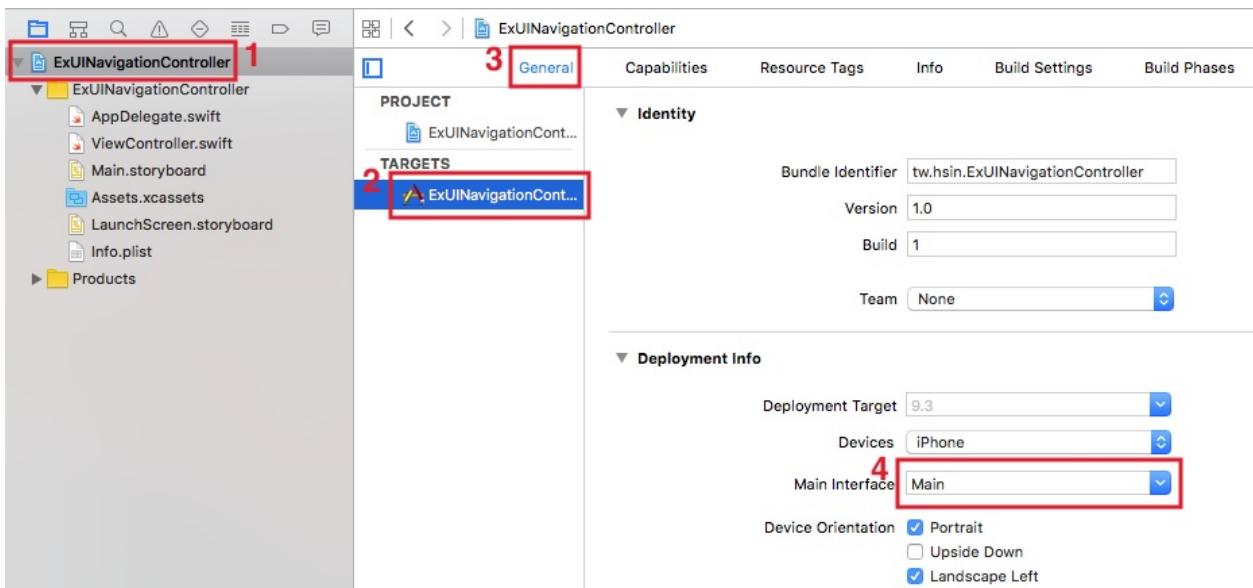
第一步先以刪除檔案的方式將 Storyboard 刪除，也就是下圖中列表的這隻檔案 Main.storyboard：



第二步接著看到 Info.plist，找到並刪除 Main storyboard file base name 這個欄位。(按減號 - 就可以刪除了。)



或是將 General > Deployment Info > Main Interface 這個欄位清空，如下：(這邊的設定會與上面指的 Info.plist 設定連動，所以兩邊清空其中一個，另一個就會一併清空，沒有的話就通通清空。)



最後一步轉往到 `AppDelegate.swift`，在 [UIKit 初探](#) 有提到，這隻檔案是負責應用程式的生命週期，所以接著要在 `AppDelegate` 類別中的 `application(application: UIApplication, didFinishLaunchingWithOptions launchOptions: [NSObject: AnyObject]?) -> Bool` 方法中手動加上頁面，這個方法執行的時間點在應用程式啓動後：

```
func application(application: UIApplication,
didFinishLaunchingWithOptions launchOptions:
[NSObject: AnyObject]?) -> Bool {
    // 建立一個 UIWindow
    self.window = UIWindow(frame:
        UIScreen.mainScreen().bounds)

    // 設置底色
    self.window!.backgroundColor = UIColor.whiteColor()

    // 設置根視圖控制器
    self.window!.rootViewController = ViewController();

    // 將 UIWindow 設置為可見的
    self.window!.makeKeyAndVisible()

    return true
}
```

上述程式中，先建立一個 `UIWindow`，用來顯示應用程式所有畫面的視窗，有點像是桌面程式的視窗，不過在寫 Mac 應用程式時可能會有多個視窗可以切換，而 iOS 下則較為單純，只會有一個視窗，也就是這邊設置的 `self.window`。

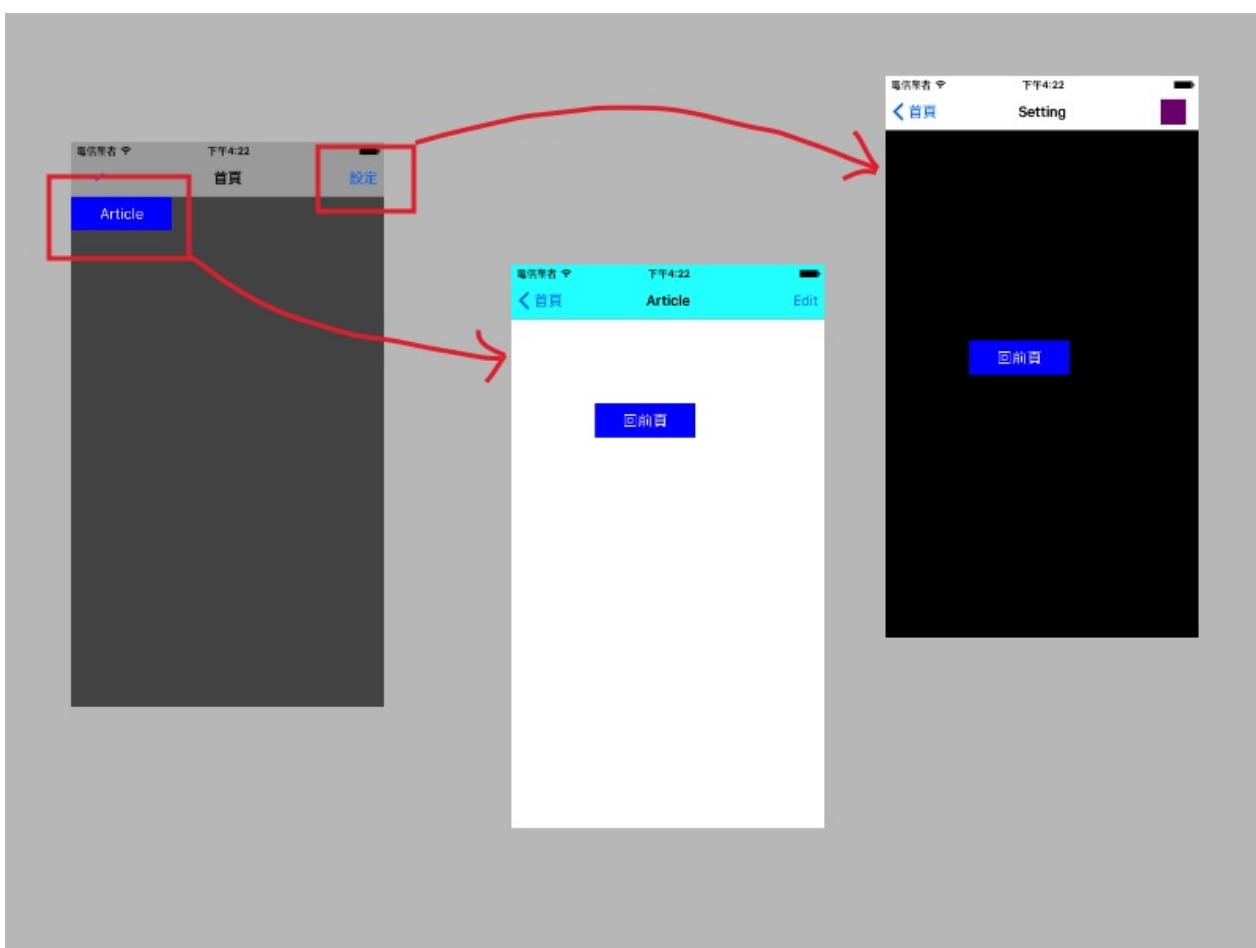
接著要為視窗設置屬性 `rootViewController` 根視圖控制器，也就是應用程式啓動後進到的第一個視圖(View)所處的視圖控制器(ViewController)，這邊設置為 `ViewController()`，你也可以依照需求設置成自己另外建立的 `UIViewController`。

最後將這個視窗以 `makeKeyAndVisible()` 方法設置為可見的，完成手動建立頁面的步驟。

之後就如同先前的學習一樣，在 `ViewController.swift` 裡面設置自己需要的功能與元件。

建立 UINavigationController

這個範例的目標如下，建立一個導覽控制器，並內建一個主頁與兩個次頁，可以使導覽列上的按鈕或是視圖中的按鈕切換頁面：



首先在 Xcode 裡，新建一個 **Single View Application** 類型的專案，取名為 ExUINavigationController。

接著先以新增檔案的方式加入兩個繼承自 UIViewController 的 .swift 檔案，分別命名為 ArticleViewController 及 SettingViewController。以及以加入檔案的方式加入一張按鈕的圖片。

AppDelegate.swift

一開始先依據前一小節的步驟移除 Storyboard 檔案與相關設定，接著在 AppDelegate.swift 中將根視圖控制器設為一個 UINavigationController，如下：

```
func application(application: UIApplication,
didFinishLaunchingWithOptions launchOptions:
[NSObject: AnyObject]?) -> Bool {
    // 建立一個 UIWindow
    self.window = UIWindow(frame:
        UIScreen mainScreen().bounds)

    // 設置底色
    self.window!.backgroundColor = UIColor.whiteColor()

    // 設置根視圖控制器
    let nav = UINavigationController(
        rootViewController: ViewController())
    self.window!.rootViewController = nav

    // 將 UIWindow 設置為可見的
    self.window!.makeKeyAndVisible()

    return true
}
```

上述程式可以看到將 self.window 的 rootViewController 設為導覽控制器 UINavigationController，而這個導覽控制器只是一個容器，所以它也需要設置一個 rootViewController，這邊則是設置成已經存在的 ViewController()，你也可以依照需求設置成自己另外建立的 UIViewController。

ViewController.swift

接著轉到 ViewController.swift 的 `viewDidLoad()` 方法中，將導覽列的設定與按鈕設置寫入：

```
// 底色
self.view.backgroundColor = UIColor.darkGrayColor()

// 導覽列標題
self.title = "首頁"

// 導覽列底色
self.navigationController?.navigationBar.barTintColor =
UIColor.lightGrayColor()

// 導覽列是否半透明
self.navigationController?.navigationBar.translucent = false

// 導覽列左邊按鈕
let leftButton = UIBarButtonItem(
image: UIImage(named:"check"),
style:.Plain ,
target:self ,
action: #selector(ViewController.check))
// 加到導覽列中
self.navigationItem.leftBarButtonItem = leftButton

// 導覽列右邊按鈕
let rightButton = UIBarButtonItem(
title:"設定",
style:.Plain,
target:self,
action:#selector(ViewController.setting))
// 加到導覽列中
self.navigationItem.rightBarButtonItem = rightButton

// 建立一個按鈕
let myButton = UIButton(frame: CGRect(
x: 0, y: 0, width: 120, height: 40))
myButton.setTitle("Article", forState: .Normal)
```

```

myButton.backgroundColor = UIColor.blueColor()
myButton.addTarget(
    self,
    action: #selector(ViewController.article),
    forControlEvents: .TouchUpInside)
self.view.addSubview(myButton)

```

上述程式中，首先看

到 `self.navigationController?.navigationBar.translucent` 這個屬性，用來表示導覽列是否要半透明，但需要在導覽列沒有設置底色時才有這個效果。另外還影響到內部視圖的原點位置，如果設為 `true`，則原點與導覽列的原點一樣，都是整個畫面的左上角，而設為 `false` 時，內部視圖的原點則會被設在導覽列下方，你可以將這個屬性設為不同值來看看兩種情況。

接著看到設置導覽列按鈕是使用 `UIBarButtonItem()` 方法，而不是原始的 `UIButton()`，它有四種常見的不同初始化方式，這邊先介紹兩個，稍後兩個次頁會分別各介紹一種。按鈕設置完成後要使用 `self.navigationItem` 的 `leftBarButtonItem` 或 `rightBarButtonItem` 屬性來加到導覽列中。

首先兩個方式為，導覽列左邊按鈕是將按鈕設置為一個圖片。而導覽列右邊按鈕則是設置為一個自定義文字。

接著則在 `ViewController` 中加上按鈕執行動作的方法：

```

func article() {
    self.navigationController?.pushViewController(
        ArticleViewController(), animated: true)
}

func check() {
    print("check button action")
}

func setting() {
    self.navigationController?.pushViewController(
        SettingViewController(), animated: true)
}

```

上述程式可以看到，導覽控制器用來切換頁面的方法為 `pushViewController()`，參數分別為要前往的頁面的視圖控制器及是否要有過場動畫。

ArticleViewController.swift

接著看到 ArticleViewController.swift 的 `viewDidLoad()`：

```
// 底色
self.view.backgroundColor = UIColor.whiteColor()

// 導覽列標題
self.title = "Article"

// 導覽列底色
self.navigationController?.navigationBar.barTintColor =
    UIColor.cyanColor()

// 導覽列是否半透明
self.navigationController?.navigationBar.translucent = false

// 導覽列右邊按鈕
let rightButton = UIBarButtonItem(
    barButtonSystemItem: .Edit,
    target: self,
    action: #selector(ArticleViewController.edit))
// 加到導覽列中
self.navigationItem.rightBarButtonItem = rightButton

// 建立一個按鈕
let myButton = UIButton(frame: CGRect(
    x: 100, y: 100, width: 120, height: 40))
myButton.setTitle("回前頁", forState: .Normal)
myButton.backgroundColor = UIColor.blueColor()
myButton.addTarget(
    self,
    action: #selector(ArticleViewController.back),
    forControlEvents: .TouchUpInside)
self.view.addSubview(myButton)
```

進到這頁你應該可以發現，導覽列左邊按鈕已經預設為回前頁的按鈕，當然你也可以再重新設定左邊按鈕的功能。

而導覽列右邊按鈕，這邊是第三種方式，設置為系統內建樣式的按鈕，參數 `barButtonItem` 提供了很多預設的文字可以設定，使用這個方式的好處是，它會依照你系統預設的語系顯示文字，如果有設置多國語系功能的話，這個按鈕就不需要再設置不同語言的文字。

接著則在 `ArticleViewController` 中加上按鈕執行動作的方法：

```
func edit() {
    print("edit action")
}

func back() {
    self.navigationController?.popViewControllerAnimated(true)
}
```

上述程式可以看到，雖然導覽列左邊按鈕已經有預設回前頁的按鈕，但這邊仍然建立一個自定義的按鈕，用來示範如何手動設置回前頁，導覽控制器用來返回頁面的方法為 `popViewControllerAnimated()`，參數為是否要有過場動畫。

SettingViewController.swift

最後看到 `SettingViewController.swift` 的 `viewDidLoad()`：

```
// 底色
self.view.backgroundColor = UIColor.blackColor()

// 導覽列標題
self.title = "Setting"

// 導覽列底色
self.navigationController?.navigationBar.barTintColor =
UIColor.whiteColor()

// 導覽列是否半透明
self.navigationController?.navigationBar.translucent = false

// 導覽列右邊 UIView
let myUIView = UIView(frame: CGRect(
    x: 0, y: 0, width: 30, height: 30))
myUIView.backgroundColor = UIColor.purpleColor()
let rightButton = UIBarButtonItem(customView: myUIView)
// 加到導覽列中
self.navigationItem.rightBarButtonItem = rightButton

// 建立一個按鈕
let myButton = UIButton(frame: CGRect(
    x: 100, y: 250, width: 120, height: 40))
myButton.setTitle("回前頁", forState: .Normal)
myButton.backgroundColor = UIColor.blueColor()
myButton.addTarget(
    self,
    action: #selector(SettingViewController.back),
    forControlEvents: .TouchUpInside)
self.view.addSubview(myButton)
```

上述程式可以看到最後一種建立 UIBarButtonItem 的方式，如果前面三種方式都不合你意的話，你可以設置一個自定義的 UIView 來代替按鈕，所以實際上你要擺什麼視圖在這上面都行(大部分元件都是繼承自 UIView)。

以上便為這節範例的內容。

圖片來源

- https://www.iconfinder.com/icons/510851/affirmative_check_mark_success_yes_icon

範例

本節範例程式碼放在 [uikit/uinavigationcontroller](#)

標籤列控制器 **UITabBarController**

前面兩節分別介紹了在多個頁面間切換以及導覽控制器 **UINavigationController**，這節要介紹另一個也是相當常見的元件：標籤列控制器 **UITabBarController**。

標籤列控制器與 **UINavigationController** 類似，也像是個容器，可以用來放置多個頁面，不同的地方在於，它會將可以前往的頁面以標籤列的方式列出，像是內建的健康 **App** 就是一個例子，如下：



建立 UITabBarController

這個範例的目標如下，有四個頁面可供切換，皆列在標籤列中：



首先在 Xcode 裡，新建一個 **Single View Application** 類型的專案，取名為 `ExUITabBarController`。

接著先以新增檔案的方式加入三個繼承自 `UIViewController` 的 `.swift` 檔案，分別命名

為 `ArticleViewController`、`IntroViewController` 及 `SettingViewController`。以及以加入檔案的方式加入四張按鈕的圖片。

AppDelegate.swift

一開始先依據[手動建立頁面](#)的步驟移除 Storyboard 檔案與相關設定，接著在 AppDelegate.swift 中將根視圖控制器設為一個 UITabBarController，如下：

```
func application(application: UIApplication,
didFinishLaunchingWithOptions launchOptions:
[NSObject: AnyObject]?) -> Bool {
    // 建立一個 UIWindow
    self.window = UIWindow(frame:
        UIScreen.mainScreen().bounds)

    // 設置底色
    self.window!.backgroundColor = UIColor.whiteColor()

    // 建立 UITabBarController
    let tabBar = UITabBarController()

    // 設置標籤列
    // 使用 UITabBarController 的屬性 tabBar 的各個屬性設置
    tabBar.tabBar.backgroundColor = UIColor.clearColor()

    // 建立頁面 使用系統圖示
    let mainViewController = ViewController()
    mainViewController.tabBarItem =
        UITabBarItem(tabBarSystemItem: .Favorites, tag: 100)

    // 建立頁面 使用自定義圖示 有預設圖片及按下時圖片
    let articleViewController = ArticleViewController()
    articleViewController.tabBarItem = UITabBarItem(
        title: "文章",
        image: UIImage(named: "article"),
        selectedImage: UIImage(named: "articleSelected"))

    // 建立頁面 使用自定義圖示 只有預設圖片
    let introViewController = IntroViewController()
    introViewController.tabBarItem = UITabBarItem(
        title: "介紹",
        image: UIImage(named: "profile"),
        tag: 200)
```

```
// 建立頁面 使用自定義圖示 可使用 tabBarItem 的屬性各自設定
let settingViewController = SettingViewController()
settingViewController.tabBarItem.image =
    UIImage(named: "setting")
settingViewController.tabBarItem.title = "設定"

// 加入到 UITabBarController
myTabBar.viewControllers = [
    mainViewController, articleViewController,
    introViewController, settingViewController]

// 預設開啓的頁面 (從 0 開始算起)
myTabBar.selectedIndex = 2

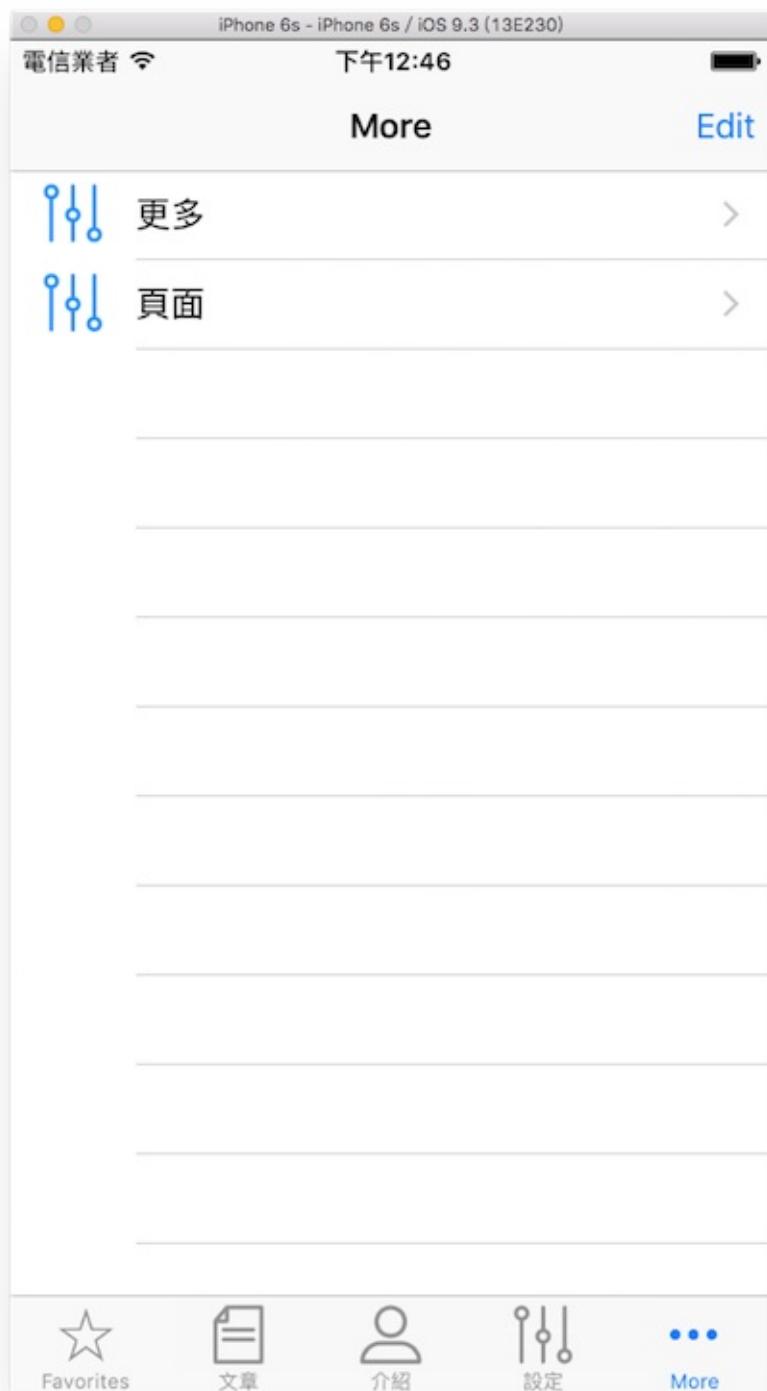
// 設置根視圖控制器
self.window!.rootViewController = myTabBar

// 將 UIWindow 設置為可見的
self.window!.makeKeyAndVisible()

return true
}
```

上述程式可以看到，如果要設置標籤列的樣式，是使用 `UITabBarController` 的 `tabBar` 屬性設置。而各頁面可以使用不同的方式設置標籤列樣式，除了可以使用系統內建的圖示外，也可以設置自定義圖示。

標籤列最多可以放五個圖示，超過的話，最右邊的會變成一個 `More` 的圖示，按下後會列出來後續可前往的頁面，如下：



各頁面都放置一個 `UILabel` 來代表不同內容，這邊便不再複述，完整程式碼請參考文末的範例程式碼。

以上即為本節範例的內容。

圖片來源

- http://www.flaticon.com/free-icon/file_118714
- http://www.flaticon.com/free-icon/speech-bubble_118712
- http://www.flaticon.com/free-icon/profile_118781
- http://www.flaticon.com/free-icon/settings_118769

範例

本節範例程式碼放在 [uikit/uitabbarcontroller](#)

手勢 **UIGestureRecognizer**

UIKit 提供了六種不同的手勢可供監聽，分別為 **Tap** 輕點、**Long Press** 長按、**Swipe** 滑動、**Pan** 拖曳、**Pinch** 縮放及**Rotation** 旋轉，你可以為元件加上這些手勢的監聽，並執行觸發時的動作。

其實在前面介紹的輸入多行文字 [UITextView](#)就已經有使用過了，當時是加上輕點手勢來關閉鍵盤，這節會完整的介紹支援的手勢。

這節會分開為兩個應用程式作為示範，第一個會以單純的 **UIView** 示範 **Tap** 輕點、**Long Press** 長按、**Swipe** 滑動及 **Pan** 拖曳，第二個則是搭配圖片示範 **Pinch** 縮放及 **Rotation** 旋轉

搭配 **UIView** 示範

這個範例的目標如下，輕點(**Tap**)、長按(**Long Press**)與滑動(**Swipe**)是加在基底視圖 `self.view` 上，輕點與長按會印出文字，而滑動時會控制藍色方塊上下左右移動。拖曳(**Pan**)則是加在橘色方塊上，可以將這個方塊拖曳到要移動的位置：



首先在 Xcode 裡，新建一個 **Single View Application** 類型的專案，取名為 ExUIGestureRecognizer 。

一開始先為 `ViewController` 建立三個屬性：

```
class ViewController: UIViewController {  
    var fullSize :CGSize!  
    var myUIView :UIView!  
    var anotherUIView :UIView!  
  
    // 省略  
}
```

以及在 `viewDidLoad()` 中取得螢幕尺寸，以供後續使用，如下：

```
// 取得螢幕的尺寸  
fullSize = UIScreen.mainScreen().bounds.size
```

Tap 輕點

Tap 輕點手勢可以支援單指與多指，請在實機上測試或是模擬機上按住 `option` 鍵，會出現兩個灰色圈圈，即可模擬兩指操作。以下先在 `viewDidLoad()` 加入程式，示範單指與兩指的手勢：

```
// 雙指輕點 (雙指以上手勢只能用實機測試)
let doubleFingers =
    UITapGestureRecognizer(
        target:self,
        action:#selector(ViewController.doubleTap(_:)))

// 點幾下才觸發 設置 1 時 則是點一下會觸發 依此類推
doubleFingers.numberOfTapsRequired = 1

// 幾根指頭觸發
doubleFingers.numberOfTouchesRequired = 2

// 為視圖加入監聽手勢
self.view.addGestureRecognizer(doubleFingers)

// 單指輕點
let singleFinger = UITapGestureRecognizer(
    target:self,
    action:#selector(ViewController.singleTap(_:)))

// 點幾下才觸發 設置 2 時 則是要點兩下才會觸發 依此類推
singleFinger.numberOfTapsRequired = 2

// 幾根指頭觸發
singleFinger.numberOfTouchesRequired = 1

// 雙指輕點沒有觸發時 才會檢測此手勢 以免手勢被蓋過
singleFinger.requireGestureRecognizerToFail(doubleFingers)

// 為視圖加入監聽手勢
self.view.addGestureRecognizer(singleFinger)
```

接著則是在 `ViewController` 加上觸發手勢後執行動作的方法：

```
// 觸發單指輕點兩下手勢後 執行的動作
func singleTap(recognizer:UITapGestureRecognizer){
    print("單指連點兩下時觸發")

    // 取得每指的位置
    self.findFingersPositon(recognizer)
}

// 觸發雙指輕點一下手勢後 執行的動作
func doubleTap(recognizer:UITapGestureRecognizer){
    print("雙指點一下時觸發")

    // 取得每指的位置
    self.findFingersPositon(recognizer)
}

func findFingersPositon(recognizer:UITapGestureRecognizer) {
    // 取得每指的位置
    let number = recognizer.numberOfTouches()
    for i in 0..
```

上述程式可以看到，還可以獲得每指輕點時的位置(是一個點 CGPoint)。

Long Press 長按

以下先在 `viewDidLoad()` 加入程式：

```
// 長按
let longPress = UILongPressGestureRecognizer(
    target: self,
    action: #selector(ViewController.longPress(_:)))
// 為視圖加入監聽手勢
self.view.addGestureRecognizer(longPress)
```

接著則是在 `ViewController` 加上觸發手勢後執行動作的方法：

```
// 觸發長按手勢後 執行的動作
func longPress(recognizer:UILongPressGestureRecognizer) {
    if recognizer.state == .Began {
        print("長按開始")
    } else if recognizer.state == .Ended {
        print("長按結束")
    }
}
```

Swipe 滑動

以下先在 `viewDidLoad()` 加入程式，因為每個方向滑動都算獨立的手勢，所以這邊會加上四個滑動手勢，分別為是上、左、下、右：

```
// 一個可供移動的 UIView
myUIView = UIView(frame: CGRect(
    x: 0, y: 0, width: 100, height: 100))
myUIView.backgroundColor = UIColor.blueColor()
self.view.addSubview(myUIView)

// 向上滑動
let swipeUp = UISwipeGestureRecognizer(
    target:self,
    action:#selector(ViewController.swipe(_:)))
swipeUp.direction = .Up

// 幾根指頭觸發 預設為 1
```

```
swipeUp.numberOfTouchesRequired = 1

// 為視圖加入監聽手勢
self.view.addGestureRecognizer(swipeUp)

// 向左滑動
let swipeLeft = UISwipeGestureRecognizer(
    target:self,
    action:#selector(ViewController.swipe(_:)))
swipeLeft.direction = .Left

// 為視圖加入監聽手勢
self.view.addGestureRecognizer(swipeLeft)

// 向下滑動
let swipeDown = UISwipeGestureRecognizer(
    target:self,
    action:#selector(ViewController.swipe(_:)))
swipeDown.direction = .Down

// 為視圖加入監聽手勢
self.view.addGestureRecognizer(swipeDown)

// 向右滑動
let swipeRight = UISwipeGestureRecognizer(
    target:self,
    action:#selector(ViewController.swipe(_:)))
swipeRight.direction = .Right

// 為視圖加入監聽手勢
self.view.addGestureRecognizer(swipeRight)
```

接著則是在 `ViewController` 加上觸發手勢後執行動作的方法：

```
// 觸發滑動手勢後 執行的動作
func swipe(recognizer:UISwipeGestureRecognizer) {
```

```
let point = myUIView.center

if recognizer.direction == .Up {
    print("Go Up")
    if point.y >= 150 {
        myUIView.center = CGPoint(
            x: myUIView.center.x,
            y: myUIView.center.y - 100)
    } else {
        myUIView.center = CGPoint(
            x: myUIView.center.x, y: 50)
    }
} else if recognizer.direction == .Left {
    print("Go Left")
    if point.x >= 150 {
        myUIView.center = CGPoint(
            x: myUIView.center.x - 100,
            y: myUIView.center.y)
    } else {
        myUIView.center = CGPoint(
            x: 50, y: myUIView.center.y)
    }
} else if recognizer.direction == .Down {
    print("Go Down")
    if point.y <= fullSize.height - 150 {
        myUIView.center = CGPoint(
            x: myUIView.center.x,
            y: myUIView.center.y + 100)
    } else {
        myUIView.center = CGPoint(
            x: myUIView.center.x,
            y: fullSize.height - 50)
    }
} else if recognizer.direction == .Right {
    print("Go Right")
    if point.x <= fullSize.width - 150 {
        myUIView.center = CGPoint(
            x: myUIView.center.x + 100,
            y: myUIView.center.y)
    } else {
```

```

        myUIView.center = CGPointMake(
            x: fullSize.width - 50,
            y: myUIView.center.y)
    }
}
}

```

上述程式將觸發四個方向滑動的手勢寫在同一個方法裡，並用屬性 `direction` 來辨別是哪一個方向，這邊會判斷是否會超出畫面，如果超出時就會改成移動到邊界，而不會真的超出去。

請記得手勢是加在 `self.view` 上，所以整個畫面上都可以滑動，而不是只點擊藍色方塊滑動。

Pan 拖曳

以下先在 `viewDidLoad()` 加入程式：

```

// 一個可供移動的 UIView
anotherUIView = UIView(frame: CGRectMake(
    x: fullSize.width * 0.5, y: fullSize.height * 0.5,
    width: 100, height: 100))
anotherUIView.backgroundColor = UIColor.orangeColor()
self.view.addSubview(anotherUIView)

// 拖曳手勢
let pan = UIPanGestureRecognizer(
    target:self,
    action:#selector(ViewController.pan(_:)))

// 最少可以用幾指拖曳
pan.minimumNumberOfTouches = 1

// 最多可以用幾指拖曳
pan.maximumNumberOfTouches = 1

// 為這個可移動的 UIView 加上監聽手勢
anotherUIView.addGestureRecognizer(pan)

```

接著則是在 `ViewController` 加上觸發手勢後執行動作的方法：

```
// 觸發拖曳手勢後 執行的動作
func pan(recognizer: UIPanGestureRecognizer) {
    // 設置 UIView 新的位置
    let point = recognizer.locationInView(self.view)
    anotherUIView.center = point
}
```

手勢是加在橘色方塊上，所以只能拖曳橘色方塊來移動。

手勢的傳遞

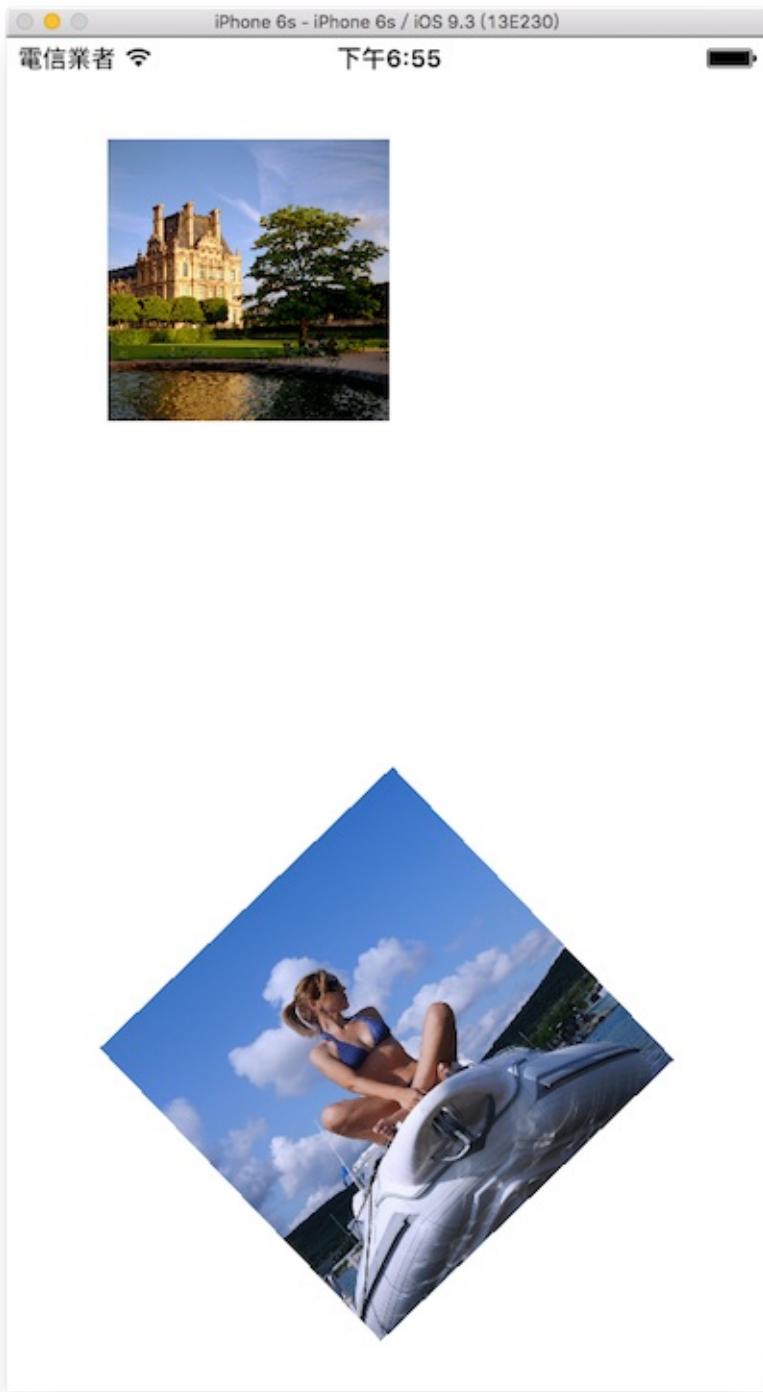
上面四個手勢都加上去後，你可能會不經意發現，在藍色方塊上雙指輕點或是單指輕點兩下，仍然會觸發到輕點的手勢，這是因為監聽手勢會傳遞。

當一個元件接收到手勢動作時，如果這個元件本身沒有可以執行的動作，那它會將手勢傳遞到它的父視圖，如果還是沒有，就會一直往上傳遞，以這邊來說，就是藍色方塊 `myUIView` 將手勢動作傳遞給父視圖 `self.view`，所以仍然會觸發到輕點手勢。

以上就是這個範例的內容。

搭配圖片示範

這個範例的目標如下，一張圖片可以縮放，另一張圖片可以旋轉：



首先在 Xcode 裡，新建一個 **Single View Application** 類型的專案，取名為 `ExImageGestureRecognizer`。

一開始先以加入檔案的方式加入兩張用來縮放與旋轉的圖片。接著為 `ViewController` 建立三個屬性：

```
class ViewController: UIViewController {  
    var fullSize :CGSize!  
    var myImageView :UIImageView!  
    var anotherImageView :UIImageView!  
  
    // 省略  
}
```

以及在 `viewDidLoad()` 中取得螢幕尺寸，以供後續使用，如下：

```
// 取得螢幕的尺寸  
fullSize = UIScreen.mainScreen().bounds.size
```

這個範例用到的縮放及旋轉手勢，都需要兩指操作，請在實機上測試或是模擬機上按住 `option` 鍵，會出現兩個灰色圈圈，即可模擬兩指操作。

Pinch 縮放

以下先在 `viewDidLoad()` 加入程式：

```
// 建立一個用來縮放的圖片  
myImageView = UIImageView(image: UIImage(named: "01.jpg"))  
myImageView.frame = CGRect(  
    x: 50, y: 50, width: 200, height: 200)  
self.view.addSubview(myImageView)  
  
let pinch = UIPinchGestureRecognizer(  
    target:self,  
    action:#selector(ViewController.pinch(_:)))  
  
self.view.addGestureRecognizer(pinch)
```

接著則是在 `ViewController` 加上觸發手勢後執行動作的方法：

```
// 觸發縮放手勢後 執行的動作
func pinch(recognizer: UIPinchGestureRecognizer) {
    if recognizer.state == .Began {
        print("開始縮放")
    } else if recognizer.state == .Changed {
        // 圖片原尺寸
        let frm = myImageView.frame

        // 縮放比例
        let scale = recognizer.scale

        // 目前圖片寬度
        let w = frm.width

        // 目前圖片高度
        let h = frm.height

        // 縮放比例的限制為 0.5 ~ 2 倍
        if w * scale > 100 && w * scale < 400 {
            myImageView.frame = CGRect(
                x: frm.origin.x, y: frm.origin.y,
                width: w * scale, height: h * scale)
        }
    } else if recognizer.state == .Ended {
        print("結束縮放")
    }
}
```

Rotation 旋轉

以下先在 `viewDidLoad()` 加入程式：

```
// 建立一個用來旋轉的圖片
anotherImageView = UIImageView(
    image: UIImage(named: "02.jpg"))
anotherImageView.frame = CGRect(
    x: 0, y: 0, width: 200, height: 200)
anotherImageView.center = CGPoint(
    x: fullSize.width * 0.5, y: fullSize.height * 0.75)
self.view.addSubview(anotherImageView)

let rotation = UIRotationGestureRecognizer(
    target: self,
    action: #selector(ViewController.rotation(_:)))
self.view.addGestureRecognizer(rotation)
```

接著則是在 `ViewController` 加上觸發手勢後執行動作的方法：

```
// 觸發旋轉手勢後 執行的動作
func rotation(recognizer:UIRotationGestureRecognizer) {
    // 弧度
    let radian = recognizer.rotation

    // 旋轉的弧度轉換為角度
    let angle = radian * (180 / CGFloat(M_PI))

    anotherImageView.transform =
        CGAffineTransformMakeRotation(radian)

    print("旋轉角度： \(angle)")
}
```

以上就是這個範例的內容。

圖片來源

- <https://www.flickr.com/photos/boklm/11908643634/>
- <https://www.flickr.com/photos/53812099@N04/11844277903/>

範例

本節範例程式碼放在 [uikit/uigesturerecognizer](#)

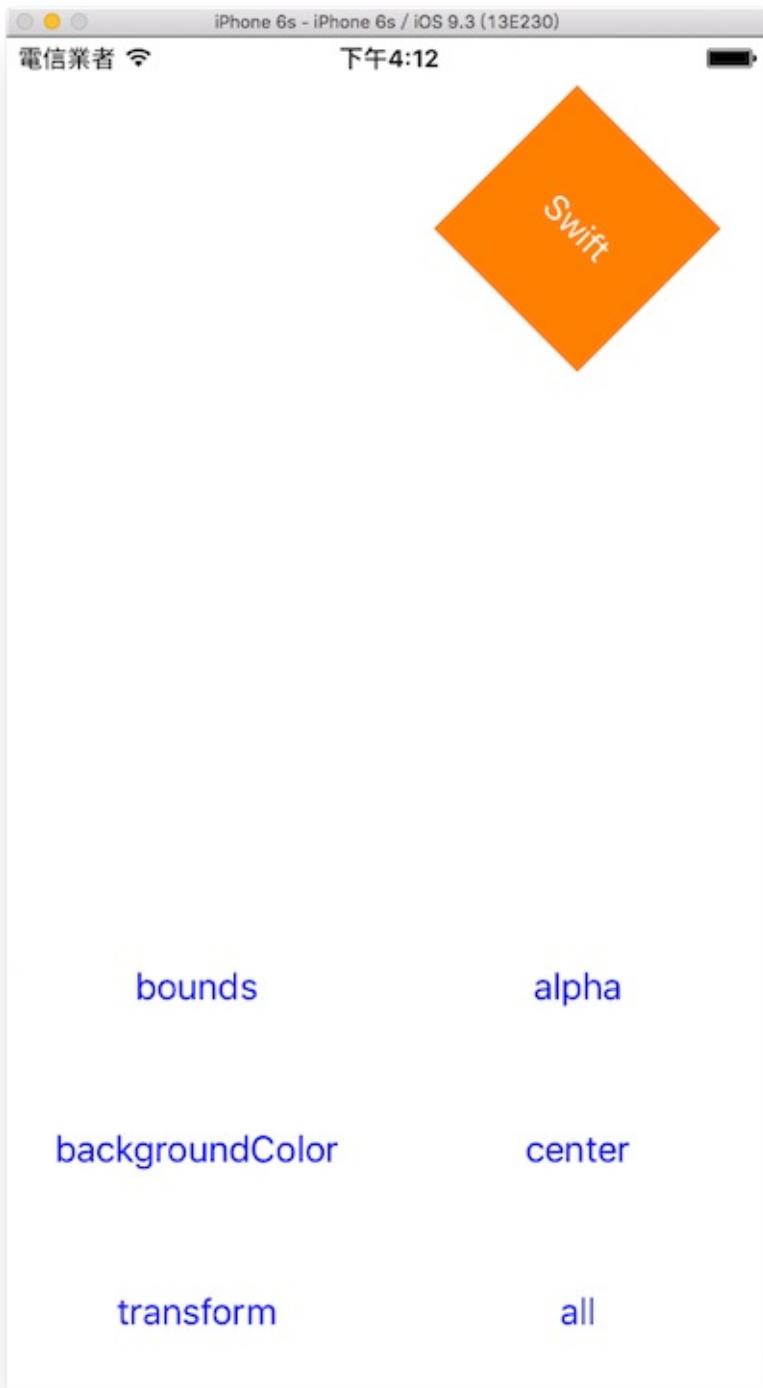
簡單動畫 Animations

UIKit 提供了一些簡單動畫的功能，這邊說的動畫指的是將視圖(UIView)的屬性以漸進的方式改變，以達成動畫的目的，視圖可供改變的屬性如下：

- `frame`：視圖的位置與尺寸。
- `bounds`：視圖的尺寸。
- `center`：視圖的位置。
- `alpha`：視圖的透明度。
- `backgroundColor`：視圖的背景顏色。
- `transform`：平移、縮放或旋轉視圖。
- `contentStretch`：拉伸一部分的視圖。

利用這些動畫，可以讓介面變得更為彈性。

本節範例會示範其中的 `bounds` 、 `center` 、 `alpha` 、 `backgroundColor` 與 `transform` ，利用點擊按鈕來執行動畫效果，五個按鈕各別負責上述其中一個動畫功能，以及一個 `all` 按鈕來執行全部的動畫，如下：



首先在 Xcode 裡，新建一個 **Single View Application** 類型的專案，取名為 ExAnimations。

一開始先為 `ViewController` 建立十二個屬性與設置預設值，以供後續使用，如下：

```
class ViewController: UIViewController {
```

```

var fullSize: CGSize!
var myLabel: UILabel!

let arrBounds = [
    CGSizeMake(100, 100), CGSizeMake(50, 50),
    CGSizeMake(150, 150), CGSizeMake(50, 50)]
var arrCenter :[CGPoint]!
let arrAlpha = [0.25, 0.75, 0.5, 1.0]
let arrBackgroundColor = [
    UIColor.cyanColor(), UIColor.greenColor(),
    UIColor.orangeColor(), UIColor.blackColor()]
let arrTransform =
[CGAffineTransformMakeRotation(CGFloat(M_PI * 0.25)),
CGAffineTransformMakeRotation(CGFloat(M_PI * 1.25)),
CGAffineTransformMakeRotation(CGFloat(M_PI * 1.75)),
CGAffineTransformMakeRotation(CGFloat(M_PI * 2))]

var indexBounds = 0
var indexCenter = 0
var indexAlpha = 0
var indexBackgroundColor = 0
var indexTransform = 0

override func viewDidLoad() {
    super.viewDidLoad()

    // 取得螢幕的尺寸
    fullSize = UIScreen.mainScreen().bounds.size

    arrCenter = [
        CGPointMake(x: fullSize.width * 0.75,
                    y: fullSize.width * 0.25),
        CGPointMake(x: fullSize.width * 0.75,
                    y: fullSize.width * 0.75),
        CGPointMake(x: fullSize.width * 0.25,
                    y: fullSize.width * 0.75),
        CGPointMake(x: fullSize.width * 0.25,
                    y: fullSize.width * 0.25)]


    // 建立 UIButton 與 UILabel

```

```
    self.createButtonsAndLabel()

}

// 省略
}
```

上述程式是用來示範動畫的值，每個動畫都設置四個循環變動的值，像是要變換位置的 `center`，就是一個型別為 `[CGPoint]` 的陣列，其內依序放置四個位置(`CGPoint`)，稍後就可以點擊 `center` 按鈕來切換位置，其他動畫都是類似的方式。

建立 `UIButton` 與 `UILabel` 的部份寫在 `self.createButtonsAndLabel()` 中，因為與之前範例相似，就省略這個部分，如有需要詳細內容請參考節末的完整範例程式碼。

執行動畫

常用來執行動畫的方法有四種：

- `UIView.animateWithDuration(_:, animations:)`
- `UIView.animateWithDuration(_:, animations:, completion:)`
- `UIView.animateWithDuration(_:, delay:, options:, animations:, completion:)`
- `UIView.animateWithDuration(_:, delay:, usingSpringWithDamping:, initialSpringVelocity:, options:, animations:, completion:)`

以下會依序利用不同的屬性動畫介紹。

bounds

`bounds` 可以改變視圖的尺寸，按下 `bounds` 按鈕後執行動作的方法如下：

```

func AnimateBounds() {
    let newSize = self.arrBounds[self.indexBounds]
    let originCenter = self.myLabel.center
    UIView.animateWithDuration(
        0.5,
        animations: {
            self.myLabel.bounds = CGRect(
                x: 0, y: 0,
                width: newSize.width, height: newSize.height)
            self.myLabel.center = originCenter
        })
    self.updateIndex("bounds")
}

```

上述程式中，先取得目前要變成的尺寸 `newSize`，以及目前的位置 `originCenter`，接著用到第一個動畫方法：

```
UIView.animateWithDuration(_:, animations:)
```

有兩個參數：

- 第一個參數為整個動畫執行的時間，單位為秒。
- 第二個參數則是一個閉包(`closure`)，放置要達成的動畫結果，這邊將尺寸 `self.myLabel.bounds` 設置為前面取得的 `newSize`，動畫的過程便會將視圖在執行時間內，漸進地變大(或變小)尺寸。因為改變尺寸是以原點為準，所以 `center` 會變，這邊再把 `center` 設置為原本的值 `originCenter`。

最後使用 `self.updateIndex()` 方法來讓示範的值可以循環使用，內容請參考節末的完整範例程式碼，稍後的範例就不再複述。

alpha

`alpha` 可以改變視圖的透明度，按下 `alpha` 按鈕後執行動作的方法如下：

```

func AnimateAlpha() {
    UIView.animateWithDuration(0.5, animations: {
        self.myLabel.alpha =
            CGFloat(self.arrAlpha[self.indexAlpha])
    }, completion: { _ in
        print("Animation Alpha Complete")
    })

    self.updateIndex("alpha")
}

```

這邊看到第二個動畫方法：

```
UIView.animateWithDuration(_:, animations:, completion:)
```

與先前的比較多了第三個參數 `completion`，同樣也是個閉包(`closure`)，在動畫完成後執行，在這裡你可以再接續下一個動畫，或是其他整理、清除的動作。

backgroundColor

`backgroundColor` 可以改變視圖的背景顏色，按下 `backgroundColor` 按鈕後執行動作的方法如下：

```

func AnimateBackgroundColor() {
    UIView.animateWithDuration(
        1,
        delay: 0.2,
        options: .CurveEaseIn,
        animations: {
            self.myLabel.backgroundColor =
                self.arrBackgroundColor[self.indexBackgroundColor]
        }, completion: { _ in
            print("Animation BackgroundColor Complete")
    })

    self.updateIndex("backgroundColor")
}

```

這邊看到第三個動畫方法：

```
UIView.animateWithDuration(_:, delay:, options:, animations:, completion:)
```

與先前的比較多了兩個參數，`delay` 是設置為要延遲開始的時間，單位為秒，當你有多個動畫要一起執行時，如果想要有不同時間開始執行的效果可以設置。另一個參數 `options` 可以設置如何執行動畫的方式。(依據改變的視圖屬性不同，可能有些會沒有效果。)

center

`center` 可以改變視圖的位置，按下 `center` 按鈕後執行動作的方法如下：

```
func AnimateCenter() {
    UIView.animateWithDuration(
        1.5,
        delay: 0.1,
        usingSpringWithDamping: 0.4,
        initialSpringVelocity: 0,
        options: .CurveEaseInOut,
        animations: {
            self.myLabel.center =
                self.arrCenter[self.indexCenter]
        }, completion: { _ in
            print("Animation Center Complete")
        })
    self.updateIndex("center")
}
```

這邊看到第四個動畫方法：

```
UIView.animateWithDuration(_:, delay:, usingSpringWithDamping:, initialSpringVelocity:, options:, animations:, completion:)
```

又較先前的多了兩個參數，這個方法可以讓你的動畫更為生動活潑：

- `usingSpringWithDamping` 是設置視圖的回彈量，可設置 `0 ~ 1`，越接近 `1` 則彈越少下。
- `initialSpringVelocity` 則是設置初始速度，數字越大則會越快，如果要平滑變化的話可以設置為 `0`。

transform

transform 可以平移、縮放或旋轉視圖，這邊示範旋轉視圖，按下 transform 按鈕後執行動作的方法如下：

```
func AnimateTransform() {
    UIView.animateWithDuration(0.5, animations: {
        self.myLabel.transform =
            self.arrTransform[self.indexTransform]
    })
    self.updateIndex("transform")
}
```

最後則是 all 按鈕，可以一次執行所有的動畫，請參考節末的完整範例程式碼。

以上即為本節範例的內容。

範例

本節範例程式碼放在 [uikit/animations](#)

儲存資訊 **UserDefaults**

iOS 系統提供儲存資訊的方式有很多種，最為簡單的就是 **UserDefaults**，像是儲存使用者的開啓次數、使用時間或是有沒有使用過了哪些功能等等，這些少量資訊就可以使用 **UserDefaults** 來儲存。(大量資訊就不建議使用 **UserDefaults** 儲存)

實際上 **UserDefaults** 是屬於 Foundation Framework 的功能(**NS** 開頭的函式像是前面提過的 **NSDate**、**NSTimer** 大多都是)，但因為這個功能實在太常見又方便了，幾乎每個應用程式都會使用，所以仍然分了一節作說明。

以下是本節範例目標，設置一個輸入框來儲存一段文字，可以按下按鈕儲存及更新，以及另一個按鈕用來刪除，上方導覽列右邊的按鈕可以前往下一頁，會顯示出這個儲存的文字：



首先在 Xcode 裡，新建一個 **Single View Application** 類型的專案，取名為 **ExUserDefaults**。

因為有使用到導覽列，請依照前面介紹的導覽控制器 **UINavigationController**，建立一個有導覽列的應用程式。並以**新增檔案**的方式加入一個繼承自 **UIViewController** 的 **.swift** 檔案，命名為 **DisplayViewController**。

請依照之前學習過的內容，在 `ViewController` 的 `viewDidLoad()` 方法中，設置導覽列、輸入框以及按鈕，這邊便省略這個部分，如有需要詳細內容請參考節末的完整範例程式碼。

除此之外，為了這節範例所需，會為 `ViewController` 建立一個屬性 `myUserDefaults`：

```
class ViewController: UIViewController {  
    var myUserDefaults : UserDefaults!  
  
    // 省略  
}
```

儲存及更新資訊

在應用程式建立時，系統已經為 `NSUserDefaults` 建立了一些應用程式所需的資訊。這邊在 `viewDidLoad()` 取得資訊：

```
// 取得儲存的預設資料  
myUserDefaults = UserDefaults.standard
```

接著看到按下更新按鈕執行動作的方法：

```
func updateInfo() {  
    print("update info")  
  
    // 結束編輯 把鍵盤隱藏起來  
    self.view.endEditing(true)  
  
    myUserDefaults.setObject(  
        myTextField.text, forKey: "info")  
    myUserDefaults.synchronize()  
}
```

要儲存資訊是使用到 `NSUserDefaults` 的方法 `.setObject(_:forKey:)`，兩個參數分別為要儲存的資訊以及 **key** 值，有點像是字典的鍵值對 **key : value**。

如果尚未有這個 `key` 值資訊時，就會新增，如果已經有了的話，則是會更新它。

設置好新的值之後，有時系統不會即時更新儲存內容，如果你要確實讓它儲存起來時，就是使用 `UserDefaults` 的方法 `synchronize()`，這可以讓資訊確實儲存。

移除資訊

接著看到按下移除按鈕執行動作的方法：

```
func removeInfo() {
    print("remove info")

    myUserDefaults.removeObjectForKey("info")

    myTextField.text = ""
}
```

移除資訊則是使用到 `UserDefaults` 的方法 `removeObjectForKey()`，依照傳入的 `key` 值移除資訊。

存取資訊

最後看到下一頁 `DisplayViewController`，其內的 `viewDidLoad()` 中，建立一個 `UILabel` 來顯示資訊：

```
if let info = myUserDefaults.objectForKey("info") as? String {
    myLabel.text = info
} else {
    myLabel.text = "尚未儲存資訊"
    myLabel.textColor = UIColor.redColor()
}
```

上述程式省略其他部分，僅說明存取資訊，使用 `UserDefaults` 的方法 `objectForKey()` 來取得資訊，因為不只可以儲存單純文字，所以這邊會將它先轉為字串(`String`)再作後續使用。

以上即為這節範例的內容。

範例

本節範例程式碼放在 [uikit/nsuserdefaults](#)

資料庫

前面章節 [儲存資訊 UserDefaults](#) 已經提過，可以使用 `UserDefaults` 來儲存少量資料。而當需要儲存大量資料時，就需要使用到資料庫了，這章會分別介紹 `SQLite` 以及 `Core Data`。

`SQLite` 是一個輕量的資料庫，可以使用絕大部分的 `SQL` 指令，如果之前已有使用 `SQL` 資料庫的經驗，`SQLite` 其實可以很快就上手(因為操作方式幾乎一樣)。

而 `Core Data` 的背後其實也是操作 `SQLite`，但將需要操作 `SQL` 的指令封裝起來，讓你可以更為快速的操作資料庫，如果先前沒有 `SQL` 資料庫的實作經驗，你也可以選擇使用 `Core Data` 來快速的建立並使用資料庫。

SQLite

iOS 系統支援很常見的資料庫 SQLite，這是一個輕量的關聯式資料庫管理系統(Relational Database Management System，縮寫為 RDBMS)，所有的資料內容其實就是一個檔案，而且絕大部分的 SQL 指令都可以使用。

以下會先介紹在應用程式中如何加入 SQLite，接著會介紹如何新增、讀取、更新與刪除資料，最後會將 SQLite 功能獨立寫在一個類別中，以便日後其他應用程式需要時可以重複使用。

本節內容已假設你已經會使用簡單的 SQL 指令，所以不會詳細介紹如何使用，如果尚未熟悉，請先了解相關內容再繼續本節的範例。

加入 SQLite

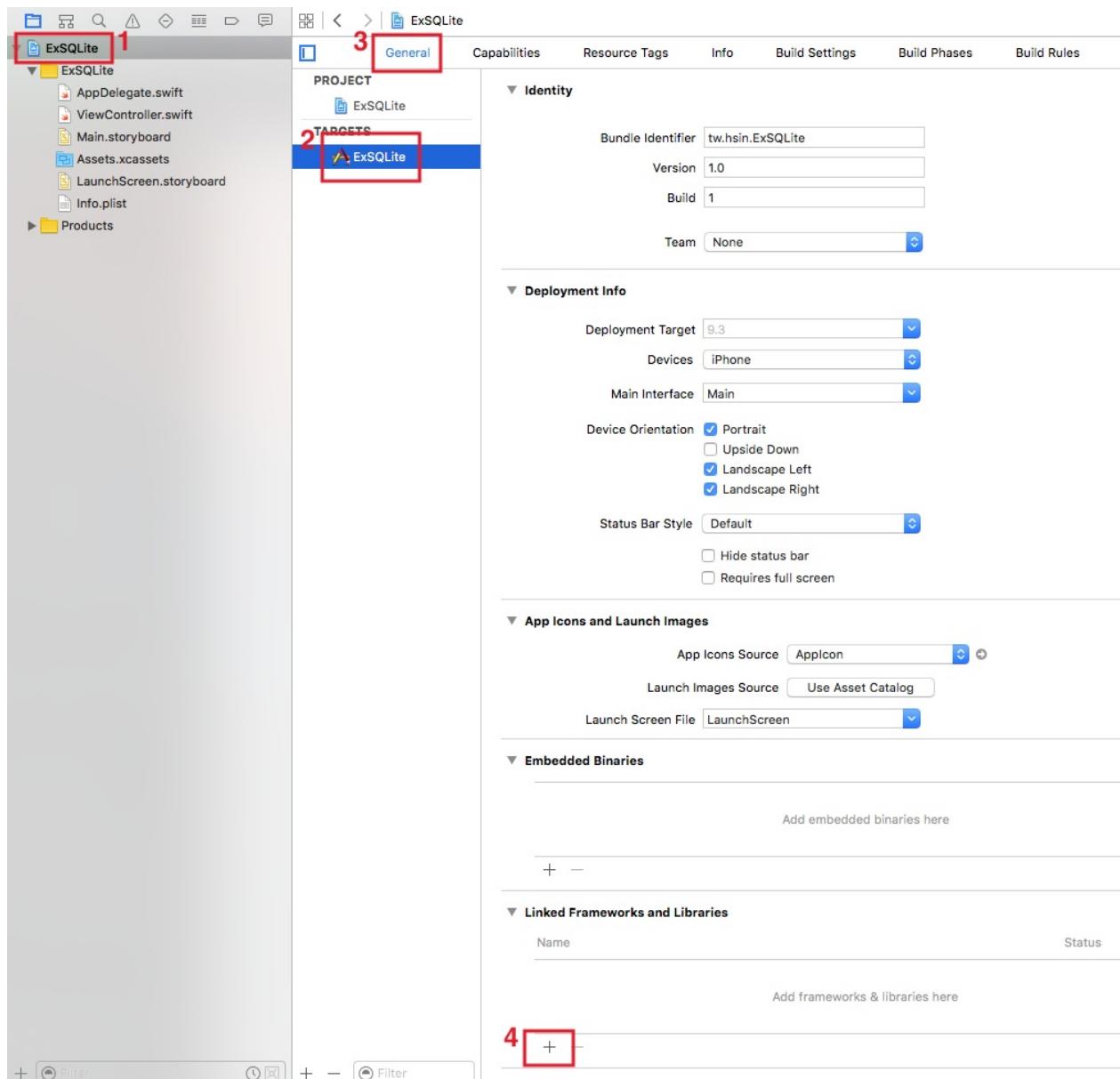
其實 Swift 本身是無法直接使用 SQLite 的功能，必須要利用 Objective-C (在 Swift 之前用來編寫 iOS 應用程式的程式語言)來與 SQLite 連結。

首先在 Xcode 裡，新建一個 **Single View Application** 類型的專案，取名為 ExSQLite 。

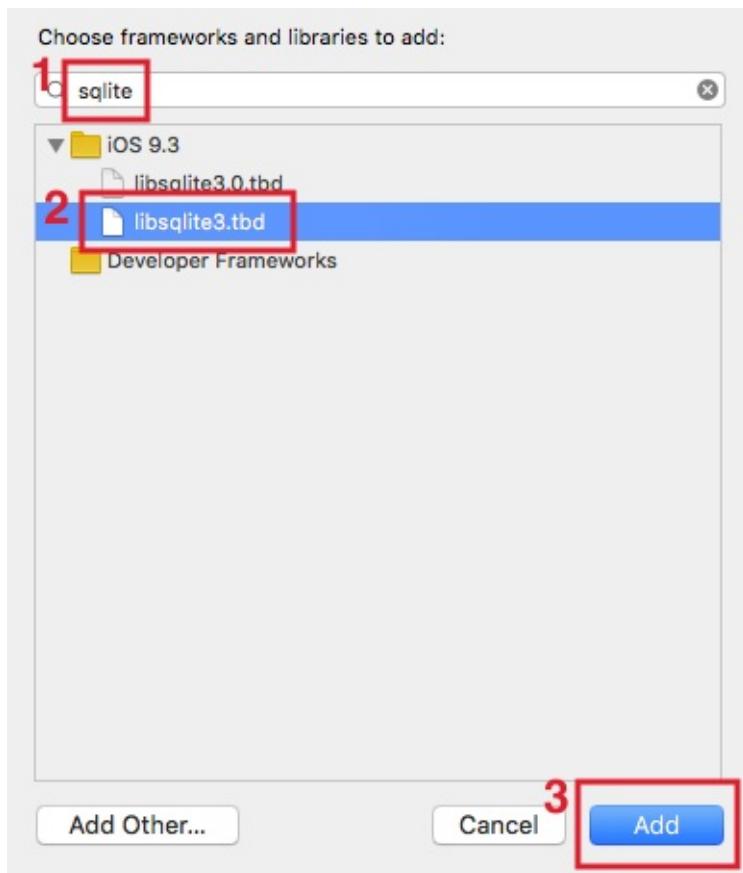
以下會介紹如何加入的步驟：

加入 **sqlite** 函式庫

▼ 預設的應用程式沒有引入 **sqlite** 函式庫，所以需要手動加入，先找到 TARGETS > ExSQLite > General > Linked Frameworks and Libraries ，並按下加號 + ，如下圖：



▼ 這邊會列出來所有可以加入的函式庫，所以填入 `sqlite` 來過濾，最後會出現兩個類似的函式庫，實際上兩個指的是一樣的東西，所以選一個加入就好，這邊選取 `libsqLite3.tbd`，按下 `Add` 加入：

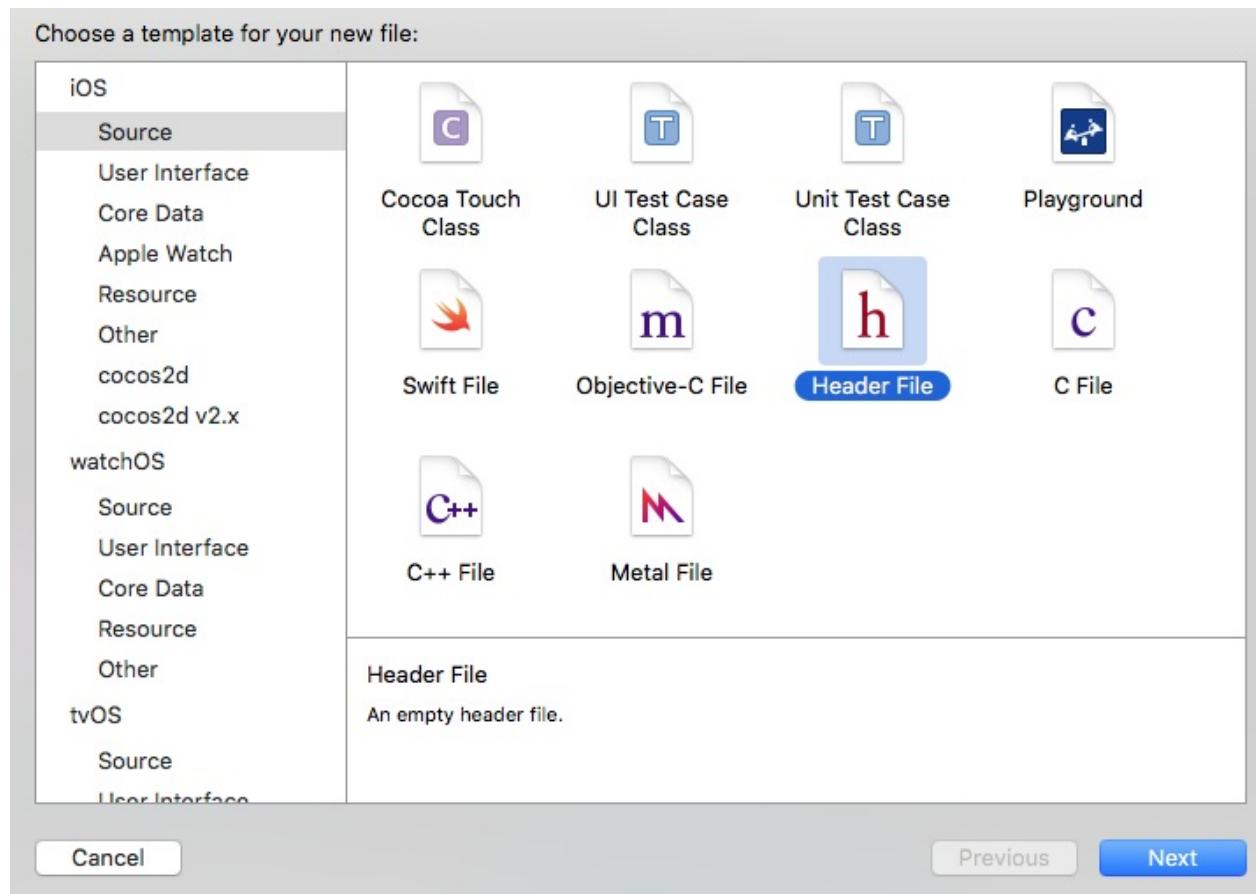


▼ 如果有順利加入的話，sqlite 函式庫就會出現在列表中，如下圖：

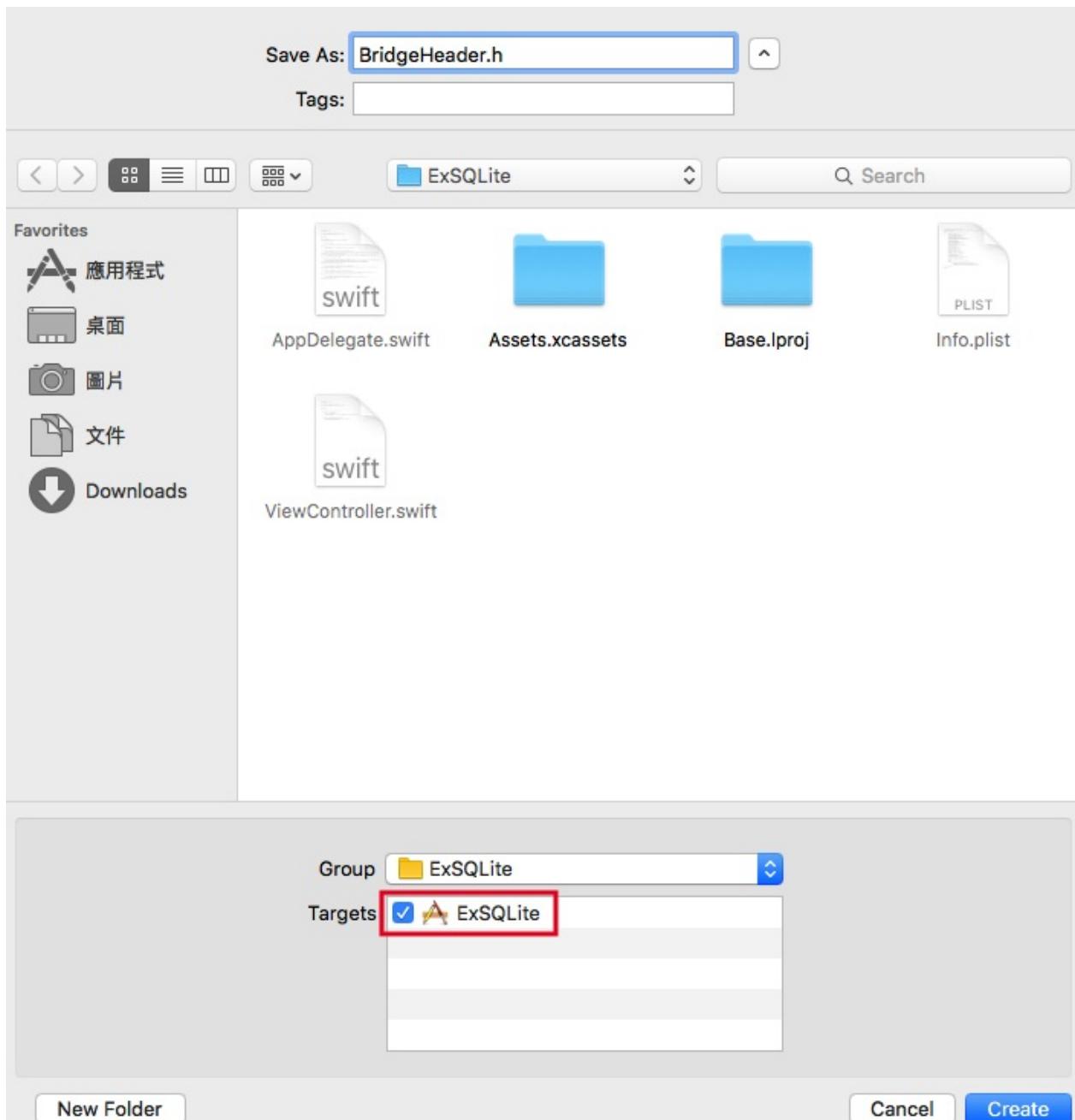
Linked Frameworks and Libraries	
Name	Status
libsdlite3.tbd	Required
+	-

新增 header 檔案

▼ 前面提到需要利用 Objective-C 來與 SQLite (也就是前一步驟加入的 sqlite 函式庫)連結，所以接著必須為應用程式加上一個 header 檔案以連結。先以[新增檔案](#)的方式加入，但請注意選擇檔案類型時，要選擇 `iOS > Source > Header File`，如下圖：



▼ 接著將檔案命名為 BridgeHeader.h (.h 就是一個 header 檔案)，請記得要勾選 Targets ，如下圖：



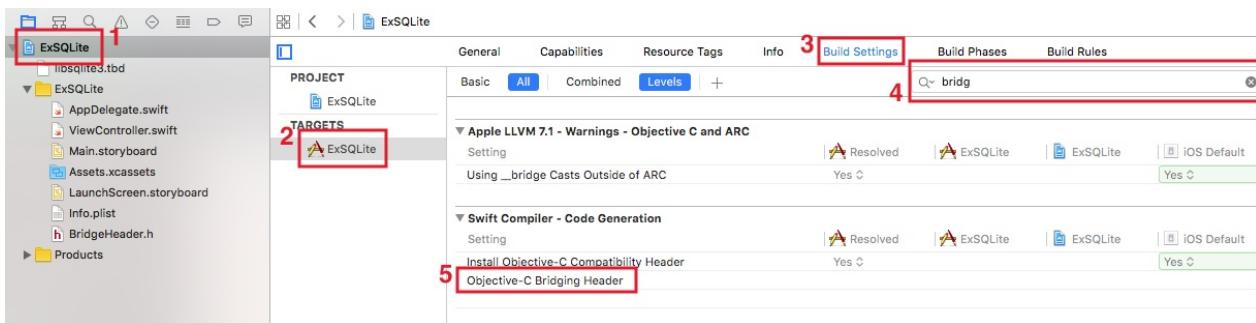
建立好 header 檔案後，請點開這隻檔案，並填入一行程式，如下：

```
#include "sqlite3.h"
```

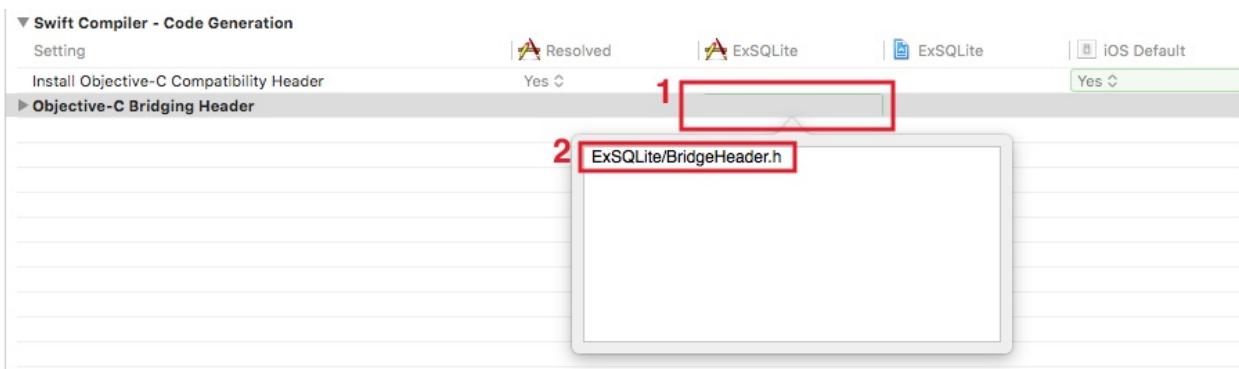
這隻 header 檔案就是用來引入 sqlite 函式庫，所以填寫這一行程式即可。

與 Objective-C 連結

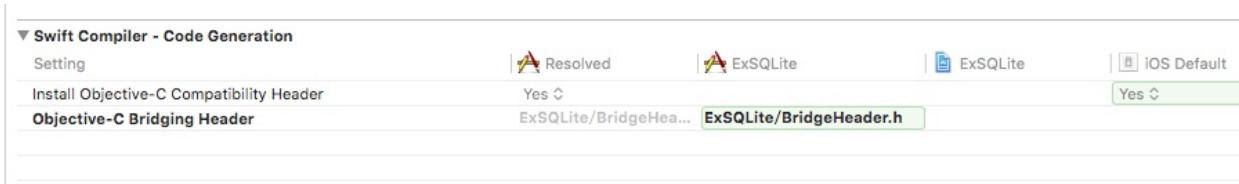
▼ 最後要將 Swift 與前一步驟設置好的 header 檔案連結起來，首先找到 TARGETS > ExSQLite > Build Settings > Objective-C Bridging Header，這邊有很多東西可以設定，所以你可以在右上角的搜尋框輸入 bridg 來過濾，如下圖：



▼ 接著對下圖標示 1 的地方點兩下，會彈出一個輸入框，填入 header 檔案的路徑與名稱(請記得路徑也要填)：



▼ 如果順利加入的話，會顯示如下圖：



這樣便完成了加入 SQLite 的步驟，你可以先試著 Build (cmd + b)專案，看看有沒有錯誤，如果有的話請檢查前面步驟有沒有做錯，如果沒有錯誤的話就是順利加入 SQLite 了。

使用 SQLite

接著進入到程式碼的部份，首先會需要宣告一個變數來儲存 SQLite 的連線資訊，型別為 `COpaquePointer`，後續的資料庫操作都會需要這個變數：

```
var db :COpaquePointer = nil
```

資料庫檔案路徑

前面有提過 SQLite 其實操作的是一個檔案，所以要先取得這個資料庫檔案的路徑：

```
// 資料庫檔案的路徑
let urls = FileManager.defaultManager()
    .URLsForDirectory(
        .DocumentDirectory,
        inDomains: .UserDomainMask)
let sqlitePath = urls[urls.count-1].absoluteString
    + "sqlite3.db"
```

上述程式會取得應用程式的 Documents 目錄，這是開放給開發者儲存檔案的路徑，有任何需要儲存的檔案都是放在這裡，而 `sqlite3.db` 則是這個資料庫檔案名稱，你也可以命名為 `db.sqlite` 之類，其他可供辨識的檔案名稱。

如果沒有這個檔案，系統會自動嘗試建立起來。

開啟資料庫連線

接著要與資料庫連線，會用到前面兩個步驟宣告的變數：

```
if sqlite3_open(sqlitePath, &db) == SQLITE_OK {
    print("資料庫連線成功")
} else {
    print("資料庫連線失敗")
}
```

使用 `sqlite3_open()` 函式來連線，請注意第二個參數前面必須加上 `&`，這是一個指標的概念(與前面章節提過的輸入輸出參數 In-Out Parameters 類似)，函式內使用的就是傳入參數 `db` 本身，所以稍後操作資料庫時可以直接使用這個 `db` 變數。

建立資料表

建立一個名為 **students** 的資料表，欄位分別為 `id`, `name`, `height`，欄位類型依序為 `integer`, `text`, `double`：

```

let sql = "create table if not exists students "
    + "( id integer primary key autoincrement, "
    + "name text, height double)" as NSString

if sqlite3_exec(db, sql.UTF8String, nil, nil, nil)
== SQLITE_OK{
    print("建立資料表成功")
}

```

使用 `sqlite3_exec()` 函式來建立資料表，第一個參數就是前面建立資料庫連線後的 `db`，第二個參數就是 SQL 指令，這邊會先轉成 `NSString` 型別，再將文字編碼轉成 UTF8。

如果返回為 `SQLITE_OK`，則表示建立成功。

Hint

- 如果要查看模擬器的 SQLite 檔案，可以使用桌機的 Firefox 瀏覽器的一個套件 [SQLite Manager](#)，它可以讓你檢視與編輯本機上的 SQLite 檔案。實際檔案路徑可藉由將前面提過的 `sqlitePath` 印出來取得。

新增資料

這邊需要另一個型別為 `COpaquePointer` 的變數 `statement`，用來取得操作資料庫後回傳的資訊：

```

var statement :COpaquePointer = nil
let sql = "insert into students "
    + "(name, height) "
    + "values ('小強', 178.2)" as NSString

if sqlite3_prepare_v2(
    db, sql.UTF8String, -1, &statement, nil) == SQLITE_OK{
    if sqlite3_step(statement) == SQLITE_DONE {
        print("新增資料成功")
    }
    sqlite3_finalize(statement)
}

```

新增資料是使用 `sqlite3_prepare_v2()` 函式，前面兩個參數與 `sqlite3_exec()` 使用的相同，第三個參數則是設定資料庫可以讀取的最大資料量，單位是位元組(Byte)，設為 -1 表示不限制讀取量，第四個參數是用來取得操作返回的資訊，記得參數前面要加上 & 。

`statement` 要再當做 `sqlite3_step()` 函式的參數傳入，如果返回 `SQLITE_DONE`，則是表示新增成功。

最後要使用 `sqlite3_finalize()` 函式來釋放掉 `statement`，以免發生記憶體洩漏的問題。

讀取資料

讀取資料方式如下：

```
var statement :COpaquePointer = nil
var sql = "select * from students"
sqlite3_prepare_v2(
    db, (sql as NSString).UTF8String, -1, &statement, nil)

while sqlite3_step(statement) == SQLITE_ROW{
    let id = sqlite3_column_int(statement, 0)
    let name = String.fromCString(
        UnsafePointer<CChar>(sqlite3_column_text(statement, 1)))
    let height = sqlite3_column_double(statement, 2)
    print("\(id). \(name!) 身高: \(height)")
}

sqlite3_finalize(statement)
```

讀取資料以及後續的更新、刪除資料都與新增資料使用一樣的函式 `sqlite3_prepare_v2()`，參數都是一樣的意思，也就不再複述，請看前面新增資料的說明。

回傳的資料存在變數 `statement` 中，以 `while` 迴圈來一筆一筆取出，當等於 `SQLITE_ROW` 時就是有資料，會一直取到不為 `SQLITE_ROW`，就會結束迴圈。(如果只有一筆資料的話，也可以使用 `if` 條件句即可。)

迴圈中每筆資料的每個欄位則是使用 `sqlite3_column_` 資料類型() 函式來取出，像是 `int` 的欄位就是使用 `sqlite3_column_int()`，`text` 的欄位就是使用 `sqlite3_column_text()`，`double` 的欄位就是使用 `sqlite3_column_double()`，以此類推。

取出欄位的函式有兩個參數，第一個都固定是返回資訊 `statement`，第二個則是這個欄位的索引值，範例有三個欄位：`id, name, height`，則索引值從 0 開始算起，依序為 0, 1, 2。

更新資料

更新資料方式如下：

```
var statement :C0paquePointer = nil
var sql = "update students set name='小強' where id = 1"

if sqlite3_prepare_v2(
    db, (sql as NSString).UTF8String, -1, &statement, nil)
== SQLITE_OK {
    if sqlite3_step(statement) == SQLITE_DONE {
        print("更新資料成功")
    }
    sqlite3_finalize(statement)
}
```

刪除資料

刪除資料方式如下：

```

var statement :COpaquePointer = nil
var sql = "delete from students where id = 3"

if sqlite3_prepare_v2(
    db, (sql as NSString).UTF8String, -1, &statement, nil)
== SQLITE_OK {
    if sqlite3_step(statement) == SQLITE_DONE {
        print("刪除資料成功")
    }
    sqlite3_finalize(statement)
}

```

以上便為基本操作資料庫的方式。

將 **SQLite** 功能獨立出來

這一小節會將前面講過的資料庫基本功能獨立出來寫成一個類別，以便日後其他應用程式需要時可以重複使用。

首先以[新增檔案](#)的方式加入一個 `.swift` 檔案，命名為 `SQLiteConnect.swift`，記得檔案類型要選擇 `Swift File`：

`iOS > Source > Swift File`

接著打開這隻檔案，建立一個類別，以及其內的方法：

```

class SQLiteConnect {

    var db :COpaquePointer = nil
    let sqlitePath :String

    init?(path :String) {
        sqlitePath = path
        db = self.openDatabase(sqlitePath)

        if db == nil {
            return nil
        }
    }
}

```

```

// 連結資料庫 connect database
func openDatabase(path :String) -> COpaquePointer {
    var connectdb: COpaquePointer = nil
    if sqlite3_open(path, &connectdb) == SQLITE_OK {
        print("Successfully opened database \(path)")
        return connectdb
    } else {
        print("Unable to open database.")
        return nil
    }
}

// 建立資料表 create table
func createTable(
    tableName :String, columnsInfo :[String]) -> Bool {
    let sql = "create table if not exists \(tableName) "
        + "(\(columnsInfo.joinWithSeparator(",")))"
        as NSString

    if sqlite3_exec(
        self.db, sql.UTF8String, nil, nil, nil) == SQLITE_OK{
        return true
    }

    return false
}

// 新增資料
func insert(
    tableName :String, rowInfo :[String:String]) -> Bool {
    var statement :COpaquePointer = nil
    let sql = "insert into \(tableName) "
        + "(\(rowInfo.keys.joinWithSeparator(","))) "
        + "values "
        + "(\(rowInfo.values.joinWithSeparator(",")))"
        as NSString

    if sqlite3_prepare_v2(
        self.db, sql.UTF8String, -1, &statement, nil)

```

```

        == SQLITE_OK {
            if sqlite3_step(statement) == SQLITE_DONE {
                return true
            }
            sqlite3_finalize(statement)
        }

        return false
    }

    // 讀取資料
    func fetch(
        tableName :String, cond :String?, order :String?)
        -> COpaquePointer {
        var statement :COpaquePointer = nil
        var sql = "select * from \(tableName)"
        if let condition = cond {
            sql += " where \(condition)"
        }

        if let orderBy = order {
            sql += " order by \(orderBy)"
        }

        sqlite3_prepare_v2(
            self.db, (sql as NSString).UTF8String, -1,
            &statement, nil)

        return statement
    }

    // 更新資料
    func update(
        tableName :String,
        cond :String?, rowInfo :[String:String]) -> Bool {
        var statement :COpaquePointer = nil
        var sql = "update \(tableName) set "

        // row info
        var info :[String] = []

```

```

        for (k, v) in rowInfo {
            info.append("\(k) = \(v)")
        }
        sql += info.joinWithSeparator(", ")

        // condition
        if let condition = cond {
            sql += " where \(condition)"
        }

        if sqlite3_prepare_v2(
            self.db, (sql as NSString).UTF8String, -1,
            &statement, nil) == SQLITE_OK {
            if sqlite3_step(statement) == SQLITE_DONE {
                return true
            }
            sqlite3_finalize(statement)
        }

        return false
    }

    // 刪除資料
    func delete(tableName :String, cond :String?) -> Bool {
        var statement :COpaquePointer = nil
        var sql = "delete from \(tableName)"

        // condition
        if let condition = cond {
            sql += " where \(condition)"
        }

        if sqlite3_prepare_v2(
            self.db, (sql as NSString).UTF8String, -1,
            &statement, nil) == SQLITE_OK {
            if sqlite3_step(statement) == SQLITE_DONE {
                return true
            }
            sqlite3_finalize(statement)
        }
    }
}

```

```

    }

    return false
}

}

```

上述程式可以看到，這邊為類別建立一個新的建構器(initializer)，而且是可失敗建構器(failable initializer)，是為了確保有正確連結上資料庫。

接著就可以使用這個類別來操作資料庫，將前一小節的基本操作內容改成如下，在 `viewDidLoad()` 裡：

```

// 資料庫檔案的路徑
let sqlitePath = NSHomeDirectory() + "/Documents/sqlite3.db"

// 印出儲存檔案的位置
print(sqlitePath)

// SQLite 資料庫
db = SQLiteConnect(path: sqlitePath)

if let mydb = db {

    // create table
    mydb.createTable("students", columnsInfo: [
        "id integer primary key autoincrement",
        "name text",
        "height double"])

    // insert
    mydb.insert(
        "students", rowInfo:
        ["name": "'大強'", "height": "178.2"])

    // select
    let statement = mydb.fetch(
        "students", cond: "1 == 1", order: nil)
    while sqlite3_step(statement) == SQLITE_ROW{

```

```
    let id = sqlite3_column_int(statement, 0)
    let name =
        String.fromCString(UnsafePointer<CChar>(
            sqlite3_column_text(statement, 1)))
    let height = sqlite3_column_double(statement, 2)
    print("\(id). \(name!) 身高: \(height)")

}

sqlite3_finalize(statement)

// update
mydb.update(
    "students",
    cond: "id = 1",
    rowInfo: ["name": "'小強'", "height": "176.8"])

// delete
mydb.delete("students", cond: "id = 5")

}
```

以上便為本節範例的內容。

範例

本節範例程式碼放在 [database/sqlite](#)

Core Data

Core Data 是一個設計用來儲存資料的框架，背後操作的雖然仍是 SQLite，但其簡化了資料庫的處理，讓你不用了解 SQL 指令也可以快速的為應用程式建立並使用資料庫。

如果你是第一次接觸資料庫相關的知識，以下會簡單的介紹一下運作方式：

資料庫顧名思義，是一個用來儲存大量資料的容器，以現實生活來說，最簡單的資料庫可以用一個文件夾來比喻。

例如，一個文件夾是用來存放所有學生的資訊，裡頭每一頁都代表一名學生的資訊，每一名學生都會有各式各樣的資訊，像是姓名、座號、血型或出生年月日等等。

以上例子了解後，我們將它與 Core Data 的內容對比在一起，如下：

現實生活	文件夾	每一頁學生	學生的各個資訊
Core Data	Entity (實體)	每筆資料	Attribute (屬性)

所以假設當我們要為 Core Data 新增一筆資料時，這個步驟為：

1. 首先找出要操作的 Entity (拿出文件夾)。
2. 接著將要新增的一筆資料的各個 attribute 設定好(拿出一張新的紙，將一位新學生的基本資料填上)。
3. 在 Entity 增加這筆資料(為文件夾加入新的一頁，也就是新增這位學生的資訊)。
4. 儲存這個增加資料的動作(文件整理完畢，將文件夾關上)。

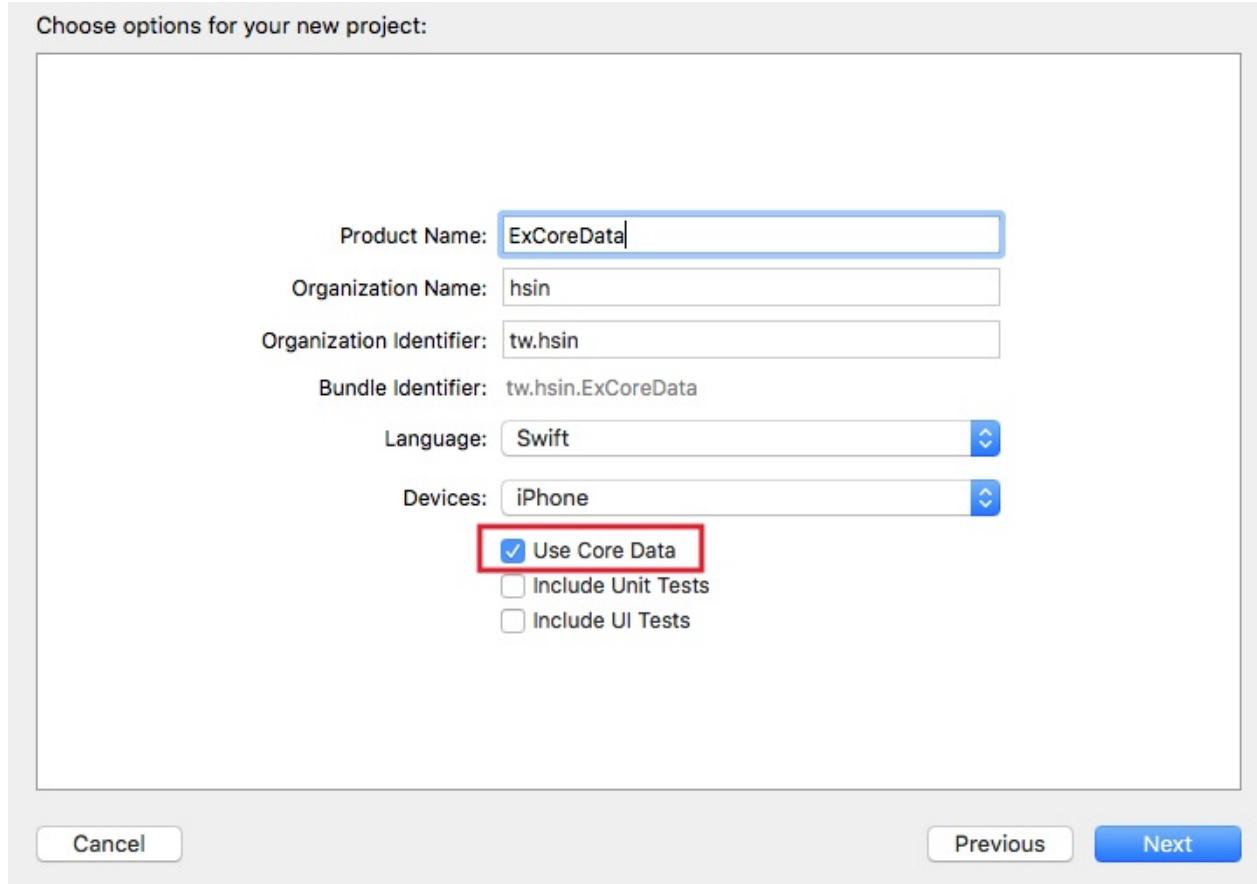
Hint

- 如果以關聯式資料庫的概念來對比的話，Core Data 的 Entity 與 Attribute 大約可以比對到 Table (資料表)與 Field (欄位)。
- 本節僅會介紹基本的功能，實際的資料庫操作可能會更為複雜。

以下會先介紹在應用程式中如何加入 Core Data，接著會介紹如何新增、讀取、更新與刪除資料，最後會將 Core Data 功能獨立寫在一個類別中，來把實際操作 Core Data 的程式碼封裝起來。

加入 Core Data

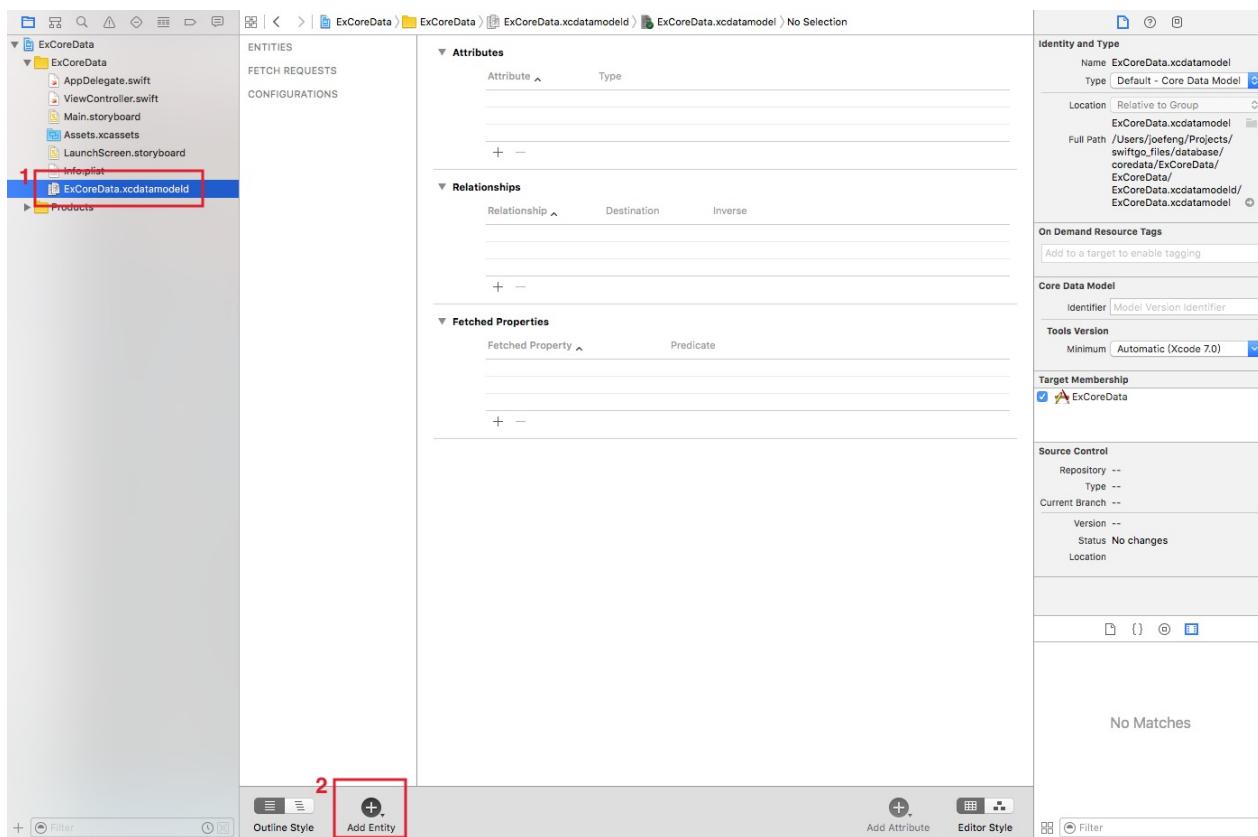
首先在 Xcode 裡，新建一個 **Single View Application** 類型的專案，取名為 `ExCoreData`。建立專案的過程中，請記得將 `Use Core Data` 打勾，如下圖：



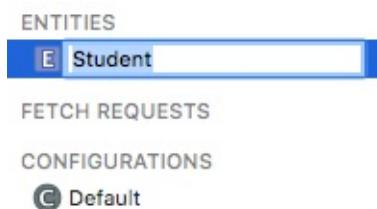
設定 Entity 與 Attribute

建立好專案後，可以看到左邊的專案檔案列表中，有一個名為 `ExCoreData.xcdatamodeld` 的檔案，這是用來設定 Entity 與 Attribute 的檔案。請點開這隻檔案並點擊下方的 `Add Entity` 按鈕，如下圖：

Core Data



接著將這個 Entity 命名為 Student (點擊兩下可命名)，如下圖：

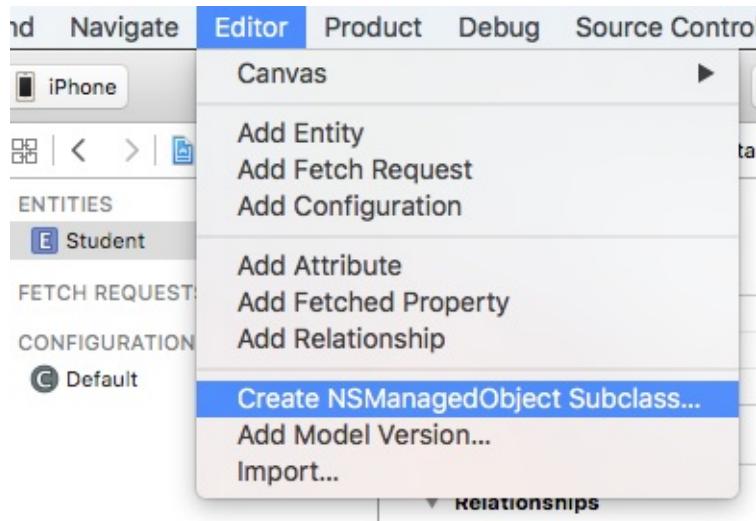


增加完 Entity 後，接著點擊 Attributes 的加號按鈕，依序增加三個 Attribute，分別為 id, name, height，Type 也就是每一個 Attribute 的類型，依序設定為 Integer 32, String, Double，如下圖：

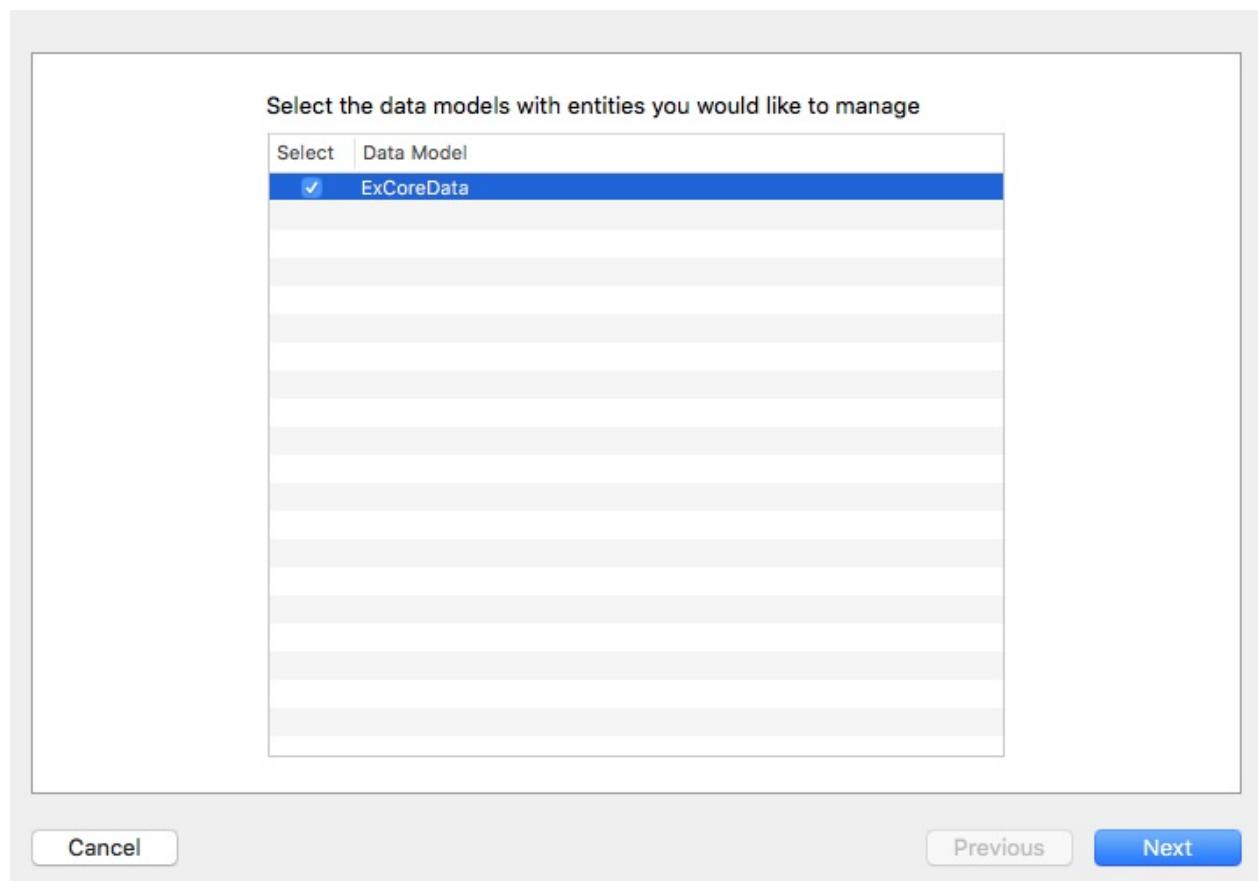


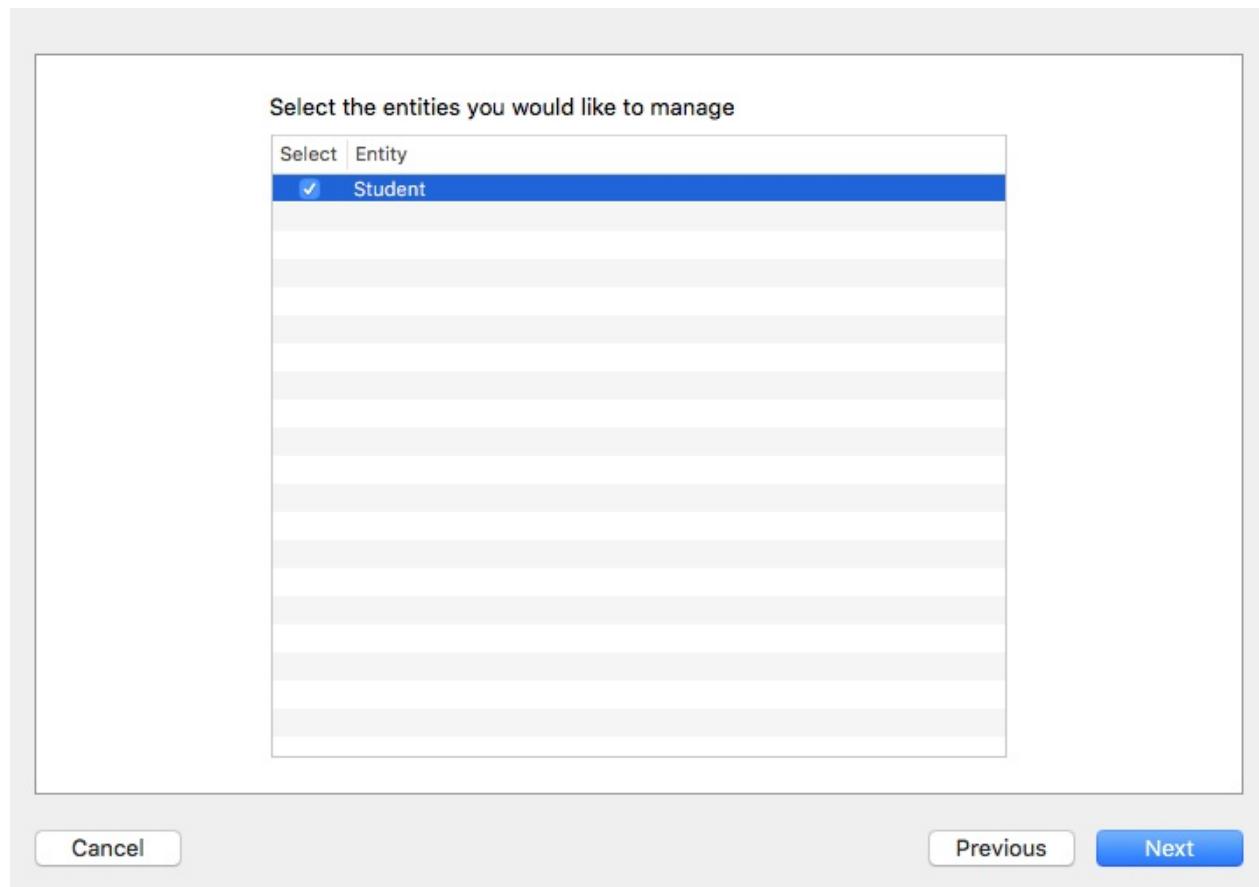
建立 Entity 的類別

為了要讓程式碼中可以使用這個 Entity，接著要建立一個繼承自 NSManagedObject 的 Student 類別。請點選 Xcode 工具列中的 Editor > Create NSManagedObject Subclass... 來讓 Xcode 為我們自動生成相關檔案，如下：

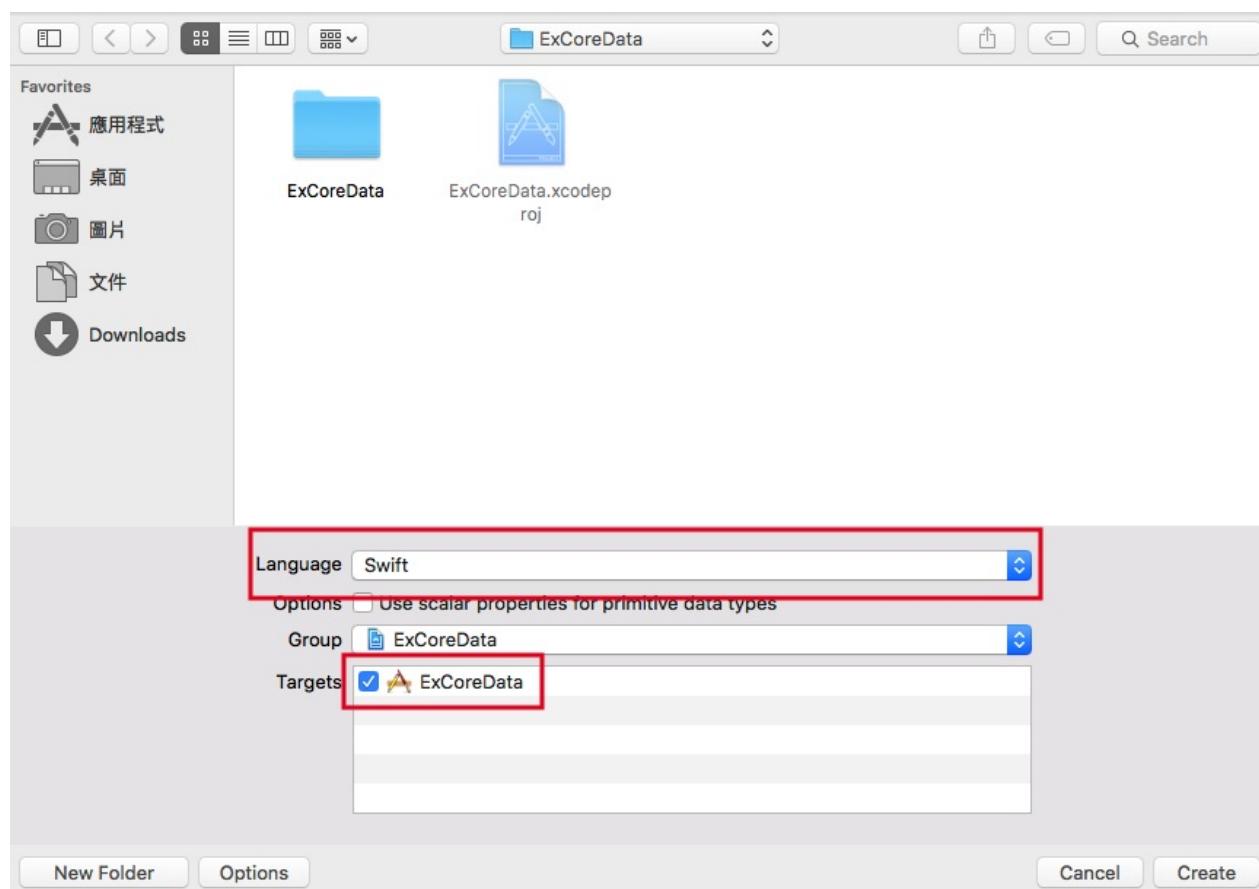


接著兩個步驟都是按 Next 繼續，如下面兩張圖：

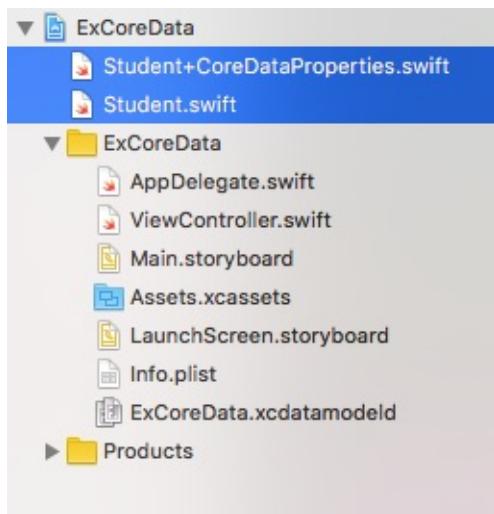




建立檔案的存放路徑時，記得 Language 要選擇 Swift，並記得 Targets 要打勾，如下圖：



建立完畢後，就會在左側檔案列表中看到兩隻新檔案，分別為一個繼承自 `NSManagedObject` 的 `Student` 類別，以及這個類別的擴展，如下：



這樣就完成了加入 Core Data 的步驟。

使用 Core Data

接著進入到程式碼的部份，首先宣告一個用來操作 Core Data 的常數：

```
// 用來操作 Core Data 的常數
let moc = (UIApplication.sharedApplication().delegate
           as! AppDelegate).managedObjectContext
```

在建立專案時如果有打勾 `Use Core Data`，建立完成後會為你自動生成相關程式碼在 `AppDelegate.swift` 中，裡面你可以看到操作的資料庫仍然是一個 `sqlite` 檔案，以及其他建立好的設定。

所以上述程式是以委任的方式，取得 `AppDelegate.swift` 的屬性 `managedObjectContext`，來操作 Core Data。

接著宣告一個 Entity 的名稱(記得要與前一小節設定的名稱一樣)，以供後續使用：

```
let myEntityName = "Student"
```

新增資料

新增資料的方式如下：

```
// insert
let student =
NSEntityDescription.insertNewObjectForEntityForName(
    myEntityName, inManagedObjectContext: moc)
as! Student

student.id = 1
student.name = "小強"
student.height = 173.2

do {
    try moc.save()
} catch {
    fatalError("\n(error)")
}
```

上述程式經由 `NSEntityDescription` 類別的 `insertNewObjectForEntityForName()` 方法來新增一筆資訊，這個方法的兩個參數分別為 **Entity** 名稱及一開始宣告的用來操作 **Core Data** 的常數。

接著這個方法回傳一個 `Student` 的實體並指派給常數 `student`，這時的進度就與稍前文件夾例子中的 **2.** 拿出一張新的紙相同，所以接著要將 `student` 的各個 `attribute` 設定好(將一位新學生的基本資料填上)。

目前已經有一筆新資料了，但尚未將這筆資料儲存，所以接著要使用常數 `moc` 的 `save()` 方法來儲存資料，而因為這個方法是一個**拋出函式**，所以使用 `do-catch` 語句來定義錯誤的捕獲。

如果沒有發生錯誤，即是順利儲存一筆新的資料。

讀取資料

讀取資料的方式如下：

```

// select
let request = NSFetchedRequest(entityName: myEntityName)

// 依 id 由小到大排序
request.sortDescriptors =
    [NSSortDescriptor(key: "id", ascending: true)]

do {
    let results =
        try moc.executeFetchRequest(request) as! [Student]

    for result in results {
        print("\(result.id). \(result.name!)")
        print("身高： \(result.height)")
    }
} catch {
    fatalError("\(error)")
}

```

要取得資料首先必須使用類別 `NSFetchedRequest` 來設置要取得的 Entity，以用來建立一個取得資料的請求(`request`)。

接著其實可以直接取得資料，但這個例子使用了屬性 `sortDescriptors` 額外設定取得資料排序的方式，這是一個型別為 `[NSSortDescriptor]` 的陣列，可以填入多個排序方式，上述例子中只填入一個 `NSSortDescriptor(key: "id", ascending: true)`，兩個參數依序為要依照哪一個 **attribute** 排序以及是否由小排到大。所以這個例子為：取得的資料要依照 `id` 的值由小排到大。(與關聯式資料庫的 `order by` 類似。)

最後使用 `moc` 的方法 `executeFetchRequest()` 來取得資料，這個方法的參數就是由類別 `NSFetchedRequest` 返回指派的常數。順利取回的資料會是一個型別為 `[Student]` 的陣列，便可以使用 `for-in` 迴圈來依序取得每筆資料。

更新資料

更新資料的方式如下：

```
// update
let request = NSFetchedRequest(entityName: myEntityName)
request.predicate = nil
let updateID = 1
request.predicate =
    NSPredicate(format: "id = \((updateID)")"

do {
    let results =
        try moc.executeFetchRequest(request)
        as! [Student]

    if results.count > 0 {
        results[0].height = 155
        try moc.save()
    }
} catch {
    fatalError("\((error))")
}
```

更新資料前需要先讀取資料，所以一開始與稍前的程式碼類似，同樣使用類別 `NSFetchedRequest` 來設置要取得的 Entity，以建立一個取得資料的請求(`request`)。

除了稍前介紹可以額外設定排序方式，這邊示範另一個屬性 `predicate`，這可以讓你設定取得資料的條件，例如這個例子設定條件為 `NSPredicate(format: "id = 1")`：取得 `id = 1` 的資料。(與關聯式資料庫的 `where` 條件類似。)

接著與稍前例子一樣，使用 `moc.executeFetchRequest()` 來取得資料，而這個例子因為是要更新資料，所以在順利取得後，將要更新的屬性設置完畢，再以 `moc.save()` 來儲存這個更新的動作。

Hint

- 這個例子中的 `request.predicate = nil` 不是必須的，是用來提醒你，如果有多个查詢資料庫的需求，在每次新的查詢要設定屬性 `predicate` 前，要先將其設置為 `nil` 以清空查詢條件。
- 如果查詢條件的類型為 `text`，記得參數 `format` 中要將該值以單引號 ' 包含起來，像是 `NSPredicate(format: "name = '小強'")` 這樣。

刪除資料

刪除資料方式如下：

```
// delete
let request = NSFetchedResultsController(entityName: myEntityName)
request.predicate = nil
let deleteID = 3
request.predicate =
    NSPredicate(format: "id = \(\(deleteID))")

do {
    let results =
        try moc.executeFetchRequest(request)
    as! [Student]

    for result in results {
        moc.deleteObject(result)
    }
    try moc.save()

} catch {
    fatalError("\(error)")
}
```

刪除資料與更新資料的方式類似，所以請參考稍前的例子，主要注意到 `moc.deleteObject()` 這個方法是用來刪除資料，而最後仍然要記得使用 `moc.save()` 來儲存這個刪除的動作。

以上便為基本操作 Core Data 的方式。

將 Core Data 功能獨立出來

這一小節會將 Core Data 功能獨立寫在一個類別中，來把實際操作 Core Data 的程式碼封裝起來，這樣一般在使用時就不會使用到 Core Data 相關的類別或函式。

首先以新增檔案的方式加入一個 `.swift` 檔案，命名為 `CoreDataConnect.swift`，記得檔案類型要選擇 `Swift File`：

iOS > Source > Swift File

接著打開這隻檔案，先建立一個類別及其內的屬性跟建構器：(記得要先 import CoreData)

```
class CoreDataConnect {
    var moc :NSManagedObjectContext!
    typealias MyType = Record

    init(moc:NSManagedObjectContext) {
        self.moc = moc
    }

}
```

上述程式中，使用 typealias 的特性設置一個型別別名，這樣在後續的資料操作，使用這個別名 MyType 即可，不用使用原始的 Entity 名稱 Record 。

新增資料

首先在上面這個類別中，定義新增資料的方法：

```
// insert
func insert(myEntityName:String,
attributeInfo:[String:String]) -> Bool {
    let insetData =
NSEntityDescription.insertNewObjectForEntityForName(
    myEntityName, inManagedObjectContext: self.moc)
    as! MyType

    for (key,value) in attributeInfo {
        let t =
insetData.entity.attributesByName[key]?.attributeType

        if t == .Integer16AttributeType
        || t == .Integer32AttributeType
        || t == .Integer64AttributeType {
            insetData.setValue(Int(value),
                forKey: key)
```

```

        } else if t == .DoubleAttributeType
        || t == .FloatAttributeType {
            insetData.setValue(Double(value),
                forKey: key)
        } else if t == .BooleanAttributeType {
            insetData.setValue(
                (value == "true" ? true : false),
                forKey: key)
        } else {
            insetData.setValue(value, forKey: key)
        }
    }

    do {
        try moc.save()

        return true
    } catch {
        fatalError("\(error)")
    }

    return false
}

```

這個方法與稍前介紹新增資料時的程式碼一樣，有一點要注意的是，這邊因為其中一個傳入的參數：要新增的 **attribute** 及其值，是統一以字串傳入，所以這個方法內需要根據 **attribute** 的類型

`student.entity.attributesByName[key]?.attributeType` 來轉換型別為 `Int, Double, Bool` 或是原本的字串，再以方法 `setValue(_, forKey:)` 設置值並儲存。

讀取資料

接著定義讀取資料的方法：

```

// select
func fetch(myEntityName:String, predicate:String?,
sort:[[String:Bool]]?, limit:Int?) -> [MyType]? {
    let request = NSFetchedRequest(

```

```
entityName: myEntityName)

// predicate
if let myPredicate = predicate {
    request.predicate =
        NSPredicate(format: myPredicate)
}

// sort
if let mySort = sort {
    var sortArr :[NSSortDescriptor] = []
    for sortCond in mySort {
        for (k, v) in sortCond {
            sortArr.append(
                NSSortDescriptor(
                    key: k, ascending: v))
        }
    }

    request.sortDescriptors = sortArr
}

// limit
if let limitNumber = limit {
    request.fetchLimit = limitNumber
}

do {
    let results =
        try moc.executeFetchRequest(request)
    as! [MyType]

    return results
} catch {
    fatalError("\\"(error)")
}

return nil
}
```

讀取資料的方法將講過的兩個額外查詢功能：查詢條件 `predicate` 與排序方式 `sortDescriptors` 以及限制查詢筆數 `fetchLimit` 都加入，並將其都設為可選型別，這樣如果不需要時填入 `nil` 即可，返回的是一個型別為 `[Student]` 的陣列。

更新資料

定義更新資料的方法：

```
// update
func update(myEntityName:String, predicate:String?, 'attributeInfo:[String:String]) -> Bool {
    if let results = self.fetch(
        myEntityName,
        predicate: predicate, sort: nil, limit: nil) {
        for result in results {
            for (key,value) in attributeInfo {
                let t =
result.entity.attributesByName[key]?.attributeType

                if t == .Integer16AttributeType
                || t == .Integer32AttributeType
                || t == .Integer64AttributeType {
                    result.setValue(
                        Int(value), forKey: key)
                } else if t == .DoubleAttributeType
                || t == .FloatAttributeType {
                    result.setValue(
                        Double(value), forKey: key)
                } else if t == .BooleanAttributeType {
                    result.setValue(
                        (value == "true" ? true : false),
                        forKey: key)
                } else {
                    result.setValue(
                        value, forKey: key)
                }
            }
        }
    }
}
```

```
do {
    try self.moc.save()

    return true
} catch {
    fatalError("\(error)")
}

return false
}
```

這邊會先以讀取資料方法，取得要更新的資料，再將各 `attribute` 設置好後才再儲存，與新增資料相同，統一以字串傳入，所以需要根據 `attribute` 類型來轉換型別。

刪除資料

定義刪除資料的方法：

```

// delete
func delete(myEntityName:String, predicate:String?) -> Bool {
    if let results = self.fetch(myEntityName,
        predicate: predicate, sort: nil, limit: nil) {
        for result in results {
            self.moc.deleteObject(result)
        }
    }

    do {
        try self.moc.save()

        return true
    } catch {
        fatalError("\(error)")
    }
}

return false
}

```

這邊會先以讀取資料方法，取得要刪除的資料，再將取得的資料刪除，並儲存刪除的動作。

使用這個類別

將 Core Data 功能寫在一個類別後，接著將 ViewController.swift 的 `viewDidLoad()` 內容改寫為：

```

let myEntityName = "Student"
let coreDataConnect = CoreDataConnect(moc: self.moc)

// auto increment
let myUserDefaults =
    UserDefaults.standardUserDefaults()
var seq = 1
if let idSeq = myUserDefaults.objectForKey("idSeq")
    as? Int {

```

```
    seq = idSeq + 1
}

// insert
let insertResult = coreDataConnect.insert(
    myEntityName, attributeInfo: [
        "id" : "\u{seq}",
        "name" : "'小強'",
        "height" : "176.1"
    ])
if insertResult {
    print("新增資料成功")

    myUserDefaults.setObject(seq, forKey: "idSeq")
    myUserDefaults.synchronize()
}

// select
let selectResult = coreDataConnect.fetch(
    myEntityName,
    predicate: nil, sort: [[["id":true]]], limit: nil)
if let results = selectResult {
    for result in results {
        print("\(result.id). \(result.name!)")
        print("身高: \(result.height)")
    }
}

// update
let updateName = "二強"
var predicate = "name = '\u{updateName}'"
let updateResult = coreDataConnect.update(
    myEntityName,
    predicate: predicate,
    attributeInfo: ["height":"162.2"])
if updateResult {
    print("更新資料成功")
}

// delete
```

```
let deleteID = 2
predicate = "id = \u{deleteID}"
let deleteResult = coreDataConnect.delete(
    myEntityName, predicate: predicate)
if deleteResult {
    print("刪除資料成功")
}
```

上述程式可以發現已經看不到操作 Core Data 相關的類別與函式，因為已經都寫在 CoreDataConnect.swift 中了。

其中要提醒的是，因為 Core Data 沒有提供 auto increment 的功能(每次新增資料都自動為其中一個 attribute 遞增的功能)，所以這邊以 NSUserDefaults 儲存一個數值來手動建立 auto increment 功能，每次新增資料成功時都將這個值加一，下次新增時會再取出這個值來使用。

以上便為本節範例的內容。

範例

本節範例程式碼放在 [database/coredatabase](#)

iPhone Apps

在介紹完 Swift 語法、UIKit 元件以及資料庫後，這節的內容會介紹三個完整的 iPhone App，來綜合應用從一開始到這邊為止學習到的知識。除了前面介紹的內容外，如果應用程式需要額外功能，也會另開一小節介紹。

以下是這三個應用程式的簡介，你可以依序閱讀或是按所需選擇其中一節的內容閱讀也可以：

事 待辦事項

新增、編輯、點擊完成或是刪除待辦事項。

你可以在 App Store 中找到這個應用程式 [待辦事項](#)。

北 遊玩臺北

條列出臺北市的景點、公園、廁所與住宿資訊，並依照距離使用者定位位置由近至遠排序。

你可以在 App Store 中找到這個應用程式 [遊玩臺北](#)。

財 記帳

新增、編輯或是刪除消費記錄，並會以月為單位列表顯示所有記錄。

你可以在 App Store 中找到這個應用程式 [記帳](#)。

待辦事項

本節會介紹如何建立一個待辦事項應用程式，用來新增、編輯、點擊完成或是刪除事項。

你可以在 App Store 中找到這個應用程式，名稱為待辦事項 。

以下列出會使用到的 UIKit 元件與功能，如果還有尚未了解的地方，可以先往前面章節複習一下：

- 文字標籤 `UILabel`
- 文字輸入 `UITextField`
- 按鈕 `UIButton`
- 提示框 `UIAlertController`
- 圖片 `UIImageView`
- 開關 `UISwitch`
- 表格 `UITableView`
- 導覽控制器 `UINavigationController`
- 手勢 `UIGestureRecognizer`
- 儲存資訊 `NSUserDefaults`
- `Core Data`

除了上述項目外，因應此應用程式的需求，後面幾小節會額外介紹需要的功能：

- 程式之外的設定
- 播放音效
- `UITableView` 的編輯模式

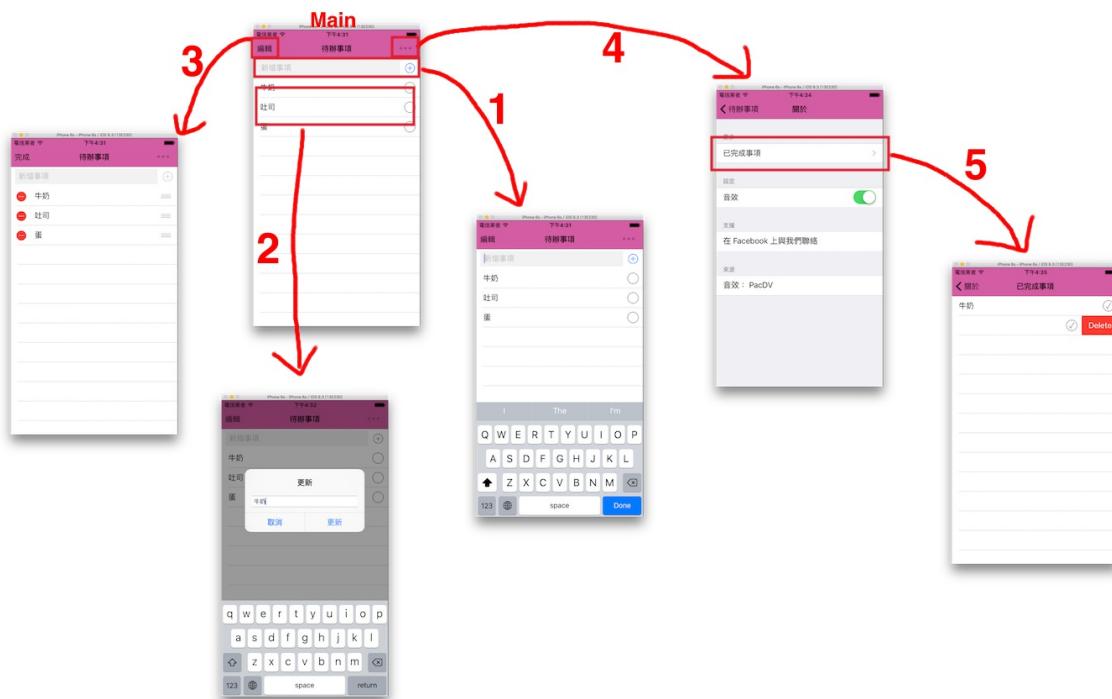
範例

本節範例程式碼放在 `apps/todo`

規劃與實作

請先在 Xcode 打開這個專案([apps/todo/Todo](#))以供後續與文章內容比對檢視，本小節說明僅會提示部分內容，不會將所有程式碼都寫出來，請以專案程式碼為主。

首先介紹待辦事項應用程式可以操作的動作，先看下圖流程：



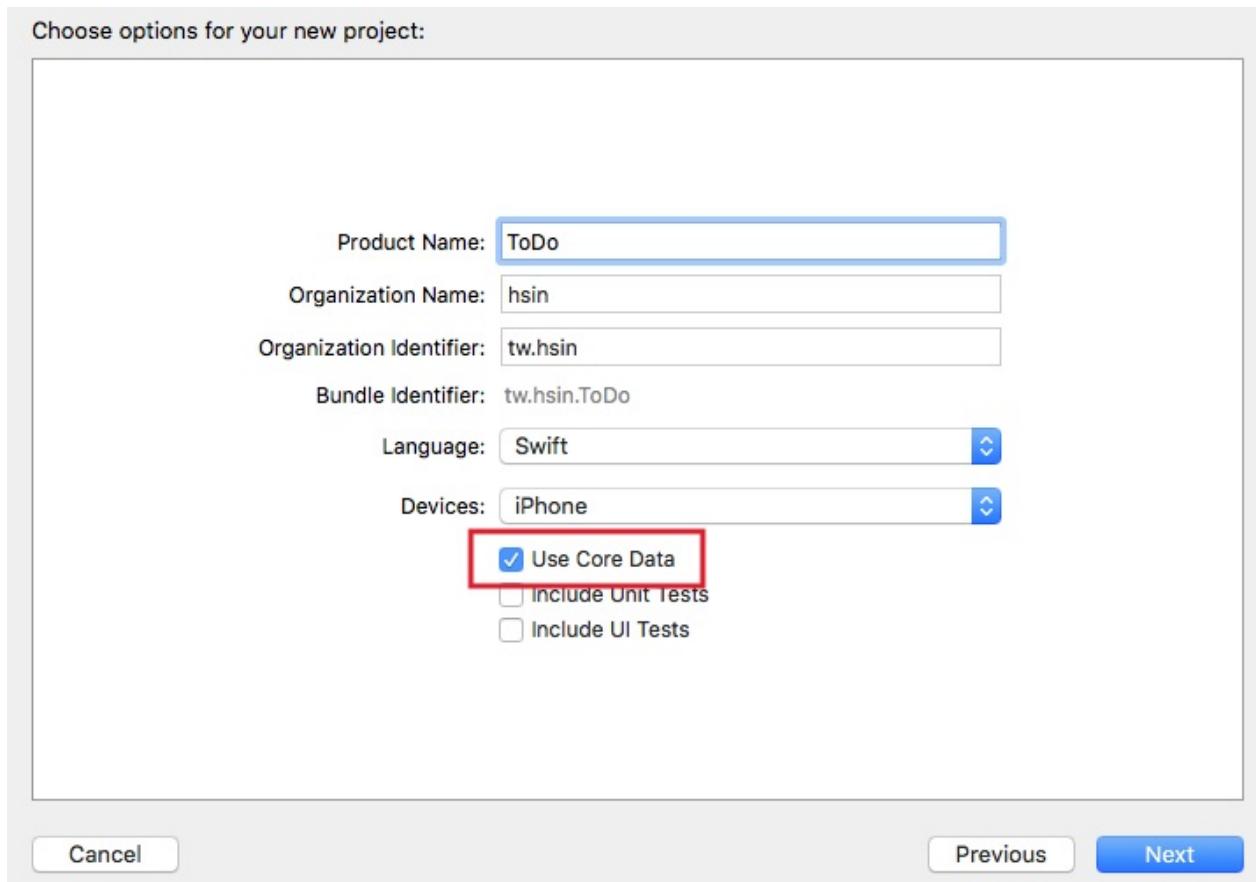
上圖從 Main 為首頁開始，依序可以操作如下的動作：

1. 新增：在首頁第一行輸入事項文字，點擊右邊的加號，即可新增事項。
2. 刪除與編輯：待辦事項會列在首頁下方，點擊項目可以修改事項文字。可以點擊各筆項目右邊的確認框打勾以完成事項。
3. 排序或刪除：點擊首頁左上角編輯，可以為事項排序或是刪除事項。
4. 關於：點擊首頁右上角關於，可以進入關於頁。可以檢視已完成事項、關閉或開啟音效(新增、打勾、刪除事項時的音效)、前往支援網頁與來源網頁。
5. 已完成事項：打勾的已完成事項會列在這頁，你也可以再次點擊確認框來取消完成，事項會回到首頁列表，或是也可左滑刪除這個事項(首頁列表也可以左滑刪除)。

前置作業

首先在 Xcode 裡，新建一個 **Single View Application** 類型的專案，取名為 ToDo。

這個專案會使用 Core Data 來儲存與操作資料，所以建立專案的過程中，請記得將 **Use Core Data** 打勾，如下圖：



接著請依照 **Core Data** 的內容依序建立需要的檔案與功能，Entity 取名為 Record，以及建立四個 Attributes，如下圖示：

Attribute	Type
S content	String
B done	Boolean
N id	Integer 32
N seq	Integer 32

上述 Core Data 設定完後，專案應該新增好下列檔案：

- Record+CoreDataProperties.swift
- Record.swift
- CoreDataConnect.swift

在寫程式碼之前，先依照 [程式之外的設定](#) 的步驟設定好。以及將需要的 Swift 檔案、圖示檔案、音效檔案都先加入至專案中：

- BaseViewController.swift
- MoreViewController.swift
- CheckedRecordsViewController.swift
- icons 目錄中的三個圖示檔案
- sounds 目錄中的四個音效檔案

AppDelegate.swift

在進入應用程式的畫面前，需要在 AppDelegate.swift 中先做些設定：

- 在 UserDefaults 中儲存是否開啟音效的值，以供後續使用與更新。
- 設置導覽列的一些預設樣式。
- 依照 [導覽控制器 UINavigationController](#) 的說明，將根視圖設為一個 UINavigationController，並指定 ViewController 為第一個視圖控制器。

首頁

因為首頁 ViewController 與已完成事項頁 CheckedRecordsViewController 要做的事情差不多，所以將重複的功能都寫在一個繼承自 UIViewController 的 BaseViewController，並讓兩者都再繼承自他，這樣重複的功能便不用寫兩遍。

BaseViewController.swift

先注意以下 BaseViewController 的幾個屬性：

- 建立屬性 checkStatus 來辨別目前執行動作的是首頁還是已完成事項頁。
- 建立陣列屬性 myRecords 存放取得的事項列表，在 UITableView 的委任方法中，則是直接對這個陣列做操作。
- 建立屬性 cehckTagTemp 用來幫助設置每個確認框所代表的事項資訊。

viewDidLoad()

在 viewDidLoad() 方法中，先連接 Core Data 與設置基本設定。因為其他頁的設定改變，可能導致這頁的內容會有所不同，所以大多的功能是寫在 viewWillAppear() 方法裡()。

viewWillAppear()

在 `viewWillAppear()` 方法中，先確認音效是否開啟，接著取得事項列表(使用屬性 `checkStatus` 來辨別是首頁還是已完成事項頁)，最後依據是否開啟音效來設置播放器或是設為 `nil`。(請參考後續小節播放音效的介紹。)

UITableView 的委任方法

`tableView(tableView: UITableView, cellForRowAt indexPath: IndexPath) -> UITableViewCell` 方法是用來顯示每筆事項的內容，在前面章節介紹 `UITableView` 時可以得知 `UITableViewCell` 是重複使用的，所以要注意先將內部的 `UIButton` 移除，才不會出現非預期中的按鈕，如下：

```
// 移除舊的按鈕
for view in cell.contentView.subviews {
    if let v = view as? UIButton {
        v.removeFromSuperview()
    }
}
```

接著才依照屬性 `checkStatus` 加上勾選確認框的 `UIButton`。

按下按鈕執行動作的方法

`checkBtnAction(_:)` 方法是負責按下事項打勾確認框時的動作，會先依照 `UIButton` 的 `tag` 來找出這筆事項的 `id`，再作打勾或取消打勾的動作，這邊可以注意到沒有依據音效是否開啟來播放音效，如下：

```
// 音效
doneSound?.play()
```

因為在稍前 `viewWillAppear()` 方法中，已設置成播放器或 `nil`，當設為 `nil` 時是不會發生任何事，所以只要寫上面這樣即可。

ViewController.swift

繼承自 `BaseViewController` 的 `ViewController` 只需要再將新增事項的部份加上就差不多了。

viewDidLoad()

首先設置屬性 `checkStatus` 為 `false`，表示這頁的列表是待辦事項。接著依序建立 `UITableView`、前往更多頁面按鈕、新增事項輸入框及按鈕。

viewWillAppear()

使用 `super.viewWillAppear(true)` 來執行父類別中的動作。接著為了每次到這頁時都確實不會是 `UITableView` 的編輯模式，所以會執行下列程式：

```
// 進入 非 編輯模式  
myTableView.setEditing(true, animated: false)  
self.editBtnAction()
```

按下按鈕執行動作的方法

`editBtnAction()` 方法是負責按下左上編輯按鈕時的動作，除了切換自身按鈕的樣式外，還要切換新增事項輸入框的開放與限制和每筆事項的打勾確認框顯示與隱藏。

`addBtnAction()` 方法負責新增事項的動作，這邊加上一個沒有填入文字時無法新增的限制。

UITableView 的委任方法

`UITableView` 可以使用 `setEditing(_:, animated:)` 方法來切換編輯模式。

在編輯模式時，`tableView(tableView: UITableView, moveRowAtIndexPath: NSIndexPath, sourceIndexPath: NSIndexPath, toIndexPath destinationIndexPath: NSIndexPath)` 方法用來執行排序後的動作，更多內容請參考 [UITableView 的編輯模式](#)。

其他方法

點擊列表中的事項會執行 `updateRecordContent(_:)` 方法來編輯事項內容，這部份使用 `UIAlertController` 跳出一個提示框來編輯。

更多頁面

這頁使用 `UITableView` 的 `.Grouped` 特性各別列出不同的功能：

1. 前往已完成事項頁面。
2. 使用 **UISwitch** 元件來開關音效。
3. 前往外部 **Facebook** 網頁。
4. 前往外部 音效資源 網頁。

前往外部網頁時，使用下列方式：

```
let requestUrl = NSURL(string:  
    "https://www.facebook.com/1640636382849659")  
UIApplication.sharedApplication().openURL(requestUrl!)
```

已完成事項 頁面

繼承自 **BaseViewController** 的 **ChechedRecordsViewController** 其實功能很簡單，首先設置屬性 **checkStatus** 為 **true**，表示這頁的列表是已完成事項，再將 **UITableView** 設置完成。其餘的事已經在 **BaseViewController** 都做完了。

範例

此應用程式範例程式碼放在 [apps/todo/Todo](#)

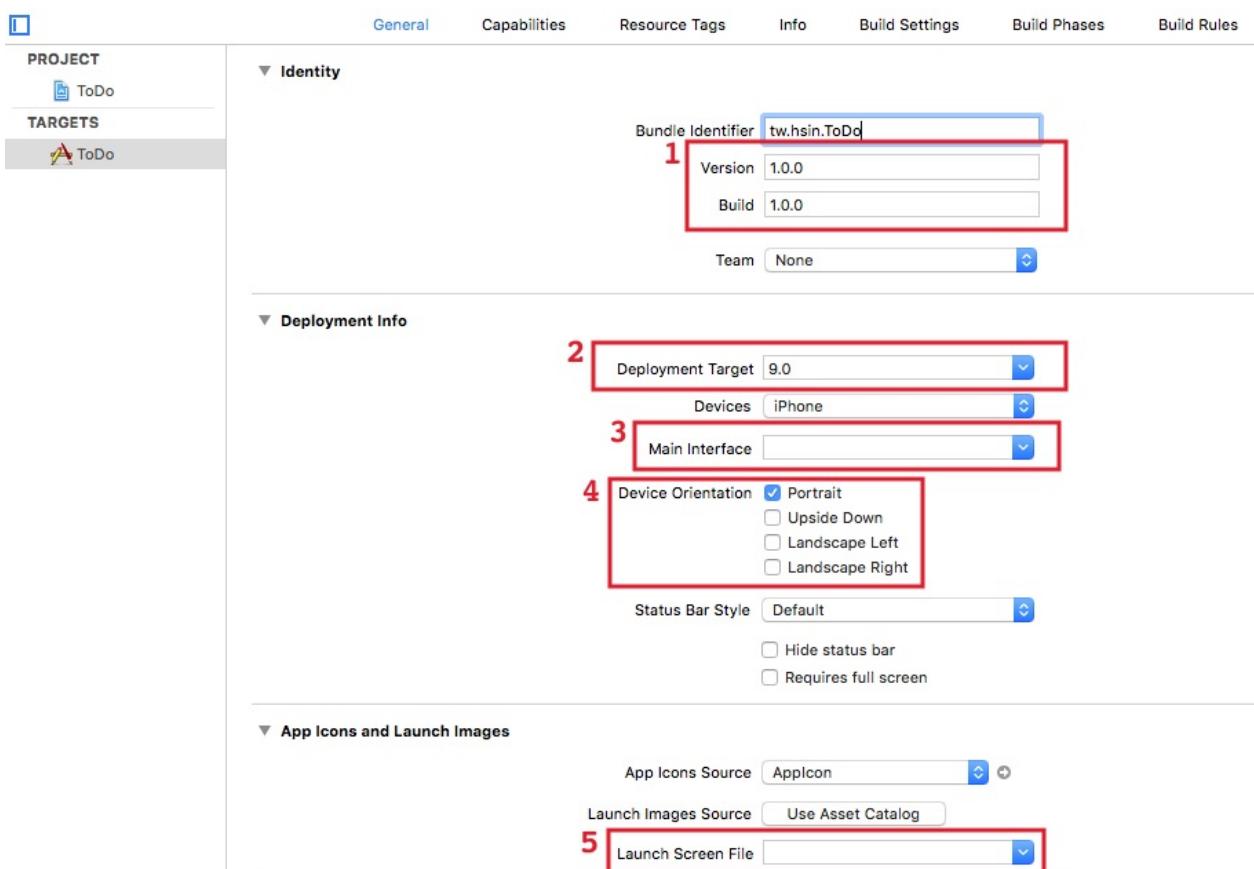
你也可以在 App Store 中找到這個應用程式，名稱為 [待辦事項](#) 。

程式之外的設定

除了編寫 Swift 程式，要上架應用程式前，還需要設置一些程式之外的設定，以下會依序介紹。

基本設定

專案建立完成後，先看到 TARGETS > ToDo > General ，設定如下圖的部分：



1. 將 Version 與 Build 版本都設定為初版 1.0.0，版本數字沒有特別規定，主要限制是每次更新上架的 Version 都必須比前一版更高，而 Build 指的是開發的版本，每次 Archive (打包送審)到 App Store 審核的 Build 都必須不同，如果因為一些原因被拒絕，修正後則需要設定更高的 Build 版本才能再送審。所以一個 Version 可能會因為一些原因，會有多個 Build 版本，兩者數字為獨立各別使用。
2. 支援的 iOS 作業系統版本，這邊設置為支援 iOS 9.0 以上的裝置。
3. 因為要以純程式碼來編寫應用程式，所以將 Storyboard 刪掉。
4. 可以支援的 iPhone 畫面方向，這邊以較單純的方式進行，只留 Portrait，也

就是原本直立的方向。

5. 應用程式啓動時的畫面，原本也可以使用一個 Storyboard 來呈現，這邊簡化成使用圖片，所以這欄位也是將 Storyboard 刪掉。

Hint

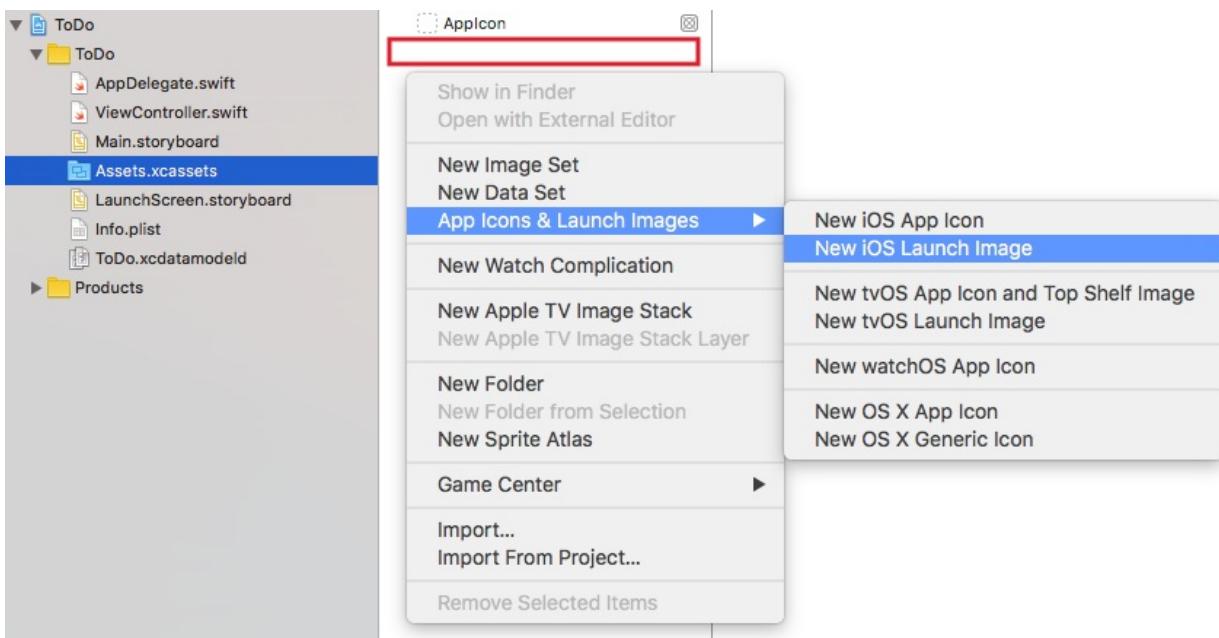
- 更多內容請參考[介面簡介](#)。

接著看到 PROJECT > ToDo > Info ，設定如下圖的 iOS 作業系統版本支援部分：



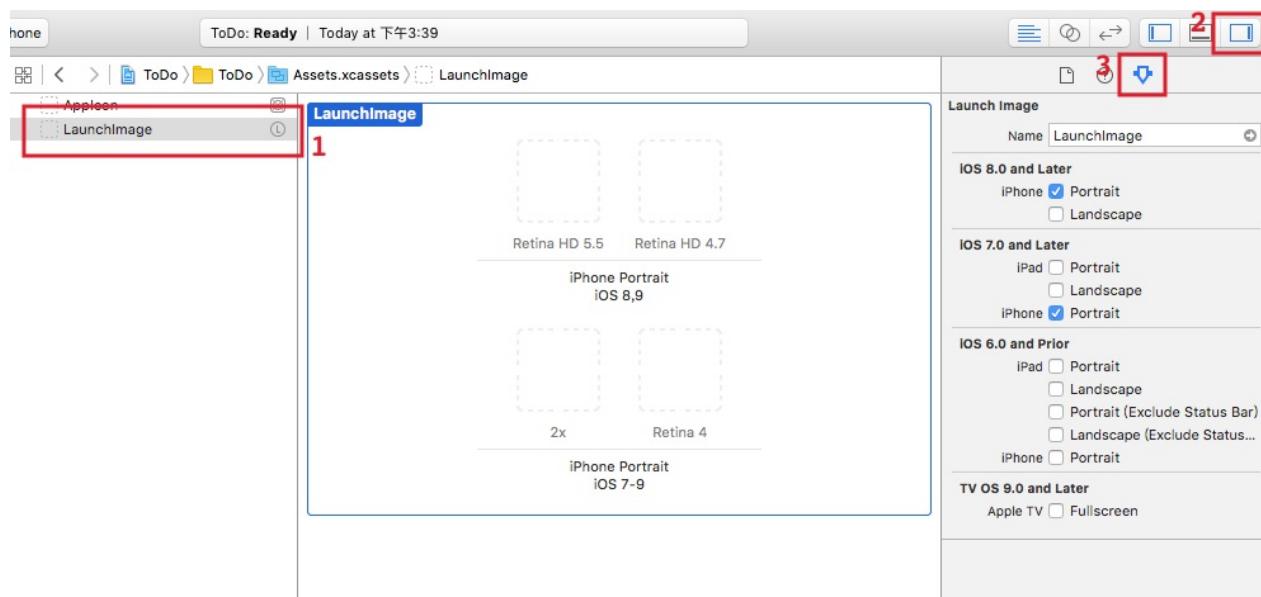
Launch Images

預設是沒有應用程式啓動時的圖片檔案，所以必須手動加入，先點選 Assets.xcassets 檔案，並在空白處點右鍵，選擇 App Icons & Launch Images > New iOS Launch Image ，如下圖：

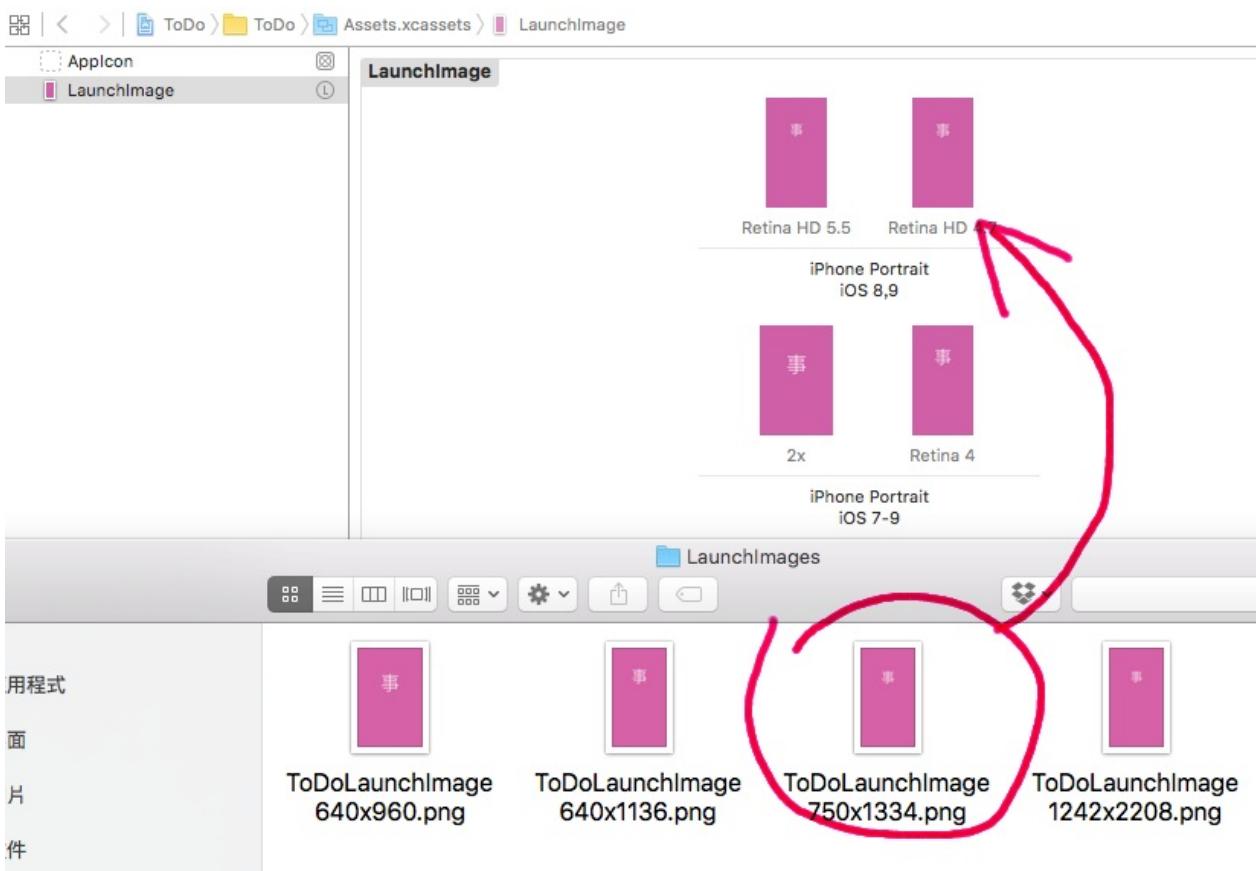


接著便會建立為如下圖的格式，依序點擊後可看到右側欄的設定，請依照應用程式實際支援的版本及類型，事先設計好要使用的各尺寸圖片，這邊會支援 iPhone 4S 以上的機型，所以需要如下這些尺寸：

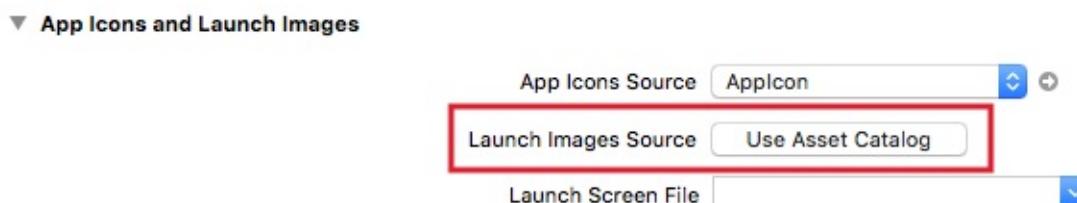
- 640x960 : iPhone 4S。
- 640x1136 : iPhone 5、iPhone 5s、iPhone SE。
- 750x1334 : iPhone 6、Phone 6s。
- 1242x2208 : iPhone 6 Plus、iPhone 6s Plus。



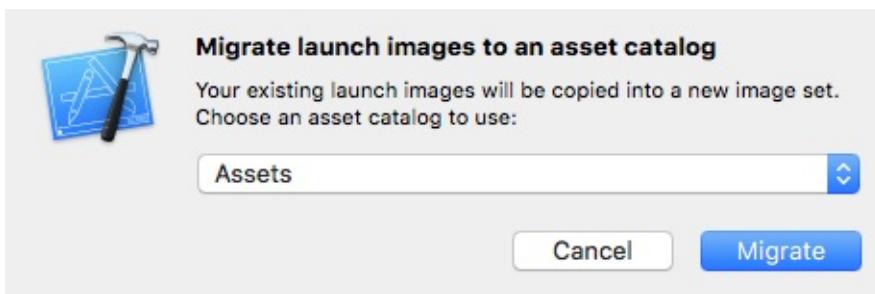
直接將 Finder 的圖片檔案拖曳進 Xcode 的列表中，如下：



接著回到 TARGETS > ToDo > General，點選 Use Asset Catalog 來將前面設定的啓動圖檔與專案做連結，如下：

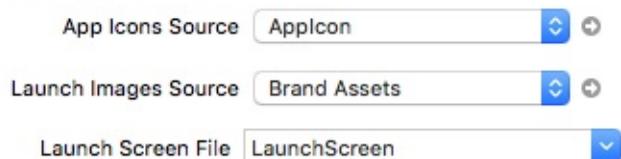


會有一個確認提示，請選擇預設存在的 Assets 即可，如下：

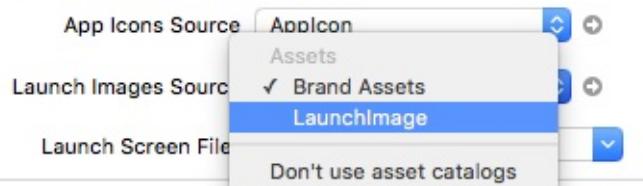


選擇後，Launch Images Source 可能會出現一個 Brand Assets，先別緊張，點擊他後選擇稍前設定好的 LaunchImage 即可，如下：

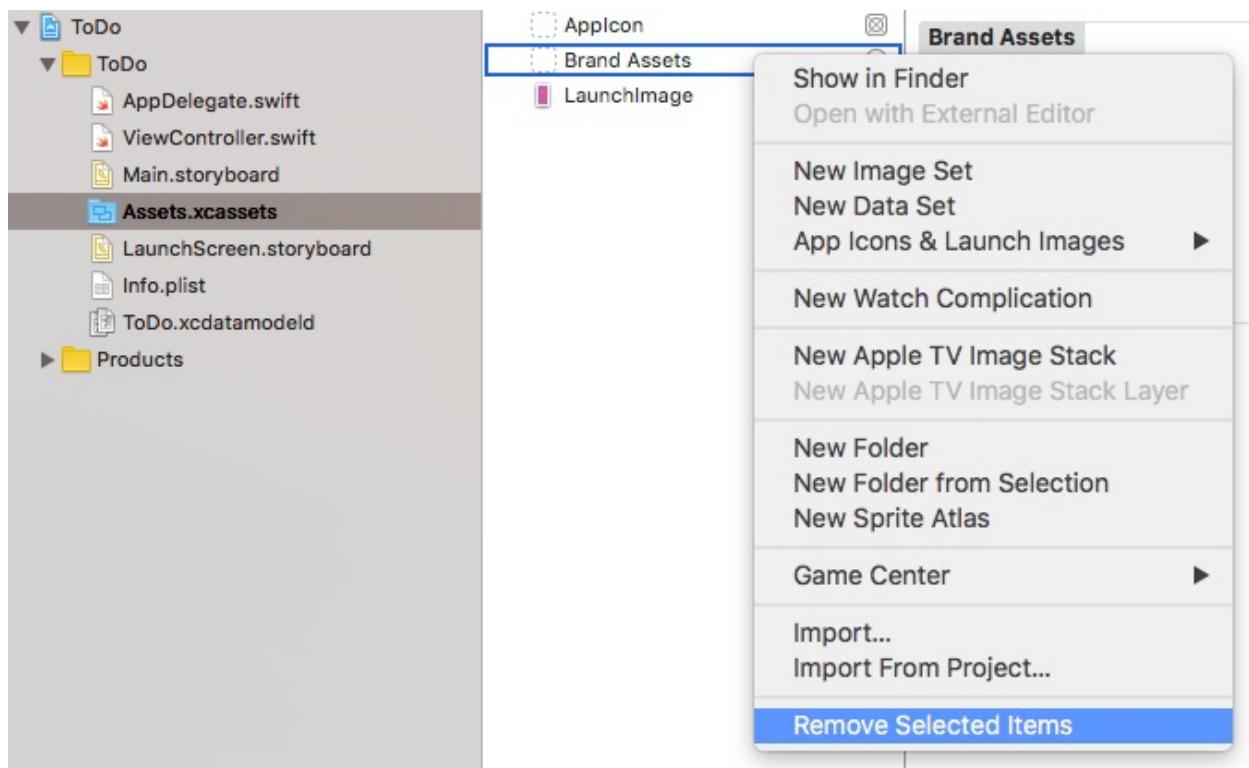
▼ App Icons and Launch Images



▼ App Icons and Launch Images



接著再切換到 Assets.xcassets 檔案，將 Brand Assets 刪除，如下圖：

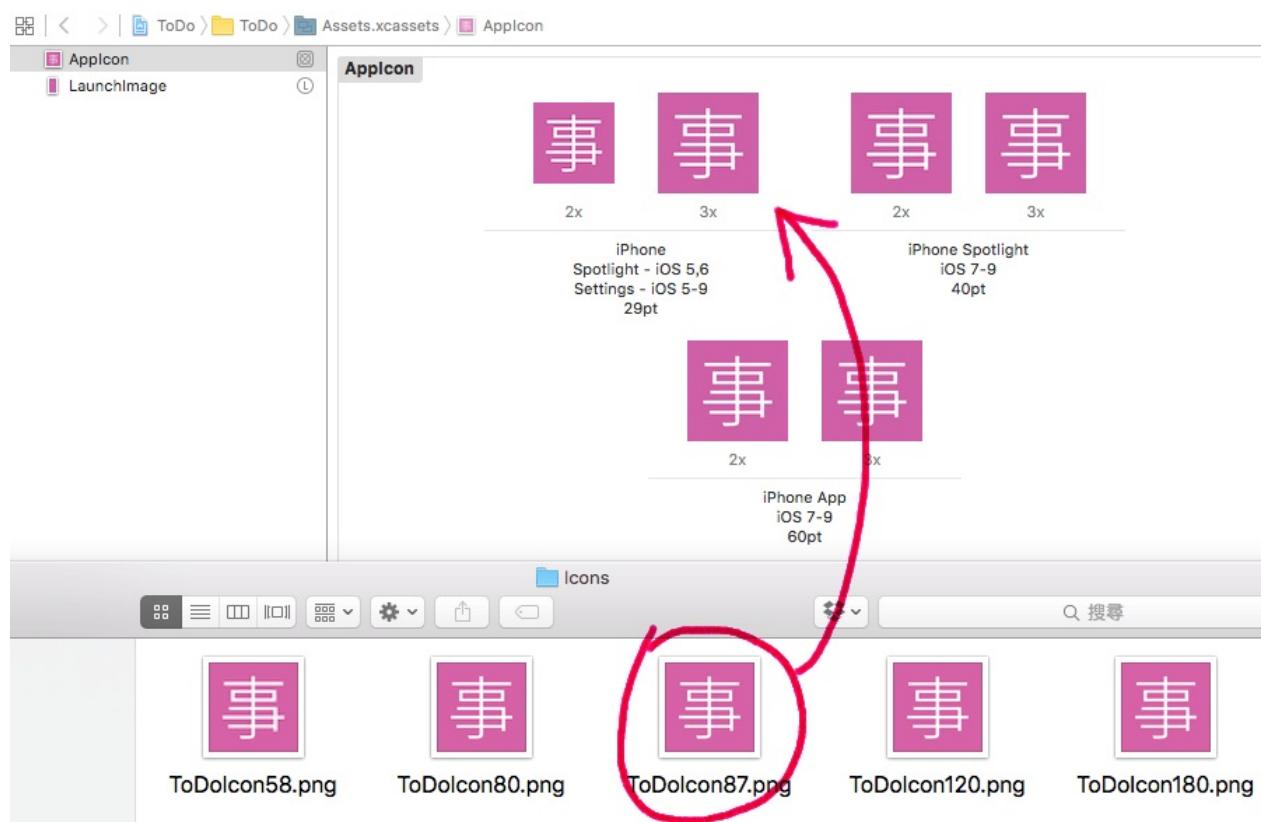


Icons

在 Assets.xcassets 檔案中，可以看到另一個 AppIcon 列表，是用來設置應用程式的圖示，依序點擊後可看到右側欄的設定，請依照應用程式實際支援的版本及裝置，事先設計好要使用的各尺寸圖片：



直接將 Finder 的圖片檔案拖曳進 Xcode 的列表中，如下：



刪除 Storyboard

因為 Storyboard 檔案都已經不需要了，所以以刪除檔案的方式將 Main.storyboard 與 LaunchScreen.storyboard 兩個檔案刪除。

應用程式名稱

先開啟 Info.plist 檔案，看到 Bundle name 這個欄位，代表應用程式的名稱，預設是顯示專案名稱 \$(PRODUCT_NAME) ，如下：

Key	Type	Value
▼ Information Property List Dictionary (12 items)		
Localization native development re...	String	en
Executable file	String	\$(EXECUTABLE_NAME)
Bundle identifier	String	\$(PRODUCT_BUNDLE_IDENTIFIER)
InfoDictionary version	String	6.0
Bundle name	String	\$(PRODUCT_NAME)
Bundle OS Type code	String	APPL
Bundle versions string, short	String	1.0.0
Bundle creator OS Type code	String	????
Bundle version	String	1.0.0
Application requires iPhone environ...	Boolean	YES
► Required device capabilities	Array	(1 item)
► Supported interface orientations	Array	(1 item)

這邊將其改為欲設置的名稱，如下：



這樣在 iPhone 的應用程式列表中，便會顯示上面設置的名稱，而不是專案名稱：



播放音效

首先在 Xcode 裡，新建一個 **Single View Application** 類型的專案，取名為 ExSound。

一開始先以加入檔案的方式加入一個音效檔案，並為 **ViewController** 建立一個變數以供後續使用，如下：

```
class ViewController: UIViewController {  
    var myPlayer :AVAudioPlayer!  
  
    // 省略  
}
```

引入函式庫

預設的應用程式沒有包含音效相關的函式，所以需先引入 **AVFoundation** 函式庫，如下：

```
import AVFoundation
```

建立播放器變數

在 `viewDidLoad()` 中建立變數，用來控制音效相關的動作：

```
// 建立播放器
let soundPath = NSBundle.mainBundle().pathForResource(
    "sound0132", ofType: "wav")
do {
    myPlayer = try AVAudioPlayer(
        contentsOfURL: NSURL.fileURLWithPath(soundPath!))

    // 重複播放次數 設為 0 則是只播放一次 不重複
    myPlayer.numberOfLoops = 0

} catch {
    print("error")
}
```

上述程式先取得音效檔案的路徑，再將音效檔案傳入初始化播放器當做參數建立。這是一個拋出函式，所以需以 do-catch 語句來定義錯誤的捕獲及處理。

接著則是建立一個按鈕，按下時會播放音效：

```
// 建立一個按鈕
let myButton = UIButton(frame: CGRect(
    x: 100, y: 200, width: 100, height: 60))
myButton.setTitle("音效", forState: .Normal)
myButton.setTitleColor(
    UIColor.blueColor(), forState: .Normal)
myButton.addTarget(
    self,
    action: #selector(ViewController.go),
    forControlEvents: .TouchUpInside)
self.view.addSubview(myButton)
```

按下按鈕後執行動作的方法：

```
func go() {
    // 播放音效
    myPlayer.play()
}
```

以上即為這個範例的內容。

音效來源

- <http://www.pacdv.com/sounds/>

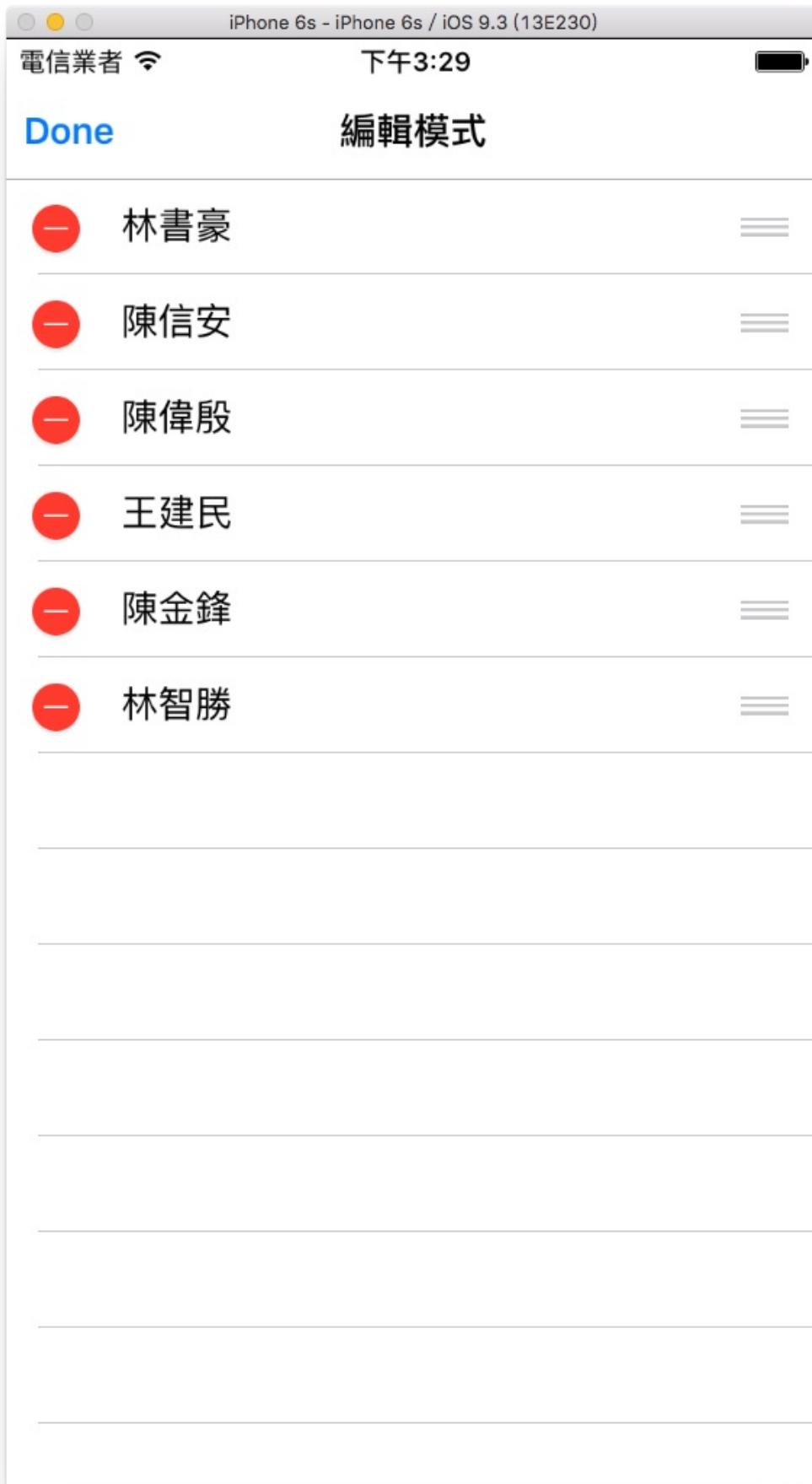
範例

本小節範例程式碼放在 `apps/todo`

UITableView 的編輯模式

在稍前 [表格 UITableView](#) 章節已經介紹過基本的使用方式，其實 UITableView 提供的功能非常的多，這小節會再介紹另一個也是十分常使用到的編輯模式。

以下是本小節目標：



首先在 Xcode 裡，新建一個 **Single View Application** 類型的專案，取名為 ExEditUITableView。

並為 **ViewController** 建立兩個屬性以供後續使用：

```
class ViewController: UIViewController {  
    var myTableView : UITableView!  
    var info =  
        ["林書豪", "陳信安", "陳偉殷", "王建民", "陳金鋒", "林智勝"]  
  
    // 省略  
}
```

為了方便設置切換 UITableView 編輯模式的按鈕，這邊先在 **AppDelegate.swift** 設置 **導覽控制器 UINavigationController**，後續便可在導覽列左右設置按鈕。

首先在 **viewDidLoad()** 中設置基本設定、UITableView 及導覽列按鈕：

```
// 基本設定
let fullsize = UIScreen.mainScreen().bounds.size
self.view.backgroundColor = UIColor.whiteColor()
self.title = "編輯模式"
self.navigationController?.navigationBar.translucent =
false

// 建立 UITableView
myTableView = UITableView(frame: CGRect(
    x: 0, y: 0,
    width: fullsize.width,
    height: fullsize.height - 64),
    style: .Plain)
myTableView.registerClass(
    UITableViewCell.self, forCellReuseIdentifier: "Cell")
myTableView.delegate = self
myTableView.dataSource = self
myTableView.allowsSelection = true
self.view.addSubview(myTableView)

// 導覽列左邊及右邊按鈕 編輯 & 新增
myTableView.setEditing(true, animated: false)
self.editBtnAction()
```

上述將導覽列按鈕另外設置在一個方法 `editBtnAction()` 中，以供切換編輯模式時可以一併變動。

切換編輯模式

按下編輯按鈕來切換編輯模式，如下：

```
// 按下編輯按鈕時執行動作的方法
func editBtnAction() {
    myTableView.setEditing(
        !myTableView.editing, animated: true)
    if (!myTableView.editing) {
        // 顯示編輯按鈕
        self.navigationItem.leftBarButtonItem =
            UIBarButtonItem(barButtonSystemItem: .Edit,
                            target: self,
                            action:
                                #selector(ViewController.editBtnAction))

        // 顯示新增按鈕
        self.navigationItem.rightBarButtonItem =
            UIBarButtonItem(barButtonSystemItem: .Add,
                            target: self,
                            action:
                                #selector(ViewController.addBtnAction))
    } else {
        // 顯示編輯完成按鈕
        self.navigationItem.leftBarButtonItem =
            UIBarButtonItem(barButtonSystemItem: .Done,
                            target: self,
                            action:
                                #selector(ViewController.editBtnAction))

        // 隱藏新增按鈕
        self.navigationItem.rightBarButtonItem = nil
    }
}
```

上述程式根據 UITableView 的屬性 `editing` 來判斷目前是否處於編輯模式，並使用方法 `setEditing(_:animated:)` 來切換。切換的同時也一併設置新的左右兩邊的按鈕，只有非編輯模式時才有新增的按鈕。

要設置每筆 row 是否可以進入編輯模式，必須實作以下這個委任方法：

```
// 各 cell 是否可以進入編輯狀態 及 左滑刪除
func tableView(tableView: UITableView,
  canEditRowAtIndexPath indexPath: NSIndexPath)
-> Bool {
  return true
}
```

你可以根據 indexPath 來讓特定的 section 的 row 不能編輯。

新增一筆資料

在非編輯模式時，可以按導覽列右邊按鈕新增一筆資料，如下：

```
// 按下新增按鈕時執行動作的方法
func addBtnAction() {
  print("新增一筆資料")
  info.insert("new row", atIndex: 0)

  // 新增 cell 在第一筆 row
  myTableView.beginUpdates()
  myTableView.insertRowsAtIndexPaths(
    [NSIndexPath(forRow: 0, inSection: 0)],
    withRowAnimation: .Fade)
  myTableView.endUpdates()
}
```

上述程式先為示範的陣列新增一筆資料，再在 UITableView 的兩個方法 `beginUpdates()` 與 `endUpdates()` 中間使用方法 `insertRowsAtIndexPaths(_:_withRowAnimation:)` 新增一筆資料。

排序切換位置

必須實作下面這個委任方法，才會出現排序功能：

```
// 編輯狀態時 拖曳切換 cell 位置後執行動作的方法
// (必須實作這個方法才會出現排序功能)
func tableView(tableView: UITableView,
```

```
moveRowAtIndexPath sourceIndexPath: NSIndexPath,
toIndexPath destinationIndexPath: NSIndexPath) {
    print("\(sourceIndexPath.row) to")
    print("\(destinationIndexPath.row)")

    var tempArr:[String] = []

    if(sourceIndexPath.row > destinationIndexPath.row)
    { // 排在後的往前移動
        for (index, value) in info.enumerate() {
            if index < destinationIndexPath.row
            || index > sourceIndexPath.row {
                tempArr.append(value)
            } else if
                index == destinationIndexPath.row {
                tempArr.append(info[sourceIndexPath.row])
            } else if index <= sourceIndexPath.row {
                tempArr.append(info[index - 1])
            }
        }
    } else if (sourceIndexPath.row <
    destinationIndexPath.row) {
        // 排在前的往後移動
        for (index, value) in info.enumerate() {
            if index < sourceIndexPath.row
            || index > destinationIndexPath.row {
                tempArr.append(value)
            } else if
                index < destinationIndexPath.row {
                    tempArr.append(info[index + 1])
            } else if
                index == destinationIndexPath.row {
                    tempArr.append(info[sourceIndexPath.row])
            }
        }
    } else {
        tempArr = info
    }

    info = tempArr
```

```
    print(info)  
}
```

上述程式除了重新排序 UITableView 之外，也必須將示範的陣列重新排序。

刪除資料

編輯模式下刪除或是左滑刪除時，會執行以下方法：

```
// 編輯狀態時 按下刪除 cell 後執行動作的方法  
// (另外必須實作這個方法才會出現左滑刪除功能)  
func tableView(tableView: UITableView,  
    commitEditingStyle editingStyle:  
    UITableViewCellEditingStyle, forRowAtIndexPath  
indexPath: NSIndexPath) {  
    let name = info[indexPath.row]  
  
    if editingStyle == .Delete {  
        info.removeAtIndex(indexPath.row)  
  
        tableView.beginUpdates()  
        tableView.deleteRowsAtIndexPaths(  
            [indexPath], withRowAnimation: .Fade)  
        tableView.endUpdates()  
  
        print("刪除的是 \(name)")  
    }  
}
```

除了刪除 UITableView 的資料外，也必須刪除示範陣列所屬的值。

以上即為本小節的示範內容。

範例

本小節範例程式碼放在 [apps/todo](#)

遊玩臺北

本節會介紹如何建立一個取得外部 **API** 資訊並儲存與顯示的應用程式，主要使用 [Data.Taipei 臺北市政府資料開放平台](#) 的 API 資料作為示範，這個專案會依序取得臺北市的景點、公園、廁所與住宿資訊。並且利用 iPhone 的定位功能，依照距離由近至遠排序。

你可以在 App Store 中找到這個應用程式，名稱為 [遊玩臺北](#) 。

以下列出會使用到的 UIKit 元件與功能，如果還有尚未了解的地方，可以先往前面章節複習一下：

- 文字標籤 `UILabel`
- 按鈕 `UIButton`
- 進度條 `UIProgressView`
- 表格 `UITableView`
- 導覽控制器 `UINavigationController`
- 標籤列控制器 `UITabBarController`
- 儲存資訊 `NSUserDefaults`
- `SQLite`

除了上述項目外，因應此應用程式的需要，後面幾小節會額外介紹需要的功能：

- 取得遠端 API 資料並儲存
- 地圖與定位

範例

本節範例程式碼放在 [apps/taipeitravel](#)

規劃與實作

請先在 Xcode 打開這個專案([apps/taipeitravel/TaipeiTravel](#))以供後續與文章內容比對檢視，本小節說明僅會提示部分內容，不會將所有程式碼都寫出來，請以專案程式碼為主。

首先介紹遊玩臺北應用程式可以操作的動作，先看下圖流程：



上圖由 UITabBarController 的第一個 Tab 景點為應用程式預設起始頁，列表中會顯示與目前定位位置的距離(如果有授權定位功能)，點擊項目後進入這個地點的詳細頁，會列出相關的資訊，當 API 有提供這個地點的位置時，則可以點擊第一列地圖以進入地圖頁以進入地圖頁。

其餘三個公園、廁所及住宿的運作方式與景點相同。

第五個 Tab 關於頁則是可以前往支援網頁與來源網頁，並提供簡單說明。

前置作業

首先在 Xcode 裡，新建一個 **Single View Application** 類型的專案，取名為 **TaipeiTravel** 。

一開始先依照 [程式之外的設定](#) 的步驟設定好。以及將需要的 Swift 檔案、圖示檔案都先加入至專案中：

- BaseMainViewController.swift
- DetailViewController.swift
- MapViewController.swift
- HotelMainViewController.swift
- HotelDetailViewController.swift
- LandmarkMainViewController.swift
- LandmarkDetailViewController.swift
- ParkMainViewController.swift
- ParkDetailViewController.swift
- ToiletMainViewController.swift
- ToiletDetailViewController.swift
- InfoMainViewController.swift
- icons 目錄中的五個圖示檔案

AppDelegate.swift

在進入應用程式的畫面前，需要在 `AppDelegate.swift` 中先做些設定：

- 向使用者要求定位功能權限。(請參考[地圖與定位](#)內容。)
- 設置導覽列的一些預設樣式。
- 依照[標籤列控制器 UITabBarController](#)的說明，將根視圖設為一個 `UITabBarController`，接著再設置每個 Tab 的根視圖控制器為 `UINavigationController`，最後依序為每個 Tab 設置 `LandmarkMainViewController`, `ParkMainViewController`, `ToiletMainViewController`, `HotelMainViewController` 為首頁。

此專案結合[標籤列控制器 UITabBarController](#)與[導覽控制器 UINavigationController](#)，所以必須先設置 `UITabBarController` 再設置 `UINavigationController`，以達到有標籤列也有導覽列的功能。

景點首頁

因為前四個 Tab 內容差不多，這邊以景點頁面作範例，其餘三個則是類似的方式進行。

Coordinate.swift

在進入到首頁之前，先新增一個用來依照距離遠近排序景點的檔案。先以[新增檔案](#)的方式加入這個檔案，檔案類型要選擇 `iOS > Source > Swift File`。

因為會使用到定位功能，所以記得先引入定位函式庫：

```
import CoreLocation
```

接著新增一個名為 `Coordinate` 的結構(`Struct`)，有三個屬性，依序為景點的索引值、緯度及經度，如下：

```
struct Coordinate {
    var index: Int
    var latitude: Double
    var longitude: Double
}
```

然後為這個結構 `Coordinate` 擴展 `Comparable` 協定(`Comparable` 是一個 Swift 內建的協定，常見型別如 `Int, Double` 都有實作)，讓型別為 `Coordinate` 的變數可以互相比較：

```
extension Coordinate: Comparable {}
```

當擴展 `Comparable` 協定時，必須實作下列兩個方法(`==` 為比較兩者是否相等，`<` 則是比較前者是否小於後者)，才可以讓變數彼此比較：

```
func ==(a: Coordinate, b: Coordinate) -> Bool {
    let myUserDefaults =
        UserDefaults.standard

    // 是否取得定位權限
    let locationAuth =
        myUserDefaults.object(forKey: "locationAuth")
        as? Bool

    if locationAuth != nil && locationAuth! {
        // 取得目前使用者座標
        let userLatitude =
```

```

        myUserDefaults.objectForKey("userLatitude")
        as? Double
    let userLongitude =
        myUserDefaults.objectForKey("userLongitude")
        as? Double
    let userLocation = CLLocation(
        latitude: userLatitude!,
        longitude: userLongitude!)

    // 兩點的座標
    let aLocation = CLLocation(
        latitude: a.latitude,
        longitude: a.longitude)
    let bLocation = CLLocation(
        latitude: b.latitude,
        longitude: b.longitude)

    return
    aLocation.distanceFromLocation(userLocation)
    ==
    bLocation.distanceFromLocation(userLocation)
} else {
    return a.index == b.index
}

}

func <(a: Coordinate, b: Coordinate) -> Bool {
    let myUserDefaults =
        NSUserDefaults.standardUserDefaults()

    // 是否取得定位權限
    let locationAuth =
        myUserDefaults.objectForKey("locationAuth")
        as? Bool

    if locationAuth != nil && locationAuth! {
        // 取得目前使用者座標
        let userLatitude =
            myUserDefaults.objectForKey("userLatitude")

```

```

        as? Double
    let userLongitude =
        myUserDefaults.objectForKey("userLongitude")
            as? Double
    let userLocation = CLLocation(
        latitude: userLatitude!,
        longitude: userLongitude!)

    // 兩點的座標
    let aLocation = CLLocation(
        latitude: a.latitude,
        longitude: a.longitude)
    let bLocation = CLLocation(
        latitude: b.latitude,
        longitude: b.longitude)

    return
        aLocation.distanceFromLocation(userLocation)
        < bLocation.distanceFromLocation(userLocation)
    } else {
        return a.index < b.index
    }
}

```

上述兩個實作的方法中，會先取得 iPhone 目前所在的定位位置，再與兩個型別為 Coordinate 的變數分別以 distanceFromLocation() 方法取得距離，再以這個距離判斷哪一個景點較近，後續會再以索引值取得景點的其餘資訊。

BaseMainViewController.swift

這邊將重複的動作寫在繼承自 UIViewController 的 BaseMainViewController 中，再讓四個首頁繼承自他。

先注意以下 BaseViewController 的幾個屬性：

- 建立屬性 refreshDays，限制向遠端 API 要求資料的次數，超過這個天數才會再次取得新的資訊。
- 建立屬性 apiDataAll 存放取得的所有景點資訊。
- 建立屬性 apiData 存放依距離較近排序的有限景點資訊，以簡化顯示內容。

- 建立屬性 `apiDataForDistance` 存放景點的經緯度資訊，以供依距離遠近排序的功能。

viewWillAppear()

在 `viewWillAppear()` 方法中，先確認是否取得定位權限，再依據是否有定位位置來排序景點資料。(當無定位權限時，會依照 API 提供的排序列出所有資料，有定位權限時，則是只列出較近距離的景點。)

viewDidDisappear()

離開這個頁面時，要記得停止定位自身位置。

其他方法

`addData()` 方法是用來區分是否要取得遠端 API 資料，這邊會限制一個天數，如果超過這個天數才會重新取得資料，否則就是直接將已經存在的 JSON 檔案拿出來使用。

如果需要向遠端 API 取得資料，則是使用 `normalGet()` 方法來取得(請參考[取得遠端 API 資料並儲存](#))，在委任方法中會執行下載完成後的動作，因為會以一個閉包(`closure`)執行，所以必須使

用 `dispatch_async(dispatch_get_main_queue(), /* add table */)` 來以主執行緒更新 `UITableView`，如下：

```

// 下載完成
func URLSession(session: NSURLSession,
    downloadTask: NSURLSessionDownloadTask,
    didFinishDownloadingToURL location: NSURL) {
    let targetUrl = NSURL(string: self.targetUrl)!
    let data = NSData(contentsOfURL: location)
    if ((data?.writeToURL(
        targetUrl, atomically: true)) != nil) {
        print("普通獲取遠端資訊的方式：儲存資訊成功")

        // 更新獲取資料的日期
        self.myUserDefaults.setObject(
            self.todayDateInt,
            forKey: "\(\self.fetchType)FetchDate")
        self.myUserDefaults.synchronize()

        dispatch_async(dispatch_get_main_queue(), {
            self.addTable(self.targetUrl)
        })
    } else {
        print("普通獲取遠端資訊的方式：儲存資訊失敗")
    }
}

```

在 `addTable()` 方法中，會先以 `jsonParse()` 方法解析取得的 JSON 檔案，並將資料存在屬性 `apiDataAll`。

接著在 `refreshAPIData()` 方法中，以 `reloadAPIData()` 方法來將屬性 `apiDataAll` 的內容依據限制再轉存至屬性 `apiData` 及設置另外的屬性 `apiDataForDistance` (有定位權限時，依距離遠近排序有限數目。無權限時則是全部內容。)。

其中屬性 `apiDataForDistance` 是一個型別為 `[Coordinate]!` 的陣列，這個陣列會在 `fillIntoAPIDataForDistanceAndSort()` 方法中使用 `apiDataForDistance.sortInPlace(<)` 方法依照距離排序內部成員(請參考稍前介紹的 [Coordinate.swift](#) 內容)，以供後續顯示在 `UITableView` 中。

UITableView 的委任方法

`tableView(tableView: UITableView, cellForRowAtIndexPath indexPath: NSIndexPath) -> UITableViewCell` 方法是用來顯示每筆事項的內容，一開始會先以 `apiDataForDistance[indexPath.row].index` 取得已經依距離遠近排序的索引值，再以這個索引值到景點資料陣列 `apiData` 中取得目前要顯示的景點資料，如下：

```
let thisData =  
    apiData[apiDataForDistance[indexPath.row].index]
```

取得景點資料之後，再作顯示名稱、距離多遠的後續處理。

LandmarkMainViewController.swift

繼承自 `BaseMainViewController` 的 `LandmarkMainViewController` 只需要再將前往細節頁的部份加上就差不多了。

在 `goDetail()` 方法中，會先取得目前所點擊的景點資訊，再將細節頁要顯示的資訊存在 `NSUserDefaults` 中，便可以前往細節頁。

景點 細節頁

細節頁也是如同首頁，將重複的動作寫在一隻檔案中，再各自繼承自他。

DetailViewController.swift

先注意以下 `DetailViewController` 的屬性：

- 建立屬性 `hasMap`，來區分目前這筆景點資訊有沒有經緯度資訊，有的話才可以進入地圖頁。

UITableView 的委任方法

`tableView(tableView: UITableView, didSelectRowAtIndexPath indexPath: NSIndexPath)` 方法負責點擊項目時的動作。這部份加上一個點擊時可展開顯示全部文字內容的功能。

LandmarkDetailViewController.swift

繼承自 `DetailViewController` 的 `LandmarkDetailViewController` 將要顯示的資料從 `NSUserDefaults` 取回並填入陣列中即可。

地圖頁

這頁會使用到地圖功能，所以要先引入地圖函式庫(請參考[地圖與定位](#)內容。)：

```
import MapKit
```

在 `viewDidLoad()` 中取得景點資訊，並在地圖上顯示景點的位置，以及使用者自己的位置(如果有取得定位授權的話)。

公園、廁所與住宿頁面與上述內容類似，便不再重複介紹。

關於頁

這頁使用 `UITableView` 的 `.Grouped` 特性各別列出不同的功能：

1. 前往外部 **Facebook** 網頁。
2. 前往外部 資料與圖示資源 網頁。
3. 應用程式說明。

前往外部網頁時，使用下列方式：

```
let requestUrl = NSURL(string:  
    "https://www.facebook.com/1640636382849659")  
UIApplication.sharedApplication().openURL(requestUrl!)
```

範例

此應用程式範例程式碼放在 [apps/taipeitravel/TaipeiTravel](#)

你也可以在 App Store 中找到這個應用程式，名稱為遊玩臺北 。

取得遠端 API 資料並儲存

這一小節會介紹如何取得遠端 API 的資料，並將資料儲存成本地檔案，以供後續使用。

我們使用 [Data.Taipei 臺北市政府資料開放平台](#) 的 API 資料作為示範，這個平台開放的資料有很多項，這邊以取得[臺北市臺北旅遊網-住宿資料\(中文\)](#)及[臺北市臺北旅遊網-景點資料\(中文\)](#)作為示範。

進入資料頁後，點擊 **使用資料 > API**，如下圖：



接著會進入到 API 說明頁，看到下圖標示起來的網址，便是這個 API 的資料：

「臺北市臺北旅遊網-住宿資料(中文)」API存取

請使用資料集ID與檔案之RID存取，存取方式請參考[開發指南](#)

資料集ID
58093ba6-4c98-4148-b27a-50ad97d7afca

範例
<http://data.taipei/opendata/datalist/apiAccess?scope=datasetMetadataSearch&q=id:58093ba6-4c98-4148-b27a-50ad97d7afca>

臺北市臺北旅遊網-住宿資料(中文)RID
6f4e0b9b-8cb1-4b1d-a5c4-febd90f62469

範例
<http://data.taipei/opendata/datalist/apiAccess?scope=resourceAquire&rid=6f4e0b9b-8cb1-4b1d-a5c4-febd90f62469>

將上圖中標示的網址以瀏覽器開啟後，會發現取得的資料是 JSON 格式，可以使用桌機的 Chrome 瀏覽器的擴充套件 [JSONView](#) 來將這個 JSON 資料結構化呈現在畫面中，比較方便檢視內容，如下圖：



```
{
  - result: {
    offset: 0,
    limit: 10000,
    count: 517,
    sort: "",
    - results: [
      - {
        _id: "1",
        RowNumber: "1",
        REF_WP: "6",
        CAT1: "住宿",
        CAT2: "一般旅館",
        SERIAL_NO: "B0487",
        MEMO_TIME: null,
        stitle: "WESTGATE HOTEL 永安樓",
        xbody: null,
        avBegin: "2008/10/20",
        avEnd: "2015/09/09",
        idpt: "臺北旅遊網",
        address: "臺北市萬華區中華路一段148號、148之1號及150號1至10樓",
        xpostDate: "2015/09/09",
        file: null,
        langinfo: "10",
        POI: "Y",
        info: null,
        longitude: "121.50808",
        latitude: "25.04294",
        MRT: null,
        { "id": "2", "RowNumber": "2", "REF_WP": "6", "CAT1": "住宿", "CAT2": "一般旅館", "S
        SERIAL_NO: "B0504",
        MEMO_TIME: null,
        stitle: "一九一旅店",
        xbody: null,
        avBegin: "2008/10/20",
        avEnd: "2016/05/09",
        idpt: "臺北旅遊網",
        address: "臺北市大同區南京西路165號6樓",
        xpostDate: "2016/05/09",
        file: null,
        langinfo: "10",
        POI: "Y",
        info: null,
        longitude: "121.512993",
        latitude: "25.054084",
        MRT: null,
        { "id": "3", "RowNumber": "3", "REF_WP": "6", "CAT1": "住宿", "CAT2": "一般旅館", "S
        SERIAL_NO: "B0151",
        MEMO_TIME: null,
        stitle: "一加一旅館",
        xbody: null,
        avBegin: "2008/10/20",
        avEnd: "2015/09/09",
        idpt: "臺北旅遊網",
        address: "臺北市萬華區昆明街34號3樓",
        xpostDate: "2015/09/09",
        file: "http://www.travel.taipei/d_upload_ttn/fronosite/tw/hotel/B0151/B0151_1.jpghttp://www.travel.taipei/d_upload_ttn/fronosite/tw/hotel/B0151/B0151_2.jpghttp://www.travel.taipei/d_upload_ttn/fronosite/tw/hotel/B0151/B0151_3.jpg",
        langinfo: "10",
        POI: "Y",
        info: null,
        longitude: "121.50808",
        latitude: "25.04294",
        MRT: null
      },
      - {
        _id: "2",
        RowNumber: "2",
        REF_WP: "6",
        CAT1: "住宿",
        CAT2: "一般旅館",
        SERIAL_NO: "B0196",
        MEMO_TIME: null,
        stitle: "一等好旅店",
        xbody: "一等好旅店近饒河夜市、世貿中心、交通相當利。我們以溫馨的服務"
      }
    ]
  }
}
```

以上為前置作業的介紹，接著會開始介紹如何在應用程式中獲取遠端 API 資料，並將這資料儲存為本地的一個 JSON 檔案，最後會說明如何解析這個 JSON 檔案並讀取其內資訊。

Hint

- JSON 是一種輕量級的資料交換格式，以純文字為基礎來儲存與傳送結構資料，可以經由特定的格式儲存任何文字資料(像是字串、數字、陣列或物件)，JSON 可以讓你很簡單的與其他程式交換資料。
- 有些 API 資料會需要事先向擁有者申請 ID，以便在接收資料前辨別獲取資料者的身分。

首先在 Xcode 裡，新建一個 [Single View Application](#) 類型的專案，取名為 ExFetchDataAndStorage 。

一開始先為 ViewController 建立三個屬性：

```
class ViewController: UIViewController {  
    var taipeiDataUrl :String!  
    var documentsPath :String!  
    var touringSiteTargetUrl :String!  
  
    // 省略  
}
```

以及在 `viewDidLoad()` 中設置儲存檔案的目錄路徑與 API 網址，以供後續使用，如下：

```
// 應用程式儲存檔案的目錄路徑  
let urls =  
    FileManager.defaultManager().URLsForDirectory(  
        .DocumentDirectory, inDomains: .UserDomainMask)  
self.documentsPath = urls[urls.count-1].absoluteString  
  
self.taipeiDataUrl =  
    "http://data.taipei/opendata/datalist"  
    + "/apiAccess?scope=resourceAquire&rid="
```

獲取遠端 API 資料

應用程式中要與遠端交換資料必須使用 `NSURLSession` 相關函式庫，這邊會介紹兩種方式。

在介紹如何獲取資料之前，請先了解 iOS 9 之後預設為只能載入 `https` 的網頁(也就是加密過的)，要如何設定成可開啟 `http` 網頁請參考前面章節[無法載入 http 的網址](#)的說明。

基本獲取遠端資訊方式

先介紹基本獲取方式，這種方式沒有用到委任模式，會單純的下載遠端檔案下來以供使用，下面將其寫在一個方法中：

```
func simpleGet(myUrl :String, targetPath :String) {  
    if let url = NSURL(string: myUrl) {  
  
        NSURLSession.sharedSession()  
            .dataTaskWithURL(url) {  
                data, response, error in  
  
                    print(  
                        NSString(  
                            data: data!,  
                            encoding: NSUTF8StringEncoding))  
  
            }.resume()  
  
    }  
}
```

上述程式中，先使用 `NSURLSession.sharedSession()` 獲得一個共用的 `NSURLSession` 實體以供連線，接著帶入要接收資料的網址 `url` 到 `dataTaskWithURL()` 方法，最後帶一個閉包來處理獲得的資料。

請注意到後面還接了一個方法 `resume()`，因為這個連線必須手動執行，所以在設置完後必須接著使用方法 `resume()` 來送出連線。

閉包的第一個參數 `data` 便為獲得的資訊，這邊先轉成字串印出來，後續會再做更多處理。

普通獲取遠端資訊方式

如果想要在下載資料的各個階段執行動作，就需要實作委任方法，首先為 `ViewController` 加上委任模式需要遵循的協定：

```
class ViewController: UIViewController,  
    NSURLSessionDelegate, NSURLSessionDownloadDelegate {  
  
    // 省略  
}
```

接著是可以實作的委任方法：

```
// 下載完成
func URLSession(
    session: URLSession,
    downloadTask: URLSessionDownloadTask,
    didFinishDownloadingToURL location: NSURL) {
    print("下載完成")
}

// 下載過程中
func URLSession(
    session: URLSession,
    downloadTask: URLSessionDownloadTask,
    didWriteData bytesWritten: Int64,
    totalBytesWritten: Int64,
    totalBytesExpectedToWrite: Int64) {
    // 如果 totalBytesExpectedToWrite 一直為 -1
    // 表示遠端主機未提供完整檔案大小資訊
    print("下載進度： \(totalBytesWritten)")
    print("/\(totalBytesExpectedToWrite)")
}
```

最後將獲取方式寫在一個方法中：

```
// 普通獲取遠端資訊的方式
func normalGet(myUrl :String) {
    if let url = NSURL(string: myUrl) {
        // 設置為預設的 session 設定
        let sessionWithConfigure =
            NSURLSessionConfiguration.
            defaultSessionConfiguration()

        // 設置委任對象
        let session = NSURLSession(
            configuration: sessionWithConfigure,
            delegate: self,
            delegateQueue: nil)

        // 設置遠端 API 網址
        let dataTask =
            session.downloadTaskWithURL(url)

        // 執行動作
        dataTask.resume()
    }
}
```

上述程式首先使用 `NSURLSessionConfiguration` 設置一個 `session` 的設定，並使用 `defaultSessionConfiguration()` 設置為預設模式。另外還可以使用 `ephemeralSessionConfiguration()`，這個模式不會將連線中的快取、`Cookie` 或認證資訊做儲存，就像是瀏覽器的隱私模式。或是使用 `backgroundSessionConfiguration()`，讓應用程式被切換到背景時仍然可以執行連線工作。

接著使用 `NSURLSession(configuration:delegate:delegateQueue:)` 設置一個 `NSURLSession` 實體(相較於基本獲取方式的共用實體，這邊設置為一個新的 `NSURLSession` 實體。)，參數傳入前面設置的 `session` 設定，以及設置委任對象。設置委任對象後，在下載過程中與完成時，都會執行委任方法。

最後使用 `downloadTaskWithURL()` 填入遠端 API 網址，及執行動作 `resume()`。

執行獲取資訊

在 `viewDidLoad()` 中，執行獲取兩個示範 API 資料，分別使用前面介紹的基本獲取資訊方式與普通獲取資訊方式：

```
// 台北住宿資料 中文
let strHotelID =
    "6f4e0b9b-8cb1-4b1d-a5c4-febd90f62469"
self.simpleGet(taipeiDataUrl + strHotelID,
    targetPath: self.documentsPath + "hotel.json")

// 台北景點資料 中文
let strTouringSiteID =
    "36847f3f-deff-4183-a5bb-800737591de5"
self.touringSiteTargetUrl =
    self.documentsPath + "touringSite.json"
self.normalGet(taipeiDataUrl + strTouringSiteID)
```

以上便會開始獲取遠端資料。方法 `simpleGet()` 的第二個參數 `targetPath` 後續會再做說明使用。

儲存為本地檔案

在前面順利獲得資料後，接著將資料存成本地檔案以供後續使用。

首先是稍前建立的方法 `simpleGet()`，獲得的資料 `data` 為 `NSData` 型別，以下為儲存方式：

```
// 建立檔案
let fileurl = NSURL(string: targetPath)
if let result = data?.writeToURL(fileurl!, atomically: true) {
    if result {
        print("簡單方式獲取遠端資訊：儲存資訊成功")
    } else {
        print("簡單方式獲取遠端資訊：儲存資訊失敗")
    }
}
```

方法 `simpleGet()` 傳入的第二個參數 `targetPath` 型別為 `String`，所以先將其轉為 `NSURL`，並帶入型別為 `NSData` 的閉包參數 `data` 的方法 `writeToURL()`，以建立一個新的本地檔案。會返回一個 `Bool?` 的值表示儲存成功或失敗。

接著是方法 `normalGet()`，會在下載完成的委任方法中獲得資料 `location`，其型別為 `NSURL`，是一個本地的暫存檔案路徑，以下為儲存方式：

```
let targetUrl = NSURL(  
    string: self.touringSiteTargetUrl)!  
let data = NSData(contentsOfURL: location)  
if ((data?.writeToURL(targetUrl, atomically: true))  
!= nil) {  
    print("普通獲取遠端資訊的方式：儲存資訊成功")  
} else {  
    print("普通獲取遠端資訊的方式：儲存資訊失敗")  
}
```

使用一開始設置的屬性 `touringSiteTargetUrl` 來生成一個 `NSURL`，用來表示新的本地檔案路徑。接著將 `location` 轉為型別 `NSData` 的資料後，再以方法 `writeToURL()` 建立一個新的本地檔案。

以上如果都順利儲存成功，會在本地的 `Documents` 目錄中，分別建立 `hotel.json` 與 `touringSite.json` 檔案。

解析 JSON 檔案

前面建立好兩個 JSON 檔案後，必須再將其做解析以取得其內的資料。首先使用先前介紹的瀏覽器擴充套件檢視一下這個 JSON 內容：

```
{
  - result: {
    offset: 0,
    limit: 10000,
    count: 517,
    sort: "",
    - results: [
      - {
        _id: "1",
        RowNumber: "1",
        REF_WP: "6",
        CAT1: "住宿",
        CAT2: "一般旅館",
        SERIAL_NO: "B0487",
      }
    ]
  }
}
```

可以發現格式如下：

```
{
  result: {
    offset: 0,
    limit: 10000,
    count: 517,
    sort: "",
    results: [
      // 省略
    ]
  }
}
```

最外層可以轉換成一個字典(Dictionary)，其內只有一筆資料， key 值為 `result`，對應著其內的資料也是一個字典，其內有五筆資料，代表意思分別為：

- `offset`：獲取資料的偏移量，如果設置為 3，則表示獲取資料要跳過前面 3 筆，從第四筆開始取得。
- `limit`：獲取資料的最多數量，如果設置為 10，則最多只會取得 10 筆資料。
- `count`：全部的資料數量。
- `sort`：排序方式，與 SQL 指令類似，如果設置為 "id asc, RowNumber desc"，則是以 `id` 從小到大排序，以及以 `RowNumber` 從大到小排序。
- `results`：獲取的資料，會依照前面設置的設定取得資料。

如果要使用這些功能，可以在稍前提到的 API 網址後面加上，像是要設置 offset 為 5 以及 limit 為 10，則是在該網址後面加上 `&offset=5&limit=10` 即可。

依照上述的格式，可以將其轉換為一個型別為 `[String:[String:AnyObject]]` 的字典，以供後續使用。接著這邊將解析 JSON 的功能寫在一個方法中，如下：

```
// 解析 json 檔案
func jsonParse(url :NSURL) {
    do {
        let dict =
            try NSJSONSerialization
                .JSONObjectWithData(
                    NSData(contentsOfURL: url)!,
                    options:
                        NSJSONReadingOptions.AllowFragments)
        as! [String:[String:AnyObject]]

        print(dict.count)

        let dataArr =
            dict["result"]!["results"] as! [AnyObject]

        print(dataArr.count)

        print(dataArr[3]["stitle"])

    } catch {
        print("解析 json 失敗")
    }
}
```

上述程式使用 `NSJSONSerialization.JSONObjectWithData()` 來解析 JSON 檔案，因為設計為一個拋出函式，所以使用 `do-catch` 語句來定義錯誤的捕獲。

以上即為這小節範例的內容。

範例

本節範例程式碼放在 [apps/taipeitravel](#)

地圖與定位

這一小節會介紹如何將地圖放到畫面中，並加上景點的地點標示，另外還會說明如何開啟定位，讓你可以看到目前在地圖上的自身位置。

因為模擬器的定位功能有時會出現問題，所以這小節的範例推薦以實機來執行與測試。

首先在 Xcode 裡，新建一個 **Single View Application** 類型的專案，取名為 ExMap。建立好專案後再以[加入檔案](#)的方式加入一張定位圖示的圖片。

因為預設的應用程式沒有地圖及定位功能，所以一開始必須先將這兩個函式庫引入，分別為 `MapKit` 及 `CoreLocation`：

```
import MapKit  
import CoreLocation
```

再來為 `ViewController` 建立兩個屬性，以供後續使用，如下：

```
class ViewController: UIViewController {  
    var myLocationManager :CLLocationManager!  
    var myMapView :MKMapView!  
  
    // 省略  
}
```

定位 **CoreLocation**

設置定位功能的步驟簡單介紹如下：

1. 建立獲得定位資訊的變數，並設置屬性。
2. 設置委任方法以獲得定位資訊。
3. 向使用者取得定位權限。
4. 設置應用程式需要的定位服務規則。
5. 開始與結束更新定位位置。

以下會如步驟所述依序介紹。

首先在 `viewDidLoad()` 中使用類別 `CLLocationManager()` 增加定位功能，如下：

```
// 建立一個 CLLocationManager  
myLocationManager = CLLocationManager()  
  
// 設置委任對象  
myLocationManager.delegate = self  
  
// 距離篩選器 用來設置移動多遠距離才觸發委任方法更新位置  
myLocationManager.distanceFilter =  
    kCLLocationAccuracyNearestTenMeters  
  
// 取得自身定位位置的精確度  
myLocationManager.desiredAccuracy =  
    kCLLocationAccuracyBest
```

上述程式中的 `distanceFilter` 與 `desiredAccuracy` 這兩個屬性，都與精確度有關，可以設置的值如下：

- `kCLLocationAccuracyBestForNavigation`：精確度最高，適用於導航的定位。
- `kCLLocationAccuracyBest`：精確度高。
- `kCLLocationAccuracyNearestTenMeters`：精確度 10 公尺以內。
- `kCLLocationAccuracyHundredMeters`：精確度 100 公尺以內。
- `kCLLocationAccuracyKilometer`：精確度 1 公里以內。
- `kCLLocationAccuracyThreeKilometers`：精確度 3 公里以內。

或是你想自己設置數值的大小，也可以設置為一個浮點數，單位為公尺。

這邊將定位功能設置好，稍後會在取得定位權限後，再開始定位自身位置。

定位功能可以設置委任對象，接著先介紹委任方法的實作。

委任方法

先為委任對象(也就是 `ViewController`)加上委任模式需要遵循的協定：

```
class ViewController: UIViewController,  
    CLLocationManagerDelegate {  
    // 省略  
}
```

以及在 `ViewController` 中實作的委任方法：

```
func locationManager(manager: CLLocationManager,  
    didUpdateLocations locations: [CLLocation]) {  
    // 印出目前所在位置座標  
    let currentLocation :CLLocation =  
        locations[0] as CLLocation  
    print("\(currentLocation.coordinate.latitude)")  
    print(", \(currentLocation.coordinate.longitude)")  
}
```

依照稍前設置的屬性 `distanceFilter` 的距離精確度，會在定位發生變化時執行上述這個方法，其中參數會獲得目前定位的資訊 `CLLocation`，裡面會有像是緯度(`latitude`)與經度(`longitude`)的數值資訊。

授權定位權限

要使用定位功能，必須向使用者額外取得定位的權限，這邊將詢問授權的動作寫在 `ViewController` 的 `viewDidAppear(_:)` 方法中，每次進到這頁面時都會確認權限，以免在多頁面的應用程式中，使用者又再把權限關閉：

```
override func viewDidAppear(animated: Bool) {
    super.viewDidAppear(animated)

    // 首次使用 向使用者詢問定位自身位置權限
    if CLLocationManager.authorizationStatus() == .NotDetermined {
        // 取得定位服務授權
        myLocationManager.requestWhenInUseAuthorization()

        // 開始定位自身位置
        myLocationManager.startUpdatingLocation()
    }

    // 使用者已經拒絕定位自身位置權限
    else if CLLocationManager.authorizationStatus() == .Denied {
        // 提示可至[設定]中開啟權限
        let alertController = UIAlertController(
            title: "定位權限已關閉",
            message: "如要變更權限，請至 設定 > 隱私權 > 定位服務 開啓",
            preferredStyle: .Alert)
        let okAction = UIAlertAction(
            title: "確認", style: .Default, handler: nil)
        alertController.addAction(okAction)
        self.presentViewController(
            alertController,
            animated: true, completion: nil)
    }

    // 使用者已經同意定位自身位置權限
    else if CLLocationManager.authorizationStatus() == .AuthorizedWhenInUse {
        // 開始定位自身位置
        myLocationManager.startUpdatingLocation()
    }
}
```

上述程式使用 `CLLocationManager.authorizationStatus()` 來確認目前的授權狀態為何。

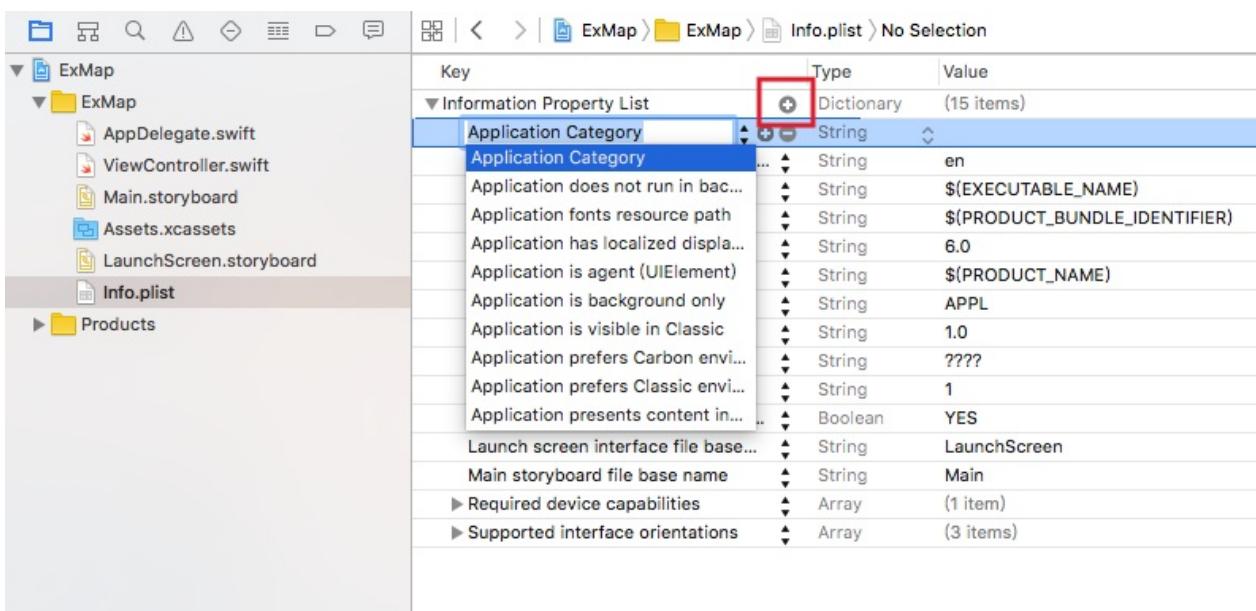
當其等於 `.NotDetermined` 時，則為首次詢問授權，便使用方法 `requestWhenInUseAuthorization()` 來取得授權，畫面上會出現確認框，詢問使用者是否要授權這個應用程式可以使用定位功能。

等於 `.Denied` 時，則是已詢問過且使用者拒絕提供定位功能，所以這邊會建立一個提示框，提醒使用者如果要使用定位功能，必須至設定中手動打開。

等於 `.AuthorizedWhenInUse` 時，則是已詢問過且使用者同意提供定位功能，所以這邊便以方法 `startUpdatingLocation()` 開始定位自身位置。

增加定位服務規則

除了需要向使用者詢問權限外，還必須在 `Info.plist` 檔案中加入一個值，請先打開左側檔案列表中的 `Info.plist`，並按下加號按鈕增加，如下：



這個欄位必須填入 `NSLocationWhenInUseUsageDescription`，其餘欄位的 Type 使用預設的 String，Value 則留空，如下：

Key	Type	Value
NSLocationWhenInUseUsageDescription	String	
Localization native development region	String	en

開始與結束更新定位位置

以上功能都設置好後，就可以開始更新定位位置，也才會開始執行稍前介紹實作的委任方法。

開始更新定位位置的方式已在稍前詢問權限時一同執行了，使用方法 `startUpdatingLocation()` 來開始定位位置。

結束更新定位位置則是寫在 `ViewController` 的 `viewDidDisappear(_:)` 方法中，可以在多頁面應用程式中，離開當前頁面時使用方法 `stopUpdatingLocation()` 結束，如下：

```
override func viewDidDisappear(animated: Bool) {  
    super.viewDidDisappear(animated)  
  
    // 停止定位自身位置  
    myLocationManager.stopUpdatingLocation()  
}
```

你也可以依照需求把結束更新定位位置寫在不同地方，像是如果只要定位一次的話，可以寫在委任方法中。

不同時機的定位功能

實際上可以授權的定位功能依照定位的時機不同有兩種，分別為開啓應用程式時定位 `WhenInUse` 及永遠定位 `Always`，前者只在開啓應用程式時，才會使用定位功能，後者則是即使未開啓應用程式，也可以在需要的時候獲得定位資訊。

因為定位功能的耗電量大，所以請依照實際需求向使用者要求權限，上述範例都是使用開啓應用程式時定位 `WhenInUse` 的權限，如果要改成永遠定位 `Always`，請注意下述部分：

- 詢問授權的方法從 `requestWhenInUseAuthorization()` 改成 `requestAlwaysAuthorization()`。
- 授權狀態的值從 `.AuthorizedWhenInUse` 改成 `.AuthorizedAlways`。
- 定位服務規則填入的值從 `NSLocationWhenInUseUsageDescription` 改成 `NSLocationAlwaysUsageDescription`。

以上便為定位 `CoreLocation` 的範例介紹，接著會介紹如何使用地圖 `MapKit`，同時也會與定位功能結合，讓你可以得知目前所在位置。

地圖 `MapKit`

設置地圖功能的步驟簡單介紹如下：

1. 建立地圖視圖，並設置屬性。
2. 加入地點圖示。
3. 設置委任方法並自定義大頭針樣式。

以下會如步驟所述依序介紹。

首先在 `viewDidLoad()` 中使用類別 `MKMapView()` 建立一個地圖視圖，如下：

```
// 取得螢幕的尺寸
let fullSize = UIScreen.mainScreen().bounds.size

// 建立一個 MKMapView
myMapView = MKMapView(frame: CGRect(
    x: 0, y: 20,
    width: fullSize.width,
    height: fullSize.height - 20))

// 設置委任對象
myMapView.delegate = self

// 地圖樣式
myMapView.mapType = .Standard

// 顯示自身定位位置
myMapView.showsUserLocation = true

// 允許縮放地圖
myMapView.zoomEnabled = true

// 地圖預設顯示的範圍大小 (數字越小越精確)
let latDelta = 0.05
let longDelta = 0.05
let currentLocationSpan:MKCoordinateSpan =
    MKCoordinateSpanMake(latDelta, longDelta)

// 設置地圖顯示的範圍與中心點座標
let center:CLLocation = CLLocation(
    latitude: 25.05, longitude: 121.515)
let currentRegion:MKCoordinateRegion =
    MKCoordinateRegion(
        center: center.coordinate,
        span: currentLocationSpan)
myMapView.setRegion(currentRegion, animated: true)

// 加入到畫面中
self.view.addSubview(myMapView)
```

上述程式可以看到，地圖可以設置屬性 `mapType` 地圖樣式，除了標準模式 `.Standard`，還可設置衛星模式 `.Satellite` 與混和模式 `.Hybrid` 等等。

屬性 `showsUserLocation` 則是可以在地圖上顯示定位，只要在前一個範例中的詢問定位權限有允許，這個屬性也設為 `true` 時，地圖上就會出現自身定位位置。

接著先以類別 `MKCoordinateRegion()` 設置中心點及範圍，再使用方法 `setRegion()` 來設置地圖顯示的中心點與預設顯示範圍大小。

加入地點圖示

在 `viewDidLoad()` 中建立好地圖視圖後，接著會示範加入兩個地點圖示，如下：

```
// 建立一個地點圖示 (圖示預設為紅色大頭針)
var objectAnnotation = MKPointAnnotation()
objectAnnotation.coordinate = CLLocation(
    latitude: 25.036798,
    longitude: 121.499962).coordinate
objectAnnotation.title = "艋舺公園"
objectAnnotation.subtitle =
    "艋舺公園位於龍山寺旁邊，原名為「萬華十二號公園」。"
myMapView.addAnnotation(objectAnnotation)

// 建立另一個地點圖示 (經由委任方法設置圖示)
objectAnnotation = MKPointAnnotation()
objectAnnotation.coordinate = CLLocation(
    latitude: 25.063059,
    longitude: 121.533838).coordinate
objectAnnotation.title = "行天宮"
objectAnnotation.subtitle =
    "行天宮是北臺灣參訪香客最多的廟宇。"
myMapView.addAnnotation(objectAnnotation)
```

使用類別 `MKPointAnnotation()` 來新增一個地點圖示，並設置地點的座標(緯度與經度)及資訊，再以方法 `addAnnotation()` 來加入到地圖中。

地點的預設圖示為紅色大頭針，如果要自定義大頭針顏色或是設置成另一個圖片，需要交由委任方法，所以接著會介紹委任方法的實作。

委任方法

先為委任對象(也就是 `ViewController`)加上委任模式需要遵循的協定：(此處包含前一個範例使用的 `CLLocationManagerDelegate` 委任協定。)

```
class ViewController: UIViewController,
    CLLocationManagerDelegate, MKMapViewDelegate {
    // 省略
}
```

以及在 `ViewController` 中實作的委任方法：

```
//自定義大頭針樣式
func mapView(mapView: MKMapView,
    viewForAnnotation annotation: MKAnnotation)
-> MKAnnotationView? {
    if annotation is MKUserLocation {
        // 建立可重複使用的 MKAnnotationView
        let reuseIdentifier = "MyPin"
        var pinView =
            mapView.dequeueReusableCellWithIdentifier(
                reuseIdentifier)
        if pinView == nil {
            // 建立一個地圖圖示視圖
            pinView = MKAnnotationView(
                annotation: annotation,
                reuseIdentifier: reuseIdentifier)
            // 設置點擊地圖圖示後額外的視圖
            pinView?.canShowCallout = false
            // 設置自訂圖示
            pinView?.image = UIImage(named:"user")
        } else {
            pinView?.annotation = annotation
        }
    }
    return pinView
} else {
    // 其中一個地點使用預設的圖示
}
```

```
// 這邊比對到座標時就使用預設樣式 不再額外設置
if annotation.coordinate.latitude
== 25.036798 &&
annotation.coordinate.longitude
== 121.499962 {
    return nil
}

// 建立可重複使用的 MKPinAnnotationView
let reuseIdentifier = "Pin"
var pinView =
mapView.dequeueReusableCellWithIdentifier(
    reuseIdentifier) as? MKPinAnnotationView
if pinView == nil {
    // 建立一個大頭針視圖
    pinView = MKPinAnnotationView(
        annotation: annotation,
        reuseIdentifier: reuseIdentifier)
    // 設置點擊大頭針後額外的視圖
    pinView?.canShowCallout = true
    // 會以落下釘在地圖上的方式出現
    pinView?.animatesDrop = true
    // 大頭針的顏色
    pinView?.pinTintColor =
        UIColor.blueColor()
    // 這邊將額外視圖的右邊視圖設為一個按鈕
    pinView?.rightCalloutAccessoryView =
        UIButton(type: .DetailDisclosure)
} else {
    pinView?.annotation = annotation
}

return pinView
}

}
```

上述程式這個實作的委任方法，是用來定義地圖圖示(大頭針)的樣式與內容，與表格 [UITableView](#) 的 `cell` 類似，地圖上的大頭針圖示，會使用方法 `dequeueReusableAnnotationViewWithIdentifier()` 來重複使用視圖。

一開始先判斷這個地圖圖示是否為自身定位位置 `if annotation is MKUserLocation {}`，當地圖視圖有設置 `showsUserLocation` 為 `true`，且有向使用者取得定位權限，這邊則會自動獲得一個自身定位位置 `MKUserLocation`。

這邊示範將定位位置圖示設置為一個新的圖片，這時如果尚未有可以重複使用的視圖，則必須以類別 `MKAnnotationView()` 來建立一個視圖，並設置包含圖片的其餘屬性。(手動加入的地點圖示要自定義圖片也是一樣的方式。)

接著則是設置手動加入的地點圖示，這邊為了示範不同的方式，所以以一個座標的緯度與經度判斷地點，來讓一個地點(艋舺公園)使用預設圖示，另一個地點(行天宮)則是自定義圖示內容。

同樣使用方法 `dequeueReusableAnnotationViewWithIdentifier()` 來重複使用視圖。如果尚未有可以重複使用的視圖時，則使用另一個類別 `MKPinAnnotationView()` 來建立一個視圖(請注意，與自訂圖片時使用的不同。)，並設置其餘屬性。

以上內容就會在地圖上顯示或移除地點圖示時重複使用。

MapKit 函式庫提供可以實作的委任方法還有很多，下面列出常使用的方法，請依照需求再各自實作：

```
func mapView(mapView: MKMapView,  
regionWillChangeAnimated animated: Bool) {  
    print("地圖縮放或滑動時")  
}  
  
func mapViewDidFinishLoadingMap(mapView: MKMapView) {  
    print("載入地圖完成時")  
}  
  
func mapView(mapView: MKMapView,  
annotationView view: MKAnnotationView,  
calloutAccessoryControlTapped control: UIControl) {  
    print("點擊大頭針的說明")  
}  
  
func mapView(mapView: MKMapView,  
didSelectAnnotationView view: MKAnnotationView) {  
    print("點擊大頭針")  
}  
  
func mapView(mapView: MKMapView,  
didDeselectAnnotationView view: MKAnnotationView) {  
    print("取消點擊大頭針")  
}
```

以上即為這小節範例的內容。

圖片來源

- https://www.iconfinder.com/icons/1153193/account_avatar_man_people_person_user_icon

範例

本節範例程式碼放在 [apps/taipeitravel](#)

記帳

本節會介紹如何建立一個記帳應用程式，你可以新增、編輯或是刪除你的花費記錄，並會以月為單位列表顯示所有記錄。

你可以在 App Store 中找到這個應用程式，名稱為記帳 **財**。

以下列出會使用到的 UIKit 元件與功能，如果還有尚未了解的地方，可以先往前面章節複習一下：

- 文字標籤 `UILabel`
- 文字輸入 `UITextField`
- 按鈕 `UIButton`
- 提示框 `UIAlertController`
- 圖片 `UIImageView`
- 選取日期時間 `UIDatePicker`
- 表格 `UITableView`
- 多頁面
- 導覽控制器 `UINavigationController`
- 手勢 `UIGestureRecognizer`
- 儲存資訊 `NSUserDefaults`
- `SQLite`

範例

本節範例程式碼放在 [apps/money](#)

規劃與實作

請先在 Xcode 打開這個專案([apps/money/Money](#))以供後續與文章內容比對檢視，本小節說明僅會提示部分內容，不會將所有程式碼都寫出來，請以專案程式碼為主。

首先介紹記帳應用程式可以操作的動作，先看下圖流程：



上圖從 Main 為首頁開始，依序可以操作如下的動作：

1. 新增：在首頁點擊下方的加號，即可新增花費記錄，需要填寫的欄位為金額、事由及時間。
2. 列表、更新與刪除：首頁會以月為單位列出所有花費記錄，依日期由新到舊排序，點擊其中一筆記錄可以更新或刪除。更新頁的左下方有刪除按鈕以供刪除這筆記錄。
3. 關於：點擊首頁右上角關於，可以進入關於頁。可以前往支援網頁與來源網頁。

前置作業

首先在 Xcode 裡，新建一個 **Single View Application** 類型的專案，取名為 Money。

這個專案會使用 **SQLite** 來儲存與操作資料，請依照 **SQLite** 的內容依序建立需要的檔案與功能。設定完後，專案應該新增好下列檔案：

- BridgeHeader.h：用以連接 Objective-C 的 Bridging Header 檔案。
- libsqlite3.tbd：加入 SQLite 函式庫後，由系統自動新增的檔案。
- SQLiteConnect.swift：將操作 SQLite 的程式碼封裝起來的類別。

在寫程式碼之前，先依照 [程式之外的設定](#) 的步驟設定好。以及將需要的 Swift 檔案、圖示檔案都先加入至專案中：

- PostViewController.swift
- MoreViewController.swift
- icons 目錄中的四個圖示檔案

AppDelegate.swift

在進入應用程式的畫面前，需要在 **AppDelegate.swift** 中先做些設定：

建立資料表

儲存一個值 `dbInit` 在 `NSUserDefaults` 中來辨識是否建立過資料表，當第一次進入應用程式時會建立資料表，欄位如下：

- `id`：整數，單筆記錄的識別碼，會自動增加(`auto increment`)。
- `title`：字串，單筆記錄的事由。
- `amount`：浮點數，單筆記錄的金額。

- `yearMonth`：字串，單筆記錄的時間，格式為年月，像是 `2016-05`。
- `createDate`：字串，單筆記錄的時間，格式為年月日，像是 `2016-05-12`。
- `createTime`：`DateTime`，單筆記錄的時間，格式為年月日時分，像是 `2016-05-12 12:36`。

初始設定

- 設置導覽列的一些預設樣式。
- 依照導覽控制器 `UINavigationController` 的說明，將根視圖設為一個 `UINavigationController`，並指定 `ViewController` 為第一個視圖控制器。

首頁

先注意以下 `ViewController` 的屬性：

- 建立陣列屬性 `days` 存放這個月份哪幾天有記錄，同時也是 `UITableView` 的 `section` 數量以及標題。
- 建立陣列屬性 `myRecords` 存放單日的每筆記錄。

在 `ViewController` 的 `viewDidLoad()` 中依序建立：

- 導覽列右上角的前往關於頁面按鈕。
- 目前顯示的年月 `UILabel`。
- 切換不同月份的往前與往後按鈕。
- 目前月份的總金額 `UILabel`。
- 目前月份的花費記錄列表 `UITableView`。
- 底部的新增記錄按鈕。

在 `viewWillAppear()` 中則是依據存在 `NSUserDefaults` 的一個值 `displayYearMonth` 來分辨目前要顯示的月份列表，這個值在新增或更新記錄時才會有值，用以顯示剛更新好的紀錄的月份列表。

點擊新增按鈕後執行的 `addBtnAction()` 方法與點選各記錄項目後執行的委任方法，都同樣會開啟 `PostViewController` 頁面，不同的是會在 `NSUserDefaults` 中儲存一個值 `postID`，這個值設為 `0` 時是新增，設為 記錄 `id` 時則是更新。

新增, 更新與刪除

PostViewController 中會新增一個結構(Struct)來存放記錄資訊，完成動作要儲存時也是以這個結構的值為主，如下：

```
struct Record {  
    var id :Int = 0  
    var title :String?  
    var amount :Double?  
    var yearMonth :String?  
    var createDate :String?  
    var createTime :String?  
}
```

在 PostViewController 的 viewDidLoad() 中，先取得 UserDefaults 的值 postID 來分辨目前是新增或更新。新增時在屬性 record 中存放預設值，更新時則是將這筆記錄的原始資料存入，以供後續使用與顯示。

刪除按鈕只有在更新記錄時才會顯示。按下刪除按鈕後，會顯示一個 UIAlertController 來確認刪除動作。

關於頁面

這頁使用 UITableView 的 .Grouped 特性各別列出不同的功能：

1. 前往外部 **Facebook** 網頁。
2. 前往外部 圖示資源 網頁。

前往外部網頁時，使用下列方式：

```
let requestUrl = NSURL(string:  
    "https://www.facebook.com/1640636382849659")  
UIApplication.sharedApplication().openURL(requestUrl!)
```

範例

此應用程式範例程式碼放在 [apps/money/Money](#)

你也可以在 App Store 中找到這個應用程式，名稱為記帳 **財**。

補充

- Xcode 介紹
- 系統關鍵字
- 駝峰式命名法

Xcode 介紹

如果你是第一次接觸 Xcode，相當建議把這章的內容瀏覽並同步操作一遍，第一次看的時候可能會似懂非懂，但有個印象就夠了，未來隨著閱讀內容的進展，會越來越熟悉。

如果已經接觸過 Xcode 一段時間，仍然建議在卡關時回來看看操作步驟。以下是本章的介紹內容：

- [安裝 Xcode](#)
- [開啟 playground](#)
- [開啟專案](#)
- [介面簡介](#)
- [新增檔案](#)
- [加入檔案](#)
- [刪除專案](#)
- [新增 Framework](#)
- [使用模擬器及實機測試](#)
- [熱鍵](#)

安裝 Xcode

要練習 Swift 或開發 iOS App，都必須先安裝 Xcode。Xcode 是一個整合開發環境 (Integrated Development Environment，簡稱 IDE)，它提供了所有開發 iOS App 需要的工具，當然也包含 iPhone 模擬器，讓你不需要有實體裝置也能測試 App。

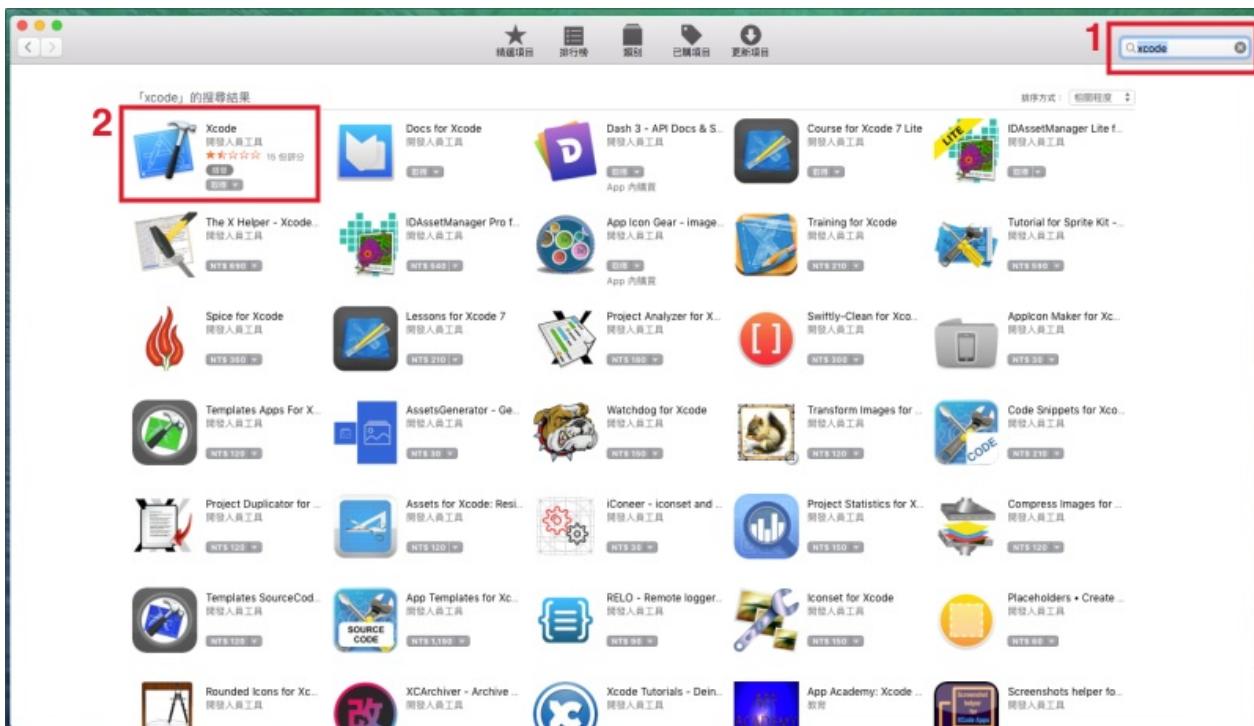
OS X 裡預設是沒有安裝 Xcode 的，所以首先必須從 App Store 裡下載安裝。

▼ 找到 Mac 中 Dock 的這個 app 並點擊開啟，如下：

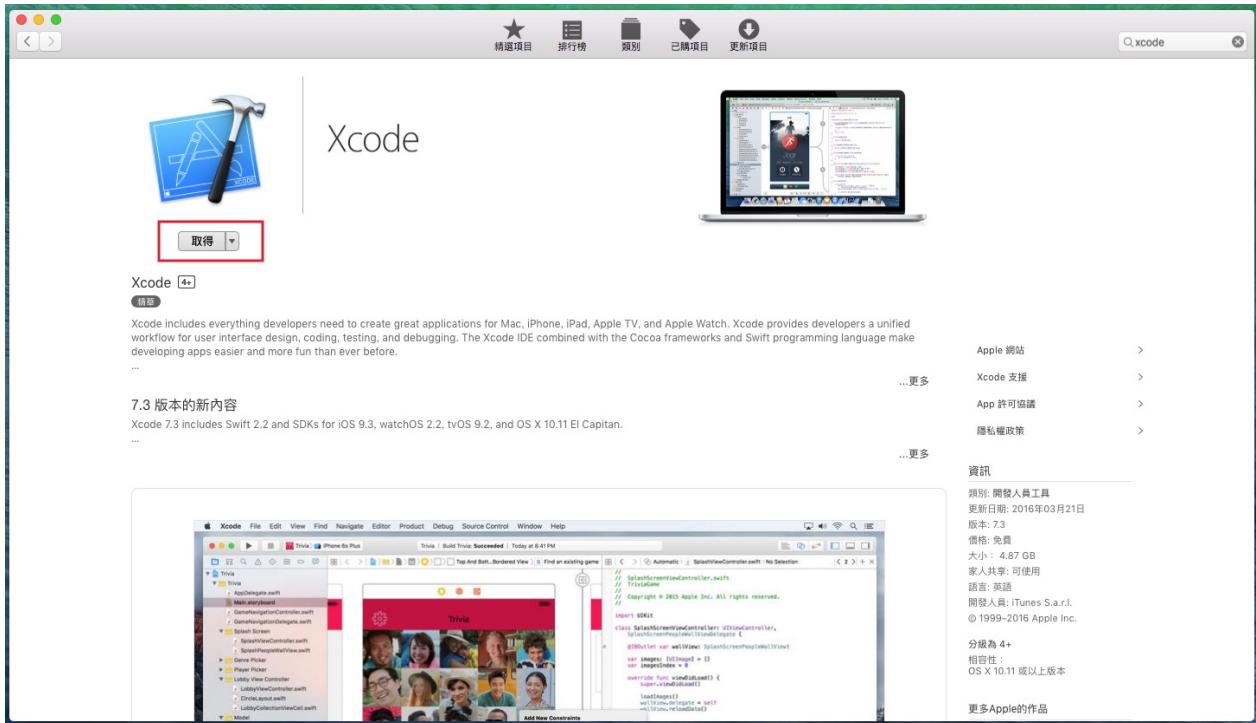


▼ 開啓後會看到下面這個畫面：

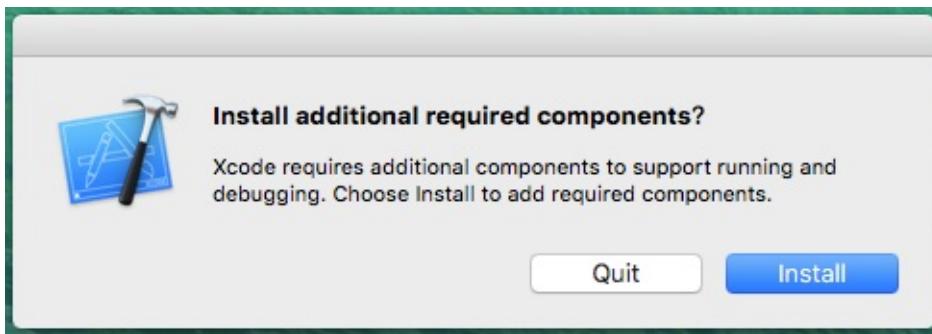
1. 輸入 `xcode`，會列出相關的 app，通常 Xcode 都會出現在第一個。
2. 點擊 Xcode 進入詳細頁面。



▼ 點擊 Icon 下方的 取得 按鈕，就會開始安裝(如果尚未登入 App Store，會先要求登入)，安裝過程可能會需要一段時間，軟體滿大的，約是 4 G 多：

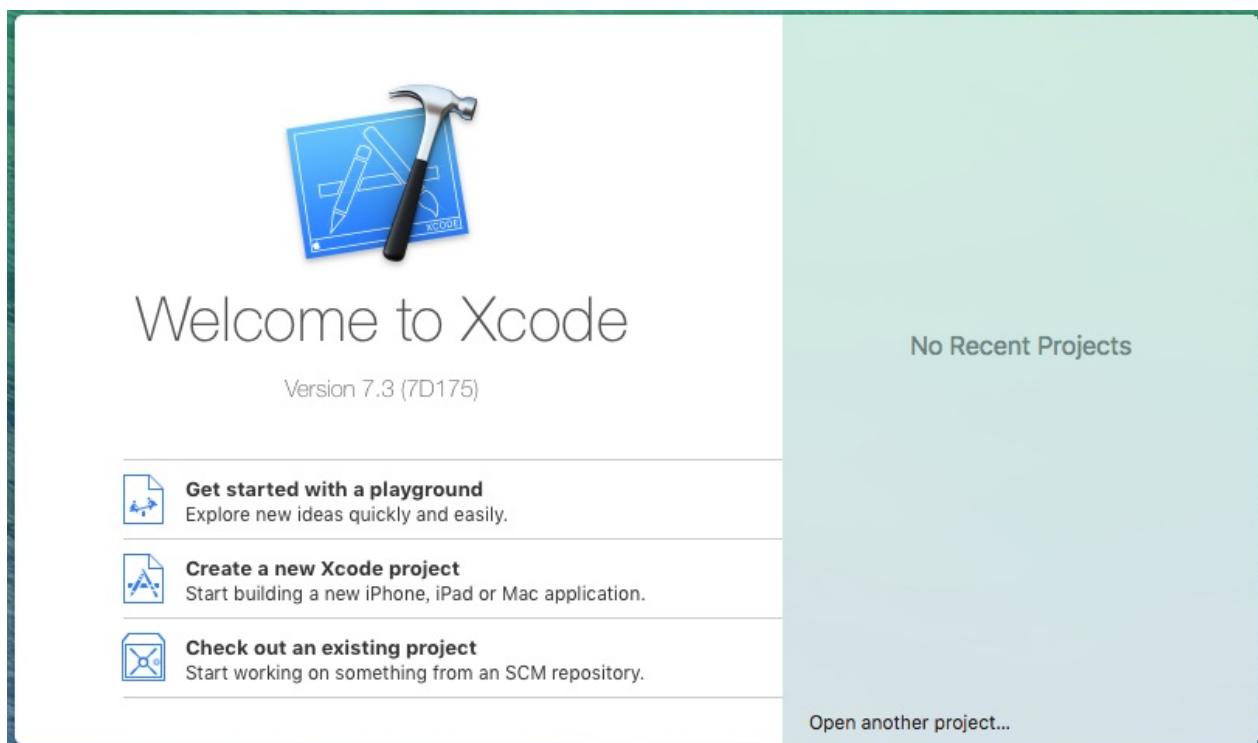


▼ 安裝完畢後，Dock 應該就會出現 Xcode App，請點擊它開啟，首次開啓時可能會詢問安裝一些額外的元件，點擊 Install 直接安裝就好：



▼ 第一次開啓 Xcode 後，會出現下面這個畫面，依序為建立一個 playground、建立一個新專案以及開啓一個已存在的專案：

(編寫文章時的 Xcode 版本為 7.3，未來如有更新，請安裝最新版的。)



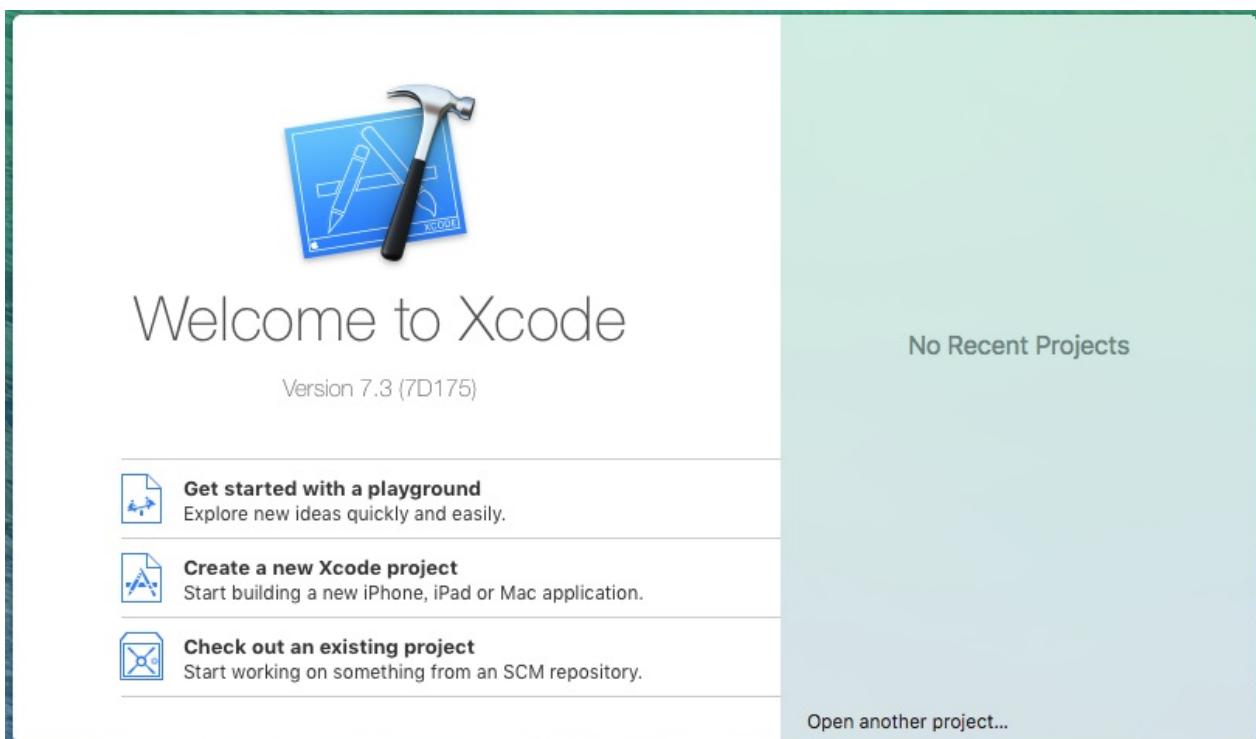
這樣就完成了 Xcode 的安裝，接著就按照需求繼續其他步驟囉。

開啟 playground

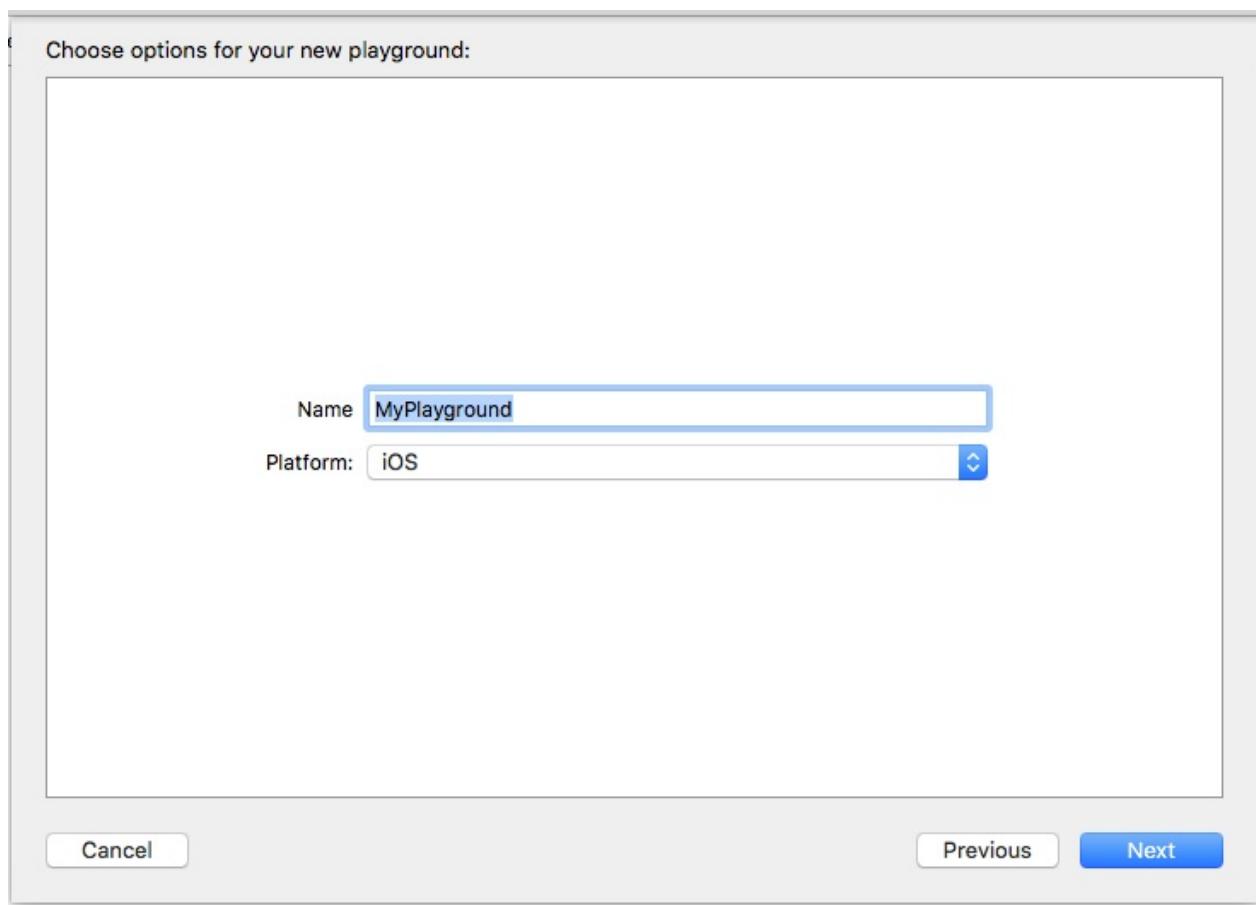
自 Xcode 6 之後，便內建了 playground，它可以讓你用即時的方式測試 Swift 程式碼，它會在你打完程式後，立即顯示結果。對於剛上手 Swift 的階段來說，是一個相當方便的工具。

建立一個新的 playground

▼ 在首次打開 Xcode 時會顯示下面這個畫面，請先點擊 `Get started with a playground` 按鈕：

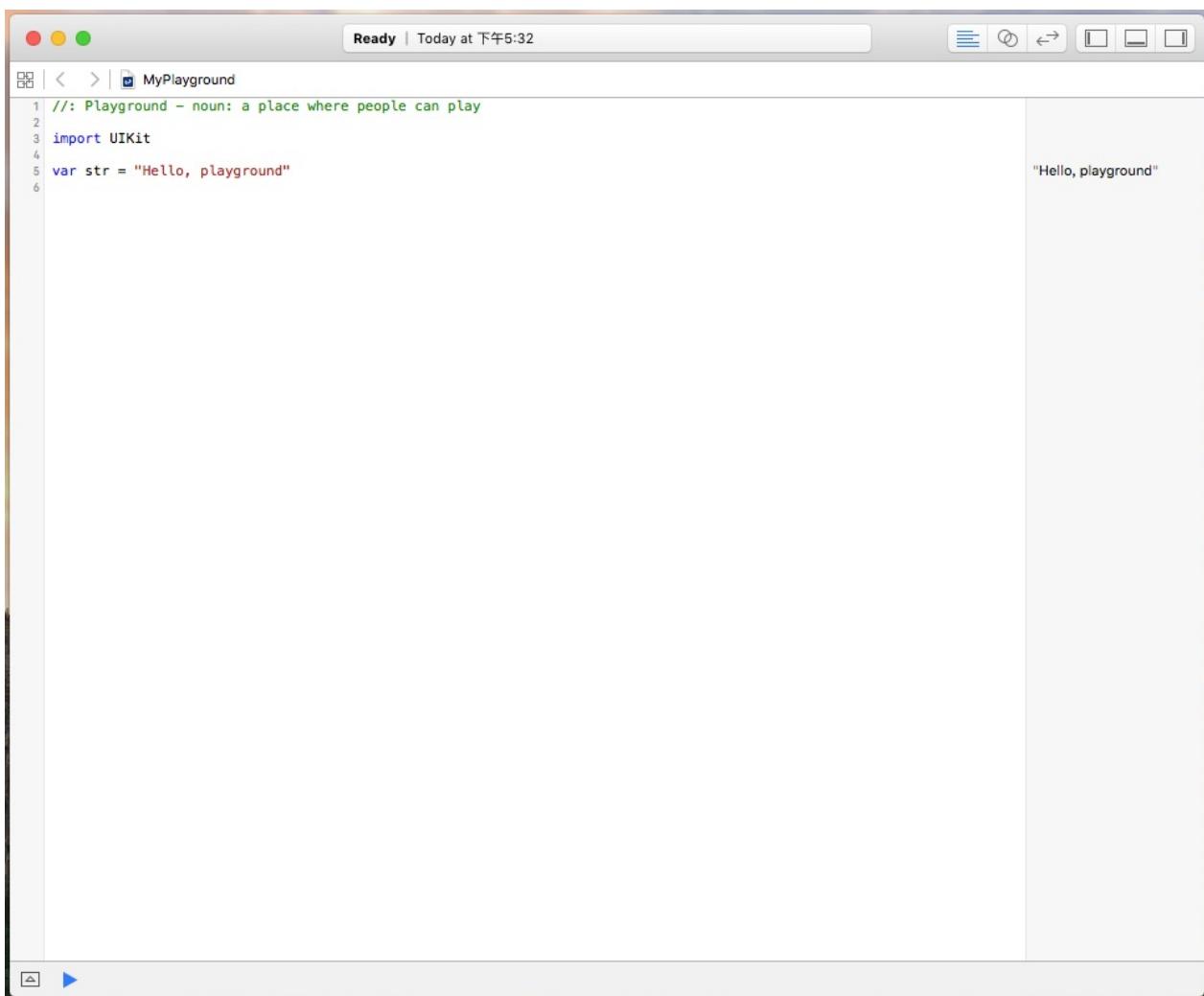


▼ 接著會詢問你檔案名稱(Name)及平台(Platform)，這邊示範都使用預設的即可，命名為 `MyPlayground` 並選擇 `iOS` 平台，按下 `Next` 後，接著會詢問要儲存在哪，找好地方後按下 `Create`，便會完成建立：

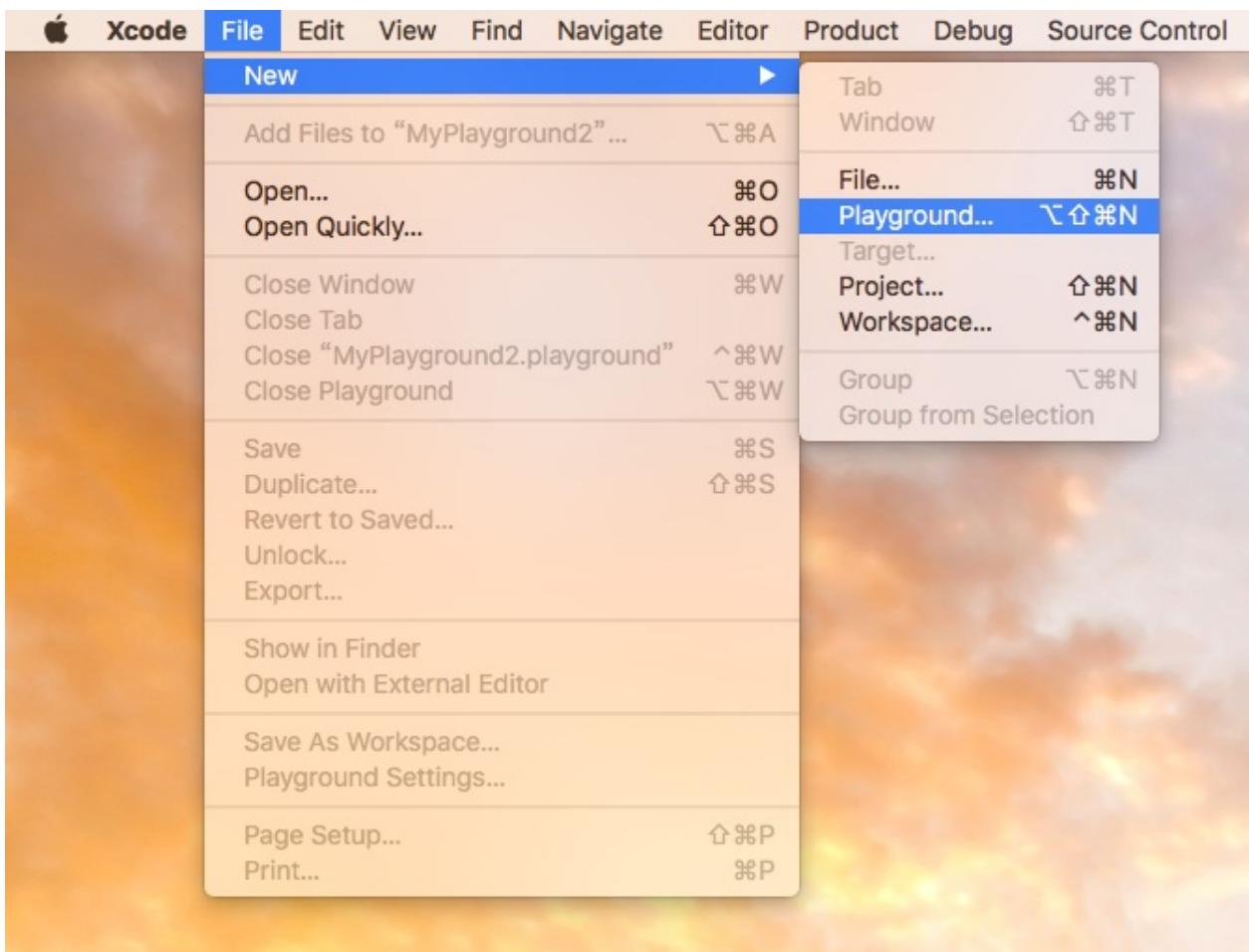


▼ 建立完 playground 後，下面可以看到最初的畫面，已預設寫了一個變數 `str`，在稍等幾秒後，右邊就會即時顯示目前有的變數、常數及各程式操作內容結果：

開啟 playground



▼ 除了起始畫面外，你也可以從 Xcode 工具列的 File > New > Playground... 建立一個新的 playground，如下：

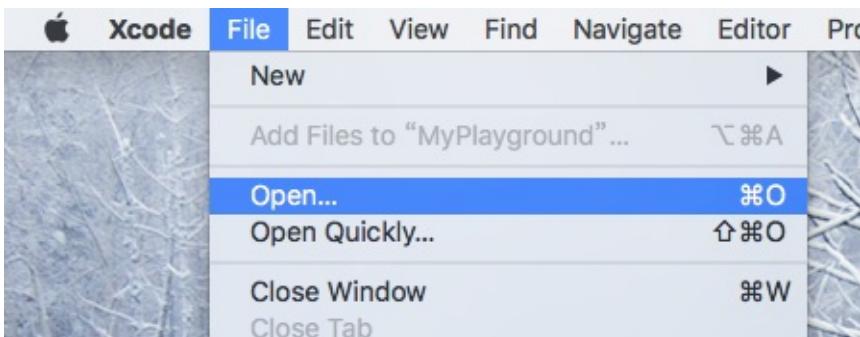


開啓一個存在的 playground

▼ 開啓一個 playground 就如同開啓一般檔案一樣，在 Finder 找到檔案後點擊，即可打開：



▼ 或是你也可以從 Xcode 工具列的 File > Open... 開啓一個 playground：



使用 playground

▼ 使用 playground 其實很直覺，就是寫什麼右邊就顯示什麼，有錯誤也會即時出現，有一點提醒：

1. 點擊視窗右上角的這個按鈕，視窗下方會顯示執行結果，當使用 `print()` 印出文字時，就會印出在這裡。
2. 程式中寫下 `print(str)` 。
3. 視窗下方就會印出結果。

A screenshot of the Xcode playground interface. The top bar shows the title 'Ready | Today at下午5:57'. The main area displays a Swift script named 'MyPlayground'. The code is:

```
1 //: Playground - noun: a place where people can play
2
3 import UIKit
4
5 var str = "Hello, playground"
6
7 print(str)
```

Line 7 is highlighted with a red box and labeled '2'. The output window on the right shows the results of the execution:

```
Hello, playground
Hello, playground\n
```

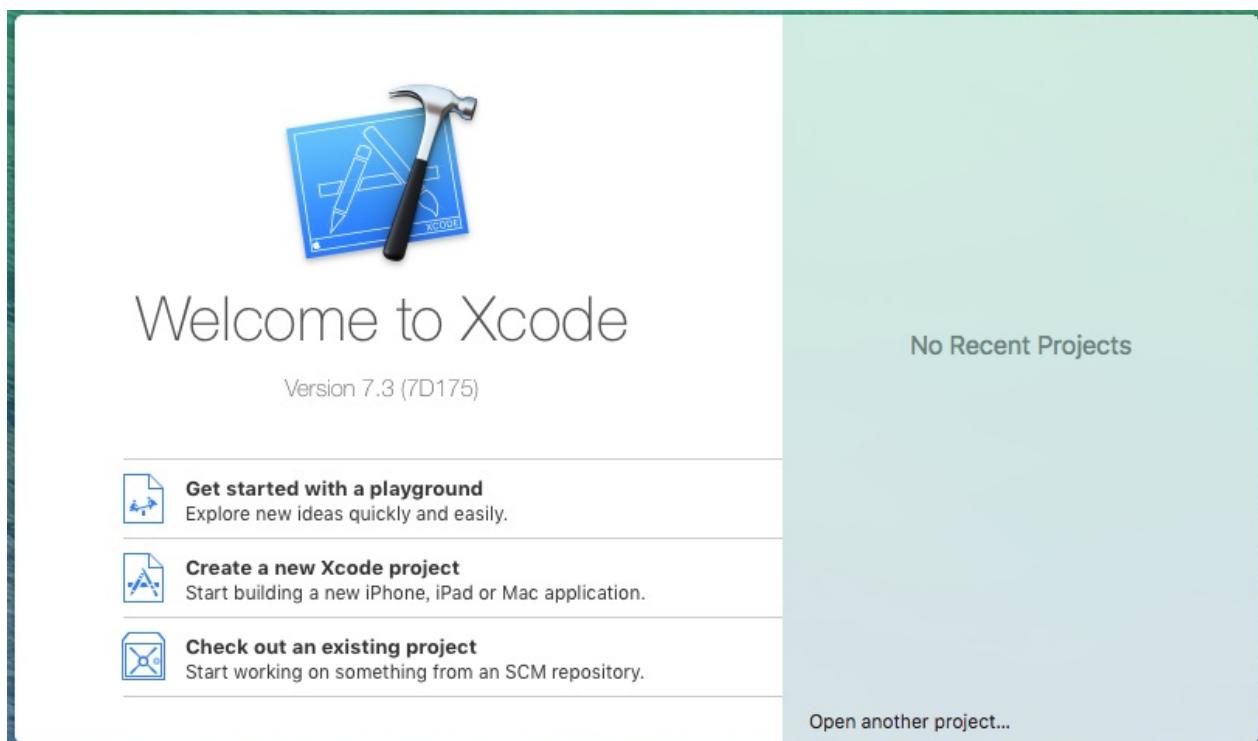
The bottom status bar shows the text 'Hello, playground' with a red box and labeled '3'.

開啓專案

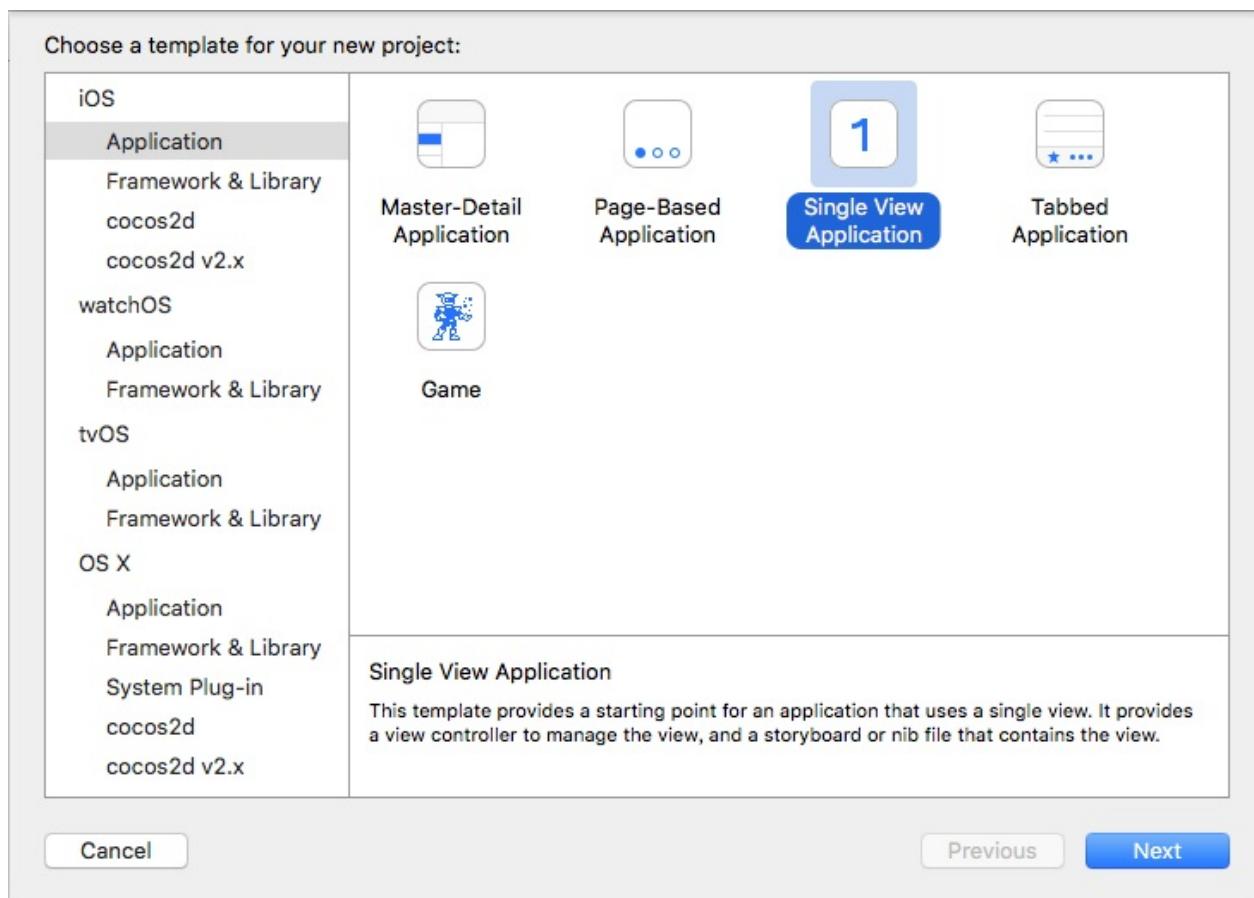
如果你要寫一個完整的 iPhone App，則是必須要建立一個專案，以下會介紹如何建立及開啓專案。

建立一個新的專案

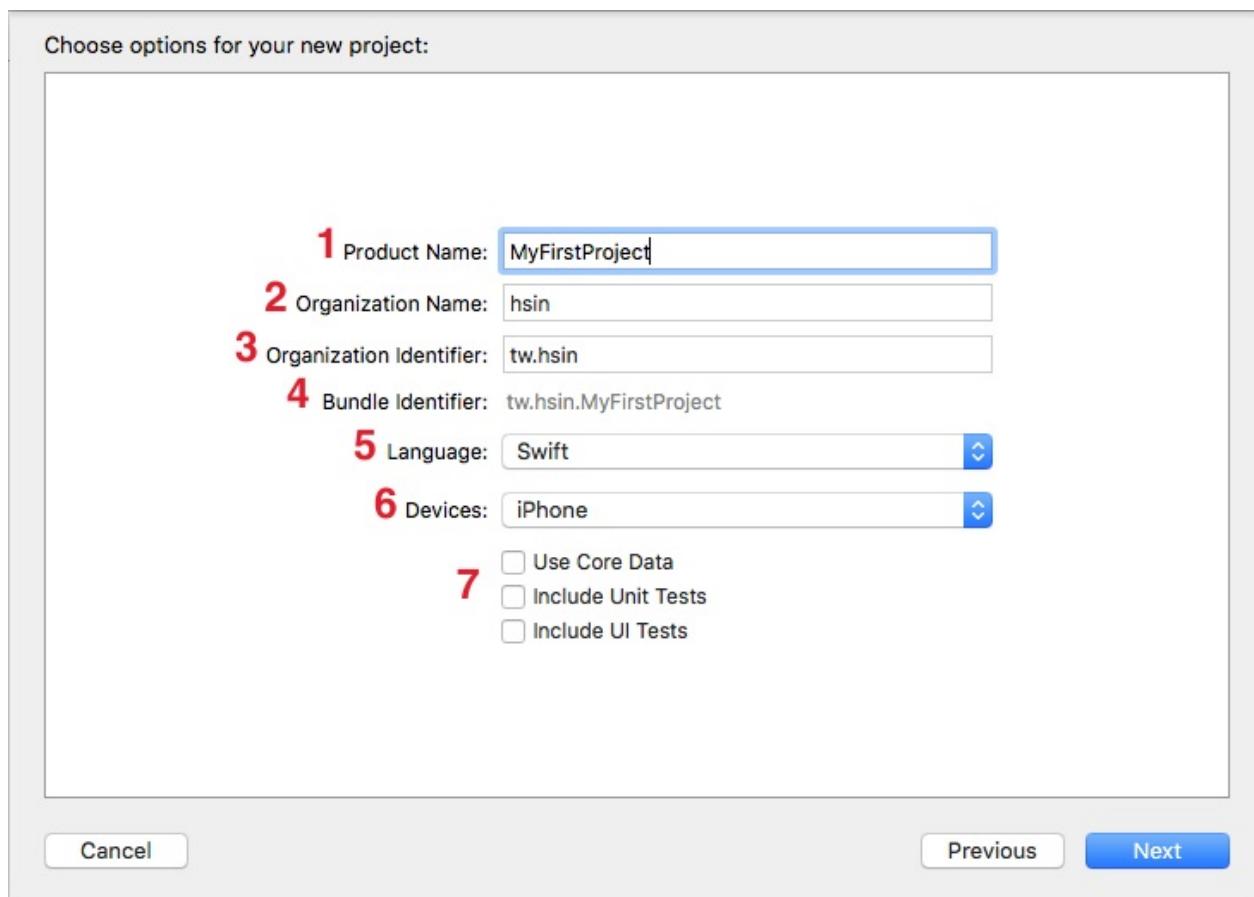
▼ 在首次打開 Xcode 時會顯示下面這個畫面，請先點擊 `Create a new Xcode project` 按鈕：



▼ 首先可以看到左邊列出可以製作應用程式的平台 iOS、watchOS、tvOS、OS X，這邊先選擇 iOS 中的 Application。右邊可以看到有數個模版可以選擇，每個模版都會預設提供一些常用的元件，可以讓你依照不同需求，快速的建立起應用程式。而此書大部分內容都是從純程式碼開始建構，所以這邊選擇 **Single View Application**，也就是一個單頁的應用程式，接著點擊 `Next` 按鈕繼續：



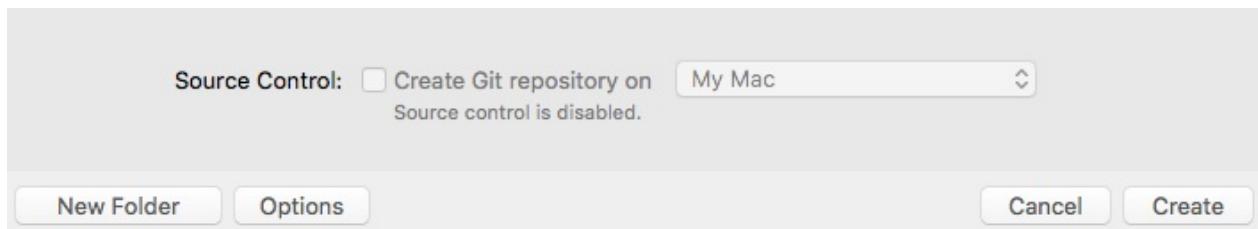
▼ 接著是下圖這個步驟，填寫專案的基本資料：



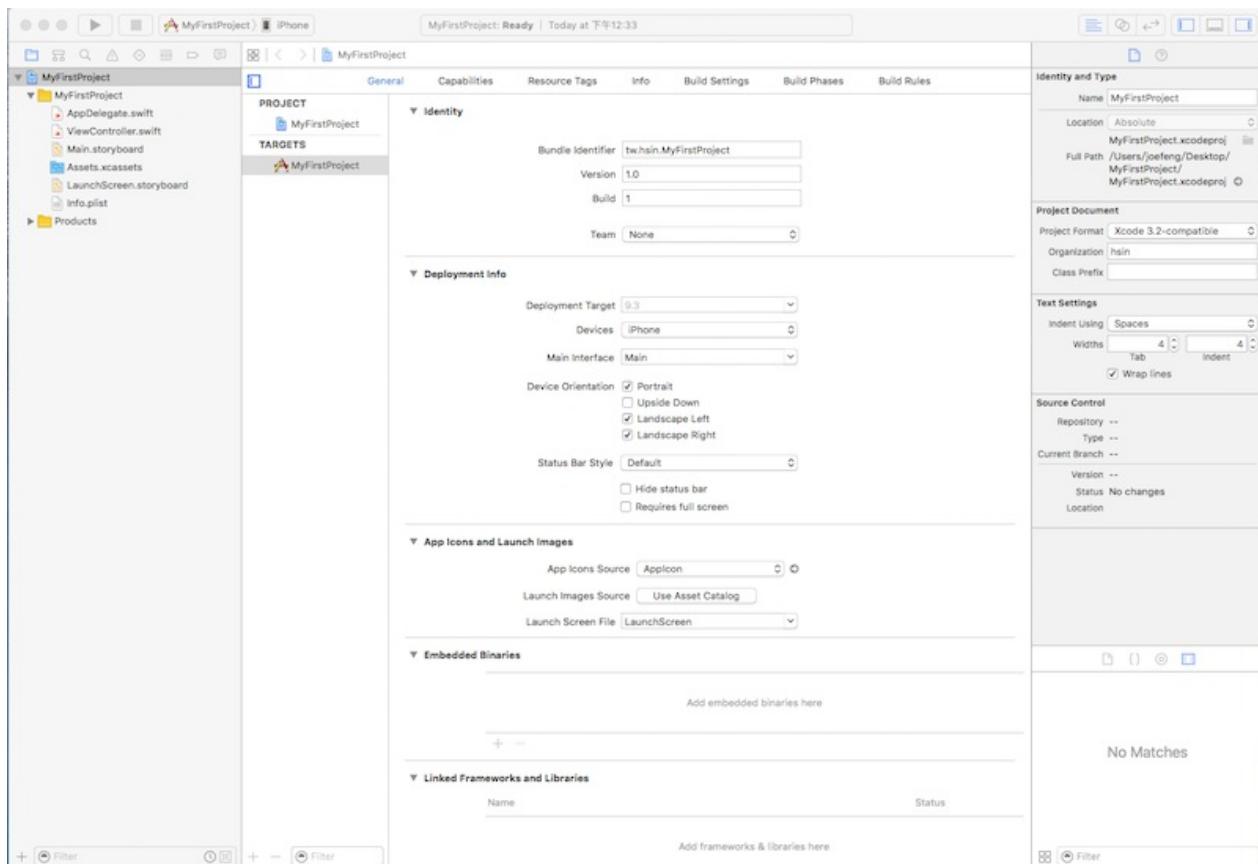
1. 專案名稱，這邊範例是填入 `MyFirstProject`。
2. 公司或個人名稱。
3. 公司或個人的唯一識別碼，這在提交應用程式給 App Store 時會用來辨識，通常會以點 `.` 來連接，像是反過來的網域名稱，這邊範例使用 `tw.hsin`，你也可以寫像是 `com.yourname` 或是 `tw.com.someone` 之類的，只要不要與別人的一樣即可(如果撞名，提交時會跟你講)。
4. 這個專案的唯一識別碼，是以公司或個人的唯一識別碼與專案名稱組成(所以這個欄位會依據前兩個名稱動態更新，無法獨自修改)，這在整個 App Store 裡面會是唯一的。
5. 選擇要使用的程式語言，目前仍然可以在 Swift 和 Objective-C 擇一使用，不過本書在講 Swift，所以當然是選擇 Swift。
6. 選擇適用的 iOS 裝置，有 iPhone、iPad 或 Universal 三個選項，看是要專屬 iPhone、iPad 或是兩者通用，本書以 iPhone 為主，所以這邊選擇 iPhone。
7. 額外的選項，Core Data 是一種資料庫的應用，是一個可以讓你簡化新增儲存刪除資料的功能，本書除非特別提起，不然都不用勾選。另外兩個都是用作程式測試的功能，本書不會提到，所以都不勾選。

都填好之後，點擊 `Next` 繼續。

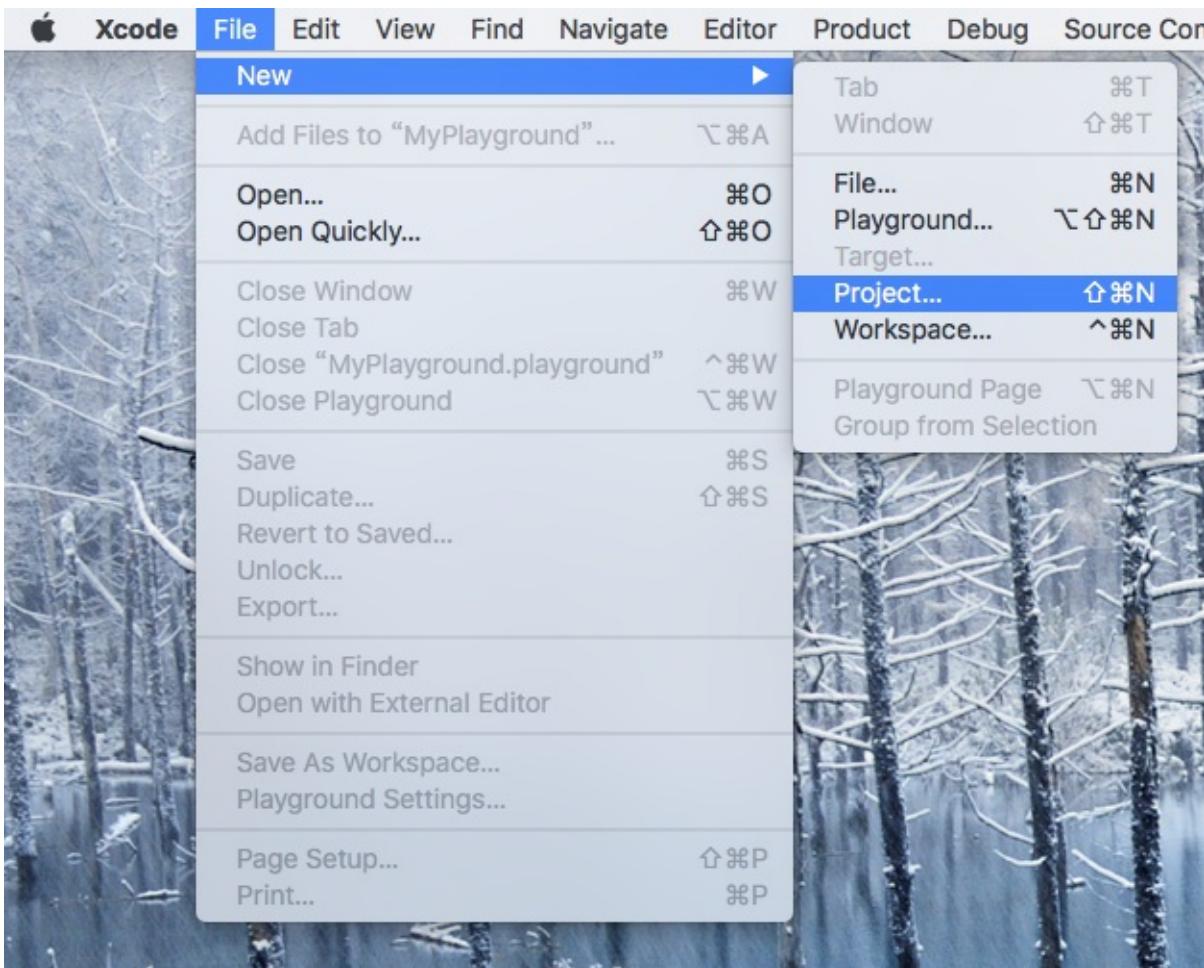
▼ 接著在儲存專案目錄的步驟下方，可以看到有個 Source Control 的選項，可以讓你的專案支援版本控制，本書不會提到，所以這邊不勾選。(預設可能是無法勾選，如果有需要請到 Xcode 設定中開啟。)



▼ 最後看到下面這個畫面時，即是完成了新專案的建立(請接著繼續閱讀下節，會詳細介紹這些介面。)：

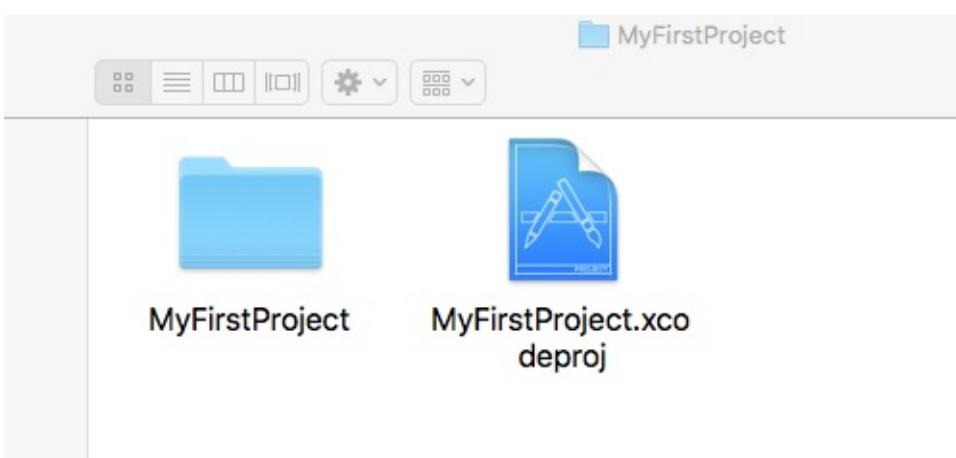


▼ 除了起始畫面外，你也可以從 Xcode 工具列的 **File > New > Project...** 建立一個新的專案，如下：

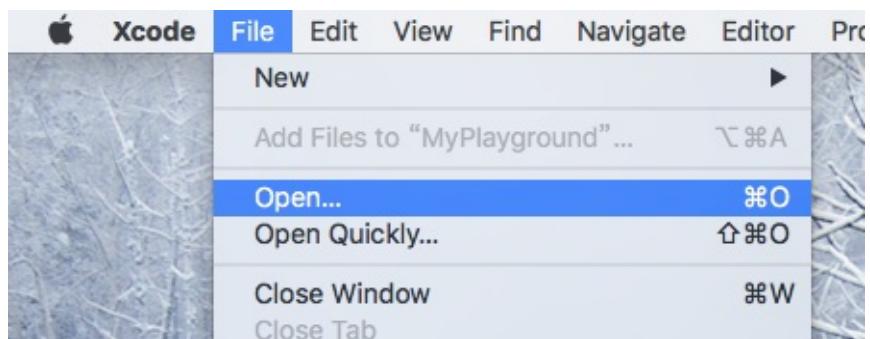


開啓一個存在的專案

▼ 在 Finder 找到專案目錄後，在裡面可以看到如下圖，有一個目錄以及一個 `.xcodeproj` 檔案，點擊這個 `.xcodeproj` 檔案即可打開專案：



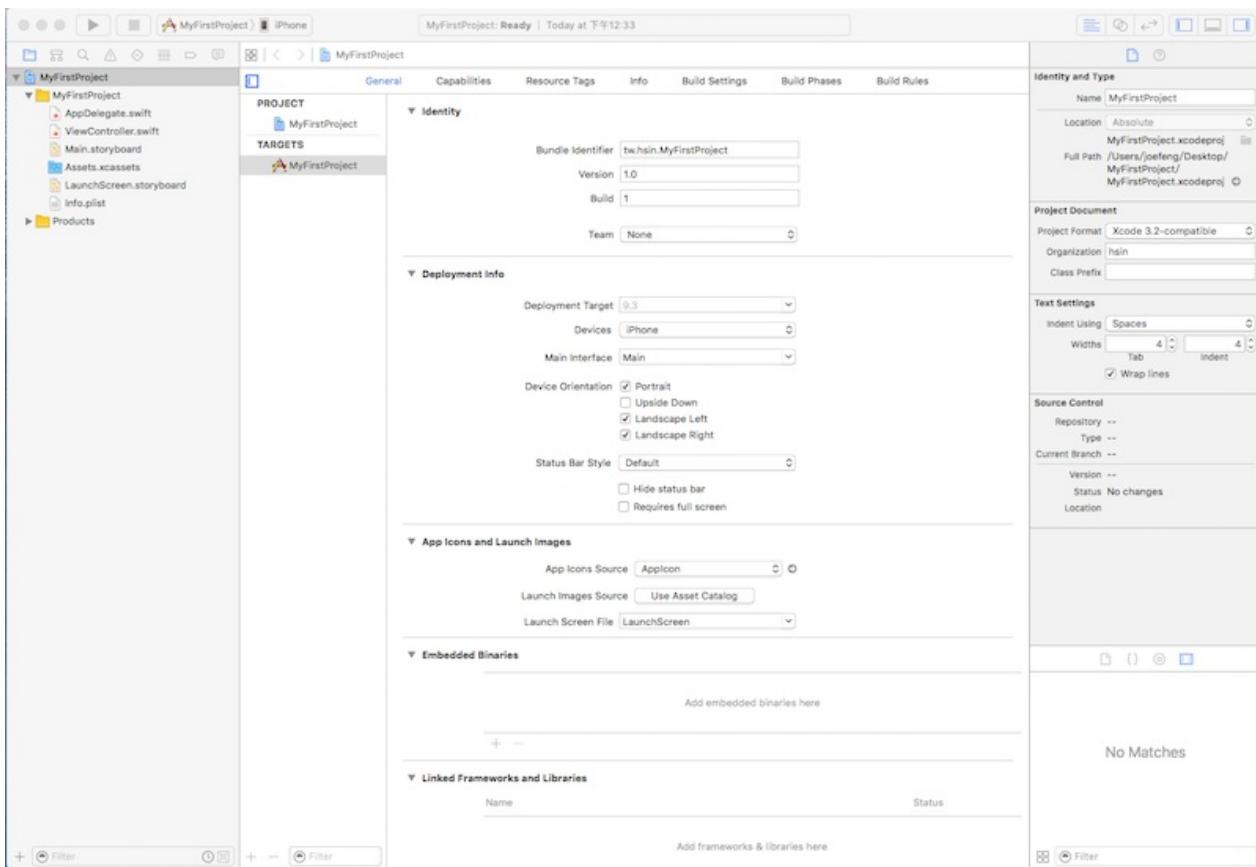
▼ 或是你也可以從 Xcode 工具列的 File > Open... 開啓一個專案：



介面簡介

在你建立或開啟一個專案後，可以看到整個專案的內容如下，這節會介紹這些常用的功能：

第一次看到這些東西可能會覺得有點過多，看完一遍後也可能還是會搞不太清楚，但是沒關係，你可以在學習中邊寫程式邊熟悉，如果看到不確定是什麼功能的東西，可以再回來這節中看看。

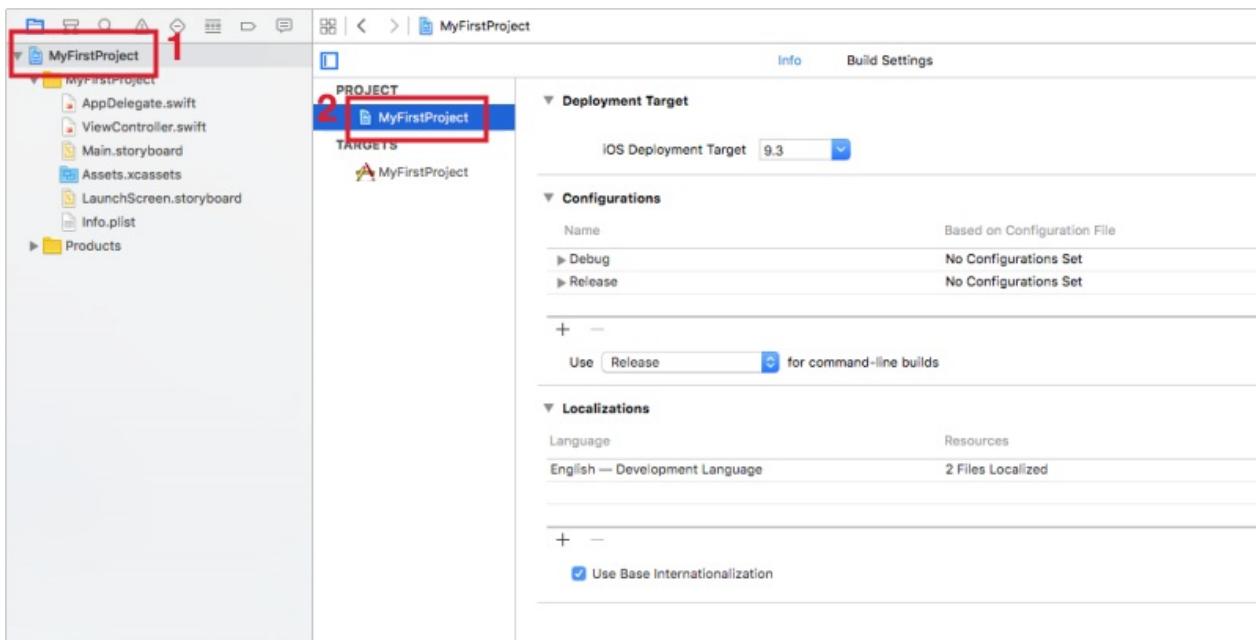


專案設定

首先看到這個專案的設定：

- 先點選左側側邊欄的 1. MyFirstProject。
- 再點選中間畫面 PROJECT 中的 2. MyFirstProject。

如下圖：

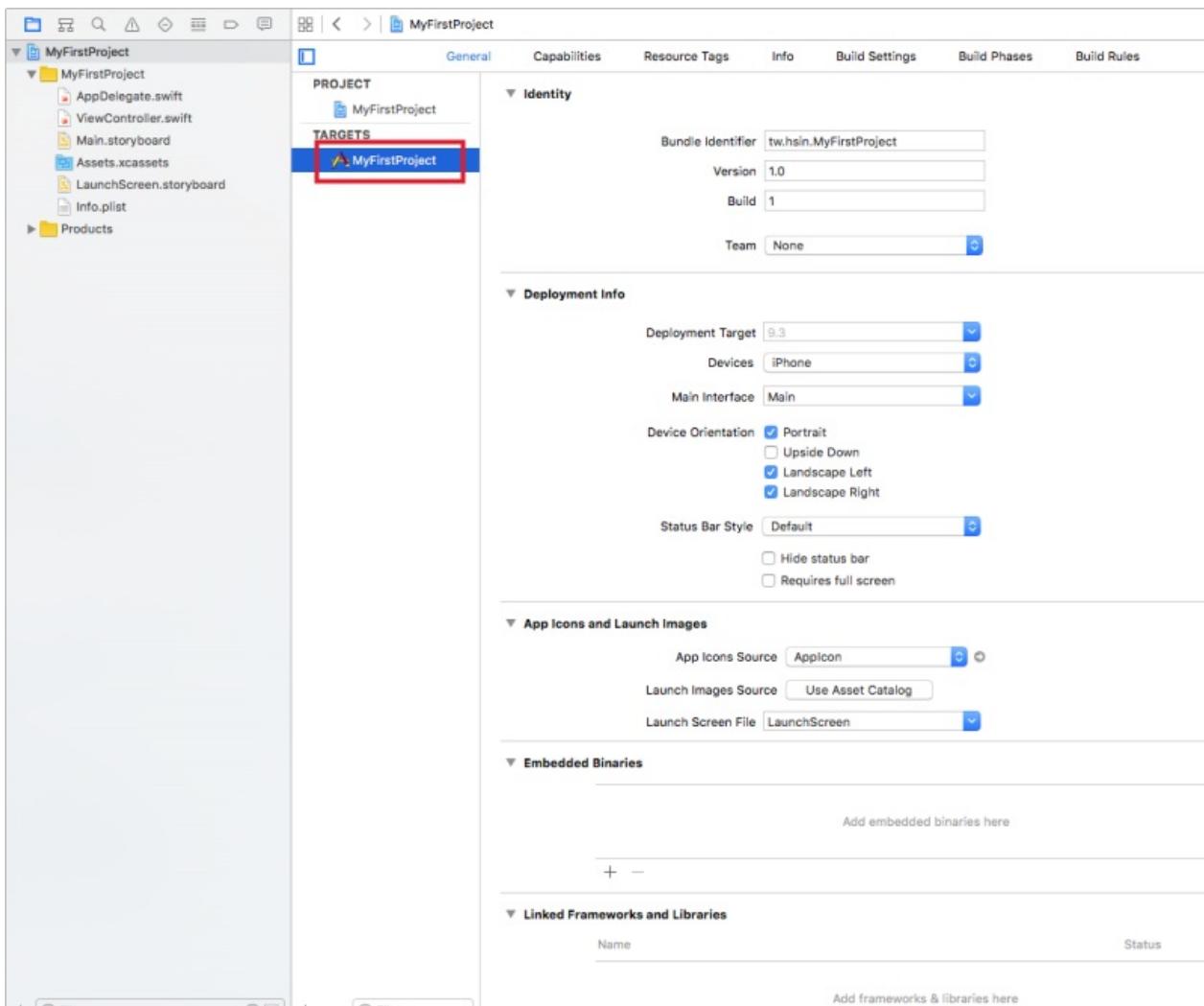


在 Info 這個頁籤中，顯示專案的一些設定：

- **Deployment Target**：應用程式支援的 iOS 版本，會向上支援但不能向下支援，所以如果應用程式要給較舊版本的 iOS 使用，記得設定較低的 iOS 版本。(像是要給 iOS 8.2 以上的裝置使用，這邊就要填寫 8.2。)
- **Configurations**：可以為不同模式下的編譯給予不同的設定，像是測試模式 (Debug) 與釋出模式(Release)。
- **Localizations**：應用程式需要支援多語系時，可在這邊增加需要支援的語言。

在 Build Settings 頁籤中則是用於編譯時的設定，初期可以先不用更動，使用預設設定就好。

接著點擊 TARGETS 中的 MyFirstProject，如下圖：



General 頁籤：

- Identity：
 - Bundle Identifier：建立專案時一同建立的專案的唯一識別碼。
 - Version：專案(也就是應用程式)的版本，會顯示在 App Store 中，每次程式有更新要提交時都必須更新，數字只能更大(像是 1.1.2 版本之後，只能更新為 1.1.3 或 1.2.0，不能再降為 1.1.1)。
 - Build：編譯版本，每次提交至 App Store 的應用程式，即使 Version 是一樣的，這個 Build 數字都必須較先前的 Build 數字更大(與 Version 規則相同)，來表示是不同編譯版本。
 - Team：與開發者帳號有關。
- Deployment Info：
 - Deployment Target：應用程式支援的 iOS 版本。
 - Devices：適用的 iOS 裝置。
 - Main Interface：主要使用的 Storyboard。
 - Device Orientation：提供支援的裝置翻轉方向，依序為 Portrait (正向)、

Upside Down (上下顛倒)、Landscape Left (向左翻轉)及 Landscape Right (向右翻轉)。

- Status Bar Style：狀態列的樣式。
- App Icons and Launch Images：
 - 設定應用程式的圖示與起始畫面。
- Embedded Binaries 與 Linked Frameworks and Libraries：
 - 使用額外功能框架或第三方套件時要匯入檔案的地方。

Capabilities 頁籤：

- 啓用 Apple 各個支援的功能，像是 iCloud、Push Notifications 或是 Game Center。開啓前都會需要開發者帳號認證。

Resource Tags 頁籤：

- 你可以為專案內使用的資源檔案設置不同的標籤(tag)，用來設定獲取這個資源的時間點。

Info 頁籤：

- 記錄一些專案的基本資料。

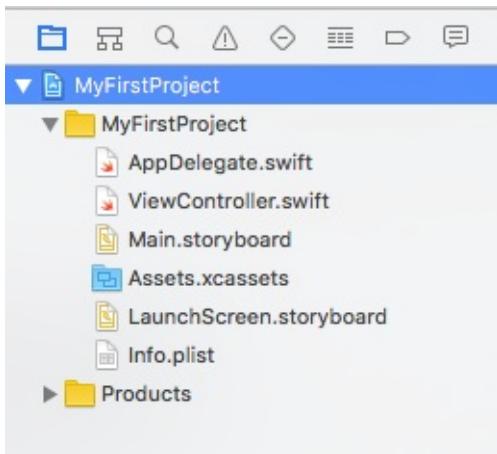
Build Settings、Build Phases 與 Build Rules 頁籤：

- 編譯時的設定，初期可以先不用更動，使用預設設定就好。

專案導覽區塊

專案導覽區塊(Project Navigator)位於畫面的左邊側邊欄，這邊會顯示有關這個專案檔案的資訊，以下依序介紹常用的部份：

▼ 點擊第一個頁籤，會顯示這個專案裡的所有檔案，如下圖：



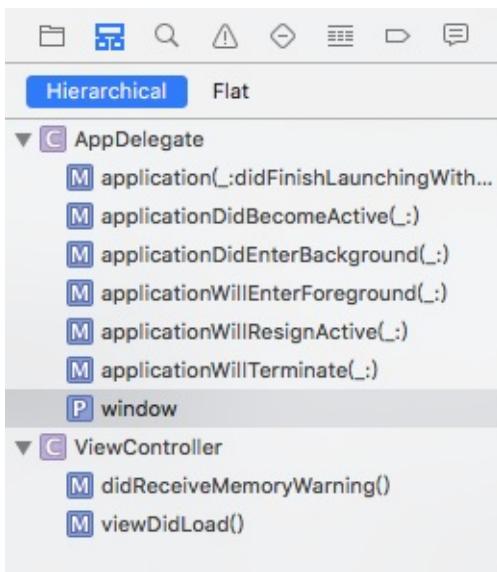
可以看到兩個目錄，一個與專案名稱相同 `MyFirstProject`，另一個為 `Products`。

- `MyFirstProject`：專案主要目錄，所有介面與程式檔案都放於此內。
- `Products`：放置專案編譯後的檔案，基本上是不會更動到。

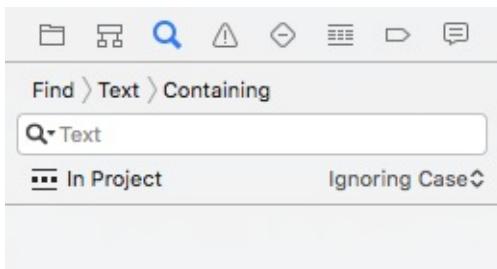
專案預設建立的個別檔案如下：

- `AppDelegate.swift`：負責這個應用程式與外部整個手機交流互動時的程式，像是應用程式啓動、閒置、進入後台、返回前台或是退出時要執行的動作。
- `ViewController.swift`：主要的視圖(`View`)控制器(`Controller`)，應用程式啓動時，會由這隻檔案的 `viewDidLoad()` 方法開始執行，如果只有一個頁面的話，大部分的程式都會寫在這裡面。
- `Main.storyboard`：可使用拖曳元件的方式設計應用程式的介面，本書因為主要使用純程式碼，`Storyboard` 部分不會詳細說明。
- `Assets.xcassets`：放置用於應用程式的圖檔，像是應用程式在列表中的圖示。
- `LaunchScreen.storyboard`：用於應用程式啓動時，尚未載入完畢前顯示的畫面，也是一個介面設計檔案(`storyboard`)。
- `Info.plist`：`plist` 是 `Cocoa` 的一種屬性列表檔案，以一種序列化的方式條列各屬性，可以看到這隻檔案裡面記錄著一些專案的基本資料。

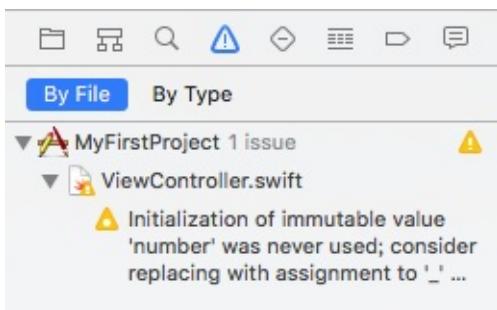
▼ 點擊第二個頁籤，會顯示這個專案裡的所有類別(`Class`)及其內所有的方法(`method`)與屬性(`property`)：



▼ 點擊第三個頁籤，可以讓你搜尋專案內的文字：



▼ 點擊第四個頁籤，會顯示編譯後出現的警告(warning)與錯誤(error)，這邊為了示範，故意寫了一個沒有使用的常數 number ，編譯後便會自動跳到這個頁籤，並告訴你有什麼警告(黃色)與錯誤(紅色)：



專案編輯區塊

專案編輯區塊(Project Editor)位於畫面大部分的中間區塊，編寫程式碼都是寫在這裡，點擊左側檔案後，編輯區塊就會顯示程式，如下：

The screenshot shows the Xcode interface. On the left is the Project Navigator, displaying the project structure: MyFirstProject > MyFirstProject > ViewController.swift. The main area is the code editor, showing the following Swift code:

```
// ViewController.swift
// MyFirstProject
//
// Created by joe feng on 2016/5/9.
// Copyright © 2016年 hsin. All rights reserved.

import UIKit

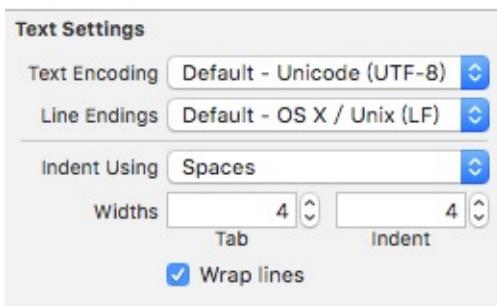
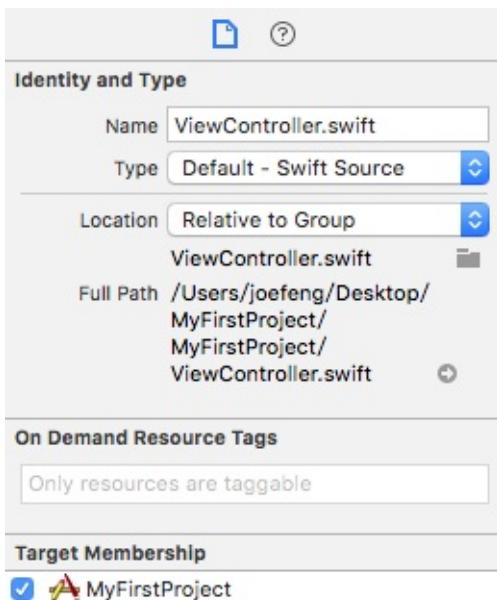
class ViewController: UIViewController {

    override func viewDidLoad() {
        super.viewDidLoad()
        // Do any additional setup after loading the view, typically from a nib.
    }

    override func didReceiveMemoryWarning() {
        super.didReceiveMemoryWarning()
        // Dispose of any resources that can be recreated.
    }
}
```

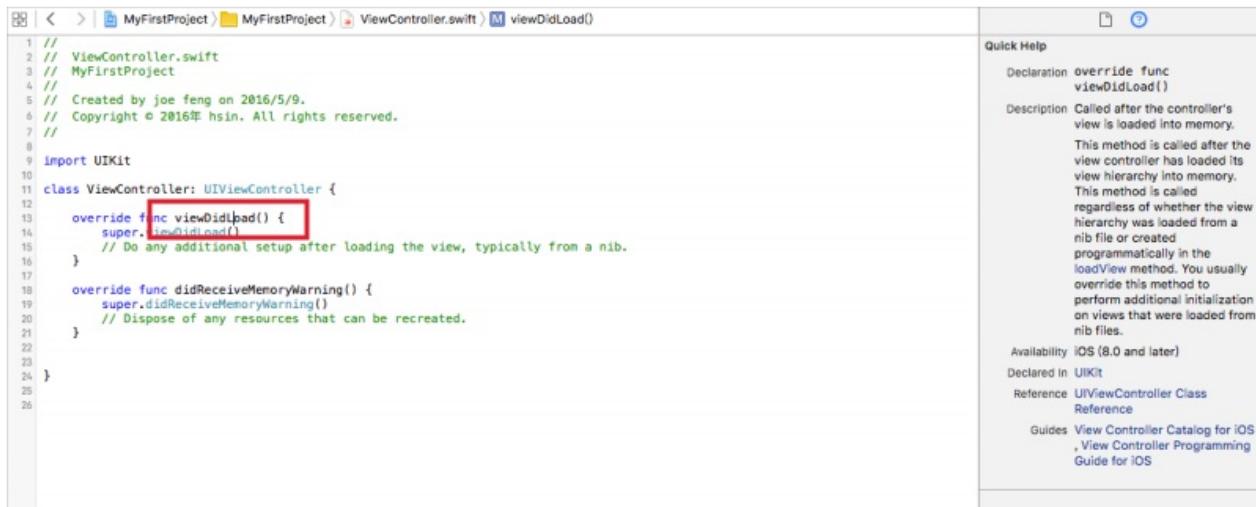
工具區塊

工具區塊(Utility)位於畫面的右側側邊欄，主要用於提示當前畫面檔案的額外資訊，如下圖：



上圖為工具區塊中左邊頁籤的內容，當左側側邊欄檔案列表點擊到 `ViewController.swift` 時，右側側邊欄會隨即顯示這個檔案的資訊，像是檔案名稱或檔案位置。

工具區塊中右邊頁籤則是會提示目前游標指到的方法或類別等等的詳細資訊，如下：



將游標點擊程式中的 `viewDidLoad()` 時，右側側邊欄則會顯示這個方法的詳細資訊，像是它的用途、從幾版的 iOS 開始支援或是繼承自哪個類別等等。

除錯區塊

畫面右上角可以看到下面這排按鈕，可以顯示及收起左側欄、底邊欄及右側欄：



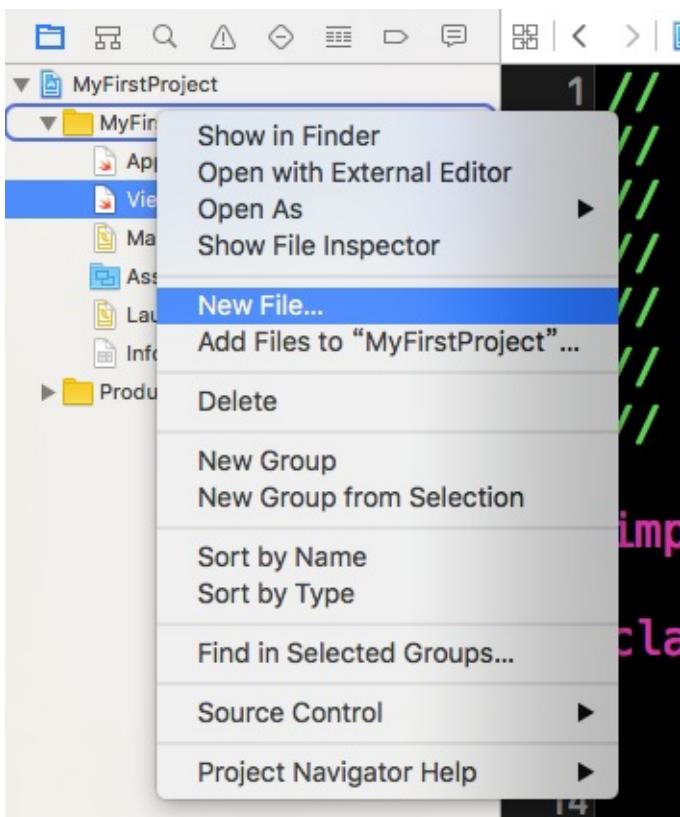
而尚未介紹到的底邊欄就是除錯區塊(Debug)，點擊這個按鈕後，下方即會顯示除錯區塊，如下：



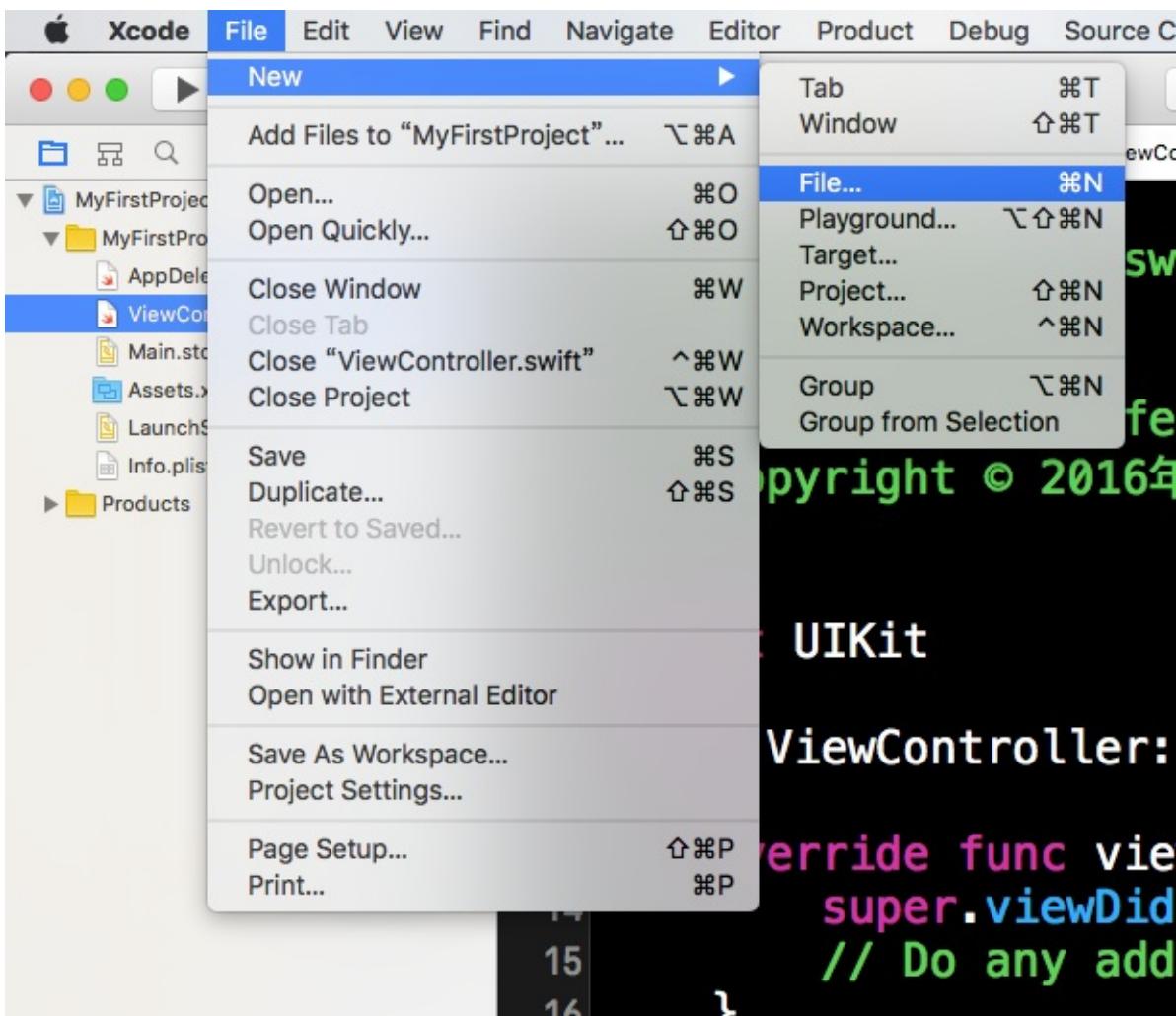
當程式使用 `print()` 印出文字、程式運行中的動作或是發生錯誤，都會顯示詳細資訊在這裡。

新增檔案

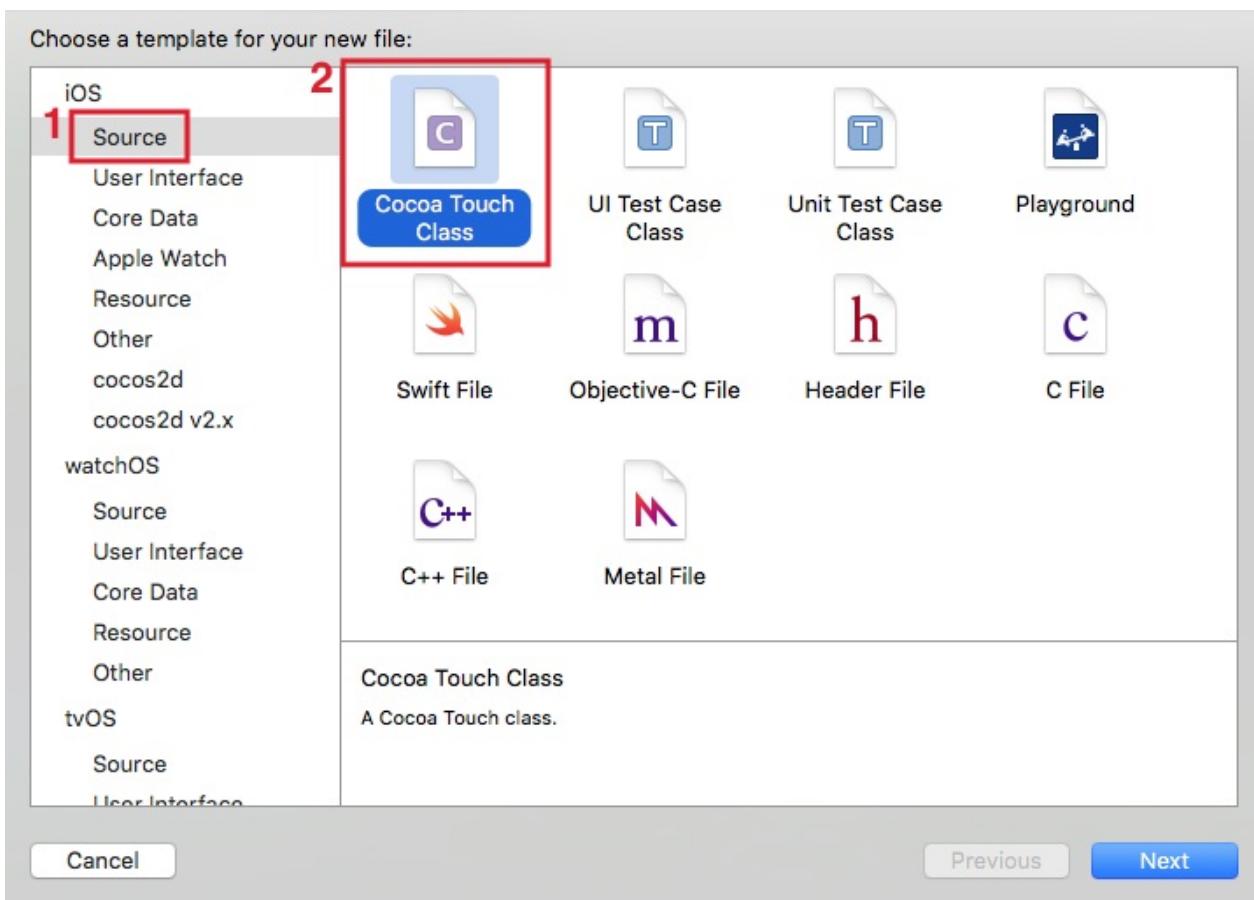
▼ 在左側專案檔案列表中，找到你想要新增檔案的目錄按下右鍵，並點擊 **New File...**，即可新增檔案，如下圖：



▼ 或是你也可以從工具列的 **File > New > File...** 來新增檔案，如下圖：



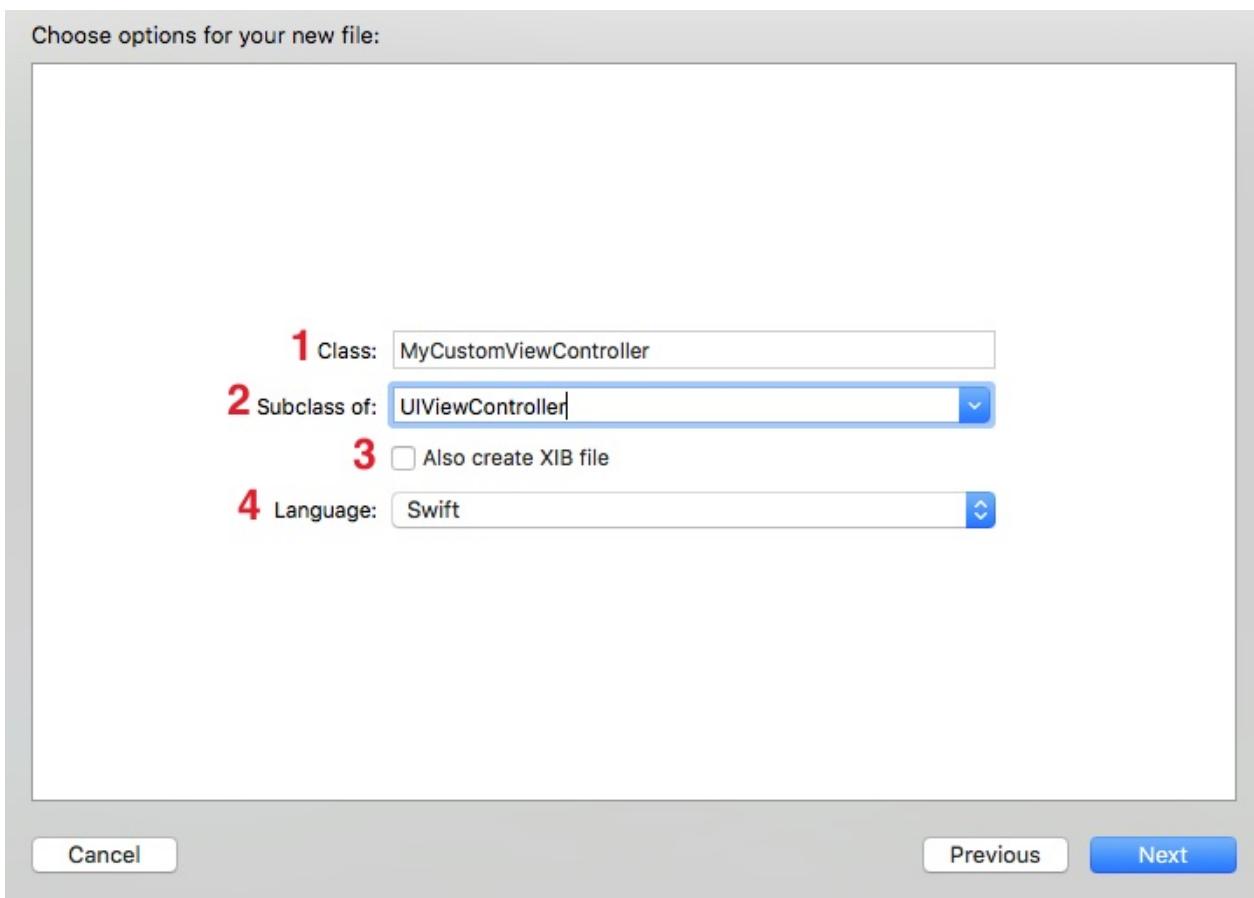
▼ 接著要選擇檔案的類型，這邊示範新增的是一個繼承自 `UIViewController` 的子類別，所以要選擇 `iOS > Source > Cocoa Touch Class` 這個模版的檔案，再點擊 `Next` 按鈕，如下圖：



Hint

- 預設的選擇可能是 OS X > Source > Cocoa Class，記得要再點選前述正確的類型。
- 這些模版檔案都會預設寫好一些程式碼，讓你可以快速的建立新檔案，當然你也可以選擇 iOS > Source > Swift File，這就會建立一個空的 Swift 檔案，再交由你自己寫後續的程式。

▼ 接著需要填寫這個檔案的資訊：



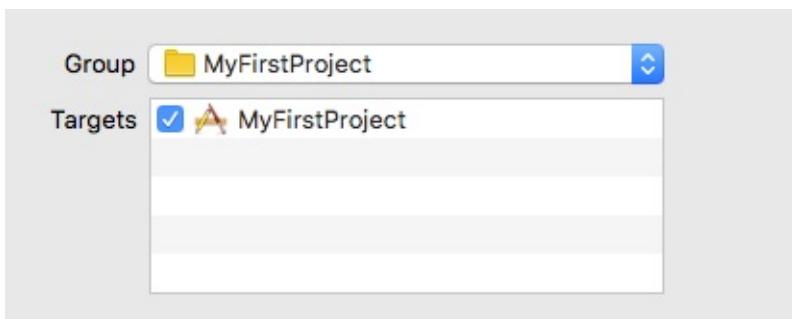
1. **Class**：是這個檔案的名稱，同時也是檔案內部的類別(`Class`)名稱，這裡你可以先填寫 `MyCustom` (或是你要取的名字，記得這邊建議使用[大駝峰式命名法](#))。
2. **Subclass of**：這個類別要繼承自哪一個父類別，因為是要建立一個自定義的 `UIViewController`，所以這邊填寫 `UIViewController`，你應該可以注意到他會提示有哪些可以用的 `UIKit` 元件並會自動補齊名稱，同時也會將前一個欄位補齊為 `MyCustomViewController`。
3. **Also create XIB file**：XIB 檔案是一個介面設計檔案，這邊不勾選。
4. **Language**：選擇 `Swift`。

都填寫完畢後，點擊 `Next` 按鈕繼續。

Hint

- 檔案的命名是習慣上的方式，不是必定要遵循，但還是強烈建議使用一樣的命名方式。
- 繼承自的父類別可以有很多種選擇，你要自定義任何一個元件，像是 `UIButton` 、 `UIImage` 或是 `UITextField` 都可以。

▼ 最後需要選擇儲存檔案的位置，會根據你一開始選擇要新增檔案的目錄為準，當然你也可以再另外選擇新的位置。有一點要注意，下方可以看到有個 Targets 列表可以勾選，記得這個要打勾，其功能是讓這個新增的檔案可以作用於哪一個 Target，最後點擊 Create 即會建立好檔案：



Hint

- 本書內容的每一個範例大多都只會有一個 Target，不過未來遇到不同功能的應用程式時，可能會有多個 Target。
- ▼ 建立好的檔案，可以看到類別名稱及繼承自的父類別，與已經有些預定義好的方法：

```

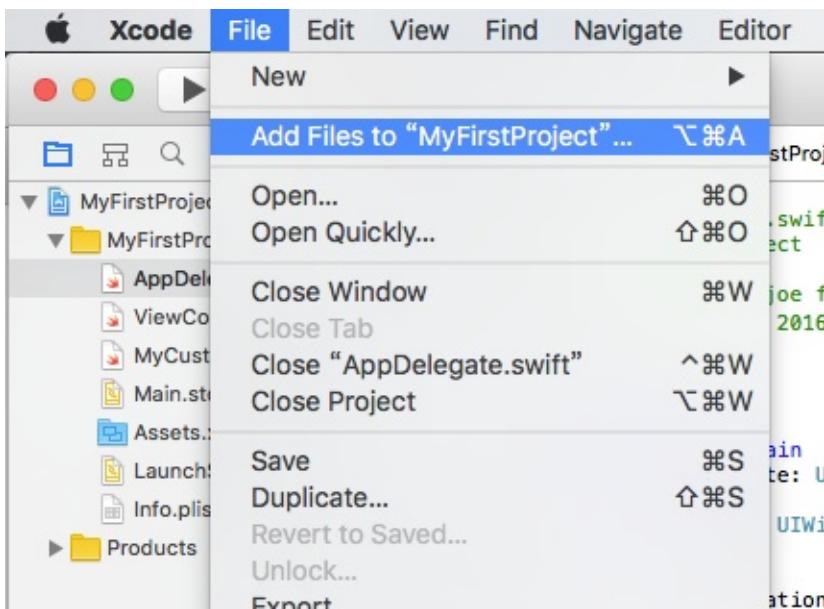
MyFirstProject | Build
MyFirstProject > iPhone
MyFirstProject > MyFirstProject > MyCustomViewController.swift

1 // MyCustomViewController.swift
2 // MyFirstProject
3 // Created by joe feng on 2016/5/10.
4 // Copyright © 2016年 hsin. All rights reserved.
5 //
6 import UIKit
7
8 class MyCustomViewController: UIViewController {
9
10    override func viewDidLoad() {
11        super.viewDidLoad()
12
13        // Do any additional setup after loading the view.
14    }
15
16    override func didReceiveMemoryWarning() {
17        super.didReceiveMemoryWarning()
18        // Dispose of any resources that can be recreated.
19    }
20
21 }
22
23 }
24
25 }

```

加入檔案

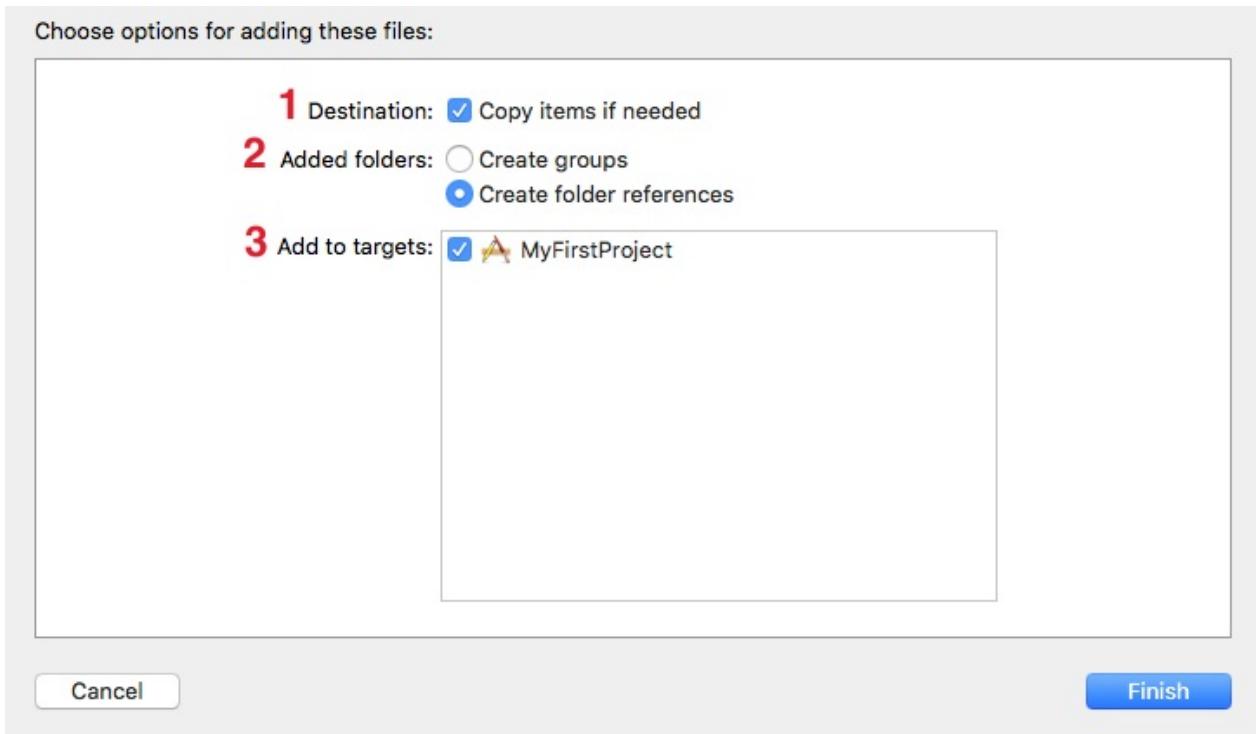
▼ 你可以直接從 Finder 拖曳檔案進 Xcode 左側檔案列表中，或是從工具列的 File > Add Files to "MyFirstProject"... 加入檔案，如下圖：



▼ 接著會詢問這個檔案加入的方式：

1. Destination：是否要將檔案真的複製進專案中，如果勾選就是會將檔案複製進專案，而不勾選時，則是會從此檔案原目錄參考的方式使用，專案目錄裡不會有另一份檔案。例如當你使用一個第三方套件時，可能有多個專案用到，這時如果不想每個專案都放一份檔案時，就可以不勾選。
2. Added folders：以 group 還是 folder 的方式加入檔案的目錄。
3. Add to targets：檔案要作用於哪一個 Target ，必須勾選才會作用。

最後按下 Finish ，就完成了加入檔案的動作。



Hint

- 簡單來說，group (黃色)與 folder (藍色)的不同在於，group 是以參考的方式對這個目錄做操作，是邏輯性的目錄，如果你將這個 group 更名或變動位置，實際的檔案目錄不會變動。而 folder 則是作為被使用的資源，是實際的目錄，對其變動都會實際更動到檔案，且程式編譯時不會被一併加入編譯，而是將整個目錄複製進 Bundle 中，另外如果程式中需要用到這個檔案資源，必須寫出完整的檔案路徑。
- 本書內容的每一個範例大多都只會有一個 Target，不過未來遇到不同功能的應用程式時，可能會有多個 Target。

刪除檔案

▼ 在左側檔案列表中，對檔案按右鍵並點擊 **Delete** 按鈕時，會出現確認視窗，如下圖(**og** 即要刪除的目錄名稱)：



Remove Reference

僅是將對於檔案的參考斷開，不會實際將檔案刪除。對於參考檔案的使用原因及方法，請於[加入檔案](#)章節更進一步了解。

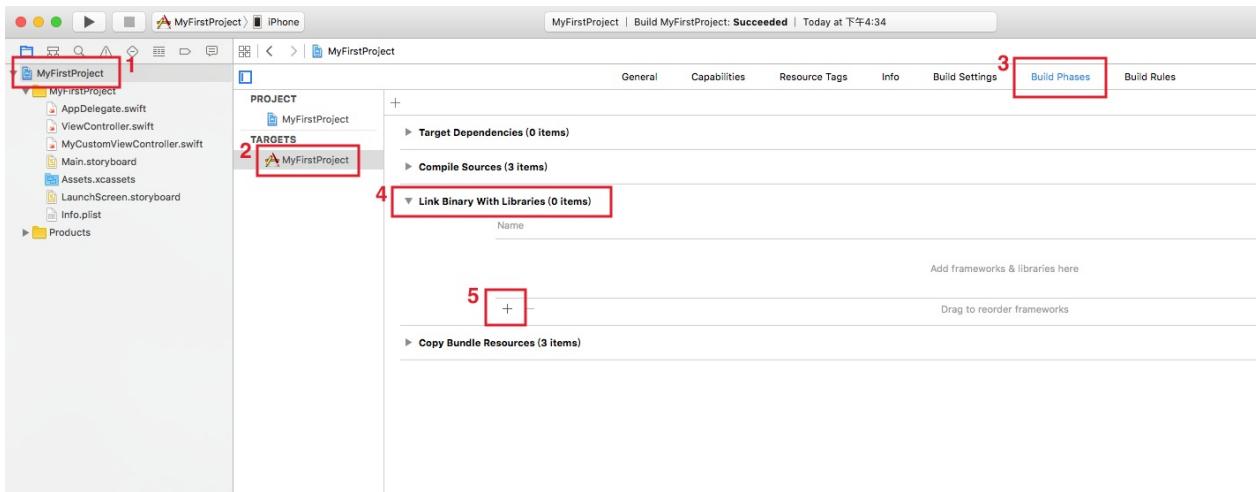
Move to Trash

實際將檔案刪除。

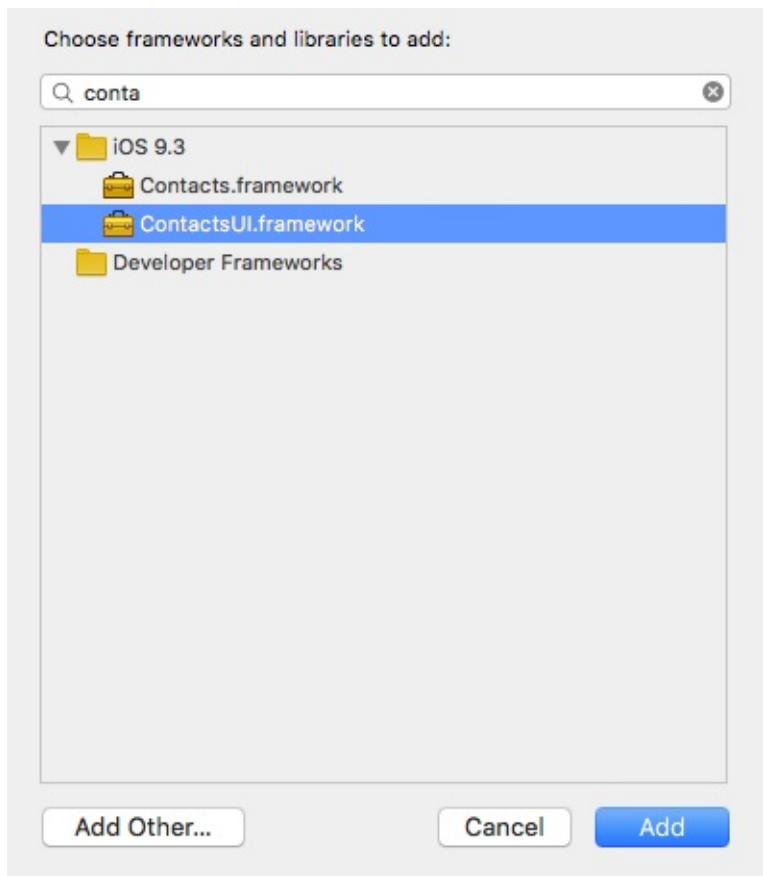
新增 Framework

Xcode 內建了很多功能的 Framework (像是要使用通訊錄、相機或是資料庫等等)，但預設不會先加入專案裡，所以必須依照需求自己將 Framework 加入專案。

▼ 請依照下圖的提示，找到 Framework 的列表所在位置，位於 TARGET > MyFirstProject > Build Phases > Link Binary With Libraries，點擊加號按鈕：



▼ 原先會列出全部可以載入的 Framework，這邊示範要加入通訊錄使用的 ContactsUI.framework，所以在搜尋框中填入 contact，會依據搜尋的文字篩選出 framework，接著點選這個 ContactsUI.framework 並點擊 Add 加入：



▼ 這樣便完成了加入 Framework 的步驟：

▼ Link Binary With Libraries (1 item)

Name

ContactsUI.framework

+ -

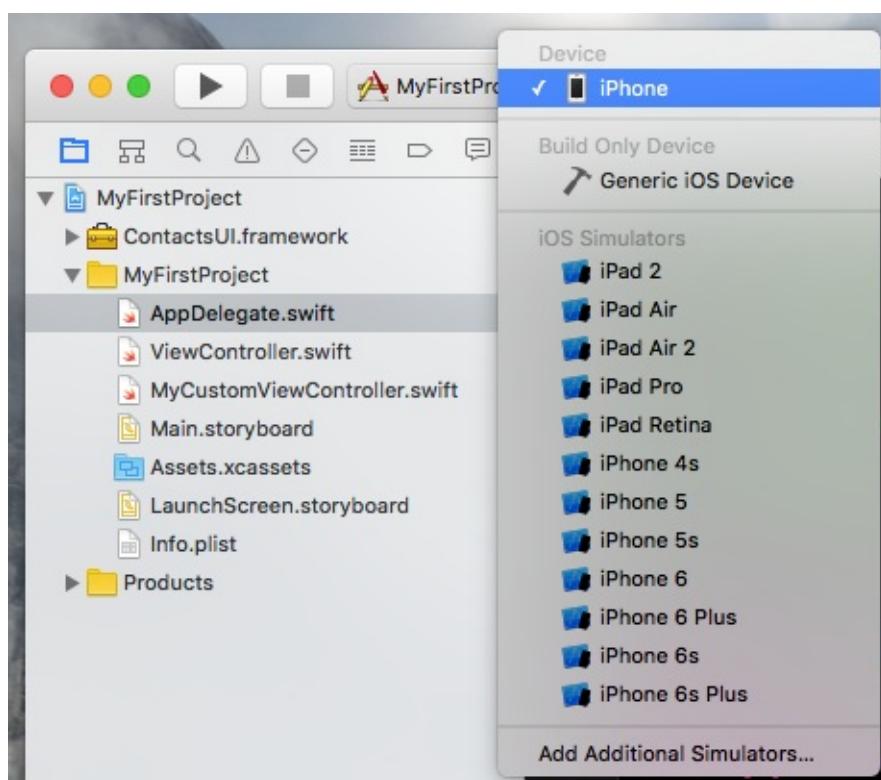
使用模擬器及實機測試

▼ 完成應用程式後，可以直接使用模擬器或實機測試，你可以看到 Xcode 畫面的左上角這一排按鈕，如下圖：



1. 點擊便會開始使用模擬器或實機執行測試。(或你也可以使用熱鍵 cmd + r 執行。)
2. 停止測試。
3. 選擇要執行測試的實機或是模擬器型號。

▼ 在你點擊前圖的第 3 點時，會列出來目前可供測試的實機(Device)與模擬器(iOS Simulators)，如下圖：



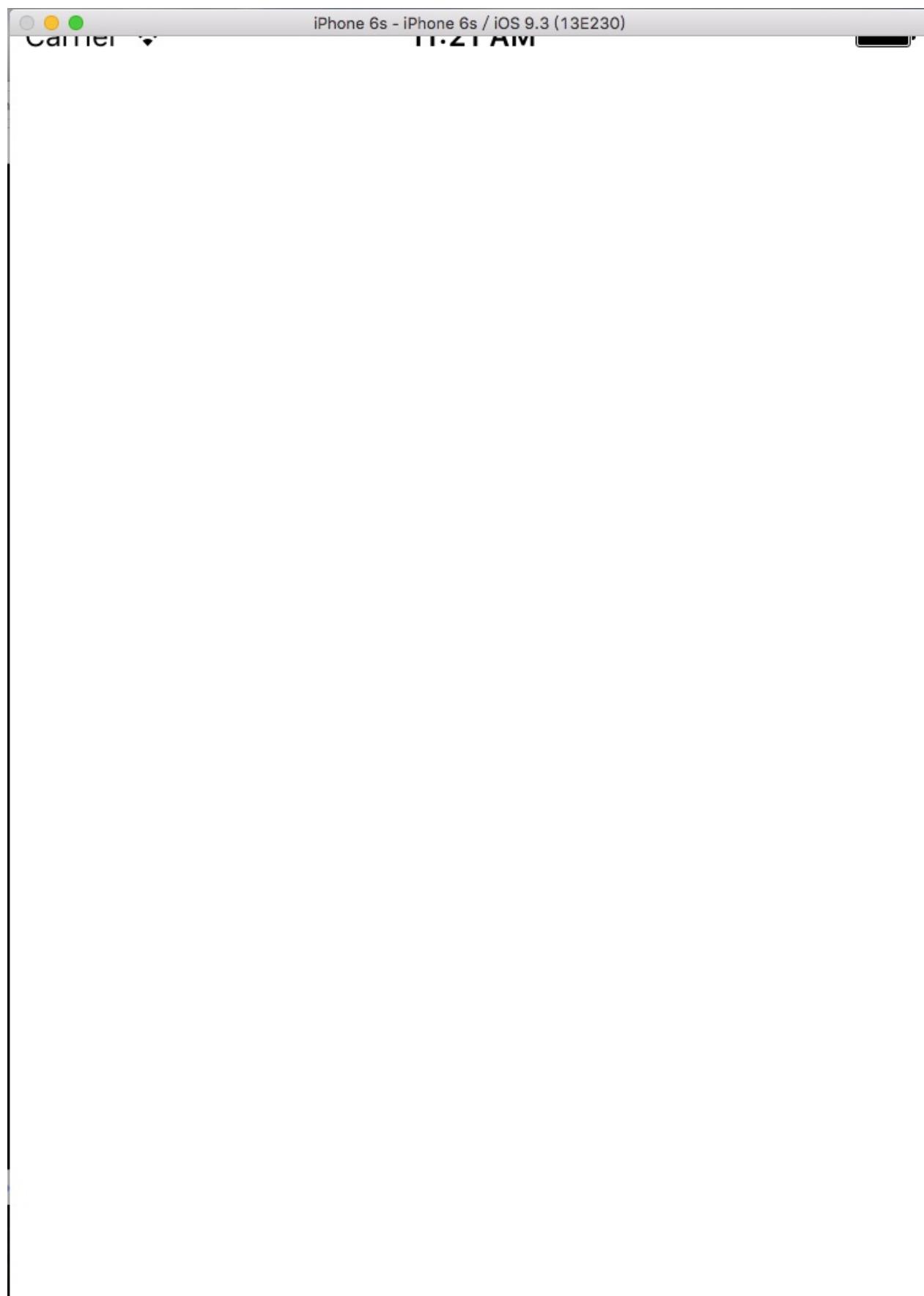
Hint

- 實機名稱不一定為 iPhone，是看你為裝置設定的名稱為何，如果有多台裝置連至 Mac 時，這邊都會全部列出來。
- 實機測試需要登入 Apple ID 。

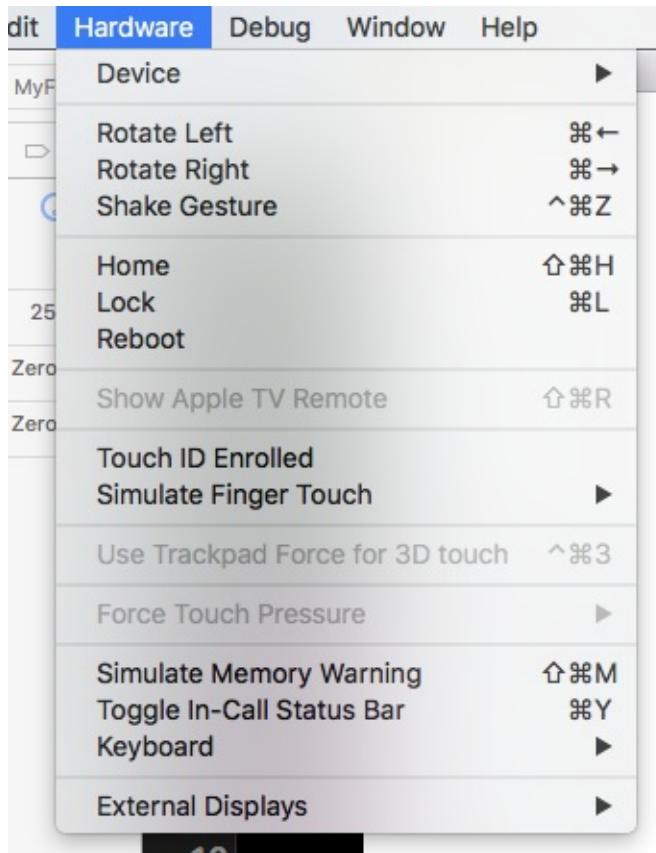
▼ 這邊選擇 iPhone 6S 作為模擬器測試，第一次啓動時會需要初始化，會需要幾秒鐘啓動：



▼ 啓動完後，即會直接進入到應用程式的畫面，目前因為是一個空的專案，所以會顯示空白，如果電腦螢幕不夠大的話，可以上下捲動：

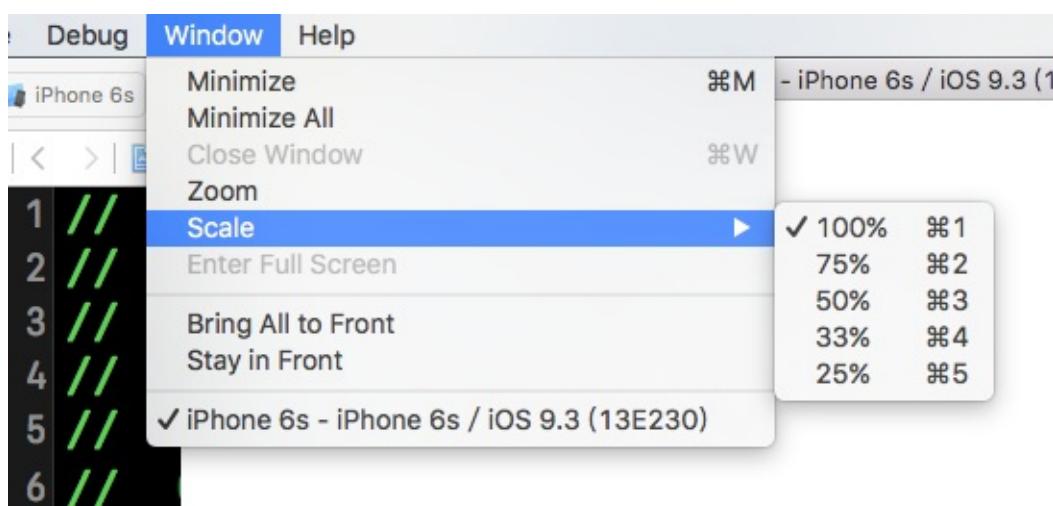


▼ 你可以在 OS X 的工具列中的 Hardware 看到一些用來模擬手機動作的功能，像是你可以把模擬器往左擺(Rotate Left)、往右擺(Rotate Right)或是按下 Home 鍵：

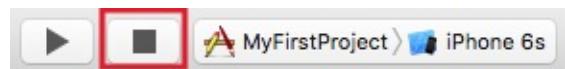


Hint

- 撷取模擬器畫面則是按下 cmd + s ，畫面會存在 OS X 的桌面上。
- ▼ 如果你覺得上下捲動很不方便，可以在 OS X 的工具列中的 Window 看到以下這些功能，你可以將模擬器的視窗縮小為 100% ~ 25%：



▼ 測試完成後要結束時，則是點擊方形按鈕，模擬器就會結束測試：



熱鍵

這邊介紹幾個可以加速開發時間的熱鍵，當然不使用也是可以，但習慣了可以讓你的開發速度更為順暢：

shift + cmd + o

快速打開檔案或是尋找所有檔案內的類別(`class`)、方法(`method`)。

ctrl + 6

快速尋找目前所在檔案內的方法(`method`)。

shift + cmd + j

顯示目前所在檔案位於整個專案的位置，左側邊欄會展開並高亮度這個檔案位置。

cmd + /

將多行程式反白後按下 `cmd + /`，會使用 `//` 將每一行註解起來，在開發階段測試時很有用，已註解起來的多行程式再按一次 `cmd + /` 就會將每個註解去掉。

系統關鍵字

Swift 系統內建的關鍵字有以下幾種類型：

Declarations

associatedtype, class,_deinit, enum, extension, func, import, init, inout, internal, let, operator, private, protocol, public, static, struct, subscript, typealias, var.

Statements

break, case, continue, default, defer, do, else, fallthrough, for, guard, if, in, repeat, return, switch, where, while.

Expressions and types

as, catch, dynamicType, false, is, nil, rethrows, super, self, Self, throw, throws, true, try, #column, #file, #function, #line.

Particular contexts

associativity, convenience, dynamic, didSet, final, get, infix, indirect, lazy, left, mutating, none, nonmutating, optional, override, postfix, precedence, prefix, Protocol, required, right, set, Type, unowned, weak, willSet.

命名

不建議使用系統內建的關鍵字來命名變數、常數、函式、類別及其他自定義的型別，如果真的一定要使用的話，必須以一對反引號 ` ` 把名稱包起來，如下：

```
// 無法這樣命名 這行會報錯誤
let class = 20

// 必須以一對反引號 ` 包起名稱
var `class` = 12

// 使用也必須加上反引號
print("It is \(`class`)")
```

駝峰式命名法

Swift 在命名常數、變數、函式、類別或其他自定義型別時，通常習慣使用駝峰式命名法。

這種命名方式是一種習慣，沒有絕對與強制，為的是增加識別與可讀性。

當自定義名稱是由二個或多個單字連結在一起，而構成的唯一識別字時，單字之間不以空格、連結號(-)或底線(_)隔開，有兩種格式：

小駝峰式命名法(`lower camel case`)

第一個單字以小寫字母開始，第二個及之後的單字的首字母則使用大寫，像是：`firstName` 、`somePerson` 。

Swift 中通常命名常數、變數、函式、屬性、方法及下標時，會使用小駝峰式命名法。

大駝峰式命名法(`upper camel case`)

每個單字的首字母都使用大寫字母，像是 `LastName` 、`SomeClass` 。

Swift 中通常名列舉、結構、類別、擴展、協定及其他自定義型別時，會使用大駝峰式命名法。

關於本書

封面圖片來源

- <http://barnimages.com/iphone-and-macbook-pro/>
- http://www.freepik.com/free-vector/minimalist-mosaic-background_758288.htm

版本歷史

v1.0.2

更新部分文字說明及修正錯誤

v1.0.1

更新各頁連結使用的錨點語法

v1.0.0

初版