Swift程式設計入門

1. 學習前的準備與認知

Swift是一個非常年輕的程式語言，自2014年6月的Apple Worldwide Developers Conference (Apple WWDC蘋果全球開發者大會)上推出之後，至今不過三年左右的時間Apple公司推出Swift的最主要目的，就是希望由它逐漸取代Objective-C成為新一代的「蘋果程式語言」。所謂的蘋果程式語言，是指它可用在蘋果公司的桌機(iMacMac Mini與Mac Pro)筆電(Mac BookMac Book Pro與Mac Book Air)行動電話(iPhone)智慧型手錶(Apple Watch)平板電腦(iPad)乃至於智慧聯網電視(Apple TV)等所有產品上，進行應用程式的開發。

本章將就學習Swift語言前的準備工作以及相關的認知進行說明，其中包含相關的程式開發工具的安裝、互動式的直譯環境，以及簡單的Swift程式範例。

1.1 安裝與準備

由於Swift是由Apple公司所推出的全新的程式語言，很自然地它可以在Mac OS環境中進行開發，只要安裝有Xcode即可以進行Swift的程式開發。其基本需求如下：

- 安裝有Mac OS X 10.10 (Yosemite)或以上的Macintosh電腦；
- Xcode 7或以上版本；
- 加入Applet Developer Program蘋果開發者計劃）

以上所列舉的只是最低的要求，建議讀者還是先將系統及Xcode都升級到最新版本，再開始Swift的學習。

如果說Swift是「蘋果程式語言」，那麼在Mac OS系統上的Xcode就可以視為是「蘋果開發工具」，我們可以在Xcode裡完成所有蘋果相關產品的軟體開發。在開始學習Swift程式語言之前，請先在你的系統上安裝好最新版本的Xcode(或是將其更新至最新版本)。目前Xcode提供我們三種不同的方式來進行Swift程式的開發，分別是：

- 直譯式的互動環境REPLXcode所提供的一個獨立的工具，開發人員可以在這個類似直譯式的互動環境中，進行Swift的程式設計並且可以得到即時性的運作結果，有助於學習及瞭解程式的語法以及敘述的意義與作用，常用於初學者學習之用。
- PlaygroundPlayground與REPL相似，同樣可以提供即時的程式碼運作結果。但是Playground是Xcode所提供的一個開發選項，它提供了更為完整的即時運作結果給開發者參考，並且可以使用在Xcode中的程式編輯器的程式碼自動完成(code completion)與語法突顯(syntax highlight)等功能。使用Playground所撰寫的程式碼，也可以移轉到相關的軟體專案中繼續使用。
- 應用程式開發：我們也可以在Xcode中使用Swift語言來進行Mac OSiOSwatchOS與tvOS等應用程式的開發。畢竟做為新一代的「蘋果程式語言Swift的目的就是要做為Apple公司全產品的應用軟體開發語言。

本書將從Swift程式設計基礎出發，以REPL或Playground來進行相關的學習，並於後續使用Xcode來進行進階的Mac OS與iOS的應用程式設計。

<note> 如果你曾經在系統上安裝有一個以上的Xcode為避免使用到舊的Xcode版本，請在Terminal中輸入下列指令，以確保預設的Xcode為目前安裝在「應用程式/Applications資料夾中的Xcode

```
sudo xcode-select -s /Applications/Xcode.app/Contents/Developer/
```

</note>

<note tip> 其實，除了Mac OS系統外，Swift也有推出Linux的版本，有興趣的讀者請自行參閱Swift官方網站以取得進一步的說明。至於使用Windows的開發者也不必擔心，也有第三方的工具可以幫助您在Windows系統上開發Swift應用程式，例如Swift for Windows就是一個可以在Windows系統上開發Swift語言應用程式的工具。 </note>

1.2 REPL運行環境

REPL是Read-Eval-Print-Loop的縮寫，可以翻譯為「讀取-執行-輸出-循環」，是一種直譯式的Swift運行環境，是初學者最常用以學習Swift的入門環境。

請在Terminal中輸入以下指令[1)]，以啟動Swift的REPL

```
xcrun swift
```

同樣要啟動REPL，也可以直接在Terminal中輸入「swift」

```
swift
```

首次執行時，將會出現如figure 1的要求授權畫面，請輸入你的帳號及密碼以完成此授權:

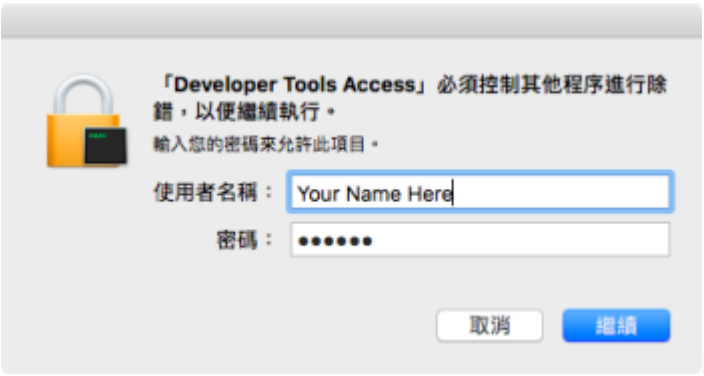

Fig. 1: 授權REPL執行的畫面

如果一切順利的話，你將會看到以下的畫面：

```
Welcome to Apple Swift version 3.1 (swiftlang-802.0.48 clang-802.0.48). Type
:help for assistance.
  1>
```

這個REPL的環境提供了一個反覆運行的循環(Loop)，每次「讀取(Read)」一個指令加以「執行(Eval)」，然後將結果「輸出(Print)」，這就是類似「Mac OS」Linux系統的Terminal」或是「Windows系統中的文字命令列模式」的操作方式。在每個Loop開始時，REPL都會顯示一個提示字串（其中包含一個數字與一個 > 符號），當我們輸入相關的指令並按下Enter後，該指令就會被加以「讀取(Read)」，然後經過「執行(Eval)」後，將

結果「輸出(Print)」出來」REPL將反覆進行此」Read-Eval-Print」的「循環(Loop)」」直到結束為止。其中在每個Loop的提示字串中的數字，就代表截至目前為止所進行的Loop次數。

1.2.1 第一個指令:help

我們要學習的第一個REPL的指令，就是可以顯示REPL的操作輔助說明文件的」:help」指令，請參考下面的畫面：

```
$ xcrun swift
Welcome to Apple Swift version 3.1 (swiftlang-802.0.48 clang-802.0.48). Type
:help for assistance.
  1> :help

The REPL (Read-Eval-Print-Loop) acts like an interpreter.  Valid statements,
expressions, and declarations are immediately compiled and executed.

The complete set of LLDB debugging commands are also available as described
below.
Commands must be prefixed with a colon at the REPL prompt (:quit for
example.)
Typing just a colon followed by return will switch to the LLDB prompt.

Debugger commands:

  apropos           -- List debugger commands related to a word or subject.
  breakpoint        -- Commands for operating on breakpoints (see 'help b'
for
                       shorthand.)
  bugreport         -- Commands for creating domain-specific bug reports.
  command           -- Commands for managing custom LLDB commands.
  disassemble       -- Disassemble specified instructions in the current
target.
                       Defaults to the current function for the current
thread and
                       stack frame.
  expression        -- Evaluate an expression on the current thread.
Displays any
                       returned value with LLDB's default formatting.
  frame             -- Commands for selecting and examing the current
thread's
                       stack frames.
  gdb-remote        -- Connect to a process via remote GDB server.  If no
host is
                       specifed, localhost is assumed.
  gui               -- Switch into the curses based GUI mode.
  help              -- Show a list of all debugger commands, or give details
about
```

```
                        a specific command.
  kdp-remote        -- Connect to a process via remote KDP server.  If no
UDP port
                        is specified, port 41139 is assumed.
  language          -- Commands specific to a source language.
  log               -- Commands controlling LLDB internal logging.
  memory            -- Commands for operating on memory in the current
target
                        process.
  platform          -- Commands to manage and create platforms.
  plugin            -- Commands for managing LLDB plugins.
  process           -- Commands for interacting with processes on the
current
                        platform.
  quit              -- Quit the LLDB debugger.
  register          -- Commands to access registers for the current thread
and
                        stack frame.
  script            -- Invoke the script interpreter with provided code and
                        display any results.  Start the interactive
interpreter if
                        no code is supplied.
  settings          -- Commands for managing LLDB settings.
  source            -- Commands for examining source code described by debug
                        information for the current target process.
  target            -- Commands for operating on debugger targets.
  thread            -- Commands for operating on one or more threads in the
                        current process.
  type              -- Commands for operating on the type system.
  version           -- Show the LLDB debugger version.
  watchpoint        -- Commands for operating on watchpoints.

Current command abbreviations (type ':help command alias' for more info):

  add-dsym  -- Add a debug symbol file to one of the target's current
modules by
               specifying a path to a debug symbols file, or using the
options to
               specify a module to download symbols for.
  attach    -- Attach to process by ID or name.
  b         -- Set a breakpoint using one of several shorthand formats.
  bt        -- Show the current thread's call stack.  Any numeric argument
               displays at most that many frames.  The argument 'all'
displays all
               threads.
  c         -- Continue execution of all threads in the current process.
  call      -- Evaluate an expression on the current thread.  Displays any
               returned value with LLDB's default formatting.
  continue  -- Continue execution of all threads in the current process.
  detach    -- Detach from the current target process.
  di        -- Disassemble specified instructions in the current target.
```

```
Defaults
               to the current function for the current thread and stack
frame.
  dis        -- Disassemble specified instructions in the current target.
Defaults
               to the current function for the current thread and stack
frame.
  display    -- Evaluate an expression at every stop (see 'help target stop-
hook'.)
  down       -- Select a newer stack frame.  Defaults to moving one frame, a
               numeric argument can specify an arbitrary number.
  env        -- Shorthand for viewing and setting environment variables.
  exit       -- Quit the LLDB debugger.
  f          -- Select the current stack frame by index from within the
current
               thread (see 'thread backtrace'.)
  file       -- Create a target using the argument as the main executable.
  finish     -- Finish executing the current stack frame and stop after
returning.
               Defaults to current thread unless specified.
  image      -- Commands for accessing information for one or more target
modules.
  j          -- Set the program counter to a new address.
  jump       -- Set the program counter to a new address.
  kill       -- Terminate the current target process.
  l          -- List relevant source code using one of several shorthand
formats.
  list       -- List relevant source code using one of several shorthand
formats.
  n          -- Source level single step, stepping over calls.  Defaults to
current
               thread unless specified.
  next       -- Source level single step, stepping over calls.  Defaults to
current
               thread unless specified.
  nexti      -- Instruction level single step, stepping over calls.  Defaults
to
               current thread unless specified.
  ni         -- Instruction level single step, stepping over calls.  Defaults
to
               current thread unless specified.
  p          -- Evaluate an expression on the current thread.  Displays any
               returned value with LLDB's default formatting.
  parray     -- Evaluate an expression on the current thread.  Displays any
               returned value with LLDB's default formatting.
  po         -- Evaluate an expression on the current thread.  Displays any
               returned value with formatting controlled by the type's
author.
  poarray    -- Evaluate an expression on the current thread.  Displays any
```

```
                returned value with LLDB's default formatting.
  print     -- Evaluate an expression on the current thread.  Displays any
                returned value with LLDB's default formatting.
  q         -- Quit the LLDB debugger.
  r         -- Launch the executable in the debugger.
  rbreak    -- Sets a breakpoint or set of breakpoints in the executable.
  repl      -- Evaluate an expression on the current thread.  Displays any
                returned value with LLDB's default formatting.
  run       -- Launch the executable in the debugger.
  s         -- Source level single step, stepping into calls.  Defaults to
current
                thread unless specified.
  si        -- Instruction level single step, stepping into calls.  Defaults
to
                current thread unless specified.
  sif       -- Step through the current block, stopping if you step directly
into
                a function whose name matches the TargetFunctionName.
  step      -- Source level single step, stepping into calls.  Defaults to
current
                thread unless specified.
  stepi     -- Instruction level single step, stepping into calls.  Defaults
to
                current thread unless specified.
  t         -- Change the currently selected thread.
  tbreak    -- Set a one-shot breakpoint using one of several shorthand
formats.
  undisplay -- Stop displaying expression at every stop (specified by stop-
hook
                index.)
  up        -- Select an older stack frame.  Defaults to moving one frame, a
                numeric argument can specify an arbitrary number.
  x         -- Read from the memory of the current target process.

For more information on any command, type ':help <command-name>'.
  1>
$
```

1.2.2 結束與離開的指令

在前面的:help指令中，你有沒有找到用以「結束並離開REPL的指令？以及其對應的短指令？ 沒錯，用以結束並離開REPL環境的指令就是:quit以及其對應的短指令:q]不過要記得REPL的指令必須以「:」開頭，請參考下面的Terminal操作畫面，在啟動後輸入:quit或「:q來結束REPL

```
$ xcrun swift
Welcome to Apple Swift version 3.1 (swiftlang-802.0.48 clang-802.0.48). Type
:help for assistance.
  1> :quit
$ swift
```

```
Welcome to Apple Swift version 3.1 (swiftlang-802.0.48 clang-802.0.48). Type
:help for assistance.
  1> :q
$
```

1.2.3 Hello REPL

本節將以一個簡單的例子出發，讓大家先寫出一個可以輸出「Hello REPL!」的Swift程式。

```
$ xcrun swift
Welcome to Apple Swift version 3.1 (swiftlang-802.0.48 clang-802.0.48). Type
:help for assistance.
  1> print("Hello REPL!")
Hello REPL!
  2>
```

這就是一個最簡單的Swift程式，有沒有發現連一般程式語言慣有的「Entry Point」[2]也沒有，可以直接撰寫相關的程式敘述(statement)「例如上述例子中的「print(“Hello Swift!”)」就是一行敘述。要特別注意的是Swfit和許多程式語言不同，分號(;)不再需要添加在每個敘述的結尾處[3]」

1.3 Xcode Playground

除了REPL之外「Xcode也有提供一個直譯式的互動環境讓我們開發及測試Swift程式，這個環境就叫作「Playground(遊樂場)[4]」。請參考figure 2的畫面，你可以在此選擇「Get Started with a Playground」選項以啟動一個Playground:

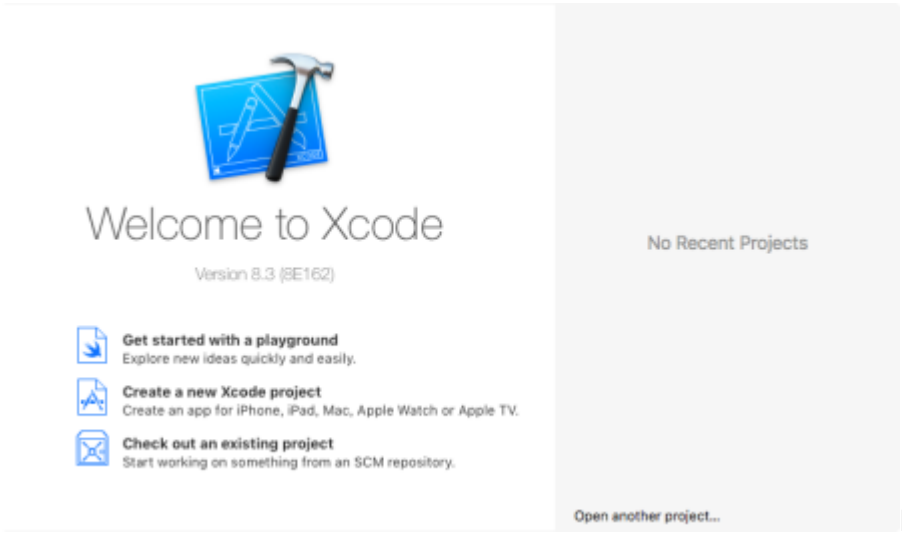

Fig. 2: 授權REPL執行的畫面

接者請參考figure 3，為你的Playground取個適合的名稱，例如「MyPlayground」並接著為它選擇一個適當的儲存位置:

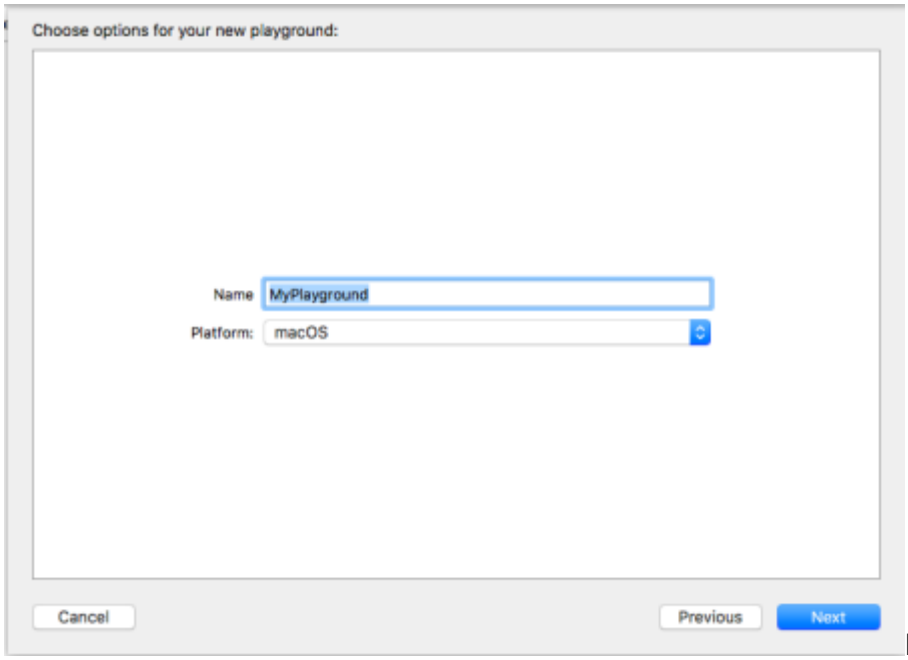

Fig. 3: 命名Playground

這些都完成後，你就會得到如figure 4的畫面：

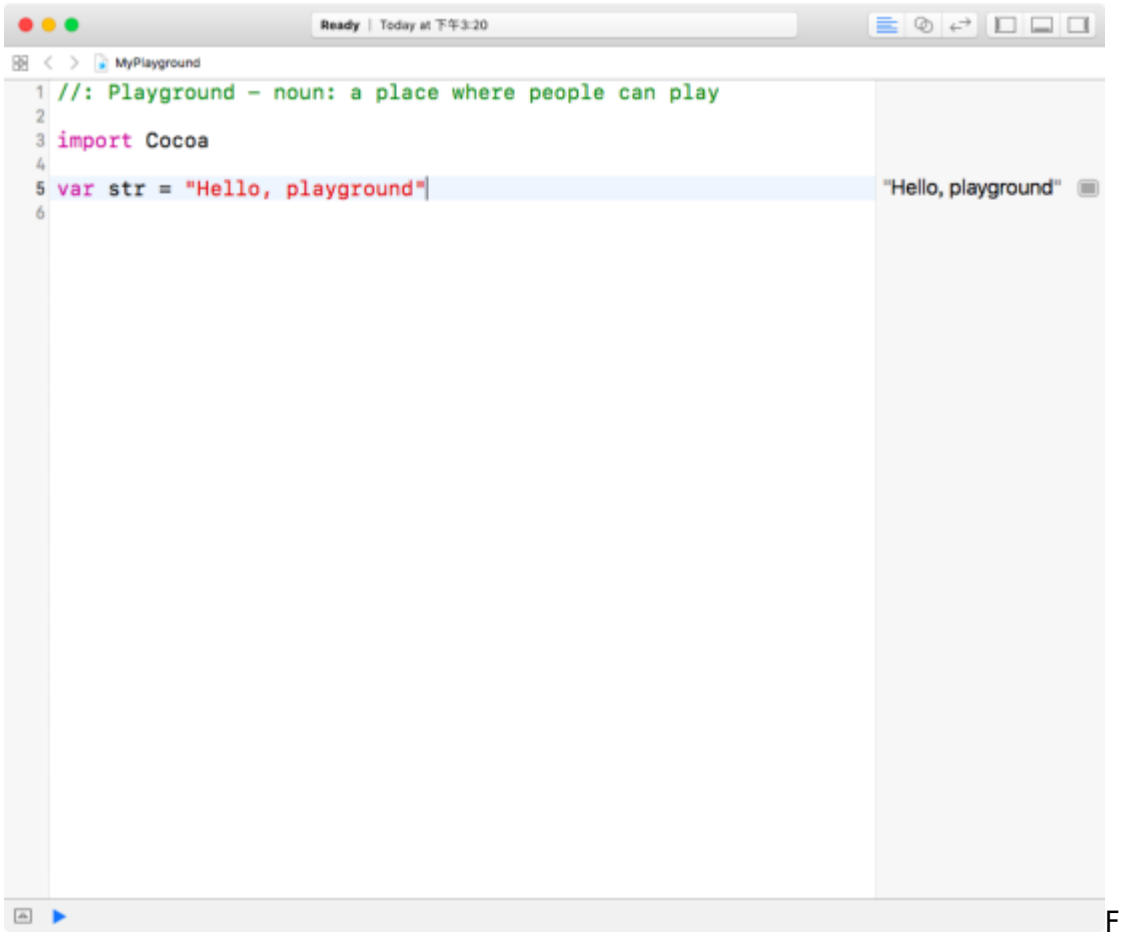

Fig. 4: Playground執行畫面

從figure 4中可以看到，它已經有如下的Swift程式在其中：

```
//: Playground - noun: a place where people can play
```

```
import Cocoa

var str = "Hello, playground"
```

為了便利起見，我們將會使用類似註解的方式，在需要呈現即時運算結果的程式碼旁，以「//|」開頭標示其即時的運算結果。例如上面的程式碼將可以呈現如下：

```
//: Playground - noun: a place where people can play

import Cocoa

var str = "Hello, playground"  //| "Hello, playground"
```

你可以發現Playground比起REPL更能夠提供完整的程式碼的開發與測試環境。舉例來說REPL只是一行一行地將程式碼輸入，並得到立即的運算結果；反觀Playground則提供一個完整的環境，你可以用以開發設計一個完整的程式，可以來回地修改程式，並且可以在畫面的右側得到即時性的運算結果。請注意在第3行的import Cocoa是要載入名為Cocoa的框架，這是用以撰寫Mac OS的視窗程式的框架。當然目前暫時還不會用到這個框架的內容，你可以先加以忽略。

請試著將上述程式碼修改如下：

```
//: Playground - noun: a place where people can play

import Cocoa

var str = "Hello, playground"  //| "Hello, playground"
print(str)                     //| "Hello, playground\n"
```

有沒有發現所增加的這一行print(str)所產生的運算結果和前一行的var str=“Hello, playground”非常相似，只相差一個n而已。不過第5行是一個變數宣告的敘述[5]，因此其右側顯示的結果是該變數的數值；但第6行是一個print()函式的執行，其右側所顯示的是其輸出的結果，因此此處會比起前一行多了一個\n代表其輸出包含一個換行的意思。

1.4 LLVM

---

[1] xcrun代表Xcode run也就是啟動或執行Xcode的相關工具。
[2] Entry point就是指程式開始執行的起點，又被稱為進入點，例如C或C++語言的main function
[3] 但是你還是可以在同一行中使用分號(;)來區別兩個以上的敘述。
[4] 哈~就是這麼歡樂啊!
[5] 關於變數宣告將在後續的章節中為你詳細的說明