

目錄

前言	1.1
Selector	1.2
Objective-C Class/Object 到底是什麼？	1.2.1
Selector 有什麼用途？	1.2.2
呼叫 performSelector: 需要注意的地方	1.2.3
Selector 是 Objective-C 中所有魔法的開始	1.2.4
相關閱讀	1.2.5
練習：小計算機	1.2.6
Category	1.3
什麼時候應該要使用 Category	1.3.1
實作 Category	1.3.2
Category 還可以有什麼用途？	1.3.3
Extensions	1.3.4
Category 是否可以增加新的成員變數或屬性？	1.3.5
對 NSURLSessionTask 撰寫 Category	1.3.6
相關閱讀	1.3.7
練習：字串反轉	1.3.8
記憶體管理 Part 1 - Retain、Release	1.4
Reference Count/Retain/Release	1.4.1
Auto-release	1.4.2
基本原則	1.4.3
Getter/Setter 與 Property 語法	1.4.4
相關閱讀	1.4.5
記憶體管理 Part 2 - ARC	1.5
記憶體管理 Part 3 - Memory Warnings	1.6
Delegate 與 Protocol	1.7
從其他平台來看 Objective-C 的 Delegate	1.7.1
設計 Protocol 與實作 Delegate 的方式	1.7.2
注意事項	1.7.3
Data Source 與 Delegate 的差別？	1.7.4
Formal Protocol 與 Informal Protocol	1.7.5
無所不在的 Delegate	1.7.6
其他平台上所謂的 Delegate	1.7.7
我們曾經犯過的低級錯誤	1.7.8

相關閱讀	1.7.9
練習：貪食蛇	1.7.10
單元測試	1.8
AAA 原則	1.8.1
執行測試	1.8.2
測試驅動開發 (TDD)	1.8.3
覆蓋率	1.8.4
相關閱讀	1.8.5
Blocks	1.9
Block 的語法	1.9.1
Block 如何代替了 Delegate	1.9.2
Block 與 Delegate 都可以想成是 Event Handler	1.9.3
什麼時候該用 Blocks？什麼時候該用 Delegate？	1.9.4
block 與 weak 關鍵字	1.9.5
Block 作為 Objective-C 物件	1.9.6
哪些事情不要拿 Block 來做	1.9.7
Callback Hell	1.9.8
相關閱讀	1.9.9
練習：將 Web Service API 包裝成 SDK	1.9.10
練習：將 Web Service API 包裝成 SDK Part2	1.9.11
NotificationCenter	1.10
接收與發送 Notification	1.10.1
Notification Queue	1.10.2
Mac 上的其他 Notification Center	1.10.3
CFNotificationCenter	1.10.4
相關閱讀	1.10.5
練習：填字遊戲	1.10.6
練習：自行實作 NSNotificationCenter	1.10.7
所謂的設計模式	1.11
圖解設計模式	1.11.1
為什麼要使用設計模式	1.11.2
再談 Singleton	1.11.3
練習：探索 Cocoa/Cocoa Touch Framework	1.11.4
練習：閱讀程式碼	1.11.5
一些新手常常搞混的東西	1.12
bool 與 BOOL	1.12.1
NSInteger 與 NSUInteger	1.12.2

NULL、nil、Nil...	1.12.3
Responder	1.13
RunLoop	1.13.1
Application	1.13.2
Window	1.13.3
View	1.13.4
ViewController	1.13.5
UITouch	1.13.6
相關閱讀	1.13.7
Threading	1.14
performSelector:	1.14.1
GCD	1.14.2
NSOperation 與 NSOperationQueue	1.14.3
相關閱讀	1.14.4
練習：一個發送多個連線的 Operation	1.14.5
NSCoding	1.15
實作 NSCoding	1.15.1
常見用途	1.15.2
相關閱讀	1.15.3
Crash Reports	1.16
如何收集 Crash Reports	1.16.1
記憶體不足時產生的 Crash Report	1.16.2
Crash Reports 的三部分	1.16.3
解開記憶體位置	1.16.4
常見 Crash 的類型	1.16.5
禪與 App 維修的藝術	1.16.6
實戰：Bad Access	1.16.7
實戰：因為 Category 造成的 Crash	1.16.8
Crash Report	1.16.8.1
實戰：JB 造成的 Exception	1.16.9
Crash Report	1.16.9.1
實戰：背景作業執行太久	1.16.10
Crash Report	1.16.10.1
相關閱讀	1.16.11
練習：閱讀 Crash Reports	1.16.12
Core Animation	1.17
CALayer	1.17.1

CATransaction	1.17.2
CAAnimation	1.17.3
CAAnimation 的細部設定	1.17.4
CADisplayLink	1.17.5
在 Mac 上使用 Core Animation	1.17.6
教學影片	1.17.7
練習：KKBOX 動態歌詞	1.17.8
練習：Flappy Bird	1.17.9
練習：自由發揮	1.17.10
Audio API	1.18
基礎知識	1.18.1
iOS 與 Mac OS X 的 Audio API 概觀	1.18.2
使用 Audio Queue 開發播放軟體	1.18.3
使用 Audio Unit Processing Graph 開發播放軟體	1.18.4
在 AUGraph 中串接 AudioUnit	1.18.5
修改 LPCM 資料	1.18.6
Core Audio 到底難在哪？	1.18.7
使用 AVAudioEngine 開發播放軟體	1.18.8
從 Audio Unit Processing Graph 到 AVAudioEngine	1.18.9
Audio Session	1.18.10
打造 Player 的完整功能	1.18.11
CarPlay	1.18.12
MFI 助聽器	1.18.13
教學影片	1.18.14
相關閱讀	1.18.15
練習：iOS Audio Player 開發	1.18.16
Auto Layout	1.19
什麼是約束 (NSConstraint)	1.19.1
固有內容尺寸 (Intrinsic Content Size)	1.19.2
UIScrollView 與 Auto Layout	1.19.3
.xib 在 UIViewController lifecycle 的陷阱	1.19.4
Self-sizing UITableViewCell height	1.19.5
Animation with Auto Layout	1.19.6
Accessibility	1.20
如何使用 VoiceOver	1.20.1
基本支援	1.20.2
VoiceOver 的方向	1.20.3

進階的 Accessibility 設定	1.20.4
相關閱讀	1.20.5
練習：KKBOX 動態歌詞	1.20.6
Mac 上的 WebKit	1.21
用 Objective-C 操作 DOM	1.21.1
JavaScript 與 Objective-C 的溝通	1.21.2
JavaScriptCore	1.21.3
小結	1.22
其他閱讀資料	1.23
版本記錄	1.24

KKBOX iOS/Mac OS X 基礎開發教材

2019 年前言

這份教材在 2015 年時釋出，當中主要累積了我們從 2008 年到 2015 年這幾年之間的開發經驗，以及 2011 年到 2015 年之間的內部訓練教材。從 2015 年開始，蘋果平台上的相關技術也都變化巨大，我們也不見得有時間可以更新教材，所以在您閱讀這份教材的時候，我希望您可以注意：

- 2015 年的時候 Swift 語言還並不穩定，而我們內部所開發的各種應用，都還是以 Objective-C 為主，所以這份教材花了大量時間解釋 Objective-C runtime 以及慣用的 pattern，從 2019 年回頭來看，隨著 Swift 語言與 ARC 的成熟，像是很多記憶體管理相關的知識，其實已經不需要這麼多的篇幅，在最前面的幾章其實您可以選擇跳過。
- 由於 UIKit 是用 Objective-C 寫的，所以也大量使用 Objective-C 慣用的 pattern，像是 Singleton、Delegate、Notification Center、KVO 等，不過，隨著 Swift 5.1 與 SwiftUI、Combine 等純 Swift 的 framework 的出現，蘋果也慢慢走向宣告式、React 式的程式設計典範，但如果你還是需要用到 UIKit，即使你寫的是 Swift，我還是希望這份教材中講解的 pattern 可以對您有所幫助。
- 我們在公司內部已經大量使用 Swift，不過由於跟 Swift 有關的資源眾多，我們也就沒有特地加上與 Swift 有關的教材，所以最近的一些更新還是因應我們內部的特殊需求為主，像是與音訊處理相關的部份。

簡繁用語對應

由於我們身在台灣，在這本書當中也採用台灣慣用的技術名詞，所以如果您習慣簡體中文，可能會不習慣當中的一些名詞。大抵上的對應如下：

- 實作 -> 實現
- 物件 -> 對象
- 檔案 -> 文件
- 記憶體 -> 內存
- 記憶體漏水 -> 內存洩漏
- 迴圈 -> 循環
- 指標 -> 指針

回饋

這份教材

- 位置是 <https://zonble.gitbooks.io/kkbox-ios-dev/content/>
- 同時也放在 <https://kkbox.github.io/kkbox-ios-dev/>
- 資料都放在 GitHub 上 <https://github.com/KKBOX/kkbox-ios-dev>

您如果發現任何的錯誤，可以直接在 GitHub 上發 Pull Request 修改，如果您有什麼問題，也可以寫信到 zonble @ gmail . com 。

2015 年前言

這份教材是為了 KKBOX iOS/Mac OS X 開發部門的新入訓練所設計，目的是培養出可以開發、維護 KKBOX 的 iOS 與 Mac OS X 版本，以及我們其他軟體產品的工程師。

寫給誰的？

這不太算是一本入門的教材。編寫這份教材的時候，我們假設的讀者是已經寫過半年左右的 iOS 程式，甚至有一兩個 App 在架上，因為畢竟是為了新人訓練所設計，要我們雇用完全沒有經驗的 iOS 工程師，我想也很困難（其實也有這樣的案例，但真的為數不多）。所以我們假設在使用這份教材之前：

1. 你已經會操作 Mac 電腦，也知道怎麼安裝 Xcode
2. 你已經知道一些 Objective-C 語法
3. 你已經知道一些 Foundation 物件怎麼使用
4. 你已經知道怎樣使用一些 UIKit 物件
5. 你會使用 Quartz 2D 畫圖
6. 你知道怎麼處理在 iOS 裝置上實機執行的 certificate 與 provision profile

你可以期待在這份教材看到什麼？

這份教材的主要方向是把一些 iOS 工程師天天都在使用，但往往模模糊糊懵懵懂懂的觀念說清楚，首先從 Objective-C 這門語言是怎麼運作講起，再進一步講解在 iOS 與 Mac OS X 裝置上的 GUI App 如何運作，中間也會帶過一些重要的 Design Pattern，在閱讀的過程中再搭配實際練習，透過 coding 實際體會這些重要概念，最後具備有 coding，以及能夠清楚解決各種問題的能力。

我們所重視的不是如何快速上手，不是如何用 CocoaPods 或 Carthage 的套件拼出一個 App，而是偏重由下而上（Bottom-up）的學習：先了解整個開發框架的底層，以及整個框架的基本觀念，然後才去一個個去看在這個框架中有哪些 API、以及有哪些第三方 library 可以使用。

這和我們的工作型態有關，在一些以專案為主的公司裡頭，可能注重的是如何快速完成專案、如何以最快的速度完成新 App，交付 App 之後就不再維護。但 KKBOX 算是一個有一些年紀的產品，我們在 2008 年九月推出第一版 Mac OS X 版本，在 2009 年一月推出第一版 iOS 版本，一路從 Mac OS X 10.4、iPhoneOS 2 的時代寫到現在，我們會花上許多時間解決、並且避免軟體中出現的問題。而要對系統到底怎麼運作要有一定的認識，才能夠知道怎樣閱讀 crash report，知道怎麼辨識問題並修正。

iOS 的技術不斷變動，我們也多從不太會變動的基礎概念著手。不過，我們也會按照 KKBOX 自己的需要以及技術的改變，隨時擴充或改變這份教材的內容。

同時，不同於坊間大多數的 iOS 開發書籍，因為 KKBOX 同時有 iOS 與 Mac OS X 的版本，在這份教材中，我們會同時講到 iOS 與 Mac OS X，但如果同一個重要觀念同時出現在 iOS 與 Mac OS X 中，會以 iOS 為主。

在程式語言上，我們先講解 Objective-C，因為我們的主要產品大部分的 code 還是用 Objective-C 寫成的，只有使用少量的 Swift。而就我們的經驗，學過 Objective-C 之後，（只要搞定了 Optional 這個語言特性）轉換到 Swift 上也不會花太多的時間。而且，就算先學了 Swift 語法，還是得搞清楚 selector、protocol、delegate 等，而這些概念 Swift 與 Objective-C 還是互通的。

在各種 API 與 library 的介紹上，也以 KKBOX 產品中會用到的為主，像我們在這份教材中，會打算講 Audio 相關的部份，絕大多數的 iOS 開發者可能並不需要知道如何在 iOS 上處理 Audio，但 KKBOX 是一家做音樂服務的公司。至於像遊戲開發等，雖然我們之前也用過像 Cocos2D 這些功能做過一些小專案，但不會在這邊佔上篇幅，畢竟這份教材完全是為了 KKBOX 的需要而打造。

其實在這份教材中絕大多數材料，都可以在蘋果文件與 WWDC 影片中找到，其實蘋果官方的文件比絕大多數的書籍都還要好，但這些資料非常多，當你想直接從蘋果文件學習開發的時候，這份教材也提供一份怎樣在文件之海中探索的方向。

雖然在敘述上，我們只能夠按照章節順序排列，但是在整個 Cocoa Framework 中，許多觀念其實互為因果或是互相糾纏，所以在某個章節中，可能會事先講到跟後面章節有關的事情，但是先不要介意，如果遇到這樣的狀況，我們會在後面繼續說清楚。

Selector

我個人在學習新事物的時候，通常會這麼挑戰自己：我有沒有辦法用一句話描述這件事物是什麼？還有，這件事物可以用在哪裡？我用這種挑戰，確認自己是否完全理解我想要學習的事物。

我們在接下來的章節中，也會以這樣的方式展開。其實這種作法，也就是蘋果的敘事風格，比方說，當我們去看 WWDC 2015 影片，蘋果在講什麼是 Localization 的時候¹，Localization 的簡短定義就是「讓你的 App 說你的顧客的語言」（Making your app speak your customer's language），從這麼簡短的話中，我們就可以得到可以繼續發揮的關鍵字：1. 「你的顧客的語言」，全世界各國的語言，到底有什麼幽微的不同？2. 「Making」，我們該怎麼做？有什麼技術的細節？—於是，我們繼續一步一步發展出完整的故事。

在台灣，當我們問一位 iOS 工程師「什麼是 delegate」這樣的問題時，得到的答案可能是「delegate 就是『代理』」，至於代理了什麼、為什麼要代理，卻又說不上來。我們在 KKBOX 對自己的要求並不只如此。

以我們現在要討論的 Selector 來說，可以做這樣一個簡短的定義：

Selector 就是用字串表示某個物件的某個 method

用更術語的說法會是：

Selector 就是 Objective-C 的 virtual table 中指向實際執行 function pointer 的一個 C 字串

那，Selector 有什麼用途呢？

因為 method 可以用字串表示，因此，某個 method 就可以變成可以用來傳遞的參數。

至於要更進一步了解 Selector，我們就要從一些更基本的事情開始講起：Objective-C 裡頭的物件以及 Class，到底是什麼？

¹. WWDC 2015 What's New in Internationalization <https://developer.apple.com/videos/wwdc/2015/?id=227>
，2:30 左右 ↫

Objective-C Class/Object 到底是什麼？

你應該在其他的文件裡頭聽說過，Objective-C 是 C 語言的 Superset，在 C 語言的基礎上，加上了一層稀薄的物件導向，而 Cocoa Framework 的 Cocoa 這個名字就是這麼來的—Cocoa 就是 C 加上 OO。也因此，在 Objective-C 程式中，可以直接呼叫 C 的 API，而如果你將 .m 改名叫做 .mm，程式裡頭還可以混和 C++ 語法，變成 Objective-C++。

Objective-C 的程式在 compile time 時，Compiler 實際會編譯成 C 然後繼續編譯。所有的 Objective-C Class 會變成 C 的 Structure，所有的 method（以及 block）會被編譯成 C function，接下來，在執行的時候，Objective-C runtime 才會建立某個 C Structure 與 C function 的關聯，也就是說，一個物件到底有哪些 method 可以呼叫，是在 runtime 才決定的。

Objective-C 物件會被編譯成 Structure

比方說，我們現在寫了一個簡單的 Class，裡頭只有 int a 這個成員變數：

```
@interface MyClass : NSObject {  
    int a;  
}  
@end
```

會被編譯成

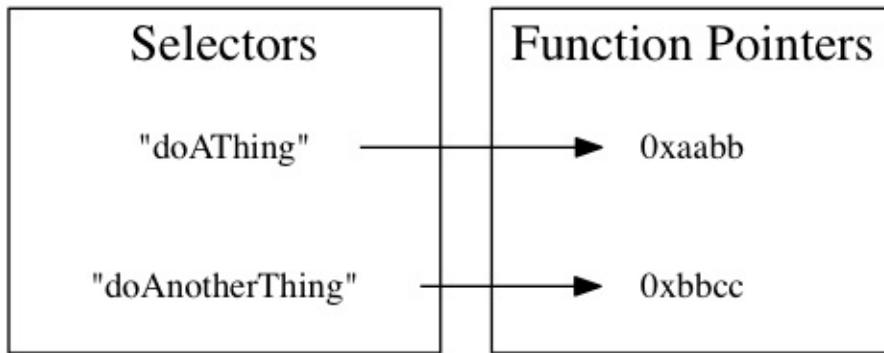
```
typedef struct {  
    int a;  
} MyClass;
```

因為 Objective-C 的物件其實就是 C 的 structure，所以當我們建立了一個 Objective-C 物件之後，我們也可以把這個物件當做呼叫 C structure 呼叫¹：

```
MyClass *obj = [[MyClass alloc] init];  
obj->a = 10;
```

對 Class 加入 method

在執行的時候，runtime 會為每個 class 準備好一張表格（專用術語叫做 virtual table），表格裡頭會以一個字串當 key，每個 key 會對應到 C function 的指標位置。Run time 裡頭，把實作的 C function 定義成 IMP 這個 type；至於拿來當作 key 的字串，就叫做 selector，type 定義成 SEL，然後我們可以使用 @selector 關鍵字建立 selector。



而其實 SEL 就是 C 字串，我們可以來寫點程式檢查一下：

```
 NSLog(@"%@", (char *)@selector(doSomething));
```

我們會順利印出「doSomething」這個 C 字串。

A screenshot of the Xcode IDE showing a terminal window. The code in the editor is:

```
1 #import <Foundation/Foundation.h>
2
3 int main (int argc, const char * argv[])
4 {
5     @autoreleasepool {
6         NSLog(@"%@", @selector(test));
7         NSLog(@"%@", strlen(@selector(test)));
8
9         char *s = calloc(100, sizeof(char));
10        strcat(s, @selector(test));
11        strcat(s, @selector(test));
12        NSLog(@"%@", s);
13        free(s);
14    }
15    return 0;
16 }
```

The terminal output shows the following lines:

```
2015-10-12 00:03:55.059 Test[34979:2834047] test
2015-10-12 00:03:55.061 Test[34979:2834047] 4
2015-10-12 00:03:55.061 Test[34979:2834047] testtest
Program ended with exit code: 0
```

每次我們對一個物件呼叫某個 method，runtime 在做的事情，就是把 method 的名稱當做字串，尋找與字串符合的 C function 實作，然後執行。也就是說，下面這三件事情是一樣的：

我們可以直接要求某個物件執行某個 method：

```
[myObject doSomthing];
```

或是透過 `performSelector:` 呼叫。`performSelector:` 是 `NSObject` 的 method，而在 Cocoa Framework 中所有的物件都繼承自 `NSObject`，所以每個物件都可以呼叫這個 method。

```
[myObject performSelector:@selector(doSomething)];
```

我們可以把 `performSelector:` 想成台灣的電視新聞用語：如果原本的句子是「我正在吃飯」，使用 `performSelector:` 就很像是「我正在進行一個吃飯的動作」。而其實，最後底層執行的是 `objc_msgSend`。

```
objc_msgSend(myObject, @selector(doSomething), NULL);
```

我們常常會說「要求某個 object 執行某個 method」、「要求某個 object 執行某個 selector」，其實是一樣的事情，我們另外也常聽到一種說法，叫做「對 receiver 傳遞 message」，這則是沿用來自 Small Talk 的術語—Objective-C 受到了 Small Talk 語言的深刻影響—但其實也是同一件事。

因為一個 Class 有哪些 method，是在 run time 一個一個加入的；所以我們就有機會在程式已經在執行的時候，繼續對某個 Class 加入新 method，一個 Class 已經存在了某個 method，也可以在 run time 用別的實作換掉，一般來說，我們會用 Category 做這件事情，不過 Category 會是下一章的主題，會在下一章繼續討論。

我們在這裡首先要記住一件非常重要的事：在 Objective-C 中，一個 class 會有哪些 method，並不是固定的，如果我們在程式中對某個物件呼叫了目前還不存在的 method，編譯的時候，compiler 並不會當做編譯錯誤，只會發出警告而已，而跳出警告的條件，也就只有是否有引入的 header 中到底有沒有這個 method 而已，所以我們一不小心，就很有可能呼叫到了沒有實作的 method（或這麼說，我們要求執行的 selector 並沒有對應的實作）。如果我們是使用 `performSelector:` 呼叫，更是完全不會有警告。直到實際執行的時候，才發生 unrecognized selector sent to instance 錯誤而導致應用程式 crash。

之所以只有警告，而不當做編譯錯誤，就是因為某些 method 有可能之後才會被加入。蘋果認為你會寫出呼叫到沒有實作的 selector，必定是因為你接下來在某個時候、某個地方，就會加入這個 method 的實作。

由於 Objective-C 語言中，物件有哪些 method 可以在 run time 變更，所以我們也會將 Objective-C 列入像是 Perl、Python、Ruby 等所謂的動態語言（Dynamic Language）之林。而在寫這樣的動態物件導向語言時，一個物件到底有哪些 method 可以呼叫，往往會比這個物件到底是屬於哪個 class 更為重要。²

如果我們不想要用 category，而想要自己動手寫點程式，手動將某些 method 加入到某個 class 中，我們可以這麼寫。首先宣告一個 C function，至少要有兩個參數，第一個參數是執行 method 的物件，第二個參數是 selector，像這樣：

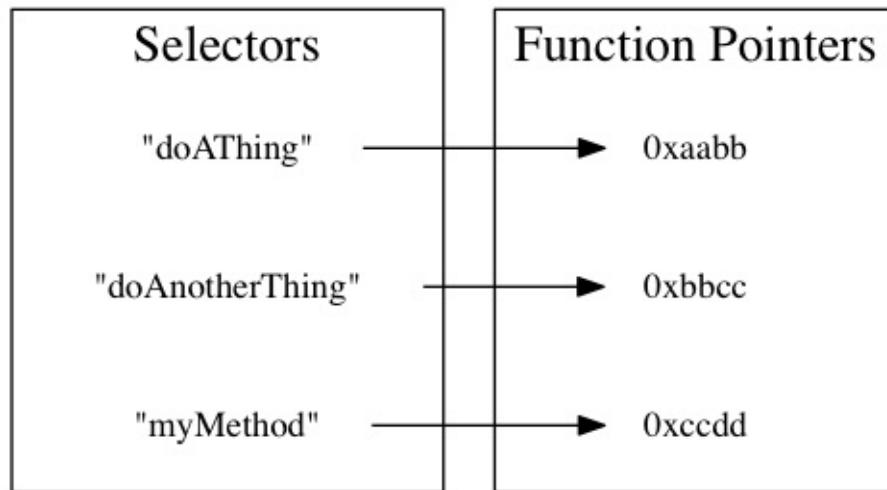
```
void myMethodIMP(id self, SEL _cmd) {
    doSomething();
}
```

接下來可以呼叫 `class_addMethod` 加入 selector 與實作的對應。

```
#import <objc/runtime.h>
// 中間省略
class_addMethod([MyClass class], @selector(myMethod), (IMP)myMethodIMP, "v@:");
```

接下來就可以這麼呼叫了：

```
MyClass *myObject = [[MyClass alloc] init];
[myObject myMethod];
```



1. 不過，如果你直接在程式裡頭這麼呼叫，Xcode 會在編譯的時候發出警告，告訴你在不久的將來會禁止這樣直接呼叫物件的成員變數，如果想要取用成員變數，必須另外寫 getter/setter。而如果這個成員變數被宣告成是 private 的，Xcode 會直接出現編譯錯誤，禁止你這樣呼叫。 ↵
2. 這種強調物件有哪些 method，會比物件繼承自哪個 Class 來得重要的觀念，有一個專有名詞，叫做 Duck Typing，中文翻譯做「鴨子型別」。觀念是：我眼前這個東西到底是不是鴨子？它是不是鳥類或是哪個種類根本就不重要，反正它走路游泳像鴨子，叫起來像鴨子，那我就當它是鴨子。可以參見 Wikipedia 上的說明：http://en.wikipedia.org/wiki/Duck_typing。 ↵

Selector 有什麼用途？

我們會在幾個地方應用 selector：

Target/Action pattern

Selector 的主要用途，就是實作 target/action。相信你應該試過在 Xcode 中建立一個新專案之後，可能在 Interface Builder 中建立了一個 UIButton 或是 NSButton 物件，然後將按鈕連接到 controller 中宣告成 IBAction 的 method 上，這時候，我們的 Controller 就是 Button 的 target，而要求 controller 執行的 method，就叫做 action。

我們在 Interface Builder 裡頭做的事情，也可以透過程式碼做到。而如果我們想要設計一套系統 Framework 裡頭所沒有的客製 UI 元件，第一步就是要了解怎麼實作 target/action。

在 UIKit 中的 Target/Action 稍微複雜一些，因為同一個按鈕可以一次連接好幾個 target 與 action，我們在這邊使用 AppKit 示範—在 Mac 上，一次只會指定單一的 target 與 action。如果想要產生一個按鈕或是其他的 custom control，我們會繼承自 NSView，然後建立兩個成員變數：target 與 action，action 是一個 selector。

```
@interface MyButton : NSView
{
    id target;
    SEL action;
}

@property (assign) IBOutlet id target;
@property (assign) SEL action;
@end

@implementation MyButton
- (void)mouseDown:(NSEvent *)e
{
    [super mouseDown:e];
    [target performSelector:action withObject:self];
}
@synthesize target, action;
@end
```

我們在這邊將 target 的型別設定為 id，代表的是任意 Objective-C 物件的指標，如同前面提到，Controller 到底是什麼 class，在這邊並不重要，而且我們也不該將 target 的 class 寫死，因為如此一來，就變成只有某些 Controller 才能使用這個按鈕。

我們接著在 `mouseDown:` 中，要求 target 執行之前傳入的 action，由於 selector 是字串，是可以傳遞的參數，所以也就可以成為按鈕的成員變數。

我們接下來也可以使用程式碼連結 target 與 action，在 Controller 的程式中，只要這麼寫即可：

```
[(MyButton *)button setTarget:self];
[(MyButton *)button setAction:@selector(clickAction:)];
```

把要做什麼事情當做參數傳遞，每個語言都有不同的作法。Objective-C用的是拿字串來尋找對應的實作 function 指標，在 C 語言裡頭就會直接傳遞指標，一些更高階的語言或著是把一段程式碼當做是字串傳遞，要使用的時候再去 evaluate 這段程式碼字串，或是一段程式碼本身就是一個物件，所以可以把程式碼當做物件傳遞，我們稱之為「匿名函式」（Anonymous Function），現在 Objective-C 也有匿名函式，叫做 block，不過，對這個 1983 年誕生的語言來說，這是很晚近才有的功能，我們會稍晚討論。

檢查 method 是否存在

前面提到，我們有可能會呼叫到並不存在的 method，如果這麼做就會產生錯誤。但我們有時候會遇到的狀況是：我們並不確定某些 method 到底有沒有實作，如果有，就呼叫，如果沒有，就略過或是使用其他的 method。

這種狀況最常遇到的就是顧及向下相容。比方說，在 iOS 4 之後，才開始支援 Retina Display，我們在繪圖的 code 中要決定現在應該繪製怎樣精細程度的圖片，需要知道目前用的是傳統的一倍品質，還是 Retina Display 的兩倍品質，就要去問 `UIScreen` 的 `scale` 屬性。但是，當我們開始支援 iOS 4 的時候，可能還要顧及 iOS 3 的使用者，導致我們不能夠貿然直接呼叫 `scale`（當然，如果你的應用程式都只支援最新版本的作業系統，那是再幸福不過的事），而是要去檢查這個屬性是否存在，如果沒有，就代表使用者的作業系統是 iOS 4 之前的版本，我們只需要提供一倍品質的圖片就可以了。在未來，只要遇到向下相容，我們就還是得處理這樣的狀況。

另外，雖然蘋果只允許 iOS 上面的應用程式只能夠是單一的執行檔，不能夠在執行時載入其他的 binary，但是在 Mac OS X 上面卻可以載入 loadable bundle，或是在應用程式中放置 private framework，一個物件的某些 method 可以不在主程式中，而是在 plug-in 中實作。我們也要做這樣的檢查。

檢查某個物件是否實作了某個 method，只要呼叫 `respondsToSelector:` 就可以了：

```
BOOL scale = 1.0;
if ([[UIScreen mainScreen] respondsToSelector:@selector(scale)]) {
    scale = [UIScreen mainScreen].scale;
}
```

在其他程式語言中，也需要這樣檢查 method 是否存在嗎？在 Ruby 語言中，有類似的 `respond_to?` 語法，至於 Python，我們或著可以用 `dir` 這個 funciton 檢查某個物件的全部 attribute 中是否存在對應到某個 method 的 key，但是更常見的作法就是使用 `try...catch` 語法，如果遇到某個 method 可能不存在，就包在 `try...catch` 的 block 中，像是：

```
try:
    myObject.doSomething()
except Exception, e:
    print "The method does not exist."
```

在 Objective-C 中，同樣也有 `try...catch` 語法，在許多語言中，善用 `try...catch`，也可以將程式寫得清楚有條理，但是我們並不鼓勵在 Objective-C 語言中使用。原因與 Objective-C 的記憶體管理機制有關，如果大量使用 `try...catch`，會導致記憶體漏水（Memory Leak）。

Objective-C 本身並不算有記憶體回收機制（Garbage Collection，以下簡稱 GC）的語言，雖然在 Mac OS X 10.5 的時代，蘋果嘗試在 Objective-C 上實作 GC，但是成果實在不甚理想，如果貿然在 Mac OS X 上大量使用 GC，實際運作會有嚴重的記憶體漏水問題；蘋果在推出 iOS 之後，也不敢將這套機制用在行動裝置上，

而是在 iOS 5 時放棄在 runtime 管理記憶體，而是推出 ARC (Automatic Reference Counter)，在 compile time 時決定什麼時候應該釋放記憶體。

由於傳統的 Objective-C 記憶體管理大量使用一套叫做 auto-release 的機制—雖然說是 auto，其實也沒多自動，頂多算是半自動—將一些應該要釋放的物件延遲釋放，在這一輪 runloop 中先不釋放，而是到了下一輪 runloop 開始時才釋放這些記憶體。如果使用 try...catch 捕捉例外錯誤，就會跳出原本的 runloop，而導致應該釋放的記憶體沒被釋放。

我們接下來還會在 [記憶體管理 Part 1 與 Responder](#) 討論這個部分。

Timer

NSObject 除了 `performSelector:` 這個 method 之外，同樣以 `performSelector` 開頭的，還有好幾組 API 可以呼叫，例如 `-performSelector:withObject:afterDelay:`，就可以讓我們在一定的秒數之後，才要求某個 method 執行。

```
[self performSelector:@selector(doSomething) withObject:nil afterDelay:1.0];
```

如果時間還不到已經預定要執行的時間，method 還沒有執行，我們也可以反悔，取消剛才預定要執行的 method，只要呼叫 `cancelPreviousPerformRequestsWithTarget:` 即可。如以下範例：

```
[NSObject cancelPreviousPerformRequestsWithTarget:self];
```

`performSelector:withObject:afterDelay:` 的效果相當於產生 `NSTimer` 物件，當我們想要延遲呼叫某個 method，或是要某件事情重複執行，都可以透過建立 `NSTimer` 物件達成，要使用 timer，我們也必須使用 selector 語法。

我們先定義一個 timer 要做的事情：

```
- (void)doSomething:(NSTimer *)timer
{
    // Do something
}
```

然後透過 `doSomething:` 的 selector 建立 timer

```
NSTimer *timer = [NSTimer scheduledTimerWithTimeInterval:1.0
                                                    target:someObject
                                                      selector:@selector(doSomething:)
                                                    userInfo:nil
                                                   repeats:YES];
```

除了透過指定 target 與 selector 之外，還可以透過指定 `NSInvocation` 呼叫建立 `NSTimer` 物件；`NSInvocation` 實際就是將 target/action 以及這個 action 中要傳遞給 target 的參數這三者，再包裝成一個物件。呼叫的 method 是 `scheduledTimerWithTimeInterval:invocation:repeats:`。

透過建立 `NSInvocation` 物件建立 timer 的方式如下。

```
NSMethodSignature *sig = [MyClass instanceMethodSignatureForSelector:  
                        @selector(doSomething:)];  
NSInvocation *invocation = [NSInvocation invocationWithMethodSignature: sig];  
[invocation setTarget:someObject];  
[invocation setSelector:@selector(doSomething:)];  
[invocation setArgument:&anArgument atIndex:2];  
NSTimer *timer = [NSTimer scheduledTimerWithTimeInterval:1.0  
                                         invocation:invocation  
                                         repeats:YES];
```

請注意，在呼叫 `NSInvocation` 的 `setArgumentAtIndex` 的時候，我們要傳遞的參數，要從 2 開始，因為在這邊我們要想成，這是給 `objc_msgSend` 呼叫用的參數，在 0 的參數是物件的 `self`，位在 1 的則是 `selector`。

接收 NSNotification

我們稍晚才會討論 `NSNotification` 以及 `NSNotificationCenter`，不過在這邊先簡單提到：如果我們要接收 `NSNotification`，我們也要在開始訂閱通知的時候，指定要由哪個 `selector` 處理這個通知。詳見 [Notification Center](#) 這一章。

在某個 Thread 執行 method

除了已經提到的 `-performSelector withObject:afterDelay:` 之外，`NSObject` 還有好幾個 `method`，是讓指定的 `selector` 丟到某個 `Thread` 執行，包括：

- `-performSelectorOnMainThread withObject:waitUntilDone:modes:`
- `-performSelectorOnMainThread withObject:waitUntilDone:`
- `-performSelector:onThread withObject:waitUntilDone:modes:`
- `-performSelector:onThread withObject:waitUntilDone:`
- `-performSelectorInBackground withObject:`

假如有一件事情—在這邊叫做 `doSomething`—會執行太久，我們可以將這件事情丟到背景，也就是另外建立一條 `Thread` 執行：

```
[self performSelectorInBackground:@selector(doSomething) withObject:nil];
```

注意，在背景執行時，這個 `method` 的內部需要建立自己的 Auto-Release Pool。

執行完畢後，我們可以透過 `-performSelectorOnMainThread withObject:waitUntilDone:`，通知主 `Thread` 我們已經把事情做完了。像是，如果我們要轉換一個比較大的檔案，就可以在背景實際轉檔，轉完之後，再告訴主 `Thread`，在 UI 上跳出提示視窗，提示使用者已經轉檔完畢。

```
- (void)doSomthing  
{  
    @autoreleasepool {  
        // Do something here.  
        [self performSelectorOnMainThread:@selector(doAnotherThing)  
                                withObject:nil  
                                waitUntilDone:NO];
```

```
    }  
}
```

Array 排序

我們今天想要對 `NSArray` 做排序，就得要告訴這個 Array 怎樣比較裡頭每個東西的大小，所以我們需要把怎麼比較大小這件事情傳遞到 array 上。Cocoa Framework 提供三種方式排序 Array，我們可以把怎麼比大小寫成 C Function，然後傳遞 C Function 的指標，現在也可以傳遞 Block，而如果 Array 裡頭的物件有負責比較大小的 method 的話，我們也可以透過 selector 指定要用哪個 method 排序。

`NSString`、`NSDate`、`NSNumber` 以及 `NSIndexPath`，都提供 `compare:` 這個 method，假如有一個 array 裡頭都是字串的話，我們就可以使用 `compare:` 排序，`NSString` 用來比較大小順序的 method 與選項（像是是否忽略大小寫，字串中如果出現數字，是否要以數字的大小排列而不是只照字元順序...等等），其中最常用的，該是 `localizedCompare:`，這個 method 會參考目前使用者所在的系統語系決定排序方式，像是簡體中文語系下用拼音排序，繁體中文語系下用筆劃排序...等等。

我們使用 `sortedArrayUsingSelector:` 產生重新排序的新 Array，如果是 `NSMutableArray`，則可以呼叫 `sortUsingSelector:`

```
NSArray *sortedArray = [anArray sortedArrayUsingSelector:  
                        @selector(localizedCompare:)];
```

我們也可以透過傳遞 selector，要求 Array 裡頭每一個物件都執行一次指定的 method。

```
[anArray makeObjectsPerformSelector:@selector(doSomething)];
```

代替 if...else 與 switch...case

因為 selector 其實就是 C 字串，除了可以當做參數傳遞之外，也可以放在 array 或是 dictionary 裡頭。有的時候，如果你覺得寫一堆 if...else 或是 switch...case 太過冗贅，例如，原本我們可能這麼寫：

```
switch(condition) {  
    case 0:  
        [object doSomething];  
        break;  
    case 1:  
        [object doAnotherThing];  
        break;  
    default:  
        break;  
}
```

如果沒有什麼會超過邊界的問題的話，其實可以考慮搭配 Xcode 4.4 之後所提供的 literal 新寫法¹，看起來就精簡一些。

```
[object performSelector:NSSelectorFromString(@[@"doSomething",  
                                              @"doAnotherThing"])[condition]);
```

我們可以使用 `NSStringFromSelector`，將 selector 轉換成 `NSString`，反之，也可以使用 `NSSelectorFromString` 將 `NSString` 轉成 selector。

...呼叫 Private API

Objective-C 裡頭其實沒有真正所謂的 private method，一個物件實作了那些 method，即使沒有 import 對應的 header，我們都呼叫得到。系統裡頭許多原本就內建的 class，有一些 header 並沒有宣告的 method，但是從一些相關網站或是其他管道，我們就是知道有這些 method，先不管究竟是什麼原因，我們有的時候就是想要呼叫看看，這時候我們往往會用 `performSelector:` 呼叫。原因也很簡單：因為我們沒有 header。

但我們並不建議做這樣的事情：今天一個 method 沒有被放在 header 裡頭，就代表在作業系統改版的時候，系統可能把整個底層的實作換掉，這個 method 可能就此消失，而造成系統升級之後，因為呼叫不存在的 method 而造成應用程式 crash。而如果你打算寫一套 iOS 應用程式，在 App Store 上架販售，蘋果的審查過程中就會拒絕使用 private API 的軟體。

¹. 參見

http://developer.apple.com/library/mac/#documentation/DeveloperTools/Conceptual/WhatsNewXcode/Articles/xcode_4_4.html ↵

呼叫 `performSelector:` 需要注意的地方

我們在呼叫 `performSelector:` 的時候要注意幾點：

對 `super` 呼叫 `performSelector:`

前面雖然提到，對一個物件直接呼叫某個 method，或是透過 `performSelector:` 呼叫，意義是一樣的，但如果是對 `super` 呼叫，卻有不一樣的結果。如果是：

```
[super doSomething];
```

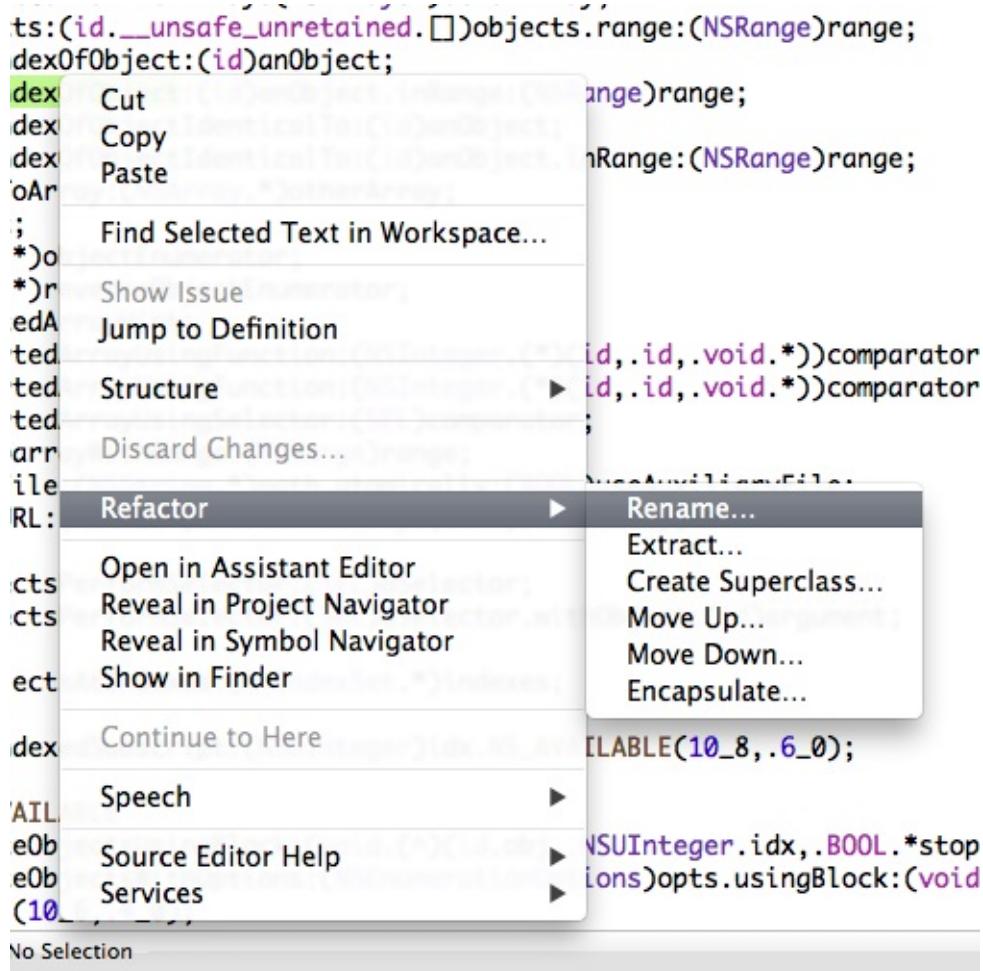
代表的是呼叫 `super` 的 `doSomething` 實作。但如果是：

```
[super performSelector:@selector(doSomething)];
```

呼叫的是 `super` 的 `performSelector:`，最後結果仍然等同於 `[self doSomething]`。

Refactor 工具

隨著專案的發展，我們可能後來覺得當初某個 method 的命名並不恰當，所以想要換個名字，這時候與其使用搜尋/替代功能，不如直接使用 Xcode 提供的 Refactoring 工具：在想要改名字的 method 上面點選滑鼠右鍵，就會出現選單，然後從「Refactor」中選擇「Rename」。

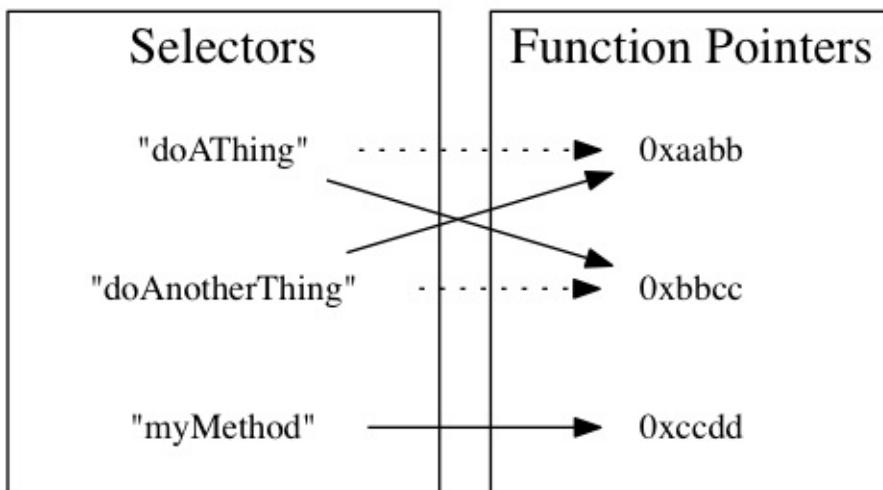


執行之後，Xcode 除了把這個 method 的名字換掉，也會同時更新所有專案中呼叫這個 method 的程式，但，如果我們當初是用 `performSelector:` 呼叫要執行的 method，Xcode 並不會把裡頭的 selector 也換掉，只會出現簡短的警吶訊息而已，如果我們忽略了這些警吶，之後執行的時候，就會出現找不到 selector 的錯誤。我們需要格外小心。

Selector 是 Objective-C 中所有魔法的開始

Objective-C 物件有哪些 method，就是這個物件的 Class 的 virtual table 中，有多少 selection/C function pointer 的 pair，這個特性造就了在 Objective-C 中可以做很多神奇的事情，同時也造成了 Objective-C 這門語言的限制。

Objective-C 的神奇之處，就在於，既然物件有哪些 method 可以在 run time 決定，因此每個物件也都可以在 run time 的時候改變。我們可以在不用繼承物件的情況下，就增加新的 method，通常最常見作法的就是我們接下來要討論的 category，我們也可以隨時把既有的 selector 指到不同的 C function pointer 上，像是把兩個 selector 所指向的 function pointer 交換，這種交換 selector 實作的作法叫做 method swizzling—我們暫時還不會討論到這邊。



由於一個 selector 只會指向一個實作，因此，Objective-C 不會有 C++、Java、C# 等語言當中的 **overloading**。所謂 overloading，就是可以容許有多個名稱相同，但是參數型別不同的 function 或 method 存在，在呼叫的時候，如果傳入了指定型別的參數，就會呼叫到屬於該參數型別的那一組 function 或 method。在 Objective-C 當中，同一個名稱的 method，就只會只有一套實作，如果有幾個名稱相同的 method，就會以最後在 runtime 載入的那一組，代替之前的實作。

Objective-C 的 virtual table 像是一個 dictionary，而 C++、Java 等語言的 virtual table 則是一個 array。在 Objective-C 中，我們呼叫 `[someObject doSomthing]` 時，我們是在表格中尋找符合 "doSomething" 這個字串的 method，在 C++ 或 Java 中，我們呼叫 `someObject.doSomething()` 時，在做的事情大概是要求「執行 virtual table 中的第八個 method」這樣的事情。

由於每做一次 method 的呼叫，都是在做一次 selector/function pointer 的查表，與其他較靜態的語言相較，這個查表的動作相當耗時，因此執行效能也比不上 C++ 等程式語言。Objective-C 其實是一門相當古老的語言，在 1984 年時便問世，但就因為效能問題，在整個 80 到 90 年代根本乏人問津，直到 2008 年 iPhone SDK 推出之後才逐漸成為主流語言。

但 Objective-C 這樣做又有另外一項優點：我們的 App 不需要連結特定版本的 runtime 與 libraries，就算 libraries 中 export 出來的 function 換了位置，只要 selector 不變，還是可以找到應該要執行的 C function，所以舊的 App 在新版本的作業系統上執行時，新版本作業系統並不需要保留舊版的 Libraries，而避免了 C++ 等語言中所謂 **DLL Hell** 問題。

到了 2014 年，蘋果在 iOS 與 Mac OS X 平台上又推出了新的程式語言 Swift，標榜 Swift 是比 Objective-C 更新、執行效能更好的程式語言，Swift 之所以效能會比 Objective-C 好，其中一點，就是因為 Swift 又改變了 virtual table 的實作，走回像是 C++、Java 等語言的 virtual table 設計。因為 Swift 也有必須連結指定版本 runtime 的問題，所以在每個 Swift App 的 App bundle 中，其實都包了一份 Swift runtime。

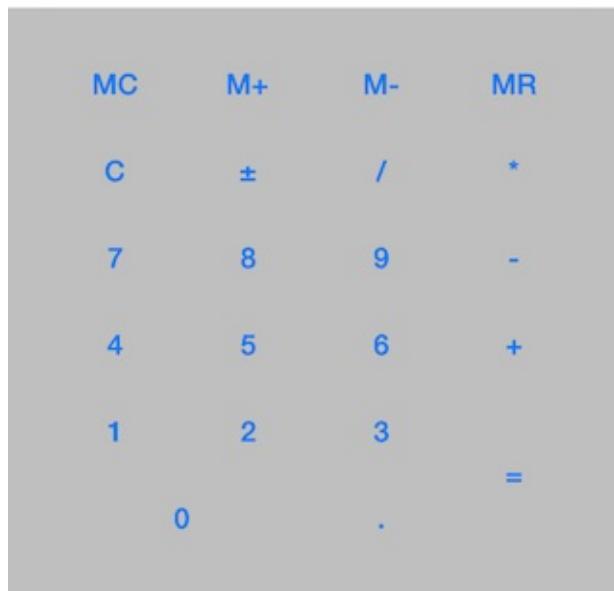
相關閱讀

- 蘋果官方文件 Programming with Objective-C
- 蘋果官方文件 Objective-C Runtime Programming Guide
- 蘋果官方文件 Cocoa Core Competencies - Selector
- COCOA SAMURAI: Understanding the Objective-C Runtime

練習：小計算機

120

+



練習範圍

- Interface Builder
- Target Action
- Selector

練習目標

我們要在 iOS 裝置上寫一個小計算機，這個計算機需要有以下功能：

- 有顯示目前輸入數字與運算結果的 text label
- 有可以輸入數字與小數點的按鈕
- 輸入數字後，按下加減乘除按鈕後，在輸入數字，然後按下等於或其他加減乘除按鈕，就會進行對應的四則運算，並輸出運算結果
- 初始數字為 0 時，如果按一次 0，還是 0
- 初始數字為 0 時，如果按一次 1，會變成 1
- 在輸入過程中，小數點只能夠出現一次，之後再按小數點沒作用
- 如果已經按了一個加減乘除按鈕，再按一次另一個加減乘除按鈕，後面輸入的運算子會代替原本的，而不會立刻執行運算
- 如果遇到任何數字除 0 的狀況，要跳出 alert

練習內容

- 首先使用 Interface Builder 拉出所有需要的 UI 元件
- 建立並連結必要的 IBOutlet 與 IBAction
- 輸入數字完畢後，請使用 NSDecimalNumber 這個 class 儲存數字
- 輸入加減乘除運算子時，請使用 NSDecimalNumber 的運算 method 的 selector 儲存，包括：
 - decimalNumberByAdding:
 - decimalNumberBySubtracting:
 - decimalNumberByMultiplyingBy:
 - decimalNumberByDividingBy:
- 執行運算時，左右兩邊的數字都應該是 NSDecimalNumber，然後使用之前所儲存的 selector，用 `performSelector:withObject:` 執行這個 selector，像是 `[leftOperand performSelector:savedSelector withObject:rightOperand]`

Category

Category：不用繼承物件，就直接增加新的 method，或替換原本的 method。

前一章提到，Objective-C 語言中，每個 class 有哪些 method，都是在 runtime 時加入的，我們可以透過 runtime 提供的一個叫做 `class_addMethod` 的 function，加入對應某個 selector 的實作。而在 runtime 時想要加入新的 method，使用 category 會是更容易理解與寫作的方法，因為可以使用與宣告 class 時差不多的語法，同時也以一般實作 method 的方式，實作我們要加入的 method。

至於在 Swift 語言中，Swift 的 Extension 特性，也與 Objective-C 的 Category 差不多。

什麼時候應該要使用 Category

如果想要擴充某個 class 功能，增加新的成員變數與 method，我們又沒有這個 class 的程式碼，正規作法就是繼承、建立新的 subclass。那，我們需要在不用繼承，就直接增加 method 這種作法的重要理由，就是我們想要擴充的 class 很難繼承。

我能想到的，大概有幾種狀況：

1. Foundation 物件
2. 用 Factory Method Pattern 實作的物件
3. Singleton 物件
4. 在專案中出現次數已經多不勝數 的物件。

Foundation 物件

Foundation 裡頭的基本物件，像是 NSString、NSArray、NSDictionary 等 Class 的底層實作，除了可以透過 Objective-C 的介面呼叫之外，也可以透過另外一個 C 的介面，叫做 Core Foundation，像 NSString 其實會對應到 Core Foundation 裡頭的 CFStringRef，NSArray 對應到 CFArrayRef，而你甚至可以直接把 Foundation 物件 cast 成 Core Foundation 的型別，當你遇到一個需要傳入 CFStringRef 的 function 的時候，只要建立 NSString 然後 cast 成 CFStringRef 傳入就可以了。

所以，當你使用 alloc、init 產生一個 Foundation 物件的時候，其實會得到一個同時有 Foundation 與 Core Foundation 實作的 subclass，而實際產生出來的物件，往往與你的認知會有很大的差距，例如，我們以為 NSMutableString 繼承自 NSString，但建立 NSString，呼叫 alloc、init 的時候，我們真正拿到的是 __NSCFConstantString，而建立 NSMutableString，拿到 __NSCFString，而 __NSCFConstantString 實際繼承自 __NSCFString！

我們來寫點程式檢查 Foundation 物件其實屬於哪些 Class：

```
#define CLS(x) NSStringFromClass([x class])
 NSLog(@"%@", CLS([NSString string]));
 NSLog(@"%@", CLS([NSMutableString string]));
 NSLog(@"%@", CLS([NSNumber numberWithInt:1]));
#undef CLS
```

執行結果：

```
NSString:__NSCFConstantString
NSMutableString:__NSCFString
NSNumber:__NSCFNumber
```

因此，當我們嘗試建立 Foundation 物件的 subclass 之後，像是繼承 NSString，建立我們自己的 MyString，假設我們並沒有 override 原本關於建立 instance 的 method，我們也不能保證，建立出來的就是 MyString 的 instance。

用 Factory Method Pattern 實作的物件

Wikipedia 上對 Factory Method Pattern 的解釋是：

...the factory method pattern is a creational pattern which uses factory methods to deal with the problem of creating objects without specifying the exact class of object that will be created.

翻譯成中文：Factory Method Pattern 是一套用來解決不用特別指定是哪個 class，就可以建立物件的方法。比方說，某個 class底下，其實有一堆 subclass，但對外部來說並不需要確實知道這些 subclass 而是只要 **對最上層的 class，輸入指定的條件，就會從挑選一個符合指定條件的 subclass、建立 instance 回傳**。

在 UIKit 中，UIButton 就是個好例子。在某些版本的 iOS 當中，在我們在建立 UIButton 物件的時候，並不是呼叫 `init` 或是 `initWithFrame:`，而是呼叫 UIButton 的 class method：`buttonWithType:`，透過傳遞按鈕的 type 建立按鈕物件。在大多數狀況下，會回傳 UIButton 物件，但假如我們傳入的 type 是 `UIButtonTypeRoundedRect`，卻會回傳繼承自 UIButton 的 `UIRoundedRectButton`。

檢查一下：

```
#define CLS(x) NSStringFromClass([x class])
NSLog(@"%@", @"UIButtonTypeCustom %@", 
        CLS([UIButton buttonWithType:UIButtonTypeCustom]));
NSLog(@"%@", @"UIButtonTypeRoundedRect %@", 
        CLS([UIButton buttonWithType:UIButtonTypeRoundedRect]));
#undef CLS
```

輸出結果：

```
UIButtonTypeCustom UIButton
UIButtonTypeRoundedRect UIRoundedRectButton
```

我們想要擴充 `UIButton`，但拿到的卻是 `UIRoundedRectButton`，而 `UIRoundedRectButton` 却無法繼承，因為這個物件不在公開的 header 中，我們也不能夠保證以後傳入 `UIButtonTypeRoundedRect` 就一定會拿到 `UIRoundedRectButton`。如此一來，就造成我們難以繼承 `UIButton`。

或這麼說：假使今天我們的需求是想要改動某個上層的 class，讓底下所有的 subclass 也都增加了一個新的 method，我們又無法改動這個上層 class 的程式，就會採用 category。比方說，我們今天希望所有的 `UIViewController` 都有一個新 method，如此我們整個應用程式中每個 `UIViewController` 的 subclass 都可以呼叫這個 method，但，我們就是無法改動 `UIViewController`。

Singleton 物件

Singleton 物件是指：**某個 class 只有、也只該有一個 instance，每次都只對這個 instance 操作，而不是建立新的 instance**。像 `UIApplication`、`NSUserDefaults`、`NSNotificationCenter` 以及 Mac OS X 上的 `NSWorkspace` 等，都採用 singleton 設計。

之所以說 singleton 物件很難繼承，我們先來看怎麼實作 singleton：我們會有一個 static 的物件，然後每次都回傳這個物件。宣告部分如下：

```
@interface MyClass : NSObject
+ (MyClass *)sharedInstance;
@end
```

實作部分：

```
static MyClass *sharedInstance = nil;

@implementation MyClass
+ (MyClass *)sharedInstance
{
    return sharedInstance ?
        sharedInstance :
        (sharedInstance = [[MyClass alloc] init]);
}
@end
```

其實現在 Singleton 大多會使用 GCD 的 `dispatch_once` 實作，但是在我們還沒有提到 GCD 之前，我們先使用這樣的寫法。我們會在討論 Threading 的時候繼續討論 GCD，至於用 GCD 實作 Singleton 的細節，請參見 [再談 Singleton](#) 這一章。

我們如果 subclass 了 MyClass，卻沒有 override 掉 `sharedInstance`，那麼，`sharedInstance` 回傳的還是 MyClass 的 singleton instance。而想要 override 掉 `sharedInstance` 又不見得這麼簡單，因為這個 method 裡頭很可能又做了許多其他事情，很可能會把一些 initialize 時該做的事情，反而放在這邊做（這不是很好的作法，但就是可能發生）。例如 MyClass 可能這麼寫：

```
+ (MyClass *)sharedInstance
{
    if (!sharedInstance) {
        sharedInstance = [[MyClass alloc] init];
        [sharedInstance doSomething];
        [sharedInstance doAnotherThing];
    }
    return sharedInstance;
}
```

如果我們並沒有 MyClass 的程式碼，這個 class 是在其他的 library 或是 framework 中，我們直接 override 了 `sharedInstance`，就很有可能有事情沒做，而產生不符合預期的結果。

在專案中出現次數已經多不勝數

隨著專案不斷成長，某些 class 已經頻繁使用到了到處都是，而我們現在需求改變，必須要增加新的 method，我們卻也沒有力氣可以把所有用到的地方統統換成新的 subclass。Category 就是解決這種狀況的救星。

實作 Category

Category 的語法很簡單，一樣是用 `@interface` 關鍵字宣告 header，在 `@implementation` 與 `@end` 關鍵字當中的範圍是實作，然後在原本的 class 名稱後面，用中括弧表示新增的 category 名稱。

舉例來說，我們今天雖然寫的是 Objective-C 語言，但是想要變得更像 Small Talk 一點，所以我們不想用 `NSLog` 印出某個物件的資料，而是每個物件都有個把自己印出來的 method，所以我們對 `NSObject` 建立了一個叫做 `SmallTalkish` 的 category。

```
@interface NSObject (SmallTalkish)
- (void)printNL;
@end

@implementation NSObject (SmallTalkish)
- (void)printNL
{
    NSLog(@"%@", self);
}
@end
```

如此一來，每個物件都增加了 `printNL` 這個 method。可以這麼呼叫：

```
[myObject printNL];
```

前一章提到，我們在排序一個裡頭都是字串的 Array 的時候，可以呼叫 `localizedCompare:`，但，假如我們希望所有的字串都一定要用中文筆劃順序排序，我們可以寫一個自己的 method，例如 `strokeCompare:。`

```
@interface NSString (CustomCompare)
- (NSComparisonResult)strokeCompare:(NSString *)anotherString;
@end

@implementation NSString (CustomCompare)
- (NSComparisonResult)strokeCompare:(NSString *)anotherString
{
    NSLocale *strokeSortingLocale = [[[NSLocale alloc]
        initWithLocaleIdentifier:@"zh@collation=stroke"]
        autorelease];
    return [self compare:anotherString
        options:0
        range:NSMakeRange(0, [self length])
        locale:strokeSortingLocale];
}
@end
```

在存檔的時候，檔名的慣例是原本的 class 名稱加上 category 的名稱，中間用加號連接，以我們剛剛建立的 `CustomCompare` 為例，存檔時就要存成 `NSString+CustomCompare.h` 及 `NSString+CustomCompare.m`。

Category 還可以有什麼用途？

除了幫原有的 class 增加新 method，我們也會在幾種狀況下使用 category。

將一個很大的 Class 切成幾個部分

由於我們可以在建立 class 之後，繼續透過 category 增加method，所以，假如一個 class 很大，裡頭有數十個method，實作上千行，我們就可以考慮將這個 class 的 method 拆分成若干個category，讓整個 class 的實作分開在不同的檔案中，方便知道某一群的 method 屬於什麼用途，也方便日後維護。

切開一個很大的 class 可以收到的好處包括：

跨專案

如果你手上同時有好幾個專案，我們在進行專案的時候，由於之前寫的程式碼可以重複使用，造成每個專案可能共用同一個class，但是每個專案又不見得都會用到這個 class 裡頭全部的實作，我們就可以考慮將只屬於某個專案的實作，拆分到一個category 中。

跨平台

做為寫 Objective-C 語言的工程師，我們非常有可能會遇到跨平台開發的需求，如果我們某段程式碼只有用到 Mac OS X 與 iOS 都有的 library 與 framework 的話，我們的程式就可以同時在 Mac OS X 與 iOS 使用。當我們打算在 Mac OS X 與 iOS 共用同一個 class，我們就可以考慮將跨平台的部份與平台相依的部份拆開，將只屬於某個平台的部份拆成另外一個category，以蘋果自己的例子來說，在 Mac OS X 與 iOS 上都有 `NSString`，但由於兩個平台在繪圖方面的實作有所不同，所以在繪製字串的部份，就被拆分到 `NSStringDrawing` 與 `UIStringDrawing` 這些 category 中。

替換原本的實作

由於一個 class 有哪些 method，是在 runtime 時加入的，所以除了可以加入新的 method 之外，假如我們嘗試再加入一個 selector 與已經存在的 method 名稱相同的實作，我們可以把已經存在的 method 的實作，換成我們要加入的實作。這麼做在 Objective-C 語言中是完全合法的，如果 category 裡頭出現了名稱相同的 method，compiler 還是容許編譯成功，只會跳出簡單的警告訊息。

在實務上，這麼做卻非常危險，假如我們自己寫了一個 class，我們又另外寫了一個 category 置換其中的 method，當我們日後想要修改這個 method 的內容，很容易忽略在 category 中的同名 method，結果就是不管我們怎麼改動原本 method 裡頭的程式，結果卻是什麼改變都沒有。

我自己曾經犯過一個低級錯誤：在開發時我建立了另外一個 git 分支，在新分支中，我覺得某個 class 太大，於是將部分 method 拆到了另外一個 category 中，但是開發主線卻又在修改這個 class，結果造成合併分支的時候，就變成原本的 class 與 category 中出現了相同的 method，花了半天的時間才找到問題出在哪裡。

除了某一個 category 中可以出現與原本 class 中名稱相同的 method，我們甚至可以在好幾個 category 中，都出現名稱相同的 method，哪一個 category 在執行時被最後載入，就會變成是這個 category 中的實作。那麼，如果有多个 category，我們怎麼知道哪一個 category 會被最後載入呢？Objective-C runtime 並不保證 category 的載入順序，所以我們必須嚴格避免寫出這種程式。

Category 還可以有什麼用途？

Extensions

Objective-C 語言中有一項叫做 extensions 的設計，也可以用來拆分一個很大的 class，語法與 category 非常相似，但是不太一樣。在語法上，extensions 像是一個沒有名字的 category，在 class 名稱之後直接加上空的括弧，而 extensions 定義的 method，需要放在原本的 class 實作中。

以下是一個使用 extensions 的例子：

```
@interface MyClass : NSObject
@end

@interface MyClass()
- (void)doSomthing;
@end

@implementation MyClass
- (void)doSomthing
{
}
@end
```

在 `@interface MyClass ()` 這段宣告中，我們並沒有在括弧中定義任何名稱，接著，`doSomthing` 又是直接在 `MyClass` 中實作。extensions 可以有幾個用途：

拆分 Header

如果我們就是打算實作一個很大的 class，但是覺得 header 裡頭已經列出了太多的 method，我們可以將一部分 method 搬到 extensions 的定義裡頭。

另外，extension 除了可以放 method 之外，也可以放成員變數，而一個 class 可以擁有不只一個 extension，所以如果一個 class 真的有非常非常多的 method 與成員變數，我們可以把這些 method 與成員變數，放在多個 extension 中。

管理 Private Methods

這其實是更常見的用途。我們在寫一個 class 的時候，內部有一些 method 不需要、我們也不想要放在 public header 中，但是如果將這些 method 放在 header 裡頭，又會出現一個困擾：在 Xcode 4.3 之前，如果這些 private method 在程式碼中不放在其他 method 前面，其他的 method 在呼叫這些 method 的時候，compiler 會不斷跳出警告，而這種無關緊要的警告一多，我們往往會忽視真正重要的警告。¹

想要避免這些警告，要不就是把 private method 都放在最前面，但這樣並不能完全解決問題，因為 private method 之間也會相互呼叫，花時間確認每個 method 之間的呼叫順序並不是很經濟的事；要不就是都用 `performSelector:` 呼叫，這樣問題更大，就像前面提到，在 method 改名、呼叫 refactoring 工具的時候，這樣非常危險。

蘋果提供的建議是，我們在 .m 或 .mm 檔案開頭的地方宣告一個 extensions，將 private method 都放在這個地方，如此一來，其他 method 就可以找到 private method 的宣告。從 Xcode 4 開始—至少到 Xcode 7 都是如此，在 Xcode 中所提供的 file template 中，如果你選擇建立一個 UIViewController 的 subclass，就可以看

到在 .m 檔案的最前面，幫你預留了一塊 extensions 的宣告。

在這邊也順便提一下 Swift 裡頭的 extension。在 Swift 語言中，我們可以直接使用 `extension` 關鍵字，建立 class 的 extension，擴充一個 class；Swift 的 extension 與 Objective-C 的 category 的主要差別是，Objective-C 的 category 要給定一個名字，而 Objective-C 的 extension 是沒有名字的 category，至於 Swift 的 extension 則統一都沒有名字。

所以，如果你有一個 Swift class 叫做 MyClass：

```
class MyClass {  
}
```

只要這樣就可以直接建立 extension：

```
extension MyClass {  
}
```

另外，Swift 除了可以用 extension 擴充 class 之外，甚至可以擴充 protocol 與 struct。我們會在後面講 delegate 的時候說明 protocol。

```
protocol MyProtocol {  
}  
  
extension MyProtocol {  
}  
  
struct MyStruct {  
}  
  
extension MyStruct {  
}
```

¹. 請見 [Objective-C Feature Availability Index](#) ↵

Category 是否可以增加新的成員變數或屬性？

因為 Objective-C 物件會被編譯成 C 的 structure，我們雖然可以在 category 中增加新的 method，但是我們卻不能夠增加新的成員變數。

在 Mac OS X 10.6 與 iOS 4 之後，蘋果提出一套叫做 Associated Objects 的辦法，讓我們可以在 category 中增加新的getter/setter，觀念差不多是：既然我們可以用一張表格記錄一個 class 有哪些method，那，我們不就也可以另外建一張表格，記錄有哪些物件與這個 class 相關？

要使用 Associated Objects，我們需要匯入 `objc/runtime.h`，然後呼叫 `objc_setAssociatedObject` 建立 setter，用 `getAssociatedObject` 建立 getter，呼叫時要傳入：我們要讓哪個物件與哪個物件之間建立關連，關連時使用的是那一個key（型別為 C 字串）。在以下的範例中，我們在 `MyCategory` 這個 category 裡，增加一個叫做 `myVar` 的 property。

```
#import <objc/runtime.h>

@interface MyClass(MyCategory)
@property (retain, nonatomic) NSString *myVar;
@end

@implementation MyClass
- (void)setMyVar:(NSString *)inMyVar
{
    objc_setAssociatedObject(self, "myVar",
                           inMyVar, OBJC_ASSOCIATION_RETAIN_NONATOMIC);
}
- (NSString *)myVar
{
    return objc_getAssociatedObject(self, "myVar");
}
@end
```

在 `setMyVar:` 中呼叫 `objc_setAssociatedObject` 時，所最後一個參數

`OBJC_ASSOCIATION_RETAIN_NONATOMIC`，是用來決定要用哪一種記憶體管理策略，管理我們傳入的參數，在我們的例子中，我們傳入的是 `NSString`，是一個 Objective-C 物件，所以我們必須要 retain 起來。這邊可以傳入的參數還可以是 `OBJC_ASSOCIATION_ASSIGN`、`OBJC_ASSOCIATION_COPY_NONATOMIC`、`OBJC_ASSOCIATION_RETAIN` 以及 `OBJC_ASSOCIATION_COPY`，與 property 語法使用的記憶體管理方式一致。而當我們的 `MyClass` 物件在 `dealloc` 的時候，所有透過 `objc_setAssociatedObject` 而 retain 起來的物件，也都會被一併釋放。

雖然我們不能在 category 中增加成員變數，但是卻可以在 extensions 中宣告。在 Xcode 4.2 之後，我們可以這麼寫：

```
@interface MyClass()
{
    NSString *myVar;
}
@end
```

我們甚至可以將成員變數直接放在 `@implementation` 的程式區塊中：

```
@implementation MyClass
{
    NSString *myVar;
}
@end
```

對 NSURLSessionTask 撰寫 Category

在寫 category 的時候，有時候你也會遇到像 NSURLSessionTask 這種地雷。

假如我們在 iOS 7 上，對 NSURLSessionTask 寫了一個 category 之後，你會發現，如果我們從 [NSURLSession sharedSession] 產生了 data task 物件，之後，對這個物件呼叫 category 裡頭的 method，很奇怪，會收到找不到 selector 的錯誤。照理說一個 data task 是 NSURLSessionDataTask，繼承自 NSURLSessionTask，為什麼我們寫的 NSURLSessionTask category 沒用？

到了 iOS 8 的環境下又正常，可以對這個物件呼叫 NSURLSessionTask category 裡頭的 method，但如果是寫成 NSURLSessionDataTask 的 category，結果又遇到找不到 selector 的錯誤。

我們來寫一個簡單的 category：

```
@interface NSURLSessionTask (Test)
- (void)test;
@end

@implementation NSURLSessionTask (Test)
- (void)test
{
    NSLog(@"test");
}
@end
```

然後執行看看

```
NSURLSessionDataTask *task = [[NSURLSession sharedSession]
    dataTaskWithURL:[NSURL URLWithString:@"http://kkbox.com"]];
[task test];
```

結果就發生錯誤了：

The screenshot shows the Xcode interface with a project named 'NSURLSessionTest'. In the center, a code editor displays a file named 'NSURLSessionTestTests.m' containing Objective-C code. The code defines a category on NSURLSessionTask and a XCTestCase subclass. The code is as follows:

```

1 #import <UIKit/UIKit.h>
2 #import <XCTest/XCTest.h>
3
4 @interface NSURLSessionTask (Test)
5 - (void)test;
6 @end
7
8 @implementation NSURLSessionTask (Test)
9 - (void)test
10 {
11     NSLog(@"%@", @"test");
12 }
13 @end
14
15 @interface NSURLSessionTestTests : XCTestCase
16 @end
17
18 @implementation NSURLSessionTestTests
19
20 - (void)testNSURLSessionTask
21 {
22     NSURLSessionDataTask *task = [[NSURLSession sharedSession] dataTaskWithURL:
23         [NSURL URLWithString:@"http://kkbox.com"]];
24     [task test];
25 }
26
27 @end
28

```

這件事情很奇怪，你於是會開始想各種可能的方向。一對了，如果有一個 category 不是直接寫在你的 app 裡頭，而是寫在某個 static library，你要在編譯 app 的最後才把這個 library link 進來，預設 category 並不會 linker 給 link 進來，你必須要另外在 Xcode 專案設定的 other linker flag，加上 -ObjC 或是 -all_load。會是這個問題嗎？你試了一下，發現沒用，還是一樣收到 unsupported selector 錯誤。

然後你想到，NSURLSessionTask 是個 Foundation 物件，而 Foundation 物件往往真正的實作與最上層的介面並不是同一個。所以，我們來看看 NSURLSessionTask 的繼承樹到底長成什麼樣子。

程式如下：

```

NSURLSessionDataTask *task = [[NSURLSession sharedSession] dataTaskWithURL:[NSURL URLWithString:@"http://kkbox.com"]];
NSLog(@"%@", [task class]);
NSLog(@"%@", [task superclass]);
NSLog(@"%@", [[[task superclass] superclass] superclass]);
NSLog(@"%@", [[[task superclass] superclass] superclass]);

```

在 iOS 8 上的結果是：

```

__NSCFLocalDataTask
__NSCFLocalSessionTask
NSURLSessionTask
NSObject

```

在 iOS 7 上的結果是：

```
__NSCFLocalDataTask  
__NSCFLocalSessionTask  
__NSCFURLSessionTask  
NSObject
```

結論，無論是 iOS 8 或 iOS 7，我們建立的 data task，都不是直接產生 NSURLSessionDataTask 物件，而是產生 __NSCFLocalDataTask 這樣的 private class 的物件。iOS 8 上，__NSCFLocalDataTask 並不繼承 NSURLSessionDataTask，而 iOS 7 上 __NSCFLocalDataTask 甚至連 NSURLSessionTask 都不是。

但如果我們去問，我們建立的這些 data task 到底是不是 NSURLSessionDataTask，像呼叫 [task isKindOfClass:[NSURLSessionDataTask class]]，還是會回傳 YES。其實 -isKindOfClass: 是可以 override 掉的，所以，即使 __NSCFLocalDataTask 根本就不是 NSURLSessionDataTask，但我們可以把 __NSCFLocalDataTask 的 -isKindOfClass: 寫成這樣：

```
- (BOOL)isKindOfClass:(Class)aClass
{
    if (aClass == NSClassFromString(@"NSURLSessionDataTask")) {
        return YES;
    }
    if (aClass == NSClassFromString(@"NSURLSessionTask")) {
        return YES;
    }
    return [super isKindOfClass:aClass];
}
```

也就是說，-isKindOfClass: 其實並不像你所想像的那麼值得信任。這也就是寫 Objective-C 這門語言的迷人之處：寫 Objective-C 就像真實的人生一樣，一個物件就像人一般，一個人能做什麼絕對不是在出生的時候就決定的，而是在他的一生當中可以隨時改變，而當你用 -isKindOfClass: 去問一個物件到底是什麼物件的時候，大概就跟你去問一個 PM 到底會不會規劃產品，或是一個號稱是 iOS 工程師的人到底會不會寫程式一樣。

相關閱讀

- 蘋果官方文件 Programming with Objective-C - Customizing Existing Classes
- 蘋果官方文件 Cocoa Core Competencies - Category
- NSHipster: Associated Objects
- King's Cocoa: Objective-C Associated Objects
- 蘋果官方文件 Objective-C Runtime Reference

練習：字串反轉

練習範圍

- Category

練習目標

說起來這個練習很簡單。

我們要對 `NSString` 寫一個 category，裡頭有一個叫做 `reversedString` 的 method，這個 method 可以輸出反轉後的字串。

- 如果原本的字串是 @"KKBOX"，要可以輸出 @"XOBKK"
- 如果原本的字串是 @"KKBOX iOS 開發教材"，要可以輸出 @"材教發開 SOi XOBKK"

記憶體管理 Part 1

記憶體管理是一件極其簡單又極其麻煩的事情。

說它簡單，因為所謂的記憶體管理不過就是兩件事情：一塊記憶體我們是要用，還是不要用；該用的時候就用，不用的時候就釋放。麻煩的地方就在於我們很容易疏忽，而造成用與不用不成對。這就很像開車，開車也不過就是前進與煞車，但是天天路上都會發生車禍。

不成對，就會造成兩種結果：如果用完之後不釋放記憶體，就會造成軟體佔用了一堆沒有用到的記憶體，記憶體用量愈來愈大，造成記憶體用盡，在iOS 上系統會強制終止我們的應用程式，這種狀況叫做記憶體漏水（Memory Leak）。

反之，如果一塊記憶體已經被釋放掉了，我們卻還以為這塊記憶體還存在我們可以呼叫的物件，所以當我們嘗試呼叫的時候，才發現這塊記憶體該存在的物件已經不在了，這種狀況叫做over-release 或是 invalid memory reference，會造成應用程式 crash，crash log 上面會告訴你錯誤類型是 EXC_BAD_ACCESS 。

處理 over-release 可能是初學者開始學習 Mac OS X 與 iOS 平台的第一個障礙（我覺得通常第二個障礙是搞不懂 delegate 是什麼）：一開始寫程式，程式卻一直莫名其妙的 crash，於是前往網路論壇求救，但是通常也不會有多少人幫忙，因為網路上的其他同行通常都樂於回答問題，但懶得幫別人修 bug。

過去三十年學界與業界也一直努力解決記憶體管理的問題，畢竟在寫軟體的時候，不把力氣放在處理程式邏輯問題，而是這種瑣碎的困擾，對於工程師的生產力是一大傷害，主要提出的解決方案是記憶體自動回收（Garbage Collection，GC），在軟體執行時，如果發現已經沒有任何一個變數指向某塊記憶體，就代表這塊記憶體再也用不到，於是開始回收這塊記憶體。90年代之後誕生的程式語言，幾乎都有 GC 機制。

蘋果曾經推出 Mac OS X 10.5 時，在 Mac OS X 上實作了 GC，同時也帶動一些使用 GC 的動態語言開始與 Cocoa Framework 橋接，誕生了 PyObjC、MacRuby、RubyCocoa、JSTalk 專案，我們因此也可以使用 Python、Ruby、JavaScript 等語言，直接撰寫 Mac OS X 應用程式。但是，蘋果所實作的 GC，問題不小。

但我過去參與一項專案，在 Mac OS X 10.6 上面以 GC 開發一個軟體，結果可說是淒慘無比：整個 Cocoa Framework 是架構在很多 C library 上的，而底層許多 library 其實並不支援 GC，我們發現只要在 main thread 之外用到像是 NSDateFormatter 、 NSNumberFormatter 這些 class（這些 class 架構在 libICU 上，是 IBM 的一項 Open Source 專案，用來處理多國語系的格式問題，同時也是一套 Regular Expression 引擎），GC 就完全沒有作用，記憶體狂漏不止。蘋果在推出 iOS 之後，也始終不在 iOS 上實作 GC。

隨著蘋果在 Compiler 的投資逐漸展露成果，逐步將 Compiler 從 GCC 換成 LLVM，最後決定改變技術方向，從另外一種方向來解決將記憶體管理自動化的問題，就是在 iOS 5 上推出的 ARC（Automatic Reference Counting），不在 runtime 回收記憶體，而是在編譯程式的時候，自動幫你加上與記憶體釋放有關的程式碼。而蘋果的下一步，就是直接在 ARC 的基礎上開發新的程式語言 Swift。

雖然 ARC 推出的目的就是希望你以後再也不要為記憶體管理問題煩惱，但問題是，實際上，你還是得知道記憶體管理是怎麼運作的。—記得有人打比方說，GC 是自排車，而 Objective-C 的記憶體管理就像開手排車，要自己知道怎麼進檔對檔，我覺得，加上 ARC 呢，其實還是手排車，只是你訓練了一隻猴子幫你在副駕駛座幫你打檔，而你接下來，還得要學會怎麼調教這一隻猴子。

如果你在開發 iOS 或 Mac App 的時候，真的完全不想知道 Objective-C 的記憶體管理細節，可能就直接寫 Swift 吧！

Reference Count/Retain/Release

Objective-C 語言裡頭每個物件，都是指向某塊記憶體的指標，在 C 語言當中，你會使用像是 `malloc`、`calloc` 這些 function 配置記憶體，用完之後，就呼叫 `free` 釋放記憶體。但是我們如何知道一塊記憶體被多少地方用到，之後這些地方又不再用到呢？所以在 Objective-C 語言發展之初，就建立了一套計算有多少地方用到某個物件的簡單機制，叫做 *reference count*，意義非常簡單：只要一個物件被某個地方用到一次，這個地方就對這個物件加一，反之就減一，如果數字減到變成了零，就該釋放這塊記憶體。

一個物件如果使用 `alloc`、`init` ... 產生，初始的 reference count 為 1。接著，我們可以使用 `retain` 增加 reference count。

```
[anObject retain]; // 加 1
```

反之就用 `release`：

```
[anObject release]; // 減 1
```

我們可以使用 `retainCount` 檢查某個物件被 retain 了多少次。

```
NSLog(@"%@", @"Retain count: %d", [anObject retainCount]);
```

有了基本概念之後，我們就可以看出以下程式有什麼問題

```
id a = [[NSObject alloc] init];
[a release];
[a release];
```

因為在第二行，`a` 所指向的記憶體已經被釋放了，所以第三行想要再釋放一次，就會造成錯誤。同樣的：

```
id a = [[NSObject alloc] init];
id b = [[NSObject alloc] init];
b = a;
[a release];
[b release];
```

在第三行中，由於 `b` 指向了 `a` 原本所指向的記憶體，但是 `b` 原本所指向的記憶體卻沒有釋放，同時再也沒有任何變數指向 `b` 原本指向的記憶體，因此這塊記憶體就發生了記憶體漏水。接著，在第四行呼叫 `[a release]` 時，這塊記憶體就已經被放掉了，但是由於 `a` 與 `b` 都已經指向了同一塊記憶體，所以第五行的 `[b release]` 也是操作同一塊記憶體，於是會發生 `EXC_BAD_ACCESS` 錯誤。

Auto-Release

如果我們今天有一個 method，會回傳一個 Objective-C 物件，假使寫成這樣：

```
- (NSNumber *)one
{
    return [[NSNumber alloc] initWithInt:1];
}
```

那麼，每次用到了由 one 這個 method 產生出來的物件，用完之後，都需要記住要 release 這個物件，很容易造成疏忽。慣例上，我們會讓這個 method 回傳 auto-release 的物件。像是寫成這樣：

```
- (NSNumber *)one
{
    return [[[NSNumber alloc] initWithInt:1] autorelease];
}
```

所謂的 auto-release 其實也沒有多麼自動，而是說，在這一輪 run loop 中我們先不釋放這個物件，讓這個物件可以在這一輪 run loop 中都可以使用，但是先打上一個標籤，到了下一輪 run loop 開始時，讓 runtime 判斷有哪些前一輪 run loop 中被標成是 auto-release 的物件，這個時候才減少 retain count 決定是否要釋放物件。

我們在這邊遇到了一個陌生的名詞：run loop，我們會在 [Responder](#) 這一章當中說明。

在建立 Foundation 物件的時候，除了可以呼叫 `alloc`、`init` 以及 `new` 之外（`new` 這個 method 其實就相當於呼叫了 `alloc` 與 `init`；比方說，我們呼叫 `[NSObject new]`，就等同於呼叫了 `[[NSObject alloc] init]`），還可以呼叫另外一組與物件名稱相同的 method。

以 `NSString` 為例，有一個叫做 `initWithString` 的 instance method，就有一個對應的 class method 叫做 `stringWithFormat`，使用這一組 method，就會產生 auto-release 的物件。也就是說，呼叫了 `[NSString stringWithFormat:...]`，相當於呼叫了 `[[[NSString alloc] initWithFormat:...] autorelease]`。使用這一組 method，可以讓程式碼較為精簡。

基本原則

先整理一下我們已經學到的事情：

- 如果是 `init` 、 `new` 、 `copy` 這些 method 產生出來的物件，用完就該呼叫 `release` 。
- 如果是其他一般 method 產生出來的物件，就會回傳 auto-release 物件、或是 `Singleton` 物件（稍晚會解釋什麼是 `Singleton`） ，就不需要另外呼叫 `release` 。

而呼叫 `retain` 與 `release` 的時機包括：

- 如果是在一般程式碼中用了某個物件，用完就要 `release` 或是 `auto-release` 。
- 如果是要將某個 Objective-C 物件，變成是另外一個物件的成員變數，就要將物件 `retain` 起來。但是 `delegate` 物件不該 `retain`，我們稍晚會討論什麼是 `delegate` 。
- 在一個物件被釋放的時候，要同時釋放自己的成員變數，也就是要在實作 `dealloc` 的時候，釋放自己的成員變數。

要將某個物件設為另外一個物件的成員變數，需要寫一組 `getter/setter`。我們接下來要討論怎麼寫 `getter/setter` 。

Getter/Setter 與 Property 語法

Getter 就是用來取得某個物件的某個成員變數的 method，setter 則是用來設定成員變數。如果某個成員變數是 C 的型別，像是 int，我們可以這麼寫。假使我們有個 Class 叫做 MyClass，成員變數是 number：

```
@interface MyClass: NSObject
{
    int number;
}
- (int)number;
- (void)setNumber:(int)inNumber;
@end
```

我們建立了 setter 叫做 `setNumber:`，而 getter 叫做 `number`。請注意，在其他語言的慣例中，getter 可能會取名叫做 `getNumber`，但是 Objective-C 語言的慣例則是只取 `number` 這樣的名稱。實作則是：

```
- (int)number
{
    return number;
}
- (void)setNumber:(int)inNumber
{
    number = inNumber;
}
```

如果是 Objective-C 物件，我們則要將原本成員變數已經指向的記憶體位置釋放，然後將傳入的物件 retain 起來。可能像這樣：

```
- (id)myVar
{
    return myVar;
}
- (void)setMyVar:(id)inMyVar
{
    [myVar release];
    myVar = [inMyVar retain];
}
```

假如今天我們在開發的應用程式裡頭用到了很多個 thread，而在不同的 thread 中，同時會用到 myVar，這麼寫其實並不安全：在某個 thread 中呼叫了 `[myVar release]` 之後，到 myVar 指定到 inMyVar 的位置之間，假使另外一個 thread 剛好用到了 myVar，這時候 myVar 剛好指到了一塊已經被釋放的記憶體，於是就造成了 `EXC_BAD_ACCESS` 錯誤。

要避免這種狀況，一種方法是加上一些 lock，讓程式在呼叫 `setMyVar:` 的時候，不讓其他 thread 可以使用 myVar；另外一種簡單的方法是，只要一直不要讓 myVar 指定到可能被釋放的記憶體位置。我們可以這麼改寫：

```
- (void)setMyVar:(id)inMyVar
```

```
{
    id tmp = myVar;
    myVar = [inMyVar retain];
    [tmp release];
}
```

我們先將 myVar 原本指向的記憶體位置，暫存在一個變數中，接著直接將 myVar 指到傳入的記憶體位置，接著再釋放 tmp 變數中所記住的、原本的記憶體位置。由於每次都要這麼寫，寫久了會覺得麻煩，通常會寫成一個macro，或是直接使用 Objective-C 2.0 裡頭的 property 語法。

相信在開始學習開發 Mac OS X 與 iOS 程式的時候，大部分書籍一開始就示範如何使用 property 語法。也就是，像我們上面的例子，用 property 語法可以寫成：

```
@interface MyClass: NSObject
{
    id myVar;
    int number;
}
@property (retain, nonatomic) id myVar;
@property (assign, nonatomic) int number;
@end

@implementation MyClass
- (void)dealloc
{
    [myVar release];
    [super dealloc];
}
@synthesize myVar;
@synthesize number;
@end
```

我們在這邊使用了 `@synthesize` 語法，在編譯我們的程式的時候，其實就會被編譯成我們在上面所寫的 getter/setter，而我們想要設定 myVar 的內容時，除了可以呼叫 `setMyVar:` 之外，也可以呼叫 dot 語法，像是 `myObject.myVar`
`= someObject .`

我們需要注意，在釋放記憶體的時候，`myVar = nil` 與 `self.myVar = nil` 這兩段程式是不一樣的，前者只是單純的將 myVar 的指標指向 nil，但是並沒有釋放原本所指向的記憶體位置，所以會造成記憶體漏水，但後者卻等同於呼叫 `[self setMyVar:nil]`，會先釋放 myVar 原本指向的位置，然後將 myVar 設成 nil。

在這邊先補充一下，在 Xcode 4.4 之後，如果用了 property 語法，我們甚至不用宣告對應的成員變數，compiler 在編譯程式的時候，會自動補上 myVar 與 number 需要對應的成員變數。

偶而我們可以在網路上面聽到一些聲音，認為初學者應該避免使用 property，主要原因除了上述 `myVar = nil` 與 `self.myVar = nil` 這兩者容易搞混之外，property 的 dot 語法，又與 C 的 structure 語法相同，在還不熟悉的狀況下，很容易讓初學搞錯哪些是 property，哪些又是屬於一個 structure。

例如，我們想要知道一個 view 的 x 座標是在哪裡，會寫出像 `self.view.frame.origin.x` 這種程式，就需要知道，`view` 是 `self` 的 property，`frame` 也是 `view` 的 property，但是 `x` 却是 `origin` 這個 `CGPoint` 裡頭的變數，而 `origin` 也是 `frame` 這個 `CGRect` 裡頭的變數，但是初學的時候很容易搞混。

我們想要取得 `x`，可以寫成 `self.view.frame.origin.x`，但想要設定 `x`的位置，如果這麼寫：

```
self.view.frame.orgin.x = 0.0;
```

程式會發生編譯錯誤。`self.view.frame.origin.x` 其實會被編譯成 `[[self view] frame].origin.x`，這沒問題，但是如果要改變 `view` 的 `frame`，我們還是要透過 `setFrame:`，所以即使只是要改變 `x`座標的位置，我們還是得要這麼寫：

```
CGRect originalFrame = self.view.frame;
originalFrame.origin.x = 0.0;
self.view.frame = originalFrame;
```

其實這兩點要搞清楚並不困難，如果因為這些緣故而不使用 property 語法，實在有些因噎廢食，因為使用 property 語法，可以大幅精簡程式碼。而由於前述 Xcode 4.4 可以在宣告 property 之後，由 compiler 自動補上對應的成員變數的特性，也不致於會搞錯 `myVar = nil` 與 `self.myVar = nil` 的差別—因為我們並沒有宣告 `myVar`，如果寫成前者，編譯時會產生錯誤。

相關閱讀

- [蘋果官方文件 Advanced Memory Management Programming Guide](#)
- [WWDC 2013 Fixing Memory Issues](#)

記憶體管理 Part 2

這一章當主要討論 ARC。前一章提到，由於 ARC 是透過靜態分析，在 Compile Time 決定應該要在程式碼的那些地方加入 retain、release，所以，要使用 ARC 基本上相當簡單，就是先把原本要手動管理記憶體的地方，把 retain、release 都拿掉，在dealloc 的地方，也把 [super dealloc] 拿掉。

但，有了 ARC，也不代表在開發 iOS 或 Mac OS X App 的時候，就不需要了解記憶體管理。例如，我們雖然很多程式會使用 Objective-C 語言開發，但是還是會經常用到 C 語言，我們還是得要了解 C 語言裡頭的記憶體管理。

而且，有時候，ARC 也會把 retain、release 加錯地方。在使用 ARC 之前，我們建議先閱讀，[ARC Best Practices](#) 乙文，裡頭提到絕大多數的問題。我們會在這邊簡單提一些「即使用了 ARC，還是必須要注意的記憶體管理問題」。

ARC 可能會錯誤釋放記憶體的時機

在大概 iOS 5 到 iOS 6 的時代，寫出這樣程式，你會收到 Bad Access 的錯誤而造成 crash：

```
#import <QuartzCore/QuartzCore.h>

@implementation ViewController
- (CGColorRef)redColor
{
    UIColor *red = [UIColor redColor];
    CGColorRef colorRef = red.CGColor;
    return colorRef;
}

- (void)viewDidLoad
{
    [super viewDidLoad];
    CGColorRef red = [self redColor];
    self.view.layer.backgroundColor = red;
}
@end
```

之所以會發生這種錯誤，就在於 Compiler 所認定的「已經不需要使用某個記憶體，因此可以釋放」的時機點有問題。以上面的程式來說，釋放記憶體的時機應該是在 self.view.layer.backgroundColor = red; 這一行之後，但是有一段時間，Compiler 却認為是在 return colorRef 這一行之前，red 這個 UIColor 物件就已經沒有被使用而該被釋放，但釋放了 red，就會造成 red 物件裡頭所包含的 CGColor 也被釋放，因此回傳了已經被釋放了 colorRef 變數而造成 Bad Access。

```
#import <QuartzCore/QuartzCore.h>

@implementation ViewController
- (CGColorRef)redColor
{
    UIColor *red = [UIColor redColor];
```

```

    CGColorRef colorRef = red.CGColor;
    // Compiler 可能會在這邊自動產生 [red release]
    return colorRef;
}

- (void)viewDidLoad
{
    [super viewDidLoad];
    CGColorRef red = [self redColor];
    self.view.layer.backgroundColor = red;
    // 正確釋放記憶體的時機應該是這邊
}
@end

```

要了解哪些地方是 weak reference

另外，ARC 有時候會在一些地方沒做 retain，結果卻又自動多做了一次release 最後導致 Bad Access 的錯誤。我們在講 Selector 的時候提到，我們可以將 target/action 與必要的參數合起來變成另外一種物件，叫做 NSInvocation，在 ARC 環境下從 NSInvocation 拿出參數時，就必須要額外注意記憶體管理問題。

比方說，我們現在要把對 UIApplication 要求開啟指定 URL 這件事情，變成一個 Invocation。

```

NSURL *URL = [NSURL URLWithString:@"http://kkbox.com"];
NSMethodSignature *sig = [UIApplication instanceMethodSignatureForSelector:
                         @selector(openURL:)];
NSInvocation *invocation = [NSInvocation invocationWithMethodSignature:sig];
[invocation setTarget:[UIApplication sharedApplication]];
[invocation setSelector:@selector(openURL:)];
[invocation setArgument:&URL atIndex:2];

```

但假如我們用以下這段 code 的方式，把 invocation 拿出參數的物件的時候，就會遇到 Bad Access 錯誤：

```

NSURL *arg = nil;
[invocation getArgument:&arg atIndex:2];
NSLog(@"%@", arg);
// 在這邊會 crash

```

之所以會 crash 的原因是，我們在透過 `getArgument:&arg atIndex:2` 拿出參數的時候，`getArgument:&arg atIndex:2` 並不會幫我們把 arg 多 retain 一次，而到了用 NSLog 印出 arg 之後，ARC 認為我們已經不會用到 arg 了，所以就對 arg 多做了一次 release，於是 retain 與 release 就變得不成對。

我們要解決這個問題的方法是要把 arg 設為 Weak Reference 或 Unsafe Unretained，讓 arg 這個Objective-C 物件的指標不被 ARC 管理，要求 ARC 不要幫這個物件做任何自動的 retain 與 release，在這邊要使用 `__weak` 或 `__unsafe_unretained` 關鍵字。程式會像這樣：

```

__weak NSURL *arg = nil;
[invocation getArgument:&arg atIndex:2];
NSLog(@"%@", arg);

```

循環 Retain

ARC 也不會排除循環 Retain (Retain Cycle) 的狀況，遇到了循環 Retain，還是會造成記憶體漏水。循環 Retain 就是，A 物件本身 retain 了 B 物件，但是 B 物件又 retain 了 A 物件，結果我們要在釋放 A 的時候才有辦法釋放 B，但是 B 又得要在 B 被釋放的時候才會釋放 A，最後導致 A 與 B 都沒有辦法被釋放。這種狀況通常最可能出現在：

1. 把 delegate 設為 strong reference，我們會在討論 delegate 的時候繼續討論這個狀況。
2. 某個物件的某個 property 是一個 block，但是在這個 block 裡頭把物件自己給 retain 了一份。我們會在討論 block 的時候討論這個狀況。
3. 使用 timer 的時候，到了 dealloc 的時候才停止 timer。

假如我們現在有一個 view controller，我們希望這個 view controller 可以定時更新，那麼，我們可能會使用 `+scheduledTimerWithTimeInterval:target:selector:userInfo:repeats:` 這個 method 建立 timer 物件，指定定時執行某個 selector。我們要特別注意，在建立這個 timer 的時候，我們指定給 timer 的 target，也會被 timer retain 一份，因此，我們想要在 view controller 在 dealloc 的時候，才停止 timer 就會有問題：因為 view controller 已經被 timer retain 起來了，所以只要 timer 還在執行，view controller 就不可能走到 dealloc 的地方。

```
@import UIKit;

@interface ViewController : UIViewController
@property (strong, nonatomic) NSTimer *timer;
@end

@implementation ViewController

- (void)dealloc
{
    [self.timer invalidate];
}

- (void)timer:(NSTimer *)timer
{
    // Update views..
}

- (void)viewDidLoad
{
    [super viewDidLoad];
    self.timer = [NSTimer scheduledTimerWithTimeInterval:1.0
                                                target:self
                                              selector:@selector(timer:)
                                              userInfo:nil
                                             repeats:YES];
}

@end
```

要修正這個問題，我們應該改成，在 `viewDidDisappear:` 的時候，就要停止 timer。

Toll-Free Bridged

前面在講 Category 的時候提到，Foundation Framework 裡頭的每個物件，都有對應的 C 實作，這一層 C 的實作叫做 Core Foundation，當我們在使用 Core Foundation 裡頭的 C 型態時，像是 CFString、CFArray 等，我們可以讓這些型態變成可以接受 ARC 的管理。這種讓 C 型態也可以被當做 Objective-C 物件，接受 ARC 的記憶體管理的方式，叫做 Toll-Free Bridged。

Toll-Free Bridged 有三個語言關鍵字：`__bridge`、`__bridge_retained`、以及 `__bridge_transfer`。我們直接翻譯蘋果官網的定義：

- `__bridge` 會把 Core Foundation 的 C 資料型態轉換成 Objective-C 物件，但是不會多做 retain 與 release。
- `__bridge_retained` 會把 Core Foundation 的 C 資料型態轉換成 Objective-C 物件，並且會做一次 retain，但是之後必須由我們自己手動呼叫 `CFRelease`，釋放記憶體。
- `__bridge_transfer` 會把 Core Foundation 物件轉換成 Objective-C 物件，並且會讓 ARC 主動添加 retain 與 release。

不見得每個 Core Foundation 型態都有辦法轉換成 Objective-C 物件。請參閱蘋果官方的 [詳細說明](#)。

其他

Objective-C 語言有了 ARC 之後，除了禁止使用 retain、release 這些關鍵字之外，也禁止我們手動建立 `NSAutoreleasePool`，同時禁止了一些我們在 ARC 之前的程式寫作方式（或是—奇技淫巧），包括我們不可以把 Objective-C 物件放進 C Structure 裡頭，Compiler 會告訴我們語法錯誤。

在有 ARC 之前，我們之所以會把 Objective-C 物件放進 C Structure 裡，大概會有幾個目的，其一是，假如我們有某個 Class 有很多成員變數，那我們可能會想以下這種寫法將成員變數分成群組：

```
@interface MyClass : NSObject
{
    struct {
        NSString *memberA;
        NSString *memberB;
    } groupA;

    struct {
        NSString *memberA;
        NSString *memberB;
    } groupB;
}
@end
```

這樣，如果我們想要使用 groupA 裡頭的 memberA，可以用 `self.groupA.memberA` 呼叫。

另外一種目的，則是有時候，我們可能會想要刻意隱藏某個 Objective-C Class 裡頭有哪些成員變數。像下面這段 code 裡頭，我們原本有一個 Class 叫做 `MyClass`，裡頭有 `privateMemberA` 與 `privateMemberB` 兩個成員變數，原本應該直接寫在 `MyClass` 的宣告裡頭，但是我們卻刻意把這兩個成員變數包進 `_Privates` 這個 C Structure 裡頭，而原本放在 `MyClass` 成員變數宣告的地方，指剩下了一個叫做 `privates` 的指標，光看到這個指標，讓人難以理解這個 Class 裡頭到底有什麼東西。

```

@interface MyClass : NSObject
{
    void *privates;
}
@end

typedef struct {
    NSString *privateMemberA;
    NSString *privateMemberB;
} _Privates;

@implementation MyClass

- (void)dealloc
{
    _Privates *privateMembers = (_Privates *)privates;
    [privateMembers->privateMemberA release];
    [privateMembers->privateMemberB release];
    free(privates);
    privates = NULL;
    [super dealloc];
}

- (instancetype)init
{
    self = [super init];
    if (self) {
        privates = calloc(1, sizeof(_Privates));
        _Privates *privateMembers = (_Privates *)privates;
        privateMembers->privateMemberA = @"A";
        privateMembers->privateMemberB = @"B";
    }
    return self;
}
@end

```

這種寫法其實是種程式碼保護的技巧，主要在防範 `class-dump`，或是從 `class-dump` 衍生出的 `class-dump-z` 這些工具。`class-dump` 可以從編譯好的 Binary 中還原出每個 class 的 header，當我們從 `class-dump` 抽出別人的 App 的 header，看出有哪些 Class，每個 Class 有哪些成員變數、有哪些 method，也就可以看出整個 App 的架構大致如何。這種寫法就是讓別人用 `class-dump` 倒出我們 App 的 header 時，不會太容易可以了解我們一些重要的 Class 是如何運作，不過，對於做軟體破解的人來說，其實只要花上時間，所有軟體都有辦法破解就是了。

怎樣做逆向工程不是這份文件的重點。總之，有了 ARC 之後，我們都無法繼續使用以上兩種的程式寫作方式。

相關閱讀

- [ARC Best Practices](#)
- [蘋果官方文件 Transitioning to ARC Release Notes](#)

- 蘋果官方文件 Advanced Memory Management Programming Guide
- 蘋果官方文件 Memory Management Programming Guide for Core Foundation
- 蘋果官方文件 Toll-Free Bridged Types
- WWDC 2013 Advances in Objective-C

記憶體管理 Part 3

本章主要討論跟 UIViewController 相關的記憶體相關問題，嚴格說起來比較像是在討論 UIViewController 的 life cycle。

總之，我們要回答的問題是—當我們建立了一個 UIViewController 之後，Xcode 紿我們的 template 中，會叫我們去實作一個叫做 `didReceiveMemoryWarning:` 的 method，然後你可能從一些相關文件上知道，當系統記憶體不夠的時候，我們應該要在這個 method 裡頭釋放一些記憶體，那麼，有哪些記憶體是應該要釋放的？我們應該怎麼實作這個 method？

記憶體不足警告（Memory Warnings）

在 Desktop 作業系統中，如果實體記憶體不足，應用程式使用的記憶體量，超過實體記憶體的數量，這時候作業系統會自動將記憶體中的部分資料，存入磁碟的虛擬記憶體（Virtual Memory）當中，需要使用的時候，再從虛擬記憶體中載回實體記憶體，Mac OS X就有這樣的機制。

iOS 在發展之初到現在，都沒有虛擬記憶體，而是會在記憶體快要用完的時候，對應用程式發出記憶體不足警告，要求釋放一些可以暫時不需要用到的物件，讓應用程式可以有足夠的記憶體繼續運作。如果無視記憶體警告，繼續放任記憶體用量成長，系統最後便會強制要求終止應用程式。

在記憶體不足的時候，除了會對 UIApplication 的 delegate（就是所謂的 AppDelegate）呼叫 `applicationDidReceiveMemoryWarning:` 之外，也會對系統中所有的UIViewController 呼叫 `didReceiveMemoryWarning:`。如果我們想要知道哪些記憶體是可以在 `.didReceiveMemoryWarning:` 釋放的，不妨先回顧一下在 iOS 6 之前，iOS 是怎麼做的一—

從 iOS 問世到 iOS 5，只要發生記憶體不足，就會把所有不在最前景的 View Controller 的 view 釋放掉。因為這些 View Controller 的 view 並不在畫面上，用戶根本看不到，所以暫時先放掉也沒有關係。

iOS 6 之前記憶體不足時系統主動釋放 View 的行為

所謂不在最前景的 view controller 就是：假如我們今天有一個 tab bar controller，tab bar 裡頭有四個項目，對應到四個 view controller，但是其實只會顯示一個，那麼，在 iOS 6 之前，只要發生記憶體警告的時候，其他三個view controller 的 view 就會被釋放。

在 navigation controller 的navigation stack 裡頭，也只有最上面的 view controller的畫面需要顯示，其他 view controller 的 view 也可以被釋放。

所以，如果你曾經在 iOS 6 之前的環境上開發過 iOS App，可能會遇到一個奇怪的 bug：你把一些狀態直接記錄在 view 裡頭，像是改變了一些 label 裡頭的文字，但是繼續做了一些操作，然後回到這個 view 之後，發現 view 莫名其妙的回復到初始值，原本放在 label 的文字不見了，其實就是遇到了記憶體警告的結果。

UIViewController 與 View 的關係

`UIViewController` 負責管理在應用程式中每個會用到的畫面，最主要的 property 就是 `view`，而這個 property 是使用**Lazy Loading** pattern 實作。Lazy Loading 就是：**我們要去使用某個物件的時候，我們才去建立那個物件**，避免在物件初始時就建立了所有的property，而達到讓初始物件這個動作加速的效果。¹

當我們在透過 `alloc` 、`init` 或 `initWithNibName:bundle:` 建立 View Controller 的時候，並不會馬上建立 view，而是當我們呼叫 `view` 這個屬性的時候才會建立。我們以下面的程式為例：

```
// 建立 MyViewController 的 instance, 這時候還沒有建立 view
MyViewController *controller = [[MyViewController alloc]
    initWithNibName:@"MyViewController" bundle:nil];
// 在被加入到 navigation stack 的時候, 會去呼叫 [controller view]
// 這時候 view 才被建立起來
[navigationController pushViewController:controller animated:YES];
[controller release];
```

用 Lazy Loading 的方式實作一個 getter 的方式大致如下。在我們自己的程式中，想要有效使用記憶體，我們也可以嘗試這麼寫。

```
- (UIView *)view
{
    if (!_view) {
        _view = [[UIView alloc]
            initWithFrame:[UIScreen mainScreen].bounds];
    }
    return view;
}
```

不過，`UIViewController` 在還沒有 view，而要去建立 view 的時候，會呼叫的其實是 `loadView` 這個 method，在 view 成功載入之後，則會呼叫 `viewDidLoad`。我們雖然不知道蘋果到底是怎麼實作 `UIViewController`，但不外乎類似這樣：

```
- (UIView *)view
{
    if (!_view) {
        [self loadView];
        if (_view) {
            [self viewDidLoad];
        }
    }
    return view;
}
```

所以，如果你有天不小心寫出像下面的程式碼，就會進入無窮迴圈：因為呼叫 `[self view]` 的時候發現沒有 view，就會呼叫 `loadView`，但 `loadView` 又去呼叫 `[self view]`。

```
- (void)loadView
{
    [self view];
}
```

在 iOS 6 之前，如果某個 View Controller 不在最上層，發出記憶體警告時，系統就會通知這些 View Controller 把 view 指向 nil；而當我們再次需要使用這個 View Controller 的時候，就會因為呼叫到 `view`，而把 view 重新載入回來。

所以我們要注意，`viewDidLoad` 並不是 `UIViewController` 的 Initializer，—雖然我們在開始使用某個 view controller 的時候，一定會呼叫到一次 `viewDidLoad`，我們也通常會在這個地方，做一些初始化這個 view controller的事情—但 `viewDidLoad` 是有機會在 View Controller 的 Life Cycle 中被重複呼叫好幾遍—在建立了 view 之後，view 也可以再次指向 nil，所以 view controller 可能會被重複釋放與載入 view，`viewDidLoad` 也會被重複呼叫。

所以在 iOS 6 之前，你曾經遇到某個 View Controller 回復到初始值這樣的問題，就是：原本有狀態的 view 因為記憶體警告被釋放了，而我們如果在 `viewDidLoad` 再次被呼叫的時候，沒有正確還原狀態，自然只有初始狀態的 view。

iOS 如何知道哪個 View Controller 位在最上層？

那麼，view controller自己怎麼知道自己位在最前景呢？其實很簡單：view controller 被放到最上層時，會被呼叫到 `viewWillAppear:` 以及 `viewDidAppear:`，離開最上層時，會呼叫 `viewWillDisappear:` 與 `viewDidDisappear:`。

只有呼叫過 `viewWillAppear:` 以及 `viewDidAppear:`，而沒有呼叫過 `viewWillDisappear:` 與 `viewDidDisappear:` 的 View Controller，就是位在最前景的 View Controller。

我們經常會 override `viewWillAppear:` 這些 method，在做 override 的時候，應該要呼叫一次 super 的實作，因為 super 的 `viewWillAppear:` 這些 method 實際作了一些必要的事情，在 iOS 6 之前是用來確保哪些 view 該被釋放，雖然蘋果推出 iOS 6 時，或許是認為像是 iPhone 5這樣的裝置在可用資源上遠遠超越過去的硬體，因此不再刻意釋放 view，但呼叫一下 super 的實作還是比較保險。

所以我們應該要在 `didReceiveMemoryWarning:` 做什麼？

從過去的經驗來看，不在最上層的 view 實際可以釋放，在 iOS 6 之後，當我們遇到記憶體不足時，我們也可以選擇性的決定要不要釋放 view，像 web view 這種記憶體怪物在沒用到的時候實在應該要放掉。

以下是蘋果的範例程式：

```
- (void) didReceiveMemoryWarning
{
    [super didReceiveMemoryWarning];
    if ([self.view window] == nil) {
        self.view = nil
    }
}
```

相關閱讀

- Resource Management in View Controllers

¹. 也可以參見 Wikipedia 上的說明 http://en.wikipedia.org/wiki/Lazy_loading ↪

Delegate 與 Protocol

在閱讀這一章之前，相信你應該已經寫過一些簡單的 iOS 程式，知道如果我們想要一個像是系統設定 (Setting.app) 那樣的表格介面，我們會建立 `UITableViewController` 的 subclass，這個 controller 的 view 是 `UITableView`。

當我們想要設定表格中的內容，像是這個表格中有多少個 section、每個 section 裡頭有多少 row、每個 row 裡頭又是哪些內容... 我們不是直接呼叫 `UITableView` 的 method，像是呼叫 `[myTableView setSectionCount:3]` 或是 `[myTableView setRowCount:3 atSection:4]`，而是去實作 `UITableViewController` 裡頭幾個像是 template method 的東西，像 `numberOfSectionsInTableView:`、`tableView:numberOfRowsInSection:` 等等。

為什麼不是直接去改變 view，而是 controller 要準備一些不知道會被誰呼叫的 method？理由是：`UITableViewController` 是 `UITableView` 的 data source 與 delegate。那，什麼是 delegate？

用我的話來說，delegate 就是 **將眾多的 callback，集中在一個物件上**。

如果你還不知道怎麼建立 Table View 或 Collection View，請先做完 *Beginning iOS 7 Development Exploring the iOS SDK* 這本書的第七章到第十章的相關練習。這本書好像中文只有翻譯出 iOS 5 的版本，叫做《探索 iOS 5 程式開發實戰》，不過章節位置差不多。

從其他平台來看 Objective-C 的 Delegate

Delegate 算是入門 Objective-C的另外一個障礙，但了解之後就會知道其實很簡單，一開始可能覺得並不好懂的主要原因是，delegate的這套作法，跟其他平台在做同樣的事情的時候，作法比較不一樣。

如果你曾經在其他的平台上開發過應用程式，像是微軟的 .Net 平台的話，可以發現，在 C# 語言中在講一個 class 有哪些成員的時候，會包括 properties、methods 以及 events；但是我們在講 Objective-C 的 class 時，並不會提到 events，C# 中使用 events 在做的事情，在 Objective-C 中，我們往往使用 target/action 與 delegate 實作。¹

雖然在 Mac OS X 與 iOS 上分別有 `NSEvent` 與 `UIEvent`，但是這邊的 events 與 .Net 裡頭講的 events 又是兩件事情：`NSEvent` 是用來描述鍵盤按了哪個按鍵，滑鼠移到了什麼位置，而 `UIEvent` 則用來描述觸控事件以及耳機上的播放控制按鈕等等，單純用來描述從硬體輸入了什麼事件，透過作業系統傳到我們的應用程式中。而 C# 裡頭所稱的 events，則是用來處理 callback。

所謂 callback 是指，當我們呼叫了一個 function 或 method 之後，可能會花上許多時間，或是計算的是大量資料，或是需要透過網路連線，所以我們並不馬上要求得到回傳的結果，而是等到一段時間之後，計算結果才會透過另外一個 function/method 傳回來。

現在絕大多數的應用程式開發環境都採用 MVC 架構，我們將物件分成三類：model、view、controller，雖然在不同的平台上往往實作出來的結果不太一樣，像是在微軟的平台上，往往把 window（或是 form）當做是 controller 使用，由 window 物件負責管理放在這個 window 上面所有的 control 元件，但是 Mac OS X 上面 window 並不拿來當做 controller，而是被當成 view，controller 在 window 之外，而且一個 controller 也可以控制多個 window...但都大抵如此。

在 MVC 的架構中，我們通常會先建好 controller class，然後加入 model 與 view，變成 controller 的成員變數；因為 model 與 view 是 controller 的成員變數，所以 controller 可以直接呼叫 model 與 view，那麼，當 model 與 view 發生了變化，要回來通知 controller，我們也可以稱之為 callback，例如，controller 建立了一個按鈕，但是直到按鈕被點選之後，controller 才負責做事。

在 C# 中，我們要處理一個 event，就要提供一個 event handler，我們現在要處理的是 Click：

```
private void InitializeComponent()
{
    this.button1 = new System.Windows.Forms.Button();
    this.button1.Click += Button1_Click;
}

private void Button1_Click(object sender, System.EventArgs e)
{}
```

如果只是單純的點選事件的話，我們會用 target/action 實作。但，對照 .Net framework 裡頭，events 不只是 click 而已，還可能會有 double click、triple click、quadruple click...，如果是 C# 裡頭，會這麼寫：

```
this.button1.Click += Button1_Click;
this.button1.DoubleClick += Button1_DoubleClick;
this.button1.TripleClick += Button1_TripleClick;
this.button1.QuadrupleClick += Button1_QuadrupleClick;
```

變成 Objective-C 的話就可能變成：

```
[button1 setTarget:self];
[button1 setAction:@selector(click:)];
[button1 setDoubleTarget:self];
[button1 setDoubleAction:@selector(doubleClick:)];
[button1 setTripleTarget:self];
[button1 setTripleAction:@selector(tripleClick:)];
[button1 setQuadrupleTarget:self];
[button1 setQuadrupleAction:@selector(QuadrupleClick:)];
```

這樣寫起來實在很讓人煩躁（雖然 `NSTableView` 也的確有 `setDoubleAction: ...`），所以，當這樣的東西一多，在 Objective-C 語言裡頭，會直接準備好一個物件，這個物件準備好了所有可以呼叫的 method，這個物件就叫做 delegate，而這些可以呼叫的 method 的集合，叫做 protocol。準備好這個物件之後，我們就不用呼叫這麼多 `setDoubleAction:` 、`setTripleAction:`，只要呼叫 `setDelegate:`。

¹. 我在寫這章的時候，一直在想拿 C# 到底是不是好主意，畢竟想要學 Objective-C 語言者，不見得都有 C# 的基礎。之所以以 C# 舉例，原因是這份資料其實是來自於當初我個人在公司內部的教材，而公司當時進來的新人之前是寫 C# 的。 ↵

設計 Protocol 與實作 Delegate 的方式

我們來用 delegate 的想法來實作前面提到的狀況。

宣告 Protocol 與 Delegate 的方式

我們先來建立一個 `NSButton` 的 subclass，叫做 `MyButton`，`MyButton` 的 delegate 必須實作 `MyButtonDelegate` 這個 protocol。`.h` 檔案中宣告如下：

```
#import <Cocoa/Cocoa.h>

@class MyButton;

@protocol MyButtonDelegate
- (void)myButtonDidBecomeClicked:(MyButton *)button;
- (void)myButtonDidBecomeDoubleClicked:(MyButton *)button;
- (void)myButtonDidBecomeTripleClicked:(MyButton *)button;
- (void)myButtonDidBecomeQuadrupleClicked:(MyButton *)button;
@end

@interface MyButton : NSButton
{
    id <MyButtonDelegate> delegate;
}
@property (weak, nonatomic) id <MyButtonDelegate> delegate;
@end
```

逐行解釋這個 header 裡頭的內容：

1. `#import <Cocoa/Cocoa.h>`：因為 `NSButton` 是在 AppKit 裡頭，所以我們必須呼叫對應的 header。
2. `@class MyButton;`：因為在我們接下來的 protocol 告知中，會用到 `MyButton` 這個 class，但是 `MyButton` 其實是宣告在 `MyButtonDelegate` 的下面，所以我們需要先預先宣告 `MyButton` 這個 class 的存在。
3. 從 `@protocol MyButtonDelegate` 開始，就是在宣告 `MyButtonDelegate` 這個 protocol 裡頭的四個 method。
4. 接下來宣告 `MyButton` 這個 class，裡頭有一個叫做 delegate 的變數。

實作 Delegate Methods

假如我們有一個叫做 `MyController` 的 controller 物件，要成為 `MyButton` 的 delegate，我們會這麼做。首先是 `.h` 部分：

```
@interface MyController : NSObject <MyButtonDelegate>
{
    IBOutlet MyButton *myButton;
}
@end
```

我們要先宣告 `MyController` 有實作 `MyButtonDelegate` 這個 protocol，如此一來，假如 `MyController` 漏了實作哪些定義在 `MyButtonDelegate` 裡頭的 method，在編譯的時候就會跳出警告，要求我們修正，如果我們還是不實作的話，執行時，就會發生找不到 selector 對應的實作的錯誤而 crash。

如果 `MyController` 又是其他物件的 delegate的話，我們可以在這段用大於與小於間包起來的宣告，繼續加入其他的 protocol的名稱，例如 `<MyButtonDelegate, AnotherProtocol>`。至於一個物件是否有實作某個 protocol，我們可以用 `conformsToProtocol:` 檢查。

在 `MyController` 的實作中，我們就只要將 `myButton` 的 delegate設成自己，然後實作該實作的 method。我個人不太喜歡在 protocol的宣告中出現大量註解，因為在實作 protocol的時候，最方便的方式就是直接把protocol宣告的 method 複製貼上，接著逐一把肉放進 protocol定義的骨幹裡；如果當中出現註解，複製貼上之後，還要把這些註解刪掉，其實還頂煩人。

```
@implementation MyController

- (void)awakeFromNib
{
    [myButton setDelegate:self];
}

#pragma mark - MyButtonDelegate

- (void)myButtonDidBecomeClicked:(MyButton *)button
{
}
- (void)myButtonDidBecomeDoubleClicked:(MyButton *)button
{
}
- (void)myButtonDidBecomeTripleClicked:(MyButton *)button
{
}
- (void)myButtonDidBecomeQuadrupleClicked:(MyButton *)button
{
}

@end
```

我們在這邊透過 `setDelegate:` 指定 delegate，如果我們把 delegate宣告成是一個 IBOutlet的話，也可以直接在 Interface Builder 中連結。

Delegate Methods 是怎麼被呼叫的？

`MyButton` 是怎樣呼叫 delegate 的呢？其實很簡單。

```
@implementation MyButton

- (void)mouseDown:(NSEvent *)theEvent
{
    switch ([theEvent clickCount]) {
        case 1:
            [delegate myButtonDidBecomeClicked:self];
```

```
        break;
    case 2:
        [delegate myButtonDidBecomeDoubleClicked:self];
        break;
    case 3:
        [delegate myButtonDidBecomeTripleClicked:self];
        break;
    case 4:
        [delegate myButtonDidBecomeQuadrupleClicked:self];
        break;
    default:
        break;
    }
}

@synthesize delegate;
@end
```

注意事項

在上面的範例中，我們看到了設計 delegate 與 protocol 應該注意的地方：

Delegate 物件不應該指定 Class

我們將 delegate 物件宣告成 `id <MyButtonDelegate> delegate`，意思就是不需要管這個物件屬於哪個 class，只要是個 Objective-C 物件即可，但是這個物件必須實作 `MyButtonDelegate` protocol。

我們其實可以將 delegate 物件是那個 class 寫死，例如把 `MyButton` 的 delegate 的 class 指定成 `MyController`，但這樣做非常不好，如此一來，就只有 `MyController` 可以使用 `MyButton`，其他 controller 都無法使用，就大大減少了重複使用 `MyButton` 的彈性。

Delegate 這種設計方式，也方便我們在同時開發 Mac OS X 與 iOS 跨平台專案時共用程式碼，我們在撰寫某個 model 物件的時候，只使用 Foundation 或是其他兩個平台都有的 framework，至於與平台相依的部份，就放進 delegate 中，然後在 Mac OS X 與 iOS 上各自實作 delegate 物件。

總之，在實作 `delegate` 的時候，`delegate` 屬於哪個 class 並不重要，重要的是 `delegate` 物件有沒有實作我們想要呼叫的 `method`。

Delegate 屬性應該要用 Weak，而非 Strong

在使用 property 語法的時候，如果這個 property 是 Objective-C 物件，我們照理說應該要設定成 `strong` 或 `retain`，但是遇到的是 `delegate`，我們應該設成 `weak` 或 `assign`。

原因是：需要設計 `delegate` 物件的這個物件，往往是其 `delegate` 物件的成員變數。在我們的例子中，`MyButton` 的 instance 是 `myButton`，是 `MyController` 的成員變數，自己可能已經被 `MyController` `retain` 了一份。如果 `MyButton` 又 `retain` 了一次 `MyController`，就會出現循環 `retain` 的問題—我已經被別人 `retain`，我又把別人 `retain` 一次。

如此，會造成我們會無法釋放 `MyController`：在該釋放 `MyController` 的時候，`MyController` 還是被自己的成員變數 `retain`，`MyController` 得要走到 `dealloc` 才會釋放 `myButton`，但是自己卻因為被 `myButton` 給 `retain` 起來，而始終走不到 `dealloc`。

Delegate Method 的命名方式

Delegate method 的命名有個鮮明的特色，就是這個 method 至少會傳入一個參數，就是把到底是誰呼叫了這個 delegate method 傳遞進來。同時，這個 method 也往往以傳入的 class 名稱開頭，讓我們可以辨識這是屬於哪個 class 的 delegate method。以 `UITableViewDelegate` 為例，假如我們在 iOS 的表格中，選擇了某一列，就會呼叫

```
- (void)tableView:(UITableView *)tableView
didSelectRowAtIndexPath:(NSIndexPath *)indexPath
```

Method 的名稱就以「`tableView`」開頭，讓我們知道這是屬於 Table View 的 delegate method，然後第一個參數把這個 Table View 的 instance 傳入，接下來才傳入到底是哪一列被選起來的資訊。

至少把是誰呼叫了這個 delegate method 傳入的理由很簡單。以我們的 `MyController` 為例，這個 controller 可能有好幾個 `MyButton`，而這些 `MyButton` 全都把 delegate 指到同一個 controller 上，那麼，controller 就需要知道到底是被哪個 button 呼叫。判斷方式只要簡單比對指標就好了：

```
- (void)myButtonDidBecomeQuadrupleClicked:(MyButton *)button
{
    if (button == myButton1) {
    }
    else if (button == myButton2) {
    }
}
```

Data Source 與 Delegate 的差別？

我們現在可以來看看 `UITableView` 與 `UIViewController` 是怎麼運作的。

`UIViewController` 在 `loadView` 中建立了一個 `UITableView` 的 instance，指定成是自己的 view，同時將這個 view 的 delegate 與 data source 設定成自己。一個 class 可以根據需要，將 delegate 拆成好幾個，以 `UITableView` 來說，跟表格中有什麼資料有關的，就放在 data source 中，其餘的 method 放在 delegate 中。

我們在 Mac OS X 會用到的最龐大的 UI 元件，莫過於 `WebView`，雖然在 iOS 上 `UIWebView` 被閹割到只剩下四個 delegate method，但是 Mac OS X 上足足有五大類 delegate method，網頁頁框的載入進度、個別圖片檔案的載入進度、下載檔案的 UI 呈現、該不該開新視窗或是新分頁、沒有安裝 Java 或是 Flash 要怎麼呈現、用 JavaScript 跳出 alert 該怎麼呈現...都是一堆 delegate method。

假如先不管 `UITableView` 怎麼重複使用 `UITableViewCell` 的機制（這個機制還頂複雜），我們要更新 `UITableView` 的資料時，先指定 data source 物件後，要呼叫一次 `reloadData`。 `reloadData` 可能是這樣寫的：

```
- (void)reloadData
{
    NSInteger sections = 1;
    if ([dataSource respondsToSelector:
        @selector(numberOfSectionsInTableView:)]) {
        sections = [dataSource numberOfSectionsInTableView:self];
    }
    for (NSInteger section = 0; section < sections; section++) {
        NSInteger rows = [dataSource tableView:self
            numberOfRowsInSection:section];
        for (NSInteger row = 0; row < rows; row++) {
            NSIndexPath *indexPath = [NSIndexPath indexPathForRow:row
                inSection:section];
            UITableViewCell *cell = [dataSource tableView:self
                cellForRowAtIndexPath:indexPath];
            // Do something with the cell...
        }
    }
}
```

我們注意到幾件事情：首先，因為 `numberOfSectionsInTableView:` 被定義成是 optional 的 delegate method，delegate 不見得要實作，所以我們會用 `respondsToSelector:` 檢查是否有實作。我們可以在 protocol 的宣告中，指定某個 delegate method 是 required 或是 optional，如果不特別指定的話，預設都是 required。我們簡單看一下 `UITableViewDataSource` 就知道如何定義 required 與 optional 的 delegate method。

```
@protocol UITableViewDataSource<NSObject>
@required
- (NSInteger)tableView:(UITableView *)tableView
    numberOfRowsInSection:(NSInteger)section;
- (UITableViewCell *)tableView:(UITableView *)tableView
    cellForRowAtIndexPath:(NSIndexPath *)indexPath;
```

```
@optional  
- (NSInteger)numberOfSectionsInTableView:(UITableView *)tableView;  
// ....  
@end
```

另外，就是定義在 data source 的 method，是在 `reloadData` 中被呼叫，因此我們可以知道 `UITableView` 的 data source 與 delegate的最大差別：我們絕對不可以在 data source 定義的 method 中呼叫 `reloadData`，不然就會進入無窮迴圈！

Formal Protocol 與 Informal Protocol

`@protocol` 這個關鍵字是在 Objective-C 2.0 之後出現的，在這之前要定義 protocol，則是寫成 `NSObject` 的 category，前者叫做 formal protocol，後者則稱為 informal protocol。UIKit 問世時就採用 Objective-C 2.0 語法，至於 Mac OS X，蘋果在 2008 年開始大幅改寫 Foundation 與 AppKit，現在（2012 年）絕大多數可以看到的 protocol，都是 formal protocol，但如果你在 maintain 一份稍微有點歷史的程式，或是在蘋果少數的 API 中，還是可以看到 informal protocol。

在 Core Animation 裡頭，就可以看到 `CALayerDelegate` 、`CALayoutManager` 、`CAAnimationDelegate` ，都還是 informal protocol。其中 `CALayerDelegate` 、`CALayoutManager` 兩者之間還夾著 `CAACTION` 這個 formal protocol—在兩個 informal protocol 中間夾著一個 formal protocol，實在讓人很反感—為什麼不起改掉呢？至於 `CAAnimationDelegate` 也很怪異：CAAnimation 的 delegate 不是用 `assign`，而是會 `retain` 起來。

無所不在的 Delegate

由於在 Objective-C 語言中，delegate 相當於 event handler 的用途，所以，當你在其他平台中看到 event handler 用得多頻繁，就等於 delegate 用得多頻繁。舉例來說：

- 在使用 `NSURLConnection` 抓取網路上的資料的時候，無論收到了 HTTP response code、是否連線失敗、是否連線結束...都是透過 delegate 回傳。
- 在使用 Core Location 的時候，如果 `CLLocationManager` 找到了我們的所在位置，或是發現我們正在移動，也都會透過 delegate 通知。
- 當我們要使用手機拍照、傳送簡訊或是電子郵件等等，當照片拍完，會用 delegate 回傳 image 物件，簡訊或是電子郵件傳送成功，也會用 delegate 告訴我們執行完畢。

甚至，當我們在寫一個 iOS 程式的第一步，其實都是在實作一個 delegate method。我們在 Xcode 裡頭開了一個新專案之後，下一步往往是實作 `application:didFinishLaunchingWithOptions:` 這個 method，但是要了解整個程式的進入點，我們要從 `main.m` 來看。裡頭通常只有簡短的幾行：

```
int main(int argc, char *argv[])
{
    @autoreleasepool {
        return UIApplicationMain(argc, argv, nil, nil);
    }
}
```

一個 iOS 程式是從 `main` 這個 function 開始，接著透過呼叫 `UIApplicationMain` 建立 `UIApplication` 這個 Singleton 物件。`UIApplication` 用來代表一個應用程式的基本狀態，包括 icon 上面該顯示多少則 push notification 的數量、支援水平還是垂直畫面、是否顯示狀態列等，當 `UIApplication` 物件被建立起來後，就要通知它的 delegate—程式已經開啟了，請進行下一步，這個 delegate method 就是

`application:didFinishLaunchingWithOptions:`，我們在這邊建立基本的 view controller 與 window，顯示出來。

也就是說，當我們在開始寫第一行 iOS 程式的時候，我們就起碼需要了解什麼是 Singleton 和 delegate，但是在了解之後，想要知道 Mac OS X 與 iOS 中眾多的元件該如何使用，以及怎樣用比較好的方式設計自己的元件，就不是問題了

其他平台上所謂的 Delegate

在其他平台中，也用到了 delegate 這個詞，但是意義不太一樣。

Design Pattern 中所講的 Delegate

就我的理解，Design Pattern 中所講的 Delegate Pattern，比較像是做一個 Wrapper，有一個 class 在實作 method 時，其實是直接把這個 method 的實作傳遞到自己的成員變數物件的實作上。以 Objective-C 語言實作會像這樣：

首先產生一個內部的物件，叫做 `MyInnerClass`：

```
@interface MyInnerClass : NSObject
- (void)doSomething;
@end
@implementation MyInnerClass
- (void)doSomething
{
    NSLog(@"Do something");
}
@end
```

然後，`MyClass` 會把該做的事情，都交給 `MyInnerClass`：

```
@interface MyClass : NSObject
{
    MyInnerClass *innerObject;
}
- (void)doSomething;
@end
@implementation MyClass
- (void)dealloc
{
    [innerObject release];
    [super dealloc];
}
- (id)init
{
    self = [super init];
    if (self) {
        innerObject = [[MyInnerClass alloc] init];
    }
    return self;
}
- (void)doSomething
{
    [innerObject doSomething];
}
@end
```

在 Cocoa Framework 中，會比較像是 `NSButton` 與 `NSButtonCell` 的關係。你或許會問，為什麼 Objective-C 裡頭的 delegate 與 Design Pattern 裡頭講的 Delegate Pattern 意義不一樣？為什麼 Objective-C 不按照這套用法？但其實是，Objective-C 使用 delegate 這個觀念，早於 Design Pattern 成書。

C# 中所謂的 Delegate

C# 語言中也有 delegate 這個關鍵字，不過用途卻是處理 anonymous function，以我們上面的例子，我們打算用 C# 增加按鈕被點選的 event handler，原本是這麼寫：

```
private void InitializeComponent()
{
    this.button1 = new System.Windows.Forms.Button();
    this.button1.Click += Button1_Click;
}

private void Button1_Click(object sender, System.EventArgs e)
{}
```

在 C# 2.0 可以寫成這樣：¹

```
private void InitializeComponent()
{
    this.button1 = new System.Windows.Forms.Button();
    this.button1.Click += delegate(object sender, System.EventArgs e) {
        // Do something here.
    };
}
```

關於在 Objective-C 語言中怎麼使用 anonymous function，我們會在接下來的章節，講 block 的時候討論。

¹. 參見 <http://msdn.microsoft.com/en-us/library/bb882516.aspx> ↩

我們曾經犯過的低級錯誤

最後來提一個我們之前花了半個月才找出問題在哪的 bug。問題出在 protocol 不該這麼設計。我們寫了一個 class，這個 class 有兩個 method：begin 與 stop，按下 begin 的時候要開始做一件事情，之後想要停止，就呼叫 stop，要開始或要結束的時候，都會通知 delegate。程式大概是這樣：

```

@class MyClass;
@protocol MyClassDelegate <NSObject>
- (void)myClassWillBegin:(MyClass *)myClassss;
- (void)myClassDidBegin:(MyClass *)myClassss;
- (void)myClassWillStop:(MyClass *)myClassss;
- (void)myClassDidStop:(MyClass *)myClassss;
@end

@interface MyClass : NSObject
{
    id <MyClassDelegate> delegate;
}
- (void)begin;
- (void)stop;
@property (assign, nonatomic) id <MyClassDelegate> delegate;
@end

@implementation MyClass

- (void)begin
{
    [delegate myClassWillBegin:self];
    // Do something
    [delegate myClassDidBegin:self];
}
- (void)stop
{
    [delegate myClassWillStop:self];
    // Do something
    [delegate myClassDidStop:self];
}
@synthesize delegate;
@end

```

這個 protocol 有什麼問題呢？前面提到，在 UITableView 的 data source 的 method 裡頭不該呼叫 reloadData 一樣，這邊的幾個 delegate method 的實作裡頭，也都不該隨意的呼叫 begin 與 stop，而我在 myClassWillBegin: 裡頭想要做一些檢查，如果在某些條件下，這件事情不該跑起來，而應該停止，所以我在 myClassWillBegin: 裡頭呼叫了 stop。但這麼做，並不會讓這件事情結束，因為 begin 這個 method 在對 delegate 呼叫完 myClassWillBegin: 之後，程式還是會繼續往下走，所以還是把 begin 整個做完了。

這個 protocol 應該這麼設計：

```
@class MyClass;
@protocol MyClassDelegate <NSObject>
- (BOOL)myClassShouldBegin:(MyClass *)myClassss;
- (void)myClassDidBegin:(MyClass *)myClassss;
- (BOOL)myClassShouldStop:(MyClass *)myClassss;
- (void)myClassDidStop:(MyClass *)myClassss;
@end

@interface MyClass : NSObject
{
    id <MyClassDelegate> delegate;
}
- (void)begin;
- (void)stop;
@property (assign, nonatomic) id <MyClassDelegate> delegate;
@end
@implementation MyClass

- (void)begin
{
    if (![delegate myClassShouldBegin:self]) {
        return;
    }
    // Do something
    [delegate myClassDidBegin:self];
}
- (void)stop
{
    if (![delegate myClassShouldStop:self]) {
        return;
    }
    // Do something
    [delegate myClassDidStop:self];
}
@synthesize delegate;
@end
```

相關閱讀

- 蘋果官方文件 Working with Protocols

練習：貪食蛇

練習範圍

- MVC
- DTO (Data Transfer Objects)
- Quartz 2D
- Delegate
- Queue 資料結構
- NSTimer
- UIGestureRecognizer

練習目標

我們要寫一個 iOS 版本的貪食蛇遊戲：

- 這個遊戲一開始畫面中央只有一個開始按鈕，按下開始之後才會開始遊戲
- 遊戲開始時，畫面中央有一條長度為 2 單位的蛇，另外有一個水果，蛇的初始方向是往左方走
- 每格 0.5 秒蛇會移動一格
- 我們可以用 swipe 手勢改變蛇的方向，但蛇只能夠往左右轉。也就是，當蛇在往左方走的時候，只有往上或往下的 swipe 手勢有用
- 如果蛇的頭碰到了水果，蛇的身體長度就會加 2 單位
- 如果蛇的頭碰到了畫面邊界，會從畫面的另外一邊冒出來
- 如果蛇的頭碰到了自己的身體，遊戲結束，重新出現開始按鈕。按下開始按鈕會重新開始遊戲

練習內容

MVC

我們使用 MVC 架構寫這個小遊戲

- Model:
 - 蛇的 Model
 - property 包括：
 - 蛇目前的身體每個座標點的 array，座標點是我們自己定義的 Class，只有兩個屬性：x 與 y，都是 NSInteger
 - 蛇目前的行進方向
 - method 包括：
 - 要求蛇移動一格
 - 要求蛇增加長度
 - 詢問蛇現在頭是否碰到自己的身體
 - 詢問蛇的頭是否剛好碰到某個點
 - 水果：就只是一個座標點
- View:
 - 負責在畫面中使用 Quartz 2D 繪製蛇與水果

- 上面被加上了 UIGestureRecognizer 的動作，指向 Controller，當發生 Swipe 事件時，會通知 Controller 要改變蛇的方向
- Controller:
 - 擁有蛇與水果的 Mode
 - 擁有 View
 - View 與 Controller 是 delegate 的關係，當 View 要重繪時，會跟 Controller 索取一次蛇與水果的 Model
 - 執行一個 Timer，每執行一次，會要求蛇移動一格，並要求 View 重繪一次。並且檢查蛇是否撞到了水果或自己的身體，決定是否要延長蛇的身體，或是宣布遊戲結束

Queue 資料結構

蛇的身體其實是一個 Queue，Queue 裡頭是一堆座標，當蛇在移動的時候，事實上是位在尾巴的座標物件被 pop 掉，而我們從蛇的頭的位置 push 進一個座標。

單元測試

我們要對蛇的 Model 寫單元測試。寫單元測試時，要注意 AAA 原則：Arrange、Act、Assert。像是：

- Arrange：先定義蛇如果要移動一次，移動後身體應該會出現在什麼位置
- Act：要求蛇移動一次
- Assert：確定蛇在移動之後，身體位置跟我們在 Arrange 時設定的位置一致

我們應該先寫蛇的 Model，然後寫單元測試，最後才去寫 View 與 Controller。

單元測試

單元測試（Unit Test）就是以程式測試程式。—當你在開發軟體的時候，你不但撰寫了原本的功能，同時也寫出每個功能對應的程式碼，測試每個功能是否正常運作。

每次在講到單元測試的時候，總是會有「軟體開發過程是否應該要寫單元測試」這樣的疑問，你會看到有不少人說寫單元測試會佔用額外的時間，如果開發軟體是為了創業，那麼目標應該是要儘快完成、儘快上線，才能夠儘快了解用戶與市場真正的需求。

我想，對這個問題，我們先不要用爭辯的方式討論，我們來換個方式—我們用體驗的。我們先不要從別人的意見中判斷單元測試好不好、有沒有用，我們先來寫一次單元測試，再來評價—看看你覺得寫單元測試會不會花很多時間？對你有沒有幫助？如何？

在上一章結束時，我們要寫一個 [貪食蛇](#) 遊戲當做練習。我們在這個遊戲中，我們於是寫了相關的程式，像是控制蛇的移動，還有在蛇吃到水果的時候，尾巴要加長...等等。接下來就是執行、測試，我們把程式跑起來，可能會覺得這條蛇可能哪裡怪怪的，可能在吃到水果的瞬間，尾巴並沒有立刻變長，而是等蛇再走了一兩格才變得比較對，或，尾巴長出去的方向，好像不太對？

我們可以選擇用眼睛這類的感官檢查程式是否有問題，但是在貪食蛇這個程式中，Timer 每隔 0.5 秒就會觸發一次，蛇每隔 0.5 秒就會移動一格，如果剛吃到水果的時候，因為我們的程式邏輯有問題，尾巴長出去的方向不對，我們只有 0.5 秒的時間可以用肉眼捕捉這個問題，那，我們真的有辦法在程式出錯的時候，有效發覺問題嗎？你真的以為，光是貪食蛇這麼簡單的小遊戲，你就不會寫出 bug 嗎？

而假如我們寫的程式，其實並不是貪食蛇這種輕鬆的小遊戲呢？

AAA 原則

那，我們來試試看單元測試這條途徑。

在 Xcode 裡頭建立專案的時候，Xcode 會幫我們的 App 同時建立一個單元測試的 Bundle—其實蘋果也鼓勵你寫單元測試—在這個 Bundle 中，會出現一個繼承自 XCTestCase 的 class，在裡頭撰寫任何用 test 開頭的 method，像 `-testHit`，都是一條 test case。也就是，我們寫測試的時候，就是寫出一群用 test 為開頭的 method。

Xcode 從 5.1 版現在的測試框架叫做 XCTest，在這之前則是使用一套叫做 OCUnit 的測試框架（這邊的 OC 其實就是 Objective-C 的意思），除此之外，iOS 與 Mac OS X 平台上還有許多有名的測試框架，像是 GHUnit、Kiwi 等等。我們在這邊只先講解 XCUnit，如果你對單元測試有些心得，覺得 XCUnit 已經無法滿足你的需求，不妨也可以試試看其他的測試框架。

在撰寫測試的時候，基本原則是一次只測試一項 function 或 method，同時一個 test case 會包含所謂的 AAA 原則：Arrange、Act 與 Assert。

- Arrange: 先設定我們在這次測試中，所預期的結果
- Act: 就是我們想要測試的 function 或 method
- Assert: 確認在 Action 發生後，確認在執行了想要測試 function 或 method 後，的確符合我們在 Arrange 階段設定的目標

舉個例子，我們預期一條長度為 6、正在往左邊移動的蛇，在先往上走一格、再往右走一格、再往下走一格之後，這條蛇的頭一定會撞到自己的身體，如果我們的程式說蛇頭沒有撞到，就一定有 Bug。就可以拆解成：

- Arrange: 頭應該會撞到身體
- Action: 讓蛇執行往上右下移動的動作
- Assert: 確認頭真的撞到身體了

這個 case 或許會像這樣：

```
- (void)testHit
{
    KKSnake *snake = [[KKSnake alloc]
        initWithFrame:KKMakeSnakeWorldSize(10, 10) length:6];
    [snake changeDirection:KKSnakeDirectionUp];[[snake move];
    [snake changeDirection:KKSnakeDirectionRight];[snake move];
    [snake changeDirection:KKSnakeDirectionDown];[snake move];
    XCTAssertEqual([snake isHeadHitBody], YES, @"must hit the body.");
}
```

如果我們想要測試「蛇的尾巴加長」這段程式是否正常，大概會這麼寫：原本這條蛇的長度為 2，尾巴位在 (6, 5)，假如蛇的身體要加長兩格，我們預期蛇的長度變成 4，尾巴位在 (8, 5)。

```
- (void)testIncreaseLength
{
    ZBSnake *snake = [[ZBSnake alloc] initWithFrame:ZBMakeSnakeWorldSize(10, 10) length:2];
    XCTAssertEqual((int)[snake.points count], 2, @"Length must be 2 but %d", [snake.point
```

```
s count]);
    NSInteger x;
    NSInteger y;
    x = [snake.points[[snake.points count] - 1] snakePointValue].x;
    y = [snake.points[[snake.points count] - 1] snakePointValue].y;
    XCTAssertEqual(x, 6, @"must be 6");
    XCTAssertEqual(y, 5, @"must be 5");

    [snake increaseLength:2];
    XCTAssertEqual((int)[snake.points count], 4, @"Length must be 4 but %d", [snake.points count]);
    x = [snake.points[[snake.points count] - 1] snakePointValue].x;
    y = [snake.points[[snake.points count] - 1] snakePointValue].y;
    XCTAssertEqual(x, 8, @"must be 8");
    XCTAssertEqual(y, 5, @"must be 5");
}
```

執行測試

寫了測試程式之後，我們可以在 Xcode 裡頭按下 Product->Test 執行單元測試。如果XCTAssertEqual 這行 assert 出現問題，Xcode 就會立刻出現警告。

不同於之前，我們只能夠在 0.5 秒的時間內，用肉眼判斷我們的程式是不是有 bug，我們現在可以對我們寫出的貪食蛇更有信心：在 test case 之前，要不就是能通過，要不就是不能通過。當 test case 愈多，也就代表，我們的程式的確經得起考驗。

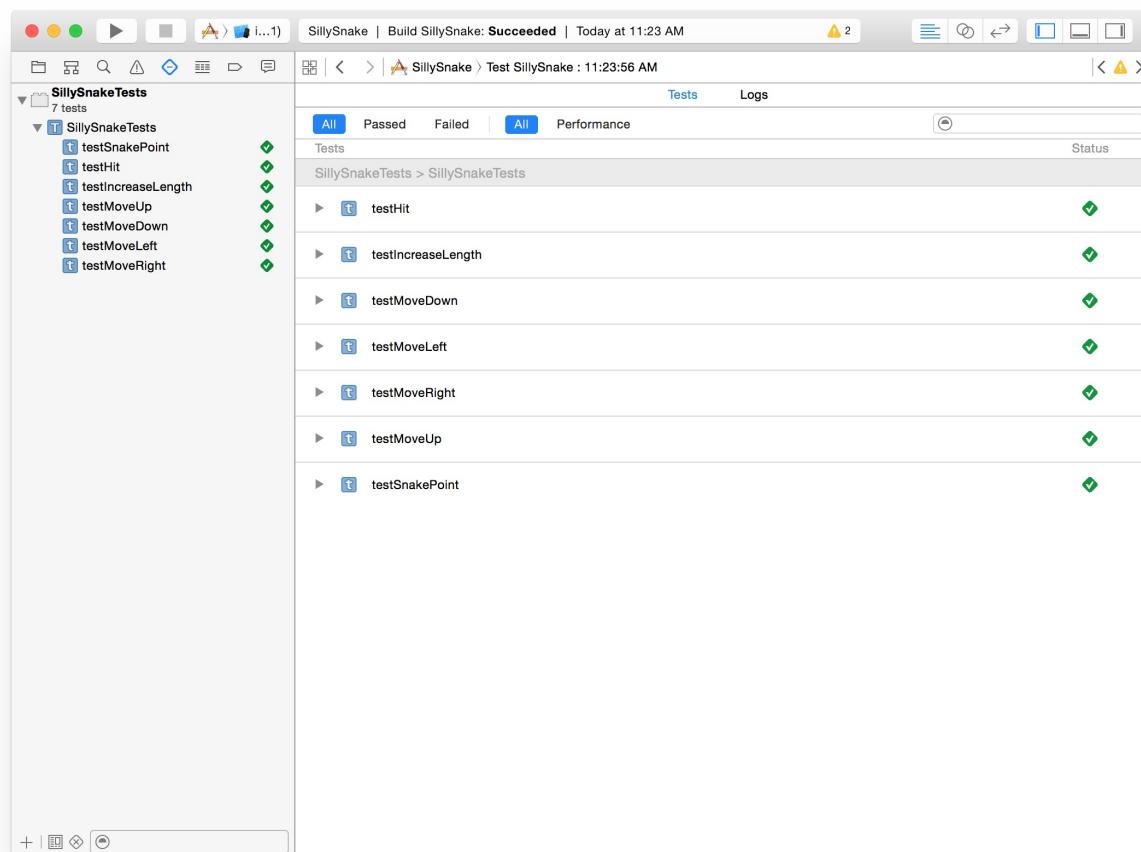
我們在 Xcode 裡頭有幾種不同的方式檢視單元測試的結果。在程式碼的編輯畫面中，每一個 test case 前方會出現一個菱形的圖示，如果這個圖示是空白的，代表還沒有執行測試，執行完畢之後，如果成功，就會是綠色，反之就會變成紅色。我們也可以直接用滑鼠按這個菱形圖示，執行 test case。

```

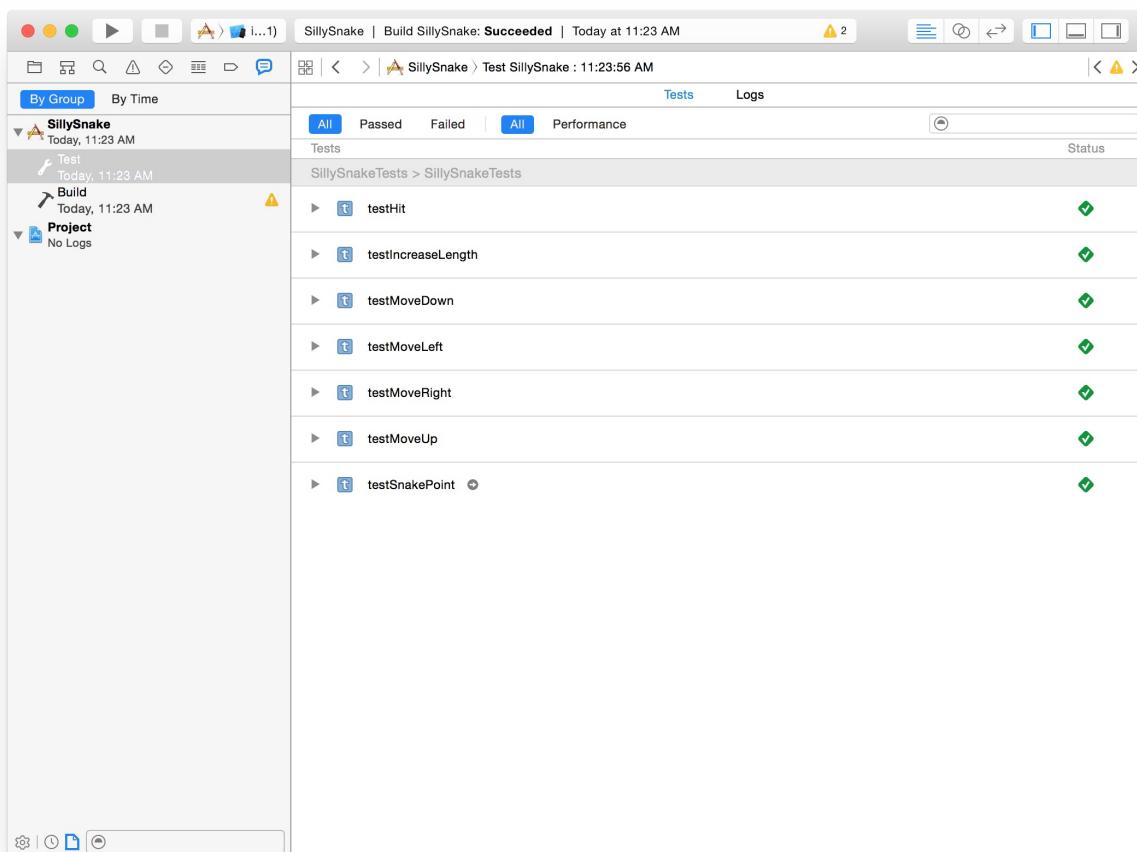
27
28 - (void)testHit
29 {
30     ZBSnake *snake = [[ZBSnake alloc] initWithWorldSize:ZBMakeSnakeWorldSize(10, 10)
31         length:6];
32     [snake changeDirection:ZBSnakeDirectionUp];
33     [snake move];
34     [snake changeDirection:ZBSnakeDirectionRight];
35     [snake move];
36     [snake changeDirection:ZBSnakeDirectionDown];
37     [snake move];
38     XCTAssertEqual([snake isHeadHitBody], YES, @"must hit the body.");
39 }
40 - (void)testIncreaseLength
41 {
42     ZBSnake *snake = [[ZBSnake alloc] initWithWorldSize:ZBMakeSnakeWorldSize(10, 10)
43         length:2];
44     XCTAssertEqual((int)[snake.points count], 2, @"Length must be 2 but %d", [snake.points
45         count]);
46     NSInteger x;
47     NSInteger y;
48     x = [snake.points[[snake.points count] - 1] snakePointValue].x;
49     y = [snake.points[[snake.points count] - 1] snakePointValue].y;
50     XCTAssertEqual(x, 6, @"must be 6");
51     XCTAssertEqual(y, 5, @"must be 5");
52     [snake increaseLength:2];
53     XCTAssertEqual((int)[snake.points count], 4, @"Length must be 4 but %d", [snake.points
54         count]);
55     x = [snake.points[[snake.points count] - 1] snakePointValue].x;
56     y = [snake.points[[snake.points count] - 1] snakePointValue].y;
57     XCTAssertEqual(x, 8, @"must be 8");
58     XCTAssertEqual(y, 5, @"must be 5");
59     [snake increaseLength:2];
60     XCTAssertEqual((int)[snake.points count], 6, @"Length must be 6 but %d", [snake.points
61         count]);
62     x = [snake.points[[snake.points count] - 1] snakePointValue].x;
63     y = [snake.points[[snake.points count] - 1] snakePointValue].y;
64     XCTAssertEqual(x, 0, @"must be 0");
65 }

```

在 Xcode 的左方側邊欄的第四項，叫做 Test Navigator，在這邊我們可以找到我們目前所在專案的所有 test case，在這邊可以看到每個 test case 是成功或失敗，也可以透過點擊，直接跳到特定 test case 的程式碼。



在 Xcode 的左方側邊欄的最後一項，叫做 Report Navigator，在這邊可以看到最近一次完整執行所有 test case 的結果。



測試驅動開發

軟體業界開始注重單元測試，大概是進入二十一世紀的頭幾年的事情，其中一個明顯的指標就是 Kent Beck 在 2002 年出版的 *Test-Driven Development: By Example* 這本書，提出了「測試驅動開發」（Test-Driven Development，通常簡稱 TDD）的觀念，顛覆了不少人寫程式的習慣與流程：在開發軟體的時候，我們不是先寫主要功能，而是先寫測試。

坊間跟測試驅動開發相關的書籍或其他閱讀資料非常多，我們不會在這邊花費太多篇幅。簡單來說，Kent Beck 提出寫程式的過程，應該是 "Red, Green, Refactor" 三個階段：

- Red: 在還沒有主要功能之前，先寫單元測試。由於主要功能都還沒有撰寫，自然無法通過剛剛寫出來的單元測試，所以會亮出紅色的燈號。
- Green: 開始實作主要功能，直到可以通過單元測試，讓測試的燈號變成綠色。
- Refactor: 繼續整理寫出的程式碼。

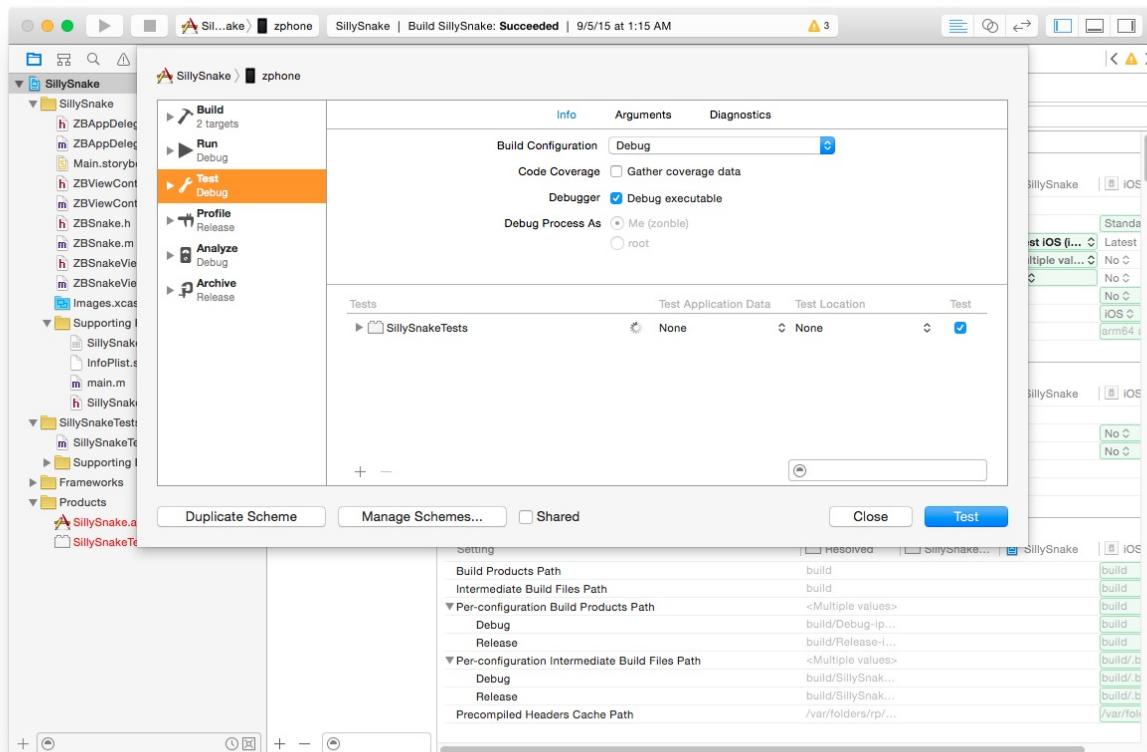
這個流程後來引起一些批評，比較有名的案例像是 Ruby on Rail 的主要作者 DHH (David Heinemeier Hansson) 在個人 blog 上寫了一篇 [TDD is dead. Long live testing.](#)，反對 TDD 流程，主要論點是，如果非要讓系統中每個部分都要能夠被執行單元測試，反而會導致不適當、甚至有害的系統設計。此文一出，後續引發了一連串的爭論。至於在你的開發流程中，是否適合使用測試驅動開發呢？還是那句話：先別排斥，先試試看吧！

覆蓋率 (Coverage)

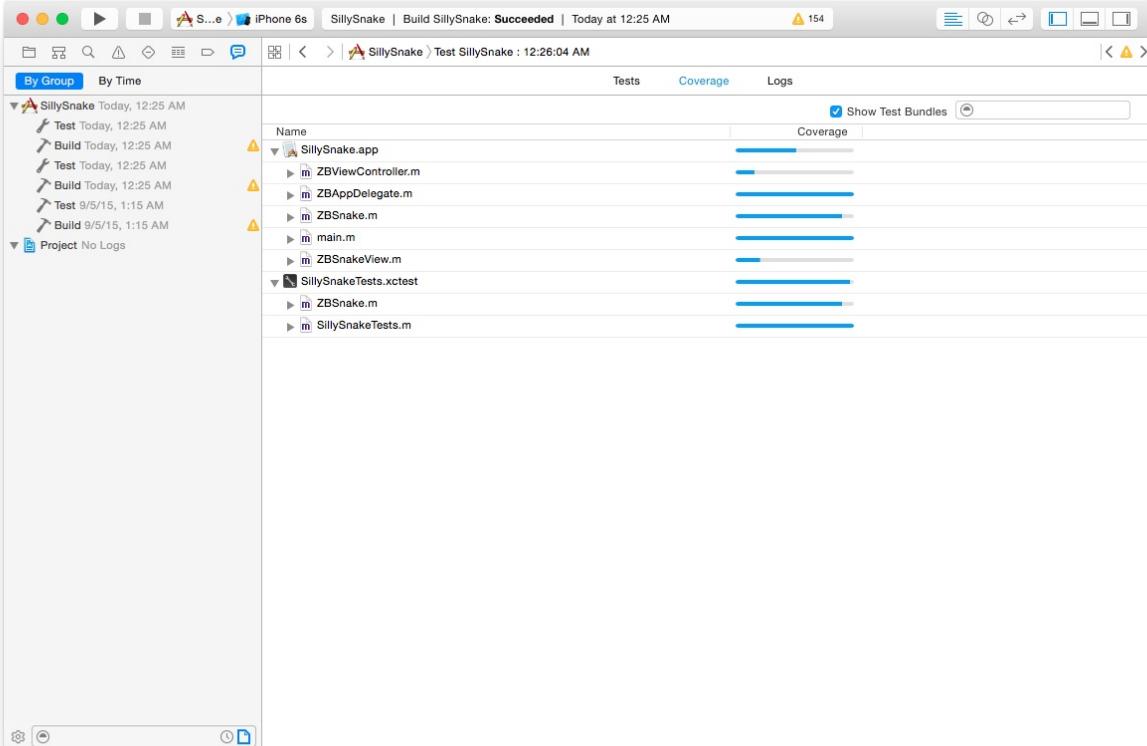
寫了單元測試之後，我們下一步就會想要計算覆蓋率。所謂覆蓋率就是我們的單元測試覆蓋了程式的多少比例，也就是，有多少程式被測試到、以及沒有被測試到。當我們發現有程式沒有被測試到之後，便進一步撰寫更多的 test case，確保我們的程式經過完整測試。

Xcode 7

Xcode 7 當中直接包含計算覆蓋率的功能。要在 Xcode 7 中顯示覆蓋率，首先是在 Scheme 中，勾選 "Gather Coverage Data"。



接著，在執行單元測試的時候，就可以看到有一個顯示 Coverage 的分頁，顯示每個檔案的覆蓋率是多少。



選擇任一檔案編輯，便可以看到在畫面的最右方，可以看到每行程式在 test case 中被執行了幾次，如果沒有執行到（執行次數為 0） ，背景就會變成紅色，提醒我們應該要對這部份寫單元測試。

```

iPhone 6s | SillySnake | Build SillySnake: Succeeded | Today at 12:25 AM
By Group By Time
SillySnake Today, 12:25 AM
  ↗ Test Today, 12:25 AM
  ↗ Build Today, 12:25 AM
  ↗ Test Today, 12:25 AM
  ↗ Build Today, 12:25 AM
  ↗ Test 9/5/15, 1:15 AM
  ↗ Build 9/5/15, 1:15 AM
Project No Logs

82 }
83 [points insertObject:[NSValue valueWithSnakePoint:ZBMakeSnakePoint(x, y)] atIndex:0];
84 }

85 - (BOOL)changeDirection:(ZBSnakeDirection)inDirection
86 {
87     if (directionLocked) {
88         return NO;
89     }
90     if (inDirection == ZBSnakeDirectionLeft || inDirection == ZBSnakeDirectionRight) {
91         if (direction == ZBSnakeDirectionUp || direction == ZBSnakeDirectionDown) {
92             direction = inDirection;
93             return YES;
94         }
95     }
96     if (inDirection == ZBSnakeDirectionUp || inDirection == ZBSnakeDirectionDown) {
97         if (direction == ZBSnakeDirectionLeft || direction == ZBSnakeDirectionRight) {
98             direction = inDirection;
99             return YES;
100        }
101    }
102 }
103 return NO;
104 }

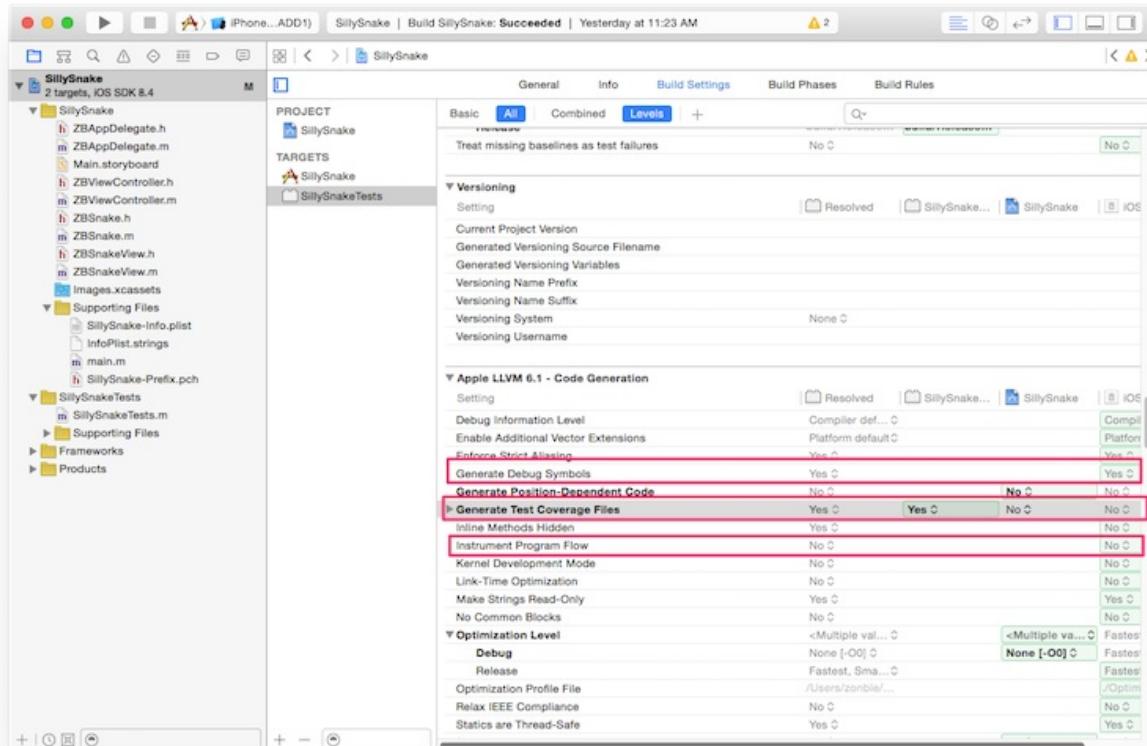
105 - (void)increaseLength:(NSUInteger)inLength
106 {
107     ZBSnakePoint lastPoint = [[points lastObject] snakePointValue];
108     ZBSnakePoint theOneBeforeLastPoint = [[points objectAtIndex:points count]-2] snakePointValue;
109     NSInteger x = lastPoint.x - theOneBeforeLastPoint.x;
110     NSInteger y = lastPoint.y - theOneBeforeLastPoint.y;
111     if (lastPoint.x == 0 && theOneBeforeLastPoint.x == worldSize.width - 1) x = 1;
112     if (lastPoint.x == worldSize.width - 1 && theOneBeforeLastPoint.x == 0) x = -1;
113     if (lastPoint.y == 0 && theOneBeforeLastPoint.y == worldSize.height - 1) y = 1;
114     if (lastPoint.y == worldSize.height - 1 && theOneBeforeLastPoint.y == 0) y = -1;
115
116     for (NSInteger i = 0; i < inLength; i++) {
117         NSInteger theX = (lastPoint.x + x * (i + 1)) % worldSize.width;
118         NSInteger theY = (lastPoint.y + y * (i + 1)) % worldSize.height;
119         [points addObject:[NSValue valueWithSnakePoint:ZBMakeSnakePoint(theX, theY)]];
120     }
121 }

```

要讓 Xcode 6 幫我們產生覆蓋率報告，我們要調整一下 Xcode 裡頭的專案設定，將 Build Setting 下 Apple LLVM 區段的三項設定設成 YES。分別是：

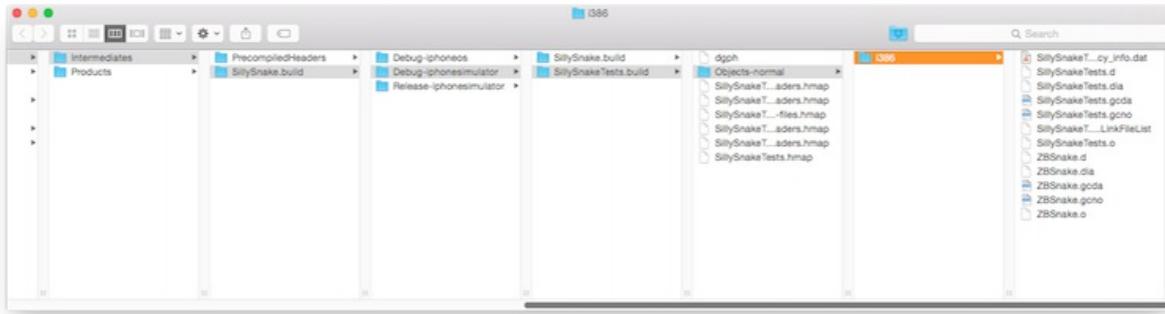
- Generate Debug Symbol
- Generate Test Coverage File
- Instrument Program Flow

這三項設定的位置如下圖：

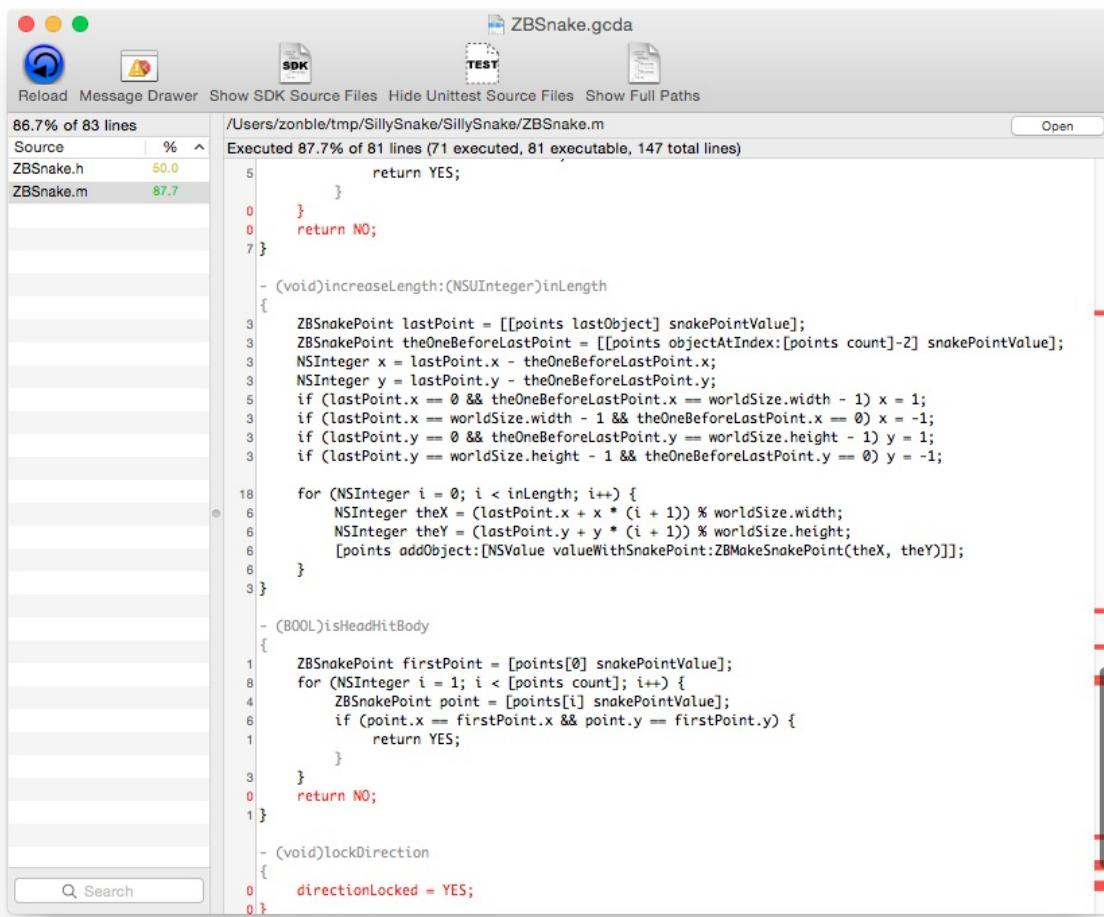


Xcode 產生出來的覆蓋率報告格式為 .gcno 檔案，但檔案產生出來的位置預設會在不是很容易找到的地方。這個檔案會出現在編譯專案時產生的 Intermediates 目錄下，如果你沒有改過 Xcode 偏好設定中 Locations -> Derived Data 的目錄設定的話，並且使用 iOS 模擬器執行測試的話，要尋找 [.gcno 檔案](#)，位置大概是：

- 首先進入你個人目錄下的 ~/Library/Developer/Xcode/DerivedData 目錄
- 繼續進入 <專案名稱>-<一串亂碼>/Build/Intermediates/<專案名稱>.build/Debug-iphonesimulator/<專案名稱>Tests.build/Objects-normal/i386 目錄。比方說，我的專案名稱叫做 SillySnake，我就在 ~/Library/Developer/Xcode/DerivedData/SillySnake-dccqebabizkqeabeylczwkqwwbj/Build/Intermediates/SillySnake.build/Debug-iphonesimulator/SillySnakeTests.build/Objects-normal/i386 目錄下，找到了對應的檔案



找到對應的檔案之後，我們可以用 [gcov](#) 這個 command line 工具開啟、閱讀 .gcno 檔案，不過，如果有 GUI 工具那就更理想了；我們可以用 [coverstory](#) 這套 open source 的工具瀏覽 .gcno 檔案。



在畫面的左方，會列出這次測試中，測試了哪些檔案，以及每個檔案的測試覆蓋率。在畫面的右方則是程式碼，有測試到的程式會以黑色顯示，沒有測試到的程式則會有紅色顯示。

相關閱讀

- [Testing with Xcode](#)
- [Technical Q&A QA1514 Configuring Xcode for Code Coverage](#)
- [NSHipster - Unit Testing](#)

Blocks

蘋果在 Mac OS X 10.6 與 iOS 4 之後導入 Block 語法，之後就大幅改變了撰寫 Objective-C 語言的方式。

Block 是 Cocoa/Cocoa Touch Framework 中的匿名函式（Anonymous Functions）的實作。所謂的匿名函式，就是一段 **具有物件性質的程式碼**，這一段程式碼可以當做函式執行，另一方面，又可以當做物件傳遞；因為可以當做物件傳遞，所以可以讓某段程式碼變成是某個物件的某個 property，或是當做 method 或是 function 的參數傳遞，就是因為這種特性，造成最常使用 block 的時機，就是拿 block 實作 callback。

在有 block 之前，在 Cocoa/Cocoa Touch Framework 上要處理 callback，最常見的就是使用 delegate（此外也可以使用比較具有 C 語言風格的方式，傳遞 callback function 的 pointer，或是使用 target/action pattern）。在 iOS 4 有了 block 之後，可以看到蘋果自己便大幅改寫了 UIKit 等 Framework 的 API，把原本使用 delegate 處理 callback 的地方，都大幅換成了 block。

Block 的語法

一直以來還是有不少人不滿 block 語法，甚至有人成立了一個叫做 fuckingblocksyntax.com 的網站。這個網站的網域名稱不怎麼優雅，不過裡頭倒是清楚整理了我們應該如何宣告 block。

將 block 宣告成變數（local variable）的語法是：

```
returnType (^blockName)(parameterTypes) = ^returnType(parameters) {...};
```

宣告成 Objective-C property 的語法是：

```
@property (nonatomic, copy) returnType (^blockName)(parameterTypes);
```

宣告成 method 的參數（method parameter）的語法是：

```
- (void)someMethodThatTakesABlock:(returnType (^)(parameterTypes))blockName;
```

在執行某個需要傳入 block 當做參數的 method 的時候，則是用這以下方式呼叫。這也是絕大多數用 block 當做 callback 的處理方式：

```
[someObject someMethodThatTakesABlock:^returnType (parameters) {...}];
```

把一種 block 宣告成 typedef：

```
typedef returnType (^TypeName)(parameterTypes);
TypeName blockName = ^returnType(parameters) {...};
```

Block 也可以當成 C function 的參數或是回傳結果的型別，但是，在這種狀況下，我們不能夠直接使用 returnType (^)(parameterTypes) 這種語法，必須要先宣告成 typedef 才行。也就是說，這樣會被當成不合規：

```
(void (^)(void)) test((void (^)(void)) block) {
    return block;
}
```

但可以寫成這樣：

```
typedef void (^TestBlock)(void);
TestBlock test(TestBlock block) {
    return block;
}
```

雖然 C function 的參數不能夠使用 returnType (^)(parameterTypes) 語法，但是一個 block 倒是可以使用這種語法撰寫輸入與回傳值的型別，但其實在這種狀況下，還是比較建議使用 typedef 告宣。比方說，我們現在要宣告一個 block，這個 block 會回傳另外一個型別為 int(^)(void) 的 block，就會寫成這樣：

```
int (^(^counter_maker)(void))(void) = ^ {
    __block int x = 0;
    return ^ {
        return ++x;
    };
};
```

可讀性實在非常差。不如寫成這樣：

```
typedef int (^CounterMakerBlock)(void);
CounterMakerBlock (^counter_maker)(void) = ^ {
    __block int x = 0;
    return ^ {
        return ++x;
    };
};
```

Block 如何代替了 Delegate

要了解在哪些場合使用 block，我們不妨先看一下蘋果自己如何使用 block。

首先是 UIView 動畫。當我們在某個 view 上面改變了某些 subview 的位置與大小，或是改變了這個 view 的一些屬性，像是背景顏色等，一般來說不會產生動畫效果，我們的改動會直接生效，但是我們也可以產生一段動畫效果，這種動畫我們稱為 UIView Animation。（UIView Animation 其實底層是透過 Core Animation 完成的，但我們稍晚才會討論 Core Animation。）

像是，我們想要改動某個 subview 的 frame：

```
self.subview.frame = CGRectMake(10, 10, 100, 100);
```

在 iOS 4 之前，我們會使用 UIView 的 +beginAnimations:context: 與 +commitAnimations 兩個 class method，把原本的 code 包起來，那麼，在這兩個 class method 之間的程式碼就會產生動畫效果。

```
[UIView beginAnimations:@"animation" context:nil];
self.subview.frame = CGRectMake(10, 10, 100, 100);
[UIView commitAnimations];
```

如果我們想要在這段動畫結束的時候做一件事情，像是執行另外一個動畫，我們應該怎麼做呢？iOS 4 之前唯一的方法就是透過 UIView 的 delegate，我們在執行動畫之前，需要先設定好 delegate，以及要執行 delegate 上的哪個 selector。像是：

```
- (void)moveView
{
    [UIView beginAnimations:@"animation" context:nil];
    [UIView setAnimationDelegate:self];
    [UIView setAnimationDidStopSelector:
        @selector(animationDidStop:finished:context:)];
    self.subview.frame = CGRectMake(10, 10, 100, 100);
    [UIView commitAnimations];
}

- (void)animationDidStop:(NSString *)animationID
    finished:(NSNumber *)finished
    context:(void *)context
{
    // do something
}
```

可以看到，如果使用 delegate pattern，一段連續的流程，會分散在很多不同的 method 中。有了 block 語法之後，我們可以將「動畫該做什麼」與「動畫完成之後要做什麼」，寫成一段集中的程式碼。像是：

```
- (void)moveView
{
    [UIView animateWithDuration:0.25 animations:^{
        self.subview.frame = CGRectMake(10, 10, 100, 100);
    }];
}
```

```
    } completion:^(BOOL finished) {
        // Do something
    }];
}
```

另外像 NSArray 裡頭的物件排序，以往我們必須以 C function pointer 或是 selector 形式傳入用來比較物件大小的 comparator。像是：

```
NSArray *array = @[@1, @2, @3];
NSArray *sortedArray = [array sortedArrayUsingSelector:@selector(compare:)];
```

也可以改用 block 語法：

```
NSArray *array = @[@1, @2, @3];
NSArray *sortedArray = [array sortedArrayUsingComparator:
    ^NSComparisonResult(id obj1, id obj2) {
        return [obj1 compare:obj2];
}];
```

Block 與 Delegate 都可以想成是 Event Handler

我們在前一章當中提到，Objective-C 裡頭的 delegate 其實相當於 C# 裡頭 event handler 的角色，但 C# 裡頭的 event handler 在語法上會更接近 Objective-C 的 block，都是匿名函式。

我們可以來看一個 [來自 MSDN 的例子.aspx](#)。以下的 C# 程式中，有一個叫做 `changed` 的 event handler。

```
using System;
namespace MyCollections
{
    using System.Collections;

    // A delegate type for hooking up change notifications.
    public delegate void ChangedEventHandler(object sender, EventArgs e);

    // A class that works just like ArrayList, but sends event
    // notifications whenever the list changes.
    public class ListWithChangedEvent: ArrayList
    {
        // An event that clients can use to be notified whenever the
        // elements of the list change.
        public event ChangedEventHandler Changed;

        // Invoke the Changed event; called whenever list changes
        protected virtual void OnChanged(EventArgs e)
        {
            if (Changed != null)
                Changed(this, e);
        }

        // Override some of the methods that can change the list;
        // invoke event after each
        public override int Add(object value)
        {
            int i = base.Add(value);
            OnChanged(EventArgs.Empty);
            return i;
        }

        public override void Clear()
        {
            base.Clear();
            OnChanged(EventArgs.Empty);
        }
    }
}
```

如果寫成 Objective-C 並且使用 delegate，會像這樣：

```
@class ListWithChangedEvent
@protocol ListWithChangedEventDelegate <NSObject>
- (void)listDidChange:(ListWithChangedEvent *)list args:(EventArgs *)e;
@end

@interface ListWithChangedEvent : NSObject
@property (weak, nonatomic) id <ListWithChangedEventDelegate> delegate;
@end

@implementation ListWithChangedEvent

- (void)onChanged:(EventArgs *)e
{
    [self.delegate listDidChange:self args:e];
}

- (void)add:(id)value
{
    [super add:value];
    [self onChanged:nil];
}

- (void)clear:(id)value
{
    [super clear:value];
    [self onChanged:nil];
}

@end
```

那如果是 block 呢？可以發現，會更接近 C# 裡頭的寫法了。

```
typedef void (^changedEventHandler)(id, EventArgs *);

@interface ListWithChangedEvent : NSObject
@property (copy, nonatomic) changedEventHandler changed;
@end

@implementation ListWithChangedEvent

- (void)onChanged:(EventArgs *)e
{
    if (self.changed) {
        self.changed(self, e);
    }
}

- (void)add:(id)value
{
    [super add:value];
    [self onChanged:nil];
}
```

```
- (void)clear:(id)value
{
    [super clear:value];
    [self onChanged:nil];
}

@end
```

還是順道一提，在 C# 裡頭，我們可能會把所有的 callback 都叫做 event，但是在 Cocoa 與 Cocoa Touch 的世界裡頭，我們只會把來自硬體的輸入叫做 event。

由於 block 的主要用途，就在於處理 callback，所以 block 經常就用在各種非同步工作的 callback 上，要執行各種非同步的工作，就往往要使用 thread，關於在 iOS 與 Mac 上如何使用 thread，我們將會在 [Threading](#) 這章中討論。

什麼時候該用 Blocks？什麼時候該用 Delegate？

即使 block 可以大幅度取代 delegate 處理 callback，但是從蘋果自己的 API 設計中可以看到，並不是所有的 delegate 都被 block 取代，在 Cocoa 與 Cocoa Touch framework 中，仍然大幅度使用 delegate。那麼，我們就要問：當我們在設計 API 的時候，什麼狀況下應該使用 block？什麼時候又該使用 delegate？

通常的區分方式是：如果一個 method 或 function 的呼叫只有單一的 callback，那麼就使用 block，如果可能會有多個不同的 callback，那麼就使用 delegate。

這麼做的好處是：當一個 method 或 function 呼叫會有多種 callback 的時候，很有可能某些 callback 是沒有必要實作的。

如果使用 delegate 實作，那麼，在 delegate 需要實作的 protocol 中，我們可以用 `@required` 與 `@optional` 關鍵字區分哪些是一定需要實作的 delegate method。

但相對的，用 block 處理 callback，就會很難區分某個 block 是否是必須要實作：在 Xcode 6.3 之前，Objective-C 並沒有 `nullable`、`nonnull` 等關鍵字，讓我們知道某個 property、或某個 method 要傳入的 block 可不可以是 nil，我們也往往搞不清楚在這些地方傳入 nil，會不會發生什麼危險的事情。¹

舉個例子。在 iOS 7 之後，蘋果鼓勵開發者使用 NSURLSession 處理網路連線，NSURLSession 就充分表現了「單一 callback 用 block、多重 callback 用 delegate」這一點。

假如我們現在想要把 KKBOX 的官網首頁抓下來，我們只要建立一個 NSURLSessionDataTask 物件，一般來說，我們只需要處理「這個連線做完事情的下一步該做什麼」，所以一般也需要實作這個 task 的 completion handler，就是傳入網路連線結束之後要執行的 block；一般連線結束，大概就是成功抓到資料或是連線失敗兩種狀況，所以我們可以透過 data 與 error 這兩物件判斷是哪種狀況：失敗的話，error 就不會是 nil，我們就要處理 error，反之就要處理 data。

```
NSURL *URL = [NSURL URLWithString:@"http://kkbox.com"];
NSURLRequest *request = [NSURLRequest requestWithURL:URL];
NSURLSessionDataTask *task = [[NSURLSession sharedSession]
    dataTaskWithRequest:request
    completionHandler:^(NSData *data, NSURLResponse *response, NSError *error) {
        if (error) {
            // handle error
            return;
        }
        // handle data
    }];
[task resume];
```

但，NSURLSession 本身也還是具有 delegate。我們在發送連線的時候，除了處理連線結束要做什麼之外，有時候也可能會想處理在連線中途所發生的其他狀況，像是：HTTP 連線收到 302 轉址、遇到有問題的 SSL 憑證、server 要求用戶輸入帳號密碼，這些狀況我們要不要提示使用者？或，如果這是一個傳遞大檔、很花時間的連線，我們有沒有必要顯示連線進度條？這些狀況還是會傳遞給 NSURLSession 的 delegate，而如果我們要處理這些狀況，就要實作以下這些 delegate methods。

```
- (void)URLSession:(NSURLSession *)session task:(NSURLSessionTask *)task
    willPerformHTTPRedirection:(NSHTTPURLResponse *)response
```

```
newRequest:(NSURLSessionRequest *)request  
completionHandler:(void (^)(NSURLSessionRequest *))completionHandler;  
- (void)URLSession:(NSURLSession *)session task:(NSURLSessionTask *)task  
didReceiveChallenge:(NSURLAuthenticationChallenge *)challenge  
completionHandler:(void (^)(NSURLSessionAuthChallengeDisposition disposition, NSURLCredential *credential))completionHandler;  
- (void)URLSession:(NSURLSession *)session task:(NSURLSessionTask *)task  
needNewBodyStream:(void (^)(NSInputStream *bodyStream))completionHandler;  
- (void)URLSession:(NSURLSession *)session task:(NSURLSessionTask *)task  
didSendBodyData:(int64_t)bytesSent  
totalBytesSent:(int64_t)totalBytesSent  
totalBytesExpectedToSend:(int64_t)totalBytesExpectedToSend;
```

¹. 實際是把 Swift 的語言特性移植回 Objective-C。 ↩

__block 關鍵字

在一個 block 裡頭如果使用了在 block 之外的變數，會將這份變數先複製一份再使用，也就是說，在沒有特別宣告的狀況下，對我們目前所在的 block 來說，所有外部的變數都是唯讀，只能讀取，不能變更。至於 block 裡頭用到的 Objective-C 物件，則都會被多 retain 一次。

如果我們想要讓某個 block 可以改動某個外部的變數，我們就要在這個需要可以被 block 改動的變數前面，加上 __block 關鍵字。

像這樣是不合法的程式：

```
int i = 1;
void (^block)(void) = ^{
    i = i + 1;
};
```

應該寫成：

```
__block int i = 1;
void (^block)(void) = ^{
    i = i + 1;
};
```

__weak 關鍵字

在使用了 block 之後，記憶體管理會變得非常複雜，所以最好是在開啟了 ARC 自動記憶體管理之後再使用 block。不過，即使開啟了 ARC，還是可能會遇到循環 retain 的問題。

由於 block 中用到的 Objective-C 物件都會被多 retain 一次，這邊所指的 Objective-C 物件也包含 self，所以，假使有個物件的 property 是一個 block，而這個 block 裡頭又用到了 self，就會遇到循環 retain 而無法釋放記憶體的問題：self 要被釋放才會去釋放這個 property，但是這個 property 作為 block 又 retain 了 self 導致 self 無法被釋放。

下面這段 code 就有循環 retain 的問題：

```
@interface MyClass : NSObject
- (void)doSomthing;
@property (copy, nonatomic) void (^myBlock)(void);
@end

@implementation MyClass

- (instancetype)init
{
    self = [super init];
    if (self) {
        self.myBlock = ^{
            [self doSomthing];
        };
    }
}
```

```
    };
}
return self;
}
- (void)doSomthing
{
}
@end
```

如果我們不想讓 self 被 myBlock 細 retain 起來，我們就要把 self 變成 weak reference 再傳入到 block 中。像是改成這樣：

```
__weak MyClass *weakSelf = self;
self.myBlock = ^{
    [weakSelf doSomthing];
};
```

Block 作為 Objective-C 物件

前面提到，Block 其實可以當成是一種物件，我們接下來就要來看 block 的作為物件的特性。

記憶體管理

由於 Objective-C 物件需要做記憶體管理，因此，如果你還在手動管理記憶體，在建立 block 的時候，也必須手動管理 Block 的記憶體。我們可以使用 `Block_Copy()` 和 `Block_Release()` 這兩個 C Function 對 block 做 copy 或 release；或，我們也可以把 block 給 cast 成 id 型別，便可以對 block 呼叫 copy 與 release。不過在啟用 ARC 之後，我們便不需要，Compiler 也會阻止我們手動管理記憶體。

Block 的型別

Objective-C 當中每個物件都具有 class，而每個 class 都繼承自 `NSObject`，由於 block 具有物件的性質，因此 block 本身也有 class—不過，一個 block 是屬於哪一種 class 平時對我們來說並不會有太大的意義，畢竟我們在建立 block 的時候，並不會指定要建立哪一種 class 的 block，我們也不會去 subclass 某種 block 的 class。基本上，當我們寫好一個 block 之後，這個 block 最後會變成哪個 class，全部都是由 compiler 決定。

在 C 語言當中，記憶體分成三塊：global、stack 與 heap，compiler 在編譯程式碼的時候，會根據我們所寫出來的 block 到底使用到哪一塊記憶體，將這個 block 變成不同的 class，包括 `_NSGlobalBlock_`、`_NSStackBlock_` 與 `_NSMallocBlock_`。知道這件事情通常對我們不會有什麼幫助，但是可以讓我們了解蘋果的 compiler 曾經發生過的 bug：在某個狀況下，有個block 應該要使用 global 的記憶體，但是 compiler 却誤判成只使用 stack 的記憶體。

如果沒有開啟 ARC，以下這段程式碼會在執行到 `block()` 這一行的時候，發生 Bad Access 錯誤：

```
- (NSArray *)blocks
{
    int i = 1;
    return @[^{return i;}];
}

- (void)callBlock
{
    int (^block)(void) = [self blocks][0];
    block();
}
```

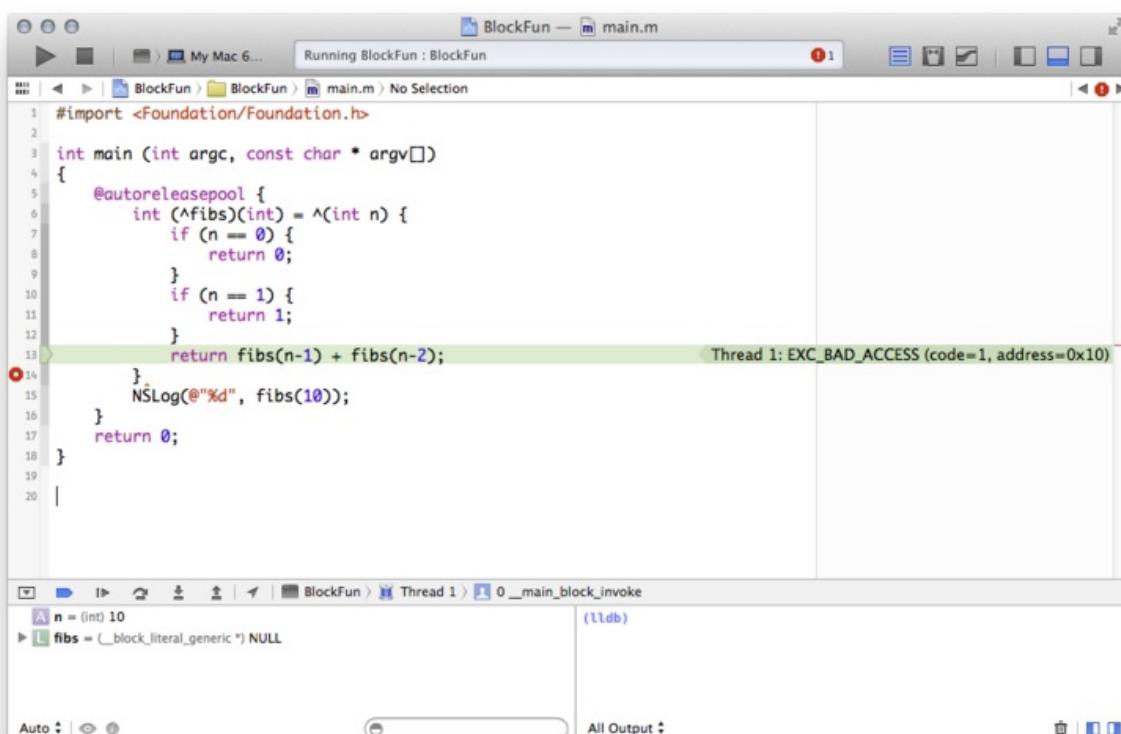
原因是：在 `-blocks` 所回傳的 `NSArray` 中所包含的 block 物件中，使用到了 `i` 這個只出現在 `blocks` 這個 method 內部的 `int` 變數，因為這個變數只在這個 method 中使用，compiler 便認為 `i` 應該使用 stack 的記憶體，因此也把回傳的 block 建立成 `_NSMallocBlock_`；於是，當我們在 `-callBlock` 裡頭呼叫 `block()` 的時候，原本的記憶體已經被釋放，於是產生記憶體管理錯誤。

哪些事情不要拿 Block 來做

在很多狀況下，使用 block 相當方便，但由於因為 block 的記憶體管理問題，有些事情使用 block 反而相當痛苦，就我個人而言，最痛苦的經驗應該就是拿 block 寫遞迴。舉個例子，假如要使用 block 來寫一個費式數列，可能會寫成這樣（這邊是開啟 ARC 的環境）：

```
int (^fibs)(int) = ^(int n) {
    if (n == 0) {
        return 0;
    }
    if (n == 1) {
        return 1;
    }
    return fibs(n-1) + fibs(n-2);
};
```

看起來好像沒有問題，但是一執行就會馬上 crash。



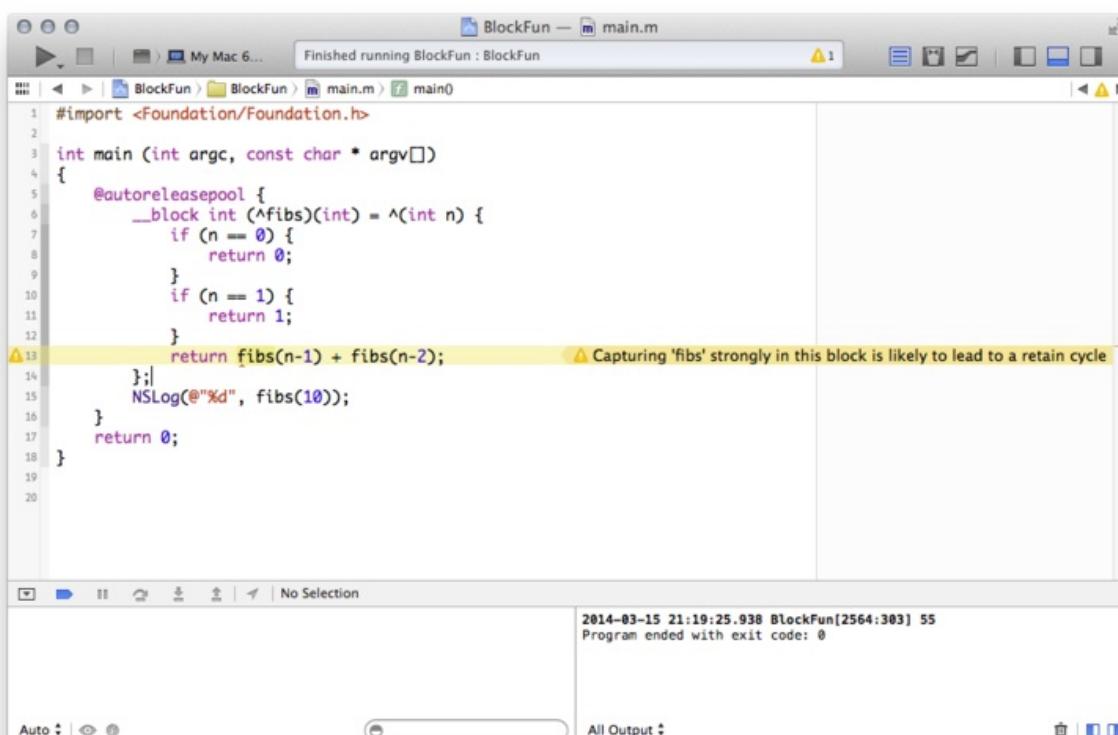
原因是在 fibs 這個 Block 的 scope 裡頭，fibs 這個變數被指向 NULL。一般來說，對 nil 物件做任何 Objective-C 呼叫都沒事，但是如果一個 Block 變數指向 NULL，一呼叫就會因為 Bad Access 錯誤而 crash。而當你寫出這段一定會 crash 的程式，compiler 也都不會發出警告...誰說用了 ARC 就不會有記憶體管理問題呢？

那麼，我們在 fibs 前面加上個 __block 看看？

```
__block int (^fibs)(int) = ^(int n) {
```

```
if (n == 0) {
    return 0;
}
if (n == 1) {
    return 1;
}
return fibs(n-1) + fibs(n-2);
};
```

結果，這樣的程式會因為循環 retain 而造成記憶體漏水。



所以這段 code 要寫成這樣才不會有問題：

```
typedef int(^fibsblock)(int);
__weak __block fibsblock fibs;
fibsblock fibs_;

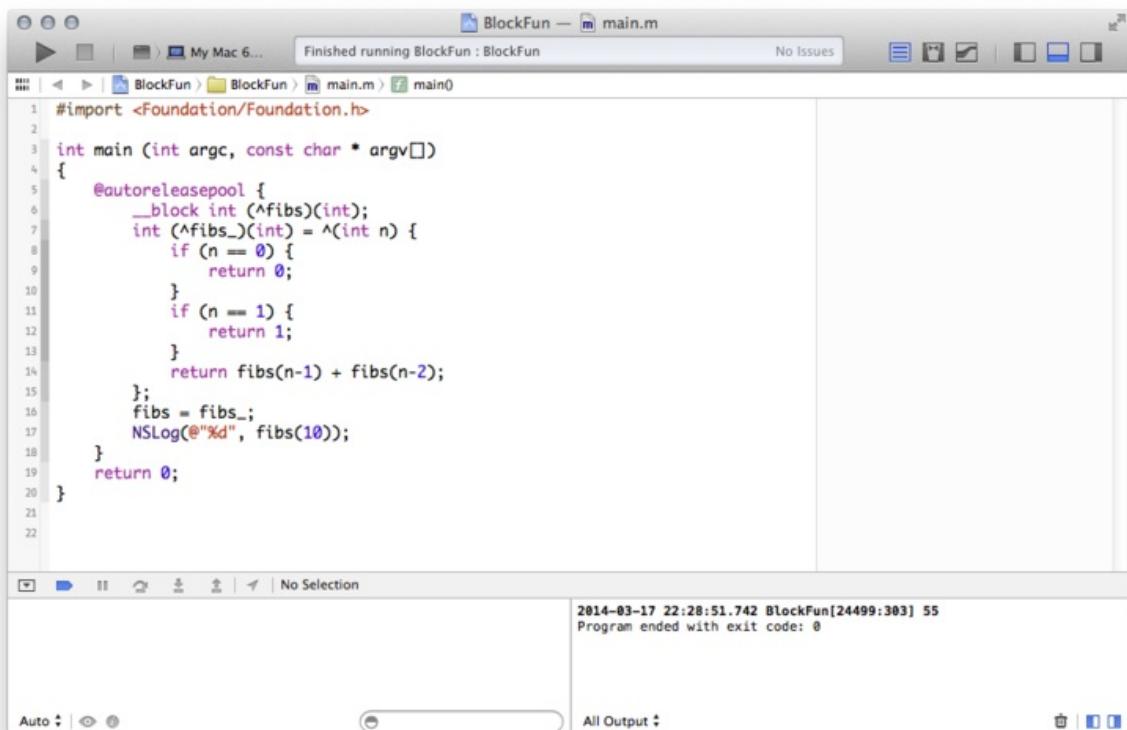
fibs_ = ^int(int n){
    if (n == 0) {
        return 0;
    }

    if (n == 1) {
        return 1;
    }

    return fibs(n-1) + fibs(n - 2);
}
```

```
};

fibs = fibs_;
```



The screenshot shows the Xcode IDE with a project named "BlockFun". The main.m file contains the following code:

```
1 #import <Foundation/Foundation.h>
2
3 int main (int argc, const char * argv[])
4 {
5     @autoreleasepool {
6         __block int (^fibs)(int);
7         int (^fibs_)(int) = ^(int n) {
8             if (n == 0) {
9                 return 0;
10            }
11            if (n == 1) {
12                return 1;
13            }
14            return fibs(n-1) + fibs(n-2);
15        };
16        fibs = fibs_;
17        NSLog(@"%@", fibs(10));
18    }
19    return 0;
20 }
```

The output window at the bottom right shows the results of the NSLog call:

```
2014-03-17 22:28:51.742 BlockFun[24499:303] 55
Program ended with exit code: 0
```

Callback Hell

Block 通常用在處理 callback，而我們往往會在 callback 裡頭又做另外一件事情，而這件事情又可能又有另外一個 callback，而這個 callback 裡頭要做的事情又有另外一個 callback...於是，我們可能會寫出這種深度非常深的程式碼：

這種狀況我們稱之為 Callback Hell—無限延長的 Callback 地獄，這種現象除了會出現在 Objective-C 的 block 之外，也出現在各式各樣的程式語言中，尤其是在 JavaScript 開發中，特別常討論 Callback Hell。為了要解決Callback Hell，從 ECMAScript 6 開始，就改成使用 Promise 的寫法處理Callback，而像是 Parse 的 [Bolts Framework](#)，便是將 Promise 從JavaScript port 到 Objective-C 中。

使用了 Botls Framework 之後，我們可以將各種要非同步執行的工作，包裝成 BFTask 物件，我們便可以將上面那段 code，改寫成這個樣子：

```
[[[[someObject doSomething] continueWithBlock:^id(BFTask *task) {
    return [someObject doSomething];
}]] continueWithBlock:^id(BFTask *task) {
    return [someObject doSomething];
}]] continueWithBlock:^id(BFTask *task) {
    return nil;
}];
```

雖然一樣是 callback 之間不斷串連，但是在程式碼中，我們把 callback 之間從不斷加深的關係，變成了不斷往下的關係，也因此變得比較好讀。我們在這邊不討論如何包裝 BFTask 物件，詳情請看 Bolts Framework 本身的文件。

相關閱讀

- 在 LLVM 官網上關於 Block 的完整 Spec
- 蘋果官方文件 Blocks Programming Topics
- 蘋果官方文件 Working with Blocks
- Parse Blog - Objective-C Blocks Quiz
- WWDC 2015 Swift and Objective-C Interoperability

練習：將 Web Service API 包裝成 SDK

練習範圍

- Block
- 網路連線
- JSON 格式處理

練習目標

在屬於 Mobile Internet 的時代裡，我們在寫的手機 App 往往不會是像貪食蛇這樣的單機遊戲，更有可能是透過網路連線，抓取或上傳資料，讓用戶可以提供源源不絕的資訊，並且讓用戶與用戶之間溝通。換言之，手機 App 往往就是一個 Internet Client，KKBOX、甚至 KKBOX 公司內的其他產品線，也是這樣的軟體。

在寫這樣的 App 的時候，我們通常會把整個 App 所有跟網路連線相關的部份集中在一起，比方說，我們把前往某個網路服務所有的網路連線呼叫，都放在同一個 Class 裡頭，不同的 Web API 變成不同的 method，而不是 App 裡頭每個個別的地方發送連線。最後寫出來的程式就會很像我們可以看到的 Facebook SDK 等 Web Service SDK 或是 library。

這麼寫有很多好處：因為所有的連線呼叫都在一起，所以，哪天我們想寫一個 Mac 版本的時候，就可以把這整塊程式搬到 Mac 上，而不用 Mac 版另外再寫一次。我們也可以對整個 library 寫單元測試，知道我們設置的連線方式是否正確，以及 server 是否發生異常。

我們要注意：雖然像 NSString、NSData 都有 `initWithContentsOfURL:` 這些 method，可以直接傳入網路上的 URL，從網路上抓取資料，但我們應該避免在 GUI app 裡頭使用這些 API，因為這些 API 會卡住 UI。我們建議使用 NSURLSession 或 NSURLConnection 等物件發送非同步的連線；由於 NSURLConnection 在 iOS 9 deprecate，我們特別建議使用 NSURLSession，而且 NSURLSession 的 callback 用的都是本章介紹的 block。

練習內容

httpbin.org 是一個讓人練習 HTTP 連線的沙箱，我們要練習寫一個 httpbin.org 的 Web Service SDK。這個服務有很多 API endpoint，會回傳 JSON、HTML 或圖片格式的資料，我們要使用以下這些 API：

- <http://httpbin.org/get>
- <http://httpbin.org/post>
- <http://httpbin.org/image/png>

包裝成像是以下的 method

```
- (void)fetchGetResponseWithCallback:(void(^)(NSDictionary *, NSError *))callback;
- (void)postCustomerName:(NSString *)name callback:(void(^)(NSDictionary *, NSError *))callback;
- (void)fetchImageWithCallback:(void(^)(UIImage *, NSError *))callback;
```

1. 收到的 JSON 資料要轉成 NSDictionary 物件，請查詢 NSJSONSerialization 的文件

2. 收到的 image data 要轉成 UIImage 物件
3. 處理 <http://httpbin.org/post> 這支 API 的時候，我們只 post `custname`，像 `custname=kkbox`
4. 這個 library 的每個連線之間都不會互相影響

我們寫完這個 library 之後，也要寫單元測試。但是非同步的單元測試跟我們前面寫過的不太一樣：要測試非同步的 API，我們必須要在 test case 裡頭等待測試回應，不然就會在連線還沒完成之前，test case 就結束了。

我們以前遇到非同步的 test case，需要自己跑 Run Loop，或是用 GCD 的 `dispatch_semaphore_t`，可以參考 [Run Loop 與 NSOperation 與 NSOperationQueue](#) 這兩章後面的章節。不過，在 Xcode 6 之後，比較容易的作法是使用 XCUnit 提供的 XCTestExpectation；如何在 Xcode 6 中做非同步的測試，請參考

- [Testing in Xcode 6](#)
- [XCTestCase/XCTTestExpectation/measureBlock\(\)](#)
- [Asynchronous Testing With Xcode 6](#)

請不要忘了 AAA 原則：

- Arrange: 連線應該要成功，而且應該要正確抓回 dictionary 或 image 物件，dictionary 裡頭也應該要有預期的 key，value 也是預期的型別
- Act: 呼叫 method，發送連線
- Assert: 驗證呼叫的結果是否符合預期

另外，如果你打算把這個 class 寫成 singleton 物件，請先跳到後面閱讀 [再談 Singleton](#) 這一章。

練習：將 Web Service API 包裝成 SDK Part 2

練習範圍

- Delegate

練習目標

一個 Web Service SDK 大概可以分成兩種模式，一種是這個 library 發送出去的連線本身不會互相干擾，另外一種則是會互相取消。

比方說，我們現在在 UI 上有一個搜尋功能，用戶打了一個關鍵字之後，前一個搜尋結果還沒有回傳，用戶又打了一個關鍵字，這個時候我們就必須先把前一個已經發送出去的連線取消掉。原因是：我們並不能夠保證每個連線的時間都是一樣的，如果第一個連線花了更多的時間才抓回資料，第二個連線都結束了一陣子之後，第一個連線才回來，那麼，最後用戶看到的是第一個連線的結果，而這並不符合用戶的期待。

我們現在要寫的就是一套會互相取消連線的 SDK。

練習內容

httpbin.org 是一個讓人練習 HTTP 連線的沙箱，我們要練習寫一個 httpbin.org 的 Web Service SDK。這個服務有很多 API endpoint，會回傳 JSON、HTML 或圖片格式的資料，我們要使用以下這些 API：

- <http://httpbin.org/get>
- <http://httpbin.org/post>
- <http://httpbin.org/image/png>

包裝成像是以下的 method

```
- (void)fetchGetResponse;
- (void)postCustomerName:(NSString *)name;
- (void)fetchImageWithCallback;
```

我們的連線物件只有一個單一的 delegate，當這些 method 的連線完成之後，都會將資料透過一個我們設計好的 protocol 傳給 delegate 物件。在呼叫這些 method 的時候，如果發現有任何還在進行中的連線，都要先取消原本的連線，才發送新的連線。

這個練習一樣要寫單元測試。

Notification Center

Notification Center 是在 Cocoa/Cocoa Touch Framework 中，物件之間可以不必互相知道彼此的存在，也可以互相傳遞訊息、交換資料/狀態的機制。

我們可以把 Notification Center 想像是一種廣播系統。當一個物件 A 的狀態發生改變，而有多個物件需要知道這個物件發生改變的狀況下，物件 A 不必直接對這些物件發出呼叫，而是告訴一個廣播中心說：「我的狀態改變了」，至於其他需要聽取狀態的物件呢，也只要對這個廣播中心訂閱 (subscribe) 指定的通知，所以當物件 A 發出通知的時候，這個廣播中心就會通知有訂閱通知的其他物件。這個廣播中心，就是 Notification Center。

我們經常使用 Notification Center 處理來自作業系統的事件。假如我們現在寫了一個日記軟體，這個軟體裡頭已經有很多 view，每個 view 裡頭都有一篇日記，每篇日記上都有該篇日記的撰寫日期與時間。我們通常會使用 NSDateFormatter，使用系統偏好設定中的語系 (Locale) 設定，將日期轉成符合語系設定的字串顯示，那麼，當用戶調整了系統偏好設定，像是把中文改成英文，那麼，我們原本用中文顯示的日期，也應該馬上變成用英文顯示—我們該怎麼做呢？

我們最常使用的通知中心是 NSNotificationCenter 這個 class，我們也通常使用這個 class 的 singleton 物件 default center（也就是說，其實 Notification Center 有好幾個，不過我們最常使用的還是這個）。當系統語系改變的時候，Notification Center 就會發出叫做 NSCurrentLocaleDidChangeNotification 的這項通知。所以，我們所有要顯示日期的畫面物件，都應該要訂閱這個通知，在收到通知的時候，就要重新產生日期字串。

接收與發送 Notification

接收 Notification

一個通知分成幾個部分

1. **object**: 發送者，是誰送出了這個通知
2. **name**: 這個通知叫做什麼名字
3. **user info**: 這個通知還帶了哪些額外資訊

所以，當我們想要監聽某個通知的時候，便是指定要收聽由誰所發出、哪個名字的通知，並且指定負責處理通知的 selector，以前面處理 locale 變更的例子來看，我們就會寫出這樣的 code：

```
- (void)viewDidLoad
{
    [super viewDidLoad];
    [[NSNotificationCenter defaultCenter] addObserver:self
        selector:@selector(localeDidChange:)
        name:NSNotificationCenterCurrentLocaleDidChangeNotification
        object:nil];
}

- (void)localeDidChange:(NSNotification *)notification
{
    // 處理 locale 變更的狀況
}

- (void)dealloc
{
    [[NSNotificationCenter defaultCenter] removeObserver:self];
}
```

意思就是，我們要指定 name 為 NSCurrentLocaleDidChangeNotification 的通知，交由 `localeDidChange:` 處理。在這邊 object 設定為 nil，代表不管是哪個物件送出的，只要符合 NSCurrentLocaleDidChangeNotification 的通知，我們統統都處理。

每個通知當中，還可能會有額外的資訊，會夾帶在 NSNotification 物件的 userInfo 屬性中，userInfo 是一個 NSDictionary。

像是我們如果讓某個 text field 變成了 first responder，那麼，在 iOS 上，就會出現螢幕鍵盤，而當螢幕鍵盤出現的時候，我們往往要調整畫面的 layout，而螢幕鍵盤在不同輸入法下的大小不一樣，像是跟英文鍵盤比較起來，中文輸入鍵盤上面往往會有一塊組字區，而造成中文鍵盤比英文鍵盤大。在螢幕鍵盤升起來的時候，我們會收到 UIKeyboardWillShowNotification 這項通知，而這項通知就會用 userInfo 告訴我們鍵盤尺寸與位置。

如果我們在 `-addObserver:selector:name:object:` 裡頭，把 name 指定成 nil，就代表我們想要訂閱所有的通知，通常不太會有這種情境，不過有時候你想要知道系統內部發生了什麼事情，可以用這種方式試試看。

當我們不需要繼續訂閱某項通知的時候，記得對 Notification Center 呼叫 `-removeObserver:`，以上面的程式為例，我們在 add observer 的時候傳入了 self，在 remove observer 的時候，就要傳入 self。我們通常在 dealloc 的時候停止訂閱。

在 iOS 4 與 Mac OS X 10.6 之後，我們可以使用 `-addObserverForName:object:queue:usingBlock:` 這組使用 block 語法的 API 訂閱通知，由於是傳入 block，所以我們就不必另外準備一個 selector，可以將處理 notification 的程式與 add observer 的這段呼叫寫在一起。而 remove observer 的寫法也會不太一樣：`-removeObserverForName:object:queue:usingBlock:` 會回傳一個 observer 物件，我們想要停止訂閱通知的時候，是對 `-removeObserver:` 傳入之前拿到的 observer 物件。範例如下。

Add observer 的時候：

```
self.observer = [[NSNotificationCenter defaultCenter]
    addObserverForName:NSCurrentLocaleDidChangeNotification
    object:nil
    queue:[NSOperationQueue mainQueue]
    usingBlock:^(NSNotification *note) {
        // 處理 locale 變更的狀況
}];
```

Remove observer 的時候：

```
[[NSNotificationCenter defaultCenter] removeObserver:self.observer];
```

發送 Notification

至於要發送 notification，則是在建立了 notification 物件之後，對 NSNotificationCenter 呼叫 `-postNotification:` 即可。

這三組 method 都可以用來發送 notification。

```
- (void)postNotification:(NSNotification *)notification;
- (void)postNotificationName:(NSString *)aName
    object:(id)anObject;
- (void)postNotificationName:(NSString *)aName
    object:(id)anObject
    userInfo:(NSDictionary *)aUserInfo;
```

Notification 與 Threading

當我們訂閱某個 notification 之後，我們並不能夠保證負責處理 notification 的 selector 或 block 會在哪個 thread 執行：這個 notification 是在哪條 thread 送出的，負責接受的 selector 或是 block，就會在哪條 thread 執行。

在慣例上，絕大多數的 notification 都會在 main thread 送出，之所以說「絕大多數」，就是因為有例外：像是在 iOS 上，如果我們接上耳機、拔除耳機，或是將音樂透過 AirPlay 送到 Apple TV 的時候，系統會透過 AVAudioSessionRouteChangeNotification 告訴我們音訊輸出設備改變了¹，這個通知就會發生在背景，而不是 main thread。

不過，當我們在撰寫自己的程式，要發送 notification 的時候，為了考慮其他開發者會預期在 main thread 收到 notification，所以我們也就在 main thread 發送 notification，像是透過 GCD，把 postNotification 的呼叫送到 `dispatch_get_main_queue()` 上。

¹. 參見蘋果官方文件 Responding to Route Changes -

<https://developer.apple.com/library/ios/documentation/Audio/Conceptual/AudioSessionProgrammingGuide/Han dlingAudioHardwareRouteChanges/HandlingAudioHardwareRouteChanges.html> ↵

Notification Queue

有的時候，我們的程式可能會在很短的時間送出大量的 notification，而造成資源的浪費或效能問題。

以 KKBOX 來說，我們在歌單中的歌曲物件發生改動的時候，會透過 notification 更新 UI：一首歌曲可能會出現在多個歌單中，而我們可能用了很多不同的 UI 物件來呈現不同張歌單，因此，像一首歌曲的播放次數改變、如這首歌被多播放了一次，歌曲物件就透過 notification center，告訴每個跟歌單 UI 相關的物件重新讀取歌單資料。

照理說，只要有一首歌曲改變，就該發出這種通知，但假如我們現在做的事情是歌單同步—把另外一台裝置上的歌單資料，同步到我們這台裝置上，那麼改動的就不只是一首歌曲，而是一大批的歌曲，如果有十首歌，就送出了十次通知；但是，其實 UI 只需要改動一次就好了，沒有重複更新十次 UI 的必要。

這時候我們就該用 NSNotificationQueue。我們可以把 NSNotificationQueue 想成 notification 的發送端與 notification center 之間的一個 buffer，這個 buffer 可以讓我們暫緩送出 notification，而在一段緩衝期之內，決定我們是否要合併通知。以前面的例子來看，我們就可以先把原本預計的十次通知先放進 NSNotificationQueue 當中，然後讓 NSNotificationQueue 幫我們把十次通知合併成只有一次通知。

我們要先建立一個 NSNotificationQueue 物件：

```
notificationQueue = [[NSNotificationQueue alloc]
    initWithNotificationCenter:[NSNotificationCenter defaultCenter]];
```

再來我們發送通知的程式原本像這樣：

```
NSNotification *n = [NSNotification
    notificationWithName:@"KKSongInfoDidChangeNotification"
    object:self];
[[NSNotificationCenter defaultCenter] postNotification:n];
```

改寫成這樣：

```
NSNotification *n = [NSNotification
    notificationWithName:@"KKSongInfoDidChangeNotification"
    object:self];
[notificationQueue enqueueNotification:n
    postingStyle:NSPostASAP
    coalesceMask:NSNotificationCoalescingOnName | NSNotificationCoalescingOnSender
    forModes:nil];
```

我們在這邊傳入了 `NSNotificationCoalescingOnName` 與 `NSNotificationCoalescingOnSender`，代表的就是請 notification queue 合併名稱相同、發送者也相同的通知。

Mac 上的其他 Notification Center

在 iOS 上面我們通常只會用到 NSNotificationCenter，特別是 NSNotificationCenter 的 defaultCenter：不過，在 Mac OS X 上，我們還有其他的 notification center 可以使用。

NSDistributedNotificationCenter

蘋果在 iOS 上的限制較為嚴格，一直以來都想辦法禁止跨 App 之間的通訊（IPC，Inter-Process Communication）。不過自從 Mac OS X 出現以來，Cocoa Framework 就有 Distributed Objects 這套 IPC 機制，讓不同 App 之間可以傳遞 Objective-C 物件，後來更推出了 XPC，可以在不同 App 之間傳遞 block。

NSDistributedNotificationCenter 就是在 Distributed Objects 技術上建立的 notification center，也就是，如果你對 NSDistributedNotificationCenter 發送了通知，便可以讓其他的 App 收到來自你目前所在 App 送出的通知。

NSWorkSpace 的 Notification Center

NSWorkSpace 這個物件在 Mac 上代表的是 Mac 的桌面環境。如果你想要要求 Mac OS X 開啟另外一個 App，處理某個檔案或 URL（在 iOS 上我們會要求 UIApplication 來 openURL:，但是在 Mac 上則是交由 NSWorkSpace 處理），或是取得某個檔案在 Finder 裡頭的代表圖示…等，就會用到 NSWorkSpace。

跟 NSWorkSpace 相關的通知，像是某個 App 是否被成功開啟、你的 Mac 電腦是否離開了休眠…等等，都不會透過 NSNotificationCenter 的defaultCenter，而是要透過 `[[NSWorkSpace sharedWorkspace] notificationCenter]` 這邊的 notification center，我們要選擇正確的 notification center 做 add observer，才能正確收到通知。

CFNotificationCenter

CFNotificationCenter 是 NSNotificationCenter 在 Core Foundation 中的 C 實作，一般來說，如果我們有比較高階的 API 可以使用的話，我們會盡量避免使用比較低階的 API，所以，只要有 NSNotificationCenter 可以使用的場合，我們應該不會用到 CFNotificationCenter。

比較有可能用到 CFNotificationCenter 的場合，大概是 iOS 8 之後，Hosting App 與 Extension 之間的溝通。iOS 8 之後推出了 Extension，可以允許開發者撰寫 Today Widget、Share Widget 以及模擬鍵盤等功能，Apple Watch 上的 Watch App 也屬於 Extension；每個 Extension 都是額外可以讓作業系統載入的 Bundle，Extension 與我們原本的 App（便是 Hosting App）之間，可以用 Shared Data 共用資料，但是當 App 發生改變要通知 Extension，卻沒有什麼比較直接的辦法。在蘋果有新的 API 之前，我們就會倚賴透過

`CFNotificationCenterGetDarwinNotifyCenter()` 取得的 darwin notification center 發送通知。

而即使我們可以發送通知，CFNotificationCenter 用起來也不是很方便，主要原因是 CFNotificationCenter 不像 NSNotificationCenter，在傳遞通知的時候可以夾帶 user info。再來，就是像前面說的，CFNotificationCenter 是 C API，而我們會盡量希望使用比較高階的 API。

所以，KKBOX 在開發 Watch App 的時候，就在 CFNotificationCenter 上面又簡單架構了一層 Objective-C API，介面類似 NSNotificationCenter，叫做 KKWatchAppNotificationCenter。程式碼如下：

KKWatchAppNotificationCenter.h

```
@import Foundation;

@interface KKWatchAppNotificationCenter : NSObject
+ (instancetype)sharedCenter;
- (void)postNotification:(NSString *)key;
- (void)addTarget:(id)target selector:(SEL)selector name:(NSString *)notification;
- (void)removeObserver:(NSObject *)observer;
@end
```

KKWatchAppNotificationCenter.m

```
#import "KKWatchAppNotificationCenter.h"

#define LFSuppressPerformSelectorLeakWarning(doPerformSelector) \
do { \
_Pragma("clang diagnostic push") \
_Pragma("clang diagnostic ignored \"-Warc-performSelector-leaks\"") \
doPerformSelector; \
_Pragma("clang diagnostic pop") \
} while (0)

@interface KKWatchAppNotificationCenter ()
{
    NSMutableDictionary *notificationKeys;
}
@end

@implementation KKWatchAppNotificationCenter
```

```

+ (instancetype)sharedCenter
{
    static KKWatchAppNotificationCenter *sharedRegister = nil;
    static dispatch_once_t onceToken;
    dispatch_once(&onceToken, ^{
        sharedRegister = [[KKWatchAppNotificationCenter alloc] init];
    });
    return sharedRegister;
}

- (instancetype)init
{
    self = [super init];
    if (self) {
        notificationKeys = [[NSMutableDictionary alloc] init];
    }
    return self;
}

- (void)postNotification:(NSString *)key
{
    CFNotificationCenterRef const center = CFNotificationCenterGetDarwinNotifyCenter();
    CFDictionaryRef const userInfo = NULL;
    BOOL const deliverImmediately = YES;
    CFNotificationCenterPostNotification(center,
                                         (CFStringRef)key, NULL, userInfo, deliverImmediately);
}

- (void)addTarget:(id)target selector:(SEL)selector name:(NSString *)notification
{
    if (!target) {
        return;
    }

    if (!selector) {
        return;
    }

    if (!notification || ![notification length]) {
        return;
    }

    NSMutableArray *a = [notificationKeys objectForKey:notification];
    BOOL needRegisterNotification = NO;

    if (!a) {
        a = [NSMutableArray array];
        needRegisterNotification = YES;
    }

    for (NSDictionary *d in a) {
        if (d[@"target"] == target &&

```

```

        NSSelectorFromString(d[@"selector"]) == selector) {
            return;
        }
    }

    NSDictionary *d = @{@"@target": target, @"selector": NSStringFromSelector(selector)};
    [a addObject:d];

    if (needRegisterNotification) {
        [self registerNotification:notification];
        [notificationKeys setObject:a forKey:notification];
    }
}

- (void)removeObserver:(NSObject *)observer
{
    NSMutableArray *notificationsToDelete = [[NSMutableArray alloc] init];
    for (NSString *notification in notificationKeys) {
        NSMutableArray *a = notificationKeys[notification];
        NSMutableArray *objectsToDelete = [NSMutableArray array];
        for (NSDictionary *d in a) {
            id target = d[@"target"];
            if (target == observer) {
                [objectsToDelete addObject:d];
            }
        }
        [a removeObjectsInArray:objectsToDelete];
        if (![a count]) {
            [notificationsToDelete addObject:notification];
        }
    }
}

for (NSString *notification in notificationsToDelete) {
    [self unregisterNotification:notification];
    [notificationKeys removeObjectForKey:notificationKeys];
}
}

- (void)registerNotification:(NSString *)key
{
    CFNotificationCenterRef const center = CFNotificationCenterGetDarwinNotifyCenter();
    CFNotificationSuspensionBehavior const suspensionBehavior = CFNotificationSuspensionBehaviorDeliverImmediately;
    CFNotificationCenterAddObserver(center,
        (__bridge const void *)(self),
        KKWatchAppNotificationRegisterCallback,
        (CFStringRef)key, NULL, suspensionBehavior);
}

- (void)unregisterNotification:(NSString *)key
{
    CFNotificationCenterRef const center = CFNotificationCenterGetDarwinNotifyCenter();
    CFNotificationCenterRemoveObserver(center,

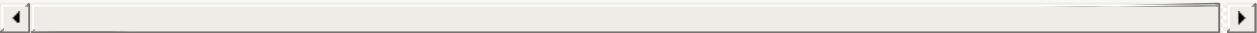
```

```
(__bridge const void *)(self),
(CFStringRef)key, NULL);
}

void KKWatchAppNotificationRegisterCallback(CFNotificationCenterRef center,
    void * observer, CFStringRef name, void const * object, CFDictionaryRef userInfo)
{
    KKWatchAppNotificationCenter *self = (__bridge KKWatchAppNotificationCenter *)observer;
    NSString *notification = (__bridge NSString *)name;

    NSArray *a = [self->notificationKeys objectForKey:name];
    for (NSDictionary *d in a) {
        id target = d[@"target"];
        SEL action = NSSelectorFromString(d[@"selector"]);
        LFSuppressPerformSelectorLeakWarning([target performSelector:action withObject:nil]);
    }
}
}

@end
```



相關閱讀

- [Notification Programming Topics](#)
- [Introduction to Workspace Services](#)

練習：填字遊戲



練習範圍

- MVC
- Notification Center
- Quartz2D
- Delegate

練習目標

我們要寫一個填字遊戲的操作介面

1. 在畫面中有一個 8x8 的表格，每個格子裡頭只有一個字
2. 當我們按到其中一個格子的時候，這個格子會被反白起來，並且允許我們可以修改格子裡頭的文字
3. 在修改格子裡頭的文字的時候，螢幕鍵盤會升起，我們要調整整個 view 的位置，避免螢幕鍵盤蓋到我們正在輸入的格子
4. 移動 view 的速度與鍵盤上升的速度要一致
5. 按下 Enter 之後，鍵盤會收回去，格子裡頭會變成修改後的內容
6. 格子裡頭只能夠允許剛好是一個字 (character)，不可以不輸入，也不可以超過一個字。不是剛好一個字，就無法在鍵盤上按下 Enter

練習內容

這個練習也是使用 MVC 架構

- View: 使用 Quartz2D 繪製格線與文字
- Controller: 保有一個 8x8 的二維 array 作為 Model

Controller 是 View 的 delegate。當 View 需要重繪內容的時候，會跟 Controller 要求 Model，然後按照 Model 繪製。

我們使用 UIGestureRecognizer 在 View 上面增加了 tap 事件，在 tap 到的時候，會計算 tap 所發生的位置，決定是哪一格被 highlight。在 highlight 的時候，我們的 View 會把一個 UITextField 疊在剛好與 highlight 的格子範圍的上方，同時這個 text field 會變成 first responder。整個 view 雖然有 8x8 個文字，但是只共用同一個 text field，其實也是剛好練習 Mac 上的 "field editor" 的觀念。

我們透過 Notification 知道螢幕鍵盤上升與收起，同時透過 user info 知道螢幕鍵盤升起需要多少時間，以及最後的位置，再根據這些資訊決定怎麼移動我們的 view。

讓 text field 只能輸入一個字，以及在按下 enter 之後會改變 Controller 中的 8x8 二維 array，並要求 view 重繪，是透過 text field 的 delegate 完成。

練習：自行實作 NSNotificationCenter

練習範圍

- Notification Center

練習目標

要了解 Notification Center 是怎麼運作的，一種很好的學習方法就是我們自己重新實作一次，所以我們要來寫一個自己的 KKNotificationCenter。

Interface 如下：

```
@interface KKNotificationCenter : NSObject  
  
+ (KKNotificationCenter *)defaultCenter;  
  
- (void)addObserver:(id)observer selector:(SEL)aSelector name:(NSString *)aName object:(id)anObject;  
- (void)postNotification:(NSNotification *)notification;  
- (void)removeObserver:(id)observer;  
@end
```

請實作這個 Class，讓這個 Class 擁有和 NSNotificationCenter 相同的功能。

所謂的設計模式

在這邊對前面的章節做一個簡短的回顧。

我們在前面提到了許多名詞，像是 Delegate、Singleton、Notification Center、Abstract Factory...等等，這些名詞用來描述我們平常撰寫程式時所使用的方法，我們通常會稱為「設計模式」（Design Pattern）。

根據 Wikipedia 的解釋，設計模式就是「在軟體設計中，針對一般問題、並且可以重複使用的解決方案」（a design pattern is a general reusable solution to a commonly occurring problem within a given context in software design）。

iOS 與 Mac OS X 中大量使用上面提到的這些設計模式，但其實平常在開發的時候，我們倒不會刻意強調設計模式，因為使用的數量已經多到像是呼吸喝水一樣自然。像我們開始寫第一行 iOS 程式的時候，可能會寫在 AppDelegate 中，而 AppDelegate 這個名詞本身就帶有兩個設計模式的意味：AppDelegate 就是 UIApplication 的 delegate，而 UIApplication 是 Singleton 物件，所以在寫第一行 code 的時候，你已經遇到了 Singleton 與 Delegate。

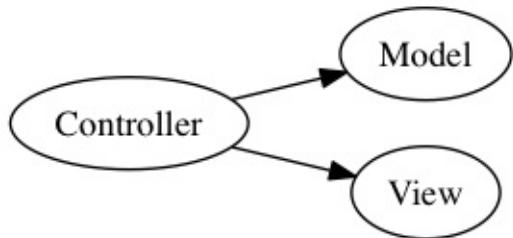
坊間有許多跟設計模式有關的專書，非常建議一讀。但我們要注意一下，學習設計模式與怎麼使用設計模式又是一回事，某些設計模式可能用來解決某些特定問題，不見得適合其他場合。並不是用了比較多設計模式就會比較好，使用設計模式並不是軍備競賽。

某方面來說，我們可以把設計模式，想成是一些物件之間的排列關係。而使用設計模式最後要做的，就是當我們的 App 或 Library 中有很多不同種類的物件時，我們怎麼把這些物件排列成合理的關係。

圖解設計模式

我們來把一些常用設計模式中的物件關係畫成圖：

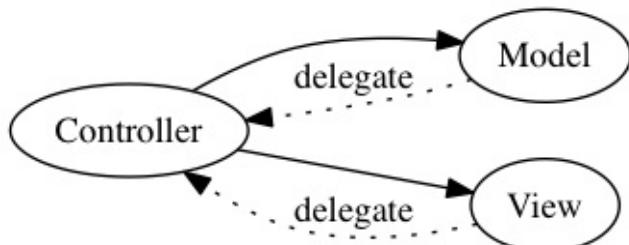
MVC



MVC 便是將物件分成 Model、View、Controller 三類，幾乎現在主要平台上都採用這種設計，不過就像前面也提到的，不是每個平台都一致。像 Windows 上往往把 window 物件當成 controller，但是在 Mac 上 window 被劃入到 view 這一塊。

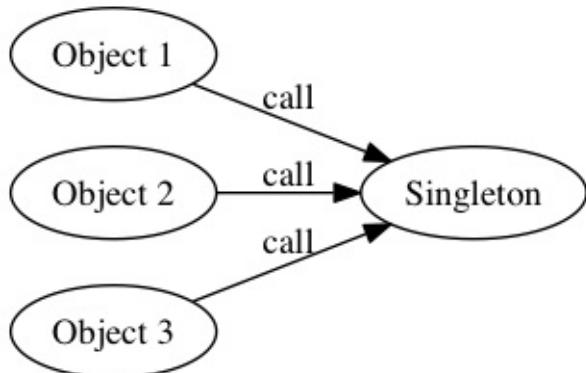
一般來說會由 Controller 擁有 Model 與 View 物件。

Delegate



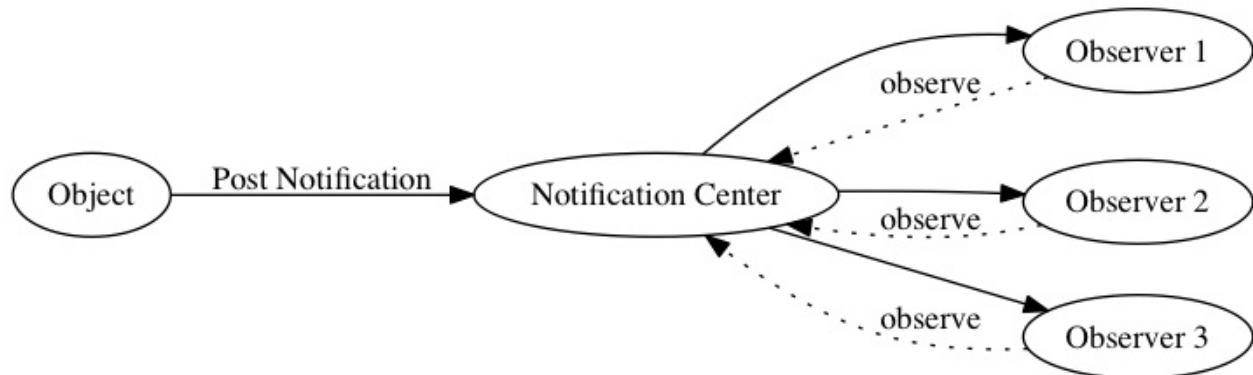
MVC 中，Controller 擁有 View 與 Model，所以 Controller 可以直接呼叫 View 與 Model 上的各種 method，但是當 View 與 Model 需要呼叫 Controller 的時候，會把 Controller 設定成 delegate，而 delegate 只需要符合 protocol 的定義，不需要是特定 Class，避免 View 與 Model 繩死在某個 Controller 上。

Singleton



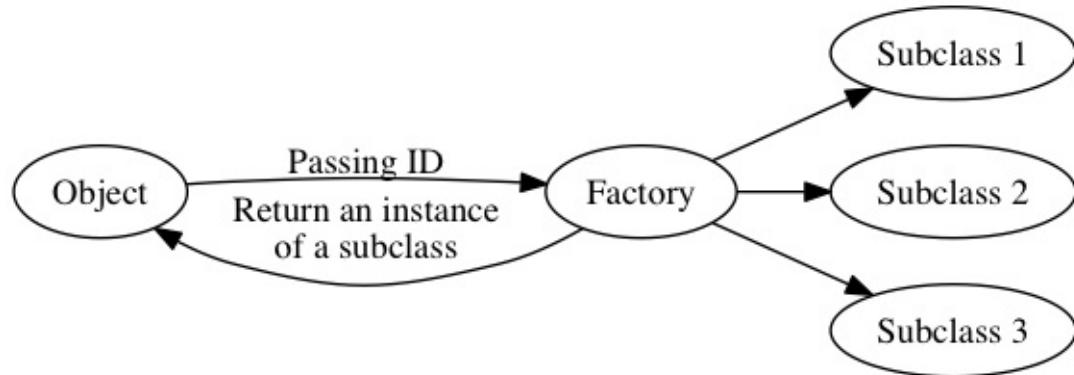
某個 class 只有一個 instance，這樣在其他地方，都可以集中找到同一個 instance。像 UIApplication 等物件就是 Singleton 物件。

Notification Center



一個物件改變狀態的時候，其他物件不需要知道這個物件存在，也跟著一起改變狀態。每個物件之間透過 Notification Center 互相通知、又互相隔絕。

Factory Method



把 Factory 變成建立物件的唯一入口，其他地方不需要知道某個 class 確實來說是怎麼做的，就可以建立需要的物件。Factory 於是可以把外部與內部實作隔絕開來。

為什麼要使用設計模式？

與他人一起工作

寫程式時要保持這種心態：就好像將來要維護你這些程式的人是一位殘暴的精神病患者，而且他知道你住在哪。（Always code as if the guy who ends up maintaining your code will be a violent psychopath who knows where you live.） - Martin Golding

寫程式並不是一件孤單的事。在開發的過程中，會有其他的同仁共同參與，你可能接手前人的程式，也可能會有人接手你現在撰寫的程式；程式不只是一批只交付給機器執行的命令，也是一種讀物，你閱讀別人的程式，別人也會閱讀你的程式，因為程式就是寫作與閱讀，所以程式本身就是溝通。

你想要读懂別人的程式，也想要讓別人可以读懂你的程式，那麼在工程師之間就需要一套共同的詞彙與修辭學。

如果我們把變數想像成單字、把程式語言的語法想成文法，那麼我們知道，有時候一篇文章就算你每個字都看得懂，但是把每個字串在一起，你就是不知道裡頭到底想要表達什麼，所以你就會需要一套把不同元素組合在一起的敘事結構，像是起承轉合、像是倒敘法...在物件導向程式中，這種排列方式就是設計模式。

學習設計模式就是學習工程師的語言，使用設計模式就是與其他工程師溝通，讓其他參與人可以快速理解整個軟體的架構。

適應軟體的變化

如果我們把設計模式理解成物件的排列方式，我們透過一套有系統的方式，把物件組織起來，也就是管理物件之間的關係。而管理物件的關係，除了代表讓一些物件有關係之外，同時也代表讓許多物件之間沒有關係，解除物件之間的相依性。

如果在一個有了一些年紀的 code base 上工作，你會發現，所謂維護一套軟體，其實就是在不斷改動軟體，你不但要讓基本功能隨著時間過去、作業系統與 SDK 的改版還能夠繼續運作，還要隨著每年的商業目標改變而滿足不同的需求，有些新的需求會誕生，有些過去的需求會不復存在，你會不斷增加新的功能，另一方面，你也會需要不斷移除沒有用的程式碼。

從軟體中移除程式碼遠比加入程式碼困難，如果我們加入程式碼的時候就像把一把紅豆灑進一缸綠豆，那麼刪除程式碼就像是要從一缸綠豆裡頭把這一把紅豆一顆一顆挑出來。如果一開始每個物件之間的關係都糾結在一起，當我們想要移除沒有用到的程式時，根本就無從下手，而放任無用的程式繼續在軟體中存活，最終就會危害整體的軟體品質。

維護軟體就是改動軟體，撰寫單元測試是為了有工具可以讓我們在改動的時候，確認沒有把軟體改爛，使用設計模式則是讓我們想要改動的時候還有辦法改得動。維護一套有一些年紀的軟體，在每次把程式加進去的時候，都要考慮以後你要怎麼把這些程式拿出來。

再談 Singleton

我們現在用 Objective-C 實作 Singleton 的時候，大概都是按照 Mike Ash 的建議—參見 [Friday Q&A 2009-10-02: Care and Feeding of Singletons](#)—使用 GCD 裡頭的 `dispatch_once` 實作。大概像這樣：

```
@interface MyClass : NSObject
+ (instancetype) sharedInstance;
@end

@implementation MyClass

+ (instancetype) sharedInstance
{
    static MyClass *instance = nil;
    static dispatch_once_t onceToken;
    dispatch_once(&onceToken, ^{
        instance = [[MyClass alloc] init];
    });
    return instance;
}
@end
```

之所以這麼寫的原因，是為了避免在多重 thread 的環境下，shared instance 可能會被重複建立的問題。我們來看如果只用 if 語法實作的 singleton：

```
@implementation MyClass
+ (instancetype) sharedInstance
{
    static MyClass *instance = nil;
    if (!instance) {
        instance = [[MyClass alloc] init];
    }
    return instance;
}
@end
```

在 `instance` 這個變數還沒有建立的狀況下，假如就已經有多個 thread 同時進入了前面這段程式的 if 判斷式當中，那麼，每個進入這段程式的 thread，都會分別執行到 `instance = [[MyClass alloc] init];` 這一行，而重複建立 `instance`。我們對於 Singleton 物件的要求是：這個 Class 只會建立一個 `instance`，在所有地方呼叫 `+sharedInstance`，也都只該拿到同一個物件，但是在這種寫法中，不同 thread 呼叫 `+sharedInstance`，就可能會拿到不同的物件。

由於 `dispatch_once` 裡頭的 block 只會執行一次，我們便透過這個特性，保證 `instance = [[MyClass alloc] init];` 只會被呼叫到一次。

相關閱讀

- [Singleton](#)

- Friday Q&A 2009-10-02: Care and Feeding of Singletons

練習：探索 Cocoa/Cocoa Touch Framework

請做一份書面報告。在這份書面報告中，請從 Cocoa/Cocoa Touch 裡頭的各個 Framework 中（像是 Foundation、UIKit...）等等，找到以下的 Class：

1. 找出五個 singleton class
2. 找出五個有 delegate 或 data source 的 class
3. 找出五個會發送 notification 的 class
4. 找出五個使用 block 的 class

然後：

- 請用一百個中文字以內，說明這個 class 的用途
- 請用一百個中文字以內，說明這個 class 為什麼是 singleton/有 delegate...

範例：

- UIDevice：用來代表目前 App 所在裝置狀態的物件，可以用 currentDevice 回傳代表目前所在裝置的物件，從物件上我們可以知道裝置名稱、作業系統版本等資訊
- 由於我們的 app 一次只會在一台裝置上執行，對我們的 app 而言，就只能夠知道一台裝置的存在，因此 UIDevice 被設計成 singleton 物件

練習：閱讀程式碼

請做一份大約 15 到 30 分鐘左右的簡報：

1. 在 GitHub 找一個 open source 的 iOS 專案，可以是 App 或 Library
2. 簡介這個專案的用途
3. 說明在這個專案中，用了哪些設計模式、用在哪邊？
4. 如果這是一個 Library，就你自己評價，這個 Library 好不好用？為什麼？
5. 如果是你的話，你會怎樣設計這個 App 或 Library？
6. 如果這是一個開發一段時間的專案，你覺得從哪些地方可以看出歲月的痕跡？如果由你接手，你會打算在哪些地方、用新的技術改寫？

一些新手常常搞混的東西

在這一章當中，我們會花一點點力氣，解釋一下在前幾個章節寫練習的時候我們搞混的東西。

bool 與 BOOL

Objective-C 語言是在 C 語言的基礎上，又加了一層物件導向與動態語言特性的語言，很多基本型別是直接來自於 C 語言。C 語言在發展之初，並沒有布林值，於是 Objective-C 語言在發展的過程中，定義了自己的 BOOL，但是在 C99 規格中，C 語言又有了自己的布林型別 bool，而 Objective-C 又可以混和 C++ 語法變成 Objective-C++，C++ 裡頭也有 bool。

那麼，Objective-C 的 BOOL，與 C99 以及 C++ 的 bool 有什麼差別呢？我們來看 裡頭的宣告，這是 iOS 9 SDK 的版本：

```
#include <stdbool.h>

...
/// Type to represent a boolean value.
#if (TARGET_OS_IPHONE && __LP64__) || TARGET_OS_WATCH
#define_OBJC_BOOL_IS_BOOL 1
typedef bool BOOL;
#else
#define_OBJC_BOOL_IS_CHAR 1
typedef signed char BOOL;
// BOOL is explicitly signed so @encode(BOOL) == "c" rather than "C"
// even if -funsigned-char is used.
#endif

#if __has_feature(objc_bool)
#define YES __objc_yes
#define NO __objc_no
#else
#define YES ((BOOL)1)
#define NO ((BOOL)0)
#endif
```

也就是說，在 iOS 的 64 位元或是在 Apple Watch 上，Objective-C 的 BOOL 會直接等於定義在 stdbool.h 裡頭的 bool，其實是 int，而如果使用了 C++，那麼stdbool.h 裡頭的定義就變成是 C++ 的 bool。但，如果是在 Mac OS X 上，或是32 位元的 iOS 環境下，BOOL 就會被定義成是一個 char，而 BOOL 與 bool，就分別是一個 byte 或是四個 bytes 的差別。

The screenshot shows the Xcode interface with the following details:

- Top Bar:** Shows the title "Finished running Size : Size".
- Toolbar:** Standard Xcode toolbar with icons for file operations.
- Editor Area:** Displays the code in main.m:


```
1 #import <Foundation/Foundation.h>
2
3 int main (int argc, const char * argv[])
4 {
5     @autoreleasepool {
6         NSLog(@"YES: %ld", sizeof(YES));
7         NSLog(@"true: %ld", sizeof(true));
8     }
9     return 0;
10 }
```
- Output Area:** Shows the command-line output:


```
2015-10-26 20:39:50.046 Size[44668:4662918] YES: 1
2015-10-26 20:39:50.047 Size[44668:4662918] true: 4
Program ended with exit code: 0
```
- Bottom Bar:** Shows "All Output" and standard Xcode interface controls.

所以，在 64 位元或 Apple Watch 上，BOOL 與 bool 並沒有差別，但我們通常不能假設我們寫的 code 只會在這種環境下執行，雖然在其他環境下，使用BOOL 或是 bool 通常也沒什麼影響，但既然某個 API 明確就是要求你傳入BOOL，那就傳入 YES 或 NO，好像也沒什麼非要傳入 true 或 false 的理由。

另外，當我們想把 BOOL 轉成 NSNumber，最簡單的寫法還是直接使用Xcode 4.4 之後的 literals 的寫法，寫成 `@(YES)` 或 `@(NO)`，另外 Core Foundation 裡頭也定義了 `kCFBooleanTrue` 與 `kCFBooleanFalse`，也具有同樣的功能。

NSInteger 與 NSUInteger

我們在使用 Objective-C 語言寫程式的時候，往往使用 NSInteger 與 NSUInteger 代表帶號與非帶號的整數。嚴格來說，NSInteger 並不能算是「Objective-C 的整數」，因為 NSInteger 其實是 C 語言的形態，而不是 Objective-C 物件，用來代表數字的 Objective-C 物件，是像 NSNumber，以及我們在前面小計算機練習中用到的 NSDecimalNumber。

NSInteger 就是 C 的整數。我們來看一下 裡頭的定義：

```
#if __LP64__ || (TARGET_OS_EMBEDDED && !TARGET_OS_IPHONE) || TARGET_OS_WIN32 || NS_BUILD_32_LIKE_64
typedef long NSInteger;
typedef unsigned long NSUInteger;
#else
typedef int NSInteger;
typedef unsigned int NSUInteger;
#endif
```

在 裡頭宣告了一些 Macro，在 64 位元環境下，NSInteger 被宣告成 long，也就是 64 位元的整數，在 32 位元環境下則是 int。同時，NSUInteger 在 64 與 32 位元環境下分別是 unsigned long 與 unsigned int。

我們寫的程式往往需要同時在 64 位元與 32 位元環境上執行，像 iPhone 5 與之前的機種使用 armv7 CPU、是 32 位元環境，iPhone 5S 之後則在 arm64 上。因此，當我們在使用整數的時候，即使我們也可以直接使用 int 或 long，但我們會盡量使用 NSInteger 與 NSUInteger，讓 compiler 幫我們決定應該是 32 或 64 位元整數。

另外，在使用浮點數的時候，也盡量使用 CGFloat。CGFloat 一樣會在不同環境下，被當成 32 或 64 位元浮點數。

NULL、nil、Nil...

在 Objective-C 語言中，有很多個代表「沒有東西」的東西，一開始也很容易搞混。包括：

- NULL
- nil
- Nil
- NSNull
- NSNotFound

NULL

NULL 實際並不是 Objective-C 的東西，而是屬於 C 語言。NULL 就是 C 語言當中的空指標，是指向 0 的指標。絕大多數狀況下，nil、Nil 與 NULL 可以代替使用，但是語意上，當某個 API 想要你傳入某個指標 (void *) 時，而不是 id 型別時，雖然你可以在這種狀況下傳入 Objective-C 物件指標，也就是可以傳入 nil，但是傳入 NULL 意義會比較清楚。

像建立 NSTimer 時，API 在 userInfo 這邊要求的是 id，我們傳入 nil 會比較好：

```
+ (NSTimer *)timerWithTimeInterval:(NSTimeInterval)seconds
    target:(id)target
    selector:(SEL)aSelector
    userInfo:(id)userInfo
    repeats:(BOOL)repeats
```

而像 UIView 的 beginAnimations:context 的定義是：

```
+ (void)beginAnimations:(NSString *)animationID context:(void *)context;
```

在這邊，傳入 NULL 就會比傳入 nil 好。

nil

nil 是空的 Objective-C 物件指標，也一樣是指向 0。如果我們建立了一個 Objective-C 物件的變數，當我們不想要使用這個物件的時候，便可以將這個變數指向 nil；我們可以對 nil 呼叫任何的 Objective-C method，都不會產生問題。

我們需要注意在 Array 與 Dictionary 中使用 nil 的狀況。在使用 NSArray 的 -arrayWithObjects: 或 NSDictionary 的 -dictionaryWithObjectsAndKeys: 這些被標為 NS_REQUIRE_NIL_TERMINATION 的 method 的時候，nil 會被當成是最後一個參數，出現在 nil 之後的參數都會被忽略，而且我們在傳入參數的時候，最後一個參數也一定要是 nil。

比方說，我們寫一段這樣的程式：

```
NSArray *a = [NSArray arrayWithObjects:@1, @2, nil, @3];
NSLog(@"%@", a);
```

這個 array 就只會有 @1 與 @2，@3 就會被截掉。主要原因是，這類 method 大概都是這樣實作的：首先使用一個 va_list 讀取所有傳入的參數，然後用迴圈呼叫 `va_arg`，只要遇到 nil 就停止迴圈，像下面這段程式：

```
void test(id arg, ...){
    va_list list;
    va_start(list, arg);
    do {
        if (arg == nil) {
            break;
        }
        NSLog(@"%@", arg);
    } while ((arg = va_arg(list, id)));
    va_end(list);
}
```

另外，當我們對 NSMutableArray 插入 nil，或使用 Xcode 4.4 之後的 literal 來寫 NSArray 或 NSDictionary 的時候，如果傳入 nil，也會發生 exception 而造成 crash。下面這段 code 一定會 crash。

```
NSMutableArray *a = [[NSMutableArray alloc] init];
[a addObject:(id)nil];
```

```
NSArray *a = @[(id)nil];
```

Nil

nil 是空的 instance，而開頭大寫的 Nil 則是指空的 class。比方說，當我們想要判斷某個 Class 是不是空的，語意上應該用 Nil 而不是 nil。

我們其實不常判斷一個 Class 是不是 Nil。比較有可能的場合，是為了處理向下相容，像某個 Class 只在某一版的新 OS 上存在，但我們還需要支援舊的 OS，所以我們會在確定某個 Class 不是 Nil 的狀況下，才執行某段程式碼：

```
Class cls = NSClassFromString(@"Abcdefg");
if (cls != Nil) {
    // Do something.
}
```

但如果我們去看，nil 與 Nil 其實是一樣的。

```
#ifndef Nil
# if __has_feature(cxx_nullptr)
#   define Nil nullptr
# else
#   define Nil __DARWIN_NULL
# endif
#endif
```

```
#ifndef nil
# if __has_feature(cxx_nullptr)
#   define nil nullptr
# else
#   define nil __DARWIN_NULL
# endif
#endif
```

NSNull

不同於 NULL、nil 與 Nil，NSNull 是確實存在的 Objective-C 物件。前面講過，我們無法在 array 或 dictionary 中插入 nil，但有的時候我們會需要用一個東西代表「沒有東西」，就會使用 [NSNull null] 這個物件。

假如我們拿到一個 JSON 檔案，然後透過 NSJSONSerialization，把 JSON 檔案轉換成 Objective-C 物件，在這個物件中，JSON dictionary 會轉成 NSDictionary，array 會轉成 NSArray，字串與數字分別會轉換成 NSString 與 NSNumber，而 JSON 裡頭的 null 則會轉成 NSNull 物件。

NSNotFound

NSNotFound 所代表的是「找不到這個東西的 index」。比方說，我們有一個 array 是 @[@1, @2, @3]，當我們想要問在這個 array 中，@4 是第幾筆資料的時候（呼叫 `indexOfObject:`），因為 @4 並不在這個 array 裡頭，所以會回傳 NSNotFound，表示我們沒有在 array 裡頭找到想要的東西。

同樣的，如果我們在一個字串裡頭找不到某一段片段，像我們想在 @"KKBOX" 這個字串裡頭找某 @"a"，呼叫 `rangeOfString:` 的結果（[@"KKBOX" `rangeOfString:@"a"]】）是一個 NSRange，這個 range 的 location 也一樣是 NSNotFound。`

NSNotFound 是整數的最大值，我們通常不會建立這麼大的 array，所以用最大的整數代表找不到。我們要注意，在 64 位元與 32 位元下的的整數最大值是不一樣的，所以在 64 位元與 32 位元下，NSNotFound 代表的是不同的數字，64 位元下是 9223372036854775807（2 的 63 次方減一），32 位元下是 2147483647（2 的 31 次方減一）。

所以下面這段程式在 32 位元下正常，但是 64 位元環境下就有問題。

```
int x = @[@1, @2, @3] indexOfObject:@4];
if (x != NSNotFound) {
    NSLog(@"Found!");
}
```

在 32 位元環境下，我們用的是 32 位元整數，所以 x 會等於 NSNotFound；在 64 位元環境下，NSNotFound 是 64 位元整數最大值，但 x 被 cast 成 32 位元整數，所以 x 就無法等於 NSNotFound 了。

WebUndefined

在 Mac OS X 上，我們還有可能遇到另外一種代表「沒有東西」的物件，便是 WebUndefined。在 Mac 上使用 WebView 的時候，如果我們讓 WebView 裡頭的 JavaScript 來呼叫 Objective-C 或 Swift 的 native 實作，那麼，在 JavaScript 裡頭的 undefined 傳到 Objective-C 或 Swift 的 method 裡頭時，就會變成 WebUndefined 物件。

NULL、nil、Nil...

相關說明可以參閱 [JavaScript 與 Objective-C 的溝通](#) 這一節。

Responder

當我們在 iPhone、iPad 等 iOS 裝置上，用手指按到一個按鈕上的時候，事實上，我們並不是真的按在一個按鈕上，而是按在螢幕上—是觸控螢幕的硬體接收了我們的輸入之後，再將我們的觸控輸入送到軟體中，最後營造了「我們的手指按到了按鈕上」的幻覺。

從工程師的角度來看，所謂的 UI 都只是幻覺，「手指按到了螢幕中的按鈕」這件事情本身並不存在。而製作 UI，就是在製造幻覺；UI 設計，就是你打算營造怎樣的幻覺。iOS 7 之前，蘋果是透過漸層與逼真細膩但靜態的圖示製造光影的幻覺，在 iOS 7 之後，圖示變成扁平化設計，但是改用 motion effect 等動畫效果，創造深度的錯覺。

不同於其他的開發平台，在 iOS 與 Mac OS X 上，事件（Event）只用來表達來自硬體的各種輸入行為。在 iOS 上的 UIEvent 包含了觸控輸入、藍芽耳機遙控換歌等，Mac OS X 上的 NSEvent 則包括了鍵盤、滑鼠事件。

事件的傳遞

在 iOS 裝置上，當硬體發生觸控事件，到我們的按鈕發生反應之間，事實上經歷了：

- 硬體把事件傳到我們的 App 中，交由 UIApplication 物件分派事件
- UIApplication 把事件傳送到 Key Window 中，接著由 Key Window 負責分派事件
- Key Window 開始尋找在 View Hierarchy 中最上層的 view controller 與 view，然後，發現最上層的 view 是我們的按鈕
- 觸發按鈕的 target/action



事件從 application 傳遞到 window，從 window 傳遞到對應的 view 之上的流程，如果我們反過來看，就會變成「誰最後應該負責處理事件」—如果有個 view 該處理，就會是 view 處理，不然就會 fallback 到 window，window 不處理又會 fallback 到 application 上。

從 application 到 window 到 view，每一層中可以處理事件的物件，都叫做 responder，要實作 NSResponder 或 UIResponder protocol。回到我們會給一個專有名詞的習慣，所謂 responder，就是「可以處理事件的物件」。

在一堆可以處理事件的物件中，最後被分派到、把事件處理掉的物件，叫做 first responder，而這種一環又一環尋找誰該處理事件的鎖鏈，叫做 Responder Chain。

而這個流程，會在 runloop 當中不斷循環。

Run loop

我們來問一個很簡單的問題：我們所寫過的程式，大多都是從頭到尾、一行一行往下執行，執行完畢，程式就結束；那麼，一個 GUI 應用程式—無論是我們現在正在學習的 iOS 與 Mac OS X、還是其他平台—為什麼不是打開之後一路執行到底結束，而是會停留在螢幕中等待我們操作？

原因很簡單，因為一個 GUI 應用程式開始執行之後，就會不斷執行一個迴圈，直到用戶決定要關閉這個應用程式的時候，才會關閉這個迴圈。這樣的迴圈在 Windows 平台上叫做 message loop，在 iOS 與 Mac OS X 上叫做 run loop，而這個迴圈當中所做的，就是收取與分派事件。

每一輪 run loop 的時間並不固定，會與這一輪 run loop 裡頭做了多少事情相關，比方說，如果我們的畫面複雜，在 App 中同時有很多 view，那麼這一輪 runloop 就得要花上比較多的時間尋找 first responder；而像我們在 UI 上放了一個按鈕，然後按鈕按下去要做一些事情，全都會算入到這一輪 run loop 的時間。如果我們的程式做了一件很花時間的事情，讓這一輪 runloop 執行非常久，就會導致「應用程式介面沒有回應」這種狀態，當介面卡住一段時間，應用程式就會被系統強制關閉。

Timer 也是倚靠 run loop 運作的。當我們建立了一個 NSTimer 物件之後，下一步就是要把 timer 物件註冊到 run loop 當中，如果只建立了 NSTimer 物件，像是只呼叫了 `alloc`、`init`，這個 timer 並不會有作用，而呼叫 `scheduledTimerWithTimeInterval:target:selector:userInfo:repeats:` 會在建立 NSTimer 物件之外，同時將 timer 加入到 run loop 中。

Timer 運作的原理是，在每一輪 run loop 裡頭，會檢查是否已經到了某個 timer 所指定的時間，如果到了，就執行 timer 所指定的 selector。所以我們可以知道幾件事：

1. 由於每一輪 runloop 的時間不一定，所以我們其實也不能夠期待 timer 會在非常精確的時間執行。前一輪 runloop 如果做了很花時間的事情，就會影響到原本應該要執行的 timer 實際執行的時間。
2. 雖然並沒有所謂的最小的時間單位這件事情，但是 timer 的時間間隔一定會有一個上限，我們不可能建立比 run loop 的頻率還要更頻繁的 timer。

我們在講 selector 與記憶體管理的時候也提到，runloop 的其中一項功能還包含管理 auto-release 物件。Mac OS X 與 iOS 早期並沒有自動化的記憶體管理，當時會使用一套叫做 auto-release 的半自動機制方便管理記憶體，在每一輪 run loop 中，如果某些物件只有在這一輪 run loop 中有用，之後就應該釋放，我們可以先把物件放進 auto-release pool 裡頭，等到這一輪 run loop 的時候，再把 auto-release pool 倒空。

講到這裡，我們可以來談 iOS 與 Mac OS X 的程式進入點到底在哪裡。我們在寫第一個 iOS App 的時候，可能第一個改寫的地方是 `-application:didFinishLaunchingWithOptions:`，就以為這個 method 是 iOS App 的程式進入點。但 iOS App 的程式進入點其實就跟所有的 C 語言程式一樣，是 `main()`。我們來看 `main.m` 裡頭寫了什麼。

```
int main(int argc, char * argv[]) {
    @autoreleasepool {
        return UIApplicationMain(argc, argv, nil, NSStringFromClass([AppDelegate class]));
    }
}
```

在進入 `-application:didFinishLaunchingWithOptions:` 之前，會

1. 建立 auto-release pool

2. 呼叫 UIApplicationMain，而這個 function 會
3. 建立 UIApplication 這個 singleton 物件
4. 開始執行 run loop
5. 這些步驟完畢後，代表 app 已經開始執行，所以
6. 對 UIApplication 的 delegate 呼叫 `-application:didFinishLaunchingWithOptions:`

在 Cocoa 與 Cocoa Touch 應用程式中，我們會使用 CFRunLoop 與 NSRunLoop 等物件，描述 runloop。

Application

Mac OS X 與 iOS 上的 application 不太相同，在 Mac OS X 上是 NSApplication，在 iOS 上則是 UIApplication，但基本上都負責相同的工作：把來自外部的種種傳遞給內部，包括硬體事件，與其他各種系統事件。

硬體事件會被傳遞給 window，而其他系統事件，像 App被開啟或關閉、被推到前景或背景、收到 push notification...等等，則是會轉發給 application 的 delegate。

由於 application 位在 responder chain 的最底層，每一個 view 與 window 都不處理的時候，才會丟給 application 處理，所以如果我們希望處理一些會影響整個 App 行為的事件的時候，就適合由 application 這一層處理。

比方說，KKBOX 是一個音樂 App，所以我們會希望用戶可以透過藍芽耳機或線控耳機上的按鈕，切換 KKBOX 當中的歌曲，換到前一首或下一首歌曲。在 iOS 7.1 之前，我們要處理線控耳機的事件，會選擇實作 UIResponder protocol 中的 `-remoteControlReceivedWithEvent:`。因為換歌這件事情應該是對整個 KKBOX 的操作，無論放在哪個 view 或 view controller 都不適合，所以應該要放在 application 這一層。

要讓 application 這一層可以做額外的事情，我們首先要建立自己的 UIApplication subclass：

```
@interface KKApplication : UIApplication
@end
```

然後，在 main.m 裡頭，告訴 `UIApplicationMain`，我們應該要使用 KKApplication，而不是原本的 UIApplication 的實作：

```
int main(int argc, char * argv[]) {
    @autoreleasepool {
        return UIApplicationMain(argc, argv,
                               NSStringFromClass([KKApplication class]),
                               NSStringFromClass([AppDelegate class]));
    }
}
```

我們就可以在 KKApplication 處理事件了：

```
@implementation KKApplication
- (void)remoteControlReceivedWithEvent:(UIEvent *)theEvent
{
    if (theEvent.type == UIEventTypeRemoteControl) {
        switch(theEvent.subtype) {
            case UIEventSubtypeRemoteControlPlay:
                break;
            case UIEventSubtypeRemoteControlPause:
                break;
            case UIEventSubtypeRemoteControlStop:
                break;
            case UIEventSubtypeRemoteControlTogglePlayPause:
                break;
        }
    }
}
```

```

        case UIEventSubtypeRemoteControlNextTrack:
            break;
        case UIEventSubtypeRemoteControlPreviousTrack:
            break;
        ...
        default:
            return;
    }
}
@end

```

當然，如果我們想要開始接收來自耳機的事件，我們還要對 UIApplication 的 singleton 物件呼叫 `-beginReceivingRemoteControlEvents:`。

雖然跟 application 這一層無關，不過提到了耳機線控，就得提一下。蘋果在推出 iOS 7.1 的時候，同時推出了 Car Play 功能，Car Play 允許用戶在車用音響的介面上控制 iOS App，由於車用音響的畫面較大，所以，除了可以用來切換前後首歌曲之外，蘋果還加入了可以對歌曲評分，表示喜歡或不喜歡等功能，於是整個改寫了處理耳機線控的這一塊，推出 MPRemoteCommandCenter 這個 class。

從 MPRemoteCommandCenter 的 singleton 物件 `sharedCommandCenter` 上，我們可以拿到許多種不同的 MPRemoteCommand，然後對 MPRemoteCommand 設定 target/action。我們之前想要開始播放，會在 `-remoteControlReceivedWithEvent:` 裡頭處理 `UIEventSubtypeRemoteControlPlay`，現在會改成向 MPRemoteCommandCenter 要求 `playCommand`，然後指定 target/action，例如：

```
[[[MPRemoteCommandCenter sharedCommandCenter] playCommand addTarget:self action:@selector(play:)];
```

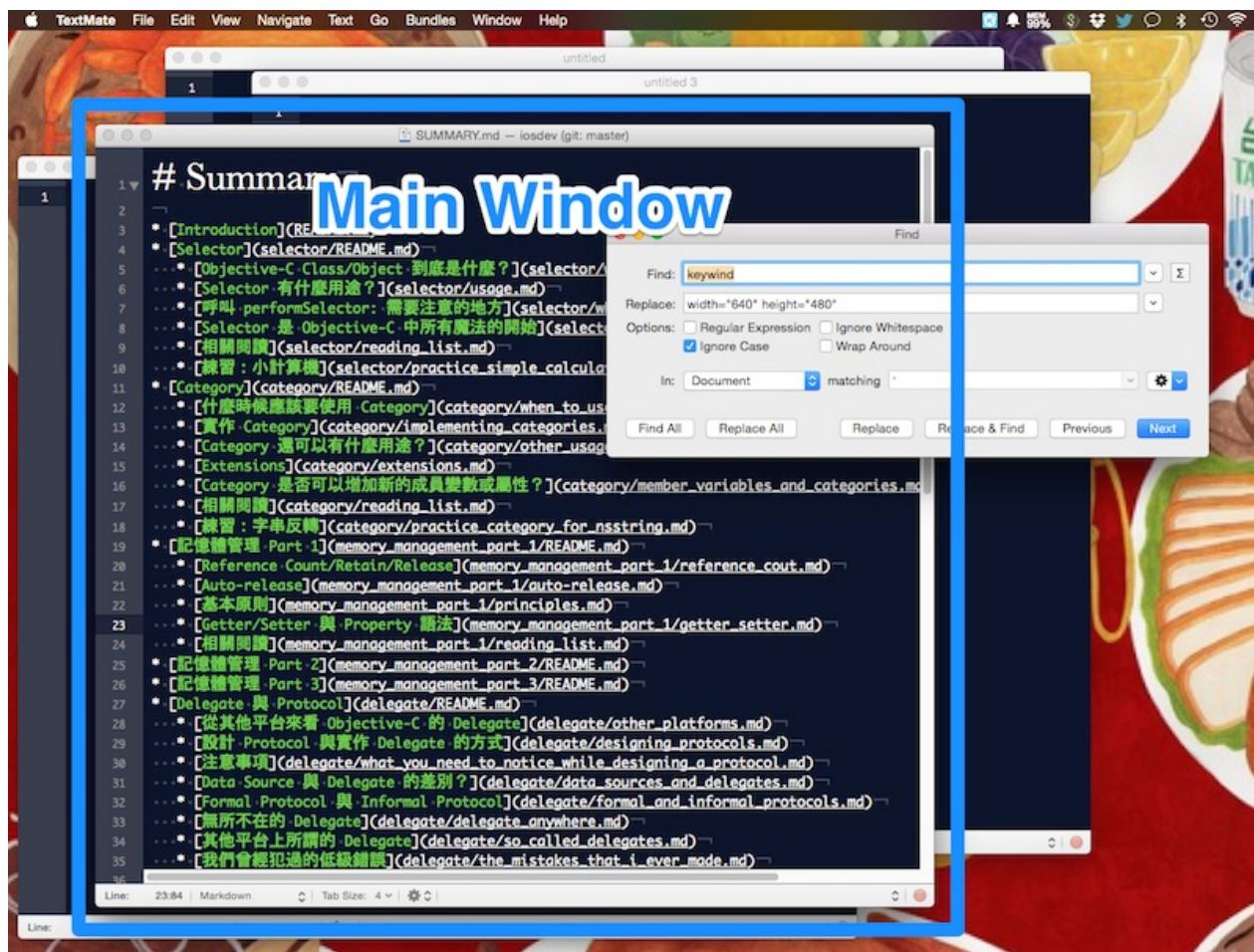
Window

Application 在收到觸控等硬體事件之後，會把事件轉發給 key window。如果你 iOS SDK 問世的前幾年就在寫 iOS App，應該會注意到，Xcode 幫你建立專案時使用的 template 中，可能就有一段像下面這樣的 `-application:didFinishLaunchingWithOptions` 的實作：

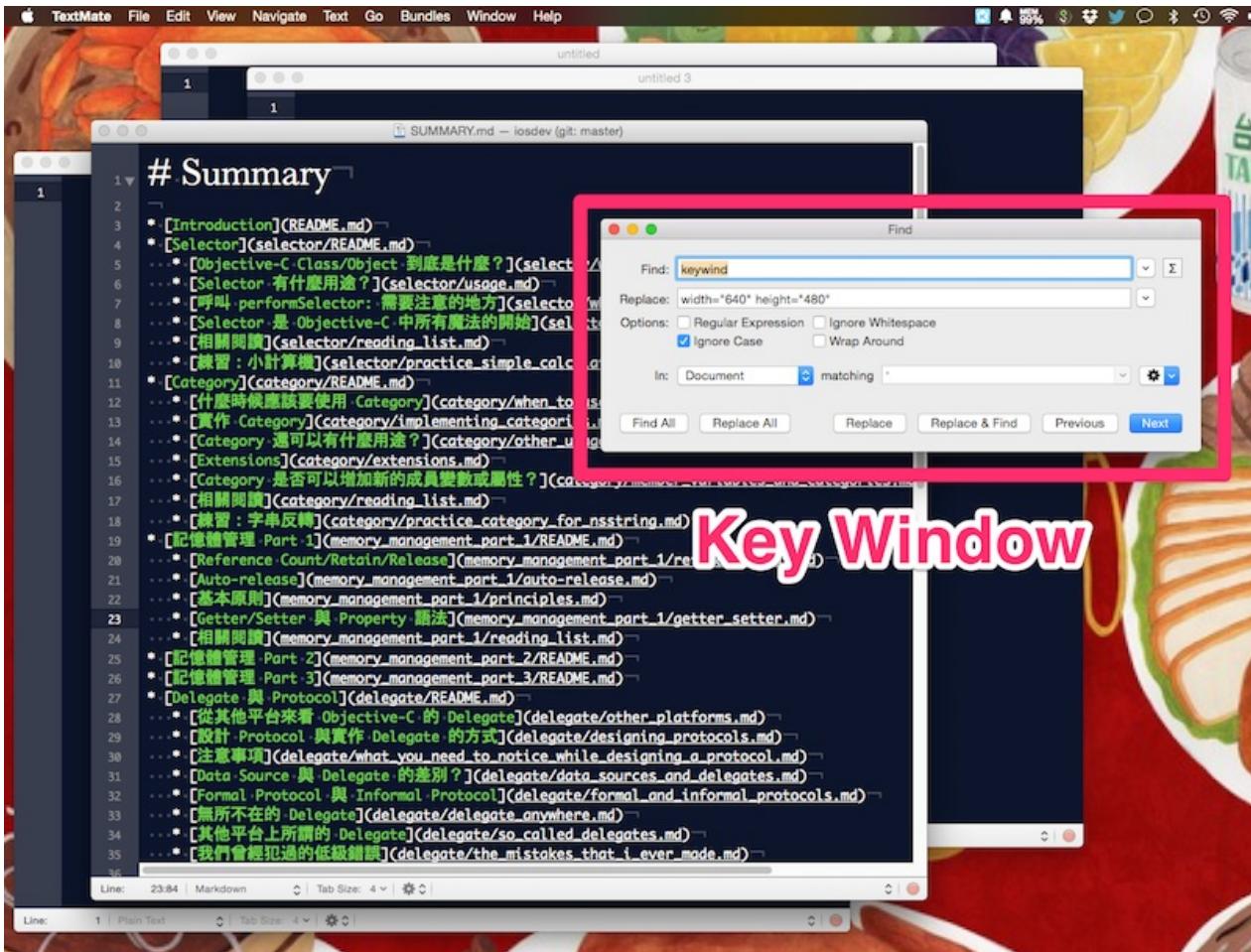
```
- (BOOL)application:(UIApplication *)application  
didFinishLaunchingWithOptions:(NSDictionary *)launchOptions  
{  
    self.window = [[UIWindow alloc] initWithFrame:[UIScreen mainScreen].bounds];  
    ViewController *controller = [[ViewController alloc] init];  
    self.navigationController = [[UINavigationController alloc] initWithRootViewController:  
                           controller];  
    window.rootViewController = self.navigationController;  
    [window makeKeyAndVisible];  
    return YES;  
}
```

當一個 window 被呼叫了 `-makeKeyAndVisible` 之後，就會變成 key window。但這樣的解釋還是很模糊—到底什麼是 key window？要回答這個問題，我們要先離開一下 iOS SDK，來看看 Mac。

在 Mac 上，一個 window 除了可以是 key window 之外，還可以成為 main window。比方說，TextMate 這套文字編輯器是一套 Document-based App，我們可以同時開很多 window，每個 window 裡頭都代表一份文字檔案，但是，我們一次也只能在一個 window 當中打字，所以，只有一個 window 的邊框會是比較深的顏色，其他 window 的邊框則是淺色，代表這些 window 沒有作用。唯一作用中的 window 就是 main window。



但假如我們現在叫出了 TextMate 的尋找功能，想要在目前正在編輯的文字檔當中，尋找指定的文字，這時候鍵盤焦點會放在搜尋 window 的關鍵字欄位上。現在搜尋功能 window 就是 key window。Key window 所謂的 key，其實就是鍵盤 keyboard 的 key，你在哪個 window 打字，那個 window 就是 key window。而由於我們是對現在正在編輯的文件做搜尋，所以我們在搜尋 window 這個 key window 打字之後，會再去尋找 main window，把進行搜尋這件工作送往 main window。



很多時候 main window 與 key window 會是同一個。比方說，我們找到了關鍵字，離開了搜尋 window，回到主要文件 window 打字，那麼，這份文字檔的 window 就同時是 main window 與 key window。

在 iOS 上，由於每個 App 主要只有一個處理事件的 window，因此沒有需要區分 main window 與 key window，因此 SDK 的設計上就只有 key window，而沒有 main window。

其實 iOS App 可以擁有多個 window。在蘋果推出 AirPlay 功能，讓 iOS App 的畫面可以投射到 AppleTV 之後，除了兩邊完全鏡像之外，我們也可以選擇在 AppleTV 的畫面上顯示不一樣的內容，像 Keynote 這套 App 在連接到 AppleTV 之後，可以選擇 AppleTV 上是目前所在的簡報頁面，iOS 裝置上則顯示下一頁投影片方便提詞，實作方式就是建立第二個 Window，然後選擇把第二個 window 放在 AppleTV 上。但是在 AirPlay 的狀況下，AppleTV 的畫面只有顯示的功能而已，我們並不會在這個 window 中打字。

在另外一個情境下，我們也可能會想要建立自己的 window。在不少 App 中，其實有使用一項 UI 設計：在畫面上方的 status bar 部分，額外顯示提示訊息，使用 status bar message 當關鍵字在 github 上搜尋，就可以看到至少有 [MTStatusBarOverlay](#)、[FDStatusBarNotifierView](#)、[JDStatusBarNotification](#) 這麼多個專案。

其實 iOS 的 status bar 本身就是一個 window，而這個window 的 level 會比我們 App 的 window 高，於是 status bar 會一直疊在我們的 App 畫面上方，不論我們對原本的 window 怎麼增加 subview，都不會影響 status bar。如果我們想要蓋過 status bar，方法就只有建立新的 window，level 設得比 status bar window 更高，然後呼叫 `makeKeyAndOrderFront`，讓這個 window 出現在螢幕上。

這時候我們要千萬注意：`makeKeyAndOrderFront` 不但會讓這個 window 出現，同時也變成 key window，所有的事件都會往這個 window 送。所以，如果提示訊息結束，我們必須要對原本的 key window 再呼叫一次 `makeKeyAndOrderFront`，把處理事件的權力交還回去，不然接下來我們的 App 就無法處理任何事件了。

當我們在 iOS App 的一個畫面上，如果有很多文字框（UITextField） ，我們想要指定其中一個文字框拿來打字，會呼叫 `becomeFirstResponder` ，讓這個文字框變成 first responder，具體來說的意義是「這個 window 上的 first responder」。這點在 Mac 的 API 上會比較清楚，如果在 Mac 的一個 window 中要指定某個文字框打字，我們不是對 NSTextField 呼叫，而是對 window 呼叫 `makeFirstResponder:` ，然後傳入 text field。

View

Application 透過 `-sendEvent:` 將事件送到 window，window 也一樣透過 `-sendEvent:` 把事件送到 view 上，而在 view 裡頭，則是透過 `-hitTest:withEvent:`，在一層又一層的 view hierarchy 裡頭尋找應該處理事件的 sub view。

比方說，我們現在有個 view，裡頭有兩個 subview：button A 與 button B，在 `-hitTest:withEvent:` 這個 method 裡頭，就會先去判斷觸控事件的座標是否在這兩個 subview 裡頭，如果觸控座標在 button A 的 frame 裡頭，那就該由 button A 來處理事件，`-hitTest:withEvent:` 就會回傳 button A。

但 `-hitTest:withEvent:` 不但會問觸控事件的座標是否在某個 subview 裡，也會問這個 subview 是否打算處理事件。假如反過來，我們現在有個 button，而裡頭有兩個 subview，一個是文字 label、一個是個 image view，這兩個 view 都只負責處理顯示內容，而不處理事件，那麼，這個 button 裡頭的 `-hitTest:withEvent:` 最後就會把自己回傳回去，表示是由這個 button 自己處理事件，而不交由任何的 subview 處理。

有時候我們會藉由修改 `-hitTest:withEvent:` 達成一些稀奇古怪的效果，例如，我們現在有一個 view，裡頭有一個 scroll view，scroll view 只佔他的super view 的一部分而已，但是我們希望在這個 view 裡頭的任何地方點選，都可以捲動裡頭的 scroll view。我們這個時候就可以考慮改寫 `-hitTest:withEvent:`，不管觸控的位置是否真的在 scroll view 裡頭，`-hitTest:withEvent:` 都回傳這個 scroll view。

當事件從 application、window 傳遞到 view 之後，就會成為一些我們之前就已經熟悉的 API，在 Mac 上會觸發 `-keyDown:`、`-mouseDown:` 這些在 NSView 裡頭的 method，至於在 iOS 上，觸控事件傳遞到 view 之後就會觸發一系列跟觸控事件有關的 method：

- `- (void)touchesBegan:(NSSet *)touches withEvent:(UIEvent *)event`
- `- (void)touchesMoved:(NSSet *)touches withEvent:(UIEvent *)event`
- `- (void)touchesEnded:(NSSet *)touches withEvent:(UIEvent *)event`
- `- (void)touchesCancelled:(NSSet *)touches withEvent:(UIEvent *)event`

從這些 method 中，我們會知道觸控事件是否開始、結束、中間手指是否移動，用戶在螢幕硬體上到底用了幾支手指，每根手指的位置又如何變化，進而知道用戶到底做了怎樣的手勢：是指用一根手指做了點選（tap）還是橫劃（swipe）、還是用了兩根以上的手製作縮放（pinch），甚至是更複雜的手勢。在蘋果官方文件 [Multitouch Events](#) 當中，就說明了怎樣用上面這些 UIResponder method，處理各些複雜手勢。

要了解事件是怎樣從 Application 一路送到 View，最簡單的方法，就是寫一個簡單的 App：這個 App 只有一個按鈕，然後我們在這個按鈕的 action 打上一個 break point，然後來看一下這個時候的 back trace：

Network	Zero KB/s
Thread 1 Queue: com.ap...ain-thread (serial)	
0 -[ViewController test:]	
1 -[UIApplication sendAction:to:from:forEvent:]	
2 -[UIControl sendAction:to:forEvent:]	
3 -[UIControl _sendActionsForEvents:withSender:]	
4 -[UIControl touchesEnded:withEvent:]	
5 -[UIWindow _sendTouchesForEvent:]	
6 -[UIWindow sendEvent:]	
7 -[UIApplication sendEvent:]	
8 -[UIApplicationAccessibility sendEvent:]	
9 _UIApplicationHandleEventQueue	
10 __CFRUNLOOP_IS_CALLING_OUT_TO_A_SOURCE_IN_AN_OBSERVER_CALLBACK_FUNCTION	
11 __CFRunLoopDoSources0	
12 __CFRunLoopRun	
13 CFRunLoopRunSpecific	
14 GSEventRunModal	
15 UIApplicationMain	
16 main	
17 start	
Thread 2 Queue: com.ap...h-manager (serial)	
Thread 7	
Thread 8	
Thread 9	

```

8
9 #import "ViewController.h"
10
11 @interface ViewController : UIViewController
12 @end
13
14 @implementation ViewController
15
16 - (void)viewDidLoad {
17     [super viewDidLoad];
18     // Do any additional setup after loading the view.
19 }
20
21 - (void) didReceiveMemoryWarning {
22     [super didReceiveMemoryWarning];
23     // Dispose of any resources that can be recreated.
24 }
25
26
27 - (IBAction)test:(id)sender {
28 {
29 NSLog(@"hi");
30 }
31
32 @end
33

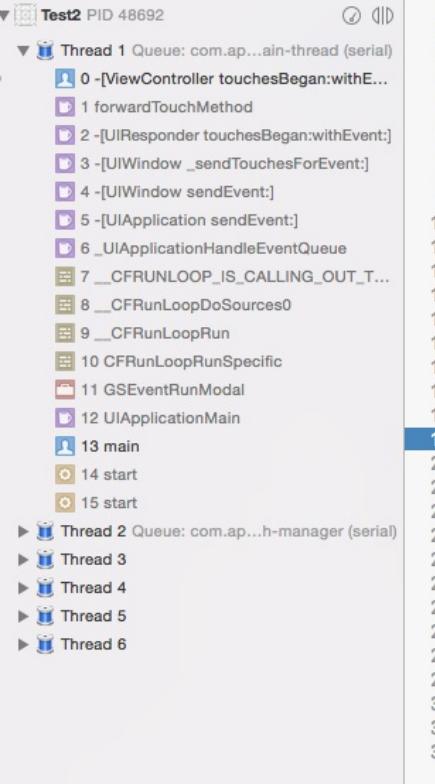
```

- frame 16 是 `main()`，裡頭執行 `UIApplicationMain`
- frame 10-14 便是在執行 Runloop
- 在 frame 7-9，可以看到 Runloop 把事件送給 `UIApplication`
- 在 frame 5-6，可以看到 `UIWindow` 在處理來自 `UIApplication` 的事件
- frame 4 在處理我們剛剛講到的 `touchesEnded:withEvent:`
- 最後在 frame 0-3，便是在處理這個按鈕上的 target/action

View Controller

View Controller 本身也是個 responder，因此也實作了 UIResponder protocol。當觸控事件發生的時候，如果某個 view controller 的 view 都不處理傳來的 UIEvent，那麼就會轉向詢問這個 view 的 view controller 本身是否處理這個事件。

在下面的例子裡，我們寫了一個 view controller，但是 view 裡頭沒有任何可以點按的物件，而 view controller 本身實作了 -touchesBegan:withEvent: 等 method。



```

1 // ViewController.m
2 // Test2
3 // Created by zonble on 10/27/15.
4 // Copyright © 2015 KKBOX Taiwan Co., Ltd. All rights reserved.
5 //
6 #import "ViewController.h"
7
8 @interface ViewController : UIViewController
9 @end
10
11 @implementation ViewController
12
13 - (void)touchesBegan:(NSSet *)touches withEvent:(UIEvent *)event
14 {
15     NSLog(@"hi"); Thread 1: break
16 }
17 - (void)touchesMoved:(NSSet *)touches withEvent:(UIEvent *)event
18 {
19 }
20 - (void)touchesEnded:(NSSet *)touches withEvent:(UIEvent *)event
21 {
22 }
23 - (void)touchesCancelled:(NSSet *)touches withEvent:(UIEvent *)event
24 {
25 }
26
27
28
29
30
31 @end
32

```

可以從這邊的 back trace 看到，來自 UIWindow 的觸控事件，被 forwardTouchMethod 這個 function，轉發到 view controller 上了。

UITouch

在觸控事件發生之後，我們會從 UIEvent 中，收到代表觸控事件的 UITouch 物件。

UITouch 起初是個非常單純的物件，我們頂多只會使用 `-locationInView:` 判斷觸控事件發生在 view 的哪個位置上 (CGPoint) ，以及用 `-tapCount` 知道碰觸了幾下，以及用 `-timestamp` 知道觸控事件的時間。

但隨著 iOS 不斷演進，UIEvent 與 UITouch 也變得愈來愈複雜，尤其是在 iOS 9 推出之後，突然一次出現了非常多新的 API。

觸控螢幕的掃描速率

在 iOS 9 之後，增加了 `coalescedTouchesForTouch` 這個屬性，主要原因是 iPad Air 2 硬體效能的提升。

當我們看到 `touchesBegan:withEvent:` 這些 API，大概可以想到，我們會在每一輪 runloop 收到一次 touch 物件，所以觸控螢幕在接收 touch 物件的速度，大概會是跟 runloop 的速度差不多。於是，當開發者在開發一些在 iPad 上的繪圖應用的時候，往往就覺得在 iPad 上會有延遲的現象：手指已經在螢幕上畫過去了，但是畫面卻是慢慢地更新。

蘋果在 iPad Air 2 的觸控螢幕上加快了對觸控事件的掃描頻率，比起之前的 iPad 快上一倍，但是，run loop 的速度並沒有改變，因此，在 iOS 9 的 API 中，會把這些比以前來得多的觸控事件，變成 UIEvent 物件的 `coalescedTouchesForTouch:` method，在這個 method 中，可以拿到更多的 UITouch，可以讓我們在搭配 iOS 9 的 iPad Air 2 上抓到更多 touch 物件，繪製更精細的線條。

UIEvent 同時也多了一組叫做 `predictedTouchesForTouch:` 的 method，預測下一個觸控事件可能出現的位置，因此，即使這個觸控事件還沒有發生，但我們便可以偷吃步先去做繪圖相關的工作，讓畫面看起來即時更新。

相關說明請參見 WWDC 2015 影片 [WWDC 2015 Advanced Touch Input on iOS](#)。

3D Touch

蘋果在 iPhone 6S 上加入了 3D Touch 功能，除了在 App 這層加了 shortcut、在 view controller 這層加入了 peek and pop 功能（實作 UIViewControllerPreviewingDelegate protocol）之外，便是 UITouch 物件本身也加入了 `force` 與 `maximumPossibleForce` 等屬性，用來判斷觸控的力道。

在 iOS 9 與 iPhone 6S 上，`touchesMoved:withEvent:` 的行為也發生了變化，原本只有 Touch 事件的 X 軸或 Y 軸有改變的時候，系統才會觸發 `touchesMoved:withEvent:`，但是在有了 3D Touch 之後，觸控壓力的改變，也會觸發 `touchesMoved:withEvent:`，換言之，這個 method 被呼叫的時候，我們不能夠假設用戶的手指真的移動了位置，很有可能只是壓力的改變而已。要知道手指的位置是否真的移動了，我們需要另外比對 UITouch 物件的 X 軸或 Y 軸的位置，不然就可能會把單點誤判成 Swipe。這部份說明請參見 [iOS 9.1 Release Note](#)。

當然，如果只是要知道用戶是否單點在一個位置上，用 UIGestureRecognizer 還是比較簡單，也是比較保險的方法。

Apple Pencil

在 iOS 9 推出的同時，Apple 同時宣布了 iPad Pro 這條產品線，在 iPad Pro 上可以使用 Apple Pencil 這款輸入裝置做更精密的手寫。於是，蘋果在 iOS 9.1 SDK 中增加不少與 Apple Pencil 相關的 API。摘錄 iOS 9.1 SDK 的 release note 部分如下：

- UITouch 增加了 `type` 屬性，用來判斷這個觸控事件是來自於直接、間接的觸控，或是來自於 Apple Pencil
- UITouch 物件原本就有 `-locationInView:` 與 `-previousLocationInView:` 這兩個 method 表示觸控發生的位置，在使用 Apple Pencil 的時候，我們可以透過 `-preciseLocationInView:` 與 `precisePreviousLocationInView:` 這兩個 method，知道更精細的觸控位置。
- `altitudeAngle`、`azimuthAngleInView:` 與 `azimuthUnitVectorInView:` 可以讓你知道 Apple Pencil 的高度與方位。
- UIEvent 中的 `predictedTouchesForTouch:` method 只預測了下一個觸控事件可能的位置，但有了 Apple Pencil 之後，我們還會想要預測接下來 Apple Pencil 的高度、方位與壓力等資訊。我們便可以透過 `estimatedProperties` 與 `estimatedPropertiesExpectingUpdates` 這些屬性取得。

相關閱讀

- [Event Handling Guide for iOS](#)
- [UIResponder Class Reference](#)
- [UIEvent Class Reference](#)
- [NSResponder Class Reference](#)
- [NSEvent Class Reference](#)
- [NSRunLoop Class Reference](#)
- [Technical Q&A QA1693 Synchronous Networking On The Main Thread](#)
- [WWDC 2015 Advanced Touch Input on iOS](#)

Threading

在前一章提到，iOS 有一項系統限制：如果我們的程式中某個操作超過一定時間，那麼系統就會認為我們的應用程式沒有回應，而會強制關閉我們的 App，當 App 被強制關閉的時候，外觀上會非常像是當機。如果是在 Mac 上，就會看到滑鼠圖示變成沙灘球不斷旋轉，最後 App 一樣被強制關閉。

但是我們的 App 往往會有許多需要花時間的操作，像是去網路上抓取資料，或是 local 的檔案處理。所以，在這些場合，我們需要將工作丟到背景 thread 執行，在工作完成之後、或是需要更新進度的時機，才告訴 main thread 更新 UI。

Thread 通常翻譯成「線程」或是「執行緒」，也就是在同一個 process（也就是同一個 app 中），會同時存在、進行多條的程式執行路徑，每條執行路徑之間，不用等到某條執行路徑結束，另外一條執行路徑才能開始。作業系統會安排某條 thread 在 CPU 的某個核心上執行，或是會先打斷某條 thread，讓其他的 thread 先執行。

以網路連線來說，我們會避免使用 NSData 或 NSString 的 `-initWithContentsOfURL:` 這個 API，而使用 NSURLSession 或 NSURLConnection 發送非同步的連線，NSURLSession 與 NSURLConnection 在做的事情，便是將抓取資料這件工作放在其他 thread 中執行，然後在必要的時候 callback — 在這邊我們要順道注意一下，其實像 NSURLSession 的 data task 的 callback block，也是在背景 thread 中執行。

在 iOS 與 Mac OS X 上，我們可以呼叫低階的 POSIX thread，不過既然有比較高階的 API，我們自然會選擇使用高階 API。我們通常在 iOS 與 Mac OS X 上使用三種方式處理 Multi-thread 的問題，分別是：

- Perform Selector
- GCD (Grand Central Dispatch)
- NSOperation 與 NSOperationQueue

Perform Selector

NSOperation 與 NSOperationQueue 是在 Mac OS X 10.5/ iOS 2.0 的時候推出的功能，GCD 則是在 Mac OS X 10.6/iOS 4 的時候推出，在這之前，使用 NSObject 的 performSelector 系列的 API 會是處理 threading 時比較容易的作法。就像我們之前在 Selector 這一章提到的，我們可以使用以下這些 API，將某些工作放在指定的 Thread 執行：

- -performSelectorOnMainThread:withObject:waitUntilDone:modes:
- -performSelectorOnMainThread:withObject:waitUntilDone:
- -performSelector:onThread:withObject:waitUntilDone:modes:
- -performSelector:onThread:withObject:waitUntilDone:
- -performSelectorInBackground:withObject:

作為比較早期的 API，和 GCD/NSOperationQueue 比較起來，這幾組 API 的最大缺點便在於不會幫你管理應該要建立多少 thread，全部都得要自己手動管理。一台機器有其硬體效能的上限，能夠開出多少 thread 也有其上限，如果我們建立超過硬體效能上限數量的 thread，最糟的狀況會造成整台機器癱瘓。

假如我們現在有一百份工作要在背景執行，如果我們使用 GCD 或是 NSOperationQueue，我們可以把這些工作寫成 block 或是 NSOperation，然後丟到 GCD 或 NSOperationQueue 中，GCD 或 NSOperationQueue 會根據目前的硬體效能決定 Thread 的數量，假如目前使用的機器只適合建立三條背景 thread，那麼，就只會建立三條背景 thread，然後將這一百份工作分批放在這三條 thread 中執行。

但，如果你寫了一個迴圈，在迴圈中呼叫了一百次 -performSelectorInBackground:withObject:，那麼，就真的會建立一百個 thread！如果我們想要避免這種狀況，那麼，在使用 -performSelectorInBackground:withObject: 時，就有必要自己寫一個 queue，將工作指派在特定的 thread 上。那麼，還是使用 GCD 或是 NSOperationQueue 會比較容易些。

在呼叫 -performSelector:onThread:withObject:waitUntilDone:modes: 與 -performSelector:onThread:withObject:waitUntilDone: 的時候，我們要特別注意一下傳入的 NSThread 物件。假如我們在 thread 這個參數傳遞 nil，那麼，就會造成程式卡在這一行，而不會繼續執行。

另外一個明顯的缺點則是，使用這些 API 的時候，一次只能夠傳遞一個參數，所以，當你有一件工作想要放在指定的 thread 執行的，又必須要傳遞多個參數的時候，就必須將這些參數包裝成 NSArray 或 NSDictionary 的物件。

如果你不想寫一個專屬用來傳遞參數的 Class，只想要傳遞一個 NSArray 過去，而你有一些 C primitive type 的參數，像是數字、指標或 C structure，為了要能夠插入到 NSArray 中，就得要先轉換成對應的 NSValue，在我們要執行的 selector 中再從 NSValue 轉換回來，於是會寫不少轉換用的 code。

另外要注意，如果我們要在背景執行一個 selector，這個 method 裡頭必須要有自己的 auto release pool，才能夠正確釋放 auto release 物件（關於auto release 請參見 [記憶體管理 Part 1](#)）。要建立 auto release pool，可以手動建立 NSAutoreleasePool 物件，也可以使用 @autoreleasepool 關鍵字，當然用 @autoreleasepool 會比較容易一點，而且在啟用 ARC 之後，也會禁止手動建立 NSAutoreleasePool，只能使用 @autoreleasepool。像這樣：

```
- (void)backgroundTask
{
    @autoreleasepool {
        // Write your code here.
    }
}
```

performSelector:

```
    }  
}
```

GCD (Grand Central Dispatch)

如果我們有一件工作，想要在某條指定的 thread 上執行，現在最簡單的方法大概就是呼叫 GCD。GCD 其實包含相當多的 API，是一群 C function 的組合，其中，我們最常用的是 `dispatch_async`。

`dispatch_async`

`dispatch_async` 這個 function，可以讓我們選擇要在哪個指定的 thread 上，用非同步的方式執行一個 block。比方說，我們現在在前景，但是想要在背景執行一件工作，就會這麼寫：

```
dispatch_async(dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0), ^{
    [someObject doSomethingHere];
});
```

`dispatch_get_global_queue` 這個 function 會讓系統根據目前的狀況，在適當時機建立一條 thread，第一個參數是這條 thread 執行工作的優先程度，優先程度會從 2 到 -2 安排，2 為最重要，-2 為最不重要；至於第二個參數則是保留參數，目前都沒有作用，直接填 0 即可。

如果我們已經在背景了，想要在 main thread 執行工作，那麼，就把 `dispatch_get_global_queue` 換成 `dispatch_get_main_queue`

```
dispatch_async(dispatch_get_main_queue(), ^{
    [someObject doSomethingHere];
});
```

我們經常會先讓某個工作在背景執行，執行完畢之後，再繼續在 main thread 更新 UI，讓用戶知道這件工作已經執行完畢，我們便可以組合前面兩個呼叫：

```
dispatch_async(dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0), ^{
    [someObject doSomethingInBackground];
    dispatch_async(dispatch_get_main_queue(), ^{
        [someObject doSomethingOnMainThread];
    });
});
```

如果我們想要讓好幾件工作都在背景執行，而每件工作並非平行執行，而是一件工作做完之後，再繼續下一件，我們便可以使用 serial 的 queue。像這樣：

```
dispatch_queue_t serialQueue = \
    dispatch_queue_create("com.kkbox.queue", DISPATCH_QUEUE_SERIAL);

dispatch_async(serialQueue, ^{
    [someObject doSomethingHere];
});

dispatch_async(serialQueue, ^{
    [someObject doSomethingHereAsWell];
```

```
});
```

dispatch_sync

不同於 `dispatch_async` 會做平行處理，呼叫 `dispatch_sync` 的時候，則是會先把 `dispatch_sync` 的這個 block 做完之後，才繼續執行到程式的下一行。

我們在呼叫 `dispatch_sync` 的時候要特別注意：如果我們已經在某一條 thread 中，而呼叫 `dispatch_sync` 時所傳入的 thread 就是目前所在的 thread，那麼會造成程式執行時卡死。比方說，我們已經在 main thread 了，但我們卻呼叫：

```
dispatch_sync(dispatch_get_main_queue(), ^{
    [someObject doSomethingHere];
});
```

這段程式就會卡住。我們可以用 `NSThread` 的一些 method 檢查我們目前正在哪條 thread，例如使用 `+isMainThread` 檢查是否是 main thread。

其他一些好用的 API

和 `dispatch_async` 與 `dispatch_sync` 相較，底下這些 API 會比較少用，但是可以解決不少麻煩的問題。

dispatch_once

`dispatch_once` 保證某個 block 只會被執行一次，現在大家最常使用這個特性實作 singleton。我們在「[再談 Singleton](#)」這一章當中也提過。

dispatch_after

可以延後執行某個 block 在某個指定的 dispatch queue 上執行，我們可以用這個 function 代替 timer。

```
dispatch_after(dispatch_time(DISPATCH_TIME_NOW, (int64_t)(2 * NSEC_PER_SEC)),
    dispatch_get_main_queue(), ^{
    [someObject doSomething];
});
```

dispatch_apply

如果我們想要重複執行某個 block，就可以考慮使用 `dispatch_apply`。`dispatch_apply` 有三個參數，第一個參數是要執行的次數，第二個參數則是要在哪個 dispatch queue 上執行：就像前面提到的，如果想要平行執行，就呼叫 `dispatch_get_global_queue`，如果想要依序執行，就建立一個 serial 的 dispatch queue。

NSOperation 與 NSOperationQueue

GCD 雖然好用，但是 GCD 的介面讓我們不太容易取消已經排程、或是已經在背景執行的作業，如果我們有中途取消某個背景作業的需要，使用 NSOperation 與 NSOperation Queue 會是更好的選擇。

NSOperation 是一個用來封裝一項作業的 Objective-C class，這樣的物件稱為 operation，在建立了一個 operation 後，便可以將 operation 丟入 opeation queue（型別為 NSOperationQueue）中排程，讓 opeation queue 決定在適當的時機、系統可以負荷的狀況下，執行我們排入排程的工作。

無論是 NSOperationQueue 與 NSOperation，都有與取消工作相關的 API 可以呼叫。我們也可以設定每個 opeation 的優先程度，以及不同 operation 之間的相依關係（dependency），要求一件工作完成之後，才可以繼續下一件工作。

建立 NSOperationQueue

假如我們有一個 Class，裡頭有一個 operation queue，只要呼叫 `alloc` 與 `init` 便可以建立。

```
#import <Foundation/Foundation.h>

@interface Test : NSObject
@property (nonatomic, strong) NSOperationQueue *queue;
@end

@implementation Test

- (instancetype)init
{
    self = [super init];
    if (self) {
        self.queue = [[NSOperationQueue alloc] init];
        self.queue.maxConcurrentOperationCount = 2;
    }
    return self;
}

@end
```

我們可以透過 `maxConcurrentOperationCount` 這個 property，設定 NSOperationQueue 可以同時平行執行幾件工作，如果超過 1，就代表允許平行執行，如果剛好是 1 的話，就代表在這個 queue 當中的所有工作都會依次執行。預設值是 -1（`NSOperationQueueDefaultMaxConcurrentOperationCount`），意思是讓系統自己決定最多可以同時建立多少 thread。

我們可以對 NSOperationQueue 呼叫 `addOperation:` 加入 operation，用 `cancelAllOperations` 取消所有排程中的作業。至於已經在執行中的作業，我們就得對特定的 operation 呼叫 `cancel` 了。

建立 NSOperation

在 Cocoa/Cocoa Touch Framework 中，已經存在兩個 NSOperation 的 subclass：NSBlockOperation 與 NSInvocationOperation。NSBlockOperation 可以讓你把一個 block 封裝成 NSOperation，至於 NSInvocationOperation 則是用來封裝 NSInvocation。

一般來說，除非是像前面說的，你希望這些工作在排程中就可以取消，或是要特別指定 operation 之間的相依關係，不然，要在背景執行某個 block 或是 invocation，其實使用 GCD API 會更容易。

我們可能會更常建立自己的 NSOperation subclass，處理更複雜的背景工作。

比方說，我們現在要開發一套食譜 App，這套 App 可以讓用戶在本機的編輯介面中編好一份食譜後上傳，上傳後要清除本機的暫存檔，這份食譜可能會包含一份包含標題、內文的 JSON 檔案，還有一張圖片，所以上傳食譜這份工作就包含上傳 JSON 文件與圖片兩件工作，而我們也希望可以在上傳的過程中隨時取消，讓用戶繼續編輯再重新上傳—這種比較複雜卻又帶有次序性質的工作，就是很適合 NSOperation 的舞台。

要 subclass 一個 NSOperation，最重要的就是要 override 掉 main 這個 method，main 這個 method 裡頭代表的是這個 operation 要做什麼事情。我們現在可以來寫我們的 operation：

```
@interface RecipetUploadOperation : NSOperation
@property (nonatomic, strong) UIImage *image;
@property (nonatomic, strong) NSString *JSON;
@end

@implementation RecipetUploadOperation
- (void)main
{
    @autoreleasepool {
        // 1. Upload image
        // 2. Upload JSON
    }
}
@end
```

在 main 裡頭，我們也要建立 auto release pool。

接下來我們會遇到一個問題：在上傳照片與 JSON 檔案的時候，我們會呼叫 NSURLSession 的相關 API，這些 API 都是非同步的，但是在 main 這個 method 裡頭，如果不做特別的處理，還沒等到連線回應，main 就已經執行結束了。我們必須要想辦法停在 main 中，等待連線 API 的回應。

在 Operation 中等待與取消

要在 operation 的中途停下來等候回應，我們大致上有兩種作法，一種是在 operation 當中執行 NSRunLoop，另外一種則是使用 GCD 當中的 semaphore。

NSRunLoop

在有 GCD 之前，我們希望一個 operation 可以在一個地方停下來等候其他事情發生，作法會是在這條 thread 裡頭執行 run loop。

前一章提到，run loop 就是那個「之所以 GUI 程式會一直執行，而不會像某個 function 或 method 從頭到尾跑完就結束」的迴圈。在 iOS 或 Mac OS X App 中，除了在 main thread 會執行最主要的 run loop (`[NSRunLoop mainRunLoop]`) 之外，每個thread/operation 裡頭，也會有屬於各自的 run loop，只要呼叫 `[NSRunLoop currentRunLoop]`，呼叫的就是屬於當前 thread 自己的 run loop—所以我們要注意，雖然在不同的 thread 中，我們呼叫的都是 `[NSRunLoop currentRunLoop]`，但這個 `+currentRunLoop` 這個 class method 回傳的並不視同一個物件。另外，`NSRunLoop` 不可以手動建立，我們只能使用系統提供的 run loop 物件。

我們希望能夠在這個 operation 執行到一半的時候可以被取消，要取消一條 operation，便是呼叫 `NSOperation` 的 `cancel` 這個 method，因為我們 subclass 了 `NSOperation`，改變了 operation 裡頭做的事情，那麼也就得 override 掉 `cancel`：當我們的 operation 在跑 run loop 時，我們的 `cancel` 必須要能夠通知 run loop 停止。

當一條 thread 在跑自己的 run loop 之後，如果不同 thread 之間想要互相溝通，那我們就必須在當前的 thread 建立 `NSPort` 物件，並且將 `NSPort` 物件註冊到 run loop 內，才能讓訊息傳遞到 run loop 裡頭。所以，當外部要求對 port 呼叫 `invalidate` 的時候，就會讓 run loop 收到訊息，停止繼續跑，繼續執行 `-main` 這個 method 接下來的動作。

`NSPort` 也有對應的 Core Foundation 實作，像 `CFMessagePort` 等，不過在 iOS 7 之後我們沒辦法在這個地方使用 `CFMessagePort`。從 iOS 7 之後，呼叫 `CFMessagePortCreateLocal` 或 `CFMessagePortCreateRemote` 這些建立 `CFMessagePort` 的 function 都無法建立物件，只會回傳 NULL（可以參見 `CFMessagePort` 的 reference），蘋果不允許我們使用 `CFMessagePort` 的原因是，`CFMessagePort` 不但可以傳遞訊息到其他 thread 的 run loop 上，甚至可以傳到其他 process 的 run loop 上，而 iOS 政策上禁止 process 互相溝通。

在 iOS 7 剛問世的時候，蘋果又完全沒有說清楚這件事，只忙著宣傳 iOS 7 的扁平化新設計。我們為了 `CFMessagePort` 的這項改變，還在 WWDC 2013 會場上跑了兩天的 Lab。

範例程式如下：

```

@interface RecipetUploadOperation : NSOperation
{
    NSPort *port;
    BOOL runloopRunning;
}
@property (nonatomic, strong) UIImage *image;
@property (nonatomic, strong) NSString *JSON;
@end

@implementation RecipetUploadOperation

- (void)main
{
    @autoreleasepool {
        [someAPI uploadImageData:UIImagePNGRepresentation(self.image) callback:^(
            [self quitRunLoop];
        )];
        [self doRunloop];
        if (self.isCancelled) {
            return;
        }
    }
}

```

```

        [someAPI uploadJSON:self.JSON callback:^(

            [self quitRunLoop];
        }];
        [self doRunLoop];
    }
}

- (void)doRunLoop
{
    runloopRunning = YES;
    port = [[NSPort alloc] init];
    [[NSRunLoop currentRunLoop] addPort:port forMode:NSRunLoopCommonModes];
    while (runloopRunning && !self.isCancelled) {
        @autoreleasepool {
            [[NSRunLoop currentRunLoop] runUntilDate:[NSDate dateWithTimeIntervalSinceNow:
0.5]];
        }
    }
    port = nil;
}

- (void)quitRunLoop
{
    [port invalidate];
    runloopRunning = NO;
}

- (void)cancel
{
    [super cancel];
    [self quitRunLoop];
}

@end

```

GCD Semaphores

有了 GCD 之後，很多事情都變得簡單許多。當我們想要在執行到一半的時候暫停下來，現在可以選擇建立 semaphore，接著：

- 只要對 semaphore 呼叫 `dispatch_semaphore_wait`，程式就會在這個地方暫停等候。
- 對已經在等候中的 semaphore，再呼叫 `dispatch_semaphore_signal`，發送 signal，程式就會繼續往下運作。

範例程式如下：

```

#import ;

@interface RecipetUploadOperation : NSOperation
@property (nonatomic, strong) UIImage *image;
@property (nonatomic, strong) NSString *JSON;

```

```
@property (nonatomic, strong) dispatch_semaphore_t semaphore;
@end

@implementation RecipetUploadOperation
- (void)main
{
    @autoreleasepool {
        self.semaphore = dispatch_semaphore_create(0);
        [someAPI uploadImageData:UIImagePNGRepresentation(self.image) callback:^ {
            dispatch_semaphore_signal(self.semaphore);
        }];
        dispatch_semaphore_wait(self.semaphore, DISPATCH_TIME_FOREVER);
        if (self.cancelled) {
            return;
        }
        self.semaphore = dispatch_semaphore_create(0);
        [someAPI uploadJSON:self.JSON callback:^ {
            dispatch_semaphore_signal(self.semaphore);
        }];
        dispatch_semaphore_wait(self.semaphore, DISPATCH_TIME_FOREVER);
    }
}

- (void)cancel
{
    [super cancel];
    dispatch_semaphore_signal(self.semaphore);
}

@end
```

相關閱讀

- [Concurrency Programming Guide](#)
- [Grand Central Dispatch \(GCD\) Reference](#)
- [How To Use NSOperations and NSOperationQueues](#)
- [WWDC 2015 Building Responsive and Efficient Apps with GCD](#)
- [WWDC 2015 Advanced NSOperations](#)
- [WWDC 2013 Asynchronous Design Patterns with Blocks, GCD, and XPC](#)

練習：一個發送多個連線的 Operation

練習範圍

- NSOperation
- GCD

練習目標

我們前面在講 block 的時候，嘗試寫過 httpbin.org 這個服務的 SDK（見 [練習：將 Web Service API 包裝成 SDK](#)）。

在前面的練習中，我們每個 method 都只會發送一個連線，但是在現實的軟體開發過程中，我們往往會想做一些比較複雜的工作，而這樣的工作需要發送好幾個連線，每個連線之間有相依關係，而這樣的工作也可以中途取消，取消的時候，要同時取消所有的連線。

我們現在就要練習寫這樣的程式。

練習內容

1. 先拿出我們在 [練習：將 Web Service API 包裝成 SDK](#) 當中完成的作業。
2. 寫一個叫做 HTTPBinManager 的 singleton 物件。
3. 在這個 HTTPBinManager 中，增加一個 NSOperationQueue 的成員變數
4. 寫一個叫做 HTTPBinManagerOperation 的 NSOperation subclass，HTTPBinManagerOperation 使用 delegate 向外部傳遞自己的狀態。HTTPBinManagerOperation 裡頭的 main method 依序要執行：
 - 對我們之前寫的 SDK 發送 fetchGetResponseWithCallback: 並等候回應。
 - 如果前一步成功，先告訴 delegate 我們的執行進度到了 33%，如果失敗就整個取消作業，並且告訴 delegate 失敗。delegate method 要在 main thread 當中執行。
 - 對我們之前寫的 SDK 發送 postCustomerName:callback: 並等候回應。
 - 如果前一步成功，先告訴 delegate 我們的執行進度到了 66%，如果失敗就整個取消作業，並且告訴 delegate 失敗。delegate method 要在 main thread 當中執行
 - 對我們之前寫的 SDK 發送 fetchImageWithCallback: 並等候回應。
 - 如果前一步成功，先告訴 delegate 我們的執行進度到了 100%，並且告訴 delegate 執行成功，並回傳前面抓取到的兩個 NSDictionary 與一個 UIImage 物件；如果失敗就整個取消作業，並且告訴 delegate 失敗。delegate method 要在 main thread 當中執行。
5. 這個 operation 要實作 cancel，發送 cancel 時，要立刻讓 operation 停止，包括清除所有進行中的連線。
6. HTTPBinManager 要加入一個叫做 executeOperation 的 method，這個 method 首先會清除 operation queue 裡頭所有的 operation，然後加入新的 HTTPBinManagerOperation。
7. HTTPBinManagerOperation 的 delegate 是 HTTPBinManager。HTTPBinManager 也有自己的 delegate，在 HTTPBinManagerOperation 成功抓取資料、發生錯誤的時候，HTTPBinManager 也會將這些事情告訴自己的 delegate。
8. 撰寫單元測試。
9. 寫一個 UI，上面有一個按鈕與進度條，按鈕按下後，就會執行 HTTPBinManager 的 executeOperation，然後進度條會顯示 HTTPBinManagerOperation 的執行進度。

NSCoding

NSCoding 是 Cocoa/Cocoa Touch Framework 中的序列化（Serialization）的實作，所謂的序列化，就是「物件變檔案、檔案變物件」：我們可以將目前程式中正在使用的物件轉換成資料格式，因此可以存成檔案，或是在網路上傳輸、交換，或反之，我們也可以把已經儲存的檔案，再恢復成物件。

對比其他的程式語言，JavaScript 的 serialization 格式就是 JSON，而 PHP 語言裡頭，我們會呼叫 `serialize()` 與 `unserialize()` 轉換 PHP 物件與字串；幾乎重要的程式語言都有自己的 serialization 機制，開發 Mac OS X 與 iOS App 如果不懂 NSCoding，相當於寫 JavaScript 却不懂什麼是 JSON — 不過最近似乎有種 JSON 快要統一天下的趨勢，我們現在大概在所有的語言當中，都可以將 JSON 與各種物件做雙向的轉換。

在 Cocoa/Cocoa Framework 中，我們也可以將資料序列化成 JSON 格式，不過在屬於這個開發 Framework 的傳統中，會更常使用 Plist 格式與 NSCoding。Plist 格式有多種格式，包括文字與 Binary 格式，而 NSCoding 在做的事情則是把各式各樣的不同物件，轉換成 NSData。

實作 NSCoding

NSCoding 是一個 protocol，裡頭只有兩個 method 要實作

```
@protocol NSCoding
- (void)encodeWithCoder:(NSCoder *)aCoder;
- (id)initWithCoder:(NSCoder *)aDecoder;
@end
```

`encodeWithCoder:` 的用途是將我們的物件透過 NSCoder 轉成 NSData，至於 `initWithCoder:` 剛好相反，就是透過 NSCoder，再把 NSData 轉回物件。

假如我們現在有一個叫做 KKSongTrack 的 class，用來代表 KKBOX 裡頭的一首歌，裡頭只有 `songName`、`albumName` 與 `artistName` 三個 property，這三個 property 都是 `NSString`。

```
@interface KKSongTrack : NSObject <NSCoding>
@property (strong, nonatomic) NSString *songName;
@property (strong, nonatomic) NSString *albumName;
@property (strong, nonatomic) NSString *artistName;
@end
```

在實作的時候，我們就可以用 NSCoder 的相關 method 對資料做 encode 與 decode。由於我們的 property 都是 `NSString`，都是 Objective-C 物件，所以我們選擇 `encodeObject:forKey:` 以及 `decodeObjectForKey:`。實作如下：

```
#import "KKSongTrack.h"

static NSString *const kSongNameKey = @"song_name";
static NSString *const kAlbumNameKey = @"album_name";
static NSString *const kArtistNameKey = @"artist_name";

@implementation KKSongTrack

- (NSString *)description
{
    return [NSString stringWithFormat:@"<%@ %p %@ - %@ - %@>",
        NSStringFromClass([self class]), self,
        self.songName, self.albumName, self.artistName];
}

- (instancetype)initWithCoder:(NSCoder *)coder
{
    self = [super init];
    if (self) {
        self.songName = [coder decodeObjectForKey:kSongNameKey];
        self.albumName = [coder decodeObjectForKey:kAlbumNameKey];
        self.artistName = [coder decodeObjectForKey:kArtistNameKey];
    }
    return self;
}
```

```
}

- (void)encodeWithCoder:(NSCoder *)aCoder
{
    [aCoder encodeObject:self.songName forKey:kSongNameKey];
    [aCoder encodeObject:self.albumName forKey:kAlbumNameKey];
    [aCoder encodeObject:self.artistName forKey:kArtistNameKey];
}

@end
```

除了 Objective-C 物件型別外， NSCoder 還有處理 BOOL、整數以及浮點數相關的 method 可以使用。

接下來如果我們想把歌曲轉成 NSData：

```
KKSongTrack *song = [[KKSongTrack alloc] init];
song.songName = @"orz 之歌";
song.albumName = @"orz 專輯";
song.artistName = @"orz";
NSData *data = [NSKeyedArchiver archivedDataWithRootObject:song];
```

把 NSData 再轉回 KKSongTrack：

```
KKSongTrack *decodeSong = [NSKeyedUnarchiver unarchiveObjectWithData:data];
```

NSCoding 的常見用途

我們會在以下這些場合用到 NSCoding：

XIB/Storyboard

我們在實作一個 UIView 的 subclass 的時候，會注意到，如果我們希望在 initialize 這個 view 的時候，就要做一些事情，不但要 override 掉 `initWithFrame:`，也要 override `initWithCoder:`。

如果這個 view 是我們用 code 建立的，那麼就會呼叫到 `initWithFrame:`，但，如果我們是在 Interface Builder 裡頭，用圖形化工具建立了一個 view，那個，當這樣的 view 在執行的時候，則會走進 `initWithCoder:` 這一段的實作。

我們在開始接觸 iOS 開發的時候，大概就會先學習如何使用 Interface Builder 拉出想要的介面，產生出 XIB 或是 Storyboard 檔案。XIB 與 Storyboard 在我們撰寫程式的期間，是 XML 格式的檔案，當我們編譯 App 的時候，Xcode 會將 XIB 與 Storyboard 編譯成 binary 格式的 data，分別是 NIB 與 storyboardc 檔案，而這些 data 其實就是序列化過的 Objective-C view 物件。

在一些其他開發平台上（像是使用 Visual Studio 拉出 Windows Form 應用程式）使用視覺化開發工具的時候，這些工具在做的事情，是把從拉出來的介面產生程式碼；不過在 Xcode 中編輯 XIB 檔案做的事情不一樣，是先產生出序列化後的檔案，然後再執行的時候，讀取這些檔案，將 data 轉成 view 物件。這個流程正是使用 NSCoding protocol，於是從 NIB/storyboardc 讀出我們的 view 的時候，所呼叫的便是 `initWithCoder:` ——我們可以從 UIView 的 interface 中，看到 UIView 實作了 NSCoding protocol。

我們在 Xcode 2 左右的年代（大概是 Mac OS X 10.4 左右）開發 Mac App 時，我們在 Xcode 中其實是直接編輯 NIB 檔案，到了 Xcode 3 與 Mac OS X 10.5 之後才出現使用 XML 格式的 XIB 檔案。這個轉變跟當時 SVN 等版本管理系統的出現有關，在版本管理系統中編輯 binary 格式的檔案，會難以 diff、merge 以及處理版本衝突，所以蘋果便從 binary 格式換成文字格式的檔案。

NSUserDefaults

如果在我們的 App 中，我們想要儲存一些偏好設定，那麼最好用的選擇莫過於 Cocoa/Cocoa Touch Framework 本身就提供的 UserDefaults 物件。操作 UserDefaults 與操作 NSDictionary 差不多，我們只要指定特定的 key，就可以將設定值存入 UserDefaults 中。

NSUserDefaults 支援 NSString、NSArray、NSDictionary、NSData 以及 int、double、float 等型別的資料。但，如果是我們自己定義的 Class，或是許多其他的 Class，會無法存入 UserDefaults 中，我們會需要先透過 NSCoding 轉換成 NSData 後存入，在取出的時候，也要多做一次 unarchive。

比方說，如果我們的 App 的某個地方可以設定顏色，我們想把 UIColor 變成設定值，UIColor 就是一種無法直接存入 UserDefaults 的物件。所以我們想把 UIColor 存入 UserDefaults，就得這麼寫：

```
UIColor *color = [UIColor colorWithHue:1.0 saturation:0.5 brightness:0.5 alpha:1.0];
NSData *data = [NSKeyedArchiver archivedDataWithRootObject:color];
[[NSUserDefaults standardUserDefaults] setObject:data forKey:@"color"];
```

Copy and Paste

我們在行動裝置上面會比較少實作 Copy and Paste 剪貼功能，原因大概是我們比較少在行動裝置上使用與開發比較複雜的編輯工作，而將這些工作留在 desktop 環境。如果我們要開發一套 Mac App，如何實作 Copy and Paste，以及 Drag and Drop，就會是不可不學的知識了。

無論是實作 Copy and Paste 與 Drag and Drop，都是透過 pasteboard 物件，實作 Drag and Drop 其實只是在開始 Drag 的時候，先把想要拖曳的資料先放在另外一個專屬的 pasteboard 中，到了要放開的時候再從 pasteboard 中取出資料。在 Mac 上，pasteboard 物件叫做 NSPasteboard，在 iOS 上叫做 UIPasteboard。

除了像 NSString、NSData 之類的基礎物件之外，許多我們想要可以被複製或拖拉的資料，如果想要存入到 pasteboard 中，還是得先透過 NSCoding 轉換成 NSData 才有辦法。

像我們之前定義了 KKSongTrack 物件，想要寫入剪貼簿，可以這麼做：

```
NSString *const KKBOXSongTrackPasteboardType = @"song_track";

KKSongTrack *song = [[KKSongTrack alloc] init];
song.songName = @"orz 之歌";
song.albumName = @"orz 專輯";
song.artistName = @"orz";
NSData *data = [NSKeyedArchiver archivedDataWithRootObject:song];

[[UIPasteboard generalPasteboard] setData:data
forPasteboardType:KKBOXSongTrackPasteboardType];
```

讀出來：

```
NSData *pasteData = [[UIPasteboard generalPasteboard] dataForPasteboardType:KKBOXSongTrackPasteboardType];
KKSongTrack *pasteSongTrack = [NSKeyedUnarchiver unarchiveObjectWithData:pasteData];
NSLog(@"pasteSongTrack%@", pasteSongTrack);
```

Document-based App

如果我們開發的 App 種類屬於生產力工具，那麼我們很有可能開發的就是一套 Document-based App。

所謂 Document-based App 包括蘋果自己的 iWork 系列，如 Keynote、Numbers、Pages 等等，主要功能就是讓你瀏覽及編輯特定種類的檔案，像 iWork 系列的每一個 App，功能就是編輯特定種類的簡報、試算表與文書檔案。

在 Cocoa/Cocoa Touch Framework 中，便使用 document—Mac 上叫做 NSDocument、iOS 上面叫做 UIDocument，對前面提到的各種不同類型文件做抽象描述，包括負責開啟檔案、儲存檔案、自動存檔以及 iCloud 備份同步等工作，以及描述檔案所在位置與目前狀態等。

在 iOS 上要寫一個 Document-based App，我們會建立一個 UIDocument 的 subclass，而這個 subclass 最重要的就是實作開檔與讀檔兩個 method。比方說，我們建立了一份叫做 KKPlaylist 的 document，裡頭有一個 array，裡頭是我們的 KKSongTrack 物件，這個 document 大概會寫成這樣：

KKPlaylist.h

```
@import UIKit;
@interface KKPlaylist : UIDocument
```

```
@end
```

KKPlaylist.m

```
#import "KKPlaylist.h"

@interface KKPlaylist()
@property (nonatomic, strong) NSMutableArray *songtracks;
@end

@implementation KKPlaylist

- (instancetype)initWithFileURL:(NSURL *)url
{
    self = [super initWithFileURL:url];
    if (self) {
        self.songtracks = [NSMutableArray array];
    }
    return self;
}

- (id)contentsForType:(NSString *)typeName error:(NSError **)outError
{
    NSData *data = [NSKeyedArchiver archivedDataWithRootObject:self.songtracks];
    return data;
}

- (BOOL)loadFromContents:(id)contents ofType:(NSString *)typeName error:(NSError **)outError
{
    NSArray *songtracks = [NSKeyedUnarchiver unarchiveObjectWithData:contents];
    [self.songtracks setArray:songtracks];
    return YES;
}

@end
```

`contentsForType:error:` 與 `loadFromContents:ofType:error:` 裡頭的 `contents` 參數是 `id` 型別，不過其實只接受 `NSFileWrapper` 與 `NSData`，如果在我們的 `document` 中有不少已經實作了 `NSCoding protocol` 的物件，我們就可以輕鬆將物件轉成 `NSData` 之後存檔，或讀取檔案轉回物件。我們通常對 `UIDocument` 做三件事情：

1. 開啟檔案，呼叫 `-openWithCompletionHandler:`
2. 關閉檔案，呼叫 `-closeWithCompletionHandler:`
3. 存檔，呼叫 `-saveToURL:forSaveOperation:completionHandler:`

State Preservation and Restoration

State Preservation and Restoration 是蘋果在 iOS 6 加入的 API，用途是讓 iOS App 可以在開啟的時候，可以立刻回復到上一次關閉 App 時的狀況，方便用戶回復到之前的動作，而不受到因為 App 關閉/開啟而打斷。像 Mail 這個 App，當你在寫一封寫到一半的時候關閉 App，下次打開，就會看到之前寫到一半的那封信，避免用戶找不到上次寫到一半的信在哪裡。

原理是，在應用程式關閉的時候，我們可以先把目前 App 的狀態—像是目前所有的view controller 物件，統統保存起來，下一次應用程式開啟的時候，如果發現存在之前所保存的狀態，就讀取出來，重建上次存起來的 view controller。

要實作 State Preservation and Restoration，首先，要能夠被保存的 view controller，要實作兩個 method：

- - (void)encodeRestorableStateWithCoder:(NSCoder *)coder;
- - (void)decodeRestorableStateWithCoder:(NSCoder *)coder

在 App Delegate 則要實作：

- -application:shouldSaveApplicationState:
- -application:shouldRestoreApplicationState:
- -application:willEncodeRestorableStateWithCoder:
- -application:didDecodeRestorableStateWithCoder:
- -application:willFinishLaunchingWithOptions:

流程是：

一、在 App 關閉的時候，首先系統會透過 -application:shouldSaveApplicationState: 詢問我們是否要保存狀態，如果要的話，我們就回傳 YES。

二、前一步回傳 YES 之後，系統就會透過 -application:willEncodeRestorableStateWithCoder:，提供我們一個 NSCoder，讓我們把必要的狀態透過這個 NSCoder archive 起來。如果我們的 App 裡頭有一個 navigation controller，而我們想把整個 navigation controller 保存起來，可以這麼寫：

```
- (void)application:(UIApplication *)application
willEncodeRestorableStateWithCoder:(NSCoder *)coder
{
    NSMutableArray *viewControllers = [self.navigationController.viewControllers copy];
    NSData *data = [NSKeyedArchiver archivedDataWithRootObject:viewControllers];
    [coder encodeObject:data forKey:@"viewControllers"];
}
```

三、在重新開啟 App 的時候，如果系統發現之前我們已經透過 NSCoder 保存狀態了，那麼，就會向我們透過 -application:shouldRestoreApplicationState:，詢問是否要使用上次的狀態，如果要的話，我們就回傳 YES。

四、接下來 -application:didDecodeRestorableStateWithCoder: 就會被呼叫到，如果我們想還原上次存起來的 navigation controller，可以這麼寫：

```
- (void)application:(UIApplication *)application
didDecodeRestorableStateWithCoder:(NSCoder *)coder
{
    NSData *data = [coder decodeObjectForKey:@"viewControllers"];
    NSArray *viewControllers = [NSKeyedUnarchiver unarchiveObjectWithData:data];
    self.navigationController.viewControllers = viewControllers;
```

}

相關閱讀

- [NSCoding Tutorial for iOS: How To Save Your App Data](#)
- [NSHipster: NSCoder / NSKeyedArchiver](#)
- [Friday Q&A 2013-08-30: Model Serialization With Property Lists](#)
- [About Preferences and Settings](#)
- [Cocoa Application Competencies for iOS: Pasteboard](#)
- [Pasteboard Programming Guide](#)
- [Document-Based App Programming Guide for iOS](#)
- [Preserving and Restoring State](#)
- [Use Your Loaf: State Preservation and Restoration](#)

Crash Reports

身為軟體工程師，在你的工作中（如果不計會議時間的話），其實你只有大約十分之一的時間真的在寫程式，其餘九成的時間，則是會花在測試、Debug 與修正問題上。而發覺已經寫出來的程式裡頭有哪些問題，這件事情，其實可能寫程式本身還要加倍困難。

我們平常要處理的 bug，除了程式最後的邏輯是否符合產品規格之外，更要處理各種稀奇古怪原因而造成的 crash。可以造成 crash 的原因太多了：在 array 或是 dictionary 中插入 nil 會 crash、一邊 enumerate 一個 mutable array 又一邊改動他會造成 crash、你從 server 抓到一份資料以為是字串或數字但 server 偏偏給你 null 會造成 crash，而以 iOS 的設計，執行速度太慢、用了太多的記憶體也都會 crash。

在開發過程中，我們可以有各種工具，像是 debugger 等，可以檢測軟體中的問題，但是當我們把軟體交付給 QA 測試、以及上線之後，軟體已經不在我們的環境，而是在用戶的環境上執行，而用戶在回報問題的時候，一定是不清不楚，我們不能夠期待客服可以幫我們從用戶身上收集問題的重現步驟。在 QA 與用戶的裝置上發生 crash 的時候，最可靠的，就只有當時產生的 crash report。

所以 iOS 工程師要具備蒐集、閱讀 crash report 的能力，因為在絕大多數時候，crash report 是我們理解、修正問題的最重要線索，甚至是唯一的線索。

而其實，我們也不希望經常從用戶的裝置上收集 crash report，因為我們並不希望在用戶的裝置上發生 crash。

我們經常在講什麼要給用戶好的使用體驗，但是在台灣，所謂的使用體驗設計似乎偏往一個奇怪的方向，變成只講視覺上的體驗，就算是一個音樂服務 App，也只關心視覺設計，而用戶聽到怎樣品質的聲音卻完全不管。甚至就連所謂視覺體驗也都只講怎樣做出一個又一個看似美美的畫面，每個畫面之間有什麼關係、流程是否合理也完全不管。可是，在講用戶體驗的時候，我們先來問—什麼是最差的使用體驗？最差的使用體驗就是 crash。

在我們的工作中常常會遇到奇怪的事情：我們拿到一份殘缺不全的 Spec 就得開工，在莫名其妙的期限之前完成，而在要出貨的時候，我們在做的並不是測試與修正，而是為了所謂的使用體驗，所以在畫面上調整一兩個 pixel 的線條位置，而邏輯有問題的程式卻留在上線的產品內。

讓人感嘆的是，這樣的狀況總是在每一輪產品開發流程中都不斷循環。

如何蒐集 Crash Reports

在閱讀 crash report 之前，我們首先要能夠蒐集 crash report。

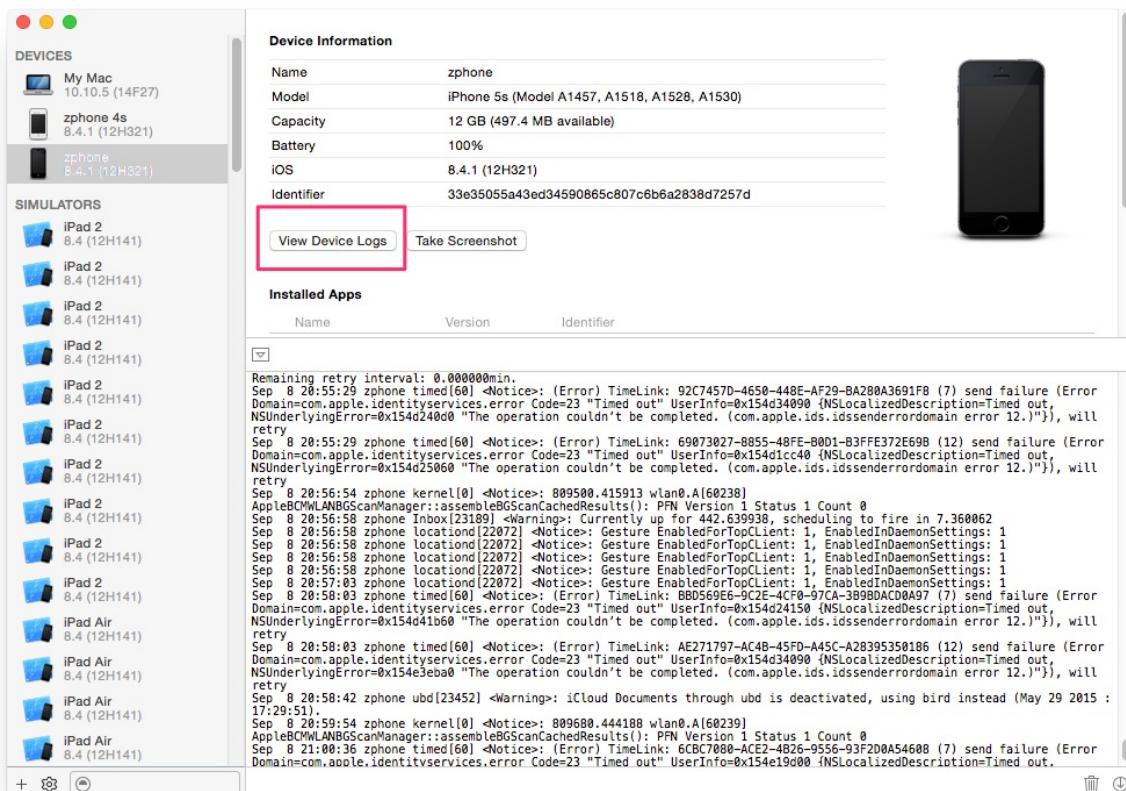
透過 Xcode 收集

如果是在我們自己的裝置上發生了 crash，要找到 crash report 最簡單的方法，就是透過 Xcode。我們先將裝置連接到 Mac 上，從 Xcode 的選單中，選擇 Window -> Devices，就會出現一個 window。

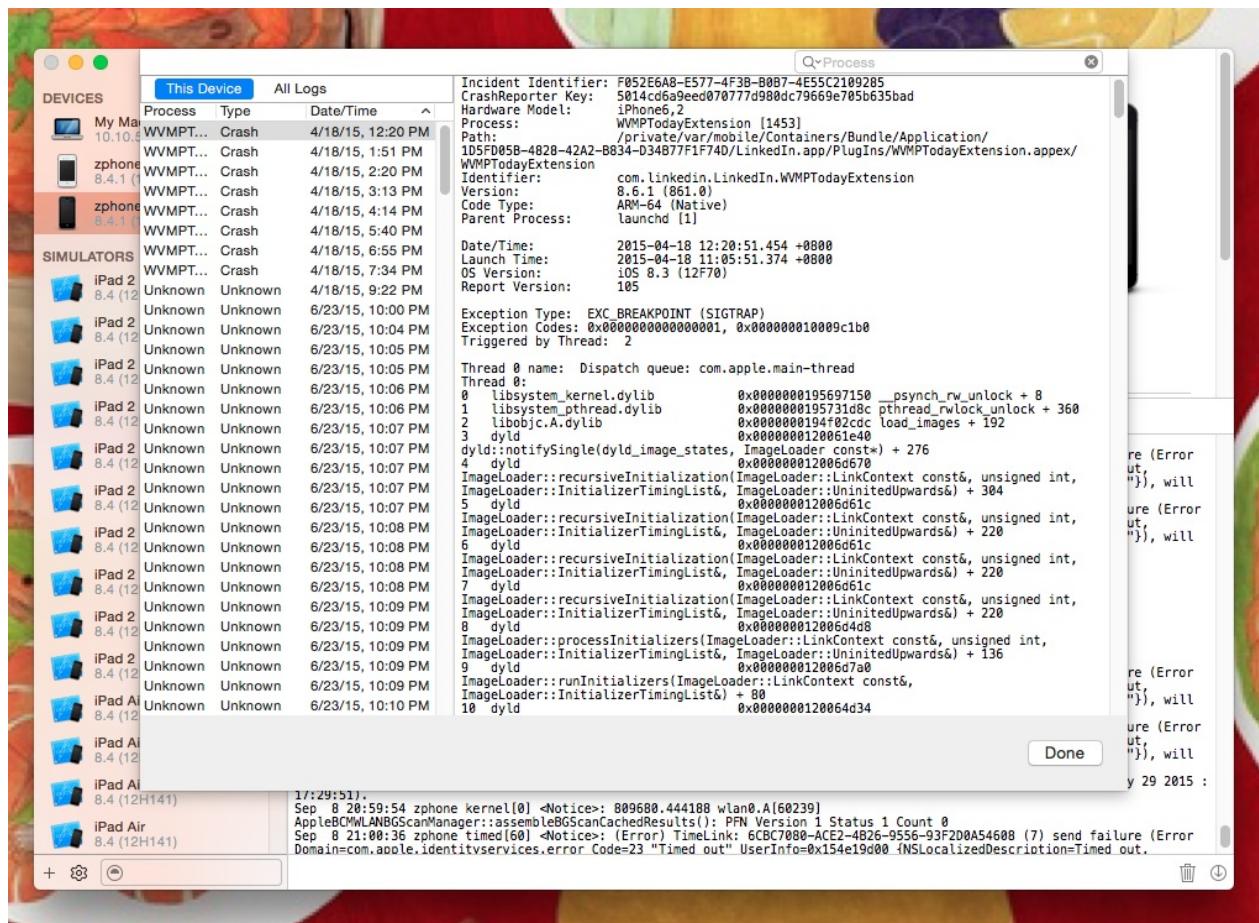
在這個 window 的左方會列出所有目前可用的裝置，如果你是在開發 Apple Watch App，Apple Watch 不會獨立顯示，而是會依附在與 Apple Watch 配對的 iPhone 上。除了連接的 iOS 設備外，由於 Xcode 也可以用來開發 Mac App，所以你自己正在使用的這台 Mac 也會被當做是一台專屬裝置。

在左下方會列出目前所有 Xcode 可以使用的 iOS 模擬器，如果你覺得平常在 Xcode 選單中的 iOS 模擬器太多—iPhone 4、5、6 都列出來一實在顯得雜亂，或有個需要用到的模擬器之前被關閉了，你也可以在這邊管理。

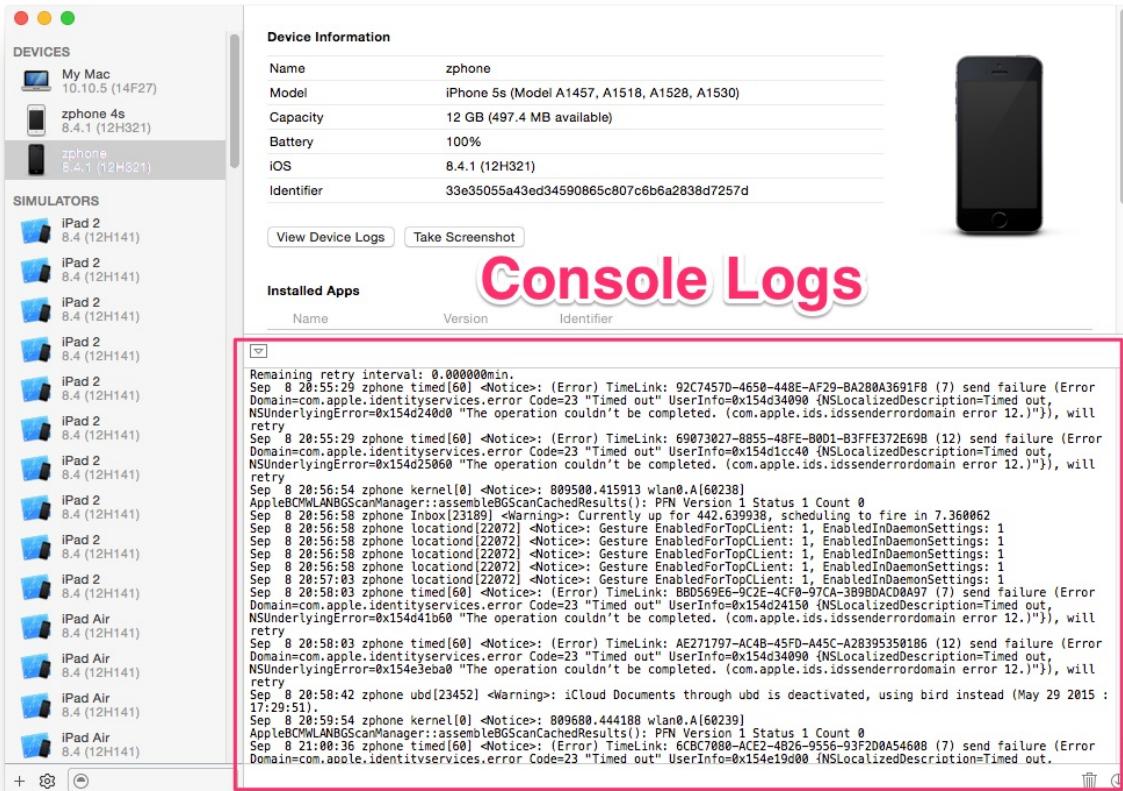
我們選擇了指定的 iOS 裝置後，從畫面右方，點選「View Device Logs」按鈕。



接下來會跳出另外一個 window。這個 window 的左方會列出目前所存放的 crash report，按照 App 名稱與時間排列。我們可以就我們所知道的 crash 發生時間，找到我們的 App 的 crash report。



如果在你的團隊中有 QA 人員，我們建議在 QA 人員也使用 Mac 電腦，同時安裝 Xcode，並且在 crash 發生的同時，除了 crash report 之外，也把 console logs 目前所有的內容也剪下來一份。因為在發生 crash 的時候，在 console 上面同時也會 print 出來一些重要資訊。



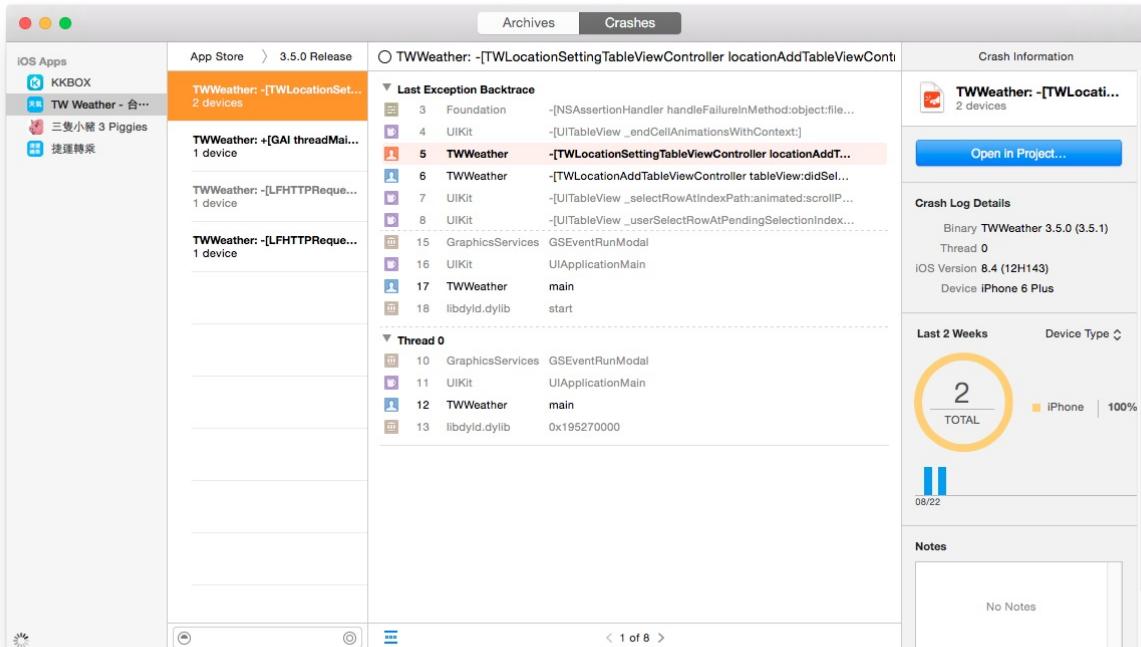
透過 iTunes Connect 收集

產品上線之後，蘋果其實也會幫我們收集 crash report。從 iTunes Connect 的 App 資訊頁面中，在最下方可以看到一個叫做「Crash Reports」的連結，接下來就會導引到查看 crash reports 的頁面。

在 iTunes Connect 上，蘋果會幫我們列出每個主要作業系統版本上的十大 crash 原因，如果我們想在推出下一個版本之前，把前一個版本的所有 crash 都修一輪，這個功能還算有用。

不過，我們更常遇到的狀況是，用戶遇到 crash 之後，直接透過電話等方式向我們的客服反應，我們需要解決的往往不只是某個版本有哪些問題，更需要的是可以在最短的時間內，解決單一用戶的問題。

蘋果在後來幾個版本的 Xcode 中，也增加了瀏覽線上 crash 的功能（可以用 Window -> Organizer 功能叫出來），讓瀏覽 crash 變得方便一些，但一樣沒有找到特定用戶 crash 的辦法。



直接從 iOS 裝置上瀏覽 crash log

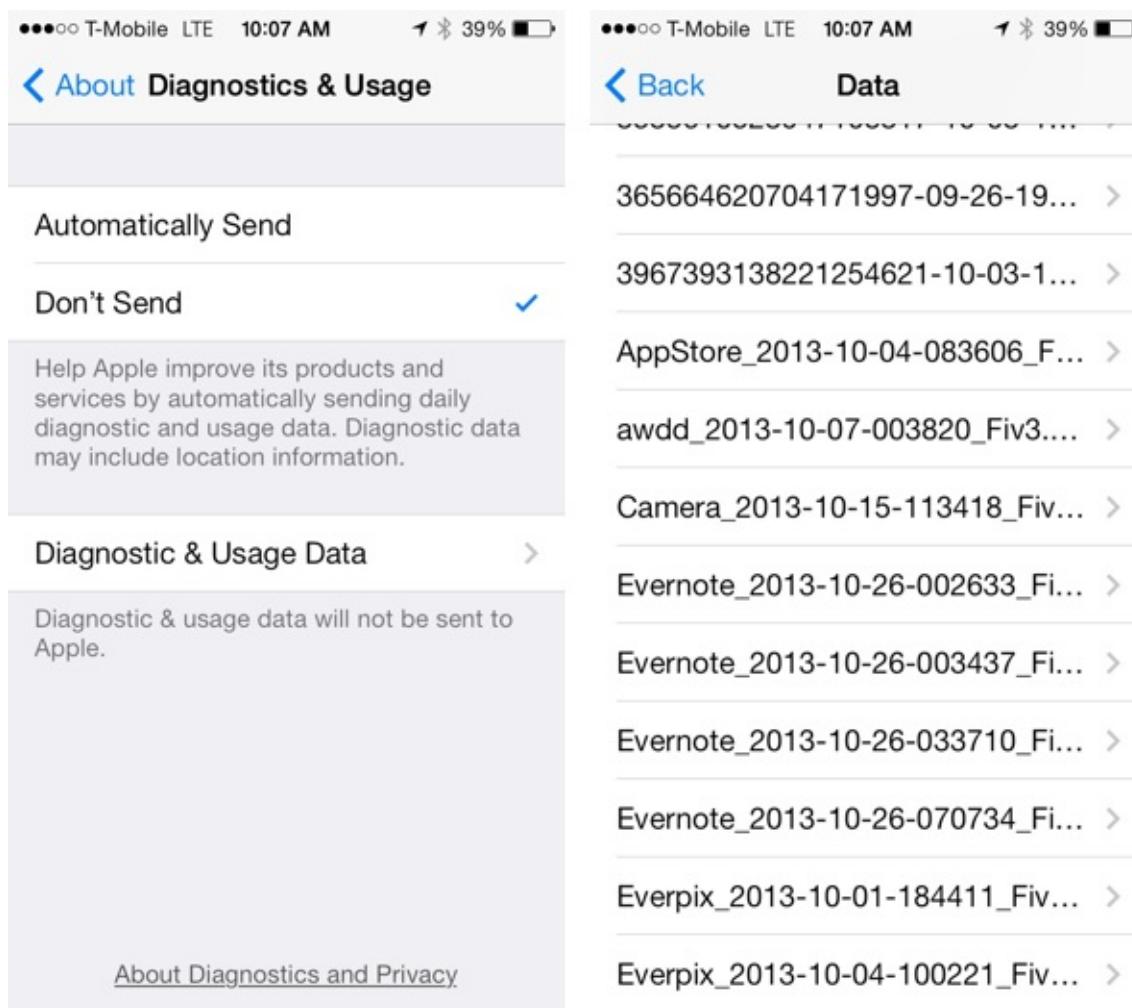
如果 crash 是發生在用戶的裝置上，我們不太有機會可以叫用戶安裝 Xcode—如果不是因為工作的需求，誰會安裝這麼肥大的軟體呢？我們也不太可能直接拿到用戶的裝置蒐集 log，畢竟我們在台灣，用戶可能會在世界的任何一個角落。很多時候，我們會想辦法請用戶直接從裝置上協助我們取得 crash report。

我們可以從系統設定 App 找到 crash report，不過，不同版本的 iOS 中，crash report 存放的位置不太一樣。

- iOS 8 之前，放在 Settings->General->About->Diagnostics & Usage
- iOS 8 開始，放在 Settings->Privacy->Diagnostics & Usage

在這個畫面中，點選了 Diagnostics & Usage Data 選項，就可以看到這台裝置上的 crash report 列表。點下去之後，就可以看到 crash report 的內容。

在這個畫面中，用戶其實很難把 crash report 拿出來。因為這個畫面就只有一個簡單的 text view 顯示 crash report，沒有任何方便匯出的功能，要不就是請用戶想辦法全選後另外找一個文字編輯 App 貼上，要不就是要請用戶拍攝螢幕截圖，而當我們收到螢幕截圖之後也很麻煩，因為接下來我們要解開 crash 的 call stack 中的記憶體位置，只有截圖，就只能對圖片中的文字做肉眼 OCR。



透過第三方服務收集

從 iTunes Connect 上很難掌握單一用戶發生的 crash，從用戶裝置上抓 log 也很麻煩，因此像 TestFlight¹、Crashlytics 或 HockeyApp 等廠商就看準了這種需求推出服務，安裝這些服務的 SDK 後，這些服務會幫我們盡可能的收集 crash report，在 SDK 中設定必要的資訊後，我們便可以透過特定的 user ID，找到特定用戶發生的 crash。

像 Crashlytics SDK，就可以呼叫 `+setUserIdentifier:`、`+setUserName:` 與 `setUserEmail`²，在 HockeyApp SDK 中，則是要在 App 中實作 `BITHockeyManagerDelegate` protocol，告訴 HockeyApp 用戶 ID。

在使用這些服務的時候，他們會告訴我們要上傳每個版本的 debug symbol，原因是，當我們發行 release build 的時候，compiler 會把程式中的 Debug 資訊抽掉—不然所有人只要一拿到 crash report，就可以輕鬆知道我們 App 是怎麼寫的，而形成安全性的風險。所以在發生 crash 時，crash log 中其實只有發生錯誤的 function/method 的記憶體位置，必須要有 debug symbol 檔案才能還原。而 Crashlytics 或 HockeyApp，可以幫我們在 server 上就還原記憶體位置，讓我們不必手動做這件事。

我們在這邊不特別推薦使用哪一家的服務，建議你各自試試看，然後就功能與價格自行比較。

如果我們在 App 中用了 Google 的 Google Analytics 服務，統計 App 中每個功能的用量，可以注意到，其實 Google 也會幫我們蒐集 crash report，但由於 Google Analytics 並不會還原記憶體，所以 Google Analytics 蒐集的資料，對我們解決問題沒有太多幫助，但 Google 的報表拿來看長時間 crash 的趨勢變化倒還頂不錯。

這些服務攔截 crash report 的原理是，當 exception 發生的時候，其實 App 會對自己發送一個 UNIX signal，原始的 signal handler 做的事情就是在 console 上 print 訊息、產生 crash report 並且停止應用程式。其實 Objective-C 裡頭的 try...catch 也是透過 UNIX signal 實作的，我們不妨想像整個 App 其實都做了一個很大的 try...catch，只是這個 catch 做的事情就只有讓 App crash。

這些服務要求你在 App 啟動的時候，也啟動他們的服務，目的就在於改變 signal handler，讓他們可以將 crash report 攔截下來，在下一次啟動的時候，再找個恰當的時間回傳。

如此一來我們可以知道：如果我們的 crash 發生在這些服務啟動之前，那麼這些服務也攔截不到 crash report，所以啟動這些服務的時機應該要盡可能早。另外，由於這些 SDK 也改變了 signal handler 的行為，原本一些狀況下 App 應該要發生 crash，當我們加了某些服務的 SDK 之後，他們的 signal handler 的實作反而會是只收集 crash report，但是當 App 繼續執行；所以，當用戶回報某個功能無法使用，其實很有可能是已經發生了 exception。

此外，如果是我們的 App 記憶體用量使用太多，導致被系統中斷，這種 crash report 也沒辦法蒐集到，因為這種 crash 是由外部的 watch dog 造成的，而不是內部的 UNIX signal 驅動的。

¹. 後來被蘋果併購 ↪

². 參考 <http://support.crashlytics.com/knowledgebase/articles/92521-how-do-i-set-user-information-> ↪

記憶體不足時產生的 Crash Report

因為系統整體記憶體不足所產生的 crash report，與其他的 crash report 長得不太一樣—這種 crash report 中只會顯示當時每個 process 使用了多少系統資源。此外，就像前一章講到的，你可能從 Xcode 或 iTunes Connect 上蒐集到這種 crash report，在 Crashlytics 或 HockeyApp 上找不到。

當我們在閱讀這類型 crash report 的時候，需要知道 rpage 的單位，一個 rpage 相當於 4k 的記憶體。所以我們可以看到，在這台 iPad Mini 上，KKBOX 用了22810 個 rpage，所以 KKBOX 大概使用了 87 MB 的記憶體 ($22810 * 4 / 1024$) 。

要怎樣修正這種問題呢？首先，87 MB 在 iPad 上，其實不算太多，由於記憶體不足不見得是單一 App 的問題，而是所有的 App 共同把記憶體用完了，我們看到幾個系統服務也把記憶體用得很兇，像 mediaserverd（系統底層一個用來播放影音的 service）用了 100 MB、SpringBoard（就是 iOS 裝置按下 Home 按鈕可以看到的 App 列表畫面）也用了將近 40 MB。所以，我們可以先試試看重新開機，把系統服務佔用的資源是放掉，再來看看執行 KKBOX 會不會出問題。

一個 App 到底可以用多少記憶體？蘋果一直沒有說得很清楚，我們最早在 iPhone 3G 上開發 KKBOX 時，KKBOX 可以使用的記憶體極限大概在 40 MB 左右，由於我們還要把一定比例的歌曲資料載入到記憶體中，所以在畫面上能使用的記憶體就必須盡量節省。在比較後來的機種上，像 iPhone 5S 或 iPad 3 之後，大概用到 200MB 左右都沒問題，由於每個機種能用的資源都不太一樣，需要的時候，我們可以寫個一直 alloc 記憶體的迴圈做實驗，不過在 KKBOX 這邊，我們手上的裝置也沒有很齊全。

如果我們真的發現 App 中記憶體用太多，就只能夠想辦法節省記憶體。想要節省記憶體用量，最重要的原則就是—哪邊用了比較多的記憶體，不要透過閱讀程式碼猜測，而是直接用 Instrument 跑 profiling 觀察，確實找到記憶體的瓶頸。

至於要節省記憶體用量的作法，大概就是，如果一份資料在當下沒有必要放在記憶體中，可以考慮先放入檔案中，至於要避免產生太多 view 佔用記憶體，可以參考 [記憶體管理 Part 3 - Memory Warnings](#) 這一章。

範例 crash report 如下：

```

Incident Identifier: 7BC7B59F-A13D-44C1-8D57-62508830C0C2
CrashReporter Key: fe242c8a27a16318ac7bbe16c6fbaaff6ce8a3a0
Hardware Model: iPad2,5
OS Version: iPhone OS 7.0.4 (11B554a)
Kernel Version: Darwin Kernel Version 14.0.0: Fri Sep 27 23:00:49 PDT 2013; root:xnu-2423.3.12~1/RELEASE_ARM_S5L8942X
Date: 2015-08-07 11:20:54 +0800
Time since snapshot: 56 ms

Free pages: 922
Active pages: 3697
Inactive pages: 2197
Speculative pages: 7
Throttled pages: 74156
Purgeable pages: 0
Wired pages: 47617
File-backed pages: 5424
Anonymous pages: 477
Compressions: 0

```

Decompressions: 0
 Compressor Size: 0
 Uncompressed Pages in Compressor: 0
 Largest process: mediaserverd

Processes

Name [reason]	<UUID> (state)	rpages	recent_max	fds
MobileMail <b3574f4bded1315cb2e50e5de205be48> [vm-pageshortage] (resume) (continuous)		1113	1113	200
tccd <1fea8c5a71943151b5cd304c7eb0fd8c> [vm-pageshortage] (daemon)		170	170	200
kbd <be2d64e41bf43e48a09a23fb129eb0b4> [vm-pageshortage] (daemon)		381	381	200
KKBOX <8367b00614e735f5861044b98c96cd1c> [vm-pageshortage] (frontmost) (resume)		22810	22810	200
ptpd <db9048c36f6c3c18a7330fc96d93a0cf> (daemon)		657	657	200
identityservices <18cc20db2e4739a782cc8e38e03eff52> (daemon)		349	349	200
syslogd <6539f4cf4dcf34daadf1d99991926680> (daemon)		157	157	50
powerd <0a253ac2a99236809422214be1700bc0> (daemon)		121	121	100
imagent <bef102e1faef39209926fb25f428a71e> (daemon)		298	298	100
iaptransportd <42faa147f61a314bb735e239f445efaf> (daemon)		219	219	50
mDNSResponder <8922e9954d893eb9a1ab27ca4723bbab> (daemon)		228	228	100
apsd <0dd1fd7c2edc3cf9899a4830541c1bac> (daemon)		468	468	100
dataaccesssd <5da732b6ce6935f2928461a022101aba> (daemon)		1228	1228	200
mediaremoted <476eb521b8423428b4e6df20d3fe4091> (daemon)		327	327	50
wifid <a5cf99e5a0f032a69bc2f65050b44291> (daemon)		1859	1859	200
mediaserverd <71101024312538ccae5799b88681e38d> (daemon)		25807	25807	200
sharingd <a95c2cea41b43cc69b0bbe9a03730d45> (daemon)		460	460	200
calaccesssd <77a5672f2f653f64acf7367126a8d173> (daemon)		420	420	200
itunesstored <c52ea3e4aceb398e9a98ff595c17669f> (daemon)		1576	1576	200
locationd <c31643022d833911b8b7461fd3964bd5> (daemon)		2497	2497	200
syslog_relay <c4c1e88d92a537888b15d6118c5fa1d1> (daemon)		107	107	200
SpringBoard <3c0e305139b331c6b37d2e9516f5804f>		10207	10207	200
backboardd <d61df126c4673b25bbfe5d9024be1d48>		13495	13495	50

	(daemon)				
aggregated	<a5dda46586ba3a3ccb298bd8aa545e50>	666	666	50	
	(daemon)				
lockdownd	<6f28a28a0025348aa5361078e41914e3>	340	340	100	
	(daemon)				
fairplayd.A2	<6cae0c124e1830598a43f6b4d790917f>	141	141	100	
	(daemon)				
configd	<c57db43e53a73f8a9360f4d0d9001704>	686	686	100	
	(daemon)				
fseventsd	<5c909a70b62f33c8856e3158834ba071>	336	336	100	
	(daemon)				
BTServer	<3933a8148924316b9f19dd3d10a23f00>	3891	3891	100	
	(daemon)				
distnoted	<38616bd8864034e7bc741f8bd7313349>	1187	1187	100	
	(daemon)				
UserEventAgent	<a3c7e56924ec3690a994a75a0ea79ee8>	850	850	100	
	(daemon)				
networkd	<84dfdb49c24132fa8dd10520deb16645>	523	523	100	
	(daemon)				
filecoordination	<4fa03f2b93363668a1159715de4b0270>	186	186	200	
	(daemon)				
EscrowSecurityAl	<65547599d6d331f2aec702509ccb1079>	175	175	200	
	(daemon)				
itunescloudd	<9a38b56ee4fd3c308766da99af8eeaed>	932	932	200	
	(daemon)				
touchsetupd	<02780826b4263a7498bda167721b5f8c>	163	163	200	
	(daemon)				
afcd	<5c18557c26f73b88a54ad94f3cd06d0c>	113	113	200	
	(daemon)				
notification_pro	<852af3fe832e3cc3a3f31d05511a5482>	124	124	200	
	(daemon)				
cplogd	<148e9e2ff86130ecb63423c183f68da5>	137	137	200	
	(daemon)				
pasteboardd	<6bb2d8a2beb530b095c82dc2c1cda0f7>	119	119	200	
	(daemon)				
wirelessproxd	<46066fc432663d45ab7e055082fe0bd6>	11	11	200	
	(daemon)				
CommCenterClassi	<b836b786e0cb3785a18a94e9b13c9991>	363	363	50	
	(daemon)				
notifyd	<35afacabfed73771889e72a017479709>	251	251	100	
	(daemon)				

Crash Report 的三部分

一份 crash report 大概分成三個部分。我們會用一份 crash report 說明，這份 crash report 來自於 KKBOX 的 QA，而且裡頭還沒有解開（symbolication）—也就是說，從 crash report 的 call stack 這一段，我們只能夠看到發生問題的 function/method 的記憶體位置，而看不到錯誤發生在程式的哪一行。

環境摘要與錯誤主要原因

在 crash report 的最上方可以看到一些基本資訊，其中比較重要的是

- 機種：在下面這份 crash report 中，可以看到用戶用的是 iPad，同於 KKBOX 的 iOS 版本同時支援 iPhone 與 iPad，而在 iPhone 與 iPad 上根本有兩套不一樣的 UI code，所以我們知道，現在應該要去看跟 iPad 有關的問題。
- 版本號碼：同一個 App 的不同 build，都會產生不一樣的 debug symbol，所以如果要找到正確的 debug symbol，就一定要確認版號。在 KKBOX 的開發版本中，我們會對 Info.plist 做一些 pre-process，把 Jenkins build 號碼（是的，KKBOX 使用 Jenkins 做持續整合，QA 都是直接從 Jenkins 上抓取需要測試的 build）與這一版程式的 git revision hash 也加到版號中，所以，我們可以知道，應該要去 Jenkins build 5145 裡頭去找 debug symbol。
- 錯誤代碼：我們在這邊可以知道 Exception Type 是 EXC_CRASH，並且在 main thread (thread 0) 發生 crash。錯誤代碼對我們理解 crash 非常重要，我們會稍後說明。

```

Incident Identifier: CC5B3783-804F-49A7-AF6B-7C7B382CAE76
CrashReporter Key: 73124d372075eabb72b5625621a4396ffe893a49
Hardware Model: iPad2,5
Process: KKBOX [155]
Path: /private/var/mobile/Containers/Bundle/Application/63493C9C-7C1B-47CA
-83D3-8CC068537B89/KKBOX.app/KKBOX
Identifier: tw.com.skysoft.iPhone
Version: 6.2.66.5145.073c882 (6.2.66)
Code Type: ARM (Native)
Parent Process: launchd [1]

Date/Time: 2015-08-06 10:41:52.859 +0800
Launch Time: 2015-08-06 10:39:28.377 +0800
OS Version: iOS 8.3 (12F69)
Report Version: 104

Exception Type: EXC_CRASH (SIGABRT)
Exception Codes: 0x0000000000000000, 0x0000000000000000
Triggered by Thread: 0

```

Call Stack

第二部分是每個 thread 當時的 call stack，讓我們可以看到每個 thread 當時在做些什麼事情。這邊會是了解問題的關鍵，因為我們要從這段資訊裡頭確實知道 crash 發生在程式的哪個地方。

根據取得 crash report 的方式不同，你可能會看到不同格式的資料。比方說，當你直接從裝置上抓下來，你會看到完全沒有解開的 call stack，像這樣：

```

Last Exception Backtrace:
(0x23e61132 0x321dec72 0x23e665f8 0x23e644d4 0x23d939d4 0x23e40272 0x24aef96 0x24aefc46
0x755ba6 0x2776be1c 0x2776bede 0x27760a60 0x2756f2b6 0x27498c16 0x26eb7440 0x26eb2c90 0x2
6eb2b18 0x26eb24ba 0x26eb22aa 0x26f05ab8 0x2bad5bfe 0x24dd9d08 0x23e16550 0x23e26a46 0x23
e269e2 0x23e25004 0x23d7099c 0x23d707ae 0x2b5201a4 0x274fb690 0xee070 0x32786aaa)

Thread 0 name: Dispatch queue: com.apple.main-thread
Thread 0 Crashed:
0  libsystem_kernel.dylib          0x3284cdf0 0x32838000 + 85488
1  libsystem_pthread.dylib         0x328cdc92 0x328ca000 + 15506
2  libsystem_c.dylib               0x327eb934 0x327a2000 + 301364
3  KKBOX                           0x00a12ee2 0x10000 + 10497762
4  KKBOX                           0x00a6e620 0x10000 + 10872352
5  CoreFoundation                 0x23e61466 0x23d57000 + 1090662
6  libobjc.A.dylib                0x321deefc 0x321d8000 + 28412
7  libc++abi.dylib                0x31a02dec 0x319eb000 + 97772
8  libc++abi.dylib                0x31a028b4 0x319eb000 + 96436
9  libobjc.A.dylib                0x321dedba 0x321d8000 + 28090
10 CoreFoundation                 0x23d70a38 0x23d57000 + 105016
11 CoreFoundation                 0x23d707ae 0x23d57000 + 104366
12 GraphicsServices               0x2b5201a4 0xb517000 + 37284
13 UIKit                           0x274fb690 0x2748c000 + 456336
14 KKBOX                           0x000ee070 0x10000 + 909424
15 libdyld.dylib                  0x32786aac 0x32785000 + 6828

```

如果你是在你自己電腦上，用 Xcode 按下 Run，執行你的 App，因為 Xcode 幫你保留了 App 的 debug symbol，所以在裝置上發生 crash 的時候，你用 Xcode 的 Devices 選單把 crash report 抓進來的時候，Xcode 會幫你用 debug symbol，完整解開整份 crash report。

但如果一份 crash report 是來自 QA，在 QA 的電腦上有 Xcode，但是並沒有對應的 debug symbol 時，Xcode 也會嘗試做 symbolicate，但只能夠解開像是 UIKit、CoreFoundation 之類的系統 library，但是看不到屬於你的 App 的哪部分。而 Xcode 是透過 spotlight 的索引找到對應的 debug symbol，就算把 debug symbol 抓下來，但 spotlight 不見得會立刻做索引，所以你可能還是看到沒有解開的 crash report。

解開系統 library 的部份往往沒什麼用，以上面那個 log 來說，如果你熟悉 iOS 的運作，就算不解開記憶體位置，也可以看出 thread 0 的 call stack 中：

- 15 是 start
- 14 是 KKBOX 的 main.m 裡頭的 main
- 13 是 UIApplicationMain
- 12-5 是在跑 run loop，而由於這是一個 exception，所以在 thread 0 中跟 KKBOX 有關的部份，大概是 Google Analytics 或 Hockey App 的 exception handler

而我們在前面拉拉雜雜講了許多基本觀念，都是為了這一章做準備：我們要了解什麼是 run loop，才有辦法了解這段 call stack 的 backtrace，對吧？

在這份 crash report 中，其實我們更有興趣的是 Last Exception Backtrace 這一段。這段資料代表的是 exception 發生當時到底發生了什麼事，也就是 exception 被 throw 出來的階段，而雖然 crash report 告訴我們發生 crash 的 thread 0，但這時候 thread 0 只是 catch exception 的地方而已。

Libraries

第三部分是我們的 App 載入了哪些 library，因為很長，所以這邊只放了部分。在這份 log 中，最重要的資訊是 KKBOX 被載入到 0x10000 - 0xc1bfff 這一段記憶體的區間中，這個資訊對我們接下來怎麼解開記憶體位置非常重要。

從一個 App 載入了哪些 library，也可以看出一些特色。比方說，KKBOX 除了載入自己的主要 App 之外，也載入了在自己 bundle 下的一份 Swift runtime，像是 libswiftCore.dylib、libswiftCoreAudio.dylib 等等，代表 KKBOX 裡頭使用了 Swift 語言。

我們在第一章就提到，Swift 與 Objective-C 的 runtime 實作不同，所以 Swift 程式必須要連結到特定版本的 Swift runtime，而用 Xcode 開發出來的 Swift App，會在每個 App bundle 裡頭都放一份 Swift runtime—那不就是裝置裡頭有多少用 Swift 寫的 App，裝置上就放了多少份 Swift runtime？的確如此，所以還好 Swift 核心 runtime 的尺寸不大，但我們也很擔心以後 Swift 會變得多大就是了。

```
Binary Images:
0x10000 - 0xc1bfff KKBOX armv7 <d91b937e9d073c21a79f5bbbe1cc4dbd> /var/mobile/Containers/Bundle/Application/63493C9C-7C1B-47CA-83D3-8CC068537B89/KKBOX.app/KKBOX
0x14e4000 - 0x164bfff libswiftCore.dylib armv7 <fa5b9494d6403f13ae80664a88301250> /var/mobile/Containers/Bundle/Application/63493C9C-7C1B-47CA-83D3-8CC068537B89/KKBOX.app/Frameworks/libswiftCore.dylib
0x17c0000 - 0x17c7fff libswiftCoreAudio.dylib armv7 <bd2181365f933ed3b330b0517f75fde0> /var/mobile/Containers/Bundle/Application/63493C9C-7C1B-47CA-83D3-8CC068537B89/KKBOX.app/Frameworks/libswiftCoreAudio.dylib
0x17d4000 - 0x17dffff libswiftCoreGraphics.dylib armv7 <95229d09c03d3eba9fbb038741503af3> /var/mobile/Containers/Bundle/Application/63493C9C-7C1B-47CA-83D3-8CC068537B89/KKBOX.app/Frameworks/libswiftCoreGraphics.dylib
0x17f4000 - 0x17fbfff libswiftCoreImage.dylib armv7 <59ca6e9173993aa39882799efdafd355> /var/mobile/Containers/Bundle/Application/63493C9C-7C1B-47CA-83D3-8CC068537B89/KKBOX.app/Frameworks/libswiftCoreImage.dylib
0x1808000 - 0x180ffff libswiftDarwin.dylib armv7 <764c0e157b49314088a4c9f8e1390a1a> /var/mobile/Containers/Bundle/Application/63493C9C-7C1B-47CA-83D3-8CC068537B89/KKBOX.app/Frameworks/libswiftDarwin.dylib
0x1820000 - 0x1823fff libswiftDispatch.dylib armv7 <e167a8d29df3694b0db2d3fc60adbb0> /var/mobile/Containers/Bundle/Application/63493C9C-7C1B-47CA-83D3-8CC068537B89/KKBOX.app/Frameworks/libswiftDispatch.dylib
0x1830000 - 0x185ffff libswiftFoundation.dylib armv7 <dde3f834069c3e6a9421572c0d01bbe3> /var/mobile/Containers/Bundle/Application/63493C9C-7C1B-47CA-83D3-8CC068537B89/KKBOX.app/Frameworks/libswiftFoundation.dylib
0x1898000 - 0x189ffff libswiftObjectiveC.dylib armv7 <786938b80ba63395aa5da6935df0c02e> /var/mobile/Containers/Bundle/Application/63493C9C-7C1B-47CA-83D3-8CC068537B89/KKBOX.app/Frameworks/libswiftObjectiveC.dylib
0x18ac000 - 0x18affff libswiftSecurity.dylib armv7 <12a8743e1ad636ebaaac523d1d709341> /var/mobile/Containers/Bundle/Application/63493C9C-7C1B-47CA-83D3-8CC068537B89/KKBOX.app/Frameworks/libswiftSecurity.dylib
0x18b8000 - 0x18c3fff libswiftUIKit.dylib armv7 <a04c41396cb939e6b3368360bfaffe10> /var/mobile/Containers/Bundle/Application/63493C9C-7C1B-47CA-83D3-8CC068537B89/KKBOX.app/Frameworks/libswiftUIKit.dylib
0x1fe5c000 - 0x1fe7fff dyld armv7 <35ecdca1a767375e95ffa0f2a78d76d0> /usr/lib/dyld
0x22663000 - 0x2267efff libJapaneseConverter.dylib armv7 <1531c07cd9613bba8fc6fe9217f33612> /System/Library/CoreServices/Encodings/libJapaneseConverter.dylib
0x228f0000 - 0x22a5dfff AVFoundation armv7 <d269609e868231debb09b8b9b65a7367> /System/Li
```

```
brary/Frameworks/AVFoundation.framework/AVFoundation
```

```
...
```

這一段資料還可以提供我們另外一種資訊。來看看這個 crash report，KKBOX 直接放在 /Users/USER 目錄下，並沒有放在 sandbox 環境中，而明明就該載入 armv7 的 library，卻載入了一堆 armv6 的 library，再仔細看看，出現了 CydiaSubstrate 這些東西。顯然用戶做了 JB。

```
0x7d000 - 0x71cff +KKBOX armv7 <f8a0ae9e6da833e0a9bd332e3d4b8a4c> /Users/USER/KKBOX.app/KKBOX
0x8b1000 - 0x8b1fff MobileSubstrate.dylib armv6 <ad3e6cb9e915360ebc71ccbf27bc4ea7>/Library/MobileSubstrate/MobileSubstrate.dylib
0x903000 - 0x905fff SubstrateLoader.dylib armv6 <974e4b1ab6e6397db859d79f37b7ab37>/Library/Frameworks/CydiaSubstrate.framework/Libraries/SubstrateLoader.dylib
0x929000 - 0x935fff Activator.dylib armv7s <24b72822b56d30f4b88c83448f3db656>/Library/MobileSubstrate/DynamicLibraries/Activator.dylib
0x972000 - 0x974fff Emphasize.dylib armv7 <4a31e195474039db8a69730adc9b7642>/Library/MobileSubstrate/DynamicLibraries/Emphasize.dylib
0x977000 - 0x97cff CydiaSubstrate armv6 <abeb3e46b03b3abeb9d3feba7fefef2fb>/Library/Frameworks/CydiaSubstrate.framework/CydiaSubstrate
0xa82000 - 0xa89fff libapplist.dylib armv7s <3ef6ef0d6b7d350982114b6b4221ef01>/usr/lib/libapplist.dylib
0xa90000 - 0xa95fff LocalIAPStore.dylib armv7s <f3341a7878d9319cb0bef7c2d2cbd896>/Library/MobileSubstrate/DynamicLibraries/LocalIAPStore.dylib
0xa9f000 - 0xaa5fff RePower.dylib armv7s <dce8d29ea843350baad4f700b0443795>/Library/MobileSubstrate/DynamicLibraries/RePower.dylib
0xaaa000 - 0xaaaffff ToneEnabler.dylib armv7s <21fc136886163804b54e0524f7d1ca25>/Library/MobileSubstrate/DynamicLibraries/ToneEnabler.dylib
0xaad000 - 0xaeaffff WeeLoader.dylib armv7 <7ff8f9166f93382eb27aba388d599a0a>/Library/MobileSubstrate/DynamicLibraries/WeeLoader.dylib
0xab1000 - 0xac2fff WinterBoard.dylib armv6 <2f7ede1815c93754b5c91a06195b485a>/Library/MobileSubstrate/DynamicLibraries/WinterBoard.dylib
0xad6000 - 0xadaffff colorYourBoardFree.dylib armv7s <2e3552b8fe5c3bb8967e25d652d5a33f>/Library/MobileSubstrate/DynamicLibraries/colorYourBoardFree.dylib
0xadff000 - 0xae0fff zeppelin_uikit.dylib armv7 <1401eeb739bb3a5d88a45926eac68b46>/Library/MobileSubstrate/DynamicLibraries/zeppelin_uikit.dylib
0x77a5000 - 0x77b0fff QuickSpeak armv7s <9932eaf14fb23095b04ab610b05ba02e>/System/Library/AccessibilityBundles/QuickSpeak.bundle/QuickSpeak
```

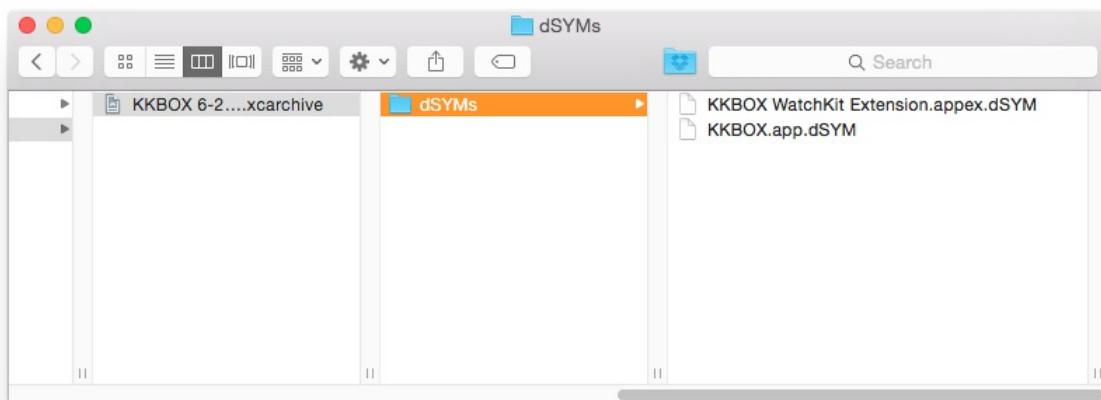
解開記憶體位置

Debug Symbol

要解開 crash report 中的記憶體位置，必須要有 debug symbol。為了在閱讀 crash report 時可以找得到 debug symbol，我們建議盡可能保留所有的 build，在提供 QA 或是公司內部其他成員測試的時候，應該要透過統一的入口。

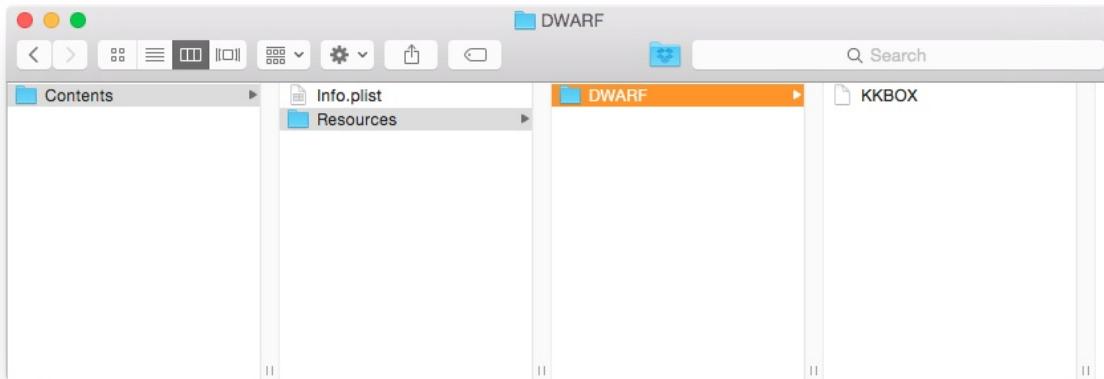
如果有位同仁找了一位 iOS 工程師安裝版本，這位工程師直接用 Xcode 跑了 run 安裝一個版本，發生 crash 的時候，這位同仁可能會找另外一位 iOS 工程師解決，這種狀況下，可能根本搞不清楚當時裝的是哪個版本，用的是哪個版本的 SDK 以及 Xcode—不同版本 SDK 與 Xcode 也會編出不同的 binary。我們建議在發行內部版本時，會透過同一台 build machine 編譯，像是架設內部的 Jenkins 等持續整合服務，或是透過 Crashlytics 或 HockeyApp 發佈內部版本。

當你在 Xcode 選擇 Product->Archive 之後，我們可以從 Organizer window 中找到剛剛建立好的 archive，debug symbol 就放在 archive 裡頭。我們對 archive 檔案按右鍵，從右鍵選單中選擇「Show Package Contents」，就會跳出這個 archive 的內容，debug symbol 就放在 dSyms 目錄下。



一個 App 可能有不只一個 debug symbol，像 KKBOX 除了主程式之外，還做了 Apple Watch 的 extension，所以也會有屬於 Apple Watch 這一端的 debug symbol。

而我們在這邊看到的兩個 .dSYM 其實也都是 bundle，我們又要用右鍵選單的「Show Package Contents」查看裡頭的內容。



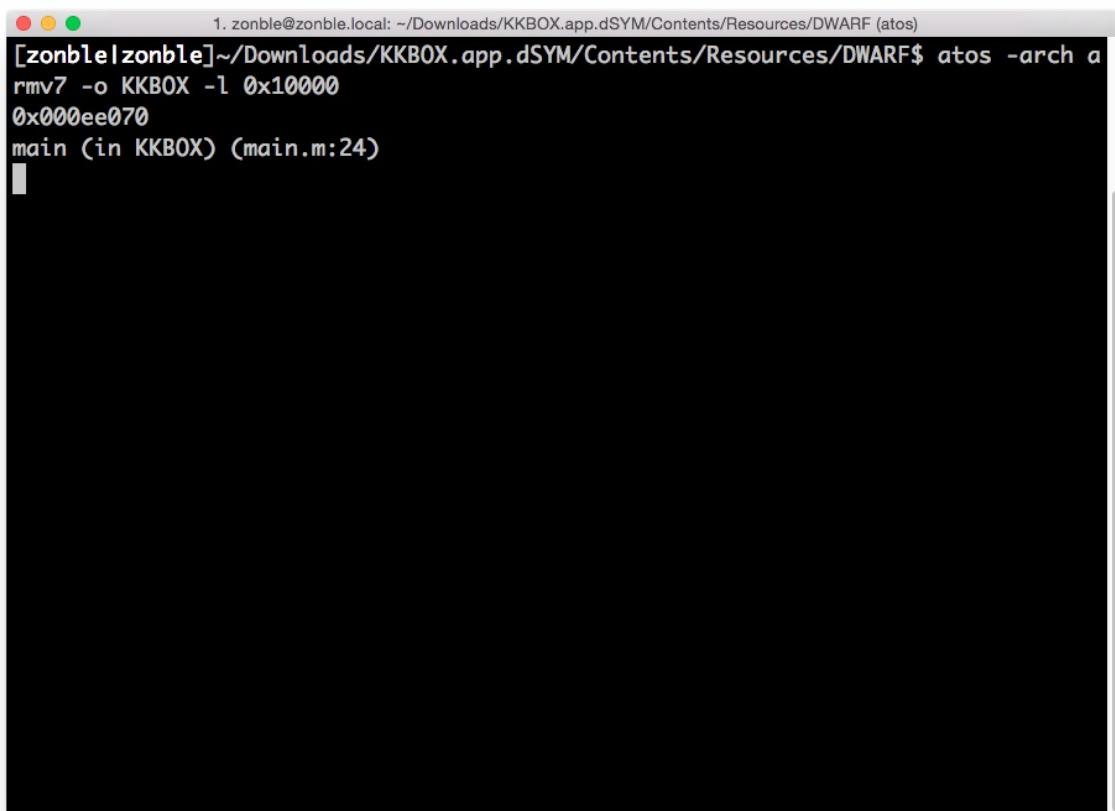
以 KKBOX 來說，在 KKBOX.app.dSYM 下的 Contents/Resources/DWARF/KKBOX，才是我們最後想使用的 debug symbol 檔案。

atos

找到 debug symbol 後，我們現在要用 atos (address to symbol) 這個 command line 指令。重要參數包括：

- -arch，我們要使用哪個 architecture。在這邊我們輸入 armv7，如果是 64 位元環境則輸入 arm64；如果是解決 Mac 的問題，則可能是 x86_64 或 i386。
- -o，debug symbol 檔案。我們傳入剛才找到的 Contents/Resources/DWARF/KKBOX 的完整路徑。
- -l，KKBOX 被載入到哪段記憶體位置。在前一章當中，我們知道是在 0x10000 - 0xc1bfff，所以輸入 0x10000。

我們首先會輸入 Thread 0 裡頭 call stack 中位在 14 的 0x000ee070 這個記憶體位置。在 main thread 中，call stack 的底層一定是 `main`，如果不是 `main`，就代表我們找錯 debug symbol 了。



A terminal window showing the output of the 'atos' command. The command is: 'atos -arch armv7 -o KKBOX -l 0x10000 0x000ee070'. The output shows the assembly code for the main function at address 0x000ee070.

```
1. zonble@zonble.local: ~/Downloads/KKBOX.app.dSYM/Contents/Resources/DWARF (atos)
[zonble@zonble]~/Downloads/KKBOX.app.dSYM/Contents/Resources/DWARF$ atos -arch armv7 -o KKBOX -l 0x10000
0x000ee070
main (in KKBOX) (main.m:24)
```

看起來沒錯。接著我們可以把整個 exception 的 call stack 貼進去。

```

1. zonble@zonble.local: ~/Downloads/KKBOX.app.dSYM/Contents/Resources/DWARF (atos)
0x000ee070
main (in KKBOX) (main.m:24)
0x23e61132 0x321dec72 0x23e665f8 0x23e644d4 0x23d939d4 0x23e40272 0x24aef96 0x2
4aefc46 0x755ba6 0x2776be1c 0x2776bede 0x27760a60 0x2756f2b6 0x27498c16 0x26eb74
40 0x26eb2c90 0x26eb2b18 0x26eb24ba 0x26eb22aa 0x26f05ab8 0x2bad5bfe 0x24dd9d08
0x23e16550 0x23e26a46 0x23e269e2 0x23e25004 0x23d7099c 0x23d707ae 0x2b5201a4 0x2
74fb690 0xee070 0x32786aaa)^H
0x23e61132
0x321dec72
0x23e665f8
0x23e644d4
0x23d939d4
0x23e40272
0x24aef96
0x24aefc46
-[KKIPadMyboxUserSearchTableViewController tableView:cellForRowAtIndexPath:] (in
KKBOX) (KKIPadMyboxUserSearchTableViewController.m:128)
0x2776be1c
0x2776bede
0x27760a60
0x2756f2b6
0x27498c16
0x26eb7440
0x26eb2c90
0x26eb2b18
0x26eb24ba
0x26eb22aa
0x26f05ab8
0x2bad5bfe
0x24dd9d08
0x23e16550
0x23e26a46
0x23e269e2
0x23e25004

```

由於我們的 debug symbol 只會包含 KKBOX，不包含 UIKit 等系統 library，所以只會顯示屬於 KKBOX 的問題。總之，我們問題出在 KKIPadMyboxUserSearchTableViewController 這個 class 裡頭的

`tableView:cellForRowAtIndexPath:` 裡頭，位在 KKIPadMyboxUserSearchTableViewController.m 第 128 行。

修正問題

接著我們就可以去檢查 KKIPadMyboxUserSearchTableViewController.m 第 128 行做了什麼，我們發現，這是 KKBOX 的用戶搜尋功能中，用來顯示搜尋結果的相關部分，我們預期用戶的名稱會是字串。搭配 QA 提供給我們的 console log

```

Info: *** Terminating app due to uncaught exception 'NSInvalidArgumentException', reason: '-[NSNull
length]: unrecognized selector sent to instance 0x3a866830'

```

看起來我們預期的是字串，但是 server 的 API 提供的 JSON 回應中，用戶名稱卻是 null。我們在 Objective-C 語言中，會把 JSON 的 null 轉成 NSNull 型別，所以呼叫屬於字串的 `length` 這個 method 時，就會出現找不到 selector 的錯誤。所以，我們在這邊要做一些型別判斷，當用戶名稱是 null 的時候，要用別的方式顯

示。

說起來 NSNull 還頂討厭的，有時候我們會希望就算程式中任何地方出現 NSNull，都不會 crash。由於跟 NSNull 有關的 crash 往往是找不到 selector，如果想要一勞永逸：讓 NSNull 可以回應所有的 selector，如何？

我們從第一章中了解 Objective-C 的動態特性，知道可以對任何 class 添加 method，最簡單的方法就是使用 category；Objective-C 物件還有另外一個特性：如果一個 class 並沒有某個 selector，在找不到 selector 的時候，會先透過 `forwardInvocation:`，詢問這個 class 要不要把這次的呼叫交給別的物件處理。

透過這兩個特性，就可以來施點動態魔法：透過 category 實作 NSNull 的 `forwardInvocation:`。

```
@interface NSNull (SafeNull)

@end

@implementation NSNull (SafeNull)

- (NSMethodSignature *)methodSignatureForSelector:(SEL)aSelector
{
    NSMethodSignature *signature = [super methodSignatureForSelector:aSelector];
    if (!signature) {
        NSMethodSignature *sig = [NSMethodSignature signatureWithObjCTypes:@encode(void)]
    ;
        return sig;
    }
    return signature;
}

- (void)forwardInvocation:(NSInvocation *)anInvocation
{
}

@end
```

常見 Crash 的類型

在蘋果官方文件 [Technical Note TN2151 Understanding and Analyzing iOS Application Crash Reports](#) 上，可以看到完整的錯誤說明，當中最常見的是 Bad Memory Access (EXC_BAD_ACCESS / SIGSEGV / SIGBUS) 與 Abnormal Exit (EXC_CRASH / SIGABRT) 這兩項。如果遇到了在這之外的錯誤，可以參考前述蘋果文件，尤其是像錯誤代碼為 00000020 這類的「其他錯誤」，大概也就只有這篇文章可以參考，去 Stack Overflow 也不見得可以找到答案。

Bad Memory — 記憶體錯誤

在 ARC 問世之後，這樣的問題已經少了很多，不然在 iOS 5 之前，記憶體錯誤幾乎佔所有 crash 的最大宗。記憶體錯誤代表的是我們嘗試使用一個不正確的記憶體指標，在 crash log 的 Exception Sub-code 這段，會出現當時嘗試使用的記憶體位置。

最常見的記憶體問題就是一個 Objective-C 物件的 retain 與 release 不成對，一個物件已經 retain count 為 0 了，我們還繼續要求這個物件 release；或是一個變數在 release 的時候沒有指向 nil，所以這個變數所指向的物件已經 retain count 為 0 了，我們還嘗試呼叫，於是呼叫到錯誤的記憶體。此外也包含 C 的記憶體錯誤，像還沒有 alloc 一塊記憶體就先呼叫。

雖然有了 ARC 之後記憶體問題少很多，但還是會發生。我們在 [記憶體管理 Part 1](#) 與 [記憶體管理 Part 2 - ARC](#) 討論了不少相關議題，在這邊就不重複。

要修正記憶體管理問題，找到 crash 發生在哪一行是第一步，可以找到是哪個物件、或是哪塊記憶體出問題，不過要修正的不見得就是直接發生 crash 的那一行：一個物件或一塊記憶體產生之後，往往會在很多地方使用過，所以 retain、release 不成對的狀況很有可能發生在 crash 的地方之前。

例如，我們現在寫一個手動管理記憶體的 UIViewController，有個叫做 button 的成員變數，我們在 loadView 的地方我們手動寫了一行 `button = [UIButton buttonWithType:UIButtonTypeCustom]`，之後把 button 變成 self.view 的 subview。在這個 UIViewController 的 dealloc 的地方，我們寫了 `[button release]`，結果發生了 crash，要修正的可能就不是這行 `[button release]`，而是一開始要把 button retain 一份，寫成 `button = [[UIButton buttonWithType:UIButtonTypeCustom] retain]`。

要找到記憶體在什麼地方不成對，可以用 Instrument 的 Zombie 這項設定做 profiling。

Abnormal Exit - 發生了 Exception

只要程式中有地方發生了 NSEException throw，或是沒有達到 NSAssert 的條件，就會觸發這種錯誤，前一節的 NSNull 問題就是這種。遇到這種錯誤，首先要看的不是 crash 的 thread，要去看「Last Exception Backtrace」，以及 console 上的訊息。

常見 Exception 包括：

找不到 selector

出現這種錯誤的時候，會跳出「unsupported selector」錯誤訊息。

這種錯誤的原因是，我們期待操作的物件，與實際上拿到的物件不一樣。我們想要一個 Array 的時候可能拿到字串，想要拿到字串的時候卻拿到 NSNull，或是我們期待的是一個 mutable 的物件，結果拿到的卻是 immutable 的。於是，這個物件沒有我們期待的 selector 可以使用。

說起來這個問題是 Objective-C 這個語言天生的問題：所有的物件都可以 cast 成 id，然後一個物件放進 array 或 dictionary 拿出來之後，也無法確實確認是哪種型別。以下面這行 code 來說：

```
NSString *s = [aDict objectForKey:@"key"];
```

我們根本不能相信 s 一定是 NSString，所以就會寫一堆這樣的 code：

```
NSString *s = [aDict objectForKey:@"key"];
if ([s isKindOfClass:[NSString class]]) {
    // 繼續做事
}
```

如果不這麼寫，就有可能發生 crash。真的要解決問題，第一個方法就是，我們以後就別寫 Objective-C 了，直接改寫 Swift，一方面 Swift 的 array 與 dictionary 可以透過 Generics 語言特性指定裡頭的物件型態，再來 Swift 語法中會經常強迫我們確認物件型別，在 Swift 中，我們可能會寫出大量的 if let 語法：

```
if let s = aDict["key"] as? NSString {
    // 繼續做事
}
```

再蘋果在 WWDC 2015 中，宣布 Objective-C 也可以選用 Generics 語法，應該也會有一些幫助。

另外一個方式是，我們盡量避免直接使用 NSArray 或 NSDictionary 當 model，而是在這些物件上另外包裝一層我們自己的 model 物件，在想要取用某個 property 的時候，這個 model class 會做好型別的判斷，確實回傳符合型別的物件。像 GitHub 推出的 open source 專案 [Mantle](#)，就可以幫助我們撰寫這類的 model 物件，在這個專案的設計中，透過大量的 transformer 物件，讓每個 property 都轉出正確的形態。

nil 的操作

無論是對 NSMutableArray 或 NSMutableDictionary 插入 nil，都會發生 crash。要避免這個問題，就是在做插入的動作之前，都先檢查一下現在要插入的物件是否是 nil；或是，如果你使用 Xcode 6.3 之後的版本，也可以使用 nullable 、 nonnull 等關鍵字，確認使用的變數是否是 nil。

要不然就是改寫 Swift：Swift 語法特別強調一個變數是否可以指向 nil，這項特性叫做 Optional，一個可以指向 nil 的變數必須設成 Optional，也就是變數後方必須加上一個問號，而這個變數以後每次出現，後方都一定會出現問號與驚嘆號。而 Objective-C 的 nullable 、 nonnull 等關鍵字其實就是為了與 Swift API 一致。

Out of Bounds

如果一個 array 只有兩筆資料，但我們卻去要第三筆資料，就會產生 out of bounds 錯誤。

一邊 enumerate 一邊改動 array

假如我們一邊 enumerate 一個 array，一邊改動它，就會跳出 exception。像我們想要把一個 mutable 的字串 array 中，長度小於 3 的字串都拿掉，如果像以下這種寫法就會 crash：

```
for (NSString *s in array) {
    if ([s length] < 3) {
```

```
[array removeObject:s]
}
}
```

我們可以先把想刪除的物件先放到另外一個 array 中，再告訴原本的 array 要刪除哪些東西。

```
NSMutableArray *arrayToDelete = [NSMutableArray array];
for (NSString *s in array) {
    if ([s length] < 3) {
        [arrayToDelete addObject:s];
    }
}
[array removeObjectsInArray:arrayToDelete];
```

不過，如果我們想做的事情是想把一些東西從某個 array 濾掉，也可以考慮改用 NSPredicate。上面的 code 實際意思也就是：把長度大於 2 的字串留下來。

```
NSPredicate *predicate = [NSPredicate predicateWithFormat:@"SELF.length > 2"];
[a filterUsingPredicate:predicate];
```

UIKit 中的 assertion

UIKit 中有不少跟資料一致性相關的 assertion。當我們要求一個 table view 刪除或加入某些列、同時帶有動畫效果的時候（透過呼叫 `-insertRowsAtIndexPaths:withRowAnimation:` 與 `-deleteRowsAtIndexPaths:withRowAnimation:` 這些 method），如果 table view 的 data source 沒有對應的變化—像原本 table view 裡頭有六列，我們要求刪除一列，但 table view 的 data source 並沒有變成五列，那麼就會造成 table view crash。

所以在遇到經常變動的 model 的時候，我們需要考慮關閉動畫效果。以 KKBOX 的歌單功能來說，我們除了可以讓用戶手動編輯歌單之外，歌單的內容也可能因為背景的同步作業、或是下載歌曲的狀態改變而更動；如果在 table view 中出現動畫的時候，發生這些狀況，就會 crash。

此外，在使用 UIKit 的各種元件的時候，我們要對 0.25 秒這個時間保持敏感，絕大多數在 UIKit 中的動畫效果都是 0.25 秒，像上面提到的 table view 新增或刪除 row 的動畫、UINavigationController push 或 pop view controller 的動畫，鍵盤升起的動畫，以及 present modal view 的動畫（這個在 iOS 7 之後倒是有一些改變）等等。如果在一個動畫執行到一半的時候，我們的 App 又要做一件跟這件動畫相反的事情（像 navigation controller push 的動畫還沒做完，我們就叫它 pop），狀況好一點，是 view hierarchy 會變亂，畫面變得亂七八糟，狀況不好就是直接 crash 了。

禪與 App 維修的藝術

幾年前我讀了一本叫做《禪與摩托車維修的藝術》的書。這本不算薄書大概的內容是這樣：作者不斷思考當代人的處境，發現當代人處在一種前所未有的無力感中，而這種無力感的來源是科技。

科技的發展一方面讓人享受便利，但另一方面卻讓人更無力，現代人離不開科技，但科技並不透命，科技的複雜卻讓人難以完全理解科技，人們無法完全理解每天都在使用的事物背後是怎麼運作的。當某項科技出了差錯，某台機器故障，人們不知道應該從何下手，因為科技，人們失去了掌握自己生活中種種的權力。

後來作者又講到，從西方傳統的科學哲學，也無法解決這種處境。科學精神就在於觀測與證明，而現在有太多的現象無法觀測，也無法給予直接的證明—我們和絕大多數的台灣人一樣缺乏西方科學哲學訓練，姑且跳過這段討論。總之，作者認為在二十世紀應該要有一門新的哲學，這門哲學的精神是鼓勵人們觀察事物的背後，了解科技背後如何運作，人們有能力對自己的生活重新掌權，而如果無法直接觀測，那麼，我們就該重新強調西方科學傳統被抗拒的直覺與想像力。

用戶使用我們的 App，還有我們在修 bug 的時候，往往也陷入這種無力感：用戶可能天天都在用我們的 App，但也搞不清楚我們是怎麼把 App 寫出來的，App 有問題不知道怎麼辦。我們在修 bug 的時候，也經常搞不清楚用戶是在怎樣的環境下執行，客訴如雪片般飛來我們卻無法重現問題，我們也沒辦法看到蘋果的程式碼，不見得知道蘋果的 library 到底做了什麼，當問題發生的時候，我們和用戶一樣束手無策。

那我們能做什麼呢？我們的眼光應該要從那些眩目的事物上稍微移開，我們或許會知道很多第三方 library，知道很多 cocoapods 可以用，我們可以用很快的速度拼裝出一個 App，但我們要了解被系統 library 隔絕開來的底下是怎麼運作，像 Objective-C run time 是怎麼回事，才有辦法理解問題，同時細心觀察我們手上有哪些線索，像 crash report，進而發揮想像力，想像問題出現的時候，到底發生了什麼。

接下來我們會進入幾個實戰案例。

實戰：Bad Access

因為在 ARC 問世後，Bad Access 造成的 crash 的確少很多，所以在這邊放的是一份在 2012 年時為了解釋 Bad Access 是如何形成而留下來的 crash report，但內容不是很完整，只保留了發生 Bad Access 時的 thread。

這是在 2012 年時 Pinterest App 發生的 crash，可以注意到當時是在 iPhone 4S 上使用 iOS 5 作業系統。在這份 crash report 中，可以清楚看到 iOS 5 之前如何處理記憶體不足警告的流程。

從 run loop 中，UIApplication 首先收到了記憶體不足警告，UIApplication 接著就透過 Notification Center 通知所有的 view controller 記憶體不足，我們在 [記憶體管理 Part 3 - Memory Warnings](#) 提過，iOS 5 之前，只要發生記憶體警告，就會要求所有不是在最前景的 view controller 將自己的 view 釋放掉，`purgeMemoryForReason:` 與 `unloadViewForced:` 在做的，就是強迫釋放 view 這件事情。在強迫釋放 view 發生 crash，於是是可以推斷，就算我們沒有把 Pinterest 的 crash report 解開，也可以看出，`0x0005995e` 這個位置會是某個 view controller 的 `viewDidUnload`。

雖然 iOS 5 就推出了 ARC，但當時很多在 iOS 5 問世之前就寫出來的 App，並沒有立刻轉換到 ARC 架構上，這份 crash report 呈現了當時這段轉換期。另外，我們在 [常見 Crash 的類型](#) 也提到，在 `viewDidUnload` 最容易發生的，就是忘記像 UIButton 這種建立時就是 auto release 的物件給 retain 起來，然後在 `viewDidUnload` 的時候多 release 了一次。

```

Incident Identifier: 3486ADCD-070E-43C8-ADC0-44E254DB92E8
CrashReporter Key: babb1c6e8923eb91911e323103f4d82fa0bc7fe2
Hardware Model: iPhone4,1
Process: Pinterest [12210]
Path: /var/mobile/Applications/4BFAD77B-FCE9-4EE1-A36D-ADFA55303130/Pinterest.app/Pinterest
Identifier: Pinterest
Version: ??? (??)
Code Type: ARM (Native)
Parent Process: launchd [1]

Date/Time: 2012-03-20 09:19:27.054 +0800
OS Version: iPhone OS 5.1 (9B179)
Report Version: 104

Exception Type: EXC_BAD_ACCESS (SIGSEGV)
Exception Codes: KERN_INVALID_ADDRESS at 0x20aba07b
Crashed Thread: 0

Thread 0 name: Dispatch queue: com.apple.main-thread
Thread 0 Crashed:
0 libobjc.A.dylib 0x35839f78 objc_msgSend + 16
1 Pinterest 0x0005995e 0x1000 + 362846
2 UIKit 0x33a84bd8 -[UIViewController unloadViewForced:] + 24
4
3 UIKit 0x33bcc492 -[UIViewController purgeMemoryForReason:] + 58
4 Pinterest 0x000a2df4 0x1000 + 663028
5 Pinterest 0x000594f0 0x1000 + 361712

```

```
6 Foundation          0x37b634f8 __57-[NSNotificationCenter addObserver:sele
ctor:name:object:]_block_invoke_0 + 12
7 CoreFoundation      0x37305540 __CFXNotificationPost_block_invoke_0 + 64
8 CoreFoundation      0x37291090 _CFXNotificationPost + 1400
9 Foundation          0x37ad73e4 -[NSNotificationCenter postNotificationName:object:userInfo:] + 60
10 Foundation         0x37ad8c14 -[NSNotificationCenter postNotificationName:object:] + 24
11 UIKit              0x33b9726a -[UIApplication _performMemoryWarning] + 7
4
12 UIKit              0x33b97364 -[UIApplication _receivedMemoryWarning] + 168
13 libdispatch.dylib   0x345942da _dispatch_source_invoke + 510
14 libdispatch.dylib   0x34591b7a _dispatch_queue_invoke$VARIANT$mp + 46
15 libdispatch.dylib   0x34591eba _dispatch_main_queue_callback_4CF$VARIANT$mp + 150
16 CoreFoundation      0x3730c2a6 __CFRunLoopRun + 1262
17 CoreFoundation      0x3728f49e CFRunLoopRunSpecific + 294
18 CoreFoundation      0x3728f366 CFRunLoopRunInMode + 98
19 GraphicsServices    0x3219c432 GSEventRunModal + 130
20 UIKit              0x33a13e76 UIApplicationMain + 1074
21 Pinterest          0x0000328a 0x1000 + 8842
22 Pinterest          0x00003248 0x1000 + 8776
```

實戰：因為 Category 造成的 Crash

KKBOX 在 2014 年十月推出 iOS 版本 6.0.26 版本，推出之後，就收到不少客訴反應應用程式在執行到特定的地方—瀏覽線上精選畫面—時會發生 crash。這件事情非常奇怪，因為這個版本在我們的開發到 QA 驗證的過程中，從來就沒有在這個地方發生過 crash，如果按照客訴在電話中的描述，我們也完全無法重現問題。

所幸我們可以收集到來自用戶的 crash report。

看到這份 crash log 的第一印象就是用戶做了 JB。用戶用的機種明明就是 iPhone 5S，應該要執行 armv7s 的 library，但是卻載入了一大堆 armv6 的 library，而你看到 CydiaSubstrate 的時候，心底也有個底了。

接著來看 crash 類型，是在 `-[__NSPlaceholderDictionary initWithObjects:forKeys:count:]` 裡頭發生了 exception，`__NSPlaceholderDictionary` 是 `NSDictionary` 的內部實作，當我們在建立 `NSDictionary` 的時候，Foundation 實質會回傳的是另外一個介面相同的 subclass 回來。至於 `NSDictionary` 會產生的 exception 幾乎都跟 nil 有關—嘗試把 nil 插入到 `NSDictionary` 裡頭造成的。

我們收集到了 exception 發生時的 console log：

```
*** Terminating app due to uncaught exception 'NSInvalidArgumentException', reason: '*** -[__NSPlaceholderDictionary initWithObjects:forKeys:count]: attempt to insert nil object from objects[2]'
```

我們嘗試建立 Dictionary 的時候，第三筆傳遞進去的資料（objects[2]）是 nil。我們繼續根據解開來的位置，檢查一下 `KKExploreCardCollectionItemCell.m` 的第 159 行，`drawRect:` 這個 method。

`KKExploreCardCollectionItemCell` 是線上精選頁面 collection view 裡頭使用到的一個 cell，符合「瀏覽線上精選」會 crash 這條描述。

以下程式從 142 行開始

```
- (void)drawRect:(CGRect)rect
{
    UIBezierPath *path = [UIBezierPath bezierPathWithRoundedRect:self.bounds cornerRadius:3.0];
    [[UIColor whiteColor] set];
    [path fill];
    [path addClip];

    NSMutableParagraphStyle *paragraphStyle = [[NSMutableParagraphStyle alloc] init];
    paragraphStyle.alignment = NSTextAlignmentLeft;

    if (self.subtitle.length) {
        CGSize titleSize = [self.title boundingRectWithSize:CGSizeMake(self.frame.size.width - 22, 20.0) options:NSStringDrawingTruncatesLastVisibleLine | NSStringDrawingUsesLineFragmentOrigin attributes:@{NSFontAttributeName: [UIFont systemFontOfSize:14.0], NSParagraphStyleAttributeName: paragraphStyle} context:nil].size;
        CGRect titleRect = CGRectMake(11.0, CGRectGetMaxY(imageFrame) + 10.0, titleSize.width, titleSize.height);
        [self.title drawWithRect:titleRect options:NSStringDrawingTruncatesLastVisibleLine | NSStringDrawingUsesLineFragmentOrigin attributes:@{NSFontAttributeName: [UIFont systemFontOfSize:14.0], NSParagraphStyleAttributeName: paragraphStyle, NSForegroundColorAttributeName: [UIColor blackColor]} context:nil];
    }
}
```

```

    CGSize subtitleSize = [self.subtitle boundingRectWithSize:CGSizeMake(self.frame.size.width - 22, 20.0) options:NSStringDrawingTruncatesLastVisibleLine | NSStringDrawingUsesLineFragmentOrigin attributes:@{NSFontAttributeName: [UIFont systemFontOfSize:12.0], NSParagraphStyleAttributeName: paragraphStyle} context:nil].size;
    CGRect subtitleRect = CGRectMake(11.0, self.frame.size.height - 10.0 - subtitleSize.height, subtitleSize.width, subtitleSize.height);
    [self.subtitle drawWithRect:subtitleRect options:NSStringDrawingTruncatesLastVisibleLine | NSStringDrawingUsesLineFragmentOrigin attributes:@{NSFontAttributeName: [UIFont systemFontOfSize:12.0], NSParagraphStyleAttributeName: paragraphStyle, NSForegroundColorAttributeName: [UIColor colorWithRed:0.2 green:0.4 blue:0.6 alpha:1.0]} context:nil];
}
else {
    CGSize titleSize = [self.title boundingRectWithSize:CGSizeMake(self.frame.size.width - 22, 40.0) options:NSStringDrawingTruncatesLastVisibleLine | NSStringDrawingUsesLineFragmentOrigin attributes:@{NSFontAttributeName: [UIFont systemFontOfSize:14.0], NSParagraphStyleAttributeName: paragraphStyle} context:nil].size;
    CGRect titleRect = CGRectMake(11.0, CGRectGetMaxY(imageFrame) + 11.0, titleSize.width, titleSize.height);
    [self.title drawWithRect:titleRect options:NSStringDrawingTruncatesLastVisibleLine | NSStringDrawingUsesLineFragmentOrigin attributes:@{NSFontAttributeName: [UIFont systemFontOfSize:14.0], NSParagraphStyleAttributeName: paragraphStyle, NSForegroundColorAttributeName: [UIColor blackColor]} context:nil];
}
}

```

當中第 159 行是

```
[self.subtitle drawWithRect:subtitleRect options:NSStringDrawingTruncatesLastVisibleLine | NSStringDrawingUsesLineFragmentOrigin attributes:@{NSFontAttributeName: [UIFont systemFontOfSize:12.0], NSParagraphStyleAttributeName: paragraphStyle, NSForegroundColorAttributeName: [UIColor colorWithRed:0.2 green:0.4 blue:0.6 alpha:1.0]} context:nil];
```

在這一行中，我們要把 subtitle 這個字串畫到畫面中，我們傳入了三個樣式設定：字體要是 12 point 大小、一個特定的段落樣式、以及將顏色設定成「#888888」，傳入三個樣式這件事情，符合前面「建立一個 Dictionary」的條件，那麼，object[2] 就會是 `[UIColor colorWithRed:0.2 green:0.4 blue:0.6 alpha:1.0]` 傳回的結果了。

`[UIColor colorWithRed:0.2 green:0.4 blue:0.6 alpha:1.0]` 為什麼會變成 nil？

`colorWithHexString:` 是 KKBOX 使用到的一個 UIColor category，可以根據傳入的色碼（Hex code）產生 UIColor 物件，平常使用都沒問題，為什麼會在用戶的環境裡頭變成 nil？

還記得我們在第一章就提到，在 Objective-C 裡頭，由於 Objective-C 的動態特性，所以每一個 method 都有可能在 run time 被換掉？由於用戶 JB 過，所以可以載入額外的 library，很有可能在用戶所載入的眾多 library 中（至於具體來說是那一個呢？鬼才曉得），也有名稱一樣叫做 `colorWithHexString:` 的 method，把我們原本的實作換掉了，更換之後的 `colorWithHexString:` 實作並不認得 @"#888888"，回傳 nil，接著我們又把 nil 插入 NSDictionary。

你可能聽說過，JB 會造成系統不穩定，這就是 JB 造成系統不穩定的好例子：因為 JB 之後，額外安裝的 library 會導致軟體本身的行為改變，超過了開發人員的預料，於是不會 crash 的地方也 crash 了。

那我們可以做什麼呢？我們可以把自己這份 `colorWithHexString:` 換個名字，避免與別人載入的 `UIColor` category 同名，所以現在許多人也建議在 category 名稱前方加上自己的 prefix，我們可能要改名叫做 `kk_colorWithHexString:`，但，這麼做的目的是為了避免讓 App 在用戶 JB 過的環境下 crash，著實讓人產生強烈的無力感。

而這種問題在測試階段是無法發現的。我們不可能測試所有 JB 之後的各種環境的組合。

```

Incident Identifier: AB2474CA-D530-4CAF-A539-741C7AC07387
CrashReporter Key: F39417F0-0875-4CD7-A31D-F303D5C74AF4
Hardware Model: iPhone5,2
Process: KKBOX [3261]
Path: /Users/USER/KKBOX.app/KKBOX
Identifier: tw.com.skysoft.iPhone
Version: 6.0.26
Code Type: ARM
Parent Process: launchd [1]

Date/Time: 2014-10-25T23:22:15Z
OS Version: iPhone OS 7.0.2 (11A501)
Report Version: 104

Exception Type: SIGABRT
Exception Codes: #0 at 0x38ef41fc
Crashed Thread: 0

Application Specific Information:
*** Terminating app due to uncaught exception 'NSInvalidArgumentException', reason: '*** -[__NSPlaceholderDictionary initWithObjects:forKeys:count:]: attempt to insert nil object from objects[2]'
Last Exception Backtrace:
0 CoreFoundation 0x2e649e8b __exceptionPreprocess + 131
1 libobjc.A.dylib 0x389446c7 objc_exception_throw + 36
2 CoreFoundation 0x2e587aef -[__NSPlaceholderDictionary initWithObjects:forKeys:count:] + 532
3 CoreFoundation 0x2e5878b3 +[NSDictionary dictionaryWithObjects:forKeys:count:] + 48
4 KKBOX 0x002989a9 -[KKExploreCardWithImageAndCaptionCell drawRect:] (KKExploreCardCollectionItemCell.m:159)
5 UIKit 0x30e4f749 -[UIView(CALayerDelegate) drawLayer:inContext:] + 370
6 QuartzCore 0x30a86049 -[CALayer drawInContext:] + 98
7 QuartzCore 0x30a6f813 CABackingStoreUpdate_ + 1856
8 QuartzCore 0x30b49735 __ZN2CA5Layer8display_Ev_block_invoke + 50
9 QuartzCore 0x30a6f0c3 x_blame_allocations + 80
10 QuartzCore 0x30a6ed77 CA::Layer::display_() + 1116
11 QuartzCore 0x30a52969 CA::Layer::display_if_needed(CA::Transaction*) + 206
12 QuartzCore 0x30a52601 CA::Layer::layout_and_display_if_needed(CA::Transaction*) + 22
13 QuartzCore 0x30a5200d CA::Context::commit_transaction(CA::Transaction*) + 226
14 QuartzCore 0x30a51e1f CA::Transaction::commit() + 312
15 UIKit 0x30dc786b _afterCACCommitHandler + 124
16 CoreFoundation 0x2e614f71 __CFRunLoop_IS_CALLING_OUT_TO_AN_OBSERVER_CALLBACK_FUNCTION__ + 18
17 CoreFoundation 0x2e6128ff __CFRunLoopDoObservers + 284
18 CoreFoundation 0x2e612c4b __CFRunLoopRun + 736

```

```

19 CoreFoundation          0x2e57d541 CFRunLoopRunSpecific + 522
20 CoreFoundation          0x2e57d323 CFRunLoopRunInMode + 104
21 GraphicsServices        0x332b42eb GSEventRunModal + 136
22 UIKit                  0x30e341e5 UIApplicationMain + 1134
23 KKBOX                  0x000e012d main (main.m:24)
24 libdyld.dylib           0x38e3dab7 start + 0

Thread 0 Crashed:
0 libsystem_kernel.dylib   0x38ef41fc __pthread_kill + 8
1 libsystem_pthread.dylib  0x38f5ba53 pthread_kill + 56
2 libsystem_c.dylib         0x38ea502d abort + 74
3 KKBOX                   0x00514937 uncaught_exception_handler + 24
4 CoreFoundation           0x2e64a18d __handleUncaughtException + 578
5 libobjc.A.dylib          0x38944927 objc_terminate() + 172
6 libc++abi.dylib          0x3830a1b3 std::__terminate(void (*)()) + 76
7 libc++abi.dylib          0x38309d17 __cxa_rethrow + 100
8 libobjc.A.dylib          0x3894480f objc_exception_rethrow + 40
9 CoreFoundation           0x2e57d5b7 CFRunLoopRunSpecific + 640
10 CoreFoundation          0x2e57d323 CFRunLoopRunInMode + 104
11 GraphicsServices        0x332b42eb GSEventRunModal + 136
12 UIKit                  0x30e341e5 UIApplicationMain + 1134
13 KKBOX                  0x000e012d main (main.m:24)
14 libdyld.dylib           0x38e3dab7 start + 0

Thread 1:
0 libsystem_kernel.dylib   0x38ee1838 kevent64 + 24
1 libdispatch.dylib         0x38e2a643 _dispatch_mgr_thread + 36

Thread 2:
0 libsystem_kernel.dylib   0x38ee1a84 mach_msg_trap + 20
1 CoreFoundation           0x2e614561 __CFRunLoopServiceMachPort + 154
2 CoreFoundation           0x2e612c81 __CFRunLoopRun + 790
3 CoreFoundation           0x2e57d541 CFRunLoopRunSpecific + 522
4 CoreFoundation           0x2e57d323 CFRunLoopRunInMode + 104
5 Foundation               0x2ef6b827 -[NSRunLoop(NSRunLoop) runMode:beforeDate:] + 252
6 Foundation               0x2efbc669 -[NSRunLoop(NSRunLoop) run] + 78
7 KKBOX                   0x0052b099 +[GAI threadMain:] + 62
8 Foundation               0x2f02ddc7 __NSThread__main__ + 1060
9 libsystem_pthread.dylib  0x38f5ac5d _pthread_body + 138
10 libsystem_pthread.dylib  0x38f5abcf _pthread_start + 100
11 libsystem_pthread.dylib  0x38f58cd0 thread_start + 6

Thread 3:
0 libsystem_kernel.dylib   0x38ef4c7c __workq_kernreturn + 8
1 libsystem_pthread.dylib  0x38f58cc4 start_wqthread + 6

Thread 4:
0 libsystem_kernel.dylib   0x38ee1a84 mach_msg_trap + 20
1 CoreFoundation           0x2e614561 __CFRunLoopServiceMachPort + 154
2 CoreFoundation           0x2e612c81 __CFRunLoopRun + 790
3 CoreFoundation           0x2e57d541 CFRunLoopRunSpecific + 522
4 CoreFoundation           0x2e57d323 CFRunLoopRunInMode + 104

```

```

5 Foundation 0x2ef6b827 -[NSRunLoop(NSRunLoop) runMode:before
Date:] + 252
6 Foundation 0x2ef8b3bb -[NSRunLoop(NSRunLoop) runUntilDate:]
+ 84
7 KKBOX 0x002827f1 -[KKCloudPlaylistPullFullOperation ma
in] (KKCloudPlaylistPullFullOperation.m:98)
8 Foundation 0x2ef78c35 -[__NSOperationInternal _start:] + 77
0
9 Foundation 0x2f01caf0 __NSOQSchedule_f + 58
10 libdispatch.dylib 0x38e2de77 _dispatch_queue_drain + 372
11 libdispatch.dylib 0x38e2af9b _dispatch_queue_invoke + 40
12 libdispatch.dylib 0x38e2e751 _dispatch_root_queue_drain + 74
13 libdispatch.dylib 0x38e2e9d1 _dispatch_worker_thread2 + 54
14 libsystem_pthread.dylib 0x38f58dff _pthread_wqthread + 296
15 libsystem_pthread.dylib 0x38f58cc4 start_wqthread + 6

Thread 5:
0 libsystem_kernel.dylib 0x38ee1a84 mach_msg_trap + 20
1 CoreFoundation 0x2e614561 __CFRunLoopServiceMachPort + 154
2 CoreFoundation 0x2e612c81 __CFRunLoopRun + 790
3 CoreFoundation 0x2e57d541 CFRunLoopRunSpecific + 522
4 CoreFoundation 0x2e57d323 CFRunLoopRunInMode + 104
5 Foundation 0x2efb8651 +[NSURLConnection(Loader) _resourceLo
adLoop:] + 318
6 Foundation 0x2f02ddc7 __NSThread_main__ + 1060
7 libsystem_pthread.dylib 0x38f5ac5d _pthread_body + 138
8 libsystem_pthread.dylib 0x38f5abcf _pthread_start + 100
9 libsystem_pthread.dylib 0x38f58cd0 thread_start + 6

Thread 6:
0 libsystem_kernel.dylib 0x38ef4440 __select + 20
1 libsystem_pthread.dylib 0x38f5ac5d _pthread_body + 138
2 libsystem_pthread.dylib 0x38f5abcf _pthread_start + 100
3 libsystem_pthread.dylib 0x38f58cd0 thread_start + 6

Thread 7:
0 libsystem_kernel.dylib 0x38ee1a84 mach_msg_trap + 20
1 CoreFoundation 0x2e614561 __CFRunLoopServiceMachPort + 154
2 CoreFoundation 0x2e612c81 __CFRunLoopRun + 790
3 CoreFoundation 0x2e57d541 CFRunLoopRunSpecific + 522
4 CoreFoundation 0x2e5c11ab CFRunLoopRun + 96
5 CoreMotion 0x2ec35399 CLSF_thorntonUpdate_6x6 + 57222
6 libsystem_pthread.dylib 0x38f5ac5d _pthread_body + 138
7 libsystem_pthread.dylib 0x38f5abcf _pthread_start + 100
8 libsystem_pthread.dylib 0x38f58cd0 thread_start + 6

Thread 8:
0 libsystem_kernel.dylib 0x38ee1ad4 semaphore_wait_trap + 8
1 MediaToolbox 0x2fa6ed0f fpa_AsyncMovieControlThread + 1752
2 CoreMedia 0x2eba923f figThreadMain + 192
3 libsystem_pthread.dylib 0x38f5ac5d _pthread_body + 138
4 libsystem_pthread.dylib 0x38f5abcf _pthread_start + 100
5 libsystem_pthread.dylib 0x38f58cd0 thread_start + 6

```

Thread 9:

0	libsystem_kernel.dylib	0x38ee1a84 mach_msg_trap + 20
1	CoreFoundation	0x2e614561 __CFRunLoopServiceMachPort + 154
2	CoreFoundation	0x2e612c81 __CFRunLoopRun + 790
3	CoreFoundation	0x2e57d541 CFRunLoopRunSpecific + 522
4	CoreFoundation	0x2e57d323 CFRunLoopRunInMode + 104
5	Foundation	0x2ef6b827 -[NSRunLoop(NSRunLoop) runMode:beforeDate:] + 252
6	KKBOX	0x0025bbed -[KKBOXAPICallOperation runloop] (KKBOXAPICallOperation.m:57)
7	KKBOX	0x0025bd2b -[KKBOXAPICallOperation main] (KKBOXAPICallOperation.m:74)
8	Foundation	0x2ef78c35 -[__NSOperationInternal _start:] + 77
9		0
9	Foundation	0x2f01caf9 __NSOQSchedule_f + 58
10	libdispatch.dylib	0x38e2d4bf _dispatch_async_redirect_invoke + 108
11	libdispatch.dylib	0x38e2e7e5 _dispatch_root_queue_drain + 222
12	libdispatch.dylib	0x38e2e9d1 _dispatch_worker_thread2 + 54
13	libsystem_pthread.dylib	0x38f58dff _pthread_wqthread + 296
14	libsystem_pthread.dylib	0x38f58cc4 start_wqthread + 6

Thread 10:

0	libsystem_kernel.dylib	0x38ee1a84 mach_msg_trap + 20
1	CoreFoundation	0x2e614561 __CFRunLoopServiceMachPort + 154
2	CoreFoundation	0x2e612c81 __CFRunLoopRun + 790
3	CoreFoundation	0x2e57d541 CFRunLoopRunSpecific + 522
4	CoreFoundation	0x2e57d323 CFRunLoopRunInMode + 104
5	Foundation	0x2ef6b827 -[NSRunLoop(NSRunLoop) runMode:beforeDate:] + 252
6	KKBOX	0x0025bbed -[KKBOXAPICallOperation runloop] (KKBOXAPICallOperation.m:57)
7	KKBOX	0x0025c295 -[KKBOXAPICallOperation main] (KKBOXAPICallOperation.m:145)
8	Foundation	0x2ef78c35 -[__NSOperationInternal _start:] + 77
9		0
9	Foundation	0x2f01caf9 __NSOQSchedule_f + 58
10	libdispatch.dylib	0x38e2d4bf _dispatch_async_redirect_invoke + 108
11	libdispatch.dylib	0x38e2e7e5 _dispatch_root_queue_drain + 222
12	libdispatch.dylib	0x38e2e9d1 _dispatch_worker_thread2 + 54
13	libsystem_pthread.dylib	0x38f58dff _pthread_wqthread + 296
14	libsystem_pthread.dylib	0x38f58cc4 start_wqthread + 6

Thread 11:

0	libsystem_kernel.dylib	0x38ef4c7c __workq_kernreturn + 8
1	libsystem_pthread.dylib	0x38f58cc4 start_wqthread + 6

Thread 12:

0	libsystem_kernel.dylib	0x38ee1a84 mach_msg_trap + 20
1	CoreFoundation	0x2e614561 __CFRunLoopServiceMachPort + 154
2	CoreFoundation	0x2e612c81 __CFRunLoopRun + 790
3	CoreFoundation	0x2e57d541 CFRunLoopRunSpecific + 522
4	CoreFoundation	0x2e57d323 CFRunLoopRunInMode + 104

```

5 Foundation 0x2ef6b827 - [NSRunLoop(NSRunLoop) runMode:before
Date:] + 252
6 KKBOX 0x0025bbcd - [KKBOXAPICallOperation runloop] (KKB
OXAPICallOperation.m:57)
7 KKBOX 0x0025bd2b - [KKBOXAPICallOperation main] (KKBOXA
PICallOperation.m:74)
8 Foundation 0x2ef78c35 -[__NSOperationInternal _start:] + 77
0
9 Foundation 0x2f01caf8 __NSOQSchedule_f + 58
10 libdispatch.dylib 0x38e2d4bf _dispatch_async_redirect_invoke + 108
11 libdispatch.dylib 0x38e2e7e5 _dispatch_root_queue_drain + 222
12 libdispatch.dylib 0x38e2e9d1 _dispatch_worker_thread2 + 54
13 libsystem_pthread.dylib 0x38f58dff _pthread_wqthread + 296
14 libsystem_pthread.dylib 0x38f58cc4 start_wqthread + 6

Thread 13:
0 libsystem_kernel.dylib 0x38ef4c7c __workq_kernreturn + 8
1 libsystem_pthread.dylib 0x38f58cc4 start_wqthread + 6

Thread 14:
0 libsystem_kernel.dylib 0x38ef4c7c __workq_kernreturn + 8
1 libsystem_pthread.dylib 0x38f58cc4 start_wqthread + 6

Thread 15:
0 libsystem_kernel.dylib 0x38ef4c7c __workq_kernreturn + 8
1 libsystem_pthread.dylib 0x38f58cc4 start_wqthread + 6

Thread 16:
0 libsystem_kernel.dylib 0x38ee1a84 mach_msg_trap + 20
1 CoreFoundation 0x2e614561 __CFRunLoopServiceMachPort + 154
2 CoreFoundation 0x2e612c81 __CFRunLoopRun + 790
3 CoreFoundation 0x2e57d541 CFRunLoopRunSpecific + 522
4 CoreFoundation 0x2e57d323 CFRunLoopRunInMode + 104
5 Foundation 0x2ef6b827 - [NSRunLoop(NSRunLoop) runMode:before
Date:] + 252
6 KKBOX 0x001b12bd - [KKAlbumCoverImageFetchOperation mai
n] (KKAlbumCoverManager.m:464)
7 Foundation 0x2ef78c35 -[__NSOperationInternal _start:] + 77
0
8 Foundation 0x2f01caf8 __NSOQSchedule_f + 58
9 libdispatch.dylib 0x38e2d4bf _dispatch_async_redirect_invoke + 108
10 libdispatch.dylib 0x38e2e7e5 _dispatch_root_queue_drain + 222
11 libdispatch.dylib 0x38e2e9d1 _dispatch_worker_thread2 + 54
12 libsystem_pthread.dylib 0x38f58dff _pthread_wqthread + 296
13 libsystem_pthread.dylib 0x38f58cc4 start_wqthread + 6

Thread 0 crashed with ARM Thread State:
  pc: 0x38ef41fc      r7: 0x27d868e8      sp: 0x27d868e8      r0: 0x00000000
  r1: 0x00000000      r2: 0x00000000      r3: 0xffffffff      r4: 0x00000006
  r5: 0x3ad2118c      r6: 0x1802de30      r8: 0x313cf0da      r9: 0x3ad21e30
  r10: 0x313b4457     r11: 0x00000019     ip: 0x000000148     lr: 0x38f5ba53
  cpsr: 0x00000010

```

Link Register Analysis:

Symbol: pthread_kill + 56

Description: We have determined that the link register (lr) is very likely to contain the return address of frame #0's calling function, and have inserted it into the crashing thread's backtrace as frame #1 to aid in analysis. This determination was made by applying a heuristic to determine whether the crashing function was likely to have created a new stack frame at the time of the crash.

Type: 1

Binary Images:

```

0x7d000 - 0x71cffff +KKBOX armv7 <f8a0ae9e6da833e0a9bd332e3d4b8a4c> /Users/USER/KKBOX
X.app/KKBOX
0x8b1000 - 0x8b1ffff MobileSubstrate.dylib armv6 <ad3e6cb9e915360ebc71ccbf27bc4ea7>
/Library/MobileSubstrate/MobileSubstrate.dylib
0x903000 - 0x905ffff SubstrateLoader.dylib armv6 <974e4b1ab6e6397db859d79f37b7ab37>
/Library/Frameworks/CydiaSubstrate.framework/Libraries/SubstrateLoader.dylib
0x929000 - 0x935ffff Activator.dylib armv7s <24b72822b56d30f4b88c83448f3db656> /Library/MobileSubstrate/DynamicLibraries/Activator.dylib
0x972000 - 0x974ffff Emphasize.dylib armv7 <4a31e195474039db8a69730adc9b7642> /Library/MobileSubstrate/DynamicLibraries/Emphasize.dylib
0x977000 - 0x97cffff CydiaSubstrate armv6 <abeb3e46b03b3abeb9d3feba7fefef2fb> /Library/Frameworks/CydiaSubstrate.framework/CydiaSubstrate
0xa82000 - 0xa89ffff libapplist.dylib armv7s <3ef6ef0d6b7d350982114b6b4221ef01> /usr
/lib/libapplist.dylib
0xa90000 - 0xa95ffff LocalIAPStore.dylib armv7s <f3341a7878d9319cb0bef7c2d2cbd896> /
Library/MobileSubstrate/DynamicLibraries/LocalIAPStore.dylib
0xa9f000 - 0xaa5ffff RePower.dylib armv7s <dce8d29ea843350baad4f700b0443795> /Library/MobileSubstrate/DynamicLibraries/RePower.dylib
0xaa0000 - 0xaaaffff ToneEnabler.dylib armv7s <21fc136886163804b54e0524f7d1ca25> /Library/MobileSubstrate/DynamicLibraries/ToneEnabler.dylib
0xaad000 - 0xaeaffff WeeLoader.dylib armv7 <7ff8f9166f93382eb27aba388d599a0a> /Library/MobileSubstrate/DynamicLibraries/WeeLoader.dylib
0xab1000 - 0xac2ffff WinterBoard.dylib armv6 <2f7ede1815c93754b5c91a06195b485a> /Library/MobileSubstrate/DynamicLibraries/WinterBoard.dylib
0xad6000 - 0xadaffff colorYOurBoardFree.dylib armv7s <2e3552b8fe5c3bb8967e25d652d5a3
3f> /Library/MobileSubstrate/DynamicLibraries/colorYOurBoardFree.dylib
0xadff000 - 0xae0ffff zeppelin_uikit.dylib armv7 <1401eeb739bb3a5d88a45926eac68b46> /
Library/MobileSubstrate/DynamicLibraries/zeppelin_uikit.dylib
0x77a5000 - 0x77b0fff QuickSpeak armv7s <9932eaf14fb23095b04ab610b05ba02e> /System/Li
brary/AccessibilityBundles/QuickSpeak.bundle/QuickSpeak
0x2d173000 - 0x2d176fff AccessibilitySettingsLoader armv7s <2c36f6b91cc63b6892f3aaf7a4d
4255a> /System/Library/AccessibilityBundles/AccessibilitySettingsLoader.bundle/Accessibil
itySettingsLoader
0x2d233000 - 0x2d31cffff RawCamera armv7s <ae3d5bd17362338584b6022dc4427d10> /System/Lib
rary/CoreServices/RawCamera.bundle/RawCamera
0x2d448000 - 0x2d549fff AVFoundation armv7s <3ea3fd2b98d1360292bcb40c93e444e3> /System/
Library/Frameworks/AVFoundation.framework/AVFoundation
0x2d54a000 - 0x2d572fff libAVFAudio.dylib armv7s <6f10a606028b3d2a862b2b1b8bf81eb3> /Sy
stem/Library/Frameworks/AVFoundation.framework/libAVFAudio.dylib
0x2d573000 - 0x2d573fff Accelerate armv7s <9340338f3cdf347abe4a88c2f59b5b12> /System/Li
brary/Frameworks/Accelerate.framework/Accelerate
0x2d57d000 - 0x2d74afff vImage armv7s <479b5c4701833284ab587a1d2fdb5627> /System/Librar
y/Frameworks/Accelerate.framework/Frameworks/vImage.framework/vImage

```

0x2d74b000 - 0x2d82dff libBLAS.dylib armv7s <da4fa367557d3028b02458e2cdf6d84d> /System/Library/Frameworks/Accelerate.framework/Frameworks/vecLib.framework/libBLAS.dylib
0x2d82e000 - 0x2dae9fff libLAPACK.dylib armv7s <066ea8372dd23f6d89011f9a4a872d6f> /System/Library/Frameworks/Accelerate.framework/Frameworks/vecLib.framework/libLAPACK.dylib
0x2daea000 - 0x2db58fff libvDSP.dylib armv7s <a5dcfe68199839b989c7be120c14ccb4> /System/Library/Frameworks/Accelerate.framework/Frameworks/vecLib.framework/libvDSP.dylib
0x2db59000 - 0x2db6bfff libvMisc.dylib armv7s <ea636bbda5ee33119a4e731aed02fa31> /System/Library/Frameworks/Accelerate.framework/Frameworks/vecLib.framework/libvMisc.dylib
0x2db6c000 - 0x2db6cff vecLib armv7s <663aefaf25bc5367baa72ca144ac26d18> /System/Library/Frameworks/Accelerate.framework/Frameworks/vecLib.framework/vecLib
0x2db6d000 - 0x2db8cff Accounts armv7s <23bffaebc2f23390817a66949208b36c> /System/Library/Frameworks/Accounts.framework/Accounts
0x2db8d000 - 0x2db8dff AdSupport armv7s <e27e7aa767c7332f833fc8449af286b9> /System/Library/Frameworks/AdSupport.framework/AdSupport
0x2db8e000 - 0x2dbf3fff AddressBook armv7s <27d4ff7aebd4386e908e1b70b76a5516> /System/Library/Frameworks/AddressBook.framework/AddressBook
0x2dbf4000 - 0x2dd05fff AddressBookUI armv7s <51bf673528f235e5991e2b00b59518b8> /System/Library/Frameworks/AddressBookUI.framework/AddressBookUI
0x2dd06000 - 0x2dd17fff AssetsLibrary armv7s <24f602d680f13ca5b00cdc6a0d157bb1> /System/Library/Frameworks/AssetsLibrary.framework/AssetsLibrary
0x2dd18000 - 0x2de5bfff AudioCodecs armv7s <555c8ebca0c9327396c2c4838cc79455> /System/Library/Frameworks/AudioToolbox.framework/AudioCodecs
0x2de5c000 - 0x2e20afff AudioToolbox armv7s <e3d0e69f3ad632a4b1a29ecbc20ea8d3> /System/Library/Frameworks/AudioToolbox.framework/AudioToolbox
0x2e20b000 - 0x2e310fff CFNetwork armv7s <aa9d0dfc3c8337e5bd095f7b6d4b53e8> /System/Library/Frameworks/CFNetwork.framework/CFNetwork
0x2e311000 - 0x2e36cff CoreAudio armv7s <73de1dcc06233a7c983aaab354f19eec> /System/Library/Frameworks/CoreAudio.framework/CoreAudio
0x2e36d000 - 0x2e383fff CoreBluetooth armv7s <3268403ee3fc33ca935038630018bd12> /System/Library/Frameworks/CoreBluetooth.framework/CoreBluetooth
0x2e384000 - 0x2e574fff CoreData armv7s <f58b2fa876db34d9babcc5e17893ae16> /System/Library/Frameworks/CoreData.framework/CoreData
0x2e575000 - 0x2e6b8fff CoreFoundation armv7s <29f79e2eff633d1acdf695ad41f916c> /System/Library/Frameworks/CoreFoundation.framework/CoreFoundation
0x2e6b9000 - 0x2e7dff CoreGraphics armv7s <b355402aa362364c838b34e9aa91acba> /System/Library/Frameworks/CoreGraphics.framework/CoreGraphics
0x2e7df000 - 0x2e81afff libCGFreetype.A.dylib armv7s <13b1f99f838c3440941004d726d7cf64> /System/Library/Frameworks/CoreGraphics.framework/Resources/libCGFreetype.A.dylib
0x2e81c000 - 0x2e826fff libCMSBuiltIn.A.dylib armv7s <95e72800562936fd9ff7b60a38c498a6> /System/Library/Frameworks/CoreGraphics.framework/Resources/libCMSBuiltIn.A.dylib
0x2ea0b000 - 0x2ea25fff libRIP.A.dylib armv7s <b443405791f93791bc6140c5c50f86bf> /System/Library/Frameworks/CoreGraphics.framework/Resources/libRIP.A.dylib
0x2ea26000 - 0x2eafefff CoreImage armv7s <c23091e38fc7316b8c3b5b94efa2689d> /System/Library/Frameworks/CoreImage.framework/CoreImage
0x2eaff000 - 0x2eb4cff CoreLocation armv7s <87d3830bbe123e45a611eac7e4bab134> /System/Library/Frameworks/CoreLocation.framework/CoreLocation
0x2eb84000 - 0x2ebfbfff CoreMedia armv7s <80585a9c30bb39679cd285697075c668> /System/Library/Frameworks/CoreMedia.framework/CoreMedia
0x2ebfc000 - 0x2eca4fff CoreMotion armv7s <07a70dfe457f31eeafccdc453d5fc09f> /System/Library/Frameworks/CoreMotion.framework/CoreMotion
0x2eca5000 - 0x2ecfdfff CoreTelephony armv7s <d1abcb27328a3d32b50e43f7d8723f88> /System/Library/Frameworks/CoreTelephony.framework/CoreTelephony
0x2ecfe000 - 0x2ed8dff CoreText armv7s <640ae778897530d089098f55db9ffb58> /System/

```

ary/Frameworks/CoreText.framework/CoreText
0x2ed8e000 - 0x2ed9ffff CoreVideo armv7s <09adc070759b39ad9a0cc38513339d1c> /System/Library/Frameworks/CoreVideo.framework/CoreVideo
0x2ed9e000 - 0x2ee5ffff EventKit armv7s <6a59cfcd59cd3c9391a60bdःa9b19fbf> /System/Library/Frameworks/EventKit.framework/EventKit
0x2ee5d000 - 0x2ef4ffff EventKitUI armv7s <86cb5a91468d365495d9107c46b9b946> /System/Library/Frameworks/EventKitUI.framework/EventKitUI
0x2ef60000 - 0x2f14afff Foundation armv7s <3baf454a0faf3f9ea8d08538fb233dfd> /System/Library/Frameworks/Foundation.framework/Foundation
0x2f14b000 - 0x2f175fff GLKit armv7s <a01140a43577352b9846e8ecd9a0f662> /System/Library/Frameworks/GLKit.framework/GLKit
0x2f328000 - 0x2f37efff IOKit armv7s <c504536b28b43297a5b874d0a4d958e8> /System/Library/Frameworks/IOKit.framework/Versions/A/IOKit
0x2f37f000 - 0x2f58ffff ImageIO armv7s <ff5ce78d675c375883a2b78ad7bc726e> /System/Library/Frameworks/ImageIO.framework/ImageIO
0x2f590000 - 0x2f7d8fff JavaScriptCore armv7s <daace5d68df3f299b1755a5f7b8cb6c> /System/Library/Frameworks/JavaScriptCore.framework/JavaScriptCore
0x2f7d9000 - 0x2f87bfff MapKit armv7s <7a13d31e0bd93ea1a42f5ce5435c71d9> /System/Library/Frameworks/MapKit.framework/MapKit
0x2f87c000 - 0x2f880fff MediaAccessibility armv7s <385b8559d9bd3721b466d4d074b4cc26> /System/Library/Frameworks/MediaAccessibility.framework/MediaAccessibility
0x2f881000 - 0x2fa69fff MediaPlayer armv7s <32b2c665039b30fe9a78ee59a9e5143b> /System/Library/Frameworks/MediaPlayer.framework/MediaPlayer
0x2fa6a000 - 0x2fd23fff MediaToolbox armv7s <9309ab10b2d3316d8ee1a47a9138cb3e> /System/Library/Frameworks/MediaToolbox.framework/MediaToolbox
0x2fd24000 - 0x2fdbffff MessageUI armv7s <8c878c2cf67535cd9c88f99d1c6b81e4> /System/Library/Frameworks/MessageUI.framework/MessageUI
0x2fdc0000 - 0x2fe23fff MobileCoreServices armv7s <7641d47fcebb30019f1b2d572b6184d3> /System/Library/Frameworks/MobileCoreServices.framework/MobileCoreServices
0x2fe7f000 - 0x2feb2fff OpenAL armv7s <ad6db8d403d7374abb78a06d40ddd6b1> /System/Library/Frameworks/OpenAL.framework/OpenAL
0x30857000 - 0x3085ffff OpenGL ES armv7s <f66e6ce6393c316784f0a6cf2c3cfa24> /System/Library/Frameworks/OpenGL ES.framework/OpenGL ES
0x30861000 - 0x30861fff libCVMSPluginSupport.dylib armv7s <d5abff68493b3a8b88099c122d49246c> /System/Library/Frameworks/OpenGL ES.framework/libCVMSPluginSupport.dylib
0x30865000 - 0x30868fff libCoreVMClient.dylib armv7s <aa8b008095d2312c8aa226e3d10b027b> /System/Library/Frameworks/OpenGL ES.framework/libCoreVMClient.dylib
0x30869000 - 0x30870fff libGFXShared.dylib armv7s <667e409cfe5b39c58e50240c841cf7ef> /System/Library/Frameworks/OpenGL ES.framework/libGFXShared.dylib
0x30871000 - 0x308b1fff libGLImage.dylib armv7s <73b69adcf3b434548d424857d93e3505> /System/Library/Frameworks/OpenGL ES.framework/libGLImage.dylib
0x30a4a000 - 0x30b8ffff QuartzCore armv7s <38ffb26d65b3385ab45faa5a8fc5d963> /System/Library/Frameworks/QuartzCore.framework/QuartzCore
0x30b90000 - 0x30be6fff QuickLook armv7s <ec4d1971e0253414948bed34e195dec> /System/Library/Frameworks/QuickLook.framework/QuickLook
0x30be9000 - 0x30c29fff Security armv7s <3bdd534aeба03353acf5eef99d47c2e0> /System/Library/Frameworks/Security.framework/Security
0x30c2a000 - 0x30c9efff Social armv7s <42a050c5a8e73bcb8039f5b34b351fa8> /System/Library/Frameworks/Social.framework/Social
0x30d5e000 - 0x30d71fff StoreKit armv7s <52e20e657f903fe3b743943579edd41e> /System/Library/Frameworks/StoreKit.framework/StoreKit
0x30d72000 - 0x30dc1fff SystemConfiguration armv7s <ec3065bf336830cb85e9072fe4480a4f> /System/Library/Frameworks/SystemConfiguration.framework/SystemConfiguration

```

0x30dc4000 - 0x314e3fff UIKit armv7s <5f5397297fa73ab0889777dd49ebd8e1> /System/Library/Frameworks/UIKit.framework/UIKit
0x314e4000 - 0x31532fff VideoToolbox armv7s <dee33b6c37d13f13b81a0160e7efa423> /System/Library/Frameworks/VideoToolbox.framework/VideoToolbox
0x317a5000 - 0x317aefff AOSNotification armv7s <d52fe916a3ea3cb88174198668a70cf4> /System/Library/PrivateFrameworks/AOSNotification.framework/AOSNotification
0x31812000 - 0x3185bfff AccessibilityUtilities armv7s <b61fc80d33663b34a1f4807bc07ed120> /System/Library/PrivateFrameworks/AccessibilityUtilities.framework/AccessibilityUtilities
0x318af000 - 0x318cffff AccountsUI armv7s <1a36a808008733a3b0dcba14017c67a7> /System/Library/PrivateFrameworks/AccountsUI.framework/AccountsUI
0x318d0000 - 0x318d4fff AggregateDictionary armv7s <b5a1d0802c003f24b347242a5293daa6> /System/Library/PrivateFrameworks/AggregateDictionary.framework/AggregateDictionary
0x31ad2000 - 0x31ae6fff AirTraffic armv7s <8c0533b7adca3eface4ea1628a80b19> /System/Library/PrivateFrameworks/AirTraffic.framework/AirTraffic
0x31ae7000 - 0x31e22fff Altitude armv7s <d3231c90d897359a93a945cf19e76563> /System/Library/PrivateFrameworks/Altitude.framework/Altitude
0x31e57000 - 0x31e94fff AppSupport armv7s <eaacd9b4974531409eda0f399b99fcb5> /System/Library/PrivateFrameworks/AppSupport.framework/AppSupport
0x31e95000 - 0x31eccfff AppleAccount armv7s <e3860cad86cb3f4ba87909e890f25e4f> /System/Library/PrivateFrameworks/AppleAccount.framework/AppleAccount
0x31f6c000 - 0x31f7cffff ApplePushService armv7s <c757d1d3414c3274ad34f1e86d1a7bd3> /System/Library/PrivateFrameworks/ApplePushService.framework/ApplePushService
0x31fb4000 - 0x31fc0fff AssetsLibraryServices armv7s <3ff8908f255132c688a2e8ae1e6bc221> /System/Library/PrivateFrameworks/AssetsLibraryServices.framework/AssetsLibraryServices
0x31fc1000 - 0x31fdcfff AssistantServices armv7s <311a52aacd91310293715414a922820b> /System/Library/PrivateFrameworks/AssistantServices.framework/AssistantServices
0x31ffd000 - 0x32000fff BTLEAudioController armv7s <631049fd95ae3dad8ab065b43840404d> /System/Library/PrivateFrameworks/BTLEAudioController.framework/BTLEAudioController
0x32001000 - 0x32024fff BackBoardServices armv7s <0238e0b01eb03e9d92409f7dc0d27c85> /System/Library/PrivateFrameworks/BackBoardServices.framework/BackBoardServices
0x32027000 - 0x3202cffff BluetoothManager armv7s <d1fb9112ad3f340d8051315445eff177> /System/Library/PrivateFrameworks/BluetoothManager.framework/BluetoothManager
0x3202d000 - 0x32051fff Bom armv7s <c7f17a5e9bf43b1291052841d478ae95> /System/Library/PrivateFrameworks/Bom.framework/Bom
0x32064000 - 0x320acfff BulletinBoard armv7s <3011c66c543531a382346a2a12f60a6f> /System/Library/PrivateFrameworks/BulletinBoard.framework/BulletinBoard
0x320f0000 - 0x320f8fff CaptiveNetwork armv7s <0863d2a2c3ce3ababd236d49bc41e26b> /System/Library/PrivateFrameworks/CaptiveNetwork.framework/CaptiveNetwork
0x320f9000 - 0x321d3fff Celestial armv7s <e54ab561b9383e31bf10f3e0611de842> /System/Library/PrivateFrameworks/Celestial.framework/Celestial
0x321d4000 - 0x321dffff CertInfo armv7s <78f93f2defb7306ca8f70cdcdf95cf54> /System/Library/PrivateFrameworks/CertInfo.framework/CertInfo
0x321e0000 - 0x321e5fff CertUI armv7s <3366ec6a08c93a9c98488e9696ebf1d2> /System/Library/PrivateFrameworks/CertUI.framework/CertUI
0x322ae000 - 0x322cefff ChunkingLibrary armv7s <719f2a07f62f3245b8b7d814ddd6c4f3> /System/Library/PrivateFrameworks/ChunkingLibrary.framework/ChunkingLibrary
0x3231f000 - 0x3232afff CommonUtilities armv7s <9c015db83e55300dbd48f4f28db35978> /System/Library/PrivateFrameworks/CommonUtilities.framework/CommonUtilities
0x3232b000 - 0x3232ffff CommunicationsFilter armv7s <9336295ddf07377681a39ed84af77aaa> /System/Library/PrivateFrameworks/CommunicationsFilter.framework/CommunicationsFilter
0x323c7000 - 0x323f7fff ContentIndex armv7s <ea1ac0c33c273e48be7b4f1a8f2ce6e6> /System/Library/PrivateFrameworks/ContentIndex.framework/ContentIndex

0x323f8000 - 0x323ffff CoreAUC armv7s <835f573269fa3329891ee5477c0c48c0> /System/Library/PrivateFrameworks/CoreAUC.framework/CoreAUC
0x32407000 - 0x3245ffff CoreDAV armv7s <15ef8f74676d36a58b25c8e4ae42df7f> /System/Library/PrivateFrameworks/CoreDAV.framework/CoreDAV
0x3249b000 - 0x32599fff CoreMediaStream armv7s <83c99a6c3e0f3023abb025162eb3ba58> /System/Library/PrivateFrameworks/CoreMediaStream.framework/CoreMediaStream
0x32633000 - 0x3263ffff CoreRecents armv7s <0351246c23a63badab2047173e7bf080> /System/Library/PrivateFrameworks/CoreRecents.framework/CoreRecents
0x3268b000 - 0x326a9fff CoreServicesInternal armv7s <81f48b4e902a36f68f38df1514fd02a0> /System/Library/PrivateFrameworks/CoreServicesInternal.framework/CoreServicesInternal
0x326aa000 - 0x326abfff CoreSurface armv7s <b841c412278b31e8ab3fdcc9ef3f4952> /System/Library/PrivateFrameworks/CoreSurface.framework/CoreSurface
0x326ac000 - 0x32712fff CoreSymbolication armv7s <bc8f947cca233d0f846455ab2bd75b29> /System/Library/PrivateFrameworks/CoreSymbolication.framework/CoreSymbolication
0x3274d000 - 0x32751fff CoreTime armv7s <57a1ef07717a37b1acbb219dee6c4305> /System/Library/PrivateFrameworks/CoreTime.framework/CoreTime
0x32752000 - 0x327acfff CoreUI armv7s <b8d30ee2ace23424b1894da1221896c8> /System/Library/PrivateFrameworks/CoreUI.framework/CoreUI
0x327ad000 - 0x327fafff CoreUtils armv7s <a0ed77908628345face830cd19fd0d> /System/Library/PrivateFrameworks/CoreUtils.framework/CoreUtils
0x327fb000 - 0x32800fff CrashReporterSupport armv7s <048637e8b7bc3cd491ac9a13c67f48c6> /System/Library/PrivateFrameworks/CrashReporterSupport.framework/CrashReporterSupport
0x32801000 - 0x32837fff DataAccess armv7s <d45ade7190993faaba5495911502ca77> /System/Library/PrivateFrameworks/DataAccess.framework/DataAccess
0x329c9000 - 0x329defff DataAccessExpress armv7s <065dee6e6811309189e682901dc2188e> /System/Library/PrivateFrameworks/DataAccessExpress.framework/DataAccessExpress
0x32a19000 - 0x32a1cffff DataMigration armv7s <9db6af4e660f36b289b5498202828768> /System/Library/PrivateFrameworks/DataMigration.framework/DataMigration
0x32a21000 - 0x32a22fff DiagnosticLogCollection armv7s <c9dc18474a693be8beb77a32e9bff97b> /System/Library/PrivateFrameworks/DiagnosticLogCollection.framework/DiagnosticLogCollection
0x32a23000 - 0x32a3dfff DictionaryServices armv7s <b11ee1719c90367a8173643f39dcae46> /System/Library/PrivateFrameworks/DictionaryServices.framework/DictionaryServices
0x32a59000 - 0x32a76fff EAP8021X armv7s <b0770c75118e3d8fb9eb8d1397409bdb> /System/Library/PrivateFrameworks/EAP8021X.framework/EAP8021X
0x32a7f000 - 0x32a8afff ExFAT armv7s <8e0b177c08bf34edabb3937b0139c002> /System/Library/PrivateFrameworks/ExFAT.framework/ExFAT
0x32a8b000 - 0x32a9bfff FTAWD armv7s <03a5778e9c2037e3802c2d9406c1d7f4> /System/Library/PrivateFrameworks/FTAWD.framework/FTAWD
0x32a9c000 - 0x32a9efff FTCClientServices armv7s <6e90e2d568893ca6b196286629aeeccb1> /System/Library/PrivateFrameworks/FTClientServices.framework/FTClientServices
0x32a9f000 - 0x32ac8fff FTServices armv7s <aa7e3982cd2e30e3897bfbd0a6ed07802> /System/Library/PrivateFrameworks/FTServices.framework/FTServices
0x32ac9000 - 0x32ee4fff FaceCore armv7s <96a335f8e5ad382283eedb4b7f627c8a> /System/Library/PrivateFrameworks/FaceCore.framework/FaceCore
0x33107000 - 0x33113fff GenerationalStorage armv7s <8ca94191735b38c296ceb0a1b7d6f9ef> /System/Library/PrivateFrameworks/GenerationalStorage.framework/GenerationalStorage
0x33114000 - 0x332acfff GeoServices armv7s <619d7e9213f631ba8198549e0ed89e4e> /System/Library/PrivateFrameworks/GeoServices.framework/GeoServices
0x332ad000 - 0x332bbfff GraphicsServices armv7s <d887316a59d5342ebd0a6a5e8ea4d7cf> /System/Library/PrivateFrameworks/GraphicsServices.framework/GraphicsServices
0x3334a000 - 0x333cffff HomeSharing armv7s <7fac964790c9320099a82d580ae9432b> /System/Library/PrivateFrameworks/HomeSharing.framework/HomeSharing

0x333d0000 - 0x333dcfff IAP armv7s <8a3fc4fb6ce23adf941905a3744ab8de> /System/Library/PrivateFrameworks/IAP.framework/IAP
0x33442000 - 0x33476fff IDS armv7s <fc20f09d1fcb3ec5bde9e7d7b8718271> /System/Library/PrivateFrameworks/IDS.framework/IDS
0x334e2000 - 0x334f3fff IDSFoundation armv7s <3936bc676b103a0d9bab55dbe0f33865> /System/Library/PrivateFrameworks/IDSFoundation.framework/IDSFoundation
0x334f4000 - 0x33558fff IMAVCore armv7s <fac1ed04f69340eab5ba9648d642d6d> /System/Library/PrivateFrameworks/IMAVCore.framework/IMAVCore
0x33559000 - 0x335e5fff IMCore armv7s <238b1756f1b23477b032eb94116c7f9f> /System/Library/PrivateFrameworks/IMCore.framework/IMCore
0x33665000 - 0x336bffff IMFoundation armv7s <48fb553357293073b55db1c37790df53> /System/Library/PrivateFrameworks/IMFoundation.framework/IMFoundation
0x336c9000 - 0x336d0fff IOMobileFramebuffer armv7s <452bfbd283c39cf95b400da3a553a12> /System/Library/PrivateFrameworks/IOMobileFramebuffer.framework/IOMobileFramebuffer
0x336d1000 - 0x336d6fff IOSurface armv7s <d8adf4a312fc34bf915b6f779450d6fa> /System/Library/PrivateFrameworks/IOSurface.framework/IOSurface
0x33723000 - 0x33728fff IncomingCallFilter armv7s <4afe562532003eee95efccc03968d10e> /System/Library/PrivateFrameworks/IncomingCallFilter.framework/IncomingCallFilter
0x33748000 - 0x33754fff Librarian armv7s <2e4847d5f83a3609bbe955ebe4982ac9> /System/Library/PrivateFrameworks/Librarian.framework/Librarian
0x33755000 - 0x3378efff MIME armv7s <9be7262f994d36198059ce25d8881c65> /System/Library/PrivateFrameworks/MIME.framework/MIME
0x3378f000 - 0x337ccfff MMCS armv7s <1298ffc6057d34dc87e65d369bf3c66c> /System/Library/PrivateFrameworks/MMCS.framework/MMCS
0x337d5000 - 0x337e0fff MailServices armv7s <00abdcca62743fd58fe43ea2405ef497> /System/Library/PrivateFrameworks/MailServices.framework/MailServices
0x33814000 - 0x3388cff MNC Configuration armv7s <eaf716901eea3605afffc783242fcd55> /System/Library/PrivateFrameworks/ManagedConfiguration.framework/ManagedConfiguration
0x3388d000 - 0x3388efff Marco armv7s <d2c12a9cb9833ad9ab47edec46b8de2c> /System/Library/PrivateFrameworks/Marco.framework/Marco
0x3388f000 - 0x33907fff MediaControlSender armv7s <4c0017a9ceea3191af04244f25aab767> /System/Library/PrivateFrameworks/MediaControlSender.framework/MediaControlSender
0x3393f000 - 0x33949fff MediaRemote armv7s <2bee47a1f52033379806849605e0cf61> /System/Library/PrivateFrameworks/MediaRemote.framework/MediaRemote
0x3394a000 - 0x33962fff MediaStream armv7s <dcb940462fe03d71970aca1cedf25596> /System/Library/PrivateFrameworks/MediaStream.framework/MediaStream
0x339c6000 - 0x33a98fff Message armv7s <f3abf378cab337e88547ffbd08f9cfaa> /System/Library/PrivateFrameworks/Message.framework/Message
0x33a9d000 - 0x33a9ffff MessageSupport armv7s <1008f2f58ca5317fa56eb49acf4da577> /System/Library/PrivateFrameworks/MessageSupport.framework/MessageSupport
0x33aaa000 - 0x33ab5fff MobileAsset armv7s <89df8176733130b7a22b93f0f6990b26> /System/Library/PrivateFrameworks/MobileAsset.framework/MobileAsset
0x33ad9000 - 0x33ae1fff MobileBluetooth armv7s <d9631876c4fa360193bac1a1369e003> /System/Library/PrivateFrameworks/MobileBluetooth.framework/MobileBluetooth
0x33af4000 - 0x33afbfff MobileIcons armv7s <47b04e3e3d1531729207515ffa1f0a54> /System/Library/PrivateFrameworks/MobileIcons.framework/MobileIcons
0x33afc000 - 0x33afffff MobileInstallation armv7s <5f4deac3041631a9bc310d32e0eba04f> /System/Library/PrivateFrameworks/MobileInstallation.framework/MobileInstallation
0x33b00000 - 0x33b08fff MobileKeyBag armv7s <e86226dd92633562ae880eb619ae2fb2> /System/Library/PrivateFrameworks/MobileKeyBag.framework/MobileKeyBag
0x33b30000 - 0x33b33fff MobileSystemServices armv7s <7ccabe1b59f635bf866ea672139080a> /System/Library/PrivateFrameworks/MobileSystemServices.framework/MobileSystemServices
0x33b52000 - 0x33b5dff MobileWiFi armv7s <3709673f12163bbb8ead912f146094db> /System/Li

```

brary/PrivateFrameworks/MobileWiFi.framework/MobileWiFi
0x33b94000 - 0x33d1bfff MusicLibrary armv7s <4e3605a1df833c55816f272507c8da8c> /System/
Library/PrivateFrameworks/MusicLibrary.framework/MusicLibrary
0x33dd0000 - 0x33dd5fff Netrb armv7s <73176bc012be3a89a8f41005b6fcdbce> /System/Library/
/PrivateFrameworks/Netrb.framework/Netrb
0x33dd6000 - 0x33dbffff NetworkStatistics armv7s <57f5831c2cc43b8281157882df7e6447> /Sy
stem/Library/PrivateFrameworks/NetworkStatistics.framework/NetworkStatistics
0x33ddc000 - 0x33df9fff Notes armv7s <86824a1374103929855cfab813d34f90> /System/Library/
/PrivateFrameworks/Notes.framework/Notes
0x33dfa000 - 0x33dfcffff OAuth armv7s <f8769c1821ad3b1f9bfa004f8d9c7506> /System/Library/
/PrivateFrameworks/OAuth.framework/OAuth
0x34554000 - 0x3458ffff OpenCL armv7s <1e7f9319adb7309c9c50f2a4b860b67c> /System/Librar
y/PrivateFrameworks/OpenCL.framework/OpenCL
0x34b36000 - 0x34b5cffff PersistentConnection armv7s <9dd93dcc3b5133f2b92cb8ce2c854a11>
/System/Library/PrivateFrameworks/PersistentConnection.framework/PersistentConnection
0x34cc0000 - 0x34e35fff PhotoLibraryServices armv7s <2ce8ae3b632e3b3793b81524c50bd402>
/System/Library/PrivateFrameworks/PhotoLibraryServices.framework/PhotoLibraryServices
0x34f73000 - 0x34fa0fff PhysicsKit armv7s <aa8b66ecce7d3d4aba28370333f5b7c5> /System/Li
brary/PrivateFrameworks/PhysicsKit.framework/PhysicsKit
0x34fa1000 - 0x34fa4fff PowerLog armv7s <b39b62177cf538f4be20f518402ba9a9> /System/Libr
ary/PrivateFrameworks/PowerLog.framework/PowerLog
0x35020000 - 0x3508cffff Preferences armv7s <e0be490e64303934a5b8bbcf30691a42> /System/L
ibrary/PrivateFrameworks/Preferences.framework/Preferences
0x3508d000 - 0x350c4fff PrintKit armv7s <826a71b346d23f05a3c6c20019d2cd48> /System/Libr
ary/PrivateFrameworks/PrintKit.framework/PrintKit
0x350c8000 - 0x3514ffff ProofReader armv7s <c4d261a63f0237ec9419a46abf0c89ec> /System/L
ibrary/PrivateFrameworks/ProofReader.framework/ProofReader
0x35150000 - 0x3515afff ProtocolBuffer armv7s <502c8769742f354d97a3273260e8584f> /Syst
em/Library/PrivateFrameworks/ProtocolBuffer.framework/ProtocolBuffer
0x3515b000 - 0x3518ffff PrototypeTools armv7s <107f7cf35c263171b10f4adfb81ae35b> /Syst
em/Library/PrivateFrameworks/PrototypeTools.framework/PrototypeTools
0x3518c000 - 0x35200fff Quagga armv7s <19af82b757173d0db60429300235c4e7> /System/Librar
y/PrivateFrameworks/Quagga.framework/Quagga
0x35201000 - 0x35250fff Radio armv7s <2dcae1a18ff53de0a6114d93237a1933> /System/Library/
/PrivateFrameworks/Radio.framework/Radio
0x352d9000 - 0x35359fff SAObjects armv7s <87143b37d6d837b3917ddf065991efc8> /System/Lib
rary/PrivateFrameworks/SAObjects.framework/SAObjects
0x35366000 - 0x35385fff ScreenReaderCore armv7s <fb14cddcab1f3c1497acd638d41d931b> /Sys
tem/Library/PrivateFrameworks/ScreenReaderCore.framework/ScreenReaderCore
0x35459000 - 0x3547bfff SpringBoardFoundation armv7s <9fdc36b95e983f32b86b98515040ea4a>
/System/Library/PrivateFrameworks/SpringBoardFoundation.framework/SpringBoardFoundation
0x3547c000 - 0x35490fff SpringBoardServices armv7s <8e31d0a20f6e311e814c9441ee066040> /Syst
em/Library/PrivateFrameworks/SpringBoardServices.framework/SpringBoardServices
0x35491000 - 0x354aafff SpringBoardUI armv7s <3115f76db448369ca2ea23d665c621ab> /System/
Library/PrivateFrameworks/SpringBoardUI.framework/SpringBoardUI
0x354ab000 - 0x354c1fff SpringBoardUIServices armv7s <3104518cb2dd36e687e106f767b00768>
/System/Library/PrivateFrameworks/SpringBoardUIServices.framework/SpringBoardUIServices
0x356a0000 - 0x357b7fff StoreServices armv7s <49b5dd271ebb3ad28c3ab0bd7025eab0> /System/
Library/PrivateFrameworks/StoreServices.framework/StoreServices
0x357b8000 - 0x357c7fff StreamingZip armv7s <fb0b01aaeae360094bace6460c91549> /System/
Library/PrivateFrameworks/StreamingZip.framework/StreamingZip
0x35866000 - 0x35868fff TCC armv7s <a5f1bc68799e3089b97fdbc682608bb7> /System/Library/P
rivateFrameworks/TCC.framework/TCC

```

0x35869000 - 0x358b1fff TelephonyUI armv7s <cebe06cf1adb3a5397da55f562f86575> /System/Library/PrivateFrameworks/TelephonyUI.framework/TelephonyUI
0x358b2000 - 0x358d3fff TelephonyUtilities armv7s <89161379ee943028814036c307c89b5a> /System/Library/PrivateFrameworks/TelephonyUtilities.framework/TelephonyUtilities
0x358d4000 - 0x35c45fff KBLAYOUTS_iPhone.dylib armv7s <1aecca20df35304e8b984eeb590a0b6d> /System/Library/PrivateFrameworks/TextInput.framework/KBLAYOUTS_iPhone.dylib
0x35c46000 - 0x35c6afff TextInput armv7s <cdaeeb7ed09033ef91e91e71587be2e1> /System/Library/PrivateFrameworks/TextInput.framework/TextInput
0x35c6b000 - 0x35e3afff TextToSpeech armv7s <053f080a453330fb8217d02a6142509e> /System/Library/PrivateFrameworks/TextToSpeech.framework/TextToSpeech
0x35e3b000 - 0x35e61fff ToneKit armv7s <f5d1b20366d035e2905b50da3e8776ee> /System/Library/PrivateFrameworks/ToneKit.framework/ToneKit
0x35e62000 - 0x35e77fff ToneLibrary armv7s <314e3d36e4493fbaa3026b7f92f52be2> /System/Library/PrivateFrameworks/ToneLibrary.framework/ToneLibrary
0x35ec5000 - 0x35f85fff UIFoundation armv7s <00ae39c404023c32b393035110a9a4f9> /System/Library/PrivateFrameworks/UIFoundation.framework/UIFoundation
0x35f86000 - 0x35f9cfff Ubiquity armv7s <bf747b36030e3dadbd25c67289f9b65> /System/Library/PrivateFrameworks/Ubiquity.framework/Ubiquity
0x35f9d000 - 0x35fa0fff UserFS armv7s <77579190a1c23bf7b2d596a9ee667e66> /System/Library/PrivateFrameworks/UserFS.framework/UserFS
0x35fb6000 - 0x36205fff VectorKit armv7s <08567db9172935a3a0296280e4897e93> /System/Library/PrivateFrameworks/VectorKit.framework/VectorKit
0x363a5000 - 0x363c2fff VoiceServices armv7s <45df838dc2d030b08bbfeb205c6b9305> /System/Library/PrivateFrameworks/VoiceServices.framework/VoiceServices
0x363e5000 - 0x3640afff WebBookmarks armv7s <846e49f755b43ab7b101c8a56a2d8f33> /System/Library/PrivateFrameworks/WebBookmarks.framework/WebBookmarks
0x36420000 - 0x36edafff WebCore armv7s <8aa6d34110cd3bde93ccfa8e6b56f4ae> /System/Library/PrivateFrameworks/WebCore.framework/WebCore
0x36edb000 - 0x36f9bfff WebKit armv7s <fbfccca71232a3e8190a32ba05f1f1700> /System/Library/PrivateFrameworks/WebKit.framework/WebKit
0x370d9000 - 0x370dffff XPCKit armv7s <94cec8e834283668b2d78bf51917b6cb> /System/Library/PrivateFrameworks/XPCKit.framework/XPCKit
0x370e0000 - 0x370e8fff XPCObjects armv7s <94c1f17fd60333a39fecf47051e98256> /System/Library/PrivateFrameworks/XPCObjects.framework/XPCObjects
0x3728c000 - 0x372affff iCalendar armv7s <1d51d0a6ba91303db54714a58eee4570> /System/Library/PrivateFrameworks/iCalendar.framework/iCalendar
0x372b4000 - 0x372f5fff iTunesStore armv7s <18ccf7eb84ea3fb1a19a6d48481f78dd> /System/Library/PrivateFrameworks/iTunesStore.framework/iTunesStore
0x37e47000 - 0x37e48fff libAXSafeCategoryBundle.dylib armv7s <b8b6203bf8493d9d8178fddbfc67e124> /usr/lib/libAXSafeCategoryBundle.dylib
0x37e49000 - 0x37e4efff libAXSpeechManager.dylib armv7s <d3ce1f3fc3623d02a94471d73e4ac8f9> /usr/lib/libAXSpeechManager.dylib
0x37e4f000 - 0x37e56fff libAccessibility.dylib armv7s <13499cacfbf13a128c6624edcd609983> /usr/lib/libAccessibility.dylib
0x3804f000 - 0x38065fff libCRFSuite.dylib armv7s <88d76daeba8234e789924c051a426607> /usr/lib/libCRFSuite.dylib
0x38079000 - 0x3807afff libMobileCheckpoint.dylib armv7s <a64358a01d513c61ada96bf6af828820> /usr/lib/libMobileCheckpoint.dylib
0x3807b000 - 0x38090fff libMobileGestalt.dylib armv7s <01cef4af33d43389b8a52005fe468291> /usr/lib/libMobileGestalt.dylib
0x38091000 - 0x38097fff libMobileGestaltExtensions.dylib armv7s <dfc8865f817a3858b4d1a9542eaaf578> /usr/lib/libMobileGestaltExtensions.dylib
0x380ae000 - 0x380affff libSystem.B.dylib armv7s <c612702fe67f319894598600bad61560> /us

```
r/lib/libSystem.B.dylib
0x3811a000 - 0x38146fff libTelephonyUtilDynamic.dylib armv7s <4e2a592c1bcd3527b3b7b960d
621f817> /usr/lib/libTelephonyUtilDynamic.dylib
0x3828f000 - 0x3829ffff libbsm.0.dylib armv7s <34a4b8ea80e4390ab8a146e0de95b6b1> /usr/l
ib/libbsm.0.dylib
0x3829c000 - 0x382a6fff libbz2.1.0.dylib armv7s <b246a3f7a5243be189afe4f7582042fa> /usr/
lib/libbz2.1.0.dylib
0x382a7000 - 0x382f2fff libc++.1.dylib armv7s <18b3a243f7923c39951c97ab416ed3e6> /usr/l
ib/libc++.1.dylib
0x382f3000 - 0x3830dff libc++abi.dylib armv7s <2e20d75c97d339a297a21de19c6a6d4b> /usr/
lib/libc++abi.dylib
0x3831d000 - 0x38324fff libcupolicy.dylib armv7s <4540ecc8ecdb3c95ad6d0b5175d3b120> /us
r/lib/libcupolicy.dylib
0x38350000 - 0x38350fff libgcc_s.1.dylib armv7s <3b247b24b05b31ba824fd703217be8aa> /usr/
lib/libgcc_s.1.dylib
0x3836b000 - 0x38458fff libiconv.2.dylib armv7s <ff50709f8e04318da55e13c9096bba03> /usr/
lib/libiconv.2.dylib
0x38459000 - 0x385aaff libicucore.A.dylib armv7s <4584cde425f43a2686bd362c62ff8d9f> /u
sr/lib/libicucore.A.dylib
0x385b2000 - 0x385b2fff liblangid.dylib armv7s <9babf315c8b739ff98c078fb894ba3c4> /usr/
lib/liblangid.dylib
0x385b3000 - 0x385bdfff libblockdown.dylib armv7s <41f0f2cd691f3b79b7bd6a9131be7b0d> /us
r/lib/libblockdown.dylib
0x385be000 - 0x385d3fff liblzma.5.dylib armv7s <cd6105f66a033d06afb45cf2ca3c1644> /usr/
lib/liblzma.5.dylib
0x388ff000 - 0x38913fff libmis.dylib armv7s <03aad1b678a43337bb007fa32accfa75> /usr/lib
/libmis.dylib
0x3893c000 - 0x38adbfff libobjc.A.dylib armv7s <0cc1bf8b5caa39fd90ca9cf94e03fc> /usr/
lib/libobjc.A.dylib
0x38ba3000 - 0x38bb8fff libresolv.9.dylib armv7s <763ddfffb38af3444b74501dde37a5949> /us
r/lib/libresolv.9.dylib
0x38be1000 - 0x38c78fff libsqlite3.dylib armv7s <0cd7d6e04761365480a2078daee86959> /usr/
lib/libsqlite3.dylib
0x38c79000 - 0x38cc6fff libstdc++.6.dylib armv7s <894bc61807683540a1d475ae8b117140> /us
r/lib/libstdc++.6.dylib
0x38cc7000 - 0x38cedfff libtidy.A.dylib armv7s <9ac4925f9e803e48a846ae28aba6d355> /usr/
lib/libtidy.A.dylib
0x38cf1000 - 0x38da4fff libxml2.2.dylib armv7s <810acee8bebe317492118d752643bde3> /usr/
lib/libxml2.2.dylib
0x38da5000 - 0x38dc6fff libxslt.1.dylib armv7s <e3269cf2460835588f0f9b8f5bed13b2> /usr/
lib/libxslt.1.dylib
0x38dc7000 - 0x38dd3fff libz.1.dylib armv7s <d14399220e74365cbf13a57859a31782> /usr/lib
/libz.1.dylib
0x38dd4000 - 0x38dd8fff libcache.dylib armv7s <371dad0c805634ac9ad03150a7bb227d> /usr/l
ib/system/libcache.dylib
0x38dd9000 - 0x38de1fff libcommonCrypto.dylib armv7s <d4150f408ea232dc87cd4501bb5214d5>
/usr/lib/system/libcommonCrypto.dylib
0x38de2000 - 0x38de6fff libcompiler_rt.dylib armv7s <880e0424cc4a399db4b46bda30abe7a8>
/usr/lib/system/libcompiler_rt.dylib
0x38de7000 - 0x38dedfff libcopyfile.dylib armv7s <0f3a2d037bfc36b083aeffe1eac9d297> /us
r/lib/system/libcopyfile.dylib
0x38dee000 - 0x38e27fff libcorecrypto.dylib armv7s <8af5878efc1a3eeb8e3ca9ec454855a8> /u
sr/lib/system/libcorecrypto.dylib
```

```
0x38e28000 - 0x38e3bfff libdispatch.dylib armv7s <4967565fe4ab3dfea12e7fdd5f858359> /usr/lib/system/libdispatch.dylib
0x38e3c000 - 0x38e3dfff libdyld.dylib armv7s <4df4c7a401b338fe82a2ccf79ccffc3b> /usr/lib/system/libdyld.dylib
0x38e3e000 - 0x38e3efff libkeymgr.dylib armv7s <2c710df49c6034d780bfef6565fbec53> /usr/lib/system/libkeymgr.dylib
0x38e3f000 - 0x38e45fff liblaunch.dylib armv7s <4ccb558bb18c369e8114546f8209bf4c> /usr/lib/system/liblaunch.dylib
0x38e46000 - 0x38e49fff libmacho.dylib armv7s <91172db864f93dcc8948bd943021fc41> /usr/lib/system/libmacho.dylib
0x38e4a000 - 0x38e4bfff libremovefile.dylib armv7s <f30347755ba53479bc6f7b5d0e7e2903> /usr/lib/system/libremovefile.dylib
0x38e4c000 - 0x38e59fff libsystem_asl.dylib armv7s <f9ac0494e76831868b220e2725ecad47> /usr/lib/system/libsystem_asl.dylib
0x38e5a000 - 0x38e5afff libsystem_blocks.dylib armv7s <cd492af4dae33e08cc18b6d4bfebe1a> /usr/lib/system/libsystem_blocks.dylib
0x38e5b000 - 0x38ebffff libsystem_c.dylib armv7s <1a06c713ca5a3e63ba8dc34cf453940c> /usr/lib/system/libsystem_c.dylib
0x38ebe000 - 0x38ec0fff libsystem_configuration.dylib armv7s <1a1bc9be539831269eb572fa4f29bb4d> /usr/lib/system/libsystem_configuration.dylib
0x38ec1000 - 0x38ec7fff libsystem_dnssd.dylib armv7s <7cb95a7379c4382da1332aaaae312177> /usr/lib/system/libsystem_dnssd.dylib
0x38ec8000 - 0x38ee0fff libsystem_info.dylib armv7s <4d68069bec1d3c6481663e452a0f126b> /usr/lib/system/libsystem_info.dylib
0x38ee1000 - 0x38ef9fff libsystem_kernel.dylib armv7s <71ddc20d2095384b84d5f42d20cc2474> /usr/lib/system/libsystem_kernel.dylib
0x38efa000 - 0x38f18fff libsystem_m.dylib armv7s <f1cc6d9397ad3e5eb351f4074e23dbe6> /usr/lib/system/libsystem_m.dylib
0x38f19000 - 0x38f2afff libsystem_malloc.dylib armv7s <f9f95a0430403e0b8a5fc0025d59798b> /usr/lib/system/libsystem_malloc.dylib
0x38f2b000 - 0x38f4afff libsystem_network.dylib armv7s <e590650d0ec332d49eeb4b2f8d7c5b39> /usr/lib/system/libsystem_network.dylib
0x38f4b000 - 0x38f52fff libsystem_notify.dylib armv7s <0e409b117a5535d9b293f931e2e88d8d> /usr/lib/system/libsystem_notify.dylib
0x38f53000 - 0x38f57fff libsystem_platform.dylib armv7s <2a845c549ab93dfc92b008de8cb1a5a8> /usr/lib/system/libsystem_platform.dylib
0x38f58000 - 0x38f5dfff libsystem_pthread.dylib armv7s <9d185d6eca7b373a95bd2c90294a8fc1> /usr/lib/system/libsystem_pthread.dylib
0x38f5e000 - 0x38f5ffff libsystem_sandbox.dylib armv7s <2cda4e839a863765a8efcf1831b5b7d3> /usr/lib/system/libsystem_sandbox.dylib
0x38f60000 - 0x38f62fff libsystem_stats.dylib armv7s <5650116a20cc32b693f1d314566a8fc8> /usr/lib/system/libsystem_stats.dylib
0x38f63000 - 0x38f63fff libunwind.dylib armv7s <6fdd98b80180359199c8f01ae5272f2a> /usr/lib/system/libunwind.dylib
0x38f64000 - 0x38f7efff libxpc.dylib armv7s <9590e2d66ee03ba6869abfda05ae5dc5> /usr/lib/system/libxpc.dylib
```

實戰：JB 造成的 Exception

這是 KKBOX 的 crash report，裡頭說，在 main thread 發生了一個 exception，但是在 main thread 裡頭完全沒有呼叫到 KKBOX 的 code，但是出現了 ActionMenu.dylib 與 Dial.dylib。這些都是 JB 之後才會出現的 library，所以一眼就可以看出；用戶 JB 了他的 iPhone。

Exception 發生在 ActionMenu.dylib 的 0x00327900，我們不清楚這行程式做了什麼而造成 KKBOX crash，不過，就算知道了也沒什麼幫助，因為這個問題整個在我們的守備範圍之外，我們能做的就是請客服通知用戶解除安裝 ActionMenu，或是像 ActionMenu 的作者回報，不會是由我們出面修正 ActionMenu 的問題。

大概在 iOS 5 到 iOS 6 左右的時間，因為 JB 造成的問題也形成我們在客服上不小的負擔，有大量的客訴其實都是用戶自己破壞了軟體的穩定。最近一兩年有比較好一些。

Incident Identifier: A7C0F6A1-66AF-40AF-B3EB-F9D211BFEBB6
CrashReporter Key: 21569c479b7db7a7ce22d28eec0d33231b7a3f24
Hardware Model: iPod2,1
Process: KKBOX [8520]
Path: /var/mobile/Applications/A8CEAE40-267A-4566-B6CA-FA80216A5E5F/KKBOX.app/
KKBOX
Identifier: KKBOX
Version: ??? (???)
Code Type: ARM (Native)
Parent Process: punchd [1]

Date/Time: 2012-04-02 01:00:17.661 +0800
OS Version: iPhone OS 4.2.1 (8C148)
Report Version: 104

Exception Type: EXC_CRASH (SIGABRT)
Exception Codes: 0x00000000, 0x00000000
Crashed Thread: 0

Thread 0 Crashed:

0	libSystem.B.dylib	0x32d25ad0 0x32c9c000 + 563920
1	libSystem.B.dylib	0x32d25abe 0x32c9c000 + 563902
2	libSystem.B.dylib	0x32d25ab2 0x32c9c000 + 563890
3	libSystem.B.dylib	0x32d3cd5e 0x32c9c000 + 658782
4	libstdc++.6.dylib	0x3582fa00 0x357ca000 + 416256
5	libobjc.A.dylib	0x301888d8 0x30180000 + 35032
6	libstdc++.6.dylib	0x3582d100 0x357ca000 + 405760
7	libstdc++.6.dylib	0x3582d178 0x357ca000 + 405880
8	libstdc++.6.dylib	0x3582d2a0 0x357ca000 + 406176
9	libobjc.A.dylib	0x30186f28 0x30180000 + 28456
10	CoreFoundation	0x35f08abc 0x35e64000 + 674492
11	CoreFoundation	0x35f08af0 0x35e64000 + 674544
12	Foundation	0x33fc600 0x33fb3000 + 99840
13	ActionMenu.dylib	0x00327900 0x319000 + 59648
14	Dial.dylib	0x03417cb8 0x3417000 + 3256
15	ActionMenu.dylib	0x00327418 0x319000 + 58392
16	UIKit	0x339410d4 0x3381b000 + 1204436
17	ActionMenu.dylib	0x00320aec 0x319000 + 31468
18	UIKit	0x33940504 0x3381b000 + 1201412
19	ActionMenu.dylib	0x00320394 0x319000 + 29588
20	UIKit	0x339401a8 0x3381b000 + 1200552
21	UIKit	0x3393e584 0x3381b000 + 1193348
22	Foundation	0x33fd44d4 0x33fb3000 + 136404
23	CoreFoundation	0x35ebd2fe 0x35e64000 + 365310
24	CoreFoundation	0x35ebcccd2 0x35e64000 + 363730
25	CoreFoundation	0x35e8ca8a 0x35e64000 + 166538
26	CoreFoundation	0x35e8c504 0x35e64000 + 165124
27	CoreFoundation	0x35e8c412 0x35e64000 + 164882
28	GraphicsServices	0x35261d1c 0x3525d000 + 19740
29	UIKit	0x33865574 0x3381b000 + 304500
30	UIKit	0x33862550 0x3381b000 + 292176
31	KKBOX	0x00050c24 0x1000 + 326692

32 KKBOX	0x000003e60 0x1000 + 11872		
Thread 1:			
0 libSystem.B.dylib	0x32cd1974 0x32c9c000 + 219508		
1 libSystem.B.dylib	0x32da02fc 0x32c9c000 + 1065724		
2 libSystem.B.dylib	0x32d9fd68 0x32c9c000 + 1064296		
3 libSystem.B.dylib	0x32d9f788 0x32c9c000 + 1062792		
4 libSystem.B.dylib	0x32d28970 0x32c9c000 + 575856		
5 libSystem.B.dylib	0x32d1f2fc 0x32c9c000 + 537340		
Thread 2:			
0 libSystem.B.dylib	0x32c9d3b0 0x32c9c000 + 5040		
1 libSystem.B.dylib	0x32c9f894 0x32c9c000 + 14484		
2 CoreFoundation	0x35e8cf7c 0x35e64000 + 167804		
3 CoreFoundation	0x35e8c780 0x35e64000 + 165760		
4 CoreFoundation	0x35e8c504 0x35e64000 + 165124		
5 CoreFoundation	0x35e8c412 0x35e64000 + 164882		
6 WebCore	0x369e7d14 0x368cc000 + 1162516		
7 libSystem.B.dylib	0x32d27b44 0x32c9c000 + 572228		
8 libSystem.B.dylib	0x32d197a4 0x32c9c000 + 513956		
Thread 3:			
0 libSystem.B.dylib	0x32d2654c 0x32c9c000 + 566604		
1 libSystem.B.dylib	0x32cd2f70 0x32c9c000 + 225136		
2 libSystem.B.dylib	0x32cd2910 0x32c9c000 + 223504		
3 CoreMedia	0x373e1e50 0x373df000 + 11856		
4 CoreMedia	0x373e1d2c 0x373df000 + 11564		
5 MediaToolbox	0x341c092c 0x341bc000 + 18732		
6 CoreMedia	0x3740c200 0x373df000 + 184832		
7 libSystem.B.dylib	0x32d27b44 0x32c9c000 + 572228		
8 libSystem.B.dylib	0x32d197a4 0x32c9c000 + 513956		
Thread 4:			
0 libSystem.B.dylib	0x32d292fc 0x32c9c000 + 578300		
1 libSystem.B.dylib	0x32d28b50 0x32c9c000 + 576336		
2 libSystem.B.dylib	0x32d1f2fc 0x32c9c000 + 537340		
Thread 0 crashed with ARM Thread State:			
r0: 0x00000000	r1: 0x00000000	r2: 0x00000001	r3: 0x3e1b1308
r4: 0x00000006	r5: 0x090d69cc	r6: 0x00000000	r7: 0x2fdfc1b4
r8: 0x00000015	r9: 0x0000000a	r10: 0x00000000	r11: 0x090fc060
ip: 0x00000025	sp: 0x2fdfc1b4	lr: 0x32d25ac5	pc: 0x32d25ad0
cpsr: 0x00000010			
Binary Images:			
0x1000 - 0x173fff +KKBOX armv6 <2a292001fcb43656a80e31f678eba2c5> /var/mobile/Applications/A8CEAE40-267A-4566-B6CA-FA80216A5E5F/KKBOX.app/KKBOX			
0x1c8000 - 0x1c8fff +MobileSubstrate.dylib armv6 <cf2cce379dc3a4c970e3196b908b0b6> /Library/MobileSubstrate/MobileSubstrate.dylib			
0x1f8000 - 0x1f9fff +SubstrateLoader.dylib armv6 <30381ec9e24c3c289f447bf428bda2c1> /Library/Frameworks/CydiaSubstrate.framework/Libraries/SubstrateLoader.dylib			
0x300000 - 0x301fff dns.so armv6 <88b569311cca4a9593b2d670051860d1> /usr/lib/info/dns.so			

```

0x319000 - 0x32afff +ActionMenu.dylib armv6 <f7f083b7369a3464a6380e0c8a67c583> /Library/MobileSubstrate/DynamicLibraries/ActionMenu.dylib
0x331000 - 0x33afff +Activator.dylib armv6 <8a7d931c6871371e8c011cb2b9d60a4b> /Library/MobileSubstrate/DynamicLibraries/Activator.dylib
0x342000 - 0x347fff +Backgrounder.dylib armv6 <5ec9abbee88c3abba52fdb4cb35c72cc> /Library/MobileSubstrate/DynamicLibraries/Backgrounder.dylib
0x34b000 - 0x34efff libsubstrate.dylib armv6 <d375337d03a7324c9cfb608b7231eeea> /usr/lib/libsubstrate.dylib
0x352000 - 0x352fff +BlackKeyboard.dylib armv6 <0f117831dd8e3f5baf12bf56492bb161> /Library/MobileSubstrate/DynamicLibraries/BlackKeyboard.dylib
0x356000 - 0x356fff +FakeClockUp.dylib arm <95a29c8436af902b0824c00155af7443> /Library/MobileSubstrate/DynamicLibraries/FakeClockUp.dylib
0x359000 - 0x364fff +WinterBoard.dylib arm /Library/MobileSubstrate/DynamicLibraries/WinterBoard.dylib
0x374000 - 0x37afff +libstatusbar.dylib armv6 <fe21097b2cb83481a70c9ab877af55e3> /Library/MobileSubstrate/DynamicLibraries/libstatusbar.dylib
0x3417000 - 0x3417fff +Dial.dylib armv6 <24d4fc2c56283ccf9a03a6c8ef995a03> /Library/ActionMenu/Plugins/Dial.dylib
0x361e000 - 0x361ffff +Copy All.dylib armv6 <0e55328007e13474af9ce7a761cf2631> /Library/ActionMenu/Plugins/Copy All.dylib
0x36d0000 - 0x36d5fff +Favorites armv6 <c2d87154f24830bb8cccd4ff05b8b817d> /Library/ActionMenu/Plugins/Favorites.bundle/Favorites
0x36d9000 - 0x36defff +History.dylib armv6 <9d7d5f6e522c3d11b67067eee312c34b> /Library/ActionMenu/Plugins/History.dylib
0x36e2000 - 0x36e3fff +Locate.dylib armv6 <89044d8918c93307af651835610be567> /Library/ActionMenu/Plugins/Locate.dylib
0x36e6000 - 0x36e8fff +Lookup.dylib armv6 <4e061bc9ebc23757afa1a7f5b8de2b07> /Library/ActionMenu/Plugins/Lookup.dylib
0x36eb000 - 0x36ecfff +Playing.dylib armv6 <9a6a028244273d41b25a48d367ea1d8e> /Library/ActionMenu/Plugins/Playing.dylib
0x36ef000 - 0x36f0fff +Scroll.dylib armv6 <4aaef1532bb91375891df13731a6e19a0> /Library/ActionMenu/Plugins/Scroll.dylib
0x36f3000 - 0x36f4fff +Send to Pastie.dylib armv6 <fb34a5ea210e339091a1f8914c724f84> /Library/ActionMenu/Plugins/Send to Pastie.dylib
0x36f7000 - 0x36f8fff +Today armv6 <0d152fe3c5e63c758e9df70e246f2307> /Library/ActionMenu/Plugins/Today.bundle/Today
0x36fb000 - 0x36fdfff +Tweet armv6 <1c7f26b28ca33c4987805f8be66b8dd6> /Library/ActionMenu/Plugins/Tweet.bundle/Tweet
0x2fe00000 - 0x2fe29fff dyld armv6 <617f6daf4103547c47a8407a2e0b90de> /usr/lib/dyld
0x30005000 - 0x30129fff MusicLibrary armv6 <dfa6591b4161a4c9dd2fc1e755a8e05b> /System/Library/PrivateFrameworks/MusicLibrary.framework/MusicLibrary
0x30180000 - 0x30247fff libobjc.A.dylib armv6 <429841269f8bcecd4ba3264a8725dad6> /usr/lib/libobjc.A.dylib
0x30248000 - 0x30356fff CFNetwork armv6 <d6eeee83216ee9c553134f069f37cbc2> /System/Library/Frameworks/CFNetwork.framework/CFNetwork
0x30359000 - 0x3037afff libRIP.A.dylib armv6 <22c6da37f3adf325f99c3a0494e04c02> /System/Library/Frameworks/CoreGraphics.framework/Resources/libRIP.A.dylib
0x3037b000 - 0x3037efff libgcc_s.1.dylib armv6 <bed95ed187350ce27d22ed241ef892ea> /usr/lib/libgcc_s.1.dylib
0x303e6000 - 0x3059cff AudioToolbox armv6 <bb65e8ed531fe5923eb8ac00a7c0d87d> /System/Library/Frameworks/AudioToolbox.framework/AudioToolbox
0x305a2000 - 0x306ffff libGLProgrammability.dylib armv6 <aec6b54ffd532bb607aab4acb679b6> /System/Library/Frameworks/OpenGLES.framework/libGLProgrammability.dylib

```

0x3076a000 - 0x30770fff MBX2D armv6 <fad4955cab36e0179df6f8f27d365b8f> /System/Library/PrivateFrameworks/MBX2D.framework/MBX2D
0x30859000 - 0x30866fff CoreVideo armv6 <7b100fd5fdf98db1cd0f0649e7f6f316> /System/Library/Frameworks/CoreVideo.framework/CoreVideo
0x3089e000 - 0x308a3fff CaptiveNetwork armv6 <f41df4b358b77b29ff85e0ea88ee1d> /System/Library/PrivateFrameworks/CaptiveNetwork.framework/CaptiveNetwork
0x308a4000 - 0x30965fff AddressBookUI armv6 <fea72732451610277e22a667d35ad76d> /System/Library/Frameworks/AddressBookUI.framework/AddressBookUI
0x309d7000 - 0x309d8fff DataMigration armv6 <d2de7c0db77278484236669c2cdccabb> /System/Library/PrivateFrameworks/DataMigration.framework/DataMigration
0x30adb000 - 0x30b1efff CoreTelephony armv6 <cabbce0fa7630065dc7e7d3ca3bc616c> /System/Library/Frameworks/CoreTelephony.framework/CoreTelephony
0x30b27000 - 0x32c64fff TextInput armv6 <3fa14e6e5749e0230becd6ea34a8da7a> /System/Library/PrivateFrameworks/TextInput.framework/TextInput
0x32c9c000 - 0x32ddafff libSystem.B.dylib armv6 <70571c1e697e2ae7f7a9b1a499453bb6> /usr/lib/libSystem.B.dylib
0x32ddb000 - 0x32ea5fff Celestial armv6 <11172a6ee53bdf067548cd4496bc5fe0> /System/Library/PrivateFrameworks/Celestial.framework/Celestial
0x32ee6000 - 0x32f42fff libGLImage.dylib armv6 <7c1049f20c4e64591c09d3ac00c7d3ab> /System/Library/Frameworks/OpenGLES.framework/libGLImage.dylib
0x33035000 - 0x33035fff vecLib armv6 <8f914b3e8a581d49fb21d2c0ff75be03> /System/Library/Frameworks/Accelerate.framework/Frameworks/vecLib.framework/vecLib
0x3319f000 - 0x331a7fff ProtocolBuffer armv6 <8ed6e560e85eecfaf625219a8247aea4> /System/Library/PrivateFrameworks/ProtocolBuffer.framework/ProtocolBuffer
0x331fc000 - 0x331fffff MobileInstallation armv6 <456ed7fe6dd9fc8e78df425085b1452> /System/Library/PrivateFrameworks/MobileInstallation.framework/MobileInstallation
0x33201000 - 0x3320cffa libbz2.1.0.dylib armv6 <6aa8a4ed0906a495d059ace9125f525d> /usr/lib/libbz2.1.0.dylib
0x33268000 - 0x3326dff IOSurface armv6 <ffd66ca04dfe7d382d6961f0df3839ff> /System/Library/PrivateFrameworks/IOSurface.framework/IOSurface
0x3326e000 - 0x33273fff libGFXShared.dylib armv6 <bd1c480607cc286288db1ca1aec64180> /System/Library/Frameworks/OpenGLES.framework/libGFXShared.dylib
0x33274000 - 0x33276fff IOMobileFramebuffer armv6 <f42bbbf67195a7b98d67ad021bba4784> /System/Library/PrivateFrameworks/IOMobileFramebuffer.framework/IOMobileFramebuffer
0x3328b000 - 0x3328cff CoreSurface armv6 <5e290514380c626e9b0f9f9985b9dc7a> /System/Library/PrivateFrameworks/CoreSurface.framework/CoreSurface
0x332f8000 - 0x332fbfff MediaRemote armv6 <6f41c7f17d2bfff36ad289acaace4e68d> /System/Library/PrivateFrameworks/MediaRemote.framework/MediaRemote
0x3330c000 - 0x33317fff MobileWiFi armv6 <c7532e63e083a1dd2a0ef7352b85749d> /System/Library/PrivateFrameworks/MobileWiFi.framework/MobileWiFi
0x3337a000 - 0x3337efff ApplePushService armv6 <0560b630d26e261e205fc58942e1885c> /System/Library/PrivateFrameworks/ApplePushService.framework/ApplePushService
0x3337f000 - 0x3337ffff Accelerate armv6 <cdde24a7ad004b2b2e600cd4f3ac5eb7> /System/Library/Frameworks/Accelerate.framework/Accelerate
0x33417000 - 0x33476fff CoreAudio armv6 <ccc4bace0d6eca79a32ed84d566f72e9> /System/Library/Frameworks/CoreAudio.framework/CoreAudio
0x33477000 - 0x3348dff PersistentConnection armv6 <006723906b8ac250c1681a1821fbe94d> /System/Library/PrivateFrameworks/PersistentConnection.framework/PersistentConnection
0x334a1000 - 0x334e5fff AddressBook armv6 <1f30c3370dad27331a491ba4b190813c> /System/Library/Frameworks/AddressBook.framework/AddressBook
0x334e6000 - 0x33505fff EAP8021X armv6 <fa56845b5396c3ebb368c2368331643c> /System/Library/PrivateFrameworks/EAP8021X.framework/EAP8021X
0x33506000 - 0x33509fff ArtworkCache armv6 <9cf3f86a9d03fd428d82e2417f26ec59> /System/L

```
library/PrivateFrameworks/ArtworkCache.framework/ArtworkCache
0x33528000 - 0x33549fff libKoreanConverter.dylib armv6 <1d0e9f0057bf874a030ea24e0c5f322
b> /System/Library/CoreServices/Encodings/libKoreanConverter.dylib
0x3358f000 - 0x335fffff libsqlite3.dylib armv6 <87b9bb47687902d9120d03d1da9eb9fc> /usr/
lib/libsqlite3.dylib
0x33600000 - 0x33605fff AssetsLibraryServices armv6 <224b3cf992a01814f91481244e3213eb>
/System/Library/PrivateFrameworks/AssetsLibraryServices.framework/AssetsLibraryServices
0x33620000 - 0x33627fff libMobileGestalt.dylib armv6 <de9b417e7278742e90b30b1ad45f31fa>
/usr/lib/libMobileGestalt.dylib
0x3362d000 - 0x33671fff VideoToolbox armv6 <101dbbcd34cc3231a8be3fd6392556aa> /System/L
ibrary/PrivateFrameworks/VideoToolbox.framework/VideoToolbox
0x33672000 - 0x337bafff libmecabra.dylib armv6 <fc962eeb4e6cfe4ad5ebee6fb4b1d5c1> /usr/
lib/libmecabra.dylib
0x337c7000 - 0x3381afff IOKit armv6 <20da5e822f21a8d0a7c5b3e149330efd> /System/Library/
Frameworks/IOKit.framework/Versions/A/IOKit
0x3381b000 - 0x33c97fff UIKit armv6 <14ec6c926b8bda71b73136f6e1a6ac1b> /System/Library/
Frameworks/UIKit.framework/UIKit
0x33ced000 - 0x33d0bfff libresolv.9.dylib armv6 <9c94634beea733e754dc115737b6e63c> /usr
/lib/libresolv.9.dylib
0x33e15000 - 0x33f9cff CoreGraphics armv6 <9a1d72fa9549d83abc1e735ba37a4dc2> /System/L
ibrary/Frameworks/CoreGraphics.framework/CoreGraphics
0x33fb3000 - 0x340d4fff Foundation armv6 <6bdeb19a1fc93e2930dad50416f881> /System/Lib
rary/Frameworks/Foundation.framework/Foundation
0x340d5000 - 0x340e8fff libmis.dylib armv6 <dba9c086b49bd9540930ff27211570d6> /usr/lib/
libmis.dylib
0x340ea000 - 0x340edfff libAccessibility.dylib armv6 <74e0f77cc276a9412be268c795fdcbca>
/usr/lib/libAccessibility.dylib
0x341b7000 - 0x341bbfff MobileIcons armv6 <c649339d26250cc96d36bdc36e2909f8> /System/Li
brary/PrivateFrameworks/MobileIcons.framework/MobileIcons
0x341bc000 - 0x34389fff MediaToolbox armv6 <21ceabd0e5de17ad4e883c85fc34d51> /System/L
ibrary/PrivateFrameworks/MediaToolbox.framework/MediaToolbox
0x343fb000 - 0x3449afff ProofReader armv6 <2734920b62f174c17aeeb15f371615ef> /System/Li
brary/PrivateFrameworks/ProofReader.framework/ProofReader
0x3449b000 - 0x344a7fff libkxld.dylib armv6 <f74f359de7bbe3ccdc37fa6f332aebf4> /usr/lib
/system/libkxld.dylib
0x344a8000 - 0x344f1fff CoreLocation armv6 <a69399375024b2bfae8bb96e845f4fd0> /System/L
ibrary/Frameworks/CoreLocation.framework/CoreLocation
0x346ac000 - 0x346c9fff TextInput_zh armv6 <fade6ed6d39f4ee9f3c8d5d7eba1ac39> /System/L
ibrary/TextInput/TextInput_zh.bundle/TextInput_zh
0x346d1000 - 0x346dbfff AccountSettings armv6 <eca67ab04f724e1fa7c6406c88e75433> /Syst
em/Library/PrivateFrameworks/AccountSettings.framework/AccountSettings
0x346dc000 - 0x347c8fff QuartzCore armv6 <77cd91ff21fe6c58c309f2c82eb95ca5> /System/Lib
rary/Frameworks/QuartzCore.framework/QuartzCore
0x347c9000 - 0x347f3fff CoreHandwriting armv6 <39933fdecfc8a1588d3eaf9397dc6410> /Syst
em/Library/PrivateFrameworks/CoreHandwriting.framework/CoreHandwriting
0x347f5000 - 0x34801fff SpringBoardServices armv6 <fd0c472436b3306f5b56118c93c8a423> /S
ystem/Library/PrivateFrameworks/SpringBoardServices.framework/SpringBoardServices
0x34879000 - 0x34884fff ITSsync armv6 <a451205e89373cbdf0832688085e8f72> /System/Library
/PrivateFrameworks/ITSync.framework/ITSync
0x34885000 - 0x348c7fff ManagedConfiguration armv6 <397723a33c19c3487d304d69580acbfc> /
System/Library/PrivateFrameworks/ManagedConfiguration.framework/ManagedConfiguration
0x348c8000 - 0x348cbfff CrashReporterSupport armv6 <00bc60f690e6328b64e7a7b718edf45a> /S
ystem/Library/PrivateFrameworks/CrashReporterSupport.framework/CrashReporterSupport
```

0x348cc000 - 0x349adfff AudioCodecs armv6 <73477d4964476ead5ba488df5c2a518b> /System/Library/Frameworks/AudioToolbox.framework/AudioCodecs
0x34b0f000 - 0x34edcff LibLAPACK.dylib armv6 <0eb734c91165416224b98c943ff6476b> /System/Library/Frameworks/Accelerate.framework/Frameworks/vecLib.framework/libLAPACK.dylib
0x34ef3000 - 0x34fcffff WebKit armv6 <83da207070be989ba81dba3a83d5206a> /System/Library/PrivateFrameworks/WebKit.framework/WebKit
0x34ff2000 - 0x35001fff OpenGL ES armv6 <37eda5ddcff210dd321157da35a87a5e> /System/Library/Frameworks/OpenGLES.framework/OpenGLES
0x35074000 - 0x3518ffff libicucore.A.dylib armv6 <8968ff3f62d7780bb1bd75026a7628d0> /usr/lib/libicucore.A.dylib
0x3525d000 - 0x3526cff GraphicsServices armv6 <af20aba0ec96e7b7c42bb55ac763c784> /System/Library/PrivateFrameworks/GraphicsServices.framework/GraphicsServices
0x3526d000 - 0x352b0fff SystemConfiguration armv6 <207f362e707871e74a292cf1ea7893d> /System/Library/Frameworks/SystemConfiguration.framework/SystemConfiguration
0x352bc000 - 0x352d6fff libJapaneseConverter.dylib armv6 <365e98eb95228776f7982d8fa641e10a> /System/Library/CoreServices/Encodings/libJapaneseConverter.dylib
0x354cb000 - 0x354d1fff MobileKeyBag armv6 <2d83bf6a43bab972d77a1a6e0f3b03d2> /System/Library/PrivateFrameworks/MobileKeyBag.framework/MobileKeyBag
0x354e0000 - 0x355f4fff RawCamera armv6 <a431d55459002236dc9954046ee62900> /System/Library/CoreServices/RawCamera.bundle/RawCamera
0x3564b000 - 0x3569aff Security armv6 <cf625b4dc7ea928891313444ef64a7cb> /System/Library/Frameworks/Security.framework/Security
0x357ca000 - 0x35835fff libstdc++.6.dylib armv6 <eccd1d7183e73587b2c0aa5755a19c39> /usr/lib/libstdc++.6.dylib
0x35836000 - 0x35850fff MediaControl armv6 <e28edb7fc22ba1c7ad921bf9b97b19b3> /System/Library/PrivateFrameworks/MediaControl.framework/MediaControl
0x35851000 - 0x35885fff AppSupport armv6 <783e14db9585fd063c0c2a755cd121b6> /System/Library/PrivateFrameworks/AppSupport.framework/AppSupport
0x358fc000 - 0x35936fff MobileCoreServices armv6 <beb473ce80390554bb4af21554522286> /System/Library/Frameworks/MobileCoreServices.framework/MobileCoreServices
0x35cdb000 - 0x35d25fff libCGFreetype.A.dylib armv6 <cf94cfa17958f2f94c9eff208a7dace> /System/Library/Frameworks/CoreGraphics.framework/Resources/libCGFreetype.A.dylib
0x35d51000 - 0x35e47fff libxml2.2.dylib armv6 <9c44d05cc67f1ebabd795903e581724e> /usr/lib/libxml2.2.dylib
0x35e64000 - 0x35f4ffff CoreFoundation armv6 <ab0eac0ddd5b4ae1bf8541116e3c0bd1> /System/Library/Frameworks/CoreFoundation.framework/CoreFoundation
0x35f76000 - 0x3606aff LibIconv.2.dylib armv6 <01916d6784f4de8f3746978faae9c5fa> /usr/lib/libiconv.2.dylib
0x360f0000 - 0x361edfff JavaScriptCore armv6 <3547c92c1efc0522b087e7f10eba7728> /System/Library/PrivateFrameworks/JavaScriptCore.framework/JavaScriptCore
0x361ee000 - 0x361fbfff MobileBluetooth armv6 <2b68516e1321011a4efbee2947d463c6> /System/Library/PrivateFrameworks/MobileBluetooth.framework/MobileBluetooth
0x361fc000 - 0x36225fff StoreServices armv6 <4606b36354f613074c37bdd85b2c53ed> /System/Library/PrivateFrameworks/StoreServices.framework/StoreServices
0x36226000 - 0x36252fff Preferences armv6 <afc3eab3e68a2fc0132bac2aa3fa4bd> /System/Library/PrivateFrameworks/Preferences.framework/Preferences
0x366e5000 - 0x3674dff GMM armv6 <adcdfedd491ac237b385b6c2d7f684e3> /System/Library/PrivateFrameworks/GMM.framework/GMM
0x36752000 - 0x36771fff Bom armv6 <f41bef81e23e2bff59155e5ce46762d3> /System/Library/PrivateFrameworks/Bom.framework/Bom
0x36772000 - 0x36814fff AVFoundation armv6 <da9d96f32791f51ecb439c5eaeeff59a> /System/Library/Frameworks/AVFoundation.framework/AVFoundation
0x36826000 - 0x36830fff AggregateDictionary armv6 <f7429444c955e4f13c6761d20032ab52> /S

```
ystem/Library/PrivateFrameworks/AggregateDictionary.framework/AggregateDictionary  
0x36849000 - 0x36850fff libblockdown.dylib armv6 <f470dea180ddf23886df75eb256d3888> /usr  
/lib/libblockdown.dylib  
0x368cc000 - 0x3711bfff WebCore armv6 <aa3b6827f051da7a3494c9bee4ebe290> /System/Library/  
PrivateFrameworks/WebCore.framework/WebCore  
0x3713e000 - 0x3714cff libz.1.dylib armv6 <84592e96bae1a661374b0f9a5d03a3a0> /usr/lib/  
libz.1.dylib  
0x3718e000 - 0x371dcfff CoreText armv6 <16c9582fdfffb598178287c6ce9fd6897> /System/Library/  
Frameworks/CoreText.framework/CoreText  
0x371dd000 - 0x371eefff DataAccessExpress armv6 <c112bd2791eb706526db25407ec117d4> /Sys  
tem/Library/PrivateFrameworks/DataAccessExpress.framework/DataAccessExpress  
0x37378000 - 0x373defff libBLAS.dylib armv6 <11a3677a08175a30df1b3d66d7e0951a> /System/  
Library/Frameworks/Accelerate.framework/Frameworks/vecLib.framework/libBLAS.dylib  
0x373df000 - 0x37436fff CoreMedia armv6 <cd5e9398c161f129146931e888e1c92e> /System/Library/  
Frameworks/CoreMedia.framework/CoreMedia  
0x37437000 - 0x3757aff MediaPlayer armv6 <47513ac46e4142e68d0af6d6f7abab88> /System/Li  
brary/Frameworks/MediaPlayer.framework/MediaPlayer  
0x3757d000 - 0x375a7fff PrintKit armv6 <74f9710fa01a33b5bb04c4aeabd6be7d> /System/Libr  
ary/PrivateFrameworks/PrintKit.framework/PrintKit  
0x375bf000 - 0x375c7fff IAP armv6 <0e9f5da9eff05f936c2c363ce7d179dd> /System/Library/Pr  
ivateFrameworks/IAP.framework/IAP  
0x3777f000 - 0x37840fff ImageIO armv6 <0c1b6f466667ff345f2399d8142a9d10> /System/Librar  
y/Frameworks/ImageIO.framework/ImageIO  
0x378b3000 - 0x3791bfff libvDSP.dylib armv6 <9d264733fc675943c082bd3b9b567b59> /System/  
Library/Frameworks/Accelerate.framework/Frameworks/vecLib.framework/libvDSP.dylib  
0x37a1f000 - 0x37a31fff libbsm.0.dylib armv6 <51e7bb18da9afa44a33e54e42fdb0707> /usr/li  
b/libbsm.0.dylib
```

實戰：背景作業執行太久

這是 Facebook iOS App 的 crash log。我們看到 BKProcessAssertion 這個關鍵字，就可以大致知道，這個 crash 跟背景作業有關。

從 iOS 4 開始，iOS 支援背景作業，背景作業分成時間有限與時間無限兩種。時間無限是指必須要在 Info.plist 指定特定 background mode 的那種，像 KKBOX 是一個可以在背景播放音樂的 App，就是因為設定了 audio 這種 background mode。

而其實所有的 App 在不用設定 background mode 的狀況下，在進入背景的時候，都還可以用 UIApplication 的 `beginBackgroundTaskWithName:expirationHandler:` 執行一個 block，用這個 block 把想做的事情做完，但是這個 block 有時間限制，在 iOS 7 之前最長十分鐘，iOS 7 之後只有三分鐘，如果超過時間，就會被系統強制結束。被強制結束的時候是什麼樣子呢？就是這個 BKProcessAssertion 了。

在蘋果官方文件 [App Programming Guide for iOS - Background Execution](#) 中，大概示範了 `beginBackgroundTaskWithName:expirationHandler:` 的用法，我們就用蘋果的 Sample code 說明問題出在哪裡。

```
- (void)applicationDidEnterBackground:(UIApplication *)application
{
    bgTask = [application beginBackgroundTaskWithName:@"MyTask" expirationHandler:^{
        // 在這邊你可以寫想要背景執行的 code
        // 但事情做太久，還是會被強制結束
        // 強制結束的時候會產生

        [application endBackgroundTask:bgTask];
        bgTask = UIBackgroundTaskInvalid;
    }];

    // Start the long-running task and return immediately.
    dispatch_async(dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0), ^{
        [application endBackgroundTask:bgTask];
        bgTask = UIBackgroundTaskInvalid;
    });
}
```

不過 Facebook 這個 crash log 有個地方還頂值得玩味。出問題的 thread 是第四條叫做 fbtelephonicache 的 thread，telephony 看名字是跟撥打電話有關，而我自己想不出來 Facebook 跟撥打電話有什麼關係，也搞不清楚這條 thread 在 cache 些什麼。

```

Incident Identifier: B1D7316A-FD33-4DF0-916B-DB9ACD1D4D90
CrashReporter Key: 5014cd6a9eed070777d980dc79669e705b635bad
Hardware Model: iPhone6,2
Process: Facebook [13362]
Path: /private/var/mobile/Containers/Bundle/Application/364EA2D0-A60E-4398
-8706-D1BFDC02EA47/Facebook.app/Facebook
Identifier: com.facebook.Facebook
Version: 8291884 (27.0)
Code Type: ARM (Native)
Parent Process: launchd [1]

Date/Time: 2015-04-03 14:02:13.306 +0800
Launch Time: 2015-04-03 14:01:30.127 +0800
OS Version: iOS 8.2 (12D508)
Report Version: 105

Exception Type: 00000020
Exception Codes: 0x000000008badf00d
Highlighted Thread: 4

Application Specific Information:
<BKNewProcess: 0x12de5ce60; com.facebook.Facebook; pid: 13362; hostpid: -1> has active assertions beyond permitted time:
{
    <BKProcessAssertion: 0x12dd36310> id: 7899-78F88300-DBEB-4A82-A9A4-C34702F64B09 name:
        Background Content Fetching (1065) process: <BKNewProcess: 0x12de5ce60; com.facebook.Facebook; pid: 13362; hostpid: -1> permittedBackgroundDuration: 30.000000 reason: backgroundContentFetching owner pid:7899 preventSuspend preventThrottleDownUI preventIdleSleep preventSuspendOnSleep
}
}

Elapsed total CPU time (seconds): 15.290 (user 15.290, system 0.000), 30% CPU
Elapsed application CPU time (seconds): 0.262, 1% CPU

Thread 0 name: Dispatch queue: com.apple.main-thread
Thread 0:
0  libsystem_kernel.dylib          0x36d7c474 mach_msg_trap + 20
1  libsystem_kernel.dylib          0x36d7c269 mach_msg + 37
2  CoreFoundation                 0x2883c57f __CFRunLoopServiceMachPort + 143
3  CoreFoundation                 0x2883ab25 __CFRunLoopRun + 1013
4  CoreFoundation                 0x287883ad CFRunLoopRunSpecific + 473
5  CoreFoundation                 0x287881bf CFRunLoopRunInMode + 103
6  GraphicsServices               0x2fd751fd GSEventRunModal + 133
7  UIKit                          0x2bdf2439 UIApplicationMain + 1437
8  Facebook                       0x0007181f 0x63000 + 59423
9  libdyld.dylib                  0x36cc5aad start + 1

Thread 1 name: Dispatch queue: com.apple.libdispatch-manager
Thread 1:
0  libsystem_kernel.dylib          0x36d7c224 kevent64 + 24
1  libdispatch.dylib               0x36cb10ed _dispatch_migr_invoke + 277
2  libdispatch.dylib               0x36ca5d37 _dispatch_migr_thread + 35

```

```

Thread 2:
0  libsystem_kernel.dylib          0x36d909c0 __workq_kernreturn + 8
1  libsystem_pthread.dylib        0x36e0ae39 _pthread_wqthread + 789
2  libsystem_pthread.dylib        0x36e0ab10 start_wqthread + 4

Thread 3:
0  libsystem_kernel.dylib          0x36d7c474 mach_msg_trap + 20
1  libsystem_kernel.dylib          0x36d7c269 mach_msg + 37
2  Facebook                      0x000725f3 0x63000 + 62963
3  libsystem_pthread.dylib        0x36e0ce21 _pthread_body + 137
4  libsystem_pthread.dylib        0x36e0cd93 _pthread_start + 115
5  libsystem_pthread.dylib        0x36e0ab1c thread_start + 4

Thread 4 name: fbtelephonycache
Thread 4:
0  libsystem_kernel.dylib          0x36d7c474 mach_msg_trap + 20
1  libsystem_kernel.dylib          0x36d7c269 mach_msg + 37
2  CoreFoundation                 0x2883c57f __CFRunLoopServiceMachPort + 143
3  CoreFoundation                 0x2883ab25 __CFRunLoopRun + 1013
4  CoreFoundation                 0x287883ad CFRunLoopRunSpecific + 473
5  CoreFoundation                 0x287881bf CFRunLoopRunInMode + 103
6  Facebook                      0x0009a6fb 0x63000 + 227067
7  Foundation                     0x295ab687 __NSThread_main_ + 1115
8  libsystem_pthread.dylib        0x36e0ce21 _pthread_body + 137
9  libsystem_pthread.dylib        0x36e0cd93 _pthread_start + 115
10 libsystem_pthread.dylib       0x36e0ab1c thread_start + 4

Thread 5 name: AVAudioSession Notify Thread
Thread 5:
0  libsystem_kernel.dylib          0x36d7c474 mach_msg_trap + 20
1  libsystem_kernel.dylib          0x36d7c269 mach_msg + 37
2  CoreFoundation                 0x2883c57f __CFRunLoopServiceMachPort + 143
3  CoreFoundation                 0x2883ab25 __CFRunLoopRun + 1013
4  CoreFoundation                 0x287883ad CFRunLoopRunSpecific + 473
5  CoreFoundation                 0x287881bf CFRunLoopRunInMode + 103
6  libAVFAudio.dylib              0x2748c3eb GenericRunLoopThread::Entry(void*) + 131
7  libAVFAudio.dylib              0x2747e909 CAPThread::Entry(CAPThread*) + 193
8  libsystem_pthread.dylib        0x36e0ce21 _pthread_body + 137
9  libsystem_pthread.dylib        0x36e0cd93 _pthread_start + 115
10 libsystem_pthread.dylib       0x36e0ab1c thread_start + 4

Thread 6 name: com.apple.NSURLConnectionLoader
Thread 6:
0  libsystem_kernel.dylib          0x36d7c474 mach_msg_trap + 20
1  libsystem_kernel.dylib          0x36d7c269 mach_msg + 37
2  CoreFoundation                 0x2883c57f __CFRunLoopServiceMachPort + 143
3  CoreFoundation                 0x2883ab25 __CFRunLoopRun + 1013
4  CoreFoundation                 0x287883ad CFRunLoopRunSpecific + 473
5  CoreFoundation                 0x287881bf CFRunLoopRunInMode + 103
6  CFNetwork                     0x2833c8f3+[NSURLConnection(Loader) _resourceLoadL
oop:] + 483
7  Foundation                     0x295ab687 __NSThread_main_ + 1115

```

```

8  libsystem_pthread.dylib          0x36e0ce21 _pthread_body + 137
9  libsystem_pthread.dylib          0x36e0cd93 _pthread_start + 115
10 libsystem_pthread.dylib         0x36e0ab1c thread_start + 4

Thread 7:
0  libsystem_kernel.dylib          0x36d909c0 __workq_kernreturn + 8
1  libsystem_pthread.dylib          0x36e0ae39 _pthread_wqthread + 789
2  libsystem_pthread.dylib          0x36e0ab10 start_wqthread + 4

Thread 8 name: com.apple.coremedia.player.async
Thread 8:
0  libsystem_kernel.dylib          0x36d7c4c4 semaphore_wait_trap + 8
1  libdispatch.dylib               0x36caf5db _dispatch_semaphore_wait_slow + 187
2  MediaToolbox                   0x2a290b67 fpa_AsyncMovieControlThread + 1963
3  CoreMedia                       0x28fe0c79 figThreadMain + 185
4  libsystem_pthread.dylib          0x36e0ce21 _pthread_body + 137
5  libsystem_pthread.dylib          0x36e0cd93 _pthread_start + 115
6  libsystem_pthread.dylib          0x36e0ab1c thread_start + 4

No thread state (register information) available
Binary Images:
0x63000 - 0x1caafff Facebook armv7 <50d489f108813a0e9bd2e093de509fe6> /var/mobile/Containers/Bundle/Application/364EA2D0-A60E-4398-8706-D1BFDC02EA47/Facebook.app/Facebook
0x1fed5000 - 0x1fef8fff dyld armv7s <d959cf6ea9b23eebac21b656a5551dab> /usr/lib/dyld
0x26f3f000 - 0x26f47fff AccessibilitySettingsLoader armv7s <b7ae00ad3cd73534ba0365cec43b1a32> /System/Library/AccessibilityBundles/AccessibilitySettingsLoader.bundle/AccessibilitySettingsLoader
0x26fb6000 - 0x26fbffff QuickSpeak armv7s <45d0907fcc743e76829c47375da10dc> /System/Library/AccessibilityBundles/QuickSpeak.bundle/QuickSpeak
0x27306000 - 0x27472fff AVFoundation armv7s <339e734775eb39b982ec0c089a752320> /System/Library/Frameworks/AVFoundation.framework/AVFoundation
0x27473000 - 0x274d1fff libAVFAudio.dylib armv7s <ef35407264a93f56a53a36dc11b72544> /System/Library/Frameworks/AVFoundation.framework/libAVFAudio.dylib
0x274d2000 - 0x2750afff AVKit armv7s <b32eb1e922d732f581534c647f9f1894> /System/Library/Frameworks/AVKit.framework/AVKit
0x2750b000 - 0x2750bfff Accelerate armv7s <b170327a82973885aba35a81ee82882b> /System/Library/Frameworks/Accelerate.framework/Accelerate
0x2751c000 - 0x27737fff vImage armv7s <e6895345bb03617a31c13774e775579> /System/Library/Frameworks/Accelerate.framework/Frameworks/vImage.framework/vImage
0x27738000 - 0x2781efff libBLAS.dylib armv7s <153f3233991f3c47b762ef1743c32e0a> /System/Library/Frameworks/Accelerate.framework/Frameworks/vecLib.framework/libBLAS.dylib
0x2781f000 - 0x27ae3fff libLAPACK.dylib armv7s <aa5471640b8b3bb3b4dd3fad5ed697db> /System/Library/Frameworks/Accelerate.framework/Frameworks/vecLib.framework/libLAPACK.dylib
0x27ae4000 - 0x27af5fff libLinearAlgebra.dylib armv7s <1ea6ed99c4863d4085eb884e9a616903> /System/Library/Frameworks/Accelerate.framework/Frameworks/vecLib.framework/libLinearAlgebra.dylib
0x27af6000 - 0x27b72fff libvDSP.dylib armv7s <708711e55e7c3d67a44ca33803d225af> /System/Library/Frameworks/Accelerate.framework/Frameworks/vecLib.framework/libvDSP.dylib
0x27b73000 - 0x27b85fff libvMisc.dylib armv7s <3c7e8723a7233076a6a0ff239e4c58eb> /System/Library/Frameworks/Accelerate.framework/Frameworks/vecLib.framework/libvMisc.dylib
0x27b86000 - 0x27b86fff vecLib armv7s <0a8061e9131332f28e903a478d0b6e36> /System/Library/Frameworks/Accelerate.framework/Frameworks/vecLib.framework/vecLib
0x27b87000 - 0x27baefff Accounts armv7s <09f1e91f2f903f0d9e91ea22c584cfba> /System/Library

```

```
ry/Frameworks/Accounts.framework/Accounts
0x27baf000 - 0x27baffff AdSupport armv7s <76c5dc22151d3689b300971ef780e2a8> /System/Library/Frameworks/AdSupport.framework/AdSupport
0x27bb0000 - 0x27c1efff AddressBook armv7s <fdcde0e632433fccbf0f758694bb5e40> /System/Library/Frameworks/AddressBook.framework/AddressBook
0x27c1f000 - 0x27d48fff AddressBookUI armv7s <4a01e230f037317d82a10b210d2a292a> /System/Library/Frameworks/AddressBookUI.framework/AddressBookUI
0x27d49000 - 0x27d5bfff AssetsLibrary armv7s <3fcc2aed41f7349089ac37874db2215a> /System/Library/Frameworks/AssetsLibrary.framework/AssetsLibrary
0x27ee1000 - 0x28155fff AudioToolbox armv7s <c9e12d7ce99833ce969b53099cd194ba> /System/Library/Frameworks/AudioToolbox.framework/AudioToolbox
0x282c0000 - 0x28449fff CFNetwork armv7s <d52bddca8cf330959bc40eb43e26d30d> /System/Library/Frameworks/CFNetwork.framework/CFNetwork
0x2844a000 - 0x284cbfff CloudKit armv7s <aefe5795b29833608d0fa6ca7ffc2218> /System/Library/Frameworks/CloudKit.framework/CloudKit
0x284cc000 - 0x2852bfff CoreAudio armv7s <bb6bc21edca73eb49d2631266ba46d82> /System/Library/Frameworks/CoreAudio.framework/CoreAudio
0x2853e000 - 0x28545fff CoreAuthentication armv7s <15f15f4a928d306db3dad9920dfb4d64> /System/Library/Frameworks/CoreAuthentication.framework/CoreAuthentication
0x28546000 - 0x28563fff CoreBluetooth armv7s <6ab741ad5acc333786f134f936878578> /System/Library/Frameworks/CoreBluetooth.framework/CoreBluetooth
0x28564000 - 0x2876ffff CoreData armv7s <7415681a633b37b293ad2fbe776b3396> /System/Library/Frameworks/CoreData.framework/CoreData
0x28770000 - 0x28a9dfff CoreFoundation armv7s <9924d9f1b68836cfa0dc45ad776a10ba> /System/Library/Frameworks/CoreFoundation.framework/CoreFoundation
0x28a9e000 - 0x28bc8fff CoreGraphics armv7s <e50b5639d14a39d0b83449cd48707fcc> /System/Library/Frameworks/CoreGraphics.framework/CoreGraphics
0x28c08000 - 0x28c0afff libCGXType.A.dylib armv7s <c816f78ae8113d599d194d04d501c6e1> /System/Library/Frameworks/CoreGraphics.framework/Resources/libCGXType.A.dylib
0x28c0b000 - 0x28c15fff libCMSBuiltIn.A.dylib armv7s <a5011cd35faa344d983be6e9224983c5> /System/Library/Frameworks/CoreGraphics.framework/Resources/libCMSBuiltIn.A.dylib
0x28dfd000 - 0x28e18fff libRIP.A.dylib armv7s <c2266f3ec31c38798f113d59c5102db7> /System/Library/Frameworks/CoreGraphics.framework/Resources/libRIP.A.dylib
0x28e19000 - 0x28f27fff CoreImage armv7s <44b768c0b39c3197b6ecc5bed08cd0af> /System/Library/Frameworks/CoreImage.framework/CoreImage
0x28f28000 - 0x28f80fff CoreLocation armv7s <27f6fb3e7d053990989c057d0876bd8b> /System/Library/Frameworks/CoreLocation.framework/CoreLocation
0x28fb2000 - 0x2904dfff CoreMedia armv7s <3afe18bd92493ccb94501e9ffb07eb89> /System/Library/Frameworks/CoreMedia.framework/CoreMedia
0x2904e000 - 0x2912bfff CoreMotion armv7s <c6e4968c002738f18dbf7169c232b994> /System/Library/Frameworks/CoreMotion.framework/CoreMotion
0x2912c000 - 0x2918afff CoreTelephony armv7s <580b75cd06563449931429e0fc6dd16b> /System/Library/Frameworks/CoreTelephony.framework/CoreTelephony
0x2918b000 - 0x29257fff CoreText armv7s <b8f896f010923eaebd88f04d9fa92db5> /System/Library/Frameworks/CoreText.framework/CoreText
0x29258000 - 0x2926dff CoreVideo armv7s <982f9388857d38dd98d7f4b793cefd74> /System/Library/Frameworks/CoreVideo.framework/CoreVideo
0x2926e000 - 0x29363fff EventKit armv7s <5b8e871f5568346099b77f9e6974ef9a> /System/Library/Frameworks/EventKit.framework/EventKit
0x294db000 - 0x296defff Foundation armv7s <c5d6421377e13c2e8bf4cbd917d874b1> /System/Library/Frameworks/Foundation.framework/Foundation
0x296df000 - 0x2970afff GLKit armv7s <a48c06955f193a34a468beb245689cba> /System/Library/Frameworks/GLKit.framework/GLKit
```

0x2970b000 - 0x2972afff GSS armv7s <3b3cf67b07403c02a0002d795ea46fbf> /System/Library/Frameworks/GSS.framework/GSS
0x297de000 - 0x29833fff IOKit armv7s <c23ce2b864ec3c9ca424fab1145e6241> /System/Library/Frameworks/IOKit.framework/Versions/A/IOKit
0x29834000 - 0x29a76fff ImageIO armv7s <7c57f7fe7e1c3604a8d86053c716cdf2> /System/Library/Frameworks/ImageIO.framework/ImageIO
0x29a77000 - 0x29dc5fff JavaScriptCore armv7s <7adced020bfc37a595c0811423f9fde8> /System/Library/Frameworks/JavaScriptCore.framework/JavaScriptCore
0x29f8c000 - 0x29f92fff LocalAuthentication armv7s <89a7fb34c0643f559c8a3999e3320be8> /System/Library/Frameworks/LocalAuthentication.framework/LocalAuthentication
0x29f93000 - 0x29fa2fff SharedUtils armv7s <290a076038053d4dbeb715ae41c3b98a> /System/Library/Frameworks/LocalAuthentication.framework/Support/SharedUtils.framework/SharedUtils
0x29fa3000 - 0x2a0a2fff MapKit armv7s <41e70af583b13c05bbe7d7cbe5ed3295> /System/Library/Frameworks/MapKit.framework/MapKit
0x2a0a3000 - 0x2a0abfff MediaAccessibility armv7s <4cea4ac5cf4303a86b331c10e7ce8a6> /System/Library/Frameworks/MediaAccessibility.framework/MediaAccessibility
0x2a0ac000 - 0x2a28bfff MediaPlayer armv7s <3a4bc2cf8653621ad5944b42cad8fdc> /System/Library/Frameworks/MediaPlayer.framework/MediaPlayer
0x2a28c000 - 0x2a606fff MediaToolbox armv7s <b3cab8f05a4e32e193e5e1da502727a3> /System/Library/Frameworks/MediaToolbox.framework/MediaToolbox
0x2a607000 - 0x2a6c5fff MessageUI armv7s <f41ef6b613fc32f8b1291ca26327e403> /System/Library/Frameworks/MessageUI.framework/MessageUI
0x2a6c6000 - 0x2a732fff Metal armv7s <00a89ca37d0834898a8e81b76357fc1> /System/Library/Frameworks/Metal.framework/Metal
0x2a733000 - 0x2a7c6fff MobileCoreServices armv7s <01ea079439393391821634683ccb412a> /System/Library/Frameworks/MobileCoreServices.framework/MobileCoreServices
0x2a8bd000 - 0x2a8eefff OpenAL armv7s <6e48355ff76d31c480f0ad0caed1daf7> /System/Library/Frameworks/OpenAL.framework/OpenAL
0x2b2a4000 - 0x2b2acfff OpenGL ES armv7s <02aa6ebd148a3449a5fb73365d45ea87> /System/Library/Frameworks/OpenGL ES.framework/OpenGL ES
0x2b2ae000 - 0x2b2aefff libCVMSPPluginSupport.dylib armv7s <18deef6946913a25969285502316b674> /System/Library/Frameworks/OpenGL ES.framework/libCVMSPPluginSupport.dylib
0x2b2af000 - 0x2b2b1fff libCoreFSCache.dylib armv7s <d252e7981a543f6580ba951a037d2590> /System/Library/Frameworks/OpenGL ES.framework/libCoreFSCache.dylib
0x2b2b2000 - 0x2b2b5fff libCoreVMClient.dylib armv7s <405d9912249e3a3a9e60bf9edcb19d8> /System/Library/Frameworks/OpenGL ES.framework/libCoreVMClient.dylib
0x2b2b6000 - 0x2b2befff libGFXShared.dylib armv7s <248be3146cab319292bd30806d11f792> /System/Library/Frameworks/OpenGL ES.framework/libGFXShared.dylib
0x2b2bf000 - 0x2b301fff libGLImage.dylib armv7s <1e9a22a10aa2364ebdc68d7f8a153e6> /System/Library/Frameworks/OpenGL ES.framework/libGLImage.dylib
0x2b45c000 - 0x2b50dff PassKit armv7s <04b0fadfdf4232ab8260b9424c148efa> /System/Library/Frameworks/PassKit.framework/PassKit
0x2b50e000 - 0x2b582fff Photos armv7s <af5cf98f67230bd95fac97fcf18b391> /System/Library/Frameworks/Photos.framework/Photos
0x2b7aa000 - 0x2b8fcfff QuartzCore armv7s <87f4d4be177f3a9c8c251da670dcf45b> /System/Library/Frameworks/QuartzCore.framework/QuartzCore
0x2b8fd000 - 0x2b950fff QuickLook armv7s <ab076503a86c325f80e2c50d05ffe636> /System/Library/Frameworks/QuickLook.framework/QuickLook
0x2bb3f000 - 0x2bb80fff Security armv7s <7434b508472d355298f93d5e8a5e725e> /System/Library/Frameworks/Security.framework/Security
0x2bb81000 - 0x2bbf9fff Social armv7s <50be0cb4d06334999dbe7a505708bb84> /System/Library/Frameworks/Social.framework/Social
0x2bd0e000 - 0x2bd23fff StoreKit armv7s <7b8c6b8baee035b18e2d0b09ac219d1d> /System/Library

```
ry/Frameworks/StoreKit.framework/StoreKit
0x2bd24000 - 0x2bd81fff SystemConfiguration armv7s <53b886e1d32f32a2b6c0cb2e0d93ccf2> /System/Library/Frameworks/SystemConfiguration.framework/SystemConfiguration
0x2bd84000 - 0x2c62afff UIKit armv7s <81da17cbaad83ba5acaa630781446352> /System/Library/Frameworks/UIKit.framework/UIKit
0x2c62b000 - 0x2c692fff VideoToolbox armv7s <867315c8522039d3aeb629502059f6d6> /System/Library/Frameworks/VideoToolbox.framework/VideoToolbox
0x2c6c2000 - 0x2c8a8fff WebKit armv7s <8fd2d7a7389c3efa89700e371946577f> /System/Library/Frameworks/WebKit.framework/WebKit
0x2cb99000 - 0x2cba4fff AOSNotification armv7s <d3e35dcc6918381aa0745eedc97f80a4> /System/Library/PrivateFrameworks/AOSNotification.framework/AOSNotification
0x2cc1c000 - 0x2cc41fff AXRuntime armv7s <58248fce427e3c21aef462b1d2d02956> /System/Library/PrivateFrameworks/AXRuntime.framework/AXRuntime
0x2cc6d000 - 0x2cc74fff AccessibilityUI armv7s <3db469f259e63bb2ad1ed65678ee3127> /System/Library/PrivateFrameworks/Accessibility.framework/Frameworks/AccessibilityUI.framework/AccessibilityUI
0x2cc8c000 - 0x2cc8ffff AccessibilityUIUtilities armv7s <ce7636c546db3e278945bc13b84bbaa> /System/Library/PrivateFrameworks/Accessibility.framework/Frameworks/AccessibilityUIUtilities.framework/AccessibilityUIUtilities
0x2cc92000 - 0x2cc95fff ZoomServices armv7s <57a9fb8122bd381ebdbd23cb59087649> /System/Library/PrivateFrameworks/Accessibility.framework/Frameworks/ZoomServices.framework/ZoomServices
0x2cc96000 - 0x2ccf0fff AccessibilityUtilities armv7s <0baabefefc0d38d182e271f978422b6b> /System/Library/PrivateFrameworks/AccessibilityUtilities.framework/AccessibilityUtilities
0x2ccfe000 - 0x2cd4efff AccountsDaemon armv7s <65fd994bdc603127acb44c690381e141> /System/Library/PrivateFrameworks/AccountsDaemon.framework/AccountsDaemon
0x2cd4f000 - 0x2cd70fff AccountsUI armv7s <88a4f07ebb583241b94abaa4db1856a4> /System/Library/PrivateFrameworks/AccountsUI.framework/AccountsUI
0x2cd71000 - 0x2cd75fff AggregateDictionary armv7s <1f4aae1e74f230488125530b315d01f6> /System/Library/PrivateFrameworks/AggregateDictionary.framework/AggregateDictionary
0x2cf3e000 - 0x2cf69fff AirPlaySupport armv7s <c727539915aa32bc80dd90711c9d309c> /System/Library/PrivateFrameworks/AirPlaySupport.framework/AirPlaySupport
0x2d16f000 - 0x2d1adfff AppSupport armv7s <baba5ec9cb6c3b5e8097412705a57bd7> /System/Library/PrivateFrameworks/AppSupport.framework/AppSupport
0x2d1ae000 - 0x2d1f6fff AppleAccount armv7s <018fcce59c923c9ab9e2ea148fe1f2a4> /System/Library/PrivateFrameworks/AppleAccount.framework/AppleAccount
0x2d2ce000 - 0x2d2defff AppleIDSSOAuthentication armv7s <9c9def717ceb373ea121b3e49d116078> /System/Library/PrivateFrameworks/AppleIDSSOAuthentication.framework/AppleIDSSOAuthentication
0x2d2df000 - 0x2d31cfffc AppleJPEG armv7s <7ccf7d8822283911a26b7500b3b437b2> /System/Library/PrivateFrameworks/AppleJPEG.framework/AppleJPEG
0x2d327000 - 0x2d339fff ApplePushService armv7s <32ae1ae398cc3f0e9ec0426ca0870d34> /System/Library/PrivateFrameworks/ApplePushService.framework/ApplePushService
0x2d33a000 - 0x2d340fff AppleSRP armv7s <8b69881099043bf2bd8b3ce229ad1b32> /System/Library/PrivateFrameworks/AppleSRP.framework/AppleSRP
0x2d375000 - 0x2d37efff AssertionServices armv7s <f55c714a5a013c528dfbe7c0d1689b91> /System/Library/PrivateFrameworks/AssertionServices.framework/AssertionServices
0x2d37f000 - 0x2d397fff AssetsLibraryServices armv7s <2eeb5c26a5243db08a8fc5ae68d154e7> /System/Library/PrivateFrameworks/AssetsLibraryServices.framework/AssetsLibraryServices
0x2d398000 - 0x2d3cefff AssistantServices armv7s <4231dc07a0b33cebaec3320a6ac7cf88> /System/Library/PrivateFrameworks/AssistantServices.framework/AssistantServices
0x2d3fe000 - 0x2d402fff BTLEAudioController armv7s <09bd58dd309a35fc0d0797970b740e6b> /S
```

```

ystem/Library/PrivateFrameworks/BTLEAudioController.framework/BTLEAudioController
0x2d403000 - 0x2d41afff BackBoardServices armv7s <205d23f721fd35e584853e2d487af9b3> /Sys
tem/Library/PrivateFrameworks/BackBoardServices.framework/BackBoardServices
0x2d41d000 - 0x2d452fff BaseBoard armv7s <d780982e45423486ab6b6b6ad21abd23> /System/Libr
ary/PrivateFrameworks/BaseBoard.framework/BaseBoard
0x2d453000 - 0x2d459fff BluetoothManager armv7s <753fc00efc263cd9a98eb33ac6898fc5> /Syst
em/Library/PrivateFrameworks/BluetoothManager.framework/BluetoothManager
0x2d45a000 - 0x2d480fff Bom armv7s <e4cf22bce7a53599bcccdaf89140fdc0> /System/Library/Pr
ivateFrameworks/Bom.framework/Bom
0x2d552000 - 0x2d559fff CacheDelete armv7s <796e062ad6f13a1db26efcc99360cefc> /System/Li
brary/PrivateFrameworks/CacheDelete.framework/CacheDelete
0x2d59e000 - 0x2d5c7fff CalendarFoundation armv7s <98f804e98b9a3cd8b8db61afb60fa766> /Sy
stem/Library/PrivateFrameworks/CalendarFoundation.framework/CalendarFoundation
0x2d6de000 - 0x2d6e6fff CaptiveNetwork armv7s <10fccd2e42fe35919ec80d881c735054> /System
/Library/PrivateFrameworks/CaptiveNetwork.framework/CaptiveNetwork
0x2d6e7000 - 0x2d809fff Celestial armv7s <9738f5492d733592a8c0e70100d4fe8b> /System/Libr
ary/PrivateFrameworks/Celestial.framework/Celestial
0x2d817000 - 0x2d82ffff CertInfo armv7s <81941a4b91e435e987daf432e43faef3> /System/Libra
ry/PrivateFrameworks/CertInfo.framework/CertInfo
0x2d830000 - 0x2d835fff CertUI armv7s <8f7c4dfd3aca3ef789dbb4551f65a122> /System/Library
/PrivateFrameworks/CertUI.framework/CertUI
0x2d972000 - 0x2d993fff ChunkingLibrary armv7s <4b029225b4033493aa979657e5503a16> /Syst
em/Library/PrivateFrameworks/ChunkingLibrary.framework/ChunkingLibrary
0x2dd42000 - 0x2ddedfff CloudPhotoLibrary armv7s <d0d018f2c56130dc8ed830e5acd55c42> /Sys
tem/Library/PrivateFrameworks/CloudPhotoLibrary.framework/CloudPhotoLibrary
0x2de3e000 - 0x2de40fff CommonAuth armv7s <592c71c8a6fe3c1a880313f6f345137a> /System/Lib
rary/PrivateFrameworks/CommonAuth.framework/CommonAuth
0x2de41000 - 0x2de51fff CommonUtilities armv7s <f3e078abb75f3788b1d2b2a564eb786e> /Syst
em/Library/PrivateFrameworks/CommonUtilities.framework/CommonUtilities
0x2de52000 - 0x2de56fff CommunicationsFilter armv7s <e7342616fb730888bf4c6db4ccaa4e2> /S
ystem/Library/PrivateFrameworks/CommunicationsFilter.framework/CommunicationsFilter
0x2df17000 - 0x2df1bfff ConstantClasses armv7s <44624a0f6f633645a00a3cf9a30b4c23> /Syst
em/Library/PrivateFrameworks/ConstantClasses.framework/ConstantClasses
0x2df1c000 - 0x2df54fff ContentIndex armv7s <c76c9897423c3a4cb583f0c28ff83f3a> /System/L
ibrary/PrivateFrameworks/ContentIndex.framework/ContentIndex
0x2df55000 - 0x2df58fff CoreAUC armv7s <a9b4259de0743f539097e0a709ae357b> /System/Librar
y/PrivateFrameworks/CoreAUC.framework/CoreAUC
0x2df85000 - 0x2dfd9fff CoreDAV armv7s <fe5b9ce965bd3b2093c337b03be22d1d> /System/Librar
y/PrivateFrameworks/CoreDAV.framework/CoreDAV
0x2dfda000 - 0x2dffafff CoreDuet armv7s <b44ee1884aa53ad4be7a2ad93a7ca163> /System/Libr
ary/PrivateFrameworks/CoreDuet.framework/CoreDuet
0x2e000000 - 0x2e010fff CoreDuetDaemonProtocol armv7s <589ab858f78e3372bbfc55036a1f372a>
/System/Library/PrivateFrameworks/CoreDuetDaemonProtocol.framework/CoreDuetDaemonProtoco
1
0x2e017000 - 0x2e019fff CoreDuetDebugLogging armv7s <a54577d565b83022ad7da2e3c5a5a8f1> /S
ystem/Library/PrivateFrameworks/CoreDuetDebugLogging.framework/CoreDuetDebugLogging
0x2e168000 - 0x2e268fff CoreMediaStream armv7s <ffce49555072331aa985ad018b0b79ef> /Syst
em/Library/PrivateFrameworks/CoreMediaStream.framework/CoreMediaStream
0x2e269000 - 0x2e304fff CorePDF armv7s <93e4925313663cbf970beb535d829f1b> /System/Librar
y/PrivateFrameworks/CorePDF.framework/CorePDF
0x2e365000 - 0x2e36ffff CoreRecents armv7s <7f91e59aaab938f8a9a7677f40b5a1de> /System/Li
brary/PrivateFrameworks/CoreRecents.framework/CoreRecents
0x2e370000 - 0x2e3c8fff CoreRecognition armv7s <d31dcee34a803c7f8e1983e75294eed3> /Syst

```

```
m/Library/PrivateFrameworks/CoreRecognition.framework/CoreRecognition
0x2e3e7000 - 0x2e405fff CoreServicesInternal armv7s <ecd55295b62d386c90ae564978272244> /System/Library/PrivateFrameworks/CoreServicesInternal.framework/CoreServicesInternal
0x2e54e000 - 0x2e5bffff CoreSymbolication armv7s <35cfbe3c7beb36438ddbe04a32176b37> /System/Library/PrivateFrameworks/CoreSymbolication.framework/CoreSymbolication
0x2e605000 - 0x2e685fff CoreUI armv7s <9c6bc8f4fa783388b8c09596402bd051> /System/Library/PrivateFrameworks/CoreUI.framework/CoreUI
0x2e686000 - 0x2e6f0fff CoreUtils armv7s <8054b84ba8a33c148b9689dd3f88aac6> /System/Library/PrivateFrameworks/CoreUtils.framework/CoreUtils
0x2e6f1000 - 0x2e6f6fff CrashReporterSupport armv7s <95aaae72f9fc3756b5bca68275245fab> /System/Library/PrivateFrameworks/CrashReporterSupport.framework/CrashReporterSupport
0x2e6f7000 - 0x2e6fcfff DAAPKit armv7s <e682b52c60cc310f98d26a84c01832e3> /System/Library/PrivateFrameworks/DAAPKit.framework/DAAPKit
0x2e6fd000 - 0x2e707fff DCIMServices armv7s <6da01edd76dc3d448f02f74114c16de6> /System/Library/PrivateFrameworks/DCIMServices.framework/DCIMServices
0x2e708000 - 0x2e74dff DataAccess armv7s <fb7cb30047e43d408f3d35837346d1e4> /System/Library/PrivateFrameworks/DataAccess.framework/DataAccess
0x2e93f000 - 0x2e960fff DataAccessExpress armv7s <d0e876ba262039c5ab0f92acb607eb12> /System/Library/PrivateFrameworks/DataAccessExpress.framework/DataAccessExpress
0x2e96b000 - 0x2e982fff DataDetectorsCore armv7s <ea197848c06839fe99069b46f98de1f7> /System/Library/PrivateFrameworks/DataDetectorsCore.framework/DataDetectorsCore
0x2e99e000 - 0x2e9a4fff DataMigration armv7s <94908a4ed7b034ac8e30c1cae7e4feab> /System/Library/PrivateFrameworks/DataMigration.framework/DataMigration
0x2e9ae000 - 0x2e9affff DiagnosticLogCollection armv7s <9d84890eefe132548cfeaf1f411dcb82> /System/Library/PrivateFrameworks/DiagnosticLogCollection.framework/DiagnosticLogCollection
0x2e9b0000 - 0x2e9caff DictionaryServices armv7s <e13a04ece193371f8dc1bea3773fb476> /System/Library/PrivateFrameworks/DictionaryServices.framework/DictionaryServices
0x2e9e9000 - 0x2ea08fff EAP8021X armv7s <15c71e8b120b3b6c9c232de6dff4055c> /System/Library/PrivateFrameworks/EAP8021X.framework/EAP8021X
0x2eb12000 - 0x2eb14ffff FTClientServices armv7s <44c007f094e23516ad440dcc4b8448e> /System/Library/PrivateFrameworks/FTClientServices.framework/FTClientServices
0x2eb15000 - 0x2eb44fff FTServices armv7s <5a7e144ef33837359ed59ab12f752df2> /System/Library/PrivateFrameworks/FTServices.framework/FTServices
0x2eb45000 - 0x2ef68fff FaceCore armv7s <9731ccedf231369ba7da37d22b807286> /System/Library/PrivateFrameworks/FaceCore.framework/FaceCore
0x2ef7b000 - 0x2ef86fff FindMyDevice armv7s <67de2557174d3b629f56aad569607cb9> /System/Library/PrivateFrameworks/FindMyDevice.framework/FindMyDevice
0x2efd1000 - 0x2efd1fff FontServices armv7s <68f4057f3bc0315088973b27f3183287> /System/Library/PrivateFrameworks/FontServices.framework/FontServices
0x2efd2000 - 0x2f0a7fff libFontParser.dylib armv7s <63d94b0574793ee9998ce725153a6f54> /System/Library/PrivateFrameworks/FontServices.framework/libFontParser.dylib
0x2f0c6000 - 0x2f0d5fff libGSSFontCache.dylib armv7s <d9ec77b2334a3eacba956a7343890a8a> /System/Library/PrivateFrameworks/FontServices.framework/libGSSFontCache.dylib
0x2f1b9000 - 0x2f1d4fff FrontBoardServices armv7s <6a0538ed555e396bbecb9183314da66a> /System/Library/PrivateFrameworks/FrontBoardServices.framework/FrontBoardServices
0x2fabd000 - 0x2fad3fff GenerationalStorage armv7s <39f3748c838438a3842b9a5ec5b6c62b> /System/Library/PrivateFrameworks/GenerationalStorage.framework/GenerationalStorage
0x2fad4000 - 0x2fd6bfff GeoServices armv7s <d5489b4c67a93e3680f902bc1955cf0> /System/Library/PrivateFrameworks/GeoServices.framework/GeoServices
0x2fd6c000 - 0x2fd7cff GraphicsServices armv7s <c552c842c10431dfba8141dff9beb52> /System/Library/PrivateFrameworks/GraphicsServices.framework/GraphicsServices
0x2fe9e000 - 0x2feedfff Heimdal armv7s <89658112834d3b0b99406319de0ddc69> /System/Library
```

```
y/PrivateFrameworks/Heimdal.framework/Heimdal
0x2ffb6000 - 0x3003afff HomeSharing armv7s <3c8e2e302d0339558b66726ed22436cf> /System/Library/PrivateFrameworks/HomeSharing.framework/HomeSharing
0x30099000 - 0x300f8fff IDS armv7s <c48a45fd39883c57b85320bd7a70668e> /System/Library/PrivateFrameworks/IDS.framework/IDS
0x300f9000 - 0x30124fff IDSFoundation armv7s <ed2d37f9d0bf39818eb7d200b0244dd2> /System/Library/PrivateFrameworks/IDSFoundation.framework/IDSFoundation
0x302d8000 - 0x3033cffff IMFoundation armv7s <a4ac416845d838d4ad7505a4e4c9dca4> /System/Library/PrivateFrameworks/IMFoundation.framework/IMFoundation
0x30344000 - 0x30347ffff IOAccelerator armv7s <a92bf62bbbc837f5959f8d2d0eee3af> /System/Library/PrivateFrameworks/IOAccelerator.framework/IOAccelerator
0x3034a000 - 0x30350ffff IOMobileFramebuffer armv7s <3f879ba3ba5633a1b111433ba49bc775> /System/Library/PrivateFrameworks/IOMobileFramebuffer.framework/IOMobileFramebuffer
0x30351000 - 0x30356ffff IOSurface armv7s <4bf18307118c3bc3a9917d3f773988d3> /System/Library/PrivateFrameworks/IOSurface.framework/IOSurface
0x30357000 - 0x30358ffff IOSurfaceAccelerator armv7s <f11999f87f65300a92f60c505dac2986> /System/Library/PrivateFrameworks/IOSurfaceAccelerator.framework/IOSurfaceAccelerator
0x30359000 - 0x303a0ffff ITMLKit armv7s <4f22c3313c6b336aa562abd42c4a03b1> /System/Library/PrivateFrameworks/ITMLKit.framework/ITMLKit
0x303f4000 - 0x303faffff IntlPreferences armv7s <accfbce79e30359180248894c678cc6d> /System/Library/PrivateFrameworks/IntlPreferences.framework/IntlPreferences
0x303fb000 - 0x30431ffff LanguageModeling armv7s <65b32d31324e3a2daf3fd629a907a389> /System/Library/PrivateFrameworks/LanguageModeling.framework/LanguageModeling
0x304ac000 - 0x304e8ffff MIME armv7s <98740de075f53965a75c73b437df39e5> /System/Library/PrivateFrameworks/MIME.framework/MIME
0x304e9000 - 0x30543ffff MMCS armv7s <bbe274f8019036c2bcfd1422eb2470ba> /System/Library/PrivateFrameworks/MMCS.framework/MMCS
0x30592000 - 0x3059effff MailServices armv7s <910aeb57c97631f6a6b2caeae7ce975a6> /System/Library/PrivateFrameworks/MailServices.framework/MailServices
0x305d1000 - 0x30670ffff ManagedConfiguration armv7s <67a0f6f150623b4e88b16fc847338994> /System/Library/PrivateFrameworks/ManagedConfiguration.framework/ManagedConfiguration
0x3067f000 - 0x30680ffff Marco armv7s <183a524e828a38a1967db1cebf9641c7> /System/Library/PrivateFrameworks/Marco.framework/Marco
0x30681000 - 0x306f9ffff MediaControlSender armv7s <309c78d824963df3a7de696b6232e781> /System/Library/PrivateFrameworks/MediaControlSender.framework/MediaControlSender
0x30794000 - 0x307a8ffff MediaRemote armv7s <bc65fb89e4a73ff0934f0e482147f4ee> /System/Library/PrivateFrameworks/MediaRemote.framework/MediaRemote
0x307a9000 - 0x307bbffff MediaServices armv7s <1b8727bbe1b3a4285f0bfefafa8de9995> /System/Library/PrivateFrameworks/MediaServices.framework/MediaServices
0x307bc000 - 0x307d4ffff MediaStream armv7s <27d170e32e813a3ab2efbb9fcc63d21> /System/Library/PrivateFrameworks/MediaStream.framework/MediaStream
0x30839000 - 0x30916ffff Message armv7s <57ac6626694e34e691ff3cec4692f104> /System/Library/PrivateFrameworks/Message.framework/Message
0x3091c000 - 0x3091effff MessageSupport armv7s <b4d71df6d17f355293c2a1d9ecc45a6b> /System/Library/PrivateFrameworks/MessageSupport.framework/MessageSupport
0x30930000 - 0x3093dffff MobileAsset armv7s <c0713b14b8b4330cb3bc8485a3b4b422> /System/Library/PrivateFrameworks/MobileAsset.framework/MobileAsset
0x30964000 - 0x3096dffff MobileBluetooth armv7s <154a9ed4a113346ebf210ace642cffd3> /System/Library/PrivateFrameworks/MobileBluetooth.framework/MobileBluetooth
0x30989000 - 0x30991ffff MobileIcons armv7s <004226de78993bd6b86fdcdb46623677> /System/Library/PrivateFrameworks/MobileIcons.framework/MobileIcons
0x30992000 - 0x30999ffff MobileInstallation armv7s <4df1bd55e76c3145afcd9b3abbec22ac> /System/Library/PrivateFrameworks/MobileInstallation.framework/MobileInstallation
```

0x3099a000 - 0x309a6fff MobileKeyBag armv7s <08adbb5fd2ef39f1fbfa41f1eb54c1e> /System/Library/PrivateFrameworks/MobileKeyBag.framework/MobileKeyBag
0x309d3000 - 0x309d6fff MobileSystemServices armv7s <ca2adcf15ce834c989c4af2bc7c4ae79> /System/Library/PrivateFrameworks/MobileSystemServices.framework/MobileSystemServices
0x309f8000 - 0x30a05fff MobileWiFi armv7s <36db4443a9913096a55123474eaaa027> /System/Library/PrivateFrameworks/MobileWiFi.framework/MobileWiFi
0x30a44000 - 0x30c13fff MusicLibrary armv7s <074c7ae7112d3dd3a18b4adf8cb1d0f1> /System/Library/PrivateFrameworks/MusicLibrary.framework/MusicLibrary
0x30d91000 - 0x30d9ffff NanoPreferencesSync armv7s <3e3baa16a90c3da0b208c47748c36100> /System/Library/PrivateFrameworks/NanoPreferencesSync.framework/NanoPreferencesSync
0x30da0000 - 0x30db3fff NanoRegistry armv7s <74f4c04e9cc033639828b558f3b1b505> /System/Library/PrivateFrameworks/NanoRegistry.framework/NanoRegistry
0x30dc8000 - 0x30dcffff Netrb armv7s <c7112f327abf3d48aaedc024a5ca2c33> /System/Library/PrivateFrameworks/Netrb.framework/Netrb
0x30dce000 - 0x30dd4fff NetworkStatistics armv7s <457ccadd5a1b3dada1f9236340e33c61> /System/Library/PrivateFrameworks/NetworkStatistics.framework/NetworkStatistics
0x30dd5000 - 0x30df2fff Notes armv7s <14da6174e1113217a97451d5881adcd7> /System/Library/PrivateFrameworks/Notes.framework/Notes
0x30df3000 - 0x30df7fff NotificationsUI armv7s <aedee5a1658b332793115d09573fbcc> /System/Library/PrivateFrameworks/NotificationsUI.framework/NotificationsUI
0x30df8000 - 0x30dfa0ff OAuth armv7s <31e2239fe22c393d81b0a9453fe5fc3> /System/Library/PrivateFrameworks/OAuth.framework/OAuth
0x31555000 - 0x31591fff OpenCL armv7s <880c2f8b8b3e3aa5bb847a5bf5feedcc> /System/Library/PrivateFrameworks/OpenCL.framework/OpenCL
0x3164d000 - 0x316c5fff PassKitCore armv7s <fc863e1eab963588b18cc1c4a8938e24> /System/Library/PrivateFrameworks/PassKitCore.framework/PassKitCore
0x316c6000 - 0x316edfff PersistentConnection armv7s <be24a077478c369fb6a60c13387e6504> /System/Library/PrivateFrameworks/PersistentConnection.framework/PersistentConnection
0x31858000 - 0x31ac4fff PhotoLibraryServices armv7s <3ea4f407a5483844a835853ef7c6e122> /System/Library/PrivateFrameworks/PhotoLibraryServices.framework/PhotoLibraryServices
0x31ac5000 - 0x31acefff PhotosFormats armv7s <3f11a79e1c41381ca136377572994d38> /System/Library/PrivateFrameworks/PhotosFormats.framework/PhotosFormats
0x31acf000 - 0x31b19fff PhysicsKit armv7s <04636f225d533b12ada3ec7c8091af69> /System/Library/PrivateFrameworks/PhysicsKit.framework/PhysicsKit
0x31b1a000 - 0x31b30fff PlugInKit armv7s <2d06163c8f53393ea165c280538e23d7> /System/Library/PrivateFrameworks/PlugInKit.framework/PlugInKit
0x31b31000 - 0x31b38fff PowerLog armv7s <5c785e14cb1e33ed8a77ab49962bbd86> /System/Library/PrivateFrameworks/PowerLog.framework/PowerLog
0x31d3f000 - 0x31dee0ff Preferences armv7s <5a06c9b79248347f8e2685c3a7039852> /System/Library/PrivateFrameworks/Preferences.framework/Preferences
0x31def000 - 0x31e2cff PrintKit armv7s <939d13136b4b3189bbfd9b4cecdbf4a8> /System/Library/PrivateFrameworks/PrintKit.framework/PrintKit
0x31e2d000 - 0x31e30fff ProgressUI armv7s <2fbdf50f0abe3b499f7c0a1a536a4824> /System/Library/PrivateFrameworks/ProgressUI.framework/ProgressUI
0x31e31000 - 0x31ec5fff ProofReader armv7s <325b83e61c11377fb73dc906d1290cc2> /System/Library/PrivateFrameworks/ProofReader.framework/ProofReader
0x31ec6000 - 0x31ed5fff ProtectedCloudStorage armv7s <df6216ff5f543d67ac8f750db7ff2ea2> /System/Library/PrivateFrameworks/ProtectedCloudStorage.framework/ProtectedCloudStorage
0x31ed6000 - 0x31ee2fff ProtocolBuffer armv7s <07b1f39a87a4328998ca0c0e45a99128> /System/Library/PrivateFrameworks/ProtocolBuffer.framework/ProtocolBuffer
0x31ee3000 - 0x31f14fff PrototypeTools armv7s <190a865020a930bd9091a5992fbfb710> /System/Library/PrivateFrameworks/PrototypeTools.framework/PrototypeTools
0x31f17000 - 0x31f86fff Quagga armv7s <243d25491bf73018aa0aa5238fbb6541> /System/Library

```
/PrivateFrameworks/Quagga.framework/Quagga
0x3219c000 - 0x321d8fff RemoteUI armv7s <01cce3fc80e13ed7a88ae57e7d7dd485> /System/Library/PrivateFrameworks/RemoteUI.framework/RemoteUI
0x32274000 - 0x32309fff SAObjects armv7s <45378b9b1ac2318b96c95ea98f8a5399> /System/Library/PrivateFrameworks/SAObjects.framework/SAObjects
0x32316000 - 0x32336fff ScreenReaderCore armv7s <4155dd811eab33e8bd1408c6b6554be8> /System/Library/PrivateFrameworks/ScreenReaderCore.framework/ScreenReaderCore
0x324c9000 - 0x324fbfff SpringBoardFoundation armv7s <a78b5c0ab81331da84dacd9445662cee> /System/Library/PrivateFrameworks/SpringBoardFoundation.framework/SpringBoardFoundation
0x324fc000 - 0x32516fff SpringBoardServices armv7s <67583143a2da3e69885b4f987366ab5a> /System/Library/PrivateFrameworks/SpringBoardServices.framework/SpringBoardServices
0x3252a000 - 0x3254dff SpringBoardUIServices armv7s <b49f8accf605333d937712e42e4e710d> /System/Library/PrivateFrameworks/SpringBoardUIServices.framework/SpringBoardUIServices
0x325bb000 - 0x32887fff StoreKitUI armv7s <b6920a27e76c3b0a9f9318bfaf6452d2> /System/Library/PrivateFrameworks/StoreKitUI.framework/StoreKitUI
0x32888000 - 0x329b4fff StoreServices armv7s <8a5a058ba2003680b4bfc8f34a061adf> /System/Library/PrivateFrameworks/StoreServices.framework/StoreServices
0x32a85000 - 0x32a87fff TCC armv7s <80a05e480ea53d26a4f31123beed8afb> /System/Library/PrivateFrameworks/TCC.framework/TCC
0x32a95000 - 0x32adaff TelephonyUI armv7s <ec6a1019e9ae36b297c9bbabce4bb459> /System/Library/PrivateFrameworks/TelephonyUI.framework/TelephonyUI
0x32adb000 - 0x32b20fff TelephonyUtilities armv7s <997799acc591303b94e3802ed95a1343> /System/Library/PrivateFrameworks/TelephonyUtilities.framework/TelephonyUtilities
0x336e4000 - 0x3370cff TextInput armv7s <c1ca800641f335a19e817ed327e38246> /System/Library/PrivateFrameworks/TextInput.framework/TextInput
0x3370d000 - 0x3371ffff TextToSpeech armv7s <8208dd231f8e3e8cb75d812b944f6173> /System/Library/PrivateFrameworks/TextToSpeech.framework/TextToSpeech
0x33795000 - 0x337c8fff UIAccessibility armv7s <a5b7dfecdff535eeb2e5c1cfb4193536> /System/Library/PrivateFrameworks/UIAccessibility.framework/UIAccessibility
0x337c9000 - 0x3388bfff UIFoundation armv7s <f26ac9e2df5b3b0ca86df5b79f3bd7f8> /System/Library/PrivateFrameworks/UIFoundation.framework/UIFoundation
0x338b2000 - 0x338b5fff UserFS armv7s <23943f5513333bdfa65297f2f08e55> /System/Library/PrivateFrameworks/UserFS.framework/UserFS
0x338ce000 - 0x33e08fff VectorKit armv7s <f1f2d6a1634e36909227aedea517c798> /System/Library/PrivateFrameworks/VectorKit.framework/VectorKit
0x3402e000 - 0x3404cff VoiceServices armv7s <1fb46eee95e33905868312efa8070869> /System/Library/PrivateFrameworks/VoiceServices.framework/VoiceServices
0x340e5000 - 0x3410bfff WebBookmarks armv7s <e5f0f8d22b673f448c28cc04817e21d6> /System/Library/PrivateFrameworks/WebBookmarks.framework/WebBookmarks
0x34121000 - 0x34ca1fff WebCore armv7s <e9f5aaac26533d58845011802e906140> /System/Library/PrivateFrameworks/WebCore.framework/WebCore
0x34ca2000 - 0x34d60fff WebKitLegacy armv7s <6d126fab646433b5b70e53fca3bc3909> /System/Library/PrivateFrameworks/WebKitLegacy.framework/WebKitLegacy
0x34f0a000 - 0x34f10fff XPCKit armv7s <43d8bfffab44734feacc683811255e619> /System/Library/PrivateFrameworks/XPCKit.framework/XPCKit
0x34f11000 - 0x34f19fff XPCObjects armv7s <3db1acbcfc593ad982187ae0af01d993> /System/Library/PrivateFrameworks/XPCObjects.framework/XPCObjects
0x35104000 - 0x35128fff iCalendar armv7s <126f92193371363ea4ca89295c5465ae> /System/Library/PrivateFrameworks/iCalendar.framework/iCalendar
0x35148000 - 0x35183fff iTunesStore armv7s <e37d5437eb8c3ee8ba8fe110e5df39cf> /System/Library/PrivateFrameworks/iTunesStore.framework/iTunesStore
0x35184000 - 0x35305fff iTunesStoreUI armv7s <081c3e6e4b9d3356a9dad2cfe8a501ad> /System/Library/PrivateFrameworks/iTunesStoreUI.framework/iTunesStoreUI
```

```
0x359d8000 - 0x359d9fff libAXSafeCategoryBundle.dylib armv7s <db0e24b4d9ad3847b99b1165d3  
697c1a> /usr/lib/libAXSafeCategoryBundle.dylib  
0x359da000 - 0x359e0fff libAXSpeechManager.dylib armv7s <d705ce6b23263295bd2e3680effe923  
a> /usr/lib/libAXSpeechManager.dylib  
0x359e1000 - 0x359e9fff libAccessibility.dylib armv7s <6e34291faad43781a17464523142c7a1>  
/usr/lib/libAccessibility.dylib  
0x35c3d000 - 0x35c53fff libCRFSuite.dylib armv7s <81bb6391c1643d038ea597f57762d71f> /usr  
/lib/libCRFSuite.dylib  
0x35c86000 - 0x35d8afff libFosl_dynamic.dylib armv7s <2fc617cd0aee379988c3863964aaf1ad>  
/usr/lib/libFosl_dynamic.dylib  
0x35da4000 - 0x35dbffff libMobileGestalt.dylib armv7s <9ba6174d5d1b31698d70c4855b6078da>  
/usr/lib/libMobileGestalt.dylib  
0x35dbc000 - 0x35dc4fff libMobileGestaltExtensions.dylib armv7s <30877c65f1f43ffd895c7dd  
86dd929e4> /usr/lib/libMobileGestaltExtensions.dylib  
0x35de1000 - 0x35de2fff libSystem.B.dylib armv7s <02104fb10f13d22a1cd5bf04ba0c10e> /usr  
/lib/libSystem.B.dylib  
0x35e53000 - 0x35e97fff libTelephonyUtilDynamic.dylib armv7s <7788f77375dd3756a934364b51  
2c4137> /usr/lib/libTelephonyUtilDynamic.dylib  
0x35fa7000 - 0x35fc9fff libarchive.2.dylib armv7s <3e3a22a4d4603e20967a04898efb99ba> /us  
r/lib/libarchive.2.dylib  
0x35fca000 - 0x35fcffff libassertion_extension.dylib armv7s <362ff6610704394bb58ab4c1492  
fb307> /usr/lib/libassertion_extension.dylib  
0x35ff9000 - 0x36005fff libbsm.0.dylib armv7s <ab7814c202aa3a978d5fc4066d507ded> /usr/li  
b/libbsm.0.dylib  
0x36006000 - 0x3600ffff libbz2.1.0.dylib armv7s <b174e42e490732959f042109e83b4068> /usr/  
lib/libbz2.1.0.dylib  
0x36010000 - 0x3605afff libc++.1.dylib armv7s <85cef3e4e8453efda7a44d09a528ca73> /usr/li  
b/libc++.1.dylib  
0x3605b000 - 0x36076fff libc++abi.dylib armv7s <f854f7b9943835089d64cdb166fb7944> /usr/l  
ib/libc++abi.dylib  
0x36078000 - 0x36085fff libcmph.dylib armv7s <006a8f6104073fd59f65f0ee76b7e8f7> /usr/lib  
/libcmph.dylib  
0x36086000 - 0x3608efff libcupolicy.dylib armv7s <1536179fc38339e3a413d4ed3e835285> /usr  
/lib/libcupolicy.dylib  
0x360b5000 - 0x360cefff libextension.dylib armv7s <db293f05a78935cb9b9d576d4acef7c1> /us  
r/lib/libextension.dylib  
0x360ed000 - 0x360f0fff libheimdal-asn1.dylib armv7s <70ce5b437d51333e831f09b6d70996db>  
/usr/lib/libheimdal-asn1.dylib  
0x360f1000 - 0x361defff libiconv.2.dylib armv7s <579ad521cf163395bf9b360b8700b569> /usr/  
lib/libiconv.2.dylib  
0x361df000 - 0x3634dfff libicucore.A.dylib armv7s <b57df05026d73ec0aac4a815217f6185> /us  
r/lib/libicucore.A.dylib  
0x3635a000 - 0x3635afff liblangid.dylib armv7s <1c7a4628d4f93f57b0e9122aad9a41bc> /usr/l  
ib/liblangid.dylib  
0x3635b000 - 0x36365fff liblockdown.dylib armv7s <6bcacfdd347535f8aca89e45d19be50e> /usr/  
lib/liblockdown.dylib  
0x36366000 - 0x3637bfff liblzma.5.dylib armv7s <d503bc52f6ab336a951c7498fb2818c6> /usr/l  
ib/liblzma.5.dylib  
0x366f5000 - 0x3670afff libmis.dylib armv7s <d76f316f54883fb0a44fe4178069d126> /usr/lib  
/libmis.dylib  
0x36733000 - 0x3692dff libobjc.A.dylib armv7s <8529d172d0d4376295751bc1657c184d> /usr/l  
ib/libobjc.A.dylib  
0x369e2000 - 0x369f8fff libresolv.9.dylib armv7s <db3fa6e2c18b38b38a25828c2eb2eb94> /usr
```

```
/lib/libresolv.9.dylib
0x36a23000 - 0x36ad4fff libsqlite3.dylib armv7s <c71c02e14d9e3f20b1e54eef5c0bfeec> /usr/
lib/libsqlite3.dylib
0x36b22000 - 0x36b48fff libtidy.A.dylib armv7s <97fd6bd77752374fb766bbbc31d83b6f> /usr/l
ib/libtidy.A.dylib
0x36b49000 - 0x36b51fff libtzupdate.dylib armv7s <e2906e95adfe33b48c7a9314c73fc515> /usr/
lib/libtzupdate.dylib
0x36b55000 - 0x36c0bfff libxml2.2.dylib armv7s <fad7ce69d3b03c2f97f39dc70fd1b35e> /usr/l
ib/libxml2.2.dylib
0x36c0c000 - 0x36c2dff libxslt.1.dylib armv7s <eed3c48ecd8c3f688357c7e9634c28a5> /usr/l
ib/libxslt.1.dylib
0x36c2e000 - 0x36c3aff libz.1.dylib armv7s <9e6cb733a633391eaf5eecc9b9ab4d4> /usr/lib/
libz.1.dylib
0x36c3b000 - 0x36c3ffff libcache.dylib armv7s <464e89321abb31a29f42b476cd75786d> /usr/li
b/system/libcache.dylib
0x36c40000 - 0x36c49fff libcommonCrypto.dylib armv7s <0c2911b8dc653852913d4402f021a6c6>
/usr/lib/system/libcommonCrypto.dylib
0x36c4a000 - 0x36c4efff libcompiler_rt.dylib armv7s <2e4a01eff3dd333b99ab14b54580aecd> /
usr/lib/system/libcompiler_rt.dylib
0x36c4f000 - 0x36c55fff libcopyfile.dylib armv7s <d365247190fe3bafbadb0e3a35b988fd> /usr/
lib/system/libcopyfile.dylib
0x36c56000 - 0x36ca2fff libcorecrypto.dylib armv7s <e93edd093a633c40af2fcf9de68a813b> /u
sr/lib/system/libcorecrypto.dylib
0x36ca3000 - 0x36cc3fff libdispatch.dylib armv7s <d047f787bfc33eb08eaf3a34f82a1eb5> /usr/
lib/system/libdispatch.dylib
0x36cc4000 - 0x36cc5fff libdyld.dylib armv7s <d76438e6086a38eb8d9e38b7d3a4fedd> /usr/lib
/system/libdyld.dylib
0x36cc6000 - 0x36cc6fff libkeymgr.dylib armv7s <d961518f774a38ce80d87302f5f2d270> /usr/l
ib/system/libkeymgr.dylib
0x36cc7000 - 0x36cc7fff liblaunch.dylib armv7s <94848ea6b094336d893c38753795f5ac> /usr/l
ib/system/liblaunch.dylib
0x36cc8000 - 0x36ccbfff libmacho.dylib armv7s <a7e04a25a2143498be67845bc23fff4a> /usr/li
b/system/libmacho.dylib
0x36ccc000 - 0x36ccdfbf libremovefile.dylib armv7s <99984750ecaf3483976db606e4a6d763> /u
sr/lib/system/libremovefile.dylib
0x36cce000 - 0x36cdffff libsystem_asl.dylib armv7s <983391f3b4763ae59c7730f0b80dbb0b> /u
sr/lib/system/libsystem_asl.dylib
0x36ce0000 - 0x36ce0fff libsystem_blocks.dylib armv7s <9024447e02ef3270b46771adc93133c9>
/usr/lib/system/libsystem_blocks.dylib
0x36ce1000 - 0x36d44fff libsystem_c.dylib armv7s <a84f366484fd3d21b3b0ee60ac01ac59> /usr/
lib/system/libsystem_c.dylib
0x36d45000 - 0x36d47fff libsystem_configuration.dylib armv7s <80e2c84c178837c5b91e591f6a
4d7201> /usr/lib/system/libsystem_configuration.dylib
0x36d48000 - 0x36d49fff libsystem_coreservices.dylib armv7s <9290fc392177307faa8403f9aa8
9553f> /usr/lib/system/libsystem_coreservices.dylib
0x36d4a000 - 0x36d56fff libsystem_coretls.dylib armv7s <6a0f500cccaf3e848179ce8bae2a0880
> /usr/lib/system/libsystem_coretls.dylib
0x36d57000 - 0x36d5dfff libsystem_dnssd.dylib armv7s <12b8450299ab3104bf59e0e4852c63c4>
/usr/lib/system/libsystem_dnssd.dylib
0x36d5e000 - 0x36d7afff libsystem_info.dylib armv7s <f37a73a834bb355ca035e82f81de2d26> /
usr/lib/system/libsystem_info.dylib
0x36d7b000 - 0x36d95fff libsystem_kernel.dylib armv7s <31ae14feacbd345cb0a6f3041834e3ea>
/usr/lib/system/libsystem_kernel.dylib
```

```
0x36d96000 - 0x36db5fff libsystem_m.dylib armv7s <0c284ff2f4d130299a6cf38fe5ff8303> /usr/lib/system/libsystem_m.dylib
0x36db6000 - 0x36dc8fff libsystem_malloc.dylib armv7s <5c198ca70fa7378da137cd816decbbf8> /usr/lib/system/libsystem_malloc.dylib
0x36dc9000 - 0x36df6fff libsystem_network.dylib armv7s <c1e9979a8c973feeb4acbce97e80508a> /usr/lib/system/libsystem_network.dylib
0x36df7000 - 0x36dfcff libsystem_networkextension.dylib armv7s <6d884e2846e83abf8208027e115dddea> /usr/lib/system/libsystem_networkextension.dylib
0x36dfd000 - 0x36e04fff libsystem_notify.dylib armv7s <787d39efb46b3b4d94fb3cc95b618779> /usr/lib/system/libsystem_notify.dylib
0x36e05000 - 0x36e09fff libsystem_platform.dylib armv7s <badd9ef3da483107966fde4120b12fe2> /usr/lib/system/libsystem_platform.dylib
0x36e0a000 - 0x36e10fff libsystem_pthread.dylib armv7s <2d3aa96e56a73fcdb25f3e0a536e6eeb> /usr/lib/system/libsystem_pthread.dylib
0x36e11000 - 0x36e13fff libsystem_sandbox.dylib armv7s <8cb953de567c3fbcad59887f5afbfb41> /usr/lib/system/libsystem_sandbox.dylib
0x36e14000 - 0x36e17fff libsystem_stats.dylib armv7s <d882f76e1481348aaeae6fb274046cee> /usr/lib/system/libsystem_stats.dylib
0x36e18000 - 0x36e1dff libsystem_trace.dylib armv7s <4bc82d65e90b3ecbb8317222946ff47f> /usr/lib/system/libsystem_trace.dylib
0x36e1e000 - 0x36e1eff libunwind.dylib armv7s <b95c6662d06a3a9988fe6265a26ddc42> /usr/lib/system/libunwind.dylib
0x36e1f000 - 0x36e3aff libxpc.dylib armv7s <0db6e5db4d633912804338bee3b1ea27> /usr/lib/system/libxpc.dylib
```

相關閱讀

- [Technical Note TN2151: Understanding and Analyzing iOS Application Crash Reports](#)
- [WWDC 2010 Understanding Crash Reports on iPhone OS](#)
- [Analyzing Crash Reports](#)
- [Crashes Organizer Help](#)
- [Share diagnostics and usage information with Apple in iOS](#)
- [Handling unhandled exceptions and signals](#)

練習：閱讀 Crash Reports

- 打開 Xcode，接上你的 iOS 裝置
- 把你的裝置上的所有 crash report 都拿出來
- 一篇一篇解釋你從 crash report 中看到什麼，並且推測應該如何修正問題

Core Animation

Core Animation 是在 iOS 上負責一般 UI 繪圖與動畫的 framework。當你想要讓你的 app 有更豐富的動畫效果，但是你並不打算做套 3D 遊戲、還沒打算使用 OpenGL 或 Metal 這些 3D 繪圖的 framework 的話，你就應該先看一下 Core Animation。掌握了 Core Animation 之後，絕大多數 app 裡頭可能用到的動畫，你都應該可以做出來。

真的要詳細介紹 Core Animation，其實可以寫成一本專書，我們在這邊盡量簡短介紹。Core Animation 大概分成三種主要的 class：

- CALayer
- CAAnimation
- CATransaction

我們不妨把 CALayer 想像成是演員，而 CAAnimation 則是劇本，各種動畫效果，就是讓演員按照劇本演出。至於 CATransaction 則是整體的劇場設定，比方說，我們今天要演一齣戲，這場戲原本要演兩個小時，但我們希望這齣戲可以一個小時演完，那麼，這種「整齣戲演出的速度」這樣的設定，就是 CATransaction 的功能。

CALayer

在 iOS 上如果我們要開始使用 Core Animation，會從某個 view 上面的 CALayer 開始著手。Mac 上的 NSView 預設是沒有 CALayer 的，如果想在 Mac 上使用 Core Animation，我們需要先把 NSView 的 `wantsLayer` 設定成 YES，然後自己建立一個 CALayer 物件，設成 NSView 的 layer property。

我們在這邊先講 iOS 上 UIView 裡頭的 CALayer。

CALayer 與 UIView 的關係

任何一個 view 都是兩種性質組成的，就是可以操作，以及可以被看到。一個 view 可以被操作，像是在 iOS 上按鈕可以收到觸控事件、在 Mac 上可以被滑鼠點選，這是一個 view 作為 responder 的部份。

至於一個 view 呈現出來的外觀，不同於 Mac 上 NSView 本身具有繪圖的責任，在 iOS 上，UIView 的外觀呈現，都是由 Core Animation 實作。我們看到的 UIView 的樣子，其實是裡頭的 CALayer 的樣子。

在 iOS 上，我們可以使用一系列的 UIView 的 class method 產生動畫，像是呼叫

`+animateWithDuration:delay:options:animations:completion:` 這一系列 method，其實並不是 View 本身產生動畫，而是 View 裡頭的 CALayer 物件的功勞。或這麼說：其實 UIView 裡頭的 CALayer 本身就具有產生動畫的能力，而改動 CALayer 的任何屬性，都會產生 0.25 秒的動畫，只是我們平常在設定 UIView 的時候，UIKit 的設計是刻意把動畫關閉了。

我們在前面幾節的練習中，像我們在寫貪食蛇這個練習的時候，知道如果要改變一個 view 的外觀，可以透過在 UIView 的 subclass 中 override 掉 `drawRect:` 達成。—UIView 的外觀難道不是由 `drawRect:` 的實作決定的嗎？跟 CALayer 有什麼關係呢？

`drawRect:` 其實是個 delegate call，用途不是繪製 view，而是繪製 CALayer 的內容。

每個 UIView，都是屬於自己專屬的 CALayer 物件的 delegate，當我們要重繪某個 view 的內容時，其實是叫 CALayer 重繪，重繪 CALayer 時會呼叫到 CALayer 的 `-drawInContext:`，在這個 method 中，CALayer 可以自己決定怎麼繪製內容，或是去問 delegate 該怎麼畫，而去問了 CALayer delegate 的 `-drawLayer:inContext:`。

在 UIView 的 `-drawLayer:inContext:` 的實作中，便呼叫了 `drawRect:`，因此，`drawRect:` 繪製的內容，放到 CALayer 上。

所以我們可以知道幾件事情：雖然要設定一個 layer 的內容，最簡單的方法是直接對 layer 設定 contents。contents 的型別是 id，不過在 iOS 上我們會設成 CGImageRef（在 Mac 上則要設成 NSImage 物件）；CALayer 有很多的 subclass，我們也可以按照這些 subclass 的要求設定。不過，如果我們自己 subclass 了 CALayer 的話，也可以自己 override `-drawLayer:inContext:`。

另外，CALayer 並不是 responder。於是當我們遇到了很複雜的畫面，畫面中有許多不同的元素要一直變化、移動的時候，我們不妨考慮讓 app 中出現許多 layer，而不是 view，因為每多一個 view，就會在 responder chain 當中出現一個 responder，因此會影響每一輪 run loop 的速度。一個 layer 上面可以繼續增加 layer，就像 view 可以呼叫 `addSubview:` 一樣，CALayer 也有對應的 `addSublayer:`。

最重要的是，如果你自己建立了 CALayer 物件，千萬不要把 delegate 指到某個 UIView 上，UIView 已經是自己的 layer 的 delegate 了！

畫面截圖

在 iOS 7 之前，我們往往會利用 view 的內容是由 layer 繪製這點，產生某個 view 截圖。我們只要要求某個 view 的 layer 再對某個 graphic context 畫一次圖，然後把繪製的內容放進一個 UIImage 物件即可。

```
@implementation UIView(MyExtensions)
- (UIImage *)imageOfCurrentContent
{
    UIGraphicsBeginImageContextWithOptions(self.bounds.size, YES, [UIScreen mainScreen].scale);
    [self.layer renderInContext:UIGraphicsGetCurrentContext()];
    UIImage *viewImage = UIGraphicsGetImageFromCurrentImageContext();
    UIGraphicsEndImageContext();
    return viewImage;
}
@end
```

產生畫面的截圖這點在製作很多 UI 時非常有用，像我們想要某個 view A 變形、然後另外一個 view B 出現這樣的轉場效果，中間的變化往往不是對 A 與 B 這兩張圖片做操作，而是對 A 與 B 的畫面截圖做影像相關的處理，比起直接改變兩個 view，讓 view 不斷重繪，處理兩張圖片的速度會快上許多。製作 UI 時經常需要許多的障眼法。

在 iOS 7 之後我們通常不會用這種方式產生畫面截圖，原因是 iOS 7 之後 UIView 有新的 API `drawViewHierarchyInRect:afterScreenUpdates:` 可以使用。新的 API 與透過 CALayer 繪圖的差別是，iOS 7 之後的 UI 設計大量使用半透明毛玻璃效果的 view，而用 CALayer 截出的圖片無法抓到這部份，而新的 API 就蘋果的說法，可以抓到無論是 UIKit、Quartz、OpenGL ES、SpriteKit 等這種繪圖系統產生的畫面。

相關說明請參見蘋果官方文件：[Technical Q&A QA1817 View Snapshots on iOS 7](#)

設定 CALayer 的基本樣式與屬性

CALayer 有許多的屬性與 UIView 相同，像是 frame、bounds，很多時候可以把 CALayer 想像成是 UIView 使用；不過，CALayer 有一些 UIView 所沒有的屬性可以設定，只要對每個 view 的 layer 設定這些屬性，就可以改變原本只靠 UIView 的屬性沒有辦法達到的視覺效果。

比方說，UIView 沒有跟邊框相關的屬性，但是 CALayer 則可以設定

`cornerRadius`、`borderWidth`、`borderColor`，我們可以很快就設定好 layer 的邊框寬度與顏色，甚至可以設定圓角效果。

另外，UIView 也沒有陰影可以設定，但是對 CALayer 則可以設定

`shadowOpacity`、`shadowRadius`、`shadowOffset` 與 `shadowColor`，用來設定陰影的顏色、位置、尺寸大小與透明度等。不過，我們會建議另外設定 `shadowPath`，用一個 CGPath 描述陰影的外框範圍，設上去之後的效能會快許多。

前面提到，絕大多數的 CALayer 的屬性在設定之後，會產生 0.25 的動畫效果，這種因為改變屬性而產生的動畫，蘋果的術語叫做 Implicit Animations；打開 CALayer.h，只要看到註解裡頭提到某個屬性是屬於 Animatable，就是會產生動畫效果的屬性。

因為改變任何屬性都會產生動畫，所以，當我們建立了 layer 之後，通常會先設好 frame，才把 layer 加到 super layer 上，不然，如果先加到 super layer，才去改變 frame，就會產生很奇怪的動畫效果。

CALayer 在建立完畢之後，預設都是一倍解析度，所以在 Retina Display 的裝置上，看起來都會糊糊的（尤其是使用 CATextLayer 這個用來顯示文字內容的 layer，更容易凸顯解析度的不足），所以需要告訴 CALayer 應該要用怎樣的解析度，方法是透過設定 `contentsScale` 屬性。在 iOS 上，我們通常設成 UIScreen 的 scale。

```
layer.contentsScale = [UIScreen mainScreen].scale;
```

在 Mac 上這件事情會變得比較複雜，我們在後面會說明。

實作 CALayer drawInContext: 需要注意的地方

如果我們打算自己實作 `drawInContext:`，就需要注意一下，我們平常在UIKit 用到的許多跟繪圖相關的功能，都是對 Core Graphics 的 current context 操作（也就是呼叫

`UIGraphicsGetImageFromCurrentImageContext()` 回傳的 `CGContextRef`），但是在實作 `drawInContext:` 的時候，是要把圖片繪製到指定的 context 裡頭；所以我們要先把傳入了 context 變成 current context。

要把某個 context 變成 current context，只要呼叫 `UIGraphicsPushContext()` 即可，不過，當我們離開 `drawInContext:` 的時候，要記得呼叫 `UIGraphicsPopContext()`，還原到原本的設定。

```
- (void)drawInContext:(CGContextRef)ctx
{
    UIGraphicsPushContext(ctx);
    // Your drawing code here.
    UIGraphicsPopContext();
}
```

了解 CALayer 之後，下一步就是要讓 CALayer 動起來。我們在下一節要討論的就是讓 CALayer 移動的 CAAnimation。而 CALayer 其實有許多 subclass，我們在後面也會繼續討論一些常用 CALayer subclass 的功能。

CATransaction

CATransaction 在 Core Animation framework 中主要扮演了「整體舞台設定」的角色。當我們在改變某個 CALayer 的 animatable 的屬性的時候，預設都會產生 0.25 秒的動畫，但很多時候我們想要改變這個動畫的時間長度，或是，當我們剛建立某些 layer，要把這些 layer 放上畫面的時候，我們還不希望改動這些 layer 時會產生動畫；我們這時候就會使用 CATransaction。

使用 CATransaction 的方式就是把我們想做特別設定的動畫 code，用 CATransaction 的 class method 前後包起來。比方說，我們現在希望不要產生動畫，便可以這麼寫：

```
[CATransaction begin];
[CATransaction setDisableActions:YES];
// 原本的動畫 code
[CATransaction commit];
```

[CATransaction setDisableActions:YES] 這行也可以寫成 [CATransaction setValue:@(YES) forKey:kCATransactionDisableActions]，意思是一樣的。

至於我們想要改變所有 property 動畫的時間，也是一樣的作法。比方說，我們想把 0.25 秒的動畫改成三秒：

```
[CATransaction begin];
[CATransaction setAnimationDuration:1.0];
[CATransaction commit];
```

在使用 CATransaction 的時候，我們應該避免巢狀呼叫，例如下面這個例子：

```
[CATransaction begin];
[CATransaction setDisableActions:NO];
// 一些動畫 code

[CATransaction begin];
[CATransaction setDisableActions:YES];
// 原本的動畫 code
[CATransaction commit];

[CATransaction commit];
```

在我們的經驗中，這樣寫會讓畫面產生非常不自然的閃爍。

CAAnimation

要讓一個 CALayer 動起來，除了透過改變 layer 的 animatable 的屬性外，就是建立 CAAnimation 物件，然後對 layer 呼叫 `-addAnimation:forKey:`。我們通常會選擇一種 CALayer 的 subclass，製作我們想要的動畫效果。

CATransition

我們先從 Core Animation 內建的轉場效果 CATransition 講起（請不要跟 CATransaction 搞混），因為 CATransition 大概是最容易上手，而且可以馬上有成就感的動畫。CATransition 通常用在兩個 view 之間的切換；UIView 本身就有定義一些跟轉場有幾個以 transition 開頭的 class method，像是

- `+ transitionWithView:duration:options:animations:completion:`
- `+ transitionFromView:toView:duration:options:completion:`

而這些 method 可以設定的 transition option 包括：

- `UIViewControllerAnimatedOptionTransitionNone`
- `UIViewControllerAnimatedOptionTransitionFlipFromLeft`
- `UIViewControllerAnimatedOptionTransitionFlipFromRight`
- `UIViewControllerAnimatedOptionTransitionCurlUp`
- `UIViewControllerAnimatedOptionTransitionCurlDown`
- `UIViewControllerAnimatedOptionTransitionCrossDissolve`
- `UIViewControllerAnimatedOptionTransitionFlipFromTop`
- `UIViewControllerAnimatedOptionTransitionFlipFromBottom`

也就是說，UIView 本身就已經提供了四種方向的翻轉（上下左右）、淡入淡出與翻頁幾種轉場效果。其實這些效果都是用 Core Animation 實作的，而 CATransition 所提供的效果也遠比這些多，只要我們知道怎樣使用 CATransition，就可以使用更多的轉場效果。

我們來寫一個簡單的 view controller：這個 view controller 的 view 上面我們額外建立一個 CALayer，然後加了一個按鈕，這個按鈕按下去之後的 action 是讓這個 layer 在紅色與綠色之間切換。

ViewController.h

```
@import UIKit;
#import QuartzCore;

@interface ViewController : UIViewController
- (IBAction)toggleColor:(id)sender;
@end
```

ViewController.m

```
#import "ViewController.h"

@interface ViewController ()
@property (strong, nonatomic) CALayer *aLayer;
@property (assign, nonatomic) BOOL isGreen;
```



```
@end

@implementation ViewController
- (void)viewDidLoad
{
    [super viewDidLoad];
    self.aLayer = [[CALayer alloc] init];
    self.aLayer.frame = CGRectMake(50, 50, 100, 100);
    self.aLayer.backgroundColor = [UIColor redColor].CGColor;
    [self.view.layer addSublayer:self.aLayer];
}

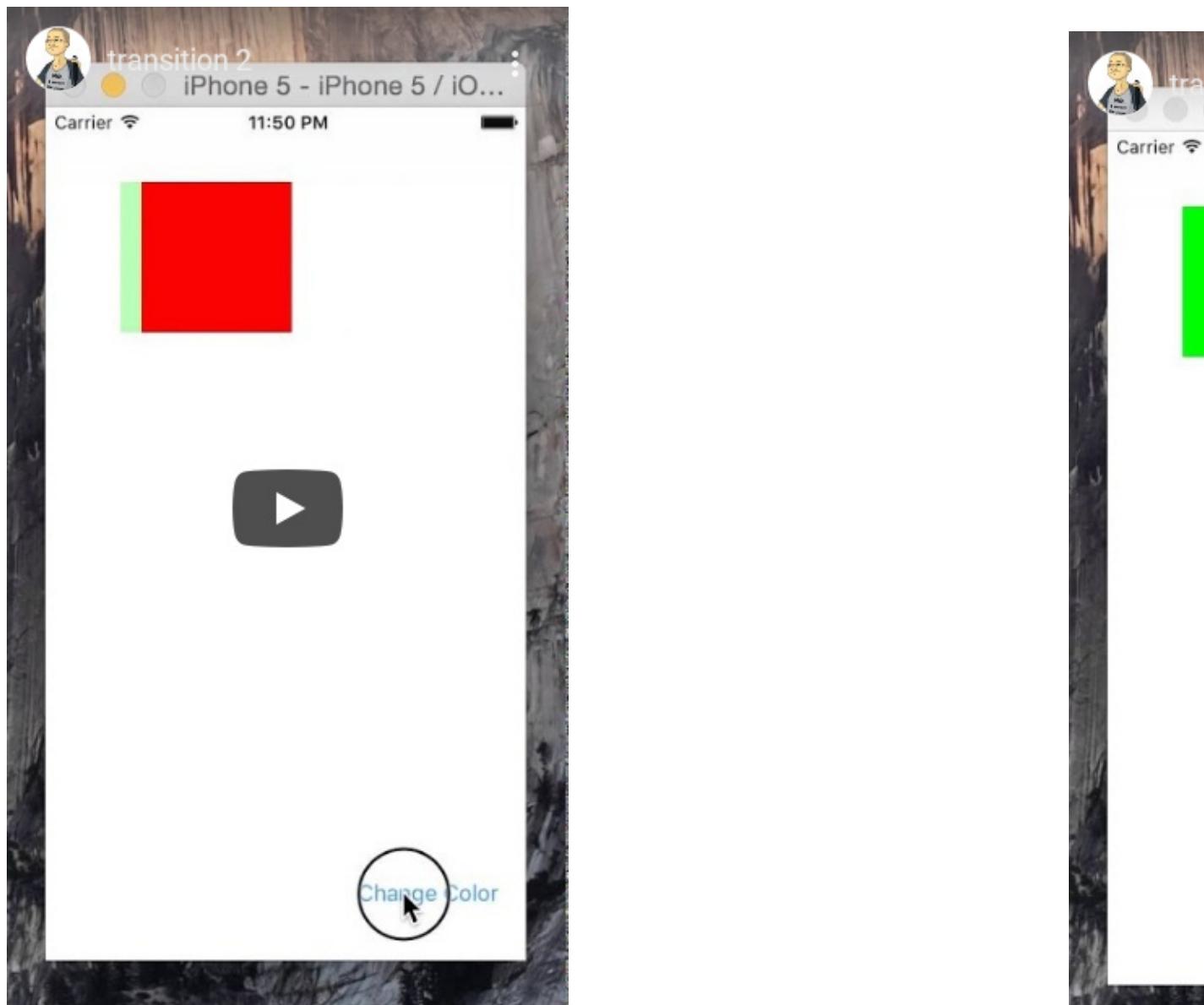
- (IBAction)toggleColor:(id)sender
{
    self.isGreen = !self.isGreen;
    self.aLayer.backgroundColor = self.isGreen ? [UIColor greenColor].CGColor : [UIColor redColor].CGColor;
}
@end
```



Carrier

原本這個版本只會產生 0.25 秒的屬性變化，使用的是淡入淡出效果。我們來加入 `kCATransitionMoveIn` 這個轉場效果看看，就會變成新的畫面從上方推入。

```
- (IBAction)toggleColor:(id)sender
{
    self.isGreen = !self.isGreen;
    self.aLayer.backgroundColor = self.isGreen ? [UIColor greenColor].CGColor : [UIColor redColor].CGColor;
    CATransition *transition = [[CATransition alloc] init];
    transition.type = kCATransitionMoveIn;
    transition.subtype = kCATransitionFromRight;
    [self.aLayer addAnimation:transition forKey:@"transition"];
}
```



CATransition 可以應用的地方不只是切換背景顏色而已，無論是改變這個 layer 的任何 animatable 的屬性，像是改變 contents，或是在這個 layer 中增加或移除 sublayer，用 `addAnimation:forKey:` 加入了 CATransition 物件之後，都可以產生轉場效果。

如果我們想要像 UIView 的 `+transitionWithView:duration:options:animations:completion:` 那樣，是在某個 view 上面增加或移除 subview 的時候產生轉場效果，用 CATransition 的作法也差不多，我們先寫一好要某個 view A 新增或移除 subview，然後把 CATransition 加到 view A 的 layer 上即可。

蘋果在 iOS 上的 CATransition 的 public header 中定義了幾個 type：

- `kCATransitionFade` : 淡入淡出動畫
- `kCATransitionMoveIn` : 新的畫面從上方移入的效果
- `kCATransitionPush` : 推擠效果，很像 navigation controller 的換頁
- `kCATransitionReveal` : 原本的畫面從上方移出的效果

不過...其實蘋果有不少 private API 可以用。比方說，我們可以把 type 指定成立體塊狀翻轉效果。

```
transition.type = @"cube";
```



在 <http://iphonedevwiki.net/index.php/CATransition> 這一頁上可以看到整理好的 private API 可以呼叫。雖然蘋果的政策是禁止在上架的 app 中呼叫 private API，如果用了可能會被 reject...不過印象中其實有很多 app 都用到這些 private API。



如果對內建的這些效果，甚至 private 的效果都不滿意的話，iOS 5 之後，CATransition 還有一個叫做 filter 的屬性，我們可以在這個屬性上加上 CIFilter 物件，客製更多的轉場效果。

另外要注意，在我們呼叫 `addAnimation:forKey:` 的時候，如果加入的是個 CATransition 動畫，無論使用了怎樣的名稱當做 key，key 都會是 transition。

CAPropertyAnimation

CAPropertyAnimation 便是透過設定某個 CALayer 的屬性產生動畫。前面提到，只要修改 layer 的 animatable 屬性會自定產生動畫效果，不過，跟使用 CAPropertyAnimation 的狀況不太一樣，我們對某個 layer 加入了 CAPropertyAnimation 之後，雖然會產生動畫，但是就只有產生動畫而已，layer 屬性原本的值並不會因此改變，用蘋果的術語，這種動畫叫做 Explicit Animations。

CAPropertyAnimation 是一層介面，我們通常使用的是 CAPropertyAnimation 的 subclass CABasicAnimation。設定 CABasicAnimation 的時候，主要會設定以下屬性：

1. `fromValue` : 要讓某個屬性產生變化的動畫時的初始值
2. `toValue` : 要讓某個屬性產生變化的動畫時結束的數值
3. `byValue` : 要讓某個屬性產生變化的動畫時，介於開始與結束的中間值，但是很多時候可以不用特別設定，設成 `nil` 即可

然後有一些設定是定義在 CAMediaTiming protocol 中，像是：

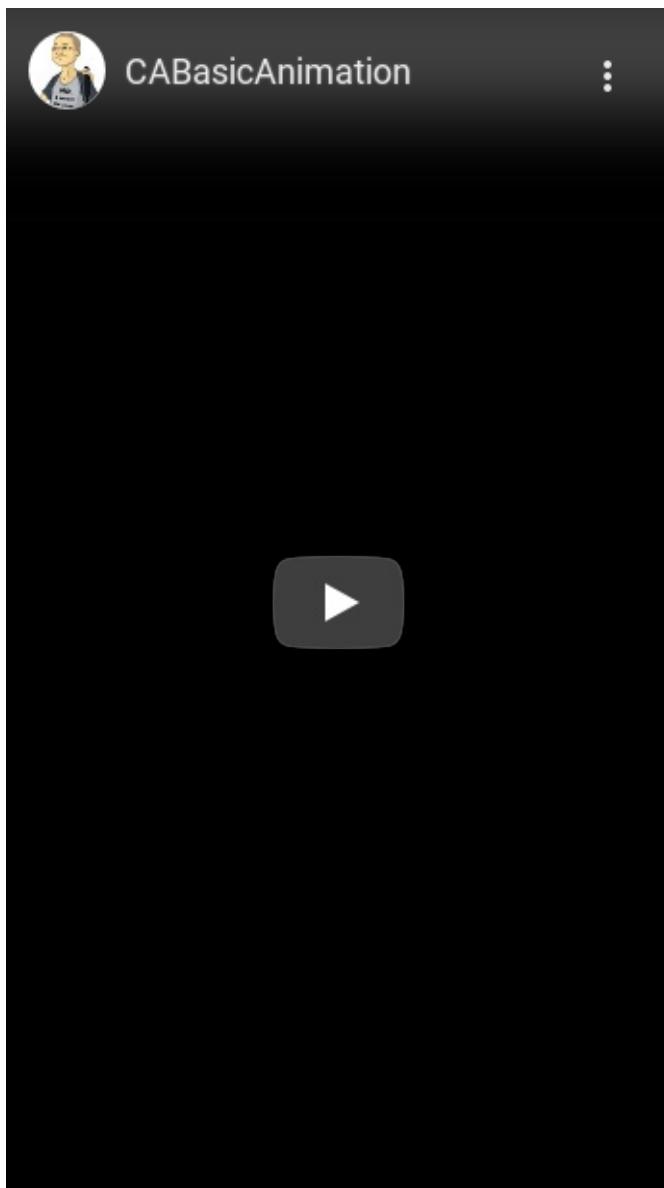
1. `duration` : 這個動畫要花上多少時間
2. `repeatCount` : 我們要執行這個動畫幾次，如果只要跑一次這個動畫，設定成 1 即可；如果我們想要這個動畫一直跑的話，不妨就把這個動畫設成 `NSNotFound`，`NSNotFound` 就是整數的最大值。

比方說，當我們想要讓某個 layer 一直不停的旋轉，我們可以修改

`transform.rotation.x`、`transform.rotation.y`、`transform.rotation.z` 等，要求這個 layer 是按照 x、y 還是 z 軸旋轉，我們可以讓 `fromValue` 設成 0，代表是初始還沒旋轉的狀態，至於 `toValue` 設成 `M_PI * 2`，代表要旋轉 360 度。像以下這段程式：

```
[super viewDidLoad];
self.aLayer = [[CALayer alloc] init];
self.aLayer.frame = CGRectMake(50, 50, 100, 100);
self.aLayer.backgroundColor = [UIColor redColor].CGColor;
CABasicAnimation *rotateAnimation = [CABasicAnimation animationWithKeyPath:@"transform.rotation.z"];
rotateAnimation.fromValue = @0.0f;
rotateAnimation.toValue = @(M_PI * 2);
rotateAnimation.autoreverses = YES;
rotateAnimation.repeatCount = NSUIntegerMax;
rotateAnimation.duration = 2.0;
[self.aLayer addAnimation:rotateAnimation forKey:@"x"];
[self.view.layer addSublayer:self.aLayer];
```

效果如下：



CAKeyframeAnimation

使用 CAKeyframeAnimation 與 CABasic Animation 的主要差別在於，我們想要透過改變某個屬性產生動畫時，不是設定初始值與結束值，而是使用一個貝茲曲線描述（是 CGPath）。所以，當我們希望動畫不只是直線前進，而是按照某種曲線移動的時候，就可以使用 CAKeyframeAnimation。

CAAnimationGroup

一個 CALayer 可以同時執行多個 CAAnimation，當我們加入了一個CAAnimation 之後，就會立刻執行這個動畫。而我們也可以把很多個 animation 物件包裝成群組（像是 layer 一邊移動位置一邊翻轉），方法就是建立 CAAnimationGroup 物件，然後把想要變成群組的其他動畫，變成 array，設定成 CAAnimationGroup 的 `animations` property。

```
CAAnimationGroup *group = [CAAnimationGroup animation];
group.duration = 0.5;
group.animations = @[positionAnimation, flipAnimation];
group.delegate = self;
```

```
[aLayer addAnimation:group forKey:@"group"];
```

CAAnimation 的細部設定

避免動畫結束時 Layer 回到初始值

我們對一個 CALayer 設定了 CABasicAnimation 或 CAKeyframeAnimation 的動畫之後，往往會遇到一個問題：我們期待在動畫結束之後，CALayer 會停在動畫結束時的屬性設定，但是 Core Animation 的預設行為是動畫結束的時候，CALayer 反而會跳回初始的狀態。這個時候我們就要修改 CAAnimation 的 `fillMode` 屬性。

每個 CAAnimation 的 `fillMode` 屬性預設是設定成 `kCAFillModeRemoved`，代表的就是，在動畫完成的時候，就把 Layer 上的這個動畫移除掉。如果我們不想要跳回初始值，就應該把 `fillMode` 設定成 `kCAFillModeForwards`。

一個動畫結束之後，再繼續另外一個動畫

如果我們同時對一個 CALayer 加了多個 animation 物件，這些動畫都會同時進行，而不會一個動畫做完、才接續下一個動畫。如果我們想要一個動畫結束之後，繼續下一個動畫，你可能會使用 timer 延遲，不過，我們建議使用使用 CAAnimation 的 delegate，讓 delegate 告訴我們動畫結束，才繼續下一步。

CAAnimation 的 delegate method 定義在 NSObject (CAAnimationDelegate)中，這種把 delegate 定義成 NSObject 的 category 的寫法，叫做 informal protocol。現在 informal protocol 不是很常見，大概就只有 Core Animation 還在用這種用法。關於 formal protocol 與 informal protocol，請參見 [Formal Protocol 與 Informal Protocol](#) 這一節。

CAAnimation 的 delegate method 包括：

- `- (void)animationDidStart:(CAAnimation *)anim;`
- `- (void)animationDidStop:(CAAnimation *)anim finished:(BOOL)flag;`

當動畫結束的時候，就會呼叫 `-animationDidStop:finished:`。不過，由於我們在這邊只會拿到 CAAnimation 物件，如果想要判斷這是哪個 Layer 上面發生的動畫，然後對這個 Layer 繼續下一步的動畫，大概得要用這種方法來判斷：

```
- (void)animationDidStop:(CAAnimation *)anim finished:(BOOL)flag
{
    if (anim == [someLayer animationForKey:@"someKey"]) {
        // 繼續其他的動畫
    }
}
```

在這邊需要特別注意：如果我們想用這樣的判斷式，判斷這個動畫發生在哪個 layer 上，那麼，在建立 animation 物件的時候，就得要把 animation 物件的 `removedOnCompletion`（代表動畫在完成的時候移除）屬性設成 NO。不然等到動畫結束的時候，呼叫 `[someLayer animationForKey:@"someKey"]` 就會拿到 nil —因為當 `removedOnCompletion` 是 YES 的狀況下，動畫完成的時候會移除，對 someLayer 來說，在 someKey 這邊的動畫就已經被移走了。

不過，如果我們還是要繼續對某個 layer 增加動畫，一直不把原本已經完成的動畫拿掉，也會是個問題。所以，我們可以考慮在加入新的動畫之前，先對 layer 呼叫一次 `removeAllAnimations`，把原本的動畫先拿掉。

CADisplayLink

我們在寫貪食蛇的練習的時候，曾經使用 NSTimer 定時要求 view 重繪（對 view 呼叫 setNeedsDisplay），而達到讓 view 產生讓蛇移動的動畫。不過，如果我們希望有更順暢的動畫，我們會選擇 Core Animation，而不是這種讓 view 一直重繪的動畫。

原因是，在 timer 所處在的 run loop 中，因為要處理多種不同的輸入，所以 timer 的最小週期是在 50 到 100 毫秒之間（相當於 0.05 到 0.1 秒），一秒鐘之內，頂多只能夠跑 20 次 NSTimer。¹

如果我們希望在螢幕上看到流暢的動畫，那麼，一秒鐘就必須要有 60 格，每格之間的間距 0.016 秒左右，NSTimer 拿不到這個速度。而 Core Animation 有另外一個 timer 物件，叫做 CADisplayLink。

雖然 CADisplayLink 也是要註冊在 runloop 中，但不像 NSTimer。NSTimer 需要在上一次 run loop 整個跑完之後才會驅動，而註冊了 CADisplayLink 之後，只要螢幕需要重繪，就會呼叫 CADisplayLink 所指定的 selector。所以我們可以看到像 Cocos2D 這類的遊戲引擎，便是使用 CADisplayLink 作為 animator，每格 60 秒重繪一次畫面。

跟 NSTimer 不同，NSTimer 可以指定 timer 發生的間隔，但是 CADisplayLink 的 timer 間隔是不能調整的，固定就是一秒鐘發生 60 次。也就是說，當我們在一般用途下，想要指定某件事情要在某個時候發生，就會選擇使用 NSTimer，但我們需要一個高效率、專門用來負責更新畫面的 timer，就會使用 CADisplayLink。

Cocos2D 這類遊戲引擎在繪製畫面時，使用的是 OpenGL。不過，其實我們也可以在 CADisplayLink 指定的 selector 中，透過 CATransaction 關閉 Implicit Animation 後（呼叫 CATransaction 的 setDisableActions: 傳入 YES），然後改變 CALayer 的屬性，像是大小、位置等，如此一來，也可以產生許多的 CALayer 動畫效果。

```
- (void)startDisplayLink
{
    self.displayLink = [CADisplayLink displayLinkWithTarget:self selector:@selector(handleDisplayLink:)];
    [self.displayLink addToRunLoop:[NSRunLoop currentRunLoop] forMode:NSDefaultRunLoopMode];
}

- (void)handleDisplayLink:(CADisplayLink *)displayLink
{
    [CATransaction begin];
    [CATransaction setDisableActions:YES];
    // 改變 layer 的屬性
    [CATransaction commit];
}
```

¹. 請參見 NSTimer 的文件

https://developer.apple.com/library/mac/documentation/Cocoa/Reference/Foundation/Classes/NSTimer_Class/



在 Mac 上使用 Core Animation

前面提到，一個 layer 應該要怎樣設定 content scale (layer 的解析度、裡頭的內容的精細程度)，會跟這個 layer 出現在哪個 window 上有關。在 iOS 上我們通常只需要問 UIScreen 的 main screen 即可，因為 iOS 的 key window 就只會出現在 main screen 上，而我們通常可以假設 iOS 的外接螢幕裝置都不會有 retina display。

至於在 Mac 上，由於一台 Mac 可能本身的螢幕具有 Retina Display，但是另外用 mini display port 等介面外接了其他不是 Retina Display 的螢幕或投影機，而一個 window 可能會被拖放到不同的 screen 上，所以，當某個 window 移動到某個 screen 上之後，上面的 layer 也要跟著反應，把這些 layer 設定成對應的解析度。

比方說，我們一開始有一個 Window，出現在一台 Retina Display 的 MacBook Pro 的主要螢幕上，這個 Window 有個 layer，那麼這個 layer 的 content scale 就要設定成兩倍。但接下來，我們把這個 Window 移動到沒有 Retina Display 的外接螢幕上，layer 的 content scale 設成兩倍並沒有意義，所以應該改回變成一倍解析度。

在 Mac 上，我們不是去問 screen 的 scale，而是問目前所在 window 的 `backingScaleFactor` 屬性，而當 window 的 `backingScaleFactor` 屬性改變時，會發送 `NSWindowDidChangeBackingPropertiesNotification` 通知。

我們通常會在擁有某個 layer 的 view 上面偵測 window 解析度的改變，在 view 被加到 window 上時，會呼叫到 NSView 的 `viewDidMoveToWindow`，至於什麼時候 view 會從 window 上移除呢？就是呼叫 `removeFromSuperview` 的時候了。所以，我們 override 掉這兩個 method，收聽 `NSWindowDidChangeBackingPropertiesNotification` 的通知，把目前 window 的 `backingScaleFactor`，設成我們的 layer 的 content scale。

程式碼如下：

```
@implementation MYView

- (void)viewDidMoveToWindow
{
    [[NSNotificationCenter defaultCenter] addObserver:self selector:@selector(scaleDidChange:) name:NSWindowDidChangeBackingPropertiesNotification object:[self window]];
    [self _updateContentScale];
}

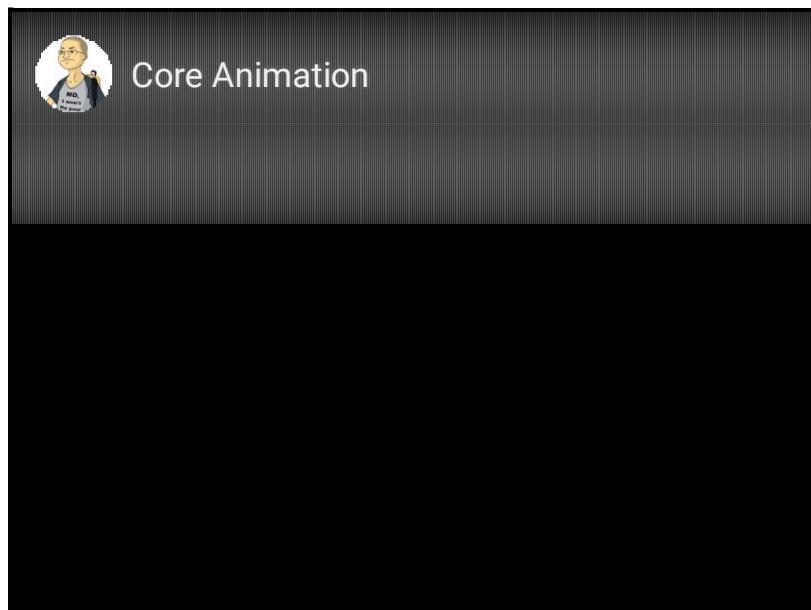
- (void)removeFromSuperview
{
    [[NSNotificationCenter defaultCenter] removeObserver:self name:NSWindowDidChangeBackingPropertiesNotification object:[self window]];
    [super removeFromSuperview];
}

- (void)scaleDidChange:(NSNotification *)n
{
    [self _updateContentScale];
}

- (void)_updateContentScale
```

```
{  
    if (![_self window]) {  
        return;  
    }  
  
    CGFloat scale = [[_self window] backingScaleFactor];  
    [_myLayer setContentsScale:scale];  
}  
@end
```

教學影片



<https://speakerdeck.com/dryman/mastering-core-animation>

練習：KKBOX 動態歌詞

Carrier 11:11 AM
快點殺了我
小護士樂團 (La Petite Nurse) ...

Lyrics by: 霧文
Composed by: 霧文

主唱: 霧文 吉他: 寶弟／霧文 貝斯: 培修
鼓: 小朱

怎麼我覺得這裡才是我的家
怎麼我貪戀你獨特香味的髮
就讓慾望在高潮的頂點燒吧
我很自大烈焰灼身也不怕

你的吸引力帶著激情的神話
像一個黑洞吞噬我把我消化

+ (••) Start Radio Lyrics

2 / 10
01:06 04:48

↻ ⏪ ⏴ ⏵ ⏩ ⏷

練習範圍

- Core Animation
- Timer

練習內容

KKBOX 的播放畫面中提供了動態歌詞功能，在畫面中有許多的 text layer，而當歌曲剛好到了某行歌詞的時候，這行歌詞會 highlight 起來。

請使用 Core Animation 實作這樣的效果。我們可以不必按照真正的歌曲進度實作這個練習，可以找一首歌曲，把每一行歌詞都變成一個 layer，然後建立 timer，每當 timer 跑一次，就把下一行歌詞 highlight 起來，原本 highlight 的那一行就取消 highlight。

練習：Flappy Bird

練習範圍

- Core Animation
- CADisplayLink

練習內容

這個遊戲也可以使用 SpriteKit 等 framework 實作，不過，我們在這邊使用 Core Animation，也一樣可以完成這個遊戲。

在 Flappy Bird 這個遊戲中，看起來像是鳥在不斷往前飛行，但其實鳥並沒有水平移動，而是只有垂直移動而已：當我們觸控螢幕的時候，鳥會往上移動，接著隨著時間就會一直往下掉落。真的在水平移動的，其實是後面的柱子。

無論是鳥或是柱子，都是 CALayer，接著，我們會安排一個 CADisplayLink，定時更新鳥與柱子的位置。

- 先放置鳥的 layer，放在畫面的正中央。鳥的大小為 40x40 pixel。
- 然後我們設定好一系列柱子的 layer。柱子的上下分別為兩個 layer，我們先用迴圈放置三十對柱子（先假設這麼難玩的遊戲應該不會有人可以突破三十關...）總共就有 60 個 layer
 - 每對柱子中，上方的柱子的高度為 130 到 $130 + (\text{螢幕高度} - 320)$ 之間
 - 上方柱子與下方柱子的距離為 130 pixel
 - 下方柱子的高度就是從上方柱子的 $y + 130$ 開始，繼續填滿螢幕高度
 - 每根柱子的寬度是 60 pixel
 - 每對柱子與柱子的水平距離是 140 pixel
 - 第一根柱子的 x 軸剛好在畫面寬度外
- 用 UIGestureRecognizer 寫一個 tap 的 action，叫做 fly。裡頭的實作是鳥的 position 的 y 會減少 100 pixel
- 寫一個 CADisplayLink，CADisplayLink 的 action 中：
 - 每執行一次，鳥的 position 的 y 軸都往下掉 4 pixel
 - 每執行一次，每根柱子的 position 的 x 軸都減 1 pixel
 - 檢查鳥的 y 是否超過螢幕範圍，如果是，判定鳥落地，遊戲結束
 - 檢查每根柱子的 frame 是否與鳥的 frame 交疊 (CGRectIntersectsRect)，如果是，遊戲結束
 - 先讓這個 CADisplayLink 保持 pause 狀態 (paused 屬性設成 YES)
- 在畫面正中央放置一個「開始遊戲按鈕」
 - 按鈕按下去後，按鈕會消失
 - 把 CADisplayLink 的 paused 設成 NO，進行遊戲

練習：自由發揮

練習範圍

- Core Animation

練習內容

我們前面做了很多指定的練習，現在來發揮你的想像力與創造力吧！看看你可以想到能用 Core Animation 做些什麼！

Audio API

KKBOX 是一套音樂服務，總是要來講一些跟 Audio 相關的開發。

如果只是想要在 iOS 裝置上播放聲音，其實有不少高階而且簡易的 API。在 AVFoundation Framework 就有 AVPlayer 與 AVAudioPlayer，AVAudioPlayer可以輕鬆播放 local 的音訊檔案，而 AVPlayer 可以播放網路上的影音串流。如果我們開發的 App 是一款遊戲，想要在遊戲過程中觸發音效，像開槍的時候可以發出槍響，在 SpriteKit 裡頭，SKAction 的 `+playSoundFileNamed:waitForCompletion:` 也非常好用。

然而，因為 KKBOX 的商業需求，以及用戶期待的功能，我們沒辦法使用這些高階 API 開發 KKBOX，必須使用更底層的 audio API，而就整個 Mac OS X 與 iOS 開發中會用到的 API 來論，跟 audio 相關的 API 都是出了名的難用。我們在這一章中會講比較少的 code，而會花比較多力氣講解寫一套 Audio Player 需要用到哪些東西。

基礎知識

在進入如何使用蘋果的 API 之前，先講一些跟 Audio 相關的基礎知識。這些資料在 wikipedia 上都可以查到，我們盡可能簡短說明。

聲音與數位音訊

聲音是空氣或是在水中的震動，是一種能夠被人耳所感知的類比訊號，單位是分貝（dB），人耳可以忍受的聲音介於 0 到 100 分貝之間。震動幅度的強弱決定了是大聲或是小聲，震動頻率的密集程度決定了是高音或是低音。

所謂的數位音訊或數位音樂，就是將聲波這種連續的類比訊號，轉換成一連串的數字，用零與一的二進位數字盡可能地重現在大自然中發生的事物；用一個數字，標示在某個時間單位時訊號的強弱程度。

之所以說「盡可能」，是因為類比轉換成數位的過程中一定會有所捨棄。我們捨棄了超過 100 分貝的部分，在大自然中存在超過 100 分貝的聲音，但是超過了人耳可以忍受的範圍；而在大自然中，時間可以不斷無限切分到人類感官無法感知的單位，但我們要將訊號記錄下來，還是必須要定義一個單位。這種將一個一個時間單位的訊號強弱程度記錄下來的過程，叫做採樣（sampling），每一個時間單位上的樣本，叫做 sample，或是一個 frame，一秒鐘內有多少 sample，叫做採樣比（sample rate）。

現在定義的 CD 音質是 44100 Hz，也就是一秒鐘內有 44100 個 sample，這個規格的原因是人耳可以聽到的頻率介於 20 到 20000 Hz 之間，根據 Nyquist–Shannon 的採樣定律，只要採樣比超過 20000 Hz 的兩倍，人耳就根本聽不出差別。被錄製下來的聲音可以用帶號、非帶號的整數或浮點數儲存，如果是非帶號整數的話，每個 sample 會介於 0 到整數最大值之間，帶號整數就可能是整數最大值到最小值之間，非帶號浮點數則會介於 0.0 到 1.0 之間，帶號浮點數就可能會在 -1 到 1 之間。

這種完全還沒有壓縮過的資料，就叫做 PCM（Pulse-code modulation）格式（請參考 https://en.wikipedia.org/wiki/Pulse-code_modulation），我們常用的 WAV、AIFF、CAF 檔案，都是這種格式，而 Audio 硬體最後要播放的，也是這種資料。所以，CD 其實就是一秒鐘有 44100 個 16 位元帶正負號的整數，而從 CD 開始，其實我們就已經進入了數位音樂時代。

沒有壓縮過的音訊檔案以現在的眼光看都還是很大。比方說，我們用 16 位元非帶號整數（兩個 byte）儲存，加上左右聲道，一首歌曲就要 30 mb ($44100 \times 2 \times 2 \times 60 \times 3 \div 1024 \div 1024$)，雖然可以存放在 CD 這種介質上，但並不適合網路傳輸。到了 90 年代，MP3 等壓縮格式開始流行，音樂變得可以在網路上傳輸、交換，從此也大幅改變了音樂產業，人們從購買 CD 變成透過網路取得音樂，也讓現在可以有 KKBOX 這樣的網路串流音樂服務。

壓縮音檔：Codec 與 Container

壓縮過的音訊檔案尺寸大大降低，但像 MP3 等格式是破壞性的壓縮，在壓縮過程中，也會造成音質的降低與失真，所以壓縮的比例同時影響壓縮後的大小與聆聽時感受到的品質。

音檔壓縮的比例我們使用 bitrate 描述，bitrate 就是這個音檔用多少的資料（注意，單位是 bit，不是 byte）表示一秒鐘的聲音，KKBOX 目前提供 128kb、192kb、320kb 三種不同品質的音檔，單位都是 bitrate。kb 是使用十進位，所以我們可以推算出一首三分鐘的 320kb 的音檔，大約 6.8 MB ($320 / 8 \text{ } 1000 \text{ } 60 * 3 / 1024 / 1024$)。

當我們將原始音檔壓縮成 MP3 格式的時候，並不是把整個檔案都壓縮起來，而是先將連續的原始音檔切成一個一個的小塊，然後一次只壓縮一個小塊，這樣的小塊叫做 packet，MP3 格式的每個 packet 為 1152 個 frame，用來將每個小塊做壓縮與解壓縮的程式就是 codec。

之後，我們會在每個 packet 的前方都加上一個簡短的檔頭，標示在這之後是一個 packet，以及 packet 的長度，一個 MP3 檔案，就是連續的檔頭與 packet 的集合，我們要播放 MP3 檔案，首先就是要透過 parser，找出每個 packet 所在的位置。AAC ADTS 格式也是使用這種方式包裝資料。

至於 MP4 檔案則是用另外一種方式包裝資料。不同於 MP3 是連續的檔頭與 packet，MP4 格式則是一種巢狀結構：在一個 MP4 檔案中，會有許多叫做 atom 的容器，一個 atom 裡頭可能還會有其他的 atom。其中最主要的兩個 atom 分別是 moov 與 mdat，mdat 裡頭是連續的 packet 資料，但是這一段資料中並不會特別著名哪一個 packet 從哪裡開始到結束，而 moov 這邊有所有的 packet 的 offset 位置，也就是，mdat 裡頭有哪些 packet，要拿 moov 這段資料當做索引。用來包裝 packet 的格式，就叫做 container。

moov 與 mdat 不一定要哪個在前哪個在後，但如果你要使用 Core Audio API 播放 MP4 檔案，moov 一定要放在 mdat 前方，不然 Core Audio API 所提供的格式 parser 會告訴你：它不支援沒有 optimized 過的 MP4 檔案。

但很奇妙，如果你把這個檔案拿去 iTunes 或 QuickTime 裡頭，卻可以正常播放；所以呢，其實蘋果自己的播放軟體裡頭用的底層，與蘋果公開的 API 並不是相同的東西，我們也不能期待用蘋果的播放軟體能夠播的檔案，我們就能夠播出來。如果想要知道某個檔案能不能用蘋果的公開 API 播放，可以使用 command line 底下的 afinfo 與 afplay 指令檢查。

當然，如果你不信任系統提供的 parser，也可以寫自己的 parser。iOS 與 Mac OS X 上的 audio format parser 叫做 Audio File Stream Service，在 Mac 上是 10.5 以後才出現的 API；KKBOX Mac 版是在 2008 年開始開發，當時還必須支援 10.4，因此我們最早也寫了自己的 MP3 parser。

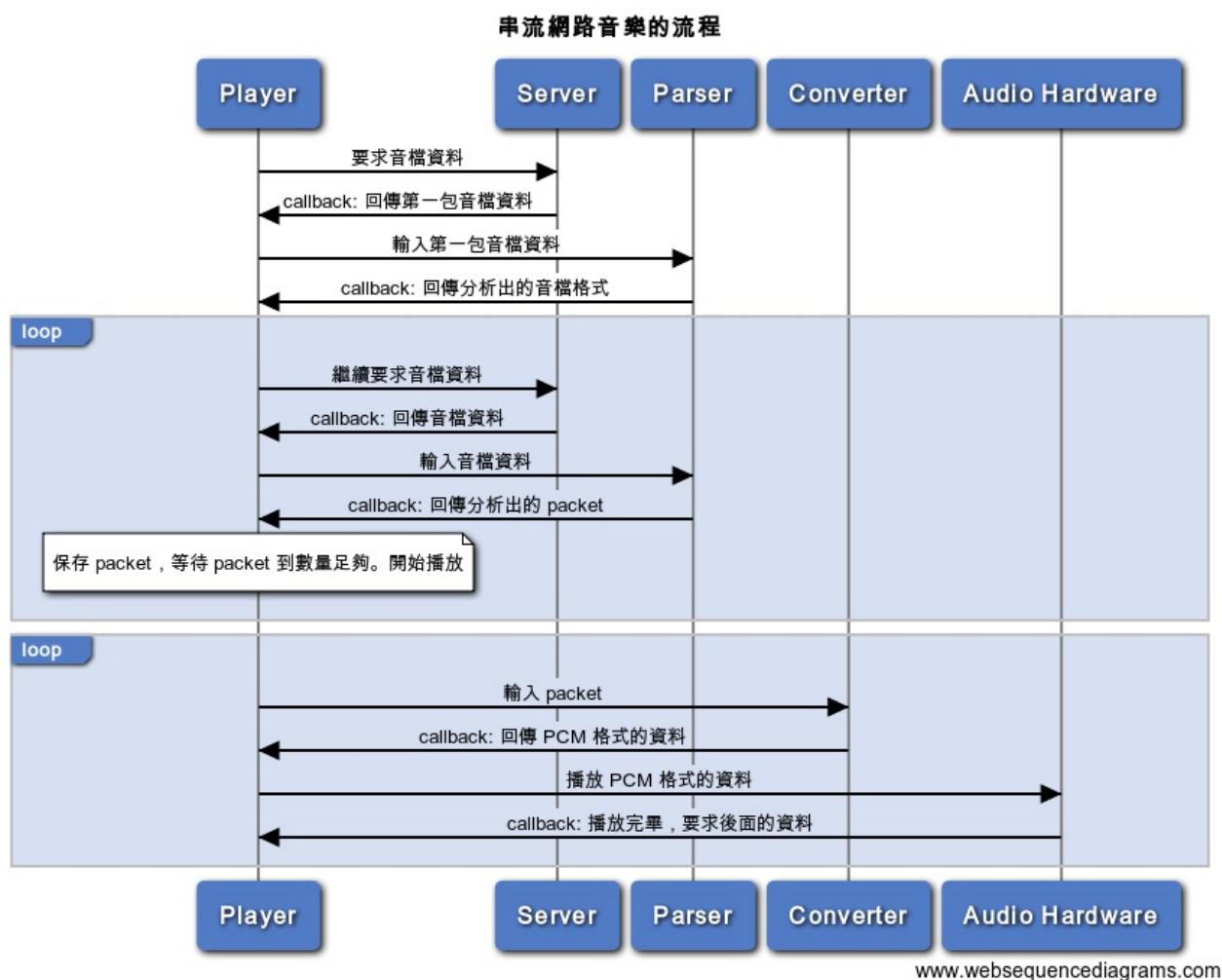
所以，當我們在稱呼一種音檔的時候，通常得要同時說明這種音檔的 codec 與 Container：

- MP3 同時是一種 codec 與 container，所以可以直接稱呼成 MP3
- AAC 是一種 codec，但是有 MP4 與 ADTS 這幾種不同的 container 格式，所以我們會稱為 AAC MP4 或是 AAC ADTS。
- OGG 是一種 container，裡頭用的 codec 通常是 Vorbis，但 FLAC 格式雖然平常放在 FLAC 自己的 container 中，但也有可能用 OGG container 格式包裝 FLAC 資料，這種格式我們就會稱為 OGG FLAC。

播放網路串流的流程

所以，如果現在我們想要播放位在 server 上的某個 MP3 或 AAC 音檔，流程大概就是：

1. 發送網路連線，跟 server 要求讀取這個檔案，並且等候連線的 callback
2. 在連線的 callback 中儲存資料，如果這個檔案經過 DRM 保護的話，也要在這時候做加解密
3. 把收到的資料送到 parser 中，parse 出檔案格式與 packet 的位置
4. 保存這些 packet
5. 按照播放硬體的播放進度，定時把這些 packet 交給 converter，讓 converter 呼叫 codec，將我們的 packet 轉換成 PCM 格式
6. 讓 Audio 硬體播放 PCM 格式的資料



www.websequencediagrams.com

iOS 與 Mac OS X 的 Audio API 概觀

我們知道串流音樂的播放流程之後，現在就來看一下 iOS 與 Mac OS X 上有哪些適合的 Audio API。我們大概分成兩群

不適合用在音樂播放的：

- System Sound
- OpenAL

可以用用在音樂播放的：

- NSSound
- QuickTime
- AVFoundation
- Audio Queue
- Audio Unit Processing Graph
- AVAudioEngine

System Sound Services

在 iOS 上，如果我們想要發出簡短的系統提示音效，像是按了某個按鈕要發出的聲響，會選擇使用 System Sound Services。

System Sound Services 提供了一些 C API 可以讓我們播放小於三十秒的 CAF (Core Audio Format) 檔案。只要透過 `AudioServicesCreateSystemSoundID` 載入指定的檔案 URL、建立 SystemSoundID 後，就可以用

```
static SystemSoundID soundFileObject;
if (!soundFileObject) {
    CFBundleRef mainBundle;
    mainBundle = CFBundleGetMainBundle();
    CFURLRef soundDataURLRef;
    soundDataURLRef = CFBundleCopyResourceURL(mainBundle, CFSTR("sound"), CFSTR("aiff"),
), NULL);
    AudioServicesCreateSystemSoundID(soundDataURLRef, &soundFileObject);
    CFRelease(soundDataURLRef);
}
AudioServicesPlaySystemSound(soundFileObject);
```

此外，當我們想要讓 iPhone 出現像來電時的震動效果，也是呼叫 System Sound Services，因為震動效果其實被定義成一種系統音效。我們想讓手機震動，就呼叫：

```
AudioServicesPlaySystemSound (kSystemSoundID_Vibrate)
```

只能播放 30 秒的未壓縮格式，顯然沒有辦法滿足我們想要播放 MP3、AAC 這些格式的需求。另外 System Sound Services 發出的聲音有個特點：只會從 iOS 裝置上的本機播放出來，就算用鏡像的 AirPlay 模式連到了 Apple TV 上，還是只會從本機播放。

OpenAL

OpenAL 是一套開放規格，用途是產生在遊戲當中提供立體的音效。當我們在玩一款第一人稱射擊遊戲的時候，如果感受到敵人的腳步聲由遠而近，或是可以用槍聲的來源感受到敵人的位置，這種效果便可以透過 OpenAL 達成。

最近似乎大家都很少直接呼叫 OpenAL API，許多的遊戲引擎都在 OpenAL 的基礎上又另外架設了一套音效引擎，就連蘋果自己的 SpriteKit 裡頭都有播放音效用的 API。

OpenAL 一樣是 C API，一樣只能夠播放未壓縮的音檔，播放流程是先將整個音檔的資料載入到 buffer 中，然後將 buffer 丟入 source 播放，然後可以對 source 做像是位置等設定。

如果要拿 OpenAL 播放 MP3... 也不是不行，我們可以先把 MP3 檔案載入到記憶體中，用 converter 將整個 MP3 檔案轉成 LPCM 格式，如果是三分鐘的歌，就把 30MB 的資料載入到記憶體中，聽起來就不太可行。

播放串流音樂的時候，我們只會把少量的資料轉換成 LPCM 格式播放，等前一包資料播放完畢後，再繼續提供下一包的資料。那我們有沒有辦法用這個原則先把整個檔案切成幾個小包，然後交給 OpenAL 播放呢？實際上不太可行，因為 OpenAL 裡頭缺乏音效播放完畢的 callback 機制，不會通知我們哪個音效播放完畢，當然也可以用 timer 算好時間播放某個 buffer，但我們前面也提到了 timer 的原理，知道 timer 其實並不精確。

NSSound、QuickTime、AV Foundation

NSSound、QuickTime 與 AV Foundation 是 Mac OS X 與 iOS 上的高階 API，只要提供檔案的遠端或本機 URL，便可以直接開始播放，

NSSound 主要是用在 Mac 上用來播放簡短的提示音效用的 API—當然，當我們只想要在 Mac 上發出一個用來提示錯誤用的聲響，也可以簡單呼叫 `NSBeep()` 就可以了—雖然定位很接近 System Sound Services，但 NSSound 實際上可以載入多種格式、而且檔案長度較長的音檔，我們還可以知道播放的檔案長度，要求指定的播放時間位置（random seek），還可以調整音量、要求一直 loop 重播等。

雖然 NSSound 可以說是一個完整的 Audio Player，不過定位上還是偏向用來播放系統提示音效在蘋果為了 API 一致、將 AV Foundation 從 iOS port 回 Mac 之前，在 Mac 上播放各種媒體檔案，會更常使用 QuickTime API 裡頭的 QTMovie 這個 Class。QTMovie 可以做到 Mac OS X 系統中 QuickTime Player 可以做到的事情，除了播放 audio 之外，還包括播放甚至編輯 QuickTime 影片的功能。不過，蘋果在 Mac OS X 10.9 中就將 QTMovie 標成 deprecated。

在 Mac 與 iOS 上的 AV Foundation 不完全相同。在 AV Foundation 裡頭有三個直接與 Audio 播放相關的 class，按照出現的時間排列，分別為 AVAudioPlayer、AVPlayer 與 AVAudioEngine。我們先來看 AVAudioPlayer 與 AVPlayer。

AVAudioPlayer 是在 iPhoneOS 2.2 上推出，大概是在 iPhoneOS SDK 問世後半年左右出現的 API。用 AVAudioPlayer 相當適合用在像是播放遊戲背景音樂等應用上，這個 class 明顯的優點是可以播放多種格式的檔案，但也有兩個明顯的缺點。

其一是 AVAudioPlayer 只能夠播放位在本機的檔案，而無法指定放在網路上的URL 播放，所以，直到 iOS 4 AVPlayer 推出之前，在播放網路上的音檔時，如果不想要用底層的 C API，要不就是先把整個檔案抓下來之後用 AVPlayer 播放，要不就是用 UIWebView 開啟，但整個畫面都會變成 web view 的播放畫面。

AVAudioPlayer 的另外一個缺點則是，蘋果在 iOS 4 開始支援背景執行，其中一項背景模式是支援背景 audio 播放，但 AVAudioPlayer 並不支援，無法在背景播放 audio。

iOS 4 時推出的 AVPlayer 是一個好用的元件，AVPlayer 支援多種現在流行的檔案格式，不但可以用來播放音訊，也可以播放影片，只要指定要播放的URL（包括本機與遠端），便可以開始播放，如果要播放的是影片，則可以搭配AVPlayerLayer 顯示畫面，也支援背景播放。

在前一節提到播放網路串流音訊的流程，AVAudioPlayer 與 AVPlayer 都做完裡頭的六個步驟；在絕大多數場合中，AVPlayer 可以滿足播放的需求，不過，如果你有以下需求：

- 因為商業上的需求，我們要播放的檔案經過加密，在播放的過程中需要先解密才能播放。
- 我們想要提供更多的播放效果，像是增加迴音、升降 key、支援 EQ 等化器等...。

那我們就只能夠使用更底層的 Audio API 了。

Audio Queue

Audio Queue 是 Mac OS X 與 iOS 上用來播放與錄製音訊的 C API。先看播放的部份，對照前一節提到的播放步驟，Audio Queue 可以幫我們簡化最後兩步：我們只要一開始指定好 Audio Queue 是哪一種是檔案格式，之後只要提供這種格式的資料就可以播放，我們不用自己動手把原本的 MP3 或 AAC 格式轉換成 LPCM，但前面分析出 packet 這步還是得自己來。

Audio Queue 是[Audio Toolbox Framework](#) 的一部分。Audio Toolbox 是一個不算小的 Framework，裡頭包含我們前面提到的 System Sound、後面要提到的 Audio Unit Processing Graph 外，也包含我們在播放過程中會用到的 parser 與 converter，parser 包括 Audio File Services 與 Audio File Stream Services 等，至於 Audio Converter Services 便是 converter。

Audio Queue API 主要由兩個主要的資料類型組成：Audio Queue 與 Audio Queue Buffer。我們不妨把 Audio Queue 想像成是一個水池，而每個 Audio Queue Buffer 則是許許多多的水桶，在要求 Audio Queue 開始播放音訊之後，Audio Queue 這個水池一開始是乾的，所以我們要提第一桶水桶，把水桶裡頭的水倒進水池裡頭（這一步叫做 enqueue buffer）。接著，這個水池會因為水慢慢地被用掉，於是慢慢變乾，但是在完全乾枯之前會通知我們（callback）水快沒了，所以跟我們要下一桶水；接著就是這樣的步驟不斷循環。

至於錄音，則是將前述步驟整個倒轉過來，Audio Queue 與 Audio Queue Buffer 仍然扮演水池與水桶的角色，只是我們要先把一個空水桶放進乾掉的水池裡頭，當錄音的資料進來的時候，水會裝進這個水池裡頭唯一的水桶裡，當一個水桶快要裝滿時，Audio Queue 就會通知我們趕快把裝滿水的水桶拿出去存檔，同時再拿一個空的水桶進來裝水。然後不斷循環這個步驟。

在 iOS 上使用 Audio Queue API 的時候，還要搭配正確的 Audio Session。Audio Session 是一種用來描述我們的 App 打算怎麼使用 Audio 的 API，要正確設定 Audio Session 的類型，並且讓 Audio Session 變成 active，系統才會允許我們做一些我們想做的事情，像我們必須告訴系統我們是媒體播放的 App，系統才會允許我們執行背景播放，要告訴系統我們是可以播放與錄音的軟體，才有辦法使用麥克風錄音。

在使用 Audio Queue API 播放音訊的時候，我們要稍微注意一下對播放時間的控制。如果我們想要知道一首歌現在播放到哪個時間，一般在 player 的 UI 上我們大概只需要精確到秒就好了，而我們最小可以到達的單位則是 packet 的大小，透過我們送出了多少 packet 而推算出播放時間。

我們在建立 Audio Queue Buffer 的時候，通常會建立比較大的 buffer，可能會是半秒、一秒甚至更多秒數的 buffer，假如我們送出了一個一秒鐘的 buffer，在播放這一秒的時候，其實我們不太能精確掌握「我們播放到了這一秒鐘的哪個地方」。Audio Queue 雖然有兩個跟播放時間相關的 C function，這兩個function 回傳的時間通常也不是很精確。

由於 Audio Queue 只要拿到 AAC、MP3 資料就可以播放，而如果我們想要做一些音效的處理，會是在 LPCM 資料這一層做。KKBOX 最早使用 Audio Queue 開發播放器，但由於會員一直要求我們能夠提供 EQ 等化器等功能，所以後來重新開發了使用 Audio Unit Processing Graph API 為基礎的播放器。

Audio Unit Processing Graph 與 AVAudioEngine

如果我們想要對音訊播放擁有最完整的控制，那我們最後的選擇，就是最底層的 Audio Unit Processing Graph 這層 C API，以及把 Audio Unit Processing Graph 再用 Objective C 包裝一層的 AVAudioEngine。

上面提到的不少 API，像 OpenAL、Audio Queue 以及 AVFoundation 等，也是在 Audio Unit Processing Graph API 上架構的。

我們先來解釋一個名詞：Core Audio。Core Audio 是蘋果的整個 audio 的架構底層，我們前面所講的 Audio Toolbox framework 的 API，是屬於 Core Audio 的應用層，Audio Unit Processing Graph 算是應用層的最底層，已經距離操控硬體不遠了。

此外還有我們平常比較不會接觸到的硬體與 codec 這幾塊，在 iOS 上我們完全無法修改，但是在 Mac OS X 上，如果我們做了什麼新的 audio 硬體，或是要讓 Mac 的 player 支援某些新的檔案格式，開發者還是可以在 Core Audio 架構上寫新的 driver 或 codec，像我們想在 Mac 上播放微軟的 WMA，或是 RealPlayer 的格式的話，就得安裝額外的 codec。至於用來控制各種外部的電子樂器，則會用到 Core MIDI framework，這也是 Core Audio 的一部分。

Audio Unit Processing Graph 是一個可以處理播放與錄音的 API，這層把 audio 播放的處理過程，抽象化變成一個個的組件，我們可以透過組合這些組件創造我們想要的錄製與播放效果。Audio Unit Processing Graph API 裡頭大概有三個主要的角色：

- AUNode 或 AudioComponent
- AudioUnit
- AUGraph

我們不妨想像我們現在身處在演唱會的舞台上，有錄製歌聲與樂器的麥克風，而從麥克風到輸出到音響之間，還串接了大大小小的效果器，在這個過程中，無論是麥克風、音響或是效果器，都是不同的 AUNode。AUNode 是這些器材的實體，而我們要操控這些器材、改變這些器材的效果屬性，就會需要透過每個器材各自的操控介面，這些介面便是 AudioUnit，最後構成整個舞台，便是 AUGraph。

AUNode 與 AudioComponent 的差別在於，其實像上面講到的各種器材，除了可以放在 AUGraph 使用之外，也可以單獨使用，比方說我們有台音響，我們除了把音響放在舞台上使用外，也可以單獨拿這台音響輸出音樂。當我們要在 AUGraph 中使用某個器材，我們就要使用 AUNode 這種形態，單獨使用時，就使用 AudioComponent。但無論是操作 AUNode 或 AudioComponent，都還是得透過 AudioUnit 這一層操作介面。

AUNode 與 AudioComponent 分成好幾類，包括輸入、輸出、混音、效果處理、格式轉換等等，彼此之間可以互相串接。輸入裝置包括像麥克風輸入或 MIDI 樂器，效果處理則包括像 EQ 等化器、殘響（reverb）、改變音調（pitch）等；至於這邊所謂的格式轉換，是指在不同的 LPCM 格式之間轉換，在這一層 API 中只

支援 LPCM 格式，但LPCM 之間又有很多種，不見得每個 node 都支援所有的LPCM 格式，像 reverb效果的 effect node 就只支援浮點數，所以要讓音訊資料通過這個 node 之前，就需要先轉換成浮點數格式的 LPCM 資料。

每個 AudioUnit 都有各自的輸入與輸出，在串接的時候，就是從某個AudioUnit 的輸出，串接到另外一個 AudioUnit 的輸入，這種輸入輸出的端子叫做 bus，而每個 AudioUnit 最少會有一個輸入與輸出的 bus，也可能會有多個 bus。以 mixer 來說，就會有多個 bus，當我們從兩個輸入 bus 將資料送到同一個 mixer 上時，就可以產生混音效果。

在 Mac OS X 上，我們通常會使用 default output 作為音訊播放的最終輸出的 AudioUnit，以 default input 作為錄音的起點。在 iOS 上則有一個特別的AudioUnit，叫做 Remote IO，這個 AudioUnit 同時代表 iOS 的輸出與輸入裝置。Remote IO 有兩個 bus，bus 0 就是 iOS 的預設輸出，bus 1 則是輸入，所以我們在 iOS 上播放音樂，就是往 Remote IO 的 bus 0 傳送資料。

Remote IO 的 bus 1 預設是關閉的，當我們要錄音的時候，我們必須先告訴Remote IO 把 bus 1 變成 enable，但我們要做這件事情的時候，我們不但要獲得使用者給予我們使用麥克風的授權，還要設定正確的 Audio Session。我們會在後面說明 Audio Session。

在使用 Audio Unit Processing Graph API 的時候，我們經常需要設定 render callback function。以錄音來說，當我們從 Remote IO 的 bus 1 收到資料後，想要儲存檔案，我們並沒有一種叫做「存檔」的 AudioUnit，而是我們要對某個AudioUnit 設定 callback function，綁定某個 bus，在這個 function 中撰寫存檔的程式。以播放來說，當我們告訴 AUGraph 或 Remote IO 開始播放，我們也要設定 render callback function，提供用來播放用的資料。

由於這邊只支援 LPCM 格式，因此我們在播放 MP3 或 AAC 資料之前，還得有一個將 MP3 或 AAC 轉換成 LPCM 格式的 converter。總之，我們提到播放網路串流音樂有六個步驟，當我們用到這一層 API 的時候，這六個步驟都得自己來了。

Audio Unit 也是一種系統 plug-in，在 Mac OS X 與 iOS 上，除了內建的Audio Unit 之外，第三方也可以撰寫自己的 Audio Unit，Mac OS X 一開始就支援，不過 iOS 方面，則是 iOS 9 才開放，而且還要包在一個 hosting app中。

講完 AUGraph 會比較容易理解 AVAudioEngine，AVAudioEngine 是 Objective-C API，用 Objective-C 物件把 AUGraph API 多包裝了一層，AVAudioEngine 裡頭的 AVAudioPlayerNode、AVAudioUnitEffect 等等，都可以找到對應的 C API，但是高階許多。不過，從AVAudioPlayerNode 的設計來看，AVAudioEngine 看起來很容易處理本機檔案，只要傳入一個 file URL 就可以輕鬆播放，並且加入各種效果。不過，如果是網路串流，看起來我們還是得自己轉成 PCM Buffer 送給 AVAudioPlayerNode。

蘋果在 iOS 13 中 deprecate 了 AUGraph，希望開發者以後不要使用，不過，AUGraph 實際上是 AVAudioEngine 的底層，所以我們預期短時間之內 AUGraph 也不會完全消失。

使用 Audio Queue 開發播放軟體

我們現在來寫一個簡單的 Audio Queue Player，這個 Player 能夠播放位在 server 上的 MP3 檔案。

為了說明如何使用 Audio Queue API，所以這邊程式做了一定程度的簡化，很多需要做的事情，其實在這邊沒做。在這個 player 中，我們首先需要以下的成員變數：

- 用來保存 packet 的 array，在這邊使用了一個 NSMutableArray，裡頭會是一堆 NSData。在真正的產品 code 中其實不該這麼寫，因為這樣等於是把所有收到的 audio 資料都丟進記憶體中，如果是 300 mb 的 MP3，就會用到 300 mb 的記憶體，在產品 code 中應該要把一部分資料寫入暫存檔中。
- AudioQueueRef：就是我們的 Audio Queue。
- AudioStreamBasicDescription：我們在建立 Audio Queue 時所使用的 Audio 檔案格式。
- AudioFileStreamID：parser。
- readHead：是一個 size_t，用來表示我們現在讀到第幾個 packet。
- 一個用來抓取資料的 NSURLConnection 物件。

這個 Player 播放的步驟包括

1. 建立 Parser 與網路連線
2. 收到部分資料並 parse packet
3. 收到 parser 分析出的檔案格式資料，建立 Audio Queue
4. 收到 parser 分析出的 packet，保存 packet
5. packet 數量夠多的時候，enqueue buffer
6. 收到 Audio Queue 播放完畢的通知，繼續 enqueue

第一步：建立 Parser 與網路連線

在建立 Audio Queue 的時候，需要用 AudioStreamBasicDescription 傳入詳細的資料格式，包括 sample rate、這個檔案有多少 channel 等等，但我們現在還不知道遠端音檔的格式，所以會稍晚建立。

我們用 `AudioFileStreamOpen` function 建立 `AudioFileStreamID`，也就是我們的 parser，在這邊我們傳入了 `kAudioFileMP3Type`，代表說，我們猜測遠端的檔案是一個 MP3 檔，而 `AudioFileStreamOpen` 其實只有大概參考這個提示而已，如果遠端的檔案是 MP3 之外的其他格式，但我們在 `AudioFileStreamOpen` 告訴 `AudioFileStreamID` 的卻是 MP3，我們的 `AudioFileStreamID` 還是有能夠辨別出到底是哪種檔案。

不過 `AudioFileStreamOpen` 很容易誤判，如果在建立 `AudioFileStreamID` 之後，我們一開始透過呼叫 `AudioFileStreamParseBytes` 給 `AudioFileStreamID` 的資料不夠多，`AudioFileStreamID` 就常常回傳誤判的結果，像明明是 MP3 格式，卻告訴我們是 MP2 或 MP1。

在 `AudioFileStreamParseBytes` 裡頭還要傳入兩個 callback function，在這邊我們傳入我們定義好的 `KKAudioFileStreamPropertyListener` 與 `KKAudioFileStreamPacketsCallback`。

`KKAudioFileStreamPropertyListener` 是屬性改變的 callback，我們在這個 callback 主要想知道的是取得檔案格式的通知。而 `KKAudioFileStreamPacketsCallback` 則是 packet 的 callback，會在分析出了 packet 的時候呼叫。

接著我們就可以用 `NSURLConnection` 抓取檔案了。

第二步：收到部分資料與 parse packet

在 NSURLConnection 的 delegate method `connection:didReceiveData:` 中，我們會收到在這一輪 run loop 中，NSURLConnection 抓到了多少資料，我們便可以透過 `AudioFileStreamParseBytes`，將收到的資料交給 AudioFileStreamID 分析。

就像前面提到，我們最好一開始呼叫 `AudioFileStreamParseBytes` 的時候，就先給一包比較大的資料，所以我們在收到資料的時候，可能先找個 NSMutableData 然後 append 進去，等到 bytes 足夠的時候才呼叫 `AudioFileStreamParseBytes`，不過這邊為了簡化所以沒有這麼做。

前面也提到，如果我們用 Core Audio API 播放 MP4 檔案，必須要將 moov atom 放在 mdat 之前，正是因為 AudioFileStreamID 無法解析這種格式的 MP4。另外，如果你要播放的是 AAC ADTS 格式的檔案，這個檔案的前方可能會有一些 ID3 檔頭，我們也要自己把 ID3 檔頭濾掉，不然 AudioFileStreamID 會無法解析。

第三步：收到 parser 分析出的檔案格式資料，建立 Audio Queue

當 AudioFileStreamID 從我們提供的資料分析出檔案格式之後，就會呼叫我建立 AudioFileStreamID 時傳入的 `KKAudioFileStreamPropertyListener`。這個 callback 是在 AudioFileStreamID 的屬性改變的時候被呼叫，因為 AudioFileStreamID 其實有不少屬性（都是 `AudioFileStreamPropertyID`），所以在很多狀況下都會呼叫這個 callback，但我們現在只想要知道資料格式而已，所以只寫了 `kAudioFileStreamProperty_DataFormat` 的相關判斷。

得到資料格式之後，就可以建立 Audio Queue 了。因為我們要建立的是輸出用的 Audio Queue，所以呼叫 `AudioQueueNewOutput` 建立，並且傳入 `KKAudioQueueOutputCallback` 這個 callback function，這個 function 會在 Audio Queue 資料快播完的時候呼叫。

```
OSStatus status = AudioQueueNewOutput(audioStreamBasicDescription,
    KKAudioQueueOutputCallback,
    (__bridge void * _Nullable)(self),
    CFRRunLoopGetCurrent(),
    kCFRunLoopCommonModes, 0, &outputQueue);
```

第四步：收到 parser 分析出的 packet，保存 packet

AudioFileStreamID 把 packet 分析出來之後，會呼叫 `KKAudioFileStreamPacketsCallback`。在這個 callback function 中，我們會收到 audio data 所在的記憶體指標，packet 的數量，以及一連串的 packet description，我們可以從 packet description 中知道每個 packet 相對於傳入的記憶體指標的 offset 與長度。

在這邊，我們把每個 packet 的資料存入 NSData 物件中，然後放在一開始建立的 NSMutableArray 中保存。

第五步：packet 數量夠多的時候，enqueue buffer

我們等到收到足夠大小的 packet 才開始播放，在這邊定義的時間是要超過三秒。因為 packet 數量不夠就開始播放，我們不會聽到連續順暢的音樂，而會是斷斷續續的雜訊，至於播放一秒鐘要多少 packet，可以用一個 packet 有多少 frame 以及一秒鐘需要多少 frame 推算。

要開始播放，就是對 Audio Queue 做 enqueue buffer。在蘋果的 sample code 中，會建立三個 Audio Queue buffer 循環使用，我們這邊的寫法比較偷懶，每次需要 enqueue buffer 的時候，都建立一個新的 Audio Queue Buffer，但是每次進入 `KKAudioQueueOutputCallback` 的時候，都呼叫一次 `AudioQueueFreeBuffer`，把之前使用的 buffer 釋放掉。

在建立 buffer 的時候，我們就要決定 buffer 的大小，在這邊我們透過要播放多少 packet 決定 buffer 的大小。我們在這邊寫成每次要 enqueue 五秒的 buffer，相當於大約 190 個 packet，所以我們跑了一個簡單的迴圈把每個 packet 的大小加總，就是 buffer 的大小，然後把放在 packet 裡頭的 bytes 用 `memcpy` 複製到 buffer 的 `mAudioData` 裡頭。

Enqueue buffer 之後，我們會調整 read head 的位置，記錄已經送出了多少 packet。

第六步：收到 Audio Queue 播放完畢的通知，繼續 enqueue

在前一個 buffer 播放快要完畢的時候（經驗中大概是完畢前一秒鐘左右），我們會收到 `KKAudioQueueOutputCallback`，這時候繼續 enqueue buffer 即可。當我們發現 read head 已經到了跟 packet 的數量一樣多，代表 packet 用完，也就是歌曲播放完畢。

寫完這個 player，只要這樣就可以播放歌曲了：

```
NSString *URL = @"http://zonble.net/MIDI/orz.mp3";
KKSimplePlayer *player = [[KKSimplePlayer alloc] initWithURL:[NSURL URLWithString:URL]];
```

接下來要做的事情

因為這是一個很簡單的 player，所以很多事情沒有做。

要讓這個 player 功能更加完整，我們首先應該要寫一個 protocol，定義這個 player 的 delegate，讓外部的 UI 知道目前 player 的狀況。再來，我們要想辦法解決播放大檔的問題，不該把所有資料都放進記憶體裡。

我們這個 player 也假設網路速度非常順暢，從網路載入資料的速度比播放速度快，如果在寫產品 code，我們還要處理「packet 已經用完，但是網路連線並沒有把資料抓完」這種狀況。

接著需要寫跟播放時間相關的程式。

Audio Queue API 提供 `AudioQueueGetCurrentTime` 與 `AudioQueueDeviceGetCurrentTime` 這兩個 function，可以取得 Audio Queue 的播放時間，一個軟體層某個 Audio Queue 開始了多久，另外一個則是某個 Audio Queue 佔用了硬體多久。其中，`AudioQueueDeviceGetCurrentTime` 會比 `AudioQueueGetCurrentTime` 來得精確。

我們需要注意，這兩個 function 回傳的時間，都是我們呼叫了 `AudioQueueStart` 之後過了多久，與我們現在要播放的歌曲播到哪裡沒有直接關係，比方說，我們只呼叫了 `AudioQueueStart`，但是並沒有 enqueue 任何 buffer，`AudioQueueGetCurrentTime` 與 `AudioQueueDeviceGetCurrentTime` 還是會繼續計算沒有聲音的時間。如果我們遇到網路斷斷續續，聲音時有時無的狀況，Audio Queue 的開啟時間與歌曲播放時間就會對不起來。

現在這邊的這個 player 一次 enqueue 大約五秒的 buffer，在這五秒當中到底是播到第兩秒還是第三秒，就沒有比較好的 API 可以知道。我們在使用 Audio Queue 計算播放時間的時候，大概會用 packet 的位置搭配 `AudioQueueGetCurrentTime` 與 `AudioQueueDeviceGetCurrentTime`，或是搭配使用 NSDate 物件計算，但這麼做總是充滿 work around 的感覺。

在 iOS 上，還要記得處理 Audio Session，這點我們稍晚說明。

我們的 Audio Queue Player 程式碼如下：

`KKSimplePlayer.h`

```
#import <Foundation/Foundation.h>
#import <AudioToolbox/AudioToolbox.h>

@interface KKSsimplePlayer : NSObject
- (instancetype)initWithURL:(NSURL *)inURL;
- (void)play;
- (void)pause;
@property (readonly, getter=isStopped) BOOL stopped;
@end
```

KKSsimplePlayer.m

```
#import "KKSsimplePlayer.h"

static void KKAudioFileStreamPropertyListener(void * inClientData,
    AudioFileStreamID inAudioFileStream,
    AudioFileStreamPropertyID inPropertyID,
    UInt32 * ioFlags);
static void KKAudioFileStreamPacketsCallback(void * inClientData,
    UInt32 inNumberBytes,
    UInt32 inNumberPackets,
    const void * inInputData,
    AudioStreamPacketDescription *inPacketDescriptions);
static void KKAudioQueueOutputCallback(void * inUserData,
    AudioQueueRef inAQ,
    AudioQueueBufferRef inBuffer);
static void KKAudioQueueRunningListener(void * inUserData,
    AudioQueueRef inAQ,
    AudioQueuePropertyID inID);

@interface KKSsimplePlayer ()
{
    NSURLConnection *URLConnection;
    struct {
        BOOL stopped;
        BOOL loaded;
    } playerStatus ;

    AudioFileStreamID audioFileStream;
    AudioQueueRef outputQueue;
    AudioStreamBasicDescription streamDescription;
    NSMutableArray *packets;
    size_t readHead;
}
- (double)packetsPerSecond;
@end

@implementation KKSsimplePlayer

- (void)dealloc
{
    AudioQueueReset(outputQueue);
```

```

    AudioFileStreamClose(audioFileStreamID);
    [URLConnection cancel];
}

- (instancetype)initWithURL:(NSURL *)inURL
{
    self = [super init];
    if (self) {
        playerStatus.stopped = NO;
        packets = [[NSMutableArray alloc] init];

        // 第一步：建立 Audio Parser, 指定 callback, 以及建立 HTTP 連線,
        // 開始下載檔案
        AudioFileStreamOpen((__bridge void * _Nullable)(self),
                           KKAudioFileStreamPropertyListener,
                           KKAudioFileStreamPacketsCallback,
                           kAudioFileMP3Type, &audioFileStreamID);
        URLConnection = [[NSURLConnection alloc] initWithRequest:[NSURLRequest requestWithURL:inURL] delegate:self];
    }
    return self;
}

- (double)packetsPerSecond
{
    if (streamDescription.mFramesPerPacket) {
        return streamDescription.mSampleRate / streamDescription.mFramesPerPacket;
    }

    return 44100.0/1152.0;
}

- (void)play
{
    AudioQueueStart(outputQueue, NULL);
}
- (void)pause
{
    AudioQueuePause(outputQueue);
}

#pragma mark -
#pragma mark NSURLConnectionDelegate

- (void)connection:(NSURLConnection *)connection
    didReceiveResponse:(NSURLResponse *)response
{
    if ([response isKindOfClass:[NSHTTPURLResponse class]]) {
        if ([[NSHTTPURLResponse *)response statusCode] != 200) {
            NSLog(@"%@", [(NSHTTPURLResponse *)response statusCode]);
            [connection cancel];
            playerStatus.stopped = YES;
        }
    }
}

```

```

    }

}

- (void)connection:(NSURLConnection *)connection didReceiveData:(NSData *)data
{
    // 第二步：抓到了部分檔案，就交由 Audio Parser 開始 parse 出 data
    // stream 中的 packet。
    AudioFileStreamParseBytes(audioFileStreamID, (UInt32)[data length], [data bytes], 0);
}

- (void)connectionDidFinishLoading:(NSURLConnection *)connection
{
    NSLog(@"Complete loading data");
    playerStatus.loaded = YES;
}
- (void)connection:(NSURLConnection *)connection didFailWithError:(NSError *)error
{
    NSLog(@"Failed to load data: %@", [error localizedDescription]);
    playerStatus.stopped = YES;
}

#pragma mark -
#pragma mark Audio Parser and Audio Queue callbacks

- (void)_enqueueDataWithPacketsCount:(size_t)inPacketCount
{
    NSLog(@"%@", __PRETTY_FUNCTION__);
    if (!outputQueue) {
        return;
    }

    if (readHead == [packets count]) {
        // 第六步：已經把所有 packet 都播完了，檔案播放結束。
        if (playerStatus.loaded) {
            AudioQueueStop(outputQueue, false);
            playerStatus.stopped = YES;
            return;
        }
    }

    if (readHead + inPacketCount >= [packets count]) {
        inPacketCount = [packets count] - readHead;
    }

    UInt32 totalSize = 0;
    UInt32 index;

    for (index = 0 ; index < inPacketCount ; index++) {
        NSData *packet = packets[index + readHead];
        totalSize += packet.length;
    }

    OSStatus status = 0;
}

```

```

AudioQueueBufferRef buffer;
status = AudioQueueAllocateBuffer(outputQueue, totalSize, &buffer);
assert(status == noErr);
buffer->mAudioDataByteSize = totalSize;
buffer->mUserData = (__bridge void * _Nullable)(self);

AudioStreamPacketDescription *packetDescs = calloc(inPacketCount,
    sizeof(AudioStreamPacketDescription));

totalSize = 0;
for (index = 0 ; index < inPacketCount ; index++) {
    size_t readIndex = index + readHead;
    NSData *packet = packets[readIndex];
    memcpy(buffer->mAudioData + totalSize, packet.bytes, packet.length);

    AudioStreamPacketDescription description;
    description.mStartOffset = totalSize;
    description.mDataByteSize = packet.length;
    description.mVariableFramesInPacket = 0;
    totalSize += packet.length;
    memcpy(&(packetDescs[index]), &description, sizeof(AudioStreamPacketDescription));
}
status = AudioQueueEnqueueBuffer(outputQueue, buffer, (UInt32)inPacketCount, packetDescs);
free(packetDescs);
readHead += inPacketCount;
}

- (void)_createAudioQueueWithAudioStreamDescription:(AudioStreamBasicDescription *)audioStreamBasicDescription
{
    memcpy(&streamDescription, audioStreamBasicDescription, sizeof(AudioStreamBasicDescription));
    OSStatus status = AudioQueueNewOutput(audioStreamBasicDescription,
        KKAudioQueueOutputCallback,
        (__bridge void * _Nullable)(self),
        CFRunLoopGetCurrent(),
        kCFRunLoopCommonModes, 0, &outputQueue);
    assert(status == noErr);
    status = AudioQueueAddPropertyListener(outputQueue,
        kAudioQueueProperty_IsRunning,
        KKAudioQueueRunningListener,
        (__bridge void * _Nullable)(self));
    AudioQueuePrime(outputQueue, 0, NULL);
    AudioQueueStart(outputQueue, NULL);
}

- (void)_storePacketsWithNumberOfBytes:(UInt32)inNumberBytes
    numberOfPackets:(UInt32)inNumberPackets
    inputData:(const void *)inInputData
    packetDescriptions:(AudioStreamPacketDescription *)inPacketDescriptions
{

```

```

    for (int i = 0; i < inNumberPackets; ++i) {
        SInt64 packetStart = inPacketDescriptions[i].mStartOffset;
        UInt32 packetSize = inPacketDescriptions[i].mDataByteSize;
        assert(packetSize > 0);
        NSData *packet = [NSData dataWithBytes:inInputData + packetStart length:packetSize];
        [packets addObject:packet];
    }

    // 第五步，因為 parse 出來的 packets 夠多，緩衝內容夠大，因此開始
    // 播放

    if (readHead == 0 && [packets count] > (int)([self packetsPerSecond] * 3)) {
        AudioQueueStart(outputQueue, NULL);
        [_enqueueWithDataWithPacketsCount: (int)([self packetsPerSecond] * 3)];
    }
}

- (void)_audioQueueDidStart
{
    NSLog(@"Audio Queue did start");
}

- (void)_audioQueueDidStop
{
    NSLog(@"Audio Queue did stop");
    playerStatus.stopped = YES;
}

#pragma mark -
#pragma mark Properties

- (BOOL)isStopped
{
    return playerStatus.stopped;
}

@end

void KKAudioFileStreamPropertyListener(void * inClientData,
    AudioFileStreamID inAudioFileStream,
    AudioFileStreamPropertyID inPropertyID,
    UInt32 * ioFlags)
{
    KKSsimplePlayer *_self = (__bridge KKSsimplePlayer *)inClientData;
    if (inPropertyID == kAudioFileStreamProperty_DataFormat) {
        UInt32 dataSize = 0;
        OSStatus status = 0;
        AudioStreamBasicDescription audioStreamDescription;
        Boolean writable = false;
        status = AudioFileStreamGetPropertyInfo(inAudioFileStream,
            kAudioFileStreamProperty_DataFormat,
            &dataSize, &writable);
    }
}

```

```

        status = AudioFileStreamGetProperty(inAudioFileStream,
            kAudioFileStreamProperty_DataFormat,
            &dataSize, &audioStreamDescription);

        NSLog(@"%@", audioStreamDescription.mSampleRate);
        NSLog(@"%@", audioStreamDescription.mFormatID);
        NSLog(@"%@", audioStreamDescription.mFormatFlags);
        NSLog(@"%@", audioStreamDescription.mBytesPerPacket);
        NSLog(@"%@", audioStreamDescription.mFramesPerPacket);
        NSLog(@"%@", audioStreamDescription.mBytesPerFrame);
        NSLog(@"%@", audioStreamDescription.mChannelsPerFrame);
        NSLog(@"%@", audioStreamDescription.mBitsPerChannel);
        NSLog(@"%@", audioStreamDescription.mReserved);

        // 第三步： Audio Parser 成功 parse 出 audio 檔案格式，我們根據
        // 檔案格式資訊，建立 Audio Queue，同時監聽 Audio Queue 是否正
        // 在執行

        [self _createAudioQueueWithAudioStreamDescription:&audioStreamDescription];
    }

}

void KKAudioFileStreamPacketsCallback(void * inClientData,
    UInt32 inNumberBytes,
    UInt32 inNumberPackets,
    const void * inInputData,
    AudioStreamPacketDescription *inPacketDescriptions)
{
    // 第四步： Audio Parser 成功 parse 出 packets，我們將這些資料儲存
    // 起來

    KKSsimplePlayer *self = (__bridge KKSsimplePlayer *)inClientData;
    [self _storePacketsWithNumberOfBytes:inNumberBytes
        numberOfPackets:inNumberPackets
        inputData:inInputData
        packetDescriptions:inPacketDescriptions];
}

static void KKAudioQueueOutputCallback(void * inUserData,
    AudioQueueRef inAQ, AudioQueueBufferRef inBuffer)
{
    AudioQueueFreeBuffer(inAQ, inBuffer);
    KKSsimplePlayer *self = (__bridge KKSsimplePlayer *)inUserData;
    [self _enqueueDataWithPacketsCount:(int)([self packetsPerSecond] * 5)];
}

static void KKAudioQueueRunningListener(void * inUserData,
    AudioQueueRef inAQ, AudioQueuePropertyID inID)
{
    KKSsimplePlayer *self = (__bridge KKSsimplePlayer *)inUserData;
    UInt32 dataSize;
    OSStatus status = 0;
    status = AudioQueueGetPropertySize(inAQ, inID, &dataSize);
}

```

```

if (inID == kAudioQueueProperty_IsRunning) {
    UInt32 running;
    status = AudioQueueGetProperty(inAQ, inID, &running, &dataSize);
    running ? [self _audioQueueDidStart] : [self _audioQueueDidStop];
}
}

```

如果用 Swift 寫，則大概像這樣（需要 Swift 2.0 以上）：

KKSimplePlayer.swift

```

import Foundation
import AudioToolbox

class KKSimplePlayer: NSObject {
    var URL: NSURL
    var URLSession: NSURLSession!
    var packets = [NSData]()
    var audioFileStreamID: AudioFileStreamID = nil
    var outputQueue: AudioQueueRef = nil
    var streamDescription: AudioStreamBasicDescription?
    var readHead: Int = 0

    var loaded = false
    var stopped = false

    init(URL: NSURL) {
        self.URL = URL
        super.init()
        let selfPointer = unsafeBitCast(self, UnsafeMutablePointer<Void>.self)
        AudioFileStreamOpen(selfPointer,
                            KKAudioFileStreamPropertyListener,
                            KKAudioFileStreamPacketsCallback,
                            kAudioFileMP3Type, &self.audioFileStreamID)
        let configuration = NSURLSessionConfiguration.defaultSessionConfiguration()
        self.URLSession = NSURLSession(configuration: configuration, delegate: self, delegateQueue: nil)
        let task = self.URLSession.dataTaskWithURL(URL)
        task.resume()
    }

    deinit {
        if self.outputQueue != nil {
            AudioQueueReset(outputQueue)
        }
        AudioFileStreamClose(audioFileStreamID)
    }

    func play() {
        if self.outputQueue != nil {
            AudioQueueStart(outputQueue, nil)
        }
    }
}

```

```

    }

    func pause() {
        if self.outputQueue != nil {
            AudioQueuePause(outputQueue)
        }
    }

    private func parseData(data: NSData) {
        AudioFileStreamParseBytes(self.audioFileStreamID, UInt32(data.length), data.bytes
, AudioFileStreamParseFlags(rawValue: 0))
    }

    private func createAudioQueue(audioStreamDescription: AudioStreamBasicDescription) {
        var audioStreamDescription = audioStreamDescription
        self.streamDescription = audioStreamDescription
        var status: OSStatus = 0
        let selfPointer = unsafeBitCast(self, UnsafeMutablePointer<Void>.self)
        status = AudioQueueNewOutput(&audioStreamDescription, KKAudioQueueOutputCallback,
selfPointer, CFRunLoopGetCurrent(), kCFRunLoopCommonModes, 0, &self.outputQueue)
        assert(noErr == status)
        status = AudioQueueAddPropertyListener(self.outputQueue, kAudioQueueProperty_IsRu
nning, KKAudioQueueRunningListener, selfPointer)
        assert(noErr == status)
        AudioQueuePrime(self.outputQueue, 0, nil)
        AudioQueueStart(self.outputQueue, nil)
    }

    private func storePackets(numberOfPackets: UInt32, numberofBytes: UInt32, data: Unsa
ePointer<Void>, packetDescription: UnsafeMutablePointer<AudioStreamPacketDescription>) {
        for i in 0 ..< Int(numberOfPackets) {
            let packetStart = packetDescription[i].mStartOffset
            let packetSize = packetDescription[i].mDataByteSize
            let packetData = NSData(bytes: data.advancedBy(Int(packetStart)), length: Int
(packetSize))
            self.packets.append(packetData)
        }

        if readHead == 0 && Double(packets.count) > self.packetsPerSecond * 3 {
            AudioQueueStart(self.outputQueue, nil)
            self.enqueueDataWithPacketsCount(Int(self.packetsPerSecond * 3))
        }
    }

    private func enqueueDataWithPacketsCount(packetCount: Int) {
        if self.outputQueue == nil {
            return
        }

        var packetCount = packetCount
        if readHead + packetCount > packets.count {
            packetCount = packets.count - readHead
        }
    }
}

```

```

        let totalSize = packets[readHead ..< readHead + packetCount].reduce(0, combine: {
$0 + $1.length })
        var status: OSStatus = 0
        var buffer: AudioQueueBufferRef = nil
        status = AudioQueueAllocateBuffer(outputQueue, UInt32(totalSize), &buffer)
        assert(noErr == status)
        buffer.memory.mAudioDataByteSize = UInt32(totalSize)
        let selfPointer = unsafeBitCast(self, UnsafeMutablePointer<Void>.self)
        buffer.memory.mUserData = selfPointer

        var copiedSize = 0
        var packetDescs = [AudioStreamPacketDescription]()
        for i in 0 ..< packetCount {
            let readIndex = readHead + i
            let packetData = packets[readIndex]
            memcpy(buffer.memory.mAudioData.advancedBy(copiedSize), packetData.bytes, packetData.length)
            let description = AudioStreamPacketDescription(mStartOffset: Int64(copiedSize),
), mVariableFramesInPacket: 0, mDataByteSize: UInt32(packetData.length))
            packetDescs.append(description)
            copiedSize += packetData.length
        }
        status = AudioQueueEnqueueBuffer(outputQueue, buffer, UInt32(packetCount), packetDescs);
        readHead += packetCount
    }
}

extension KKSsimplePlayer: NSURLSessionDelegate {
    func URLSession(session: NSURLSession, dataTask: NSURLSessionDataTask, didReceiveData
data: NSData) {
        self.parseData(data)
    }
}

func KKAudioFileStreamPropertyListener(clientData: UnsafeMutablePointer<Void>, audioFileS
tream: AudioFileStreamID, propertyID: AudioFileStreamPropertyID, ioFlag: UnsafeMutablePoi
nter<AudioFileStreamPropertyFlags>) {
    let this = Unmanaged<KKSsimplePlayer>.fromOpaque(COpaquePointer(clientData)).takeUnret
ainedValue()
    if propertyID == kAudioFileStreamProperty_DataFormat {
        var status: OSStatus = 0
        var dataSize: UInt32 = 0
        var writable: DarwinBoolean = false
        status = AudioFileStreamGetPropertyInfo(audioFileStream, kAudioFileStreamProperty_
_DataFormat, &dataSize, &writable)
        assert(noErr == status)
        var audioStreamDescription: AudioStreamBasicDescription = AudioStreamBasicDescrip
tion()
        status = AudioFileStreamGetProperty(audioFileStream, kAudioFileStreamProperty_Dat
aFormat, &dataSize, &audioStreamDescription)
        assert(noErr == status)
    }
}

```

```

        dispatch_async(dispatch_get_main_queue()) {
            this.createAudioQueue(audioStreamDescription)
        }
    }

func KKAudioFileStreamPacketsCallback(clientData: UnsafeMutablePointer<Void>, numberBytes : UInt32, numberPackets: UInt32, ioData: UnsafePointer<Void>, packetDescription: UnsafeMutablePointer<AudioStreamPacketDescription>) {
    let this = Unmanaged<KKSimplePlayer>.fromOpaque(COpaquePointer(clientData)).takeUnretainedValue()
    this.storePackets(numberPackets, numberOfRowsInSection: numberBytes, data: ioData, packetDescription: packetDescription)
}

func KKAudioQueueOutputCallback(clientData: UnsafeMutablePointer<Void>, AQ: AudioQueueRef , buffer: AudioQueueBufferRef) {
    let this = Unmanaged<KKSimplePlayer>.fromOpaque(COpaquePointer(clientData)).takeUnretainedValue()
    AudioQueueFreeBuffer(AQ, buffer)
    this.enqueueDataWithPacketsCount(Int(this.packetsPerSecond * 5))
}

func KKAudioQueueRunningListener(clientData: UnsafeMutablePointer<Void>, AQ: AudioQueueRef, propertyID: AudioQueuePropertyID) {
    let this = Unmanaged<KKSimplePlayer>.fromOpaque(COpaquePointer(clientData)).takeUnretainedValue()
    var status: OSStatus = 0
    var dataSize: UInt32 = 0
    status = AudioQueueGetPropertySize(AQ, propertyID, &dataSize);
    assert(noErr == status)
    if propertyID == kAudioQueueProperty_IsRunning {
        var running: UInt32 = 0
        status = AudioQueueGetProperty(AQ, propertyID, &running, &dataSize)
        this.stopped = running == 0
    }
}

```

使用 Audio Unit Processing Graph 開發播放軟體

如果我們選擇要使用 Audio Unit Processing Graph API 開發播放網路串流的Player，前半部的工作跟一個 Audio Queue Player 差不多，還是要先發送網路連線抓取檔案，建立 parser，並且讓 parser parse 出 packet。

但是後半部就變得不一樣，Audio Queue 會幫我們把 packet 從原來的格式轉成 LPCM 格式，所以我們只要建好 Audio Queue Buffer，再 enqueue 到 Audio Queue 中；但如果我們使用 Audio Unit Processing Graph API，我們就要自己透過 converter 將 MP3 等格式轉換成 LPCM 格式播放。

Audio Queue API 與 Audio Unit Processing Graph API 的行為也不太一樣，使用 Audio Queue API 的時候，我們會主動把 buffer 送到 Audio Queue 中，

在以下這個範例中，我們先不進入如何使用 AUGraph，只使用了 Remote IO—如果我們不需要一些複雜的播放效果，像 EQ 等化器或混音，只要單獨有 Remote IO 其實就可以播放。

建立 Remote IO 與設定 Render Callback

由於建立 Audio Queue 的時候需要傳入 audio format，所以我們是在 parser 取得了 audio format 之後，才建立 Audio Queue。不過，使用 Audio Unit Processing Graph API 開發播放軟體時，我們是用 audio format 建立 converter，所以我們可以在建立 player 時，就先建立好 Remote IO 的 Audio Unit，並且對 Remote IO 的 Audio Unit 設定好 render callback。

建立 Remote IO 的 Audio Unit 的時候，我們會先建立一個用來表示 component 條件的 AudioComponentDescription，將 componentType 設定成 kAudioUnitType_Output，代表我們要的是一個輸出用的 node，然後 subtype 設定成 kAudioUnitSubType_RemoteIO。接著使用 AudioComponentFindNext 找到符合的 node，然後從這個 node 中拿出這個 node 的操作介面，也就是 Audio Unit。

```
AudioComponentDescription outputUnitDescription;
bzero(&outputUnitDescription, sizeof(AudioComponentDescription));
outputUnitDescription.componentType = kAudioUnitType_Output;
outputUnitDescription.componentSubType = kAudioUnitSubType_RemoteIO;
outputUnitDescription.componentManufacturer = kAudioUnitManufacturer_Apple;
outputUnitDescription.componentFlags = 0;
outputUnitDescription.componentFlagsMask = 0;

AudioComponent outputComponent = AudioComponentFindNext(NULL, &outputUnitDescription);
OSStatus status = AudioComponentInstanceNew(outputComponent, &audioUnit);
```

接著是從 Audio Unit 設定輸入格式，我們接下來建立 converter 的時候，也會將 MP3 轉成這種格式的 LPCM，然後送到 Remote IO node。

```
AudioStreamBasicDescription audioFormat = KKSignedIntLinearPCMStreamDescription();
AudioUnitSetProperty(audioUnit,
kAudioUnitProperty_StreamFormat,
kAudioUnitScope_Input, 0,
&audioFormat, sizeof(audioFormat));
```

我們來看一下輸入格式的部份。前面提到，即使都叫做 LPCM，裡頭也可能有很多不同的格式，可能使用整數、也可能使用浮點數表示。

我們在這邊使用的是 16 位元整數，而每個 frame 裡頭有左右兩個聲道，因此左右兩個聲道分別有兩個 byte，一個 frame 就是兩個 byte 加起來變成四個bytes，因此 mBytesPerFrame 設定為 4。在 LPCM 格式中，一個 packet 只有一個 frame，所以每個 packet 也是四個 bytes。

```
AudioStreamBasicDescription destFormat;
bzero(&destFormat, sizeof(AudioStreamBasicDescription));
destFormat.mSampleRate = 44100.0;
destFormat.mFormatID = kAudioFormatLinearPCM;
destFormat.mFormatFlags = kLinearPCMFormatFlagIsSignedInteger;
destFormat.mFramesPerPacket = 1;
destFormat.mBytesPerPacket = 4;
destFormat.mBytesPerFrame = 4;
destFormat.mChannelsPerFrame = 2;
destFormat.mBitsPerChannel = 16;
destFormat.mReserved = 0;
```

接著就是設定 render callback。我們把 render callback 設定成 `KKPlayerAURenderCallback` 這個 function，之後只要呼叫 `AudioOutputUnitStart`，就會在專屬的 audio render thread 中呼叫這個callback function。

```
AURenderCallbackStruct callbackStruct;
callbackStruct.inputProcRefCon = (__bridge void *) (self);
callbackStruct.inputProc = KKPlayerAURenderCallback;
status = AudioUnitSetProperty(audioUnit,
    kAudioUnitProperty_SetRenderCallback,
    kAudioUnitScope_Global,
    0, // 代表這個 callback function 繩在 bus 0 上
    &callbackStruct, sizeof(callbackStruct));
```

建立 Converter

等到 Audio File Stream ID parse 出遠端檔案的格式後，我們就可以建立converter 了。在建立的過程中，要傳入輸入給 converter 的格式，以及converter 應該要轉出的格式。

```
AudioStreamBasicDescription destFormat = KKSignedIntLinearPCMStreamDescription();
AudioConverterNew(&streamDescription, &destFormat, &converter);
```

實作 Render Callback

在 render callback function 裡頭要做的事情，就是把已經收到的 packet 透過 converter 轉換成 LPCM 格式之後回傳。

我們的 render callback function 像這樣：

```
OSStatus KKPlayerAURenderCallback(void *userData,
    AudioUnitRenderActionFlags *ioActionFlags,
    const AudioTimeStamp *inTimeStamp,
```

```
UInt32 inBusNumber,
UInt32 inNumberFrames,
AudioBufferList *ioData);
```

在這個 function 中會傳入幾個參數，我們最關心的是 inNumberFrames 與 ioData，inNumberFrames 就是 Remote IO 現在要跟我們要求多少 frame 的 audio 資料，我們會根據這個數量，從 MP3 或其他原始格式中轉出多少 frame 的 LPCM，然後將資料填入到 ioData 中。

至於其他幾個參數，像 userData，就是我們在建立 callback function 的時候透過 AURenderCallbackStruct 的 inputProcRefCon 傳入的物件指標，因為我們的 packet 是放在我們的 player 物件中，自然要想辦法讓 callback function 可以有 reference 找到我們的 player。

ioActionFlags 可以讓我們對 Remote IO Unit 做一些額外的操作，假如我們並沒有 packet 可以給 Remote IO 播放，或是發生了其他錯誤，這時候我們就會在 *ioActionFlags |= kAudioUnitRenderAction_OutputIsSilence；這行裡頭告訴告訴 Remote IO 應該靜音。至於 inTimeStamp 則代表這個 callback 是在什麼時間被呼叫的。

inBusNumber 代表這個 callback function 被連接到 Remote IO 的哪個 bus，前面提到，Remote IO 的 bus 0 是輸出，用來播放，bus 1 則用來錄音，我們現在是處理播放，自然會連接到 bus 0。

如果我們不是對 Remote IO，而是對像 mixer 之類的 node 設定 callback，也可能會設定在 bus 0 之外的 bus，像一個 mixer 在 bus 0 或 bus 1 分別設定了兩個 callback，就可以對兩個輸入來源做混音處理。

實作 Conveter 的 Fill Callback

AudioConverterRef 有三個可以用來轉換格式的 C function：

- AudioConverterConvertBuffer
- AudioConverterConvertComplexBuffer
- AudioConverterFillComplexBuffer

雖然有三個 function，但實際上可以用的只有 AudioConverterFillComplexBuffer；

AudioConverterConvertBuffer 與 AudioConverterConvertComplexBuffer 都只能夠處理不同的 LPCM 格式之間的轉換，像是把整數的 LPCM 轉成浮點的 LPCM。我們想要把 MP3 或 AAC 格式轉成 LPCM，只能夠呼叫 AudioConverterFillComplexBuffer。

呼叫 AudioConverterFillComplexBuffer 的時候，必須要傳入一個 converter callback function，我們叫他 filler，我們在這邊傳入了 KKPlayerConverterFiller，並且要傳入一個 AudioBufferList。在 filler function 中，我們把 packet 一個一個餵給 converter 讓 converter 轉換，轉出的資料就會填入到傳入的 AudioBufferList；轉換完畢之後，AudioBufferList 裡頭就會是轉好的 LPCM 檔案。

接著，我們就可以把 AudioBufferList 中的資料，塞入 KKPlayerAURenderCallback 傳入的 ioData 中。如果沒有資料，就像上面說的，透過 ioActionFlags 暫停播放。

KKSsimpleAUPlayer.h

```
@import Foundation;
#import AudioToolbox;

@interface KKSsimpleAUPlayer : NSObject
- (instancetype)initWithURL:(NSURL *)inURL;
```

```

- (void)play;
- (void)pause;
@property (readonly, getter=isPlaying) BOOL playing;
@end

```

KKSsimpleAUPlayer.m

```

#import "KKSsimpleAUPlayer.h"

static void KKAudioFileStreamPropertyListener(void* inClientData,
    AudioFileStreamID inAudioFileStream,
    AudioFileStreamPropertyID inPropertyID,
    UInt32* ioFlags);
static void KKAudioFileStreamPacketsCallback(void* inClientData,
    UInt32 inNumberBytes,
    UInt32 inNumberPackets,
    const void* inInputData,
    AudioStreamPacketDescription *inPacketDescriptions);
static OSStatus KKPlayerAURenderCallback(void *userData,
    AudioUnitRenderActionFlags *ioActionFlags,
    const AudioTimeStamp *inTimeStamp,
    UInt32 inBusNumber,
    UInt32 inNumberFrames,
    AudioBufferList *ioData);
static OSStatus KKPlayerConverterFiller(AudioConverterRef inAudioConverter,
    UInt32* ioNumberDataPackets,
    AudioBufferList* ioData,
    AudioStreamPacketDescription** outDataPacketDescription,
    void* inUserData);

static const OSStatus KKAudioConverterCallbackErr_NoData = 'kknd';

static AudioStreamBasicDescription KKSignedIntLinearPCMStreamDescription();

@interface KKSsimpleAUPlayer () <NSURLConnectionDelegate>
{
    NSURLConnection *URLConnection;
    struct {
        BOOL stopped;
        BOOL loaded;
    } playerStatus ;

    AudioComponentInstance audioUnit;

    AudioFileStreamID audioFileStreamID;
    AudioStreamBasicDescription streamDescription;
    AudioConverterRef converter;
    AudioBufferList *renderBufferList;
    UInt32 renderBufferSize;

    NSMutableArray *packets;
    size_t readHead;
}

```

```

}

- (double)packetsPerSecond;
@end

AudioStreamBasicDescription KKSignedIntLinearPCMStreamDescription()
{
    AudioStreamBasicDescription destFormat;
    bzero(&destFormat, sizeof(AudioStreamBasicDescription));
    destFormat.mSampleRate = 44100.0;
    destFormat.mFormatID = kAudioFormatLinearPCM;
    destFormat.mFormatFlags = kLinearPCMFormatFlagIsSignedInteger;
    destFormat.mFramesPerPacket = 1;
    destFormat.mBytesPerPacket = 4;
    destFormat.mBytesPerFrame = 4;
    destFormat.mChannelsPerFrame = 2;
    destFormat.mBitsPerChannel = 16;
    destFormat.mReserved = 0;
    return destFormat;
}

@implementation KKSimpleAUPlayer

- (void)dealloc
{
    AudioFileStreamClose(audioFileStreamID);
    AudioConverterDispose(converter);
    free(renderBufferList->mBuffers[0].mData);
    free(renderBufferList);
    renderBufferList = NULL;

    [URLConnection cancel];
}

- (void)buildOutputUnit
{
    // 建立 remote IO node
    AudioComponentDescription outputUnitDescription;
    bzero(&outputUnitDescription, sizeof(AudioComponentDescription));
    outputUnitDescription.componentType = kAudioUnitType_Output;
    outputUnitDescription.componentSubType = kAudioUnitSubType_RemoteIO;
    outputUnitDescription.componentManufacturer = kAudioUnitManufacturer_Apple;
    outputUnitDescription.componentFlags = 0;
    outputUnitDescription.componentFlagsMask = 0;

    AudioComponent outputComponent = AudioComponentFindNext(NULL, &outputUnitDescription);
;
    OSStatus status = AudioComponentInstanceNew(outputComponent, &audioUnit);
    NSAssert(noErr == status, @"Must be no error.");

    // 設定 remote IO node 的輸入格式
    AudioStreamBasicDescription audioFormat = KKSignedIntLinearPCMStreamDescription();
    AudioUnitSetProperty(audioUnit,
    kAudioUnitProperty_StreamFormat,

```

```

        kAudioUnitScope_Input, 0,
        &audioFormat, sizeof(audioFormat));

    // 設定 render callback
    AURenderCallbackStruct callbackStruct;
    callbackStruct.inputProcRefCon = (__bridge void *)self;
    callbackStruct.inputProc = KKPlayerAURenderCallback;
    status = AudioUnitSetProperty(audioUnit,
        kAudioUnitProperty_SetRenderCallback,
        kAudioUnitScope_Global, 0,
        &callbackStruct, sizeof(callbackStruct));
    NSAssert(noErr == status, @"Must be no error.");

    // 建立 converter 要使用的 buffer list
    UInt32 bufferSize = 4096 * 4;
    renderBufferSize = bufferSize;
    renderBufferList = (AudioBufferList *)calloc(1, sizeof(UInt32) + sizeof(AudioBuffer))
    ;
    renderBufferList->mNumberBuffers = 1;
    renderBufferList->mBuffers[0].mNumberChannels = 2;
    renderBufferList->mBuffers[0].mDataByteSize = bufferSize;
    renderBufferList->mBuffers[0].mData = calloc(1, bufferSize);
}

- (instancetype)initWithURL:(NSURL *)inURL
{
    self = [super init];
    if (self) {
        [self buildOutputUnit];

        playerStatus.stopped = NO;
        packets = [[NSMutableArray alloc] init];

        // 第一步：建立 Audio Parser, 指定 callback, 以及建立 HTTP 連線,
        // 開始下載檔案
        AudioFileStreamOpen((__bridge void *)(self),
            KKAUDIOFILESTREAMPROPERTYLISTENER,
            KKAUDIOFILESTREAMPACKETSCALLBACK,
            kAudioFileMP3Type, &audioFileStreamID);
        URLConnection = [[NSURLConnection alloc] initWithRequest:[NSURLRequest requestWithURL:inURL] delegate:self];
        playerStatus.stopped = YES;
    }
    return self;
}

- (double)packetsPerSecond
{
    if (streamDescription.mFramesPerPacket) {
        return streamDescription.mSampleRate / streamDescription.mFramesPerPacket;
    }
    return 44100.0/1152.0;
}

```

```

- (void)play
{
    OSStatus status = AudioOutputUnitStart(audioUnit);
    NSAssert(noErr == status, @"AudioOutputUnitStart, error: %ld", (signed long)status);
}
- (void)pause
{
    OSStatus status = AudioOutputUnitStop(audioUnit);
    NSAssert(noErr == status, @"AudioOutputUnitStop, error: %ld", (signed long)status);
}

#pragma mark -
#pragma mark NSURLConnectionDelegate

- (void)connection:(NSURLConnection *)connection didReceiveResponse:(NSURLResponse *)response
{
    if ([response isKindOfClass:[NSHTTPURLResponse class]]) {
        if ([(NSHTTPURLResponse *)response statusCode] != 200) {
            NSLog(@"HTTP code:%ld", [(NSHTTPURLResponse *)response statusCode]);
            [connection cancel];
            playerStatus.stopped = YES;
        }
    }
}

- (void)connection:(NSURLConnection *)connection didReceiveData:(NSData *)data
{
    // 第二步：抓到了部分檔案，就交由 Audio Parser 開始 parse 出 data
    // stream 中的 packet。
    AudioFileStreamParseBytes(audioFileStreamID, (UInt32)[data length], [data bytes], 0);
}

- (void)connectionDidFinishLoading:(NSURLConnection *)connection
{
    NSLog(@"Complete loading data");
    playerStatus.loaded = YES;
}

- (void)connection:(NSURLConnection *)connection didFailWithError:(NSError *)error
{
    NSLog(@"Failed to load data: %@", [error localizedDescription]);
    [self pause];
}

#pragma mark -
#pragma mark Audio Parser and Audio Queue callbacks

- (void)_createAudioConverterWithAudioStreamDescription:(AudioStreamBasicDescription *)audioStreamBasicDescription
{
    memcpy(&streamDescription, audioStreamBasicDescription, sizeof(AudioStreamBasicDescri

```

```

ption));
    AudioStreamBasicDescription destFormat = KKSignedIntLinearPCMStreamDescription();
    AudioConverterNew(&streamDescription, &destFormat, &converter);
}

- (void)_storePacketsWithNumberOfBytes:(UInt32)inNumberBytes
    numberOfRowsInSection:(UInt32)inNumberPackets
    inputData:(const void *)inInputData
    packetDescriptions:(AudioStreamPacketDescription *)inPacketDescriptions
{
    for (int i = 0; i < inNumberPackets; ++i) {
        SInt64 packetStart = inPacketDescriptions[i].mStartOffset;
        UInt32 packetSize = inPacketDescriptions[i].mDataByteSize;
        assert(packetSize > 0);
        NSData *packet = [NSData dataWithBytes:inInputData + packetStart length:packetSize];
        [packets addObject:packet];
    }

    // 第五步，因為 parse 出來的 packets 夠多，緩衝內容夠大，因此開始
    // 播放

    if (readHead == 0 && [packets count] > (int)([self packetsPerSecond] * 3)) {
        if (playerStatus.stopped) {
            [self play];
        }
    }
}

#pragma mark -
#pragma mark Properties

- (BOOL)isStopped
{
    return playerStatus.stopped;
}

- (OSStatus)callbackWithNumberOfFrames:(UInt32)inNumberOfFrames
    ioData:(AudioBufferList *)inIoData busNumber:(UInt32)inBusNumber
{
    @synchronized(self) {
        if (readHead < [packets count]) {
            @autoreleasepool {
                UInt32 packetSize = inNumberOfFrames;
                // 第七步： Remote IO node 的 render callback 中，呼叫 converter 將 packet
                轉成 LPCM
                OSStatus status =
                AudioConverterFillComplexBuffer(converter,
                    KKPlayerConverterFiller,
                    (__bridge void *)(self),
                    &packetSize, renderBufferList, NULL);
                if (noErr != status && KKAUDIOConverterCallbackErr_NoData != status) {
                    [self pause];
                }
            }
        }
    }
}

```

```

        return -1;
    }
    else if (!packetSize) {
        inIoData->mNumberBuffers = 0;
    }
    else {
        // 在這邊改變 renderBufferList->mBuffers[0].mData
        // 可以產生各種效果
        inIoData->mNumberBuffers = 1;
        inIoData->mBuffers[0].mNumberChannels = 2;
        inIoData->mBuffers[0].mDataByteSize = renderBufferList->mBuffers[0].m
DataByteSize;
        inIoData->mBuffers[0].mData = renderBufferList->mBuffers[0].mData;
        renderBufferList->mBuffers[0].mDataByteSize = renderBufferSize;
    }
}
else {
    inIoData->mNumberBuffers = 0;
    return -1;
}
}

return noErr;
}

- (OSStatus)_fillConverterBufferWithBufferlist:(AudioBufferList *)ioData
packetDescription:(AudioStreamPacketDescription**) outDataPacketDescription
{
    static AudioStreamPacketDescription aspdesc;

    if (readHead >= [packets count]) {
        return KKAUDIOConverterCallbackErr_NoData;
    }

    ioData->mNumberBuffers = 1;
    NSData *packet = packets[readHead];
    void const *data = [packet bytes];
    UInt32 length = (UInt32)[packet length];
    ioData->mBuffers[0].mData = (void *)data;
    ioData->mBuffers[0].mDataByteSize = length;

    *outDataPacketDescription = &aspdesc;
    aspdesc.mDataByteSize = length;
    aspdesc.mStartOffset = 0;
    aspdesc.mVariableFramesInPacket = 1;

    readHead++;
    return 0;
}

@end

```

```

void KKAudioFileStreamPropertyListener(void * inClientData,
    AudioFileStreamID inAudioFileStream,
    AudioFileStreamPropertyID inPropertyID,
    UInt32 * ioFlags)
{
    KKSsimpleAUPlayer *self = (__bridge KKSsimpleAUPlayer *)inClientData;
    if (inPropertyID == kAudioFileStreamProperty_DataFormat) {
        UInt32 dataSize = 0;
        OSStatus status = 0;
        AudioStreamBasicDescription audioStreamDescription;
        Boolean writable = false;
        status = AudioFileStreamGetPropertyInfo(inAudioFileStream,
            kAudioFileStreamProperty_DataFormat, &dataSize, &writable);
        status = AudioFileStreamGetProperty(inAudioFileStream,
            kAudioFileStreamProperty_DataFormat, &dataSize, &audioStreamDescription);

        NSLog(@"%@", audioStreamDescription.mSampleRate);
        NSLog(@"%@", audioStreamDescription.mFormatID);
        NSLog(@"%@", audioStreamDescription.mFormatFlags);
        NSLog(@"%@", audioStreamDescription.mBytesPerPacket);
        NSLog(@"%@", audioStreamDescription.mFramesPerPacket);
        NSLog(@"%@", audioStreamDescription.mBytesPerFrame);
        NSLog(@"%@", audioStreamDescription.mChannelsPerFrame);
        NSLog(@"%@", audioStreamDescription.mBitsPerChannel);
        NSLog(@"%@", audioStreamDescription.mReserved);

        // 第三步： Audio Parser 成功 parse 出 audio 檔案格式，我們根據
        // 檔案格式資訊，建立 converter

        [self _createAudioConverterWithAudioStreamDescription:&audioStreamDescription];
    }
}

void KKAudioFileStreamPacketsCallback(void* inClientData,
    UInt32 inNumberBytes,
    UInt32 inNumberPackets,
    const void* inInputData,
    AudioStreamPacketDescription* inPacketDescriptions)
{
    // 第四步： Audio Parser 成功 parse 出 packets，我們將這些資料儲存
    // 起來

    KKSsimpleAUPlayer *self = (__bridge KKSsimpleAUPlayer *)inClientData;
    [self _storePacketsWithNumberOfBytes:inNumberBytes
        numberOfPackets:inNumberPackets
        inputData:inInputData
        packetDescriptions:inPacketDescriptions];
}

OSStatus KKPlayerAURenderCallback(void *userData,
    AudioUnitRenderActionFlags *ioActionFlags,
    const AudioTimeStamp *inTimeStamp,

```

```

UInt32 inBusNumber,
UInt32 inNumberFrames,
AudioBufferList *ioData)
{
    // 第六步： Remote IO node 的 render callback
    KKSsimpleAUPlayer *self = (__bridge KKSsimpleAUPlayer *)userData;
    OSStatus status = [self callbackWithNumberOfFrames:inNumberFrames
        ioData:ioData busNumber:inBusNumber];
    if (status != noErr) {
        ioData->mNumberBuffers = 0;
        *ioActionFlags |= kAudioUnitRenderAction_OutputIsSilence;
    }
    return status;
}

OSStatus KKPlayerConverterFiller (AudioConverterRef inAudioConverter,
    UInt32* ioNumberDataPackets,
    AudioBufferList* ioData,
    AudioStreamPacketDescription** outDataPacketDescription,
    void* inUserData)
{
    // 第八步： AudioConverterFillComplexBuffer 的 callback
    KKSsimpleAUPlayer *self = (__bridge KKSsimpleAUPlayer *)inUserData;
    *ioNumberDataPackets = 0;
    OSStatus result = [self _fillConverterBufferWithBufferlist:ioData
        packetDescription:outDataPacketDescription];
    if (result == noErr) {
        *ioNumberDataPackets = 1;
    }
    return result;
}

```

還有什麼要做的？

這個 player 和我們在前一章的 Audio Queue player 一樣，把 MP3 資料全部放在記憶體中，一樣會有太佔用記憶體的問題，在遠端檔案太大的時候，我們應該要把一部分資料放到暫存檔中。不過，如果我們用了暫存檔，也要記得盡量從暫存檔中一次多拿一些資料，不要每個 packet 一點一點的做 fread，因為其實 iPhone 的 storage 很慢，如果我們一邊在讀取檔案，一邊有任何的 App 在寫入檔案，就可能影響讀取的速度，而讓聲音聽起來斷斷續續。

另外，我們也沒有實作處理播放時間的部份，這個 player 距離產品 code 還有一段距離。

總之，在這個版本的 player 中，我們已經可以拿到 LPCM 資料了，我們可以用這些資料做出許多應用，像是畫出頻譜圖，或是在餵給硬體之前改變LPCM 資料，直接改變要輸出的聲音。

在 AUGraph 中串接 AudioUnit

前一節當中示範了如何只用 Remote IO 呼叫 Audio Unit Processing Graph API，在這一節中，整個 player 的架構基本上是差不多的，不過，我們從建立 Remote IO component 換成 AUGraph，並且在 AUGraph 當中串接多個 node。在這個範例中，我們建立了三個 node：mixer、EQ 等化器，然後輸出到 Remote IO 上。

所以我們有以下成員變數：

```
AUGraph audioGraph; // audio graph
AudioUnit mixerUnit; // mixer
AudioUnit EQUnit; // EQ
AudioUnit outputUnit; // remote IO
```

建立 Audio Graph

建立 Audio Graph 的過程很簡單，就是呼叫 `NewAUGraph` 與 `AUGraphOpen`，等到晚一點，我們把 Audio Graph 的內容設定完畢之後，我們還要呼叫 `AUGraphInitialize`。

```
NewAUGraph(&audioGraph);
AUGraphOpen(audioGraph);
```

建立與串接 node

我們每次要建立一個 node 的時候，都要傳入指定的 `AudioComponentDescription`，但是設定 `AudioComponentDescription` 的 code 非常冗長，而且有大量的重複。因此，不少人在寫這段 code 的時候，會想要把每個 node 再包裝一層變成 Objective-C 物件，像 [DDCoreAudio](#)、以及更有名的 [The Amazing Audio Engine](#)，就是這樣的專案。

蘋果在 iOS 7 推出的 `AVAudioEngine` 也像是一個 AUGraph 的 Objective-C wrapper，`AVAudioEngine` 這個 class 本身就像是 AUGraph，而我們可以透過呼叫 `-attachNode:` 增加 `AVAudioNode` 物件，然後用 `-connect:to:format:` 這類的 method 串接 `AVAudioNode`。

要在 Audio Graph 中建立新的 node，方法是呼叫 `AUGraphAddNode`，然後可以用 `AUGraphConnectNodeInput` 串接。這段建立與串接實在看起來很可怕：

```
// 建立 mixer node
AudioComponentDescription mixerUnitDescription;
mixerUnitDescription.componentType= kAudioUnitType_Mixer;
mixerUnitDescription.componentSubType = kAudioUnitSubType_MultiChannelMixer;
mixerUnitDescription.componentManufacturer = kAudioUnitManufacturer_Apple;
mixerUnitDescription.componentFlags = 0;
mixerUnitDescription.componentFlagsMask = 0;
AUNode mixerNode;
AUGraphAddNode(audioGraph, &mixerUnitDescription, &mixerNode);

// 建立 EQ node
AudioComponentDescription EQUnitDescription;
EQUnitDescription.componentType= kAudioUnitType_Effect;
```

```

EQUnitDescription.componentSubType = kAudioUnitSubType_AUipodEQ;
EQUnitDescription.componentManufacturer = kAudioUnitManufacturer_Apple;
EQUnitDescription.componentFlags = 0;
EQUnitDescription.componentFlagsMask = 0;
AUNode EQNode;
AUGraphAddNode(audioGraph, &EQUnitDescription, &EQNode);

// 建立 remote IO node
AudioComponentDescription outputUnitDescription;
bzero(&outputUnitDescription, sizeof( AudioComponentDescription));
outputUnitDescription.componentType = kAudioUnitType_Output;
outputUnitDescription.componentSubType = kAudioUnitSubType_RemoteIO;
outputUnitDescription.componentManufacturer = kAudioUnitManufacturer_Apple;
outputUnitDescription.componentFlags = 0;
outputUnitDescription.componentFlagsMask = 0;
AUNode outputNode;
AUGraphAddNode(audioGraph, &outputUnitDescription, &outputNode);

// 將 mixer node 連接到 EQ node
AUGraphConnectNodeInput(audioGraph, mixerNode, 0, EQNode, 0);
// 將 EQ node 連接到 Remote IO
AUGraphConnectNodeInput(audioGraph, EQNode, 0, outputNode, 0);

```

拿出 Audio Unit

Audio Unit 是 Audio Node 的操作介面，在建立了 Audio Node 之後，我們便可以用 `AUGraphNodeInfo` 拿出 Audio Unit。

```

AUGraphNodeInfo(audioGraph, outputNode, &outputUnitDescription, &outputUnit);
AUGraphNodeInfo(audioGraph, EQNode, &EQUnitDescription, &EQUnit);
AUGraphNodeInfo(audioGraph, mixerNode, &mixerUnitDescription, &mixerUnit);

```

設定輸入與輸出格式

每個 Audio Unit 在互相串接後，就會將前一個 Audio Unit 的輸出送到下一個 Audio Unit 的輸入端，因此我們要設定每個 Audio Unit 的輸入與輸出格式，讓音訊資料可以正確通過每一個 Audio Unit。設定輸入與輸出格式的方式是修改 `kAudioUnitProperty_StreamFormat` 這項屬性，如果要改輸入格式，就將 scope 設定為 `kAudioUnitScope_Input`，反之就是 `kAudioUnitScope_Output`。

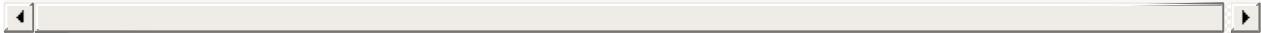
我們的第一個 Audio Unit 是 mixer，而接下來會把 render callback function 繩在 mixer 的 bus 0，後面也全部是在 bus 0 上串接，因此，我們設定了 mixer 與 EQ 的輸入與輸出格式。由於送到 Remote IO 就會播放，我們就不用設定 Remote IO 的輸出格式了。

```

AudioStreamBasicDescription audioFormat = KKSignedIntLinearPCMStreamDescription();
AudioUnitSetProperty(mixerUnit, kAudioUnitProperty_StreamFormat, kAudioUnitScope_Input, 0
, &audioFormat, sizeof(audioFormat));
AudioUnitSetProperty(mixerUnit, kAudioUnitProperty_StreamFormat, kAudioUnitScope_Output, 0
, &audioFormat, sizeof(audioFormat));
AudioUnitSetProperty(EQUnit, kAudioUnitProperty_StreamFormat, kAudioUnitScope_Input, 0, &
audioFormat, sizeof(audioFormat));

```

```
AudioUnitSetProperty(EQUnit, kAudioUnitProperty_StreamFormat, kAudioUnitScope_Output, 0,
&audioFormat, sizeof(audioFormat));
AudioUnitSetProperty(outputUnit, kAudioUnitProperty_StreamFormat, kAudioUnitScope_Input, 0
, &audioFormat, sizeof(audioFormat));
```



設定最大 Frame Per Slice

關於這項設定，要參考蘋果 Technical Q&A QA1606 這份重要的說明。Audio Unit API 是一種 pull model 的設計，不是我們主動把音訊資料推給 AUGraph，而是讓 AUGraph 透過 render callback function 跟我們索取資料，至於會跟我們要求多少資料，是由系統決定，而不是由我們決定。

而 AUGraph 會跟我們要求多少資料會變動的，平常的時候，一次會跟我們要求 1024 個 frame，但是當 iOS 裝置在 lock screen 的時候，基於節電的理由，會變成一次跟我們要比較多的資料，變成 4096 個 frame，但每個 Audio Unit 預設可以通過的 frame 數量是 1156。所以，如果不調整設定，當裝置進入 lock screen 之後，就會因為中間串接的 Audio Unit 無法通過 4096 個 frame 而造成無法播放。

因此我們要將每個 Audio Unit 可以通過的資料量設成 4096。

```
UInt32 maxFPS = 4096;
AudioUnitSetProperty(mixerUnit, kAudioUnitProperty_MaximumFramesPerSlice, kAudioUnitScope_Global, 0, &maxFPS, sizeof(maxFPS));
AudioUnitSetProperty(EQUnit, kAudioUnitProperty_MaximumFramesPerSlice, kAudioUnitScope_Global, 0, &maxFPS, sizeof(maxFPS));
AudioUnitSetProperty(outputUnit, kAudioUnitProperty_MaximumFramesPerSlice, kAudioUnitScope_Global, 0, &maxFPS, sizeof(maxFPS));
```

這個數字也決定了我們設定給 converter 使用的 renderBufferSize 的大小，我們設成 $4096 * 4$ ，原因是我們需要提供 4096 個 frame，而每個 frame 裡頭有左右聲道，所以會是兩個十六位元整數，每個十六位元整數是 2 bytes。

最後是設定 render callback，方式與前一節相同。

調整 player 的各項設定

在這個 player 中我們可以透過 Audio Unit 調整各項設定，比方說，我們想要調整這個 player 的音量，便可以透過 AudioUnitSetParameter，對 mixer 的 bus 0 調整 kMultiChannelMixerParam_Volume 屬性。

這個範例中也示範了如何使用 EQ 等化器。iOS 的 EQ 等化器有一些 preset，存放在一個 CFArray 中，我們可以從中選擇喜愛的 preset 套用。在 iPodEQPresetsArray 與 -selectEQPreset: 示範了如何使用 EQ 等化器。

```
- (CFArrayRef)iPodEQPresetsArray
{
    CFArrayRef array;
    UInt32 size = sizeof(array);
    AudioUnitGetProperty(EQUnit, kAudioUnitProperty_FactoryPresets, kAudioUnitScope_Global, 0, &array, &size);
    return array;
}
```

```

- (void)selectEQPreset:(NSInteger)value
{
    AUPreset *aPreset = (AUPreset*)CFArrayGetValueAtIndex(self.iPodEQPresetsArray, value)
;
    AudioUnitSetProperty(EQUnit, kAudioUnitProperty_PresentPreset, kAudioUnitScope_Global
, 0, aPreset, sizeof(AUPreset));
}

```

完成的範例 Player 如下：

KKSimpleAUPlayer.h

```

#import <Foundation/Foundation.h>
#import <AudioToolbox/AudioToolbox.h>

@interface KKSimpleAUPlayer : NSObject
- (id)initWithURL:(NSURL *)inURL;
- (void)play;
- (void)pause;

@property (readonly, nonatomic) CFArrayRef iPodEQPresetsArray;
- (void)selectEQPreset:(NSInteger)value;
@end

```

KKSimpleAUPlayer.m

```

#import "KKSimpleAUPlayer.h"

static void KKAudioFileStreamPropertyListener(void* inClientData,
    AudioFileStreamID inAudioFileStream,
    AudioFileStreamPropertyID inPropertyID,
    UInt32* ioFlags);
static void KKAudioFileStreamPacketsCallback(void* inClientData,
    UInt32 inNumberBytes,
    UInt32 inNumberPackets,
    const void* inInputData,
    AudioStreamPacketDescription *inPacketDescriptions);
static OSStatus KKPlayerAURenderCallback(void *userData,
    AudioUnitRenderActionFlags *ioActionFlags,
    const AudioTimeStamp *inTimeStamp,
    UInt32 inBusNumber,
    UInt32 inNumberFrames,
    AudioBufferList *ioData);
static OSStatus KKPlayerConverterFiller(AudioConverterRef inAudioConverter,
    UInt32* ioNumberDataPackets,
    AudioBufferList* ioData,
    AudioStreamPacketDescription** outDataPacketDescription,
    void* inUserData);

static const OSStatus KKAudioConverterCallbackErr_NoData = 'kknd';

```

```

static AudioStreamBasicDescription KKSignedIntLinearPCMStreamDescription();

@interface KKSsimpleAUPlayer () <NSURLConnectionDelegate>
{
    NSURLConnection *URLConnection;
    struct {
        BOOL stopped;
        BOOL loaded;
    } playerStatus ;

    AUGraph audioGraph;
    AudioUnit mixerUnit;
    AudioUnit EQUnit;
    AudioUnit outputUnit;

    AudioFileStreamID audioFileStreamID;
    AudioStreamBasicDescription streamDescription;
    AudioConverterRef converter;
    AudioBufferList *renderBufferList;
    UInt32 renderBufferSize;

    NSMutableArray *packets;
    size_t readHead;
}

- (double)packetsPerSecond;
@end

AudioStreamBasicDescription KKSignedIntLinearPCMStreamDescription()
{
    AudioStreamBasicDescription destFormat;
    bzero(&destFormat, sizeof(AudioStreamBasicDescription));
    destFormat.mSampleRate = 44100.0;
    destFormat.mFormatID = kAudioFormatLinearPCM;
    destFormat.mFormatFlags = kLinearPCMFormatFlagIsSignedInteger;
    destFormat.mFramesPerPacket = 1;
    destFormat.mBytesPerPacket = 4;
    destFormat.mBytesPerFrame = 4;
    destFormat.mChannelsPerFrame = 2;
    destFormat.mBitsPerChannel = 16;
    destFormat.mReserved = 0;
    return destFormat;
}

@implementation KKSsimpleAUPlayer

- (void)dealloc
{
    AUGraphUninitialize(audioGraph);
    AUGraphClose(audioGraph);
    DisposeAUGraph(audioGraph);

    AudioFileStreamClose(audioFileStreamID);
    AudioConverterDispose(converter);
}

```

```

    free(renderBufferList->mBuffers[0].mData);
    free(renderBufferList);
    renderBufferList = NULL;

    [URLConnection cancel];
}

- (void)buildOutputUnit
{
    // 建立 AudioGraph
    OSStatus status = NewAUGraph(&audioGraph);
    NSAssert(noErr == status, @"We need to create a new audio graph. %d", (int)status);
    status = AUGraphOpen(audioGraph);
    NSAssert(noErr == status, @"We need to open the audio graph. %d", (int)status);

    // 建立 mixer node
    AudioComponentDescription mixerUnitDescription;
    mixerUnitDescription.componentType= kAudioUnitType_Mixer;
    mixerUnitDescription.componentSubType = kAudioUnitSubType_MultiChannelMixer;
    mixerUnitDescription.componentManufacturer = kAudioUnitManufacturer_Apple;
    mixerUnitDescription.componentFlags = 0;
    mixerUnitDescription.componentFlagsMask = 0;
    AUNode mixerNode;
    status = AUGraphAddNode(audioGraph, &mixerUnitDescription, &mixerNode);
    NSAssert(noErr == status, @"We need to add the mixer node. %d", (int)status);

    // 建立 EQ node
    AudioComponentDescription EQUnitDescription;
    EQUnitDescription.componentType= kAudioUnitType_Effect;
    EQUnitDescription.componentSubType = kAudioUnitSubType_AUiPodEQ;
    EQUnitDescription.componentManufacturer = kAudioUnitManufacturer_Apple;
    EQUnitDescription.componentFlags = 0;
    EQUnitDescription.componentFlagsMask = 0;
    AUNode EQNode;
    status = AUGraphAddNode(audioGraph, &EQUnitDescription, &EQNode);
    NSAssert(noErr == status, @"We need to add the EQ effect node. %d", (int)status);

    // 建立 remote IO node
    AudioComponentDescription outputUnitDescription;
    bzero(&outputUnitDescription, sizeof(AudioComponentDescription));
    outputUnitDescription.componentType = kAudioUnitType_Output;
    outputUnitDescription.componentSubType = kAudioUnitSubType_RemoteIO;
    outputUnitDescription.componentManufacturer = kAudioUnitManufacturer_Apple;
    outputUnitDescription.componentFlags = 0;
    outputUnitDescription.componentFlagsMask = 0;
    AUNode outputNode;
    status = AUGraphAddNode(audioGraph, &outputUnitDescription, &outputNode);
    NSAssert(noErr == status, @"We need to add an output node to the audio graph. %d", (int)status);

    // 將 mixer node 連接到 EQ node
    status = AUGraphConnectNodeInput(audioGraph, mixerNode, 0, EQNode, 0);
    NSAssert(noErr == status, @"We need to connect the nodes within the audio graph. %d",

```

```

(int)status);

// 將 EQ node 連接到 Remote IO
status = AUGraphConnectNodeInput(audioGraph, EQNode, 0, outputNode, 0);
NSAssert(noErr == status, @"We need to connect the nodes within the audio graph. %d",
(int)status);

// 拿出 Remote IO 的 Audio Unit
status = AUGraphNodeInfo(audioGraph, outputNode, &outputUnitDescription, &outputUnit)
;
NSAssert(noErr == status, @"We need to get the audio unit of the output node. %d", (int)status);

// 拿出 EQ node 的 Audio Unit
status = AUGraphNodeInfo(audioGraph, EQNode, &EQUnitDescription, &EQUnit);
NSAssert(noErr == status, @"We need to get the audio unit of the EQ effect node. %d",
(int)status);

// 拿出 mixer node 的 Audio Unit
status = AUGraphNodeInfo(audioGraph, mixerNode, &mixerUnitDescription, &mixerUnit);
NSAssert(noErr == status, @"We need to get the audio unit of the mixer node. %d", (int)status);

// 設定 mixer node 的輸入輸出格式
AudioStreamBasicDescription audioFormat = KKSignedIntLinearPCMStreamDescription();
status = AudioUnitSetProperty(mixerUnit, kAudioUnitProperty_StreamFormat, kAudioUnitsScope_Input, 0, &audioFormat, sizeof(audioFormat));
NSAssert(noErr == status, @"We need to set input format of the mixer node. %d", (int)status);

status = AudioUnitSetProperty(mixerUnit, kAudioUnitProperty_StreamFormat, kAudioUnitsScope_Output, 0, &audioFormat, sizeof(audioFormat));
NSAssert(noErr == status, @"We need to set output format of the mixer effect node. %d",
(int)status);

// 設定 EQ node 的輸入輸出格式
status = AudioUnitSetProperty(EQUnit, kAudioUnitProperty_StreamFormat, kAudioUnitScope_Input, 0, &audioFormat, sizeof(audioFormat));
NSAssert(noErr == status, @"We need to set input format of the EQ node. %d", (int)status);

status = AudioUnitSetProperty(EQUnit, kAudioUnitProperty_StreamFormat, kAudioUnitScope_Output, 0, &audioFormat, sizeof(audioFormat));
NSAssert(noErr == status, @"We need to set output format of the EQ effect node. %d", (int)status);

// 設定 Remote IO node 的輸入格式
status = AudioUnitSetProperty(outputUnit, kAudioUnitProperty_StreamFormat, kAudioUnitScope_Input, 0, &audioFormat, sizeof(audioFormat));
NSAssert(noErr == status, @"We need to set input format of the remote IO node. %d",
(int)status);

// 設定 maxFPS
UInt32 maxFPS = 4096;
status = AudioUnitSetProperty(mixerUnit, kAudioUnitProperty_MaximumFramesPerSlice, kAudioUnitScope_Global, 0, &maxFPS, sizeof(maxFPS));

```

```

    NSAssert(noErr == status, @"We need to set the maximum FPS to the mixer node. %d", (int)status);
    status = AudioUnitSetProperty(EQUnit, kAudioUnitProperty_MaximumFramesPerSlice, kAudioUnitScope_Global, 0,&maxFPS, sizeof(maxFPS));
    NSAssert(noErr == status, @"We need to set the maximum FPS to the EQ effect node. %d"
, (int)status);
    status = AudioUnitSetProperty(outputUnit, kAudioUnitProperty_MaximumFramesPerSlice, kAudioUnitScope_Global, 0,&maxFPS, sizeof(maxFPS));
    NSAssert(noErr == status, @"We need to set the maximum FPS to the EQ effect node. %d"
, (int)status);

    // 設定 render callback
    AURenderCallbackStruct callbackStruct;
    callbackStruct.inputProcRefCon = (__bridge void *)(self);
    callbackStruct.inputProc = KKPlayerAURenderCallback;
    status = AUGraphSetNodeInputCallback(audioGraph, mixerNode, 0, &callbackStruct);
    NSAssert(noErr == status, @"Must be no error.");

    status = AUGraphInitialize(audioGraph);
    NSAssert(noErr == status, @"Must be no error.");

    // 建立 converter 要使用的 buffer list
    UInt32 bufferSize = 4096 * 4;
    renderBufferSize = bufferSize;
    renderBufferList = (AudioBufferList *)calloc(1, sizeof(UInt32) + sizeof(AudioBuffer))
;
    renderBufferList->mNumberBuffers = 1;
    renderBufferList->mBuffers[0].mNumberChannels = 2;
    renderBufferList->mBuffers[0].mDataByteSize = bufferSize;
    renderBufferList->mBuffers[0].mData = calloc(1, bufferSize);

    CASHow(audioGraph);
}

- (id)initWithURL:(NSURL *)inURL
{
    self = [super init];
    if (self) {
        [self buildOutputUnit];

        playerStatus.stopped = NO;
        packets = [[NSMutableArray alloc] init];

        // 第一步：建立 Audio Parser, 指定 callback, 以及建立 HTTP 連線,
        // 開始下載檔案
        AudioFileStreamOpen((__bridge void *)(self),
                           KKAUDIOFILESTREAMPROPERTYLISTENER,
                           KKAUDIOFILESTREAMPACKETSCALLBACK,
                           kAudioFileMP3Type, &audioFileStreamID);
        URLConnection = [[NSURLConnection alloc] initWithRequest:[NSURLRequest requestWithURL:inURL] delegate:self];
        playerStatus.stopped = YES;
    }
}

```

```

        return self;
    }

- (double)packetsPerSecond
{
    if (streamDescription.mFramesPerPacket) {
        return streamDescription.mSampleRate / streamDescription.mFramesPerPacket;
    }
    return 44100.0/1152.0;
}

- (void)play
{
    if (!playerStatus.stopped) {
        return;
    }
    OSStatus status = AUGraphStart(audioGraph);
    NSAssert(noErr == status, @"AUGraphStart, error: %ld", (signed long)status);
    status = AudioOutputUnitStart(outputUnit);
    NSAssert(noErr == status, @"AudioOutputUnitStart, error: %ld", (signed long)status);
    playerStatus.stopped = NO;
}

- (void)pause
{
    OSStatus status = AUGraphStop(audioGraph);
    NSAssert(noErr == status, @"AUGraphStart, error: %ld", (signed long)status);
    status = AudioOutputUnitStop(outputUnit);
    NSAssert(noErr == status, @"AudioOutputUnitStop, error: %ld", (signed long)status);
    playerStatus.stopped = YES;
}

- (CFArrayRef)iPodEQPresetsArray
{
    CFArrayRef array;
    UInt32 size = sizeof(array);
    AudioUnitGetProperty(EQUnit, kAudioUnitProperty_FactoryPresets, kAudioUnitScope_Global,
1, 0, &array, &size);
    return array;
}

- (void)selectEQPreset:(NSInteger)value
{
    AUPreset *aPreset = (AUPreset*)CFArrayGetValueAtIndex(self.iPodEQPresetsArray, value)
;
    AudioUnitSetProperty(EQUnit, kAudioUnitProperty_PresentPreset, kAudioUnitScope_Global
, 0, aPreset, sizeof(AUPreset));
}

#pragma mark -
#pragma mark NSURLConnectionDelegate

- (void)connection:(NSURLConnection *)connection didReceiveResponse:(NSURLResponse *)resp

```

```

onse
{
    if ([response isKindOfClass:[NSHTTPURLResponse class]]) {
        if ([((NSHTTPURLResponse *)response statusCode] != 200) {
            NSLog(@"%@", [((NSHTTPURLResponse *)response statusCode]);
            [connection cancel];
            playerStatus.stopped = YES;
        }
    }
}

- (void)connection:(NSURLConnection *)connection didReceiveData:(NSData *)data
{
    // 第二步：抓到了部分檔案，就交由 Audio Parser 開始 parse 出 data
    // stream 中的 packet。
    AudioFileStreamParseBytes(audioFileStreamID, (UInt32)[data length], [data bytes], 0);
}

- (void)connectionDidFinishLoading:(NSURLConnection *)connection
{
    NSLog(@"Complete loading data");
    playerStatus.loaded = YES;
}

- (void)connection:(NSURLConnection *)connection didFailWithError:(NSError *)error
{
    NSLog(@"Failed to load data: %@", [error localizedDescription]);
    playerStatus.stopped = YES;
}

#pragma mark -
#pragma mark Audio Parser and Audio Queue callbacks

- (void)_createAudioQueueWithAudioStreamDescription:(AudioStreamBasicDescription *)audioS
treamBasicDescription
{
    memcpy(&streamDescription, audioStreamBasicDescription, sizeof(AudioStreamBasicDescri
ption));
    AudioStreamBasicDescription destFormat = KKSignedIntLinearPCMStreamDescription();
    AudioConverterNew(&streamDescription, &destFormat, &converter);
}

- (void)_storePacketsWithNumberOfBytes:(UInt32)inNumberBytes
    numberOfRowsInSection:(UInt32)inNumberPackets
    inputData:(const void *)inInputData
    packetDescriptions:(AudioStreamPacketDescription *)inPacketDescriptions
{
    for (int i = 0; i < inNumberPackets; ++i) {
        SInt64 packetStart = inPacketDescriptions[i].mStartOffset;
        UInt32 packetSize = inPacketDescriptions[i].mDataByteSize;
        assert(packetSize > 0);
        NSData *packet = [NSData dataWithBytes:inInputData + packetStart length:packetSiz
e];
}

```

```

        [packets addObject:packet];
    }

    // 第五步，因為 parse 出來的 packets 夠多，緩衝內容夠大，因此開始
    // 播放

    if (readHead == 0 && [packets count] > (int)([self packetsPerSecond] * 3)) {
        if (playerStatus.stopped) {
            [self play];
        }
    }
}

#pragma mark -
#pragma mark Properties

- (BOOL)isStopped
{
    return playerStatus.stopped;
}

- (OSStatus)callbackWithNumberOfFrames:(UInt32)inNumberOfFrames
    ioData:(AudioBufferList *)inIoData busNumber:(UInt32)inBusNumber
{
    @synchronized(self) {
        if (readHead < [packets count]) {
            @autoreleasepool {
                UInt32 packetSize = inNumberOfFrames;
                // 第七步： Remote IO node 的 render callback 中，呼叫 converter 將 packet
                轉成 LPCM
                OSStatus status =
                AudioConverterFillComplexBuffer(converter,
                    KKPlayerConverterFiller,
                    (__bridge void *)(self),
                    &packetSize, renderBufferList, NULL);
                if (noErr != status && KKAUDIOCONVERTERCALLBACKERR_NODATA != status) {
                    [self pause];
                    return -1;
                }
                else if (!packetSize) {
                    inIoData->mNumberBuffers = 0;
                }
                else {
                    inIoData->mNumberBuffers = 1;
                    inIoData->mBuffers[0].mNumberChannels = 2;
                    inIoData->mBuffers[0].mDataByteSize = renderBufferList->mBuffers[0].m
DataByteSize;
                    inIoData->mBuffers[0].mData = renderBufferList->mBuffers[0].mData;
                    renderBufferList->mBuffers[0].mDataByteSize = renderBufferSize;
                }
            }
        }
    }
}

```

```

        inIoData->mNumberBuffers = 0;
        return -1;
    }
}

return noErr;
}

- (OSStatus)_fillConverterBufferWithBufferlist:(AudioBufferList *)ioData
    packetDescription:(AudioStreamPacketDescription**) outDataPacketDescription
{
    static AudioStreamPacketDescription aspdesc;

    if (readHead >= [packets count]) {
        return KKAUDIOConverterCallbackErr_NoData;
    }

    ioData->mNumberBuffers = 1;
    NSData *packet = packets[readHead];
    void const *data = [packet bytes];
    UInt32 length = (UInt32)[packet length];
    ioData->mBuffers[0].mData = (void *)data;
    ioData->mBuffers[0].mDataByteSize = length;

    *outDataPacketDescription = &aspdesc;
    aspdesc.mDataByteSize = length;
    aspdesc.mStartOffset = 0;
    aspdesc.mVariableFramesInPacket = 1;

    readHead++;
    return 0;
}

@end

void KKAudioFileStreamPropertyListener(void * inClientData,
    AudioFileStreamID inAudioFileStream,
    AudioFileStreamPropertyID inPropertyID,
    UInt32 * ioFlags)
{
    KKSsimpleAUPlayer *self = (__bridge KKSsimpleAUPlayer *)inClientData;
    if (inPropertyID == kAudioFileStreamProperty_DataFormat) {
        UInt32 dataSize = 0;
        OSStatus status = 0;
        AudioStreamBasicDescription audioStreamDescription;
        Boolean writable = false;
        status = AudioFileStreamGetPropertyInfo(inAudioFileStream,
            kAudioFileStreamProperty_DataFormat, &dataSize, &writable);
        status = AudioFileStreamGetProperty(inAudioFileStream,
            kAudioFileStreamProperty_DataFormat, &dataSize, &audioStreamDescription);

        NSLog(@"%@", audioStreamDescription.mSampleRate);
    }
}

```

```

    NSLog(@"%@", audioStreamDescription.mFormatID);
    NSLog(@"%@", audioStreamDescription.mFormatFlags);
    NSLog(@"%@", audioStreamDescription.mBytesPerPacket);
    NSLog(@"%@", audioStreamDescription.mFramesPerPacket);
    NSLog(@"%@", audioStreamDescription.mBytesPerFrame);
    NSLog(@"%@", audioStreamDescription.mChannelsPerFrame);
    NSLog(@"%@", audioStreamDescription.mBitsPerChannel);
    NSLog(@"%@", audioStreamDescription.mReserved);

    // 第三步： Audio Parser 成功 parse 出 audio 檔案格式，我們根據
    // 檔案格式資訊，建立 converter

    [self _createAudioQueueWithAudioStreamDescription:&audioStreamDescription];
}

}

void KKAudioFileStreamPacketsCallback(void* inClientData,
    UInt32 inNumberBytes,
    UInt32 inNumberPackets,
    const void* inInputData,
    AudioStreamPacketDescription* inPacketDescriptions)
{
    // 第四步： Audio Parser 成功 parse 出 packets，我們將這些資料儲存
    // 起來

    KKSsimpleAUPlayer *self = (__bridge KKSsimpleAUPlayer *)inClientData;
    [self _storePacketsWithNumberOfBytes:inNumberBytes
        numberOfPackets:inNumberPackets
        inputData:inInputData
        packetDescriptions:inPacketDescriptions];
}

OSStatus KKPlayerAURenderCallback(void *userData,
    AudioUnitRenderActionFlags *ioActionFlags,
    const AudioTimeStamp *inTimeStamp,
    UInt32 inBusNumber,
    UInt32 inNumberFrames,
    AudioBufferList *ioData)
{
    // 第六步： Remote IO node 的 render callback
    KKSsimpleAUPlayer *self = (__bridge KKSsimpleAUPlayer *)userData;
    OSStatus status = [self callbackWithNumberOfFrames:inNumberFrames
        ioData:ioData busNumber:inBusNumber];
    if (status != noErr) {
        ioData->mNumberBuffers = 0;
        *ioActionFlags |= kAudioUnitRenderAction_OutputIsSilence;
    }
    return status;
}

OSStatus KKPlayerConverterFiller (AudioConverterRef inAudioConverter,
    UInt32* ioNumberDataPackets,
    AudioBufferList* ioData,

```

```
AudioStreamPacketDescription** outDataPacketDescription,
void* inUserData)
{
    // 第八步： AudioConverterFillComplexBuffer 的 callback
    KKSsimpleAUPlayer *self = (__bridge KKSsimpleAUPlayer *)inUserData;
    *ioNumberDataPackets = 0;
    OSStatus result = [self _fillConverterBufferWithBufferlist:ioData
        packetDescription:outDataPacketDescription];
    if (result == noErr) {
        *ioNumberDataPackets = 1;
    }
    return result;
}
```

修改 LPCM 資料

在前面兩節中，我們使用 `AudioConverterFillComplexBuffer` 把 MP3 等格式的資料，轉換成 LPCM 格式後播放。由於 LPCM 格式已經可以算是最原始的資料了，因此，我們可以透過直接修改 LPCM 資料，創造各種音效。

我們在這邊示範一種最簡單的效果：用相位取消來做去人聲。假設在錄音的過程中，人聲是只用一個麥克風來錄音，並平均分佈在兩條聲道上，那麼，只要左右聲道的音量互減，然後左右聲道都變成是音量互減之後的結果，就可以拿到去除人聲的效果。不過，這樣也會同時失去立體聲：因為左右聲道的聲音都變成一樣的了。

我們先來看前一節把 `renderList` 的資料填入 `inIoData` 的這段。

```
inIoData->mNumberBuffers = 1;
inIoData->mBuffers[0].mNumberChannels = 2;
inIoData->mBuffers[0].mDataByteSize = renderBufferList->mBuffers[0].mDataByteSize;
inIoData->mBuffers[0].mData = renderBufferList->mBuffers[0].mData;
renderBufferList->mBuffers[0].mDataByteSize = renderBufferSize;
```

我們在這邊的前方，寫一段程式，讓左右聲道互減：

```
size_t sampleCount = renderBufferList->mBuffers[0].mDataByteSize / 4;
for(size_t i = 0; i < sampleCount * 4; i += 4) {
    short *frame = (short *)(renderBufferList->mBuffers[0].mData + i);
    short left = *frame;
    short right = *(frame + 1);
    short new = left - right;
    *frame = new;
    *(frame + 1) = new;
}

inIoData->mNumberBuffers = 1;
inIoData->mBuffers[0].mNumberChannels = 2;
inIoData->mBuffers[0].mDataByteSize = renderBufferList->mBuffers[0].mDataByteSize;
inIoData->mBuffers[0].mData = renderBufferList->mBuffers[0].mData;
renderBufferList->mBuffers[0].mDataByteSize = renderBufferSize;
```

這邊的寫法跟我們設定的 LPCM 格式有關：我們設定的格式是左右聲道分別是兩個 16 位元帶號整數，因此一個 sample 共有 4 個 bytes，前兩個是左聲道，後兩個是右聲道。所以 bytes 的大小除四之後，就是 sample 的數量。拿到 `left` 與 `right` 互減之後，只要這麼一小段程式碼，就可以達到去人聲的效果。—不過，假如人聲並不是平均分配在左右聲道上，這個作法也就沒什麼作用，也很有可能會產生預期之外的噪音。

技術上，我們要讓 KKBOX 的每首歌曲都去除人聲，甚至改變音高，做升 key 降 key 的功能，都是做得到的事情。不過在版權議題上，我們不能這麼做。

Core Audio 到底難在哪？

相較於 iOS 問世的前幾年，後來出現了像 AVAudioEngine，以及 [The Amazing Audio Engine](#) 與 [AudioKit](#) 等專案，會讓做 audio 的相關開發容易一些，如果現在想要做一些新的 audio 應用，是可以先從這些第三方 library 下手，可以免除掉不少直接使用像 AUGraph、AudioUnit 痛苦的地方。

使用 AUGraph 這些 audio API 最痛苦的地方，在於在開發過程中難免會有錯誤，而發生錯誤時讓人完全難以理解。

每次呼叫一個 API 的時候，會回傳一種叫做 OSStatus 的錯誤代碼，OSStatus 也就是 32 位元整數，有時候我們可以把這個整數轉換成四個 char，然後這四個 char 可能會有一些意義，可以讓你大概判讀可能的原因。

Core Audio API 在錯誤代碼的解釋方面很少，由於相對於其他種類的 App 開發，Audio App 可能是種比較冷門的領域，坊間講 Core Audio 開發的書籍目前也只看到 2012 年 Chris Adamson 所做的 *Learning Core Audio* 那麼一本，當你遇到錯誤的時候，連網路上的討論也找不到幾篇。

而有些時候，蘋果對錯誤代碼的說明，也跟你實際遇到的狀況完全沒有關係。

比方說，如果你還沒呼叫 `AUGraphInitialize`，就呼叫了 `AUGraphStart`，那麼 `AUGraphStart` 就會回傳錯誤代碼 `kAUGraphErr_CannotDoInCurrentContext` (-10863)。要修正這個問題，是記得要去呼叫一次 `AUGraphInitialize`，可是蘋果的說明文件是這麼說的：

To avoid spinning or waiting in the render thread (a bad idea!), many of the calls to AUGraph can return: `kAUGraphErr_CannotDoInCurrentContext`. This result is only generated when you call an AUGraph API from its render callback....

...有很多的 API 可能會回傳 `kAUGraphErr_CannotDoInCurrentContext`，避免在 AUGraph 的 render threading 發生卡死的狀況。這種狀況只有當你在 render thread 裡頭呼叫了 AUGraph API 才會發生...

可是我們明明是在呼叫 `AUGraphStart` 的時候就發生錯誤了，連 render thread 都還沒有進去。這個說明跟我們遇到的狀況完全無關啊！

或，我們寫好了前一節的 player 之後，想繼續玩玩看有哪些 effect node 可以使用，我們就把原本 EQ node 裡頭這行 `EQUnitDescription.componentSubType = kAudioUnitSubType_AUiPodEQ` 換成 `EQUnitDescription.componentSubType = kAudioUnitSubType_Reverb2`，想試試看 iOS 上的 reverb 效果如何，結果在設定 effect node 的輸入格式的時候，發生 `kAudioUnitErr_FormatNotSupported` 錯誤 (-10868)，為什麼呢？你可能要花上很長一段時間在網路上搜尋，才在某個論壇的某篇文章知道，iOS 上的 reverb2 這個 effect node 只接受浮點格式的 LPCM 資料，但我們現在卻是使用 16 位元整數，因此會在設定輸入格式的時候失敗。

可是，你去翻蘋果官方文件，卻也根本找不到 reverb2 只能夠接受哪些輸入格式的相關說明，唯一跟 `kAudioUnitSubType_Reverb2` 相關的說明就只有兩行：

- iPhone only. A reverb for iOS。

使用 AVAudioEngine 開發播放軟體

AVAudioEngine 是蘋果在 iOS 8 時推出的音訊處理相關 API。

相較於 Core Audio/Audio Graph/Audio Unit 這些 C API，AVAudioEngine 相對比較高階：我們可以把 AVAudioEngine 想像成是把 Core Audio API 再用 Objective-C 物件包裝一層，對一些麻煩、重複的工作做一些抽象化，不過，基本上還是要有一些跟音訊處理相關的背景知識，才有辦法操作 AVAudioEngine。

由於只是包裝一層，所以你還是可以從這些介面中，接觸到原本的 C API，像是 `AVAudioUnit` 這個 class 有個屬性，叫做 `audioUnit`，就可以拿到 `AudioUnit` 型態的資料；在 `AVAudioBuffer` 中，也可以拿到屬於 C API 的 `AudioBufferList` 資料。

在蘋果剛推出 AVAudioEngine 的時候，個人感覺是，蘋果主要希望讓像是遊戲或混音軟體的廠商用比較高階的 API，輕鬆製作多的不同 input 時的混音應用，所以一開始也只有提供 `AVAudioFile` 與 `AVAudioPCMBuffer` 這些 API，讓你透過 `AVAudioPCMBuffer` 從各種不同的音檔中讀出 PCM buffer，然後丟進 `AVAudioEngine` 混音。在這個時候，還缺少了處理串流的資料的編碼/解碼的相關 API，像是 `AVAudioCompressedBuffer` 與 `AVAudioConverter`，都是後面幾代 iOS 才陸續出現。在我們的經驗中，如果你想要盡可能使用 AVAudioEngine API 實作一個串流播放器的話，你會需要用到 iOS 11 以上的 API。

AVAudioEngine 與 Audio Unit Processing Graph 的關係

我們前面講過 Audio Unit Processing Graph API，所以我們可以了解 AVAudioEngine API 與 Core Audio之間的一些對應關係：

AVAudioEngine 包裝了 AUGraph：在使用 AVAudioEngine API 時，AVAudioEngine 這個 class 本身扮演了 AUGraph 的角色。我們在使用 AUGraph 的時候，當中所有的 AUNode都要自己手動建立，使用 AVAudioEngine 則比較輕鬆：AVAudioEngine 本身就提供幾個基本的節點，像是 `inputNode`、`outputNode`、`mainMixerNode` 等，只要直接呼叫 AVAudioEngine的這幾個 property，就會用 lazy 的方式產生出幾個基本的節點。

但，如果你要手動增加其他的節點，還是可以自己建立想要的 AUAudioNode 的 subclass，像是 EQ 等化器、Reverb 殘響效果...等等，然後透過 AVAudioEngine 的 `-attachNode:` 加入節點，以及用 `-connect:to:fromBus:toBus:format:` 將節點串連起來。AVAudioEngine 的 `-attachNode:` 對應到 AUGraph 的 `AUGraphAddNode`，`-connect:to:fromBus:toBus:format:` 則對應到 `AUGraphConnectNodeInput`。

AUAudioNode 包裝了 AUNode，像是輸入、輸出、混音，以及播放過程當中的我們想要加入的各種效果，都是各自的 AUAudioNode。以播放來說，我們會特別注意

`AVAudioPlayerNode`：`AVAudioPlayerNode` 是 `AVAudioEngine` 的播放資料來源，我們要播放音訊，首先要建立自己的 `AVAudioPlayerNode`，然後連接到 main mixer node 之類的節點上。

在使用 AUGraph API 的時候，我們的作法是準備好 render callback function，當 AUGraph 需要資料的時候，從 render callback function 中提供資料。至於在使用 AVAudioEngine 時，我們則是反過來，主動透過 schedule... 開頭的一系列 method，或是提供一個 Audio File，或是把 PCM 資料包在 `AVAudioPCMBuffer` 物件裡頭，餵入 `AVAudioPlayerNode`。

如果我們想要播放的是直接從網路抓取的音樂資料，而且是壓縮音檔，那麼就必須把 MP3、AAC 之類的檔案，手動轉換成 PCM 資料。在 AVAudioEngine API 中，有兩種不同的 buffer 物件：`AVAudioPCMBuffer` 與 `AVAudioCompressedBuffer`，兩者都繼承自 `AVAudioBuffer`。

`AVAudioPCMBuffer` 用來包裝 PCM 格式的資料，而如果是壓縮音檔，就得先裝在 `AVAudioCompressedBuffer` 這個容器中，然後，你可以用 `AVAudioConverter` 轉換格式，以播放的情境來說，你就往往需要把 `AVAudioCompressedBuffer` 轉換成 `AVAudioPCMBuffer`，在錄音的情境下則反之。

我們知道，並不是只有一種格式的 PCM，而是可以是 16 位元整數、32 位元整數或 32 位元浮點數...等等不同格式的 PCM 資料。所以我們可以看到 `AVAudioPCMBuffer` 有三個屬性：`floatChannelData`、`int16ChannelData` 與 `int32ChannelData`，裡頭分別是不同型態的 PCM 資料。請注意，在三個屬性當中，一次只會有一個屬性中會有資料，例如，如果你用的是 16 位元整數的 PCM 資料，那麼就只有 `int16ChannelData` 會拿到資料，其他兩個屬性當中都會是空的。

用 AVAudioEngine 實作播放器

由於 `AVAudioEngine` 是把 `AUGraph` 用 Objective-C 包裝起來，所以整體流程與前一章差不多。用 `AVAudioEngine` 實作一個播放器的流程大概是：

- 建立 `AVAudioEngine` 的 instance
- 將各種音效處理的 `AVAudioUnit` 物件加入到 `AVAudioEngine` 中，這時候會呼叫到 `- attachNode: ` -connect:to:fromBus:toBus:format: ...` 等等`
- 建立 `AVAudioPlayerNode` 的 instance，連接到 main mixer node 上
- 發送網路連線抓取音檔
- 將音檔的 packet 儲存起來
- 要求 `AVAudioEngine` 與 `AVAudioPlayerNode` 開始播放
- 將壓縮音檔的 packet 轉換成 `AVAudioPCMBuffer`，餵入 `AVAudioPlayerNode`。

寫起來會像是這樣：

`KKSimpleAudioEnginePlayer.h`

```
@import Foundation;

@interface KKSimpleAudioEnginePlayer : NSObject
- (instancetype)initWithURL:(NSURL *)inURL;
- (void)play;
- (void)pause;
@property (readonly, getter=isPlaying) BOOL playing;
@end
```

`KKSimpleAudioEnginePlayer.m`

```
#import "KKSimpleAudioEnginePlayer.h"
#import AVFoundation;
#import AudioToolbox;

static void KKAUDIOFILESTREAMPROPERTYLISTENER(void* inClientData,
    AudioFileStreamID inAudioFileStream,
    AudioFileStreamPropertyID inPropertyID,
    UInt32* ioFlags);
static void KKAUDIOFILESTREAMPACKETSCALLBACK(void* inClientData,
    UInt32 inNumberBytes,
```

```

    UInt32 inNumberPackets,
    const void* inInputData,
    AudioStreamPacketDescription *inPacketDescriptions);

@interface KKSsimpleAudioEnginePlayer() <NSURLConnectionDelegate>
{
    struct {
        BOOL stopped;
        BOOL loaded;
    } playerStatus ;

    AudioFileStreamID audioFileStreamID;
}

@property (strong, nonatomic) AVAudioEngine *audioEngine;
@property (strong, nonatomic) AVAudioPlayerNode *player;
@property (strong, nonatomic) NSURLConnection *URLConnection;
@property (strong, nonatomic) NSMutableArray<NSData *> *packets;
@property (assign, nonatomic) size_t readHead;
@property (strong, nonatomic) AVAudioConverter *converter;
@property (strong, nonatomic) AVAudioFormat *format;
@property (strong, nonatomic) AVAudioFormat *destinationPCMFormat;
@end

@implementation KKSsimpleAudioEnginePlayer

- (instancetype)initWithURL:(NSURL *)inURL
{
    self = [super init];
    if (self) {
        self.audioEngine = [[AVAudioEngine alloc] init];
        self.player = [[AVAudioPlayerNode alloc] init];
        [self.audioEngine attachNode:self.player];
        [self.audioEngine connect:self.player to:self.audioEngine.mainMixerNode fromBus:0
toBus:0 format:nil];
        self.packets = [[NSMutableArray alloc] init];
        self.destinationPCMFormat = [[AVAudioFormat alloc] initWithCommonFormat:AVAudioPCMFormatFloat32 sampleRate:44100 channels:2 interleaved:NO];

        // 第一步：建立 Audio Parser，指定 callback，以及建立 HTTP 連線，
        // 開始下載檔案
        AudioFileStreamOpen((__bridge void *)(self),
                           KKAudioFileStreamPropertyListener,
                           KKAudioFileStreamPacketsCallback,
                           kAudioFileMP3Type, &audioFileStreamID);
        self.URLConnection = [[NSURLConnection alloc] initWithRequest:[NSURLRequest requestWithURL:inURL] delegate:self];
        playerStatus.stopped = YES;

    }
    return self;
}

```

```

- (AVAudioPCMBuffer *)read
{
    const NSInteger packetPerSlice = 8;

    AVAudioPCMBuffer *pcmBuffer = [[AVAudioPCMBuffer alloc] initWithPCMFormat:self.destinationPCMFormat frameCapacity:self.format.streamDescription->mFramesPerPacket * packetPerSlice];
    NSError *error = nil;
    AVAudioConverterOutputStatus status = [self.converter convertToBuffer:pcmBuffer error:&error withInputFromBlock:^AVAudioBuffer * _Nullable(AVAudioPacketCount inNumberOfPackets, AVAudioConverterInputStatus * _Nonnull outStatus) {

        if (self.readHead >= self.packets.count) {
            *outStatus = AVAudioConverterInputStatus_EndOfStream;
            return nil;
        }

        NSMutableData *packetData = [NSMutableData data];
        AudioStreamPacketDescription *packetDescriptions = calloc(sizeof(AudioStreamPacketDescription), packetPerSlice);
        NSInteger i = 0;
        for (i = 0; i < packetPerSlice; i++) {
            NSData *data = self.packets[self.readHead];
            AudioStreamPacketDescription packetDescription;
            packetDescription.mVariableFramesInPacket = 1;
            packetDescription.mDataByteSize = (UInt32)data.length;
            packetDescription.mStartOffset = (UInt32)packetData.length;
            memcpy(&packetDescriptions[i], &packetDescription, sizeof(AudioStreamPacketDescription));
            [packetData appendData:data];
            self.readHead++;
            if (self.readHead >= self.packets.count) {
                break;
            }
        }
        AVAudioCompressedBuffer *compressedBuffer = [[AVAudioCompressedBuffer alloc] initWithFormat:self.format packetCapacity:packetPerSlice maximumPacketSize:self.format.streamDescription->mFramesPerPacket];

        memcpy(compressedBuffer.data, packetData.bytes, packetData.length);
        memcpy(compressedBuffer.packetDescriptions, packetDescriptions, sizeof(AudioStreamPacketDescription) * i);
        free(packetDescriptions);

        compressedBuffer.packetCount = (AVAudioPacketCount)i;
        compressedBuffer.byteLength = (uint32_t)packetData.length;
        *outStatus = AVAudioConverterInputStatus_HaveData;
        return compressedBuffer;
    }];
    if (status != AVAudioConverterOutputStatus_HaveData) {
        return nil;
    }
}

```

```

    return pcmBuffer;
}

- (void)enqueueBuffer
{
    // 第六步：讀出 PCM Buffer，加到 AVAudioPlayerNode 中
    AVAudioPCMBuffer *buffer = [self read];
    if (buffer) {
        [self.player scheduleBuffer:buffer completionHandler:^{
            [self enqueueBuffer];
        }];
    }
}

- (void)play
{
    NSError *error = nil;
    [self.audioEngine startAndReturnError:&error];
    if (error) {
        return;
    }
    [self.player play];
    [self enqueueBuffer];
    playerStatus.stopped = NO;
}

- (void)pause
{
    NSError *error = nil;
    [self.audioEngine startAndReturnError:&error];
    [self.player stop];
    playerStatus.stopped = YES;
}

- (double)packetsPerSecond
{
    if (self.format) {
        return self.format.sampleRate / self.format.streamDescription->mFramesPerPacket;
    }
    return 44100.0/1152.0;
}

- (void)_createAudioConverterWithAudioStreamDescription:(AudioStreamBasicDescription *)audioStreamBasicDescription
{
    self.format = [[AVAudioFormat alloc] initWithStreamDescription:audioStreamBasicDescription];
    self.converter = [[AVAudioConverter alloc] initFromFormat:self.format toFormat:self.destinationPCMFormat];
}

- (void)_storePacketsWithNumberOfBytes:(UInt32)inNumberBytes
    numberOfPackets:(UInt32)inNumberPackets

```

```

    inputData:(const void *)inInputData
    packetDescriptions:(AudioStreamPacketDescription *)inPacketDescriptions
{
    for (int i = 0; i < inNumberPackets; ++i) {
        SInt64 packetStart = inPacketDescriptions[i].mStartOffset;
        UInt32 packetSize = inPacketDescriptions[i].mDataByteSize;
        assert(packetSize > 0);
        NSData *packet = [NSData dataWithBytes:inInputData + packetStart length:packetSize];
        [self.packets addObject:packet];
    }

    // 第五步，因為 parse 出來的 packets 夠多，緩衝內容夠大，因此開始
    // 播放

    if (self.readHead == 0 && [self.packets count] > (int)([self.packetsPerSecond] * 3))
    {
        if (playerStatus.stopped) {
            [self play];
        }
    }
}

- (BOOL)playing
{
    return playerStatus.stopped == NO;
}

#pragma mark -
#pragma mark NSURLConnectionDelegate

- (void)connection:(NSURLConnection *)connection didReceiveResponse:(NSURLResponse *)response
{
    if ([response isKindOfClass:[NSHTTPURLResponse class]]) {
        if ([[NSHTTPURLResponse *)response statusCode] != 200) {
            NSLog(@"%@", [(NSHTTPURLResponse *)response statusCode]);
            [connection cancel];
            playerStatus.stopped = YES;
        }
    }
}

- (void)connection:(NSURLConnection *)connection didReceiveData:(NSData *)data
{
    // 第二步：抓到了部分檔案，就交由 Audio Parser 開始 parse 出 data
    // stream 中的 packet。
    AudioFileStreamParseBytes(audioFileStreamID, (UInt32)[data length], [data bytes], 0);
}

- (void)connectionDidFinishLoading:(NSURLConnection *)connection
{
    NSLog(@"Complete loading data");
}

```

```

        playerStatus.loaded = YES;
    }

- (void)connection:(NSURLConnection *)connection didFailWithError:(NSError *)error
{
    NSLog(@"Failed to load data: %@", [error localizedDescription]);
    [self pause];
}

@end

void KKAudioFileStreamPropertyListener(void * inClientData,
    AudioFileStreamID inAudioFileStream,
    AudioFileStreamPropertyID inPropertyID,
    UInt32 * ioFlags)
{
    KKSimpleAudioEnginePlayer *self = (__bridge KKSimpleAudioEnginePlayer *)inClientData;
    if (inPropertyID == kAudioFileStreamProperty_DataFormat) {
        UInt32 dataSize = 0;
        OSStatus status = 0;
        AudioStreamBasicDescription audioStreamDescription;
        Boolean writable = false;
        status = AudioFileStreamGetPropertyInfo(inAudioFileStream,
            kAudioFileStreamProperty_DataFormat, &dataSize, &writable);
        status = AudioFileStreamGetProperty(inAudioFileStream,
            kAudioFileStreamProperty_DataFormat, &dataSize, &audioStreamDescription);

        NSLog(@"%@", audioStreamDescription.mSampleRate);
        NSLog(@"%@", audioStreamDescription.mFormatID);
        NSLog(@"%@", audioStreamDescription.mFormatFlags);
        NSLog(@"%@", audioStreamDescription.mBytesPerPacket);
        NSLog(@"%@", audioStreamDescription.mFramesPerPacket);
        NSLog(@"%@", audioStreamDescription.mBytesPerFrame);
        NSLog(@"%@", audioStreamDescription.mChannelsPerFrame);
        NSLog(@"%@", audioStreamDescription.mBitsPerChannel);
        NSLog(@"%@", audioStreamDescription.mReserved);

        // 第三步： Audio Parser 成功 parse 出 audio 檔案格式，我們根據
        // 檔案格式資訊，建立 converter

        [self _createAudioConverterWithAudioStreamDescription:&audioStreamDescription];
    }
}

void KKAudioFileStreamPacketsCallback(void* inClientData,
    UInt32 inNumberBytes,
    UInt32 inNumberPackets,
    const void* inInputData,
    AudioStreamPacketDescription* inPacketDescriptions)
{
    // 第四步： Audio Parser 成功 parse 出 packets，我們將這些資料儲存
    // 起來
}

```

```
KKSsimpleAudioEnginePlayer *self = (__bridge KKSsimpleAudioEnginePlayer *)inClientData;
[self _storePacketsWithNumberOfBytes:inNumberBytes
    numberOfPackets:inNumberPackets
    inputData:inInputData
    packetDescriptions:inPacketDescriptions];
}
```

從 Audio Unit Processing Graph 到 AVAudioEngine

如果你用 AUGraph 開發了一個播放器，要移轉到 AVAudioEngine 的話，除了絕大多數的 C API 會變成 Objective-C 之外，就是這兩種 API 中所提供的各種音訊效果不盡相同。

我們可以從 AVAudioEngine 的介面設計中，看到 AVAudioEngine 試圖抹去 macOS、iOS、tvOS 等平台上的差異。像是：我們在前面提到，在 iOS/tvOS 上，我們用到了 Remote IO 輸出音訊，但 Remote IO 其實不只用在輸出而已，Remote IO 的 bus 0 的用途是用在輸出，而 bus 1 則是用在輸入，我們在使用麥克風錄製音訊的時候，是用 Remote IO 的 bus 1 傳到 AUGraph 中，但是在 macOS 上，輸出時使用的是 kAudioUnitType_Output 以及 subtype kAudioUnitSubType_HALOutput，在輸入時，則要考慮到 macOS 裝置可能有多個麥克風（用 USB 外接了多個麥克風）、「也可能一個麥克風都沒有（像是 Mac Mini）」。不過，在 AVAudioEngine 中，都統一抽象化成 input node 與 output node。

所以，一些原本在 AUGraph 中可以使用的 Audio Unit，在改用 AVAudioEngine 開發播放器的時候，就可能找不到對應的 AVAudioUnit class：因為某些 Audio Unit 只有 iOS 有，但是 macOS 沒有，或反之。AVAudioEngine 所提供的，是多個不同的蘋果平台上都提供的效果。

以 EQ 等化器來說，蘋果原本在 iOS 上提供的是 kAudioUnitSubType_AUipodEQ，在 macOS 上提供 kAudioUnitSubType_GraphicEQ，這兩種等化器的差異是，kAudioUnitSubType_AUipodEQ 只提供一些蘋果用在 iPod 上的一些 preset，像是低音強化、高音強化等，但是 kAudioUnitSubType_GraphicEQ 則提供了 10 或 31 band 的設置，可以自由調整不同頻段聲音的音量。在 AVAudioEngine 相關 API 中，只有 AVAudioUnitEQ，對應到 kAudioUnitSubType_GraphicEQ，換言之，我們不能在 AVAudioEngine 中使用對應到 kAudioUnitSubType_AUipodEQ 的 AVAudioUnit，蘋果也在 iOS 13 當中將 kAudioUnitSubType_AUipodEQ 列為 deprecated。

Audio Session

Audio Session 是一個只在 iOS 上，Mac OS X 沒有的 API，用途是用來描述目前的 App 打算如何使用 audio，以及我們的 App 與其他 App 之間在 audio 這部份應該是怎樣的關係。在 iPhoneOS SDK 問世的時候，只有 C API，到了 iPhoneOS 3.0 之後開始有 Objective-C API，放在 AVFoundation framework 中。

決定 Audio Session 的 Category

要讓我們的 App 在 audio 的表現上正常，我們必須要先選擇正確的 Audio Session category，然後將 Audio Session 設定成 active。

```
NSError *audioSessionError = nil;
[[AVAudioSession sharedInstance] setCategory:AVAudioSessionCategoryPlayback error:&audioSessionError];
[[AVAudioSession sharedInstance] setActive:YES error:&audioSessionError];
```

Audio Session 的 category 包括：

- AVAudioSessionCategoryAmbient
- AVAudioSessionCategorySoloAmbient
- AVAudioSessionCategoryPlayback
- AVAudioSessionCategoryRecord
- AVAudioSessionCategoryPlayAndRecord
- AVAudioSessionCategoryAudioProcessing
- AVAudioSessionCategoryMultiRoute

前兩種屬於「就算把聲音整個關了，其實也不影響 App 的功能」的 App，比方說各式各樣的手機遊戲：就算沒有聲音，我們還是有辦法玩神魔之塔、Angry Bird 或是 Candy Crush，不像 KKBOX 是一種音樂播放軟體，如果聽不到聲音，KKBOX 就等於沒有功能可言。

設了這種 category 之後，只要 App 退到背景、鎖上靜音鎖或是進入 lock screen，App 就不會發出聲音。至於 AVAudioSessionCategoryAmbient 與 AVAudioSessionCategorySoloAmbient 的差別就是是否允許可以與其他 App 混音，前者可以，後者不行，如果你的遊戲只有簡單的音效，像揮劍、開槍的聲音，沒有其他背景音樂，就不妨設定成 AVAudioSessionCategoryAmbient，讓用戶可以一邊玩遊戲，一邊聽 KKBOX。但如果你的遊戲有自己的背景音樂，不希望受到干擾，那就設成 AVAudioSessionCategorySoloAmbient，只要開始遊戲背景音樂，就會通知系統，要求像 KKBOX 這類背景播放音樂的 App 停止播放。

像 KKBOX 這類的媒體播放軟體，則應該將 Audio Session category 設定成 AVAudioSessionCategoryPlayback。這類的軟體可以在靜音鎖鎖上、進入 lock screen 播放聲音，也可以在背景播放。這種軟體在有來電、鬧鐘響起的時候，會收到 Interrupt 的通知，也必須處理這種通知。

如果你只要做一個錄音軟體，就設定成 AVAudioSessionCategoryRecord，至於 AVAudioSessionCategoryPlayAndRecord 則是用在像 Skype 這類的網路電話軟體，設定之後，一樣是在靜音鎖鎖上與 lock screen 時可以使用錄製功能。

最後兩種實在不常見，在網路上可以看到的資源也很少。根據蘋果文件，AVAudioSessionCategoryAudioProcessing 是用在純粹只使用蘋果的 Audio API 做音訊的處理，但是沒有任何的輸入與輸出。而設定了 AVAudioSessionCategoryMultiRoute 之後，會讓 iOS 裝置原本只會選擇一個單一的

輸入或輸出裝置的預設行為，變成可以同時使用多台裝置，比方說，我們原本用耳機孔接了耳機，接著用 Lighting 線接了某個喇叭之後，就變成 Lighting 輸出，設了 AVAudioSessionCategoryMultiRoute，就可以同時往耳機與 Lighting 輸出。這部份請參見 WWDC 2012 的說明。

決定 Audio Session Category 的時機

我們要設置正確的 Audio Session Category，才能夠正確使用 audio，不然在呼叫後續的 audio API 時就會遇到錯誤。所以我們設定 Audio Session Category 的時機，必須要在呼叫所有後續的 audio API 之前。

像如果我們沒有把 Audio Session Category 設定成 AVAudioSessionCategoryRecord 或 AVAudioSessionCategoryPlayAndRecord，當我們想要使用作為 Input 使用的 Audio Queue，或是使用 RemoteIO 的 bus 1 時，就會遇到錯誤。

而如果我們沒有把 Audio Session 設定成 AVAudioSessionCategoryPlayback，就在背景呼叫與播放相關的 Audio Queue API，或是用 AUGraphOpen 建立 AUGraph，都會遇到錯誤。

在 iOS 8 之前，我們沒有什麼機會可以在背景開啟 App，然後讓 App 繼續待在背景，從來不出現在前景。iOS 8 之前，App 開啟之後一定會先待在前景，用戶也得在前景選擇一首歌曲播放，所以我們可以在讓用戶在前景開始播放後，再設定 Audio Session Category。不過，iOS 8 之後有兩種方式會讓 App 在背景開啟，一種是 iOS 8 的 Interactive Notification，用戶可以直接在 Notification 的畫面中讓 App 做某件事情，而不用將 App 叫到前景，另外一種方式則是讓 Apple Watch 呼叫我們的 App。

為了確保 Audio Session Category 就在呼叫其他 audio API 之前被設定，我們就得在很早的地方就先設好 Audio Session Category，像是 App 一啟動時，或是在建立我們的 player class 的時候。

說到背景執行，其實 iOS 還有另外一項跟背景相關的限制，就是 iOS App 不可以在背景使用 OpenGL API，只要一在背景呼叫了 OpenGL API，就會馬上因為 exception 而 crash。所以，如果你寫了一個一個可以在背景播放 audio 的音樂 App，就不該在 UI 上使用 OpenGL。

KKBOX 在 2013 年的時候，暫時做了一個實驗性的功能，可以在 KKBOX 的 iOS App 這個播放器裡頭玩一個像是 Tap-Tap Revenge 的音樂節奏遊戲，因為這個遊戲是在以 OpenGL 為基礎的遊戲引擎上開發，所以，KKBOX 平時的 Audio Session Category 是 AVAudioSessionCategoryPlayback，但是進入節奏遊戲模式後，就必須切換到 AVAudioSessionCategorySoloAmbient，避免因為在背景呼叫 OpenGL 而 crash。

Audio Session Category Options

在設了 Audio Session Category 之後，還有許多細項可以設定，包括 Category Options 與 Mode。

先講 Category Options 的部份，共有以下幾個 option

- AVAudioSessionCategoryOptionMixWithOthers
- AVAudioSessionCategoryOptionDuckOthers
- AVAudioSessionCategoryOptionAllowBluetooth
- AVAudioSessionCategoryOptionDefaultToSpeaker

我們在設定 Category 的時候，可以同時設定 Category Mode，方法是從原本呼叫的 `-setCategory:error:`，換成 `-setCategory:withOptions:error:` 這一組 API。

在 KKBOX 的開發過程中，AVAudioSessionCategoryOptionDuckOthers 對我們來說影響比較大；並不是 KKBOX 使用了這種 option，而是當其他 App 設定了這種 Category Option 之後，KKBOX 會受到影響。這個 Category Option 只要一設定，就會讓系統中其他 App 發出的聲音的音量都減半，主要是一些車用導航軟體

會這樣設定 audio：當一套導航軟體會使用語音導引駕駛如何開車的時候，就會要求其他 App 降低音量，讓駕駛可以清楚聽到導航語音。

我們在 KKBOX 曾經收到這樣的客訴：用戶只要開啟了 Papago 的導航，就沒有辦法聽清楚 KKBOX 的音樂了。面對這樣的客訴，我們也只能夠請客戶去找 Papago，因為當 Papago 這樣設定 Audio Session Category Option 後，KKBOX 的音量一定會減半。

AVAudioSessionCategoryOptionAllowBluetooth 以及 AVAudioSessionCategoryOptionDefaultToSpeaker 與錄音有關，前者是允許拿藍芽裝置當做錄音輸入設備，後者則是在一邊錄音一邊播放的時候，允許預設使用喇叭作為輸出裝置。至於 AVAudioSessionCategoryOptionMixWithOthers 的使用情境有點微妙，如果一套音樂 App 設了這種 mode，就可以與其他音樂 App 做混音之後一起播放出來，但是在經驗中沒有看過這類的 App。

Audio Session Mode

絕大多數的 Audio Session Mode 都與如何錄音相關，設定了其中一個 mode 之後，就會決定聲音訊號從硬體送到我們的錄音 API 之間的過程中，會經過怎樣的處理後才送進來。蘋果定義了 Video Chat、Voice Chat、Game Chat、Video Recording 等類型的聊天情境，在不同情境下對人聲做了一定程度的強化，並且消除雜訊，如果想要盡可能錄製到所有的聲音，而不被 iOS 裝置本身過濾過，就切換到 AVAudioSessionModeMeasurement 這個 mode。

Interrupt

對於音樂類型的 App 來說，我們需要特別處理 Interrupt 與 Audio Route。Interrupt 是指當我們的音樂播放到一半的時候，系統發生了其他的事件，因此打斷我們的 App 造成我們必須要暫停播放，像是有電話打進來、鬧鐘突然響起，或是當我們的 App 在背景的時候，用戶在前景使用其他音樂或影片 App 播放音樂或影片。

當我們的 App 被打斷之後，可能需要回復播放，也可能不需要。像如果是來電或鬧鐘，那麼當來電或鬧鐘結束，就需要繼續播放；如果是用戶使用其他的 App 播放影音，就不應該播放。

Audio Session 在 iOS 6 之前是使用 delegate 的方式通知我們 Interrupt，在 iOS 6 之後改成使用 notification，當然用 notification 會比較好，因為同一個 App 可能好幾個地方需要處理 interrupt。

如果你一開始只有一個單純的音樂 App，那麼可能光用單一的 delegate 處理就夠了，可是，有一天這個音樂 App 被瘋狂地加入各種功能，包括播放 MV、播放演唱會 video 直播，甚至可以在裡頭玩節奏遊戲，不但是原本的audio player 需要管理 interrupt，在 MV 與演唱會直播的地方用到了AVPlayer，也要處理 interrupt，就得要分別通知了。

AVAudioSessionInterruptionNotification 就是發生 Interrupt 時會送出的 notification，我們可以從 notification 的 user info 中，透過 AVAudioSessionInterruptionTypeKey，判斷是屬於 Interrupt 開始或是結束的通知。

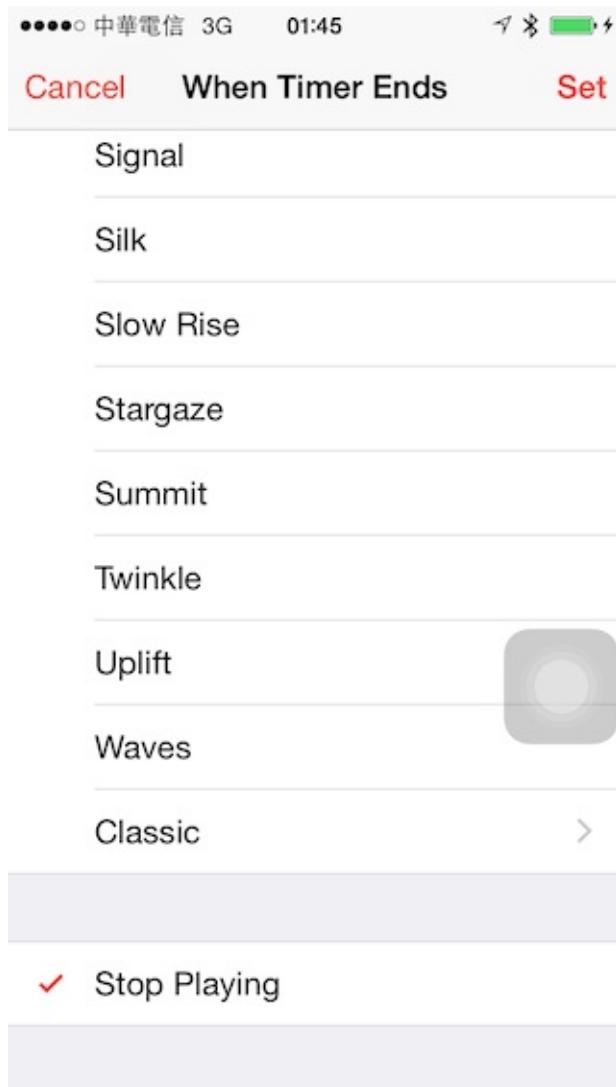
如果收到的 type 是 AVAudioSessionInterruptionTypeBegan，代表 Interrupt 開始，這時我們原本的 Audio Queue 或 Audio Unit Graph 就會被強制暫停，我們也需要將 UI 換成是暫停中的 UI；反之，如果收到 AVAudioSessionInterruptionTypeEnded，就代表 Interrupt 結束—不過，Interrupt 不一定代表我們應該繼續播放。

像來電或鬧鐘，我們會收到開始與結束的通知，但如果是其他音樂或影片 App 打斷我們，就不會收到結束的通知。

之所以說收到 Interrupt 結束不一定代表應該繼續播放，有幾個理由。一是，我們在收到 Interrupt 開始的通知的時候，我們的播放器不一定正在播放音樂，可能原本就是暫停的，結果電話打進來的時候沒在播放音樂，電話結束卻有音樂出來，怎麼看都是 bug。所以我們要透過 Interrupt 開始之前是否有在播放音樂，決定 Interrupt 結束的時候該做什麼。

收到 AVAudioSessionInterruptionTypeBegan 的時間，與我們的 Audio Unit Graph 被停止的時間雖然接近到可以說是同時，但還是有個先後順序，而這個先後順序在幾次系統改版的時候有改變過。在我們的經驗中，iOS 7 與之前，我們會先收到 AVAudioSessionInterruptionTypeBegan，Audio Unit Graph 才被暫停，但 iOS 8 之後則變成 Audio Unit Graph 被暫停之後，才收到 AVAudioSessionInterruptionTypeBegan 的通知，所以我們不能夠在收到 AVAudioSessionInterruptionTypeBegan 的時候去問 Audio Unit Graph 是否在播放，決定結束 Interrupt 時的狀態，因為在 iOS 8 上一定是關閉的，而必須要用其他方式記錄。

另外，我們會收到一種很奇特的通知，會告訴你 Interrupt 結束，但系統告訴你不該繼續播放。在系統的時鐘 App 的第四個 tab 中，我們可以設定倒數計時的碼錶，除了設定要倒數多久之外，還可以決定倒數截止的時候該做什麼，我們有兩個選擇：播放某個鈴聲，或是「停止播放」，這個「停止播放」選項會對所有在播放 audio 的 App 發送 Interrupt 通知，而且帶的 AVAudioSessionInterruptionOptionKey 並不是 AVAudioSessionInterruptionOptionShouldResume，如果我們這時候播放了，反而違反了 iOS 系統的行為。



Audio Route

一個夠盡責的音樂 App 還要注意一點：當用戶在拔除耳機的時候，我們應該要暫停播放音樂，因為不這麼做，當用戶在人多的公開場合不小心拔除耳機，就會導致聲音放出來，干擾周圍的其他人。

要知道用戶是否拔除了耳機，就要收取 Audio Route 變更的通知

(AVAudioSessionRouteChangeNotification) ，所謂的 Audio Route 就是聲音輸出的各種途徑，像內建喇叭、耳機、往 AppleTV AirPlay 投放，都是 Audio Route。

我們想知道拔除耳機這種狀況，就是在收到了 AVAudioSessionRouteChangeNotification 之後，判斷改變的理由，只要是 user info 裡頭的 AVAudioSessionRouteChangeReasonKey 是

AVAudioSessionRouteChangeReasonOldDeviceUnavailable，代表舊裝置不見了，就可以當做該停止播放的狀況，因為內建的喇叭不會無故不見，會消失的一定是像耳機等外接的裝置。

另外就像我們在講 notification 的時候講到的， AVAudioSessionRouteChangeNotification 是少數不在 main thread 發生的 notification，收到之後如果我們想要更新 UI，就得要再用 GCD 等方式，在 main thread 執行。

打造 Player 的完整功能

在開發音樂 App 的時候，如何使用底層 Audio API，以及管好 Audio Session 會是最困難的部份，接下來還要做一些事情補完整個 App 的功能，但相對來說難度容易許多。

音量控制

在一套音樂播放軟體中，我們往往需要能夠調整系統音量的 UI 介面。其實我們可以對 Audio Queue 或是 AUGraph API 設置音量，像 Audio Queue 可以用 AudioQueueSetParameter 設定 kAudioQueueParam_Volume，AUGraph 則可以對 mixer 設定 kMultiChannelMixerParam_Volume；不過目前可以看到所有的音樂App，只要出現調整音量的 UI，都不是調整這些內部元件的音量，而是整體的系統音量，大概是因為從第一支 iPhone 問世起，內建的音樂 App 就有調整系統音量的介面建立的設計慣例。

我們可以使用 Media Player Framework 中找到 MPVolumeView，這個 view 裡頭除了音量控制條之外，旁邊還會顯示跟 AirPlay 相關的控制按鈕。使用 MPVolumeView 會是提供音量調整介面最簡單的方法。

如果我們不想用 MPVolumeView，而是想設計自己的音量控制 UI，我們可以從 MPMusicPlayerController 裡頭找到 volume 這個屬性。MPMusicPlayerController 是一個用來在我們的 App 中，播放內建音樂 App 裡頭的歌曲的 class，不少運動 App 使用了這個 class，讓用戶可以在選擇記錄運動的時候，一邊選擇自己的運動歌曲，至於調整音量這個功能會放在這個 class 裡頭，說起來也頂奇怪的，想起來裝置的音量好像放在 UIDevice 還比較有道理。

MPMusicPlayerController 的 volume 選項在 iOS 7 的時候也被標成 deprecated，也沒有對應的其他 API，看來蘋果不太喜歡我們做自己的音量條，只希望我們使用 MPVolumeView。

另外，當系統音量改變的時候，我們會收到 MPMusicPlayerControllerVolumeDidChangeNotification 通知。

遙控器

我們之前在 [Responder](#) 這一章裡頭已經講過如何處理遙控器事件—在 iOS 7.1 之後，我們會偏好使用 MPRemoteCommandCenter 這組 API，然後對 MPRemoteCommandCenter 提供的許多 command，如 playCommand、stopCommand 等，設置 target/action。

我們在 MPRemoteCommandCenter 上實作的 command，除了會用在 lock screen 與控制中心（control center）外，也會影響到 iPhone 外接 CarPlay 以及其他汽車音響如何控制手機的音樂播放，這點我們會在下章 CarPlay 的部份說明。

不過，如果你所開發的音樂 App 除了播放音樂之外，有一天有人就突發奇想想在裡頭加入用 AVPlayer 播放 MV 與 Live 直播功能，還可以用 web view 看網路文章，文章裡頭還嵌入了 YouTube 影片，你除了會遇到 Audio Session 的問題：自己 App 內部的 audio player 與 video player 互相 interrupt，還會遇到你幫 audio player 設好的 MPRemoteCommandCenter 設定，被這些 video player 搶走了。

因此，我們需要在 AVPlayer 與 web view 裡頭的影片結束的時候，重設 MPRemoteCommandCenter 的設定。要知道 AVPlayer 與 MPMoviePlayer 結束的事件還算簡單，只要接收

AVPlayerItemDidPlayToEndTimeNotification 與 MPMoviePlayerPlaybackDidFinishNotification 等通知即可，但是 web view 裡頭的 YouTube 影片關閉卻沒有什麼明確的通知。

目前想到的解法是這樣：我們聽取某個 UIWindow 關閉的通知 UIWindowDidBecomeHiddenNotification，然後去檢查這個 window 的 subview 中是否包含 AVPlayerView，如果有的話，那大概就是播放 YouTube 影片的 video player 的 window 吧...。

MPNowPlayingInfoCenter

MPNowPlayingInfoCenter 是讓我們設置「現正播放」資訊的 class，我們只要對 MPNowPlayingInfoCenter 的 singleton 物件，設定 nowPlayingInfo 屬性，我們就可以將目前正在播放的歌曲名稱、封面圖等，顯示在 iOS 裝置的 lock screen 上，在 AirPlay 的時候也會投放到 AppleTV 上，當我們用 Lighting 等介面將 iOS 裝置連接到一些車用音響以及 CarPlay 系統的時候，也可以在車用音響的面板上顯示歌曲資訊。

nowPlayingInfo 是一個 NSDictionary，裡頭用到的 key 多半定義在 MPMediaItem 裡頭，MPNowPlayingInfoCenter 這個 class 裡頭多定義了幾個額外的 key。使用 MPNowPlayingInfoCenter 的時候要注意幾點：

1. 雖然你看 nowPlayingInfo 的 property 定義成 copy，但當你設定了 nowPlayingInfo 之後，其實 MPNowPlayingInfoCenter 會開另外一條背景 thread，在背景 enumerate 這個 dictionary。所以，如果你建立了一個 mutable array，把這個 mutable array 設定成 nowPlayingInfo，然後繼續改動這個 mutable array，你會遇到一邊 enumerate 一邊改動而造成的 crash。
2. 在設定 MPMediaItemPropertyPersistentID 的時候，這個 ID 應該要是 NSNumber，但如果你不小心把這個 ID 設成了字串，絕大多數狀況下沒事，但是當你接上一些車用音響的時候，卻可能把這些車用音響系統搞當。這是我們拿著筆記型電腦在地下停車場修了兩個小時 Bug 後得到的心得。

AirPlay

iOS 裝置可以透過 AirPlay，將音訊傳到 AppleTV、HomePod 或是其他智慧型電視上播放。在我們自己的 App 中，其實不用額外實作與 AirPlay 相關的各項功能，用戶可以直接在控制中心（control center）中指定要用 AirPlay 輸出。不過，您也可以在 App 的畫面中，加上 AVRoutepickerView，方便用戶快速切換輸出裝置。

CarPlay

蘋果在 iOS 7.1 開始推出 CarPlay 功能。CarPlay 的使用體驗是由 CarPlay 專屬的車用音響系統以及 iOS 兩者共同創造的，CarPlay 車用音響會有一個小型的觸控螢幕，當用戶的 iPhone 連接到音響系統後，就會把這個小螢幕變成另外一個 iPhone 的延伸螢幕，在這個螢幕上，會用圖示、文字都比較大的 UI 設計，列出 iOS 的一些基本 App，包括電話、簡訊、音樂、地圖等，由於介面中的元素較大，加上可以用 Siri 語音操作，就讓用戶在開車的時候，比較方便操作這些功能，也更能顧及行車安全。



蘋果開放軟硬體廠商開發 CarPlay 相關應用，如果你是汽車、或是汽車音響廠商，你可以針對車輛的功能開發專屬的 App，像是可以用 CarPlay 操作開關車窗、調整空調等，至於一般會在 App Store 上的 App，也只開放了導航與音樂服務兩種類型。如果你想要讓你的 App 出現在 CarPlay 的延伸螢幕中，在 code sign 的時候還需要加上額外的 entitlement，這個 entitlement 需要額外向蘋果申請，也大概會是一般開發者進入 CarPlay 開發的門檻。

蘋果不允許音樂類型的 App，像 KKBOX，在實作 CarPlay 功能的時候，客製自己的使用者 UI，而是只能夠用一種階層式的方式瀏覽 App 提供的內容，從中挑選想要播放的歌曲/歌單。也就是說，蘋果設計好了 CarPlay 上的音訊類型 App 的 UI，第三方 App 只能夠提供一種階層式的資料，讓 iOS 把我們提供的資料填入到 CarPlay UI 裡頭。蘋果提供了一個叫做 MPPlayableContentManager 的 class，我們可以指定 MPPlayableContentManager 的 data source 與 delegate，透過 data source 與 delegate 提供資料。

實作 CarPlay 功能

要讓用戶可以完整使用 CarPlay，我們需要…

- 實作 MPRemoteCommandCenter

- 實作 MPPlayableContentManager 的 data source 與 delegate
- 在放時在 MPNowPlayingInfoCenter 填入歌曲資訊

這個部分其實蘋果並沒有說得很清楚，在指定 MPPlayableContentManager 的 data source 與 delegate 之前，必須先設定 MPRemoteCommandCenter 當中的指令，不然，即使設定了 MPPlayableContentManager 的 data source，MPPlayableContentManager 也不會開始向 data source 要求資料。倒是 MPNowPlayingInfoCenter 可以稍晚設定。

再談 MPRemoteCommandCenter

一般指令

一般來說，我們至少會實作以下的 MPRemoteCommandCenter 指令：

- playCommand：開始播放
- pauseCommand：暫停播放
- stopCommand：完全停止播放。Pause 與 Stop 的差別在於，Pause 只是停止目前正在播放用的 Audio Graph/AVAudioEngine/Audio Queue，但是 Stop 會完全放開目前播放器元件參考到的歌單/歌曲物件
- togglePlayPauseCommand：檢查目前是否正在播放，播放中就執行 pause，反之則執行 play
- previousTrackCommand：跳到前一首歌曲
- nextTrackCommand：跳到下一首歌曲

播放模式

然後以下幾個指令需要特別注意：

- changeRepeatModeCommand：修改循環模式，包括循環播放、不循環播放、單手循環播放
- changeShuffleModeCommand：修改播放模式，包括循序播放、隨機播放...等

在 MPRemoteCommandCenter 當中絕大多數的指令，都是用戶真的做了手動操作、在各種地方按下按鈕之後才觸發，但是 changeShuffleModeCommand 不一樣，如果我們實作了 changeShuffleModeCommand，在用戶接上了一般的車用音響之後，就會被直接呼叫一次。

iPhone 除了支援 CarPlay 車用音響之外，也支援更早之前的車用音響。蘋果在 2001 年就推出了最早的 iPod，在 iPhone 推出之前，就已經有一套讓車用音響支援 iPod 的協定，所以，在這樣的車用音響上，iPhone 會被當成是一支 iPod，也就是說，MPRemoteCommandCenter 的指令，其實不只會用在 CarPlay 車機上，也會用在非 CarPlay 車機上。所以，當我們在實作 changeShuffleModeCommand 與 changeShuffleModeCommand 的時候，必須參考從外部傳進來的新狀態。像是：

先指定 changeRepeatModeCommand 與 changeShuffleModeCommand 的 target/action：

```
center.changeRepeatModeCommand.addTarget(self, action: #selector(changeRepeatMode(_)))
center.changeShuffleModeCommand.addTarget(self, action: #selector(changeShuffleMode(_)))
```

實作方式：

```
@objc func changeRepeatMode(_ event: MPChangeRepeatModeCommandEvent) -> MPRemoteCommandHandlerStatus {
    let type = event.repeatType
    /// 使用傳入的 repeatType
```

```

    return .success
}

@objc func changeShuffleMode(_ event: MPChangeShuffleModeCommandEvent) -> MPRemoteCommand
HandlerStatus {
    var type = event.shuffleType
    /// 使用傳入的 shuffleType
    return .success
}

```

播放進度

跟前述「播放模式」相關的指令一樣，如果我們想要讓我們的 App 可以從待機畫面 /CarPlay 畫面中，在播放進度列上拖拉調整，就要實作 changePlaybackPositionCommand。在實作 changePlaybackPositionCommand 的時候，也需要使用傳入的時間。

首先指定 target/action：

```
center.changePlaybackPositionCommand.addTarget(self, action: #selector(changePlaybackPosition(_:)))
```

然後從傳入的 event 取出用戶想指定的播放時間位置：

```

@objc func changePlaybackPosition(_ event: MPChangePlaybackPositionCommandEvent) -> MPRemoteCommandHandlerStatus {
    let time = event.positionTime
    self.seek(to: time) // 要求播放器調整到指定的位置
    self.updateNowPlayingInfo() // 更新 MPNowPlayingInfoCenter 中的資料
    return .success
}

```

以下也是可以用來調整播放進度的指令：

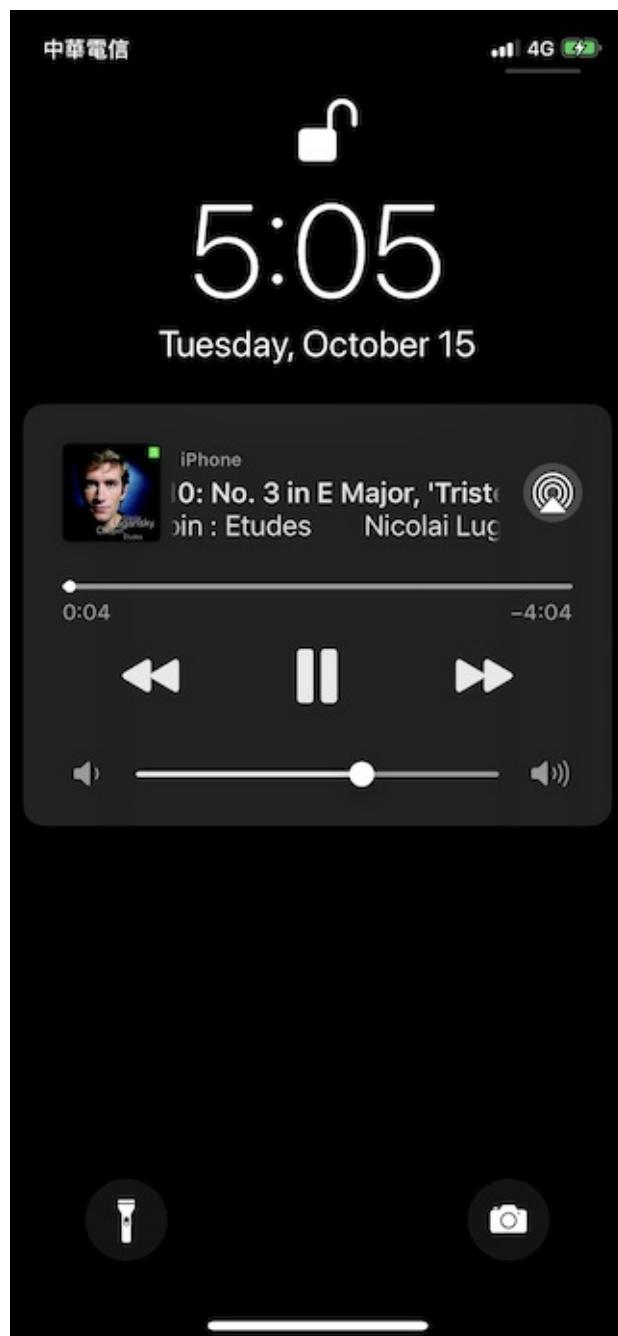
- seekBackwardCommand
- seekForwardCommand
- skipBackwardCommand
- skipForwardCommand

喜愛歌曲 (Feedback Commands)

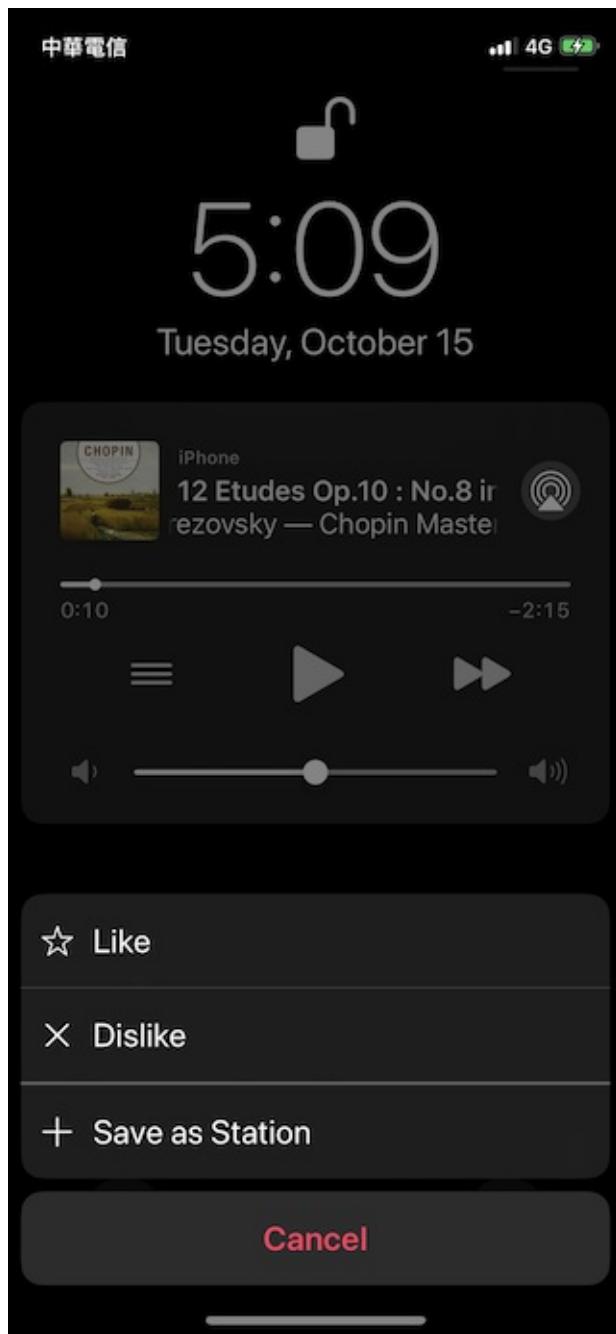
在 iOS 的待機畫面中，其實支援兩種不同模式的控制方式：

- 一般模式：可以跳到前一首、下一首歌曲，以及調整播放進度
- 電台模式：只能夠跳到下一首，不能夠跳到前一首，而前一首歌曲的按鈕位置變成一個選單，在選單中，可以讓用戶決定是否要將歌曲加到「我的最愛」或是「書籤」中

一般模式如下圖：



電台模式如下圖：



只要實作了以下指令，就會進入電台模式：

- likeCommand：將目前歌曲加入到我的最愛
- dislikeCommand：將目前的歌曲移出我的最愛
- bookmarkCommand：將目前歌曲加入書籤

不過，當我們想要從電台模式跳回一般模式的時候，只是將這些 command 設成 enabled 為 NO 是不夠的，必須要呼叫 `removeTarget` 才行。

另外，在 iOS 11 上，如果呼叫了這些指令，也會同時觸發 `seekForwardCommand`，而在呼叫 `nextTrackCommand` 與 `previousTrackCommand` 的時候，也會莫名其妙的呼叫到 `seekBackwardCommand`。由於這些行為都不符合預期，我們建議在 iOS 11 上，不要實作 `seekForwardCommand` 與 `seekBackwardCommand`。

實作 MPPlayableContentManager 的 Data Source 與 Delegate

為了方便起見，我們實作了一套叫做 [KKCarPlayManager](#) 的 library，實作 MPPlayableContentManager 的 data Source 與 delegate，並且在 GitHub 上面公開程式碼，可以用 CocoaPods 或 Swift Package Manager 的方式載入。 MPPlayableContentManager 會向他的 data source 要求每個層次的 MPContentItem 物件，組合成一個樹狀結構，最後形成在 CarPlay 螢幕上的選單介面。

在 KKCarPlayManager 中，我們有一個繼承自 MPContentItem 的 KKBasicContentItem，要完成整個 CarPlay 的功能，就是按照實際情況，繼續建立更多 KKCarPlayManager 的 subclass。

這個 item 上加了一個叫做 children 的 property，以及這兩個 method：

- `loadChildren(callback:)`
- `play(callback:)`

在這個樹狀結構中，會分成是目錄的節點、或是可以播放的節點，你可以用 MPContentItem 的 `container` 與 `playable` 這兩個 property，來標示是哪一種節點。如果某一層的節點是一個目錄的話，就要實作 `loadChildren(callback:)`，這代表我們發現用戶想要開始載入某一層目錄當中的資料，在載入成功之後呼叫傳入的 callback block。而如果是用來播放用的節點的話，就要實作 `play(callback:)`，無論成功或是失敗，都要呼叫 callback block。

在連接 CarPlay 裝置時 Audio Graph 的行為

在使用 Audio Graph 的時候，我們要盡量避免呼叫了 `AUGraphStop()` 之後，馬上再呼叫 `AUGraphStart()`。在一般的狀況下，這樣寫並沒有什麼問題，但是當 iOS 裝置接上了 CarPlay 車機之後，這樣呼叫的時候，`AUGraphStart()` 就會發生失敗而無法繼續播放。

MFI 助聽器

在 iOS 裝置上，除了可以使用內建喇叭、線接與藍芽耳機、AirPlay、CarPlay 與其他車用音響等方式播放音樂之外，還有一種比較少人注意到的裝置，就是 MFI (Made for iPhone) 助聽器。

從 iPhone 5 之後的機種開始，用戶在購買 MFI 助聽器之後，可以在「設定」>「輔助使用」>「聽力」，然後選取「助聽裝置」，與專屬助聽器連接，之後，用戶就可以直接從助聽器當中撥打電話，也可以從助聽器收聽到 iPhone 正在播放的音樂或影片的聲音。

在台灣，我們從 2018 年初開始，陸續收到用戶在使用 MFI 助聽器收聽音樂的客訴與需求，也就是從這個時候開始，MFI 助聽器逐漸在台灣普及，包括像是 Signia Pure X 等機種。相關資料可以參考蘋果官網的說明：[使用 Made for iPhone 助聽裝置](#)。

在開發聲音相關的 App 的時候，我們需要注意，當 iPhone 連接到 MFI 助聽器的時候，iOS 會開始跟你要求更大的 Audio Buffer。在一般的狀況下不太會有問題，但假如你改動了 AVAudioSession 的 Preferred IO Duration (參考 [setPreferredIOBufferDuration:error:](#))，你又使用 Audio Graph 播放的話，就可能得注意資料量太大而無法順利通過 Audio Graph 中的 Audio Unit 的問題。

我們在前面的章節〈[在 AUGraph 中串接 AudioUnit](#)〉就提到，AUGraph 會跟我們要求多少資料會變動的，平常的時候，一次會跟我們要求 1024 個 frame，但是當 iOS 裝置在 lock screen 的時候，基於節電的理由，會變成一次跟我們要比較多的資料，變成 4096 個 frame。但如果改動了 Preferred IO Duration，又接上助聽器，就很有可能會出現比 4096 更大的 frame per slice。

要解決這個問題，我們的方法是，當我們發現 frame per slice 太大，就用

`setPreferredIOBufferDuration:error:`，再把 Preferred IO Duration 改回來。

我們可以拿前面在〈[在 AUGraph 中串接 AudioUnit](#)〉當中的範例，解釋如何修正這個問題。我們可以在 Remote IO 的 Audio Unit 上 (`outputUnit`)，多加一個 render callback，攔截可能出現的錯誤，我們建立一個叫做 `_registerOutputRenderCallback` 的 method，以及一個叫做 `KKResetPreferredIOBufferDuration` 的 function。

```
static void KKResetPreferredIOBufferDuration() {
    NSError *error = nil;
    if (![[AVAudioSession sharedInstance] setPreferredIOBufferDuration:4096.0 / 44100.0 error:&error]) {
        NSLog(@"----- setPreferredIOBufferDuration failed: %@", error);
    }
}

- (void)_registerOutputRenderCallback
{
    AURenderCallbackStruct callbackStruct;
    callbackStruct.inputProcRefCon = (__bridge void *)(self);

    callbackStruct.inputProc = KKAUDIOOutputNodeRenderCallback;
    OSStatus status = AudioUnitSetProperty(outputUnit, kAudioUnitProperty_SetRenderCallback,
                                           kAudioUnitScope_Input, 0, &callbackStruct, sizeof(callbackStruct));
}
```

在 buildOutputUnit 的最後的地方呼叫：

```
[self _registerOutputRenderCallback];
KKResetPreferredIOBufferDuration()
```

至於 KKAUDIOOutputNodeRenderCallback 則是寫成

```
static OSStatus KKAUDIOOutputNodeRenderCallback(void *userData, AudioUnitRenderActionFlags *ioActionFlags, const AudioTimeStamp *inTimestamp, UInt32 inBusNumber, UInt32 inNumberFrames, AudioBufferList *ioData) {
    KKAUDIOGraph *self = (__bridge KKAUDIOGraph *)userData;

    OSStatus status = AudioUnitRender(EQUnit, ioActionFlags, inTimestamp, inBusNumber, inNumberFrames, ioData);

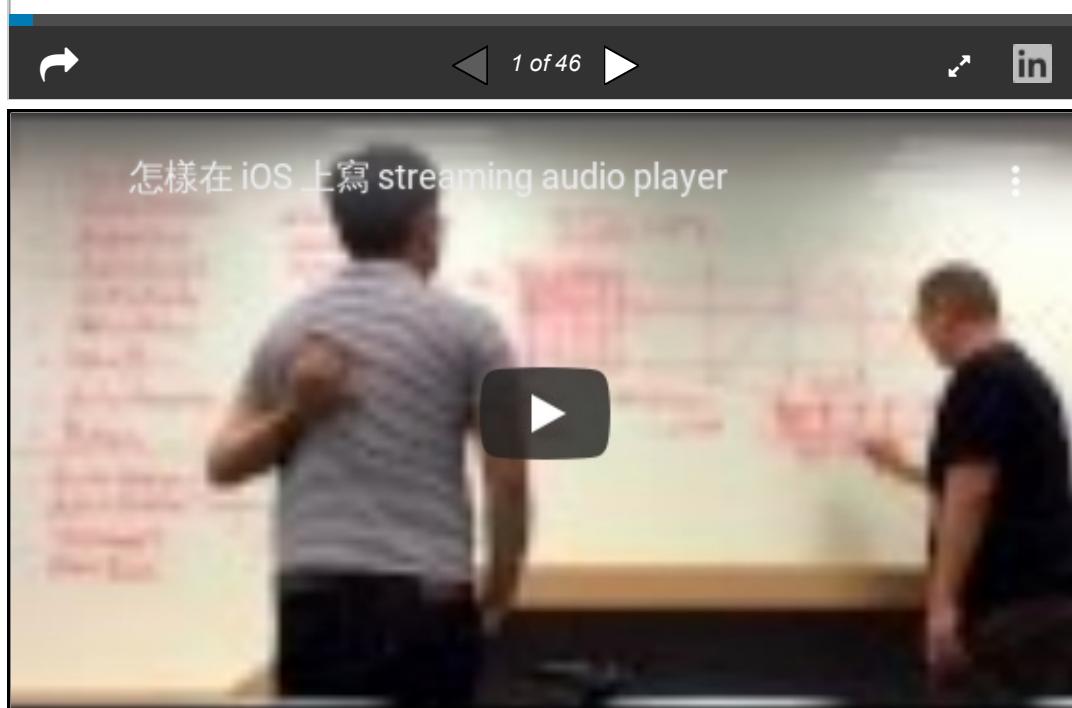
    if (status != noErr) {
        // 在 Frame per slice 太大的時候，就呼叫 KKResetPreferredIOBufferDuration() 重設。
        if (status == kAudioUnitErr_TooManyFramesToProcess) {
            KKResetPreferredIOBufferDuration();
        }
        _fillAudioBufferListWithSilence(ioData);
        *ioActionFlags |= kAudioUnitRenderAction_OutputIsSilence;
        return status;
    }

    return noErr;
}
```

教學影片

Mac OS X 與 iOS 的 Audio API

楊維中 a.k.a zonble
zonble@gmail.com



相關閱讀

- 蘋果官方文件 [Multimedia Programming Guide - Using Audio](#)
- [Wikipedia: 44,100 Hz](#)
- 蘋果官方文件 [AV Foundation](#)
- 蘋果官方文件 [Audio Queue Services Programming Guide](#)
- 蘋果官方文件 [Audio Unit Hosting Guide for iOS](#)
- 蘋果官方文件 [Audio Toolbox Framework Reference](#)
- 蘋果官方文件 [Audio Unit Processing Graph Services Reference](#)
- 蘋果官方文件 [Core Audio Glossary](#)
- [WWDC 2012 Audio Session and Multiroute Audio in iOS](#)
- [WWDC 2013 What's New in Core Audio for iOS](#)
- [WWDC 2014 What's New in Core Audio](#)
- [WWDC 2014 AVAudioEngine in Practice](#)
- [WWDC 2015 What's New in Core Audio](#)
- [WWDC 2015 Audio Unit Extensions](#)
- [Audio Tutorial for iOS: File and Data Formats \[2014 Edition\]](#)
- [Friday Q&A 2012-10-12: Obtaining and Interpreting Audio Data](#), Mike Ash
- [Friday Q&A 2012-11-02: Building the FFT](#), Chris Liscio
- [Friday Q&A 2012-10-26: Fourier Transforms and FFTs](#), Mike Ash
- [Why CoreAudio is Hard](#), Mike Ash

練習：iOS Audio Player 開發

在 iOS 平台上寫一個 Audio Player

- 使用 Audio Queue 或 Audio Unit Processing Graph API
- 具備可以播放在網路上的遠端音檔的能力
- 在 App 中有一份播放歌曲的列表，當一首歌播放完畢之後，可以繼續播放下一首，直到播完整份列表
- 要能知道目前歌曲的播放進度
- 要有可以對歌曲 random seek 的功能
- 可以背景播放
- 可以正確處理 Audio Session 傳來的 Interrupt 事件
- 拔除耳機的時候，這個 App 要能夠暫停播放音樂
- 在 App 中有介面可以調整音量

Auto Layout

Auto Layout：透過描述 views 之間相對關係動態計算 views 尺寸與位置的方法。

回過頭來看蘋果會在 WWDC2012 發表 Auto Layout 不是沒有原因的，對應同年9/21發表螢幕尺寸較長的 iPhone 5，蘋果已經找到因應多尺寸螢幕UI的解決方案。

Auto Layout 之於 Frame Layout 就好比高階程式語言之於組合語言，所以我們也可以說 Auto Layout 是 UI 界的高階語言，當然它也比較容易理解同時效能也會比直接寫低階語言慢。透過 Constraint 來描述 view 之間的關係(通常是距離)，Runtime 會把所有的 Conastaints 丟到 Auto Layout Engine 去計算，在由 Engine 告訴每一個 view 該擁有的尺寸及位置。

案例：iOS team 一行人在路上撿到 800 塊，他們正在煩惱著怎麼分這筆錢，如果以 Frame Layout 來分的話 不管撿到多少錢，Zonble 拿 500，清煩拿300，賣摳拿100，依序每個人有拿固定的數目，但就會有人拿不到。

如果以 Auto Layout 來分的話 就會變成，Zonble 拿的比清煩多200，至少要拿 200; 清煩拿的比賣摳多 100，至少要拿 50; 最多只能拿 100

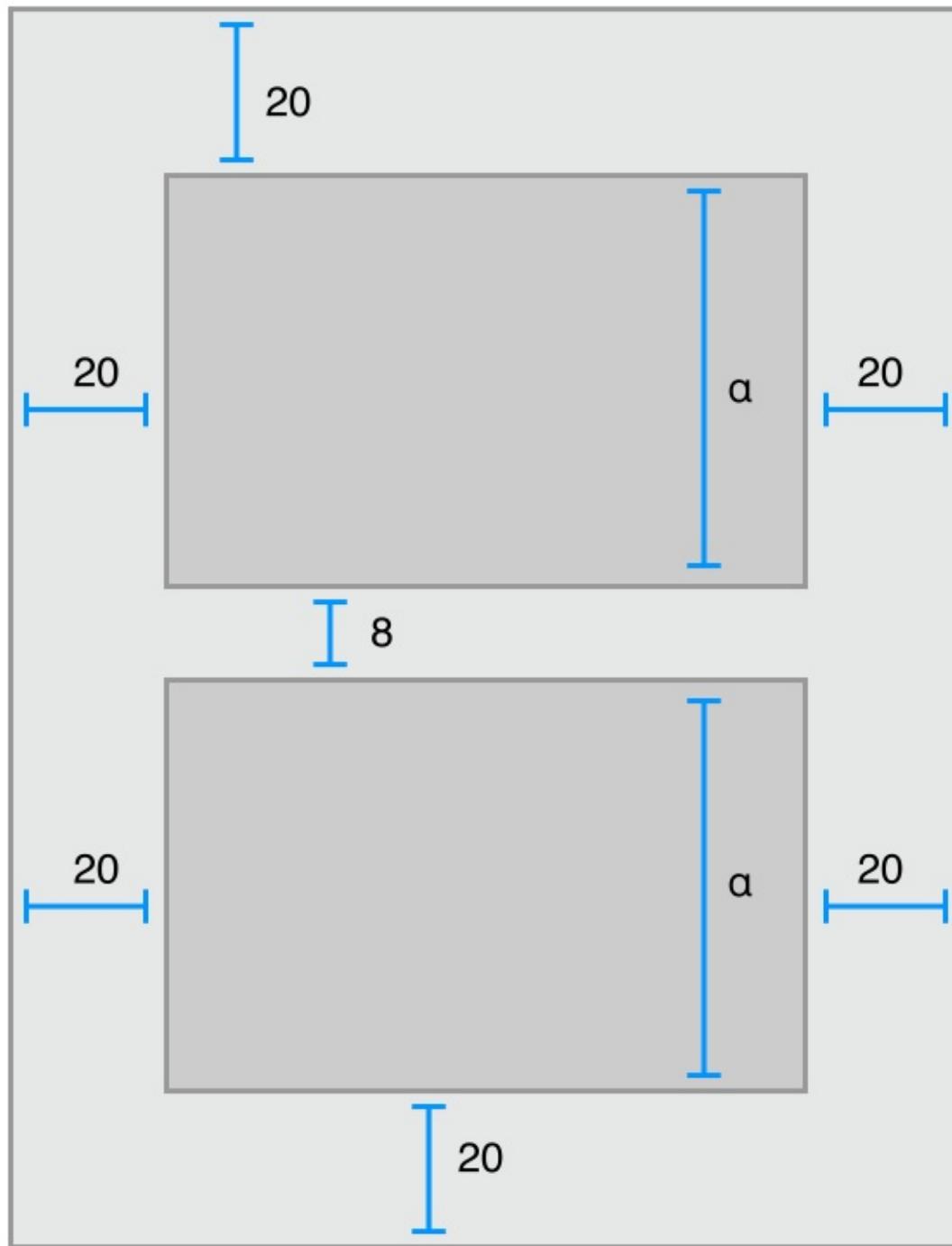
如果他們撿到 1500，會怎麼分？

在有 Auto Layout 之前即便我們有 Interface builder 的輔助，我們還是必須根據螢幕尺寸的變化去計算各個 view 相對的 frame。光是面對多尺寸螢幕裝置就已經夠複雜，當你的 app 需要支援多語系 (Localization) 面對**不同語言字串的長度**; 不同語系的閱讀方向(阿拉伯文是由右至左書寫); 不同 OS 版本使用不同字型(iOS 8 Helvetica Neue, iOS 9 San Francisco fonts)，而且 San Francisco font 還會根據字型大小自動調整字元間距；更慘絕人寰的是，WWDC2015 發表的 multitasking on iPad，一個畫面同時執行 2 個 app，意味著 Layout 不再是固定的幾種變化，而是幾百種變化。種種跡象都顯示著 Frame layout 已經不再適用，是時候擁抱 Auto Layout。

關於 Auto Layout 我們有寫一些 Sample code 可以參考 <https://github.com/gliyao/LCAutolayoutExample>

什麼是約束 Constraint?

Constraint：描述 views 之間相對關係



如上圖所示，所有藍色的線條皆為 Constraint，用來描述 view 之間的關係。

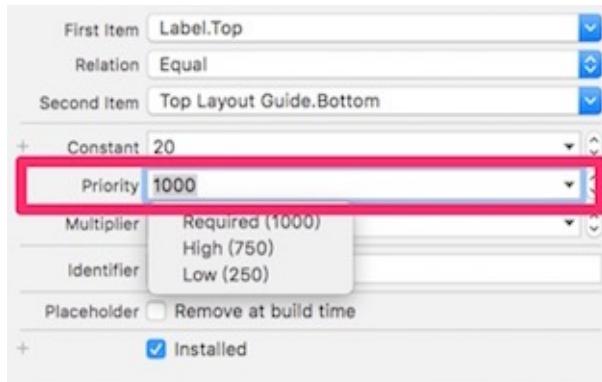
NSLayoutConstraint

```
@interface NSLayoutConstraint : NSObject  
@property UILayoutPriority priority; // 1~1000  
@property CGFloat constant;
```

```
@end
```

```
typedef float UILayoutPriority;
static const UILayoutPriority UILayoutPriorityRequired NS_AVAILABLE_IOS(6_0) = 1000;
static const UILayoutPriority UILayoutPriorityDefaultHigh NS_AVAILABLE_IOS(6_0) = 750;
static const UILayoutPriority UILayoutPriorityDefaultLow NS_AVAILABLE_IOS(6_0) = 250;
static const UILayoutPriority UILayoutPriorityFittingSizeLevel NS_AVAILABLE_IOS(6_0) = 50
;
```

在 Interface builder 之中 priority 可以設定 1~1000，但如果要在程式碼中設定，最高只能設定到 **999**。



若想要在 Runtime 時動態改變 Layout，可以透過變更 Constraint 數值(priority or contant) 或是新增/移除 Constraints 觸發 Auto Layout Engine 重算 Layout。

```
- (IBAction)makeLabelSamlllerAction:(id)sender
{
    self.userNameLabelWidth.constant = 20; // from 100 to 20
}
```

Intrinsic Content Size

依照內容尺寸去自動調整元件 Layout 的原則 view 依照內容尺寸壓縮/撐開 super view 的原則

當我們希望元件的 Layout 去自動適應內容的尺寸，尤其是面對 Localization 不同語言有著不同的文字尺寸的時候。

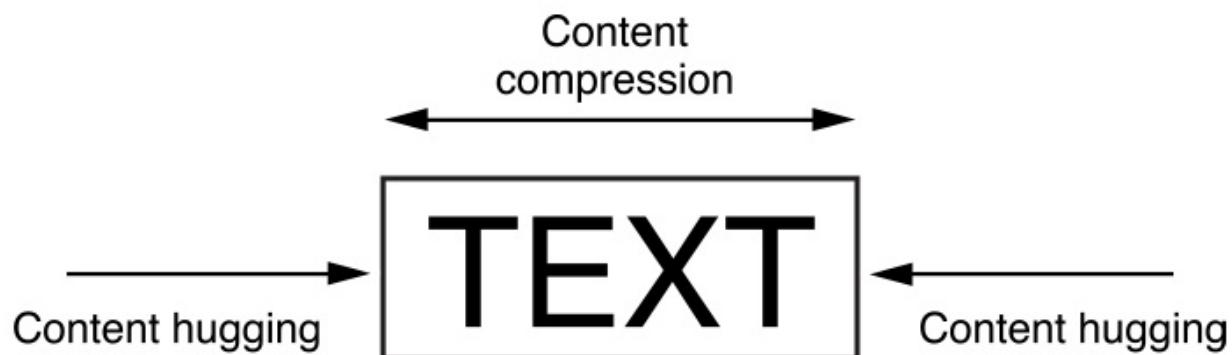
例如說一個 送出(submit) 的按鈕使用不同語言時，它要根據文字的寬度去調整 Button 的 Layout，在此情境下給一個固定寬度的 Constraint 不能滿足我們的需求。

-----	-----
submit	送出
-----	-----



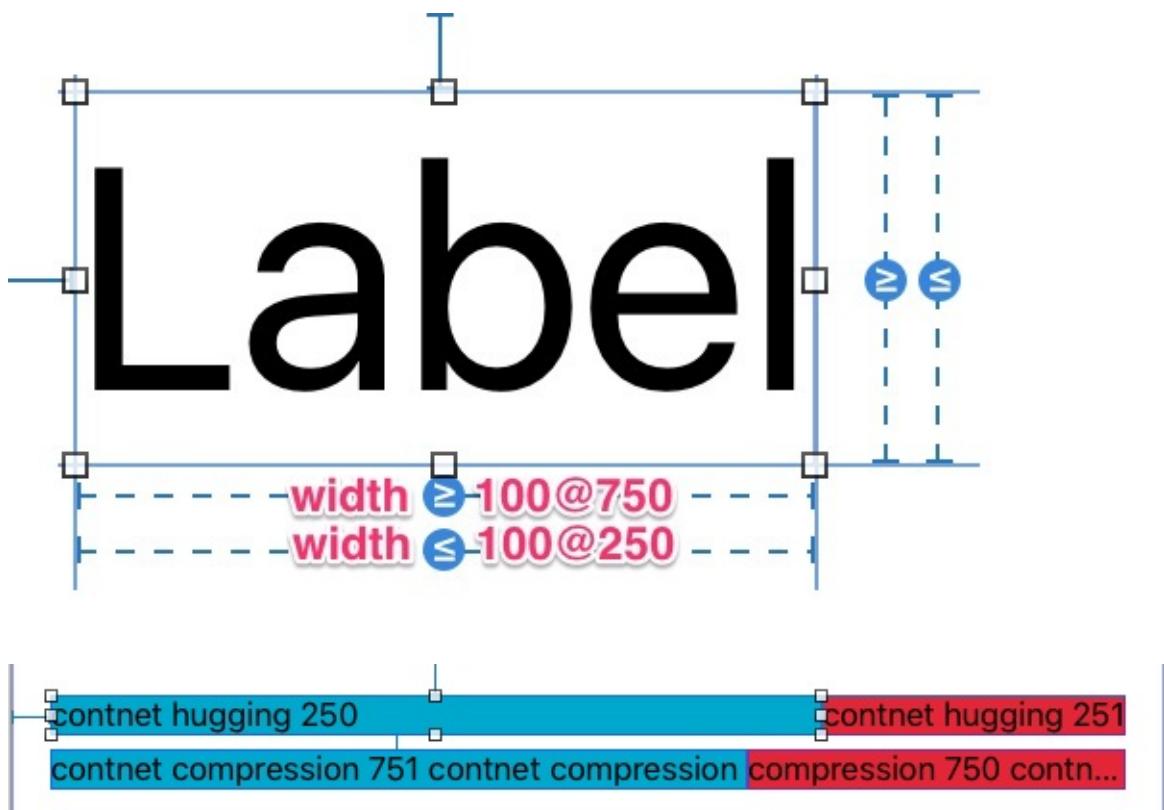
Auto Layout 將 適應內容 拆解成兩個概念，分別是 內容吸附 (Content Hugging) 和 內容壓縮阻力 (Content Compression Resistance) °

- 內容吸附 - 當元件被拉伸時(內容尺寸比 super view 小的時候)，向內拉動的力量
- 內容壓縮阻力 - 當元件被擠壓時(內容尺寸比 super view 大的時候)，向外推擠的力量



原理 Auto Layout Engine 會將固有內容尺寸和這些優先權被轉換為 Constraints。一個內容尺寸為 {100, 30} 的 label，預設水平/垂直壓縮阻力優先值為 750，水平/垂直的內容吸附性優先值為 250，這四個約束條件將會生成。

```
H:[label(>=100@750)] (content compressing : width >=100@750)
H:[label(<=100@250)] (content hugging : width <= 100@250)
V:[label(<=30@250)] (content hugging : height <= 30@250)
V:[label(>=30@750)] (content compressing : height >=30@750)
```



如上圖所示，第一組 Labels 都是擁有的空間大於內容尺寸，此時會比較哪一個 Label 的 content hugging priority 比較高，該 Label 的 Layout 就會優先包覆其 Label。

而在這組情境下，紅色Label 有著較高的 hugging，所以藍色Label 會被拉扯過去。

而第二組則是兩個 Label 都是內容大於所擁有的空間，所以比較誰的 content compression resistance priority 比較高，該 Label 就可以優先取用所需的 UI 空間。

在這組情境下，藍色 Label 的 compression 比較高，所以壓縮了紅色 Label 擁有的空間。

UIScrollView 與 Auto Layout

讓 UIScrollView 自己管理 content size

- 用一個 UIView 當做 container (contentView)
- container 與 self.view equal width

```
- UIView
  - UIScrollView
    - UIView (contentView) <---- 與 self.view equal width
      - UIView (topContainer)
      - UIView (bodyContainer)
      - UIView (footerContainer)
      ...etc
```

xib 在 UIViewController lifecycle 的陷阱

小心別中 frame 的陷阱`

當你使用 .xib 從 nib loadView 時，一開始的 frame 會是 nib 裡面的 frame，而它會延遲到 - updateViewConstraints: 的時候才會得到正確的 frame size，而使用 .storyboard 時就可以一開始拿到正確的 frame size。

尤其是我們在 -loadView or -viewDidLoad 加入 CALayer 的時候，因為 self.view.frame 還沒更新導致加入的 layer 算錯。

```
// use xib
frame = {{0, 0}, {600, 600}}, loadView
frame = {{0, 0}, {600, 600}}, viewDidLoad
frame = {{0, 0}, {600, 600}}, viewWillAppear
===
frame = {{0, 0}, {414, 736}}, updateViewConstraints (1)
frame = {{0, 0}, {414, 736}}, viewDidLayoutSubviews (1)
frame = {{0, 0}, {414, 736}}, viewDidAppear
frame = {{0, 0}, {414, 736}}, updateViewConstraints (2)
frame = {{0, 0}, {414, 736}}, viewDidLayoutSubviews (2)

// use storyboard
frame = {{0, 0}, {414, 736}} loadView
frame = {{0, 0}, {414, 736}} viewDidLoad
frame = {{0, 0}, {414, 736}} viewWillAppear
frame = {{0, 0}, {414, 736}} updateViewConstraints (1)
frame = {{0, 0}, {414, 736}} viewDidLayoutSubviews (1)
frame = {{0, 0}, {414, 736}} viewDidAppear
frame = {{0, 0}, {414, 736}} updateViewConstraints (2)
frame = {{0, 0}, {414, 736}} viewDidLayoutSubviews (2)
frame = {{0, 0}, {414, 736}} updateViewConstraints (3)
frame = {{0, 0}, {414, 736}} viewDidLayoutSubviews (3)
```

Self-sizing UITableView Cell height (iOS 8 latter)

<http://www.appcoda.com/self-sizing-cells/>

首先在客製化的 UITableViewCell 設定好 Auto Layout，切記上下左右都要設定，要讓內容可以去撐開 superView

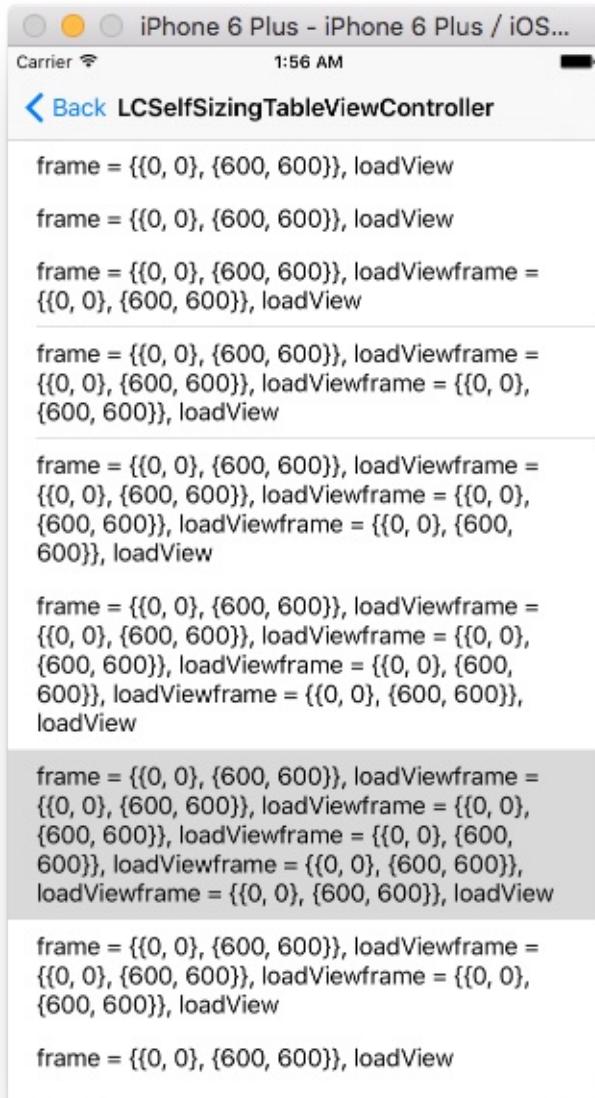


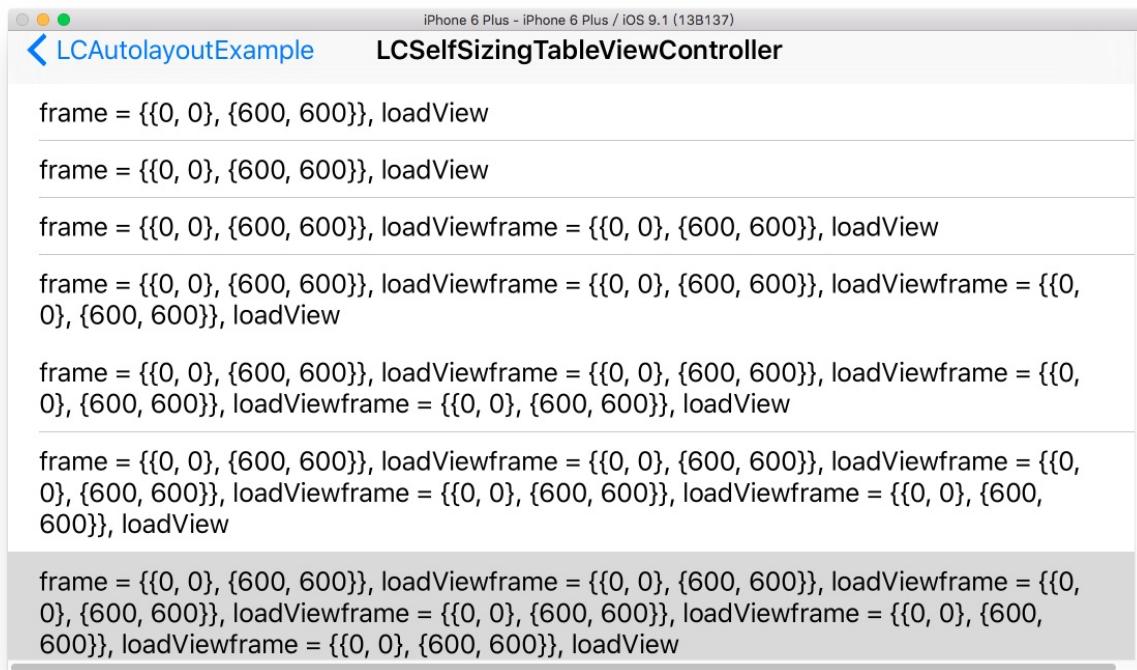
在 viewController 之中加入 `tableView.estimatedRowHeight` 與 `tableView.rowHeight = UITableViewAutomaticDimension`

```
self.tableView.estimatedRowHeight = 44.0;
self.tableView.rowHeight = UITableViewAutomaticDimension;
```

最後再 `viewDidAppear:` 中加入 `tableView.reloadData()`

```
- (void)viewDidAppear(BOOL: animated) {
    [self.tableView reloadData];
}
```





The screenshot shows a Xcode interface with a simulated iPhone 6 Plus - iPhone 6 Plus / iOS 9.1 (13B137) environment. The title bar reads "LCAutoLayoutExample" and "LCSelfSizingTableViewController". The main area displays a UITableView with several rows. Each row contains a single text element representing a frame definition. The frames are identical for all rows, starting with "frame = {{0, 0}, {600, 600}}", followed by "loadView". This pattern repeats six times, indicating six self-sizing table view cells.

```
frame = {{0, 0}, {600, 600}}, loadView
frame = {{0, 0}, {600, 600}}, loadView
frame = {{0, 0}, {600, 600}}, loadViewframe = {{0, 0}, {600, 600}}, loadView
frame = {{0, 0}, {600, 600}}, loadViewframe = {{0, 0}, {600, 600}}, loadViewframe = {{0, 0}, {600, 600}}, loadView
frame = {{0, 0}, {600, 600}}, loadViewframe = {{0, 0}, {600, 600}}, loadViewframe = {{0, 0}, {600, 600}}, loadViewframe = {{0, 0}, {600, 600}}, loadView
frame = {{0, 0}, {600, 600}}, loadViewframe = {{0, 0}, {600, 600}}, loadView
```

Animation with Auto Layout

要讓 view 依照 autolayout 去動畫只要在變更 constraint 後在 `UIView.animationWithDuration:` 中加入 `[self.view layoutIfNeeded]` 即可。

```
self.userNameLabelWidth.constant = 20; // from 100 to 20
[UIView animateWithDuration:0.25 animations:^{
    [self.view layoutIfNeeded];
}];
```

Accessibility

支援 Accessibility，就是讓具有各種身體障礙的用戶—尤其是視力不良的用戶—也可以使用你的 App。你現在在開發的是一套音樂 App，但是你可能因為 UI 設計不良，讓視力不良的用戶無法順利使用你的 App—因為眼睛看不到，所以也沒辦法用耳朵聽音樂，這實在太沒道理了。

蘋果在 iPhoneOS 3.0 的時候，就加入了三項為視力不佳者服務的設計—如果你沒有辦法清楚看到 iPhone 上預設大小的文字，你可以開啟畫面的局部放大功能，同時用三隻手指點按畫面，就可以放大手指所在的區域；如果覺得白底黑字不夠清楚，可以將畫面反白，切換成黑底白字。

而如果已經完全失去視力，完全無法透過視力操作電腦，iOS 提供一項名為 VoiceOver 的語音功能，開啟後會改變一些觸控的操作行為—原本只要按在畫面中某個元件上（例如按鈕等），就會使用這一個元件的功能，在開啟 VoiceOver 之後，單點一下，系統會先告訴你目前點到的地方是怎樣的東西，之後再連續點兩下，才是使用這個元件的功能。

因為 VoiceOver 與觸控介面，iOS 裝置變成非常適合盲人使用的資訊設備—在使用鍵盤滑鼠的操作介面中，不管怎樣，盲人都很難確定目前螢幕上滑鼠指標到底在什麼地方，但是在觸控介面上，便能夠較為清楚的選擇要點選的是畫面的上下左右；透過 VoiceOver，盲人朋友無需另外安裝軟體，就可以方便地收取郵件、閱讀網頁、當你用 iBook 翻閱的書籍時，可以幫你把書中的內容念出來…等。蘋果在推出 Apple Watch 之後，也一樣實作了 VoiceOver 功能。

在製作軟體時，我們不太需要關心局部放大與反白兩項功能，不過，在 VoiceOver 方面，還是要做一些工作，才能夠完全達成。

如果你的應用程式裡頭，都只有用到 iOS SDK 原本就設計好的那些 UI 元件，例如 UIButton、UILabel、UISlider … 等等，基本上就已經具備了支援 VoiceOver 的一定能力，在點選到這些元件上的時候，系統就會把裡頭的文字念出來。

但，假使你不是用文字代表這些元件的意義，而是放入圖片呢？如果圖片來自 bundle 的 resource 裡頭，你用 `[UIImage imageNamed:]` 載入圖片，那，VoiceOver 會把圖片檔名念出來，恐怕沒有人能聽的懂是什麼意思；而如果圖片是用程式碼產生，就什麼相關資訊都沒有了。

在 iOS 4.0 之後，無障礙支援又有另外一項意義，就是這個功能與新增的 UI 自動測試（UIAutomation）相關。當你用 Instument 執行某個 iPhone 應用程式的時候，可以選擇載入一個你之前撰寫好的 Javascript 腳本，接著就會按照腳本的內容，逐一進行各種 UI 測試項目；腳本的內容大概是，你可以透過 Javascript，將目前應用程式當前畫面當做物件呼叫，要求回傳畫面上所有可以使用的 UI 物件列表，然後，要求應用程式自己去點選這些物件。

每個 UI 元件的無障礙資源資訊，在 UI 自動測試中，就會被當做是這個 UI 物件的 id 使用。比方說，在畫面中有一個叫做「Edit」的按鈕，你從 Javascript 取得的物件列表放在 buttons 這個變數中（型態是 array），那麼，要取得這個按鈕，可以這麼呼叫：`buttons['Edit']`。如果要點選這個按鈕，就是 `buttons['Edit'].tap()`。

如何使用 VoiceOver

由於一般人往往不會開啟 VoiceOver，我們先花點時間，解釋這個功能如何使用。

開啟 VoiceOver

要開啟 VoiceOver，我們要先進入到系統設定中，選擇「一般」（General）設定，然後進入「輔助使用」（Accessibility）。進入到這一層設定之後，我們馬上便可以看到第一個選項就是 VoiceOver，我們可以在這邊選擇要開啟或關閉。



不過，每次都要進入設定切換 VoiceOver 相當麻煩，我們通常也會在「輔助使用」的設定中，設定最後一項設定「輔助使用快速鍵」（Accessibility Shortcut），然後把 VoiceOver 加入到 Shortcut 中。

••••• 中華電信 23:52 100% 

< 一般

輔助使用

通話時的周圍噪音。

左

右

調整左聲道和右聲道的音量平衡。

媒體

字幕與隱藏式字幕 >

口述影像

關閉 >

學習

引導使用模式

關閉 >

輔助使用快速鍵

詢問 >



設定了「輔助使用快速鍵」之後，我們就可以透過快速連按 Home 鍵三下，啟用我們剛才設定的功能。如果我們在「輔助使用快速鍵」設定中，只設定了 VoiceOver，那麼，就會直接開關 VoiceOver，如果是設定了額外的選項，像是縮放、反白顏色等，就會顯示一個額外的選單，顯示各種可以設定的「輔助使用」功能。

VoiceOver 的手勢

啟動了 VoiceOver 之後：

- 對畫面中任何一個 UI 元件單點，就會讓這個元件變成 focus 起來，VoiceOver 同時會朗讀這個 UI 元件的標題（這段文字叫做 accessibility label）。一些元件會有額外的說明，像是會念出這個元件怎麼使用，如「點選兩下以打開」，或，如果是一個 Slider，會告訴你目前 Slider 元件中的數值是多少（例如播放進度條會告訴你播到第幾分第幾秒），這些額外說明叫做 accessibility hint 與 accessibility value。
- 用單指點選兩下，就可以打開這個 UI 元件。
- 使用單指左右滑動 (Swipe)，可以移動到畫面中的前一個、或下一個 UI 元件。
- 如果某個元件可以調整裡頭的數值，像是 Slider，那麼就可以用往上或往下的 Swipe 手勢，切換裡頭的數值。往上是增加、往下是減少，以歌曲播放進度來看，往上就是往後快轉，往下就是往前快轉。
- 如果某個元件有多種動作—像是一個歌單 table view 裡頭的 cell，那麼，基本上就有播放歌曲與刪除這首歌曲這兩種動作—我們可以用上下的 Swipe 手勢，選擇要使用哪種動作，選擇之後就會改變單指點兩下的行為。比方說，歌單中的歌曲 cell 原本點兩下是播放歌曲，但我們往下 Swipe 一次換成了刪除

之後，單指點兩下就變成了刪除。

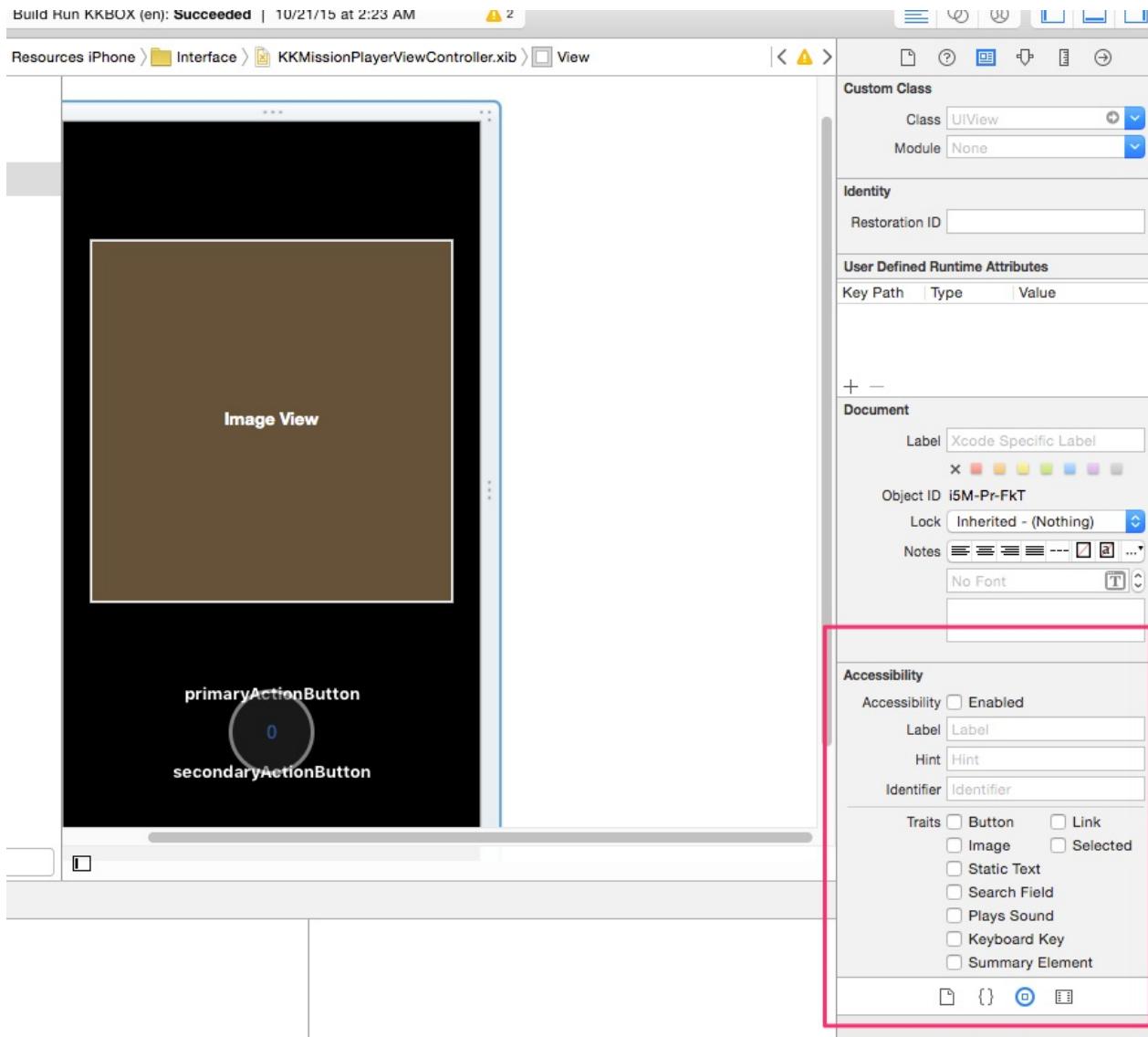
- 有時候我們會希望可以直接念出畫面中所有的 UI 元件。如果我們用雙指往下滑動，就會從目前 focus 的 UI 元件開始，念出下面的所有 UI 元件；如果是用雙指往上 Swipe，就會是從畫面的開頭開始，一直朗讀畫面中所有的元件。
- 如果遇到了像 Home Screen 的 App 列表的畫面，用上了 page control 換頁，平常我們用單指橫劃換頁，但 VoiceOver 下單指 Swipe 手勢已經被用掉了，於是我們要使用三指 Swipe 換頁。
- 如果用三指在畫面中點兩下，就可以讓 iOS 裝置的整個畫面變暗—因為對盲人朋友來說，如果完全只用 VoiceOver 操作，那麼就算畫面中什麼都不顯示，也不會影響操作，不顯示畫面反而有節電的作用。當我們在開發 VoiceOver 應用的時候，也可以使用這個模式，試試看即使什麼都看不到，我們是否有辦法使用我們的 App。再用三指點兩下就可以回復。

關於 iOS 上的 accessibility 功能如何使用，也可以參考蘋果官網上的說明：[iOS Accessibility](#)

基本支援

在絕大多數狀況下，Accessibility 資訊是加在 UIView 物件上的。所以，如果你只有用到 UIView 物件設計 UI，就可以選擇在 Interface Builder 裡頭，或是使用程式碼加入 accessibility 資訊。

在 Interface Builder 的 Identity Inspector 分頁中，就可以設定這個 UI 元件的相關資訊。



也可以透過程式碼設定，iOS SDK 定義了 `UIAccessibility` 這個 informal protocol（也就是 `NSObject` 的 category），所有繼承自 `NSObject` 的物件都具備此一 interface，你可以透過 `accessibilityLabel`、`accessibilityHint` 等屬性，標記 UI 元件的標題與詳細說明。像是，我們建立了一個叫做 `closeButton` 的按鈕時，就可以這樣寫：

```
[closeButton setAccessibilityLabel:NSLocalizedString(@"Close", @"");
[closeButton setAccessibilityIdentifier:@"Close"];
[closeButton setAccessibilityTraits:UIAccessibilityTraitButton];
```

說一下 accessibility identifier 與 accessibility label 的差別：VoiceOver 基本上只會念出 accessibility label，而不會念出 accessibility identifier，但我們往往也會額外設計 accessibility identifier，原因是，accessibility 資訊除了用在 VoiceOver 上，也會用在自動化 UI 測試上，當我們在寫 UI 自動化測試時，就會比較想使用 accessibility identifier 而不是 accessibility label 尋找特定元件，理由是 accessibility identifier 不會因為切換多國語系而影響，但是在不同語系下，accessibility label 會改變。

在設置 UISegmentedControl 的 accessibility 資訊的時候，要特別注意圖片型態的 segment。每個 segment 當中可以是文字或是圖片，我們可以用 `-setTitle:forSegmentAtIndex:` 或是 `-setImage:forSegmentAtIndex:` 設置 segment 中的內容，如果是文字的話，我們所傳遞進去的 title 自然會變成 accessibility label，但，如果是圖片呢？我們無法直接設置某個 segment 的 accessibility 資訊，所以遇到圖片，我們會直接對 UIImage 物件設置 accessibility 資訊—畢竟 UIImage 也是繼承自 NSObject 的物件，而 UIAccessibility 根本就是 NSObject 的 category。

所以遇到 UIImage 也可以這樣寫：

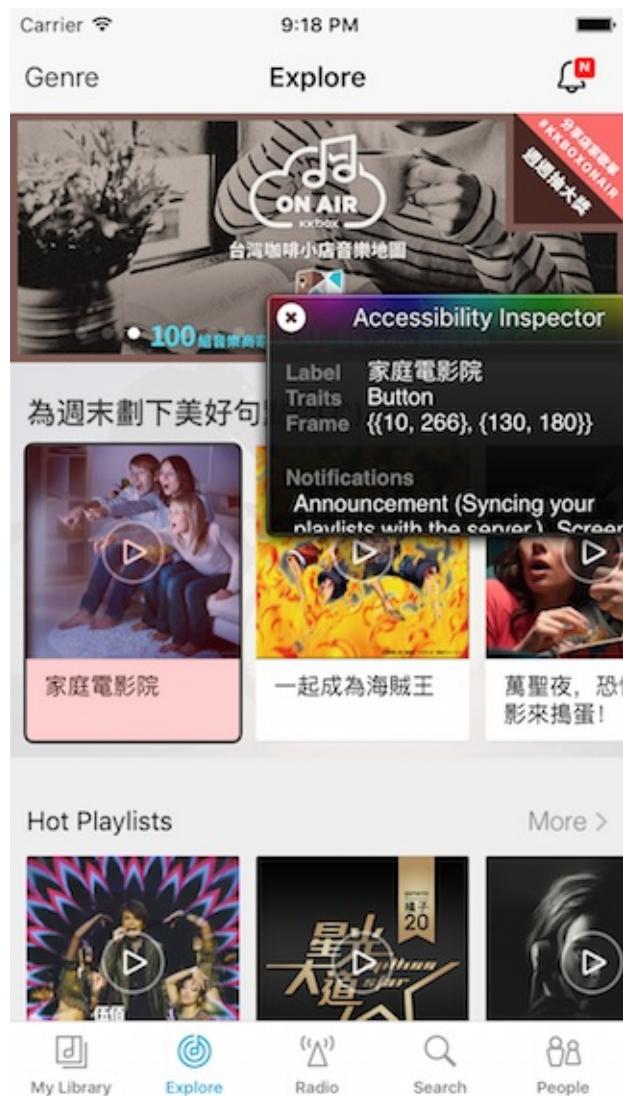
```
[closeImage setAccessibilityLabel:NSLocalizedString(@"Close", @"")];  
[closeImage setAccessibilityIdentifier:@"Close"];
```

開發過程中，除了實機測試之外，也可以使用 iOS Simulator 檢驗目前程式對 VoiceOver 的支援狀態，方法是在模擬器中的偏好設定中，把 Accessibility Inspector 打開，模擬器畫面中就會出現一個小畫面，在選取畫面中某個 UI 元件時，Accessibility Inspector 就會列出各種相關資訊。

Accessibility Inspector 在設定中的位置：

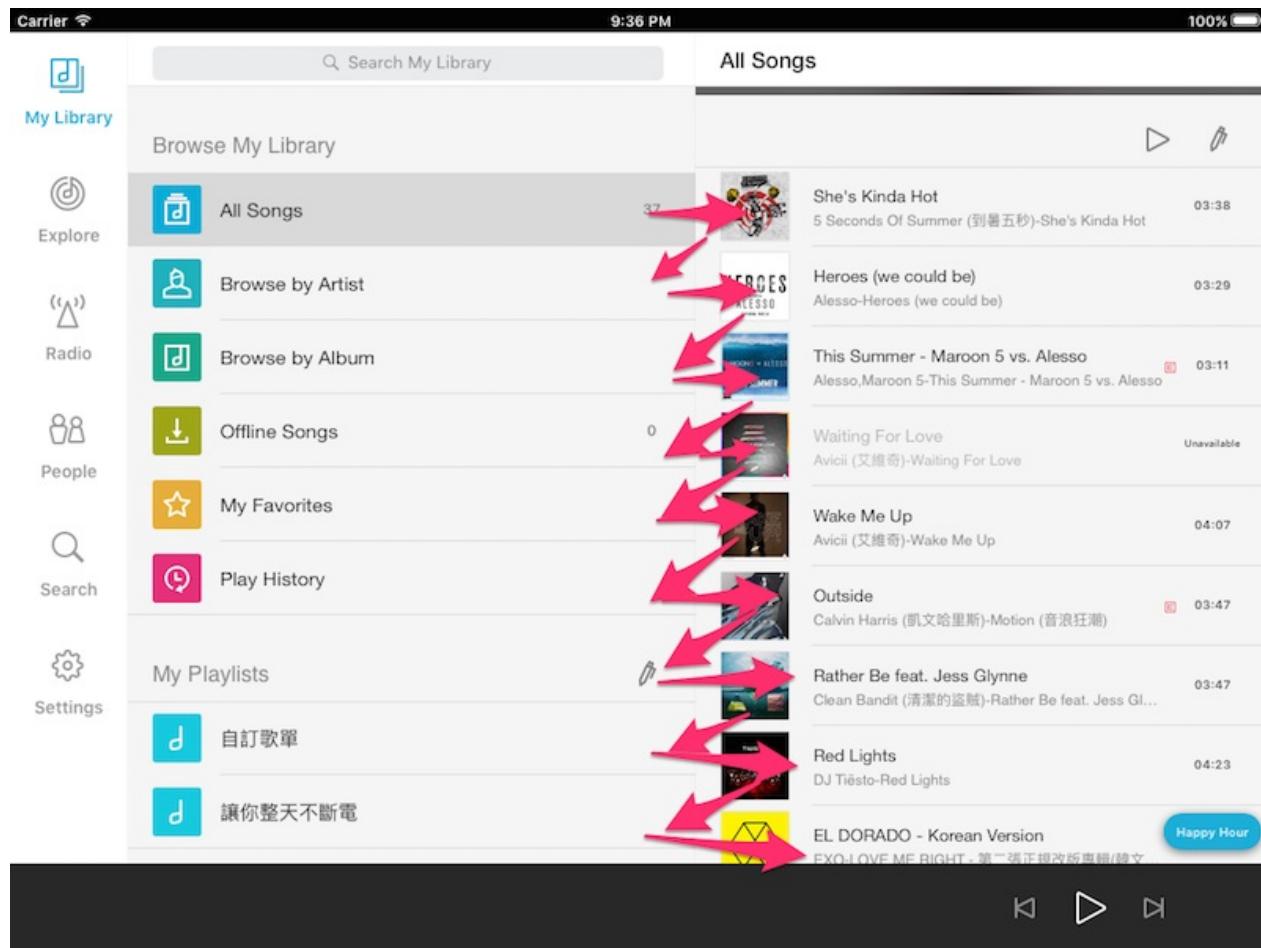


開啟之後的效果：

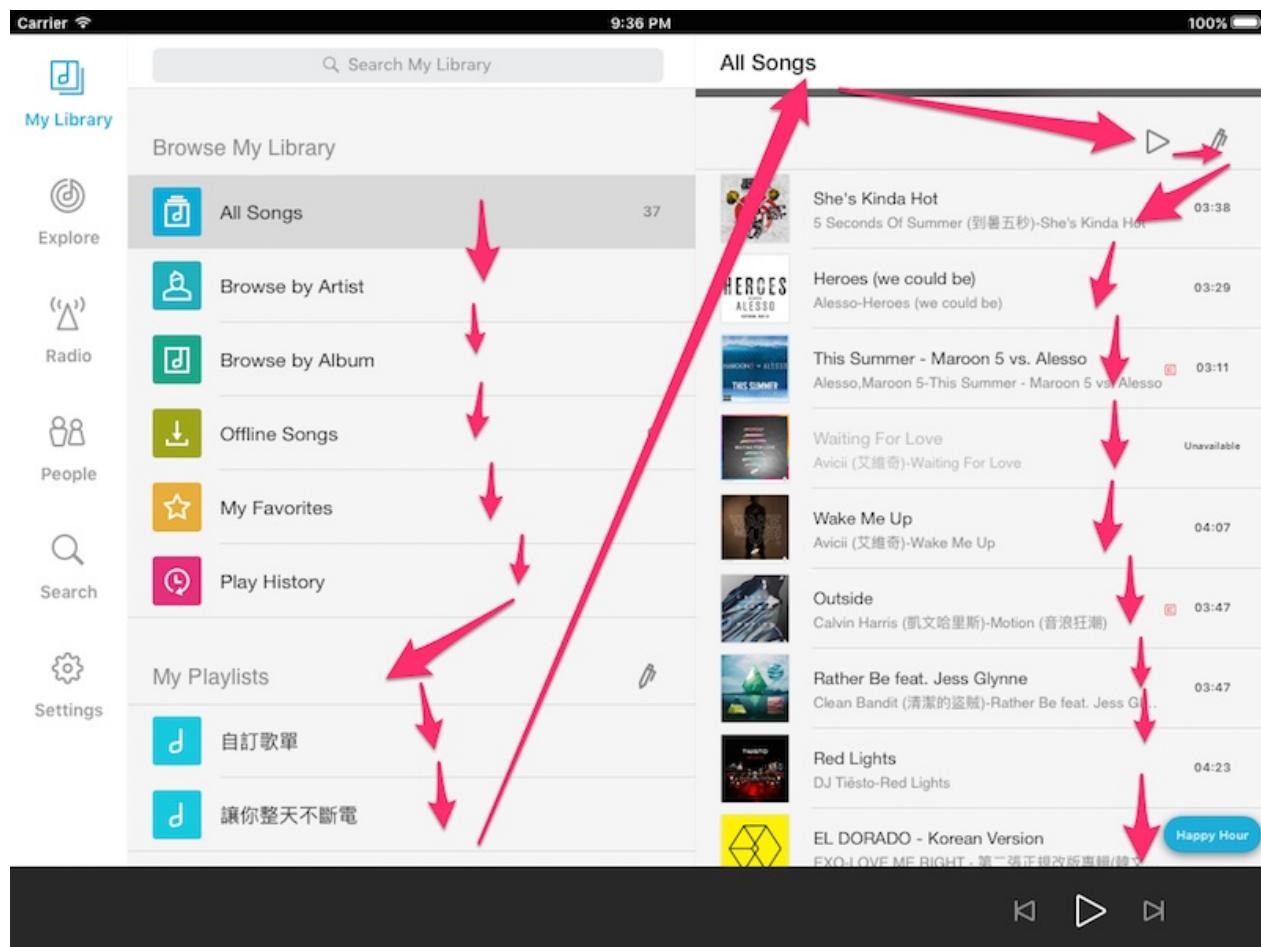


VoiceOver 的方向

VoiceOver 在朗讀螢幕上有哪些 UI 元件的時候，基本上按照由右至左，由上而下的方向，會先把橫排的資訊念完，然後繼續往下念。在有些時候這種順序會造成使用上的困擾，以 KKBOX 的 iPad 版本來說，我們有好幾個並排的 UITableViewController，如果是由右至左再由上而下，就會變成會交錯朗讀不同的 UITableViewController 裡頭的 cell。如下圖：



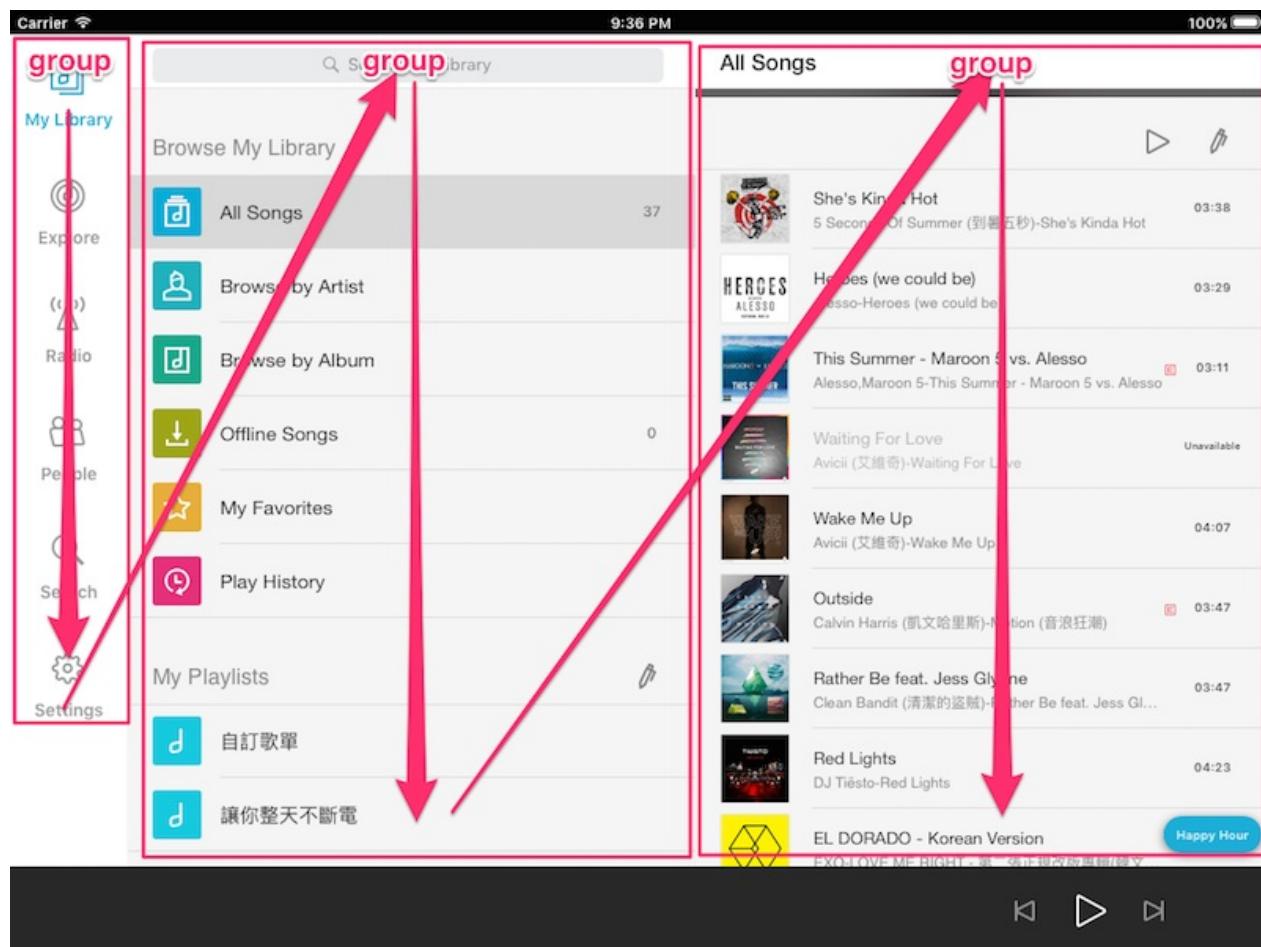
但我們期待的是，先把第一個 table 念完，再去念另外一個 table。



要修正這個問題，我們要做的事情其實很簡單：我們要去告訴某個 view，應該要把它的所有 subviews 都變成是一個群組，VoiceOver 要先念完這個群組裡頭所有的東西，再去移動到其他的 UI 元件上。只要寫這樣的程式碼：

```
[self.view setShouldGroupAccessibilityChildren:YES];
```

就可以讓 view 裡頭變成群組了。



進階的 Accessibility 設定

可以調整數值的元件

如果我們寫了一個客製化元件是個 slider，可以讓用戶調整裡頭的數值，那麼，在 VoiceOver 打開的時候，用戶就可以用向上或向下 swipe 的手勢操作。要實作這樣的行為，就要實作 `accessibilityIncrement` 與 `accessibilityDecrement`。

在以下的範例中，我們寫了一個叫做 KKProgressSlider（繼承自 UIControl） ，實作了 `accessibilityIncrement` 與 `accessibilityDecrement`，這兩個 method 都會改變 value，增加或減少 5，觸發之後，就會告訴 target 執行對應到 `UIControlEventValueChanged` 的 action。

```
- (BOOL)isAccessibilityElement
{
    return YES;
}

- (UIAccessibilityTraits)accessibilityTraits
{
    return UIAccessibilityTraitAdjustable | UIAccessibilityTraitUpdatesFrequently;
}

- (NSString *)accessibilityLabel
{
    return LFLSTR(@"Playing progress");
}

- (void)_updateValueByAccessibility
{
    if (value < self.minimumValue) {
        value = self.minimumValue;
    }
    if (value > self.maximumValue) {
        value = self.maximumValue;
    }
    for (id target in [[self allTargets] allObjects]) {
        NSArray *actions = [self actionsForTarget:target forControlEvents:UIControlEventValueChanged];
        for (NSString *action in actions) {
            [self sendAction:NSSelectorFromString(action) to:target forEvent:nil];
        }
    }
}

- (void)accessibilityIncrement
{
    value += 5.0;
    [self _updateValueByAccessibility];
}
```

```

- (void)accessibilityDecrement
{
    value -= 5.0;
    [self _updateValueByAccessibility];
}

```

UIAccessibilityContainer

很多時候，在使用者介面中的重要文字與圖片，並不是放在直接放在某個 UIView 裡頭，當你使用 Core Animation 製作 UI 的時候，就會把很多東西放在 CALayer 裡一某個 layer 裡頭的 content 屬性可能是一張重要的圖，或是用了 CATextLayer 呈現文字。

CALayer 物件本身沒辦法處理無障礙資源。想要讓系統知道某個 layer 的存在與其所代表的意義，就必須要用一個 container 物件處理，這個 container 通常是這個 layer 的 super layer 所在的 view，實作 UIAccessibilityContainer protocol。

需要注意，自己實作了 UIAccessibilityContainer protocol 之後，會完全改變這個 view 裡頭的無障礙支援的行為，所以，如果原本這個 view 的 subviews 中還有許多其他的 UIView 物件，如果我們自己實作的 UIAccessibilityContainer 並沒有處理到這些 subview，這些 subview 原本具有的無障礙支援也就會隨之失效。

我們可以針對每一個 CALayer 物件，產生一個對應的 UIAccessibilityElement 物件，然後透過 setAccessibilityFrame: 指定對應的 CALayer 在畫面上的範圍。

需要注意的是，這邊所指定的位置，必須是這個 layer 在整個應用程式畫面中的位置，而不是 layer 原本的 frame 屬性，layer 本身的 frame 屬性只是相對於上一層的 layer (super layer) 的位置而已。我們在這裡可以透過呼叫 convertRect:fromLayer:，轉換出目前 layer，與應用程式啟動時，我們所產生的那個 UIWindow 物件上的 layer，兩者之間的位置關係，這個 CGRect 才是我們想要的資訊。

當 layer 在畫面上的位置有改變時，最後記得要發一次 UIAccessibilityLayoutChangedNotification。

比方說，我們寫了一個 UIView 的 subclass，裡頭使用 Core Animation Layer 顯示許多圖片，叫做 KKPhotoGridView，這個 view 設計了一個 data source，每次呼叫 reload data 的時候，都會跟 data source 索取 KKPhotoItemLayer 物件，KKPhotoItemLayer 有個屬性叫做 accessibilityElement，屬於 UIAccessibilityElement。

```

@interface KKPhotoItemLayer : CALayer
@property (strong, nonatomic) UIAccessibilityElement *accessibilityElement;
@end

```

KKPhotoGridView 的 reload data 可以這麼寫：跟 data source 要了 layer 之後，除了把 layer 加入到 layerArray 之外，同時也建立了 UIAccessibilityElement 物件，變成是 layer 的 accessibilityElement 屬性。

```

NSUInteger itemCount = [photoGridDataSource numberOfItemsInPhotoGridView:self];
for (NSUInteger index = 0; index < itemCount; index++) {
    KKPhotoItemLayer *itemlayer = [photoGridDataSource
        photoGridView:self layerForItemAtIndex:index];
    UIAccessibilityElement *anElement = [[UIAccessibilityElement alloc]
        initWithAccessibilityContainer:self];
    anElement.accessibilityLabel = LFLSTR(@"Photo");
    anElement.accessibilityIdentifier = [NSString stringWithFormat:@"Photo %lu", (unsigned long)index];
}

```

```
d long)index];
    anElement.accessibilityTraits = UIAccessibilityTraitButton;
    itemlayer.accessibilityElement = anElement;
    [layerArray addObject:itemlayer];
}
UIAccessibilityPostNotification(UIAccessibilityLayoutChangedNotification, nil);
```

最後實作 UIAccessibilityContainer

```
- (BOOL)isAccessibilityElement
{
    return NO;
}

- (NSInteger)accessibilityElementCount
{
    return [layerArray count];
}

- (id)accessibilityElementAtIndex:(NSInteger)index
{
    KKPhotoItemLayer *aLayer = layerArray[index];
    CGRect frame = aLayer.frame;
    aLayer.accessibilityElement.accessibilityFrame = [[UIApplication sharedApplication] keyWindow convertRect:frame fromView:self];
    return aLayer.accessibilityElement;
}

- (NSInteger)indexOfAccessibilityElement:(id)element
{
    NSInteger index = 0;
    for (KKPhotoItemLayer *aLayer in layerArray) {
        if ([aLayer.accessibilityElement isEqual:element]) {
            return index;
        }
        index++;
    }
    return NSNotFound;
}
```

相關閱讀

- [Accessibility Programming Guide for iOS](#)
- [WWDC 2014 Accessibility on iOS](#)
- [WWDC 2014 Accessibility on OSX](#)
- [WWDC 2015 Apple Watch Accessibility](#)
- [WWDC 2015 iOS Accessibility](#)

練習：KKBOX 動態歌詞

練習範圍

- Accessibility

練習內容

- 請拿出我們之前使用 Core Animation 實作的的動態歌詞作業
- 在動態歌詞的 view 上，加入 accessibility 支援

Mac 上的 WebKit

蘋果後來推出了 **WKWebView**，其實現在也會比較建議使用 **WKWebView**，但是因為一些歷史因素，我們還是保留了原本的說明文件，請視需要閱讀。

KKBOX 在推出 iOS 版本之前，在 2008 年九月就推出了 Mac 版本。我們會在這一章當中，討論一個專屬 Mac OS X 平台的問題：WebKit。

iOS 上面一樣有 Web 瀏覽器元件可以使用，包括 UIWebView、WKWebview 以及 iOS 9 之後的 Safari View Controller。不過，因為蘋果政策的關係，iOS 上的 Web 瀏覽元件受限非常大，但相對的，在 Mac OS X 平台上，我們可說可以完整控制 WebKit framework 中的 WebView。

WebView 可說是 Mac OS X 平台上最複雜的 UI 元件。許多元件上都有 delegate 的設計，但 WebView 的 delegate 居然有五種之多，包括 downloadDelegate、frameLoadDelegate、policyDelegate、resourceLoadDelegate 以及 UIDelegate，從這邊便可以看出 WebView 的複雜；甚至，在蘋果網站中，其實沒有屬於 Web 裡頭的完整文件，像如果我們想要操作 WebView 裡頭的 DOM 物件，DOM 物件有哪些 Objective-C 介面，其實要去 WebKit 的網站查詢。

不同於 KKBOX 的 iOS、Android 與 WindowsPhone 等 mobile 版本，幾乎所有的內容都用 Native UI 呈現，KKBOX 的 desktop 版本大量使用來自 Web 的內容，打開 KKBOX Desktop 版本，迎面而來的線上內容，就是一個專屬 KKBOX 的網站，我們稱之為 portal site。在 portal site 裡頭的頁面，除了使用各種網頁技術之外，也經常需要與 client side app 的 native code 溝通。

在這一章中，我們就會討論如何使用 Objective-C 語言操作 WebView 裡頭的 DOM、網頁裡頭的 JavaScript 如何與 Objective-C 溝通，以及我們可以如何使用蘋果的 JavaScript 引擎：JavaScriptCore。

用 Objective-C 操作 DOM

來簡單講講 Objective-C 怎麼操作 DOM。假如我們現在有一個簡單的 HTML 檔案：

```
<div id="main">
</div>
```

那麼，在 JavaScript 裡頭，我們會這樣取得 main 這個 div 的 DOM 物件：

```
var main = document.getElementById('main');
```

我們便可以把這段程式翻譯成 ObjC：

```
DOMDocument *document = [[webView mainFrame] DOMDocument];
DOMHTMLElement *main = (DOMHTMLElement *)[document getElementById:@"main"];
```

我們也可以用一些比較新的 API，例如 querySelector。

```
var main = document.querySelector('#main');
main.style.backgroundColor = '#AAA';
```

```
DOMDocument *document = [[webView mainFrame] DOMDocument];
DOMHTMLElement *main = (DOMHTMLElement *)[document querySelector:@"#main"];
```

假如我們想要在 main 這個 div 中，產生、並加入一個新的 child node，會這樣寫：

```
var main = document.getElementById('main');
var child = document.createElement('div');
child.innerHTML = 'child:' + (main.childElementCount + 1);
main.appendChild(child);
```

話說 childElementCount 也是一個比較新的 API，WebKit 要 4.0 之後才支援，Firefox 則是在 3.5 之後才支援。把上面那段程式翻譯成 ObjC 的話：

```
DOMDocument *document = [[webView mainFrame] DOMDocument];
DOMHTMLElement *main = (DOMHTMLElement *)[document
    getElementById:@"main"];
DOMHTMLElement *newElement = (DOMHTMLElement *)[document
    createElement:@"div"];
newElement.innerHTML = [NSString stringWithFormat:@"child:%d",
    main.childElementCount + 1];
[main appendChild:newElement];
```

如果要修改某個 HTML element 的 CSS 樣式，在 JavaScript 是這樣：

```
var main = document.getElementById('main');
main.style.backgroundColor = '#AAA';
```

用 Objective-C 寫是這樣：

```
DOMDocument *document = [[webView mainFrame] DOMDocument];
DOMHTMLElement *main = (DOMHTMLElement *)[document
    getElementById:@"main"];
DOMCSSStyleDeclaration *style = main.style;
[stylesetProperty:@"background-color" value:@"#AAA" priority:@""];
// 也可以寫成 [style setBackgroundColor:@"#AAA"];
```

Javascript 與 Objective-C 的溝通

在寫 JavaScript 的時候，可以使用一個叫做 window 的物件，像是我們想要從現在的網頁跳到另外一個網頁的時候，就會去修改 `window.location.href` 的位置；在我們的 Objective-C 程式碼中，如果我們可以取得指定的 WebView 的指標，也就可以拿到這個出現在 JavaScript 中的 window 物件，也就是 `[webView windowScriptObject]`。

這個物件就是 WebView 裡頭的 JavaScript 與我們的 Objective-C 程式之間的橋樑—window 物件可以取得網頁裡頭所有的 JavaScript 函數與物件，而如果我們把一個 Objective-C 物件設定成 `windowScriptObject` 的 value，JavaScript 也便可以呼叫 Objective-C 物件的 method。於是，我們可以在 Objective-C 程式裡頭要求 WebView 執行一段 JavaScript，也可以反過來讓 JavaScript 呼叫一段用 Objective-C 實作的功能。

用 Objective-C 取得與設定 JavaScript 物件

要從 Objective-C 取得與設定得網頁中的 JavaScript 物件，也就是對 `windowScriptObject` 做一些 KVC 呼叫，像是 `valueForKey:` 與 `valueForKeyPath:`。如果我們在 JavaScript 裡頭，想要知道目前的網頁位置，會這麼寫：

```
var location = window.location.href;
```

用 ObjC 就可以這麼呼叫：

```
NSString *location = [[webView windowScriptObject]
    valueForKeyPath:@"location.href"];
```

如果我們要設定 `window.location.href`，要求開啟另外一個網頁，在 Javascript 裡頭：

```
window.location.href = 'http://zonble.net';
```

Objective-C：

```
[[webView windowScriptObject] setValue:@"https://kkbox.com" forKeyPath:@"location.href"];
```

由於 Objective-C 與 Javascript 本身的語言特性不同，在兩種語言之間相互傳遞東西之間，就可以看到兩者的差別—

- Javascript 雖然是物件導向語言，但是並沒有 class，所以將 JS 物件傳到 Objective-C 程式裡頭，除了基本字串會轉換成 `NSString`、基本數字會轉成 `NSNumber`，像是 Array 等其他物件，在 Objective-C 中，都是 `WebScriptObject` 這個 Class。也就是，Javascript 的 Array 不會幫你轉換成 `NSArray`。
- 從 Javascript 裡頭傳一個空物件給 Objective-C 程式，用的不是 Objective-C 裡頭原本表示「沒有東西」的方式，像是 `NULL`、`nil`、`NSNull` 等，而是專屬 WebKit 使用的 `WebUndefined`。

所以，如果我們想要看一個 Javascript Array 裡頭有什麼東西，就要先取得這個物件裡頭叫做 `length` 的 value，然後用 `webScriptValueAtIndex:` 去看在該 index 位置的內容。假如我們在 JS 裡頭這樣寫：

```
var JSArray = ['zonble', 'dot', 'net'];
```

```

for (var i = 0; i < JSArray.length; i++) {
    console.log(JSArray[i]);
}

```

Objective-C 裡頭就會變成這樣：

```

WebScriptObject *obj = (WebScriptObject *)JSArray;

NSUInteger count = [[obj valueForKey:@"length"] integerValue];
NSMutableArray *a = [NSMutableArray array];
for (NSUInteger i = 0; i < count; i++) {
    NSString *item = [obj webScriptValueAtIndex:i];
    NSLog(@"%@", item);
}

```

用 Objective-C 呼叫 JavaScript Function

要用 Objective-C 呼叫網頁中的 JavaScript function，大概有幾種方法。第一種是直接寫一段跟你在網頁中會撰寫的JavaScript 一模一樣的程式，叫 `windowScriptObject` 用 `evaluateWebScript:` 執行。例如，我們想要在網頁中產生一個新的 JavaScript function，內容是：

```

function x(x) {
    return x + 1;
}

```

所以在 Objective-C 中可以這樣寫；

```
[[webView windowScriptObject] evaluateWebScript:@"function x(x) {return x + 1;}"];
```

接下來我們就可以呼叫 `window.x()`：

```

NSNumber *result = [[webView windowScriptObject] evaluateWebScript:@"x(1)"];
NSLog(@"%@", result integerValue); // Returns 2

```

由於在 Javascript 中，每個 function 其實都是物件，所以我們還可以直接取得 `window.x` 叫這個物件執行自己。在 Javascript 裡頭如果這樣寫：

```
window.x.call(window.x, 1);
```

Objective-C 中便是這樣：

```

WebScriptObject *x = [[webView windowScriptObject] valueForKey:@"x"];
NSNumber *result = [x callWebScriptMethod:@"call"
                                    withArguments:[NSArray arrayWithObjects:x,
                                                 [NSNumber numberWithInt:1], nil]];

```

這種讓某個 `WebScriptObject` 自己執行自己的寫法，其實比較不會用於從 Objective-C 呼叫 Javascript 這一端，而是接下來會提到的，由 Javascript 呼叫 Objective-C，因為這樣 Javascript 就可以把一個 callback function 送到 Objective-C，因為這樣 程式裡頭。

如果我們在做網頁，我們只想要更新網頁中的一個區塊，就會利用 AJAX 的技巧，只對這個區塊需要的資料，對 server 發出 request，並且在 request 完成的時候，要求執行一段 callback function，更新這一個區塊的顯示內容。從 Javascript 呼叫 Objective-C 也可以做類似的事情，如果 Objective-C 程式裡頭需要一定時間的運算，或是我們可能是在 Obj C 裡頭抓取網路資料，我們便可以把一個 callback function 送到 Objective-C 程式裡，要求 Objective-C 程式在做完工作後，執行這段 callback function。

DOM

WebKit 裡頭，所有的 DOM 物件都繼承自 `DOMObject`，`DOMObject` 又繼承自 `WebScriptObject`，所以我們在取得了某個 DOM 物件之後，也可以從 Objective-C 程式中，要求這個 DOM 物件執行 Javascript 程式。

假如我們的網頁中，有一個 id 叫做 “#s” 的文字輸入框（text input），而我們希望現在鍵盤輸入的焦點放在這個輸入框上，在 JS 裡頭會這樣寫：

```
document.querySelector('#s').focus();
```

```
DOMDocument *document = [[webView mainFrame] DOMDocument];
[[document querySelector:@"#s"] callWebScriptMethod:@"focus"
withArguments:nil];
```

用 JavaScript 存取 Objective-C 的 Value

要讓網頁中的 JavaScript 程式可以呼叫 Objective-C 物件，方法是把某個 Objective-C 物件註冊成 JavaScript 中 `window` 物件的屬性。之後，JavaScript 便也可以呼叫這個物件的 method，也可以取得這個物件的各種 Value，只要是 KVC 可以取得的 Value，像是 `NSString`、`NSNumber`、

`NSDate`、`NSArray`、`NSDictionary`、`NSValue`...等。JavaScript 傳 Array 到 Objective-C 時，還需要特別做些處理才能變成 `NSArray`，從 Objective-C 傳一個 `NSArray` 到 JavaScript 時，會自動變成 JavaScript Array。

首先我們要注意的是將 Objective-C 物件註冊給 `window` 物件的時機，由於每次重新載入網頁，`window` 物件的內容都會有所變動—畢竟每個網頁都會有不同的 JavaScript 程式，所以，我們需要在適當的時機做這件事情。我們首先要指定 `WebView` 的 frame loading delegate（用 `setFrameLoadDelegate:`），並且實作

`webView:didClearWindowObject:forFrame:`，`WebView` 只要更新了 `windowScriptObject`，就會呼叫這一段程式。假如我們現在要讓網頁中的 `JavaScript` 可以使用目前的 controller 物件，會這樣寫：

```
- (void)webView:(WebView *)sender
didClearWindowObject:(WebScriptObject *)windowObject
forFrame:(WebFrame *)frame
{
    [windowObject setValue:self forKey:@"controller"];
}
```

如此一來，只要呼叫 `window.controller`，就可以呼叫我們的 Objective-C 物件。假如我們的 Objective-C Class 裡頭有這些成員變數：

```
@interface MyController : NSObject
{
    IBOutlet WebView *webView;
    IBOutlet NSWindow *window;

    NSString *stringValue;
    NSInteger numberWithInt;
    NSArray *arrayValue;
    NSDate *dateValue;
    NSDictionary *dictValue;
    NSRect frameValue;
}
@end
```

指定一下 Value：

```
stringValue = @"string";
numberValue = 24;
arrayValue = [NSArray arrayWithObjects:@"text", [NSNumber numberWithInt:30], nil];
dateValue = [NSDate date];
dictValue = [NSDictionary dictionaryWithObjectsAndKeys:@"value1", @"key1",
           @"value2", @"key2",
           @"value3", @"key3", nil];
frameValue = [window frame];
```

用 JavaScript 讀讀看：

```
var c = window.controller;
var main = document.getElementById('main');
var HTML = '';
if (c) {
    HTML += '<p>' + c.stringValue + '<p>';
    HTML += '<p>' + c.numberValue + '<p>';
    HTML += '<p>' + c.arrayValue + '<p>';
    HTML += '<p>' + c.dateValue + '<p>';
    HTML += '<p>' + c.dictValue + '<p>';
    HTML += '<p>' + c.frameValue + '<p>';
    main.innerHTML = HTML;
}
```

結果如下：

```
string 24 text,30 2010-09-09 00:01:04 +0800 { key1 = value1; key2 = value2; key3 = value3; } NSRect:
{{275, 72}, {570, 657}}
```

不過，如果你看完上面的範例，就直接照做，應該不會直接成功出現正確的結果，而是會拿到一堆 `undefined`，原因是，Objective-C 物件的 Value 預設被保護起來，不會讓 Javascript 直接存取。要讓 Javascript 可以存取 Objective-C 物件的 Value，需要實作 `+isKeyExcludedFromWebScript:` 針對傳入的 Key 一一處理，如果我們希望 Javascript 可以存取這個 key，就回傳 `NO`：

```
+ (BOOL)isKeyExcludedFromWebScript:(const char *)name
{
    if (!strcmp(name, "stringValue")) {
        return NO;
    }
    return YES;
}
```

除了可以讀取 Objective-C 物件的 Value 外，也可以設定 Value，相當於在 Objective-C 中使用 `setValue:forKey:`，如果在上面的 Javascript 程式中，我們想要修改 `stringValue`，直接呼叫 `c.stringValue = 'new value'` 即可。像前面提到，在這裡傳給 Objective-C 的 Javascript 物件，除了字串與數字外，class 都是 `WebScriptObject`，空物件是 `WebUndefined`。

用 JavaScript 呼叫 Objective-C method

Objective-C 的語法沿襲自 Small Talk，Objective-C 的 selector，與 JavaScript 的 function 語法有相當的差異。WebKit 預設的實作是，如果我們要在 JavaScript 呼叫 Objective-C selector，就是把所有的參數往後面擺，並且把所有的冒號改成底線，而原來 selector 如果有底線的話，又要另外處理。假使我們的 controller 物件有個 method，在 Objective-C 中寫成這樣：

```
- (void)setA:(id)a b:(id)b c:(id)c;
```

在 JS 中就這麼呼叫：

```
controller.setA_b_c_('a', 'b', 'c');
```

實在有點醜。所以 WebKit 提供一個方法，可以讓我們把某個 Objective-C selector 變成好看一點的 Javascript function。我們要實作 `webScriptNameForSelector`：

```
+ (NSString *)webScriptNameForSelector:(SEL)selector
{
    if (selector == @selector(setA:b:c:)) {
        return @"setABC";
    }
    return nil;
}
```

以後就可以這麼呼叫：

```
controller.setABC('a', 'b', 'c');
```

我們同樣可以決定哪些 selector 可以給 Javascript 使用，哪些要保護起來，方法是實作 `isSelectorExcludedFromWebScript:`。而我們可以改變某個 Objective-C selector 在 Javascript 中的名稱，我們也可以改變某個 value 的 key，方法是實作 `webScriptNameForKey:`。

有幾件事情需要注意一下：

用 JavaScript 呼叫 Objective-C 的 property

在上面，我們用 JavaScript 呼叫 `window.controller.stringValue`，與設定裡頭的 value 時，這邊很像我們使用Objective-C 2.0 的語法，但其實做的是不一樣的事情。用 JavaScript 呼叫 `controller.stringValue`，對應到的 Objective 語法是 `[controller valueForKey:@"stringValue"]`，而不是呼叫 Objective-C 物件的property。

如果我們的 Objective-C 物件有個 property 叫做 `stringValue`，我們知道，Objective-C property 其實會在編譯時，變成getter/setter method，在 JavaScript 裡頭，我們便應該要呼叫 `controller.stringValue()` 與 `controller.setStringValue_()`。

Javascript 中，Function 即物件的特性

Javascript 的 function 是物件，當一個 Objective-C 物件的 method 出現在 Javascript 中時，這個 method 在 Javascript 中，也可以或多或少當做物件處理。我們在上面產生了 `setABC`，也可以試試看把它倒出來瞧瞧：

```
console.log(controller.setABC);
```

我們可以從結果看到：

```
function setABC() { [native code] }
```

這個 function 是 native code。因為是 native code，所以我們無法對這個 function 呼叫 `call` 或是 `apply`。

另外，在把我們的 Objective-C 物件註冊成 `window.controller` 後，我們會許也會想要讓 controller 變成一個 function 來執行，像是呼叫 `window.controller()`；或是，我們就只想要產生一個可以讓 JS 呼叫的 function，而不是整個物件都放進 Javascript 裡頭。我們只要在 Objective-C 物件中，實作 `invokeDefaultMethodWithArguments:`，就可以回傳在呼叫 `window.controller()` 時想要的結果。

現在我們可以綜合練習一下。前面提到，由於我們可以把 Javascript 物件以 `WebScriptObject` 這個 class 傳入 Objective-C 程式，Objective-C 程式中也可以要求執行 `WebScriptObject` 的各項 function。我們假如想把 A 與 B 兩個數字丟進 Objective-C 程式裡頭做個加法，加完之後出現在網頁上，於是我們寫了一個 Objective-C method：

```
- (void)numberWithA:(id)a plusB:(id)b callback:(id)callback
{
    NSInteger result = [a integerValue] + [b integerValue];

    [callback callWebScriptMethod:@"call" withArguments:
        [NSArray arrayWithObjects:callback, [NSNumber
            numberWithInt:result], nil]];
}
```

Javascript 裡頭就可以這樣呼叫：

```
window.controller.numberWithA_plusB_callback_(1, 2, function(result) {
    var main = document.getElementById('main');
    main.innerText = result;
});
```

其他平台上 WebKit 的實作

除了 Mac OS X，WebKit 這幾年也慢慢移植到其他的作業系統與 framework 中，也或多或少都有 Native API 要求 WebView 執行 Js，以及從 JS 呼叫 Native API 的機制。

跟 Mac OS X 比較起來，iOS 上 `UIWebView` 的公開 API 實在少上許多。想要讓 `UIWebView` 執行一段 JavaScript，可以透過呼叫 `stringByEvaluatingJavaScriptFromString:`，只會回傳字串結果，所以能夠做到的事情也就變得有限，通常大概就拿來取得像 `window.title` 這些資訊。在 iOS 上我們沒辦法將某個 Objective-C 物件變成 JS 物件，所以，在網頁中觸發了某些事件，想要通知 Objective-C 這一端，往往會選擇使用像「`zombie://`」這類 Customized URL scheme。

Android 的 `WebView` 物件提供一個叫做 `addJavascriptInterface()` 的 method，可以將某個 Java 物件註冊成 JavaScript 的 `window` 物件的某個屬性，就可以讓 JavaScript 呼叫 Java 物件。不過，在呼叫 Java 物件時，只能夠傳遞簡單的文字、數字，複雜的 JavaScript 物件就沒辦法了。而在 Android 上想要 `WebView` 執行一段 JavaScript，在文件中沒看到相關資料，網路上面找到的說法是，可以透過 `loadUrl()`，把某段 JavaScript，用 bookmarklet 的形式傳進去。

在 `QtWebKit` 裡頭，可以對 `QWebFrame` 呼叫 `addToJavaScriptWindowObject`，把某個 `QObject` 暴露在 JavaScript 環境中，我不清楚 JavaScript 可以傳遞哪些東西到 `QObject` 裡頭就是了。在 `QtWebKit` 中也可以取得網頁裡頭的 DOM 物件（`QWebElement`、`QWebElementCollection`），我們可以對 `QWebFrame` 還有這些 DOM 物件呼叫 `evaluateJavaScript`，執行 Javascript。

GTK 方面，因為是 C API，所以在應用程式與 Javascript 之間，就不是透過操作包裝好的物件，而是呼叫 WebKit 裡頭 JavaScript Engine 的 C API。

JavaScriptCore Framework

我們在 Mac OS X 上面，也可以透過 C API，要求 WebView 執行 Javascript。我們可以使用 JavaScriptCore Framework。

先簡單示範一下。如果我們想要簡單改一下 `window.location.href` :

```
JSGlobalContextRef globalContext = [[webView mainFrame] globalContext];
JSValueRef exception = NULL;
JSStringRef script = JSStringCreateWithUTF8CString(
    "window.location.href='http://zonble.net'");
JSEvaluateScript(globalContext, script, NULL, NULL, 0, &exception);
JSStringRelease(script);
```

JavaScriptCore 簡介

JavaScriptCore 是 WebKit 的 JavaScript 引擎，一般來說，在 Mac OS X 上，我們想要製作各種網頁與 Native API 程式互動的功能，大概不會選擇使用 JavaScriptCore，因為現在寫 Mac OS X 的桌面應用程式，多半會直接選擇使用 Objective-C 語言與 Cocoa API，各種需要的功能，都有像在前面提到的 Objective-C 方案—使用 WebKit Framework 中的 `WebScriptObject` 與各種 DOM 物件。

在 Objective-C 程式中當然可以使用 JavaScriptCore 所提供的 C API，但是在實際撰寫程式時，遠比使用 Objective-C 呼叫麻煩—你的主要 controller 物件還是用 Objective-C 撰寫的，如果直接用 JavaScriptCore 產生 JavaScriptCore 可以呼叫的 callback，這些 callback C function 還是會需要向 Objective-C 物件要資料，等於是多繞了一圈。

然而如果真的有需要的話，我們的確可以混用 JavaScriptCore 與 WebKit 的 Objective-C API，每個 `WebScriptObject` 中，都可以用 `JSObject` 這個 method 取得對應的 `JSObjectRef`。就跟許多 Mac OS X 或 iOS 上的 framework 一樣，Objective-C 物件中往往包含 C 的 API，我們於是可以在 `UIColor` 得到 `CGColor`，從 `UIImage` 得到 `CGImage`，在 `WebScriptObject` 中，就是 `JSObjectRef`。

不過，在 WebKitGTK+ 上，由於 GTK 本身是 C API，所以透過 JavaScriptCore，看來就變成 GTK 應用程式與網頁內的 JS 程式相互呼叫、傳遞資料最重要的方法（不太熟悉 GTK，所以不太確定還有哪些其他的方法）。

在 GTK 網站上有一個簡單的 [範例程式](#)，除了示範怎樣產生一個新 Window，裡頭放一個 WebView 開始執行外，還包括怎樣產生一個網頁中 JavaScript 可以呼叫的 C function，並且將這個 C function 註冊到成 JavaScript 的 `window` 物件的某個成員 function。看這個程式還頂有趣的一絕大部分程式的命名規則與風格，都是屬於 GTK 的風格，但是在呼叫到 JavaScriptCore 的時候，卻又是蘋果的 CoreFoundation 的風格。

JavaScriptCore 雖然是 WebKit 的 JS 引擎，但是其實並不一定需要 WebKit。JavaScriptCore 中所有的 API 都是對著一個 context 操作，我們要操控 WebView，就是對著 WebView 中某個 frame 的 global context 操作（`[webFrame globalContext]`），而這個 context 可以不需要有一個 WebFrame 物件就可以自己產生，我們可以在完全沒有圖形介面、不用到 Web 排版引擎的狀況下，產生、執行 JavaScript 程式碼。

所以可以看到，不少專案將這個引擎從 WebKit 中拆分出來，另外橋接其他的 library，在 GTK 上就有 SEED，讓你可以用 JavaScript 呼叫 `GObject`，用 JavaScript 寫 GTK 應用程式。在 Mac OS X 上則有橋接 JavaScript 與各種 Cocoa 物件的專案，像是 JSCocoa，以及以 JSCocoa 為基礎，作為代替 AppleScript 的

JSTalk，後來則出現了 CocoaScript，像如果你要寫 Sketch 的 plug-in，就會用到 CocoaScript 語言。

不過，在這便只大概講一下用 WebView 開啟的網頁，怎麼透過 JavaScriptCore 呼叫 C Function。

JavaScriptCore 裡頭的物件

我對 JavaScriptCore 這個東西不算熟，就像前面說的，真的在寫 Cocoa 程式往往直接呼叫

`WebScriptObject`。JavaScriptCore 裡頭有幾種基本的資料：

- `JSGlobalContextRef`：執行 JavaScript 的 context
- `JSValueRef`：在 JavaScript 中所使用的各種資料，包括字串、數字以及 function，都會包裝成 `Value`，我們可以從數字、`JSSStringRef` 或 `JSObject` 產生 `JSValueRef`，也可以轉換回來。需要特別注意的是，JS 裡頭的 null 也是一個 `JSValueRef` (`JSValueMakeUndefined` 與 `JSValueMakeNull`)。
- `JSSStringRef`：JavaScriptCore 使用的字串。用完記得要 `release`。
- `JSObjectRef`：JavaScript Array、Function 等。

來寫點程式：

```
- (void)webView:(WebView *)sender
didClearWindowObject:(WebScriptObject *)windowObject
forFrame:(WebFrame *)frame
{
    JSGlobalContextRef globalContext = [frame globalContext];
    JSSStringRef name = JSStringCreateWithUTF8CString("myFunc");
    JSObjectRef obj =
        JSObjectMakeFunctionWithCallback(globalContext, name,
            (JSObjectCallAsFunctionCallback)myFunc);
    JSObjectSetProperty (globalContext, [windowObject JSObject],
        name, obj, 0, NULL);
    JSStringRelease(name);
}
```

因為每次重新載入網頁，JavaScript 裡頭的 `window` 這個物件的內容就會更新一次，所以我們要等待 `WebView` 告訴我們應該要更新 `windowObject` 的時候，我們才做我們要做的事情—在 `window` 中加入一個可以讓 JavaScript 呼叫的 C function。我們首先要做兩件事情，第一是取得 `WebFrame` 裡頭的 `globalContext` (`[frame globalContext]`)，還有 `windowObject` 這個Objective-C 物件裡頭的 `JSObject` (`[windowObject JSObject]`)。

接著，我們要用 C 產生一個 JavaScript function 物件，在這邊用的是

`JSObjectMakeFunctionWithCallback`，代表我們想要產生一個可以用來呼叫 C Function 的 JavaScript function，我們要提供這個 JavaScript function 的名稱，還有對應到哪個 C function。如果我們想產生的 JavaScript function 不需要呼叫 C function，可以改用 `JSObjectMakeFunction`；最後，我們把這個 function 物件，註冊給 `windowObject` 的 `JSObject` 上，於是我們現在便可以在 JS 中呼叫 `window.myFunc()` 了。

來個綜合練習—我們現在在 JS 中傳入兩個數字，透過 C function 加完之後，執行一段 JS callback。我們的 JS 程式這麼寫—

```
window.myFunc(1, 1, function(result) {
    var main = document.getElementById('main');
    main.innerText = result;
```

```
});
```

在 myFunc 中，我們來練習一下 JavaScriptCore 裡頭的一些東西：

```
JSValueRef myFunc(JSContextRef ctx, JSObjectRef function,
    JSObjectRef thisObject, size_t argumentCount,
    const JSValueRef arguments[], JSValueRef* exception)
{
    if (argumentCount < 3) {
        JSStringRef string = JSStringCreateWithUTF8CString("UTF8String");
        JSValueRef result = JSValueMakeString(ctx, string);
        JSStringRelease(string);
        return result;
    }
    if (!JSValueIsNumber(ctx, arguments[0])) {
        JSStringRef string =
            JSStringCreateWithCFString((CFStringRef)@"NSString");
        JSValueRef result = JSValueMakeString(ctx, string);
        JSStringRelease(string);
        return result;
    }
    if (!JSValueIsNumber(ctx, arguments[1])) {
        return JSValueMakeNumber(ctx, 42.0);
    }
    if (!JSValueIsObject(ctx, arguments[2])) {
        return JSValueMakeNull(ctx);
    }

    double leftOperand = JSValueToNumber(ctx, arguments[0], exception);
    double rightOperand = JSValueToNumber(ctx, arguments[1], exception);
    JSObjectRef callback = JSValueToObject(ctx, arguments[2], exception);
    JSValueRef result = JSValueMakeNumber(ctx, leftOperand + rightOperand);
    JSValueRef myArguments[1] = {result};
    JSObjectCallAsFunction(ctx, callback, thisObject, 1,
        myArguments, exception); return JSValueMakeNull(ctx);
}
```

我們希望至少要有三個參數傳進來，前兩個參數是數字，最後一個參數是 `JSObject`，如果不是的話，就簡單回傳一點東西—在這邊可以看到，`JSStringRef` 除了可以用 UTF8 字串產生，也可以從 `CFString` 產生。

我們接下來把 `JSValue` 轉成 `double`，簡單做個加法，最後用 `JSObjectCallAsFunction`，執行 JavaScript `callback`—其實這邊還應該要用 `JSObjectIsFunction`，來檢查一下這個 `JSObject` 到底是不是 `function` 才是。

小結

這本教材在這邊先告一段落，在完成這一階段訓練之後，接下來就是會實際參與 KKBOX 的各項專案開發了。

由於這份教材裡頭講的是在 KKBOX 裡頭，我們認為開發 iOS App 最重要的東西，其實還有很多東西沒有講。像是 KVO/KVC，auto-layout 等畫面調整的問題，還有如何實作 In-app Purchase、Push Notification、如何使用 location、相機、挑選照片、藍芽裝置…各式各樣的議題，其實很多在 KKBOX 裡頭都有用到，但是我們都還沒有整理到教材中，更何況，還有許許多多我們還沒接觸到的技術。

我們可能會在以後繼續擴充這份教材，但始終不可能涵蓋所有的技術，加上每年都不斷有新技術，最後能夠倚靠的就只有自我的不斷學習。我們在內部有一個笑話：其實想要學會怎麼寫好程式，真的不用花很多時間，只要一輩子就夠了——輩子，其實是很短的，我們不會用兩輩子來學。

我們平時就有不少開發工作，但對新技術的掌握，也不能夠等到某個專案需要的時候才開始學，我們永遠無法知道在接下來的產品開發中，可能會突然用到什麼技術，能夠做的，只能夠無時不刻都保持準備。

有了這段時間的訓練，相信應該具備了閱讀蘋果官方技術文件的能力，以及習慣了直接觀看 WWDC 的技術影片。而除了閱讀文件、書籍，還有網路上的技術blog 之外，要不斷學習的關鍵之一，就是多與外界接觸、分享、交流。例如在台北市每個月第二個週四固定舉辦的 Cocoaheads Taipei，就是我們會定時參與的 iOS 與 Mac OS X 工程師聚會；而我們相信，交流是雙向的，我們應該要先把自己的好東西與別人分享，別人才會跟我們分享好東西。

身為在蘋果平台上的工程師，有機會的話，也一定要親自去 WWDC 看看，感受一下有五千人在同一個場地的工程師盛會，學會如何直接面對蘋果的工程師解決問題，並且直接浸淫在這個社群的開發文化中。

其他閱讀資料

有一些材料其實相當值得一讀，但是一時之間不知道應該納入哪個章節，先表列如下：

- 蘋果官網的專有名詞解釋
- [API Design](#), Matt Gemmell
- [Friday Q&A 2014-01-10: Let's Break Cocoa](#), Mike Ash
- [Friday Q&A 2013-09-27: ARM64 and You](#), Mike Ash
- [Friday Q&A 2013-05-03: Proper Use of Asserts](#), Mike Ash

版本記錄

2019 年

- 合併 Auto Layout 相關章節
- 增加 AVAudioEngine 相關章節
- 增加 CarPlay 相關章節
- 增加 MFI 助聽器相關章節

2015 年

- 首次釋出