用 Xcode + Storyboard + Swift

打造你的第一個 iOS App

待辦事項清單（To-Do List）完整開發教學

從專案建立、介面設計、Core Data 資料儲存

到逐行程式解說與自動化測試

開發工具　Xcode 26
介面技術　Storyboard（UIKit）
程式語言　Swift 5
資料儲存　Core Data
適用對象　iOS 開發初學者

2026 年 7 月

目錄

第 1 章

課程簡介與成品預覽

1.1 我們要做什麼？

在這份教材中，我們會從零開始，用 **Xcode**、**Storyboard** 與 **Swift** 開發一個功能完整的「待辦事項清單（To-Do List）」App。這是 iOS 開發最經典的入門題目，因為它涵蓋了幾乎所有 App 開發的核心觀念：

- 介面設計：使用 Storyboard 拖拉元件、設定 Auto Layout 約束。
- 表格列表：使用 `UITableView` 顯示動態資料。
- 使用者互動：點擊按鈕、彈出對話框、滑動刪除 / 編輯。
- 資料永久儲存：使用 Core Data 把資料存在手機裡，關掉 App 也不會消失。
- **MVC** 架構：理解 Model（資料）、View（畫面）、Controller（邏輯）的分工。

1.2 完成後的 App 功能

- 新增：點右上角「+」，輸入文字即可新增一筆待辦事項。
- 完成：點擊某一筆事項，切換「已完成 / 未完成」；已完成的事項會加上刪除線、變灰、並顯示藍色勾勾。
- 編輯：向右滑動某一筆事項，出現藍色「編輯」按鈕，可修改文字。
- 刪除：向左滑動某一筆事項，出現紅色「刪除」按鈕。
- 永久儲存：所有資料透過 Core Data 存放在裝置上，重開 App 資料仍在。
- 空狀態提示：清單為空時，畫面中央顯示友善的提示文字。

1.3 成品畫面

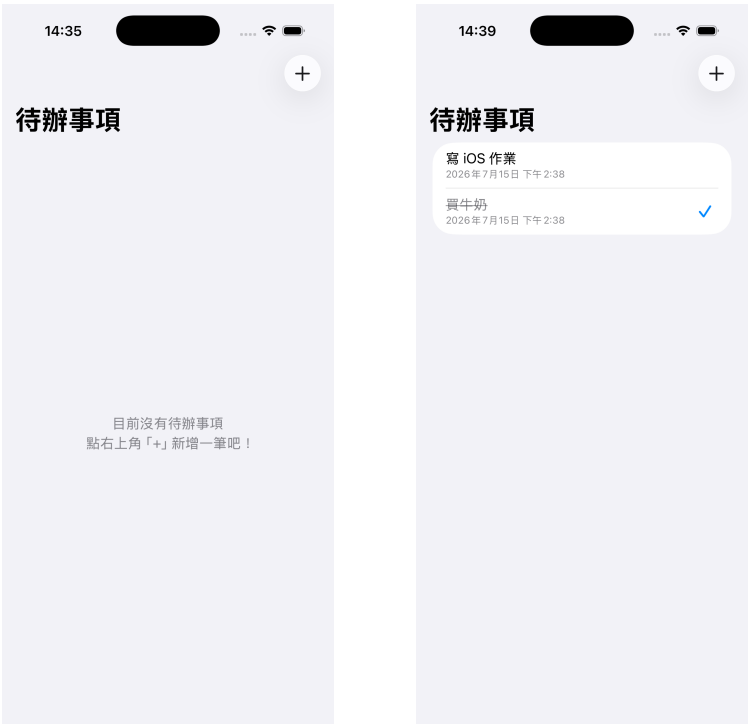

圖 1.1: 左：清單為空時的提示畫面。右：已加入兩筆事項，其中「買牛奶」已標記完成（刪除線 + 灰字 + 勾勾）。

1.4 專案檔案總覽

完成後的專案結構如下，本教材會逐一講解每個檔案的角色：

```
TodoList-Storyboard/
├── TodoList-Storyboard.xcodeproj          ← Xcode 專案檔
└── TodoList-Storyboard/
    ├── AppDelegate.swift                  ← App 進入點與 Core Data 堆疊
    ├── SceneDelegate.swift                ← 視窗（Scene）生命週期
    ├── ViewController.swift               ← 主畫面邏輯（本教材重點）
    ├── Base.lproj/
    │   ├── Main.storyboard                ← 主畫面介面設計
    │   └── LaunchScreen.storyboard        ← 啟動畫面
    ├── TodoList_Storyboard.xcdatamodeld   ← Core Data 資料模型
    ├── Assets.xcassets                    ← 圖片與顏色資源
```

```
    └── Info.plist                          ← App 設定檔
```

第 2 章

開發環境與專案建立

2.1 需要準備什麼？

- 一台 Mac 電腦（macOS）。
- **Xcode**：Apple 官方的整合開發環境（IDE），可從 Mac App Store 免費下載。本教材使用 Xcode 26。
- 不需要實體 iPhone——Xcode 內建「模擬器（Simulator）」即可執行與測試 App。

2.2 建立新專案（Step by Step）

1. 打開 Xcode，選擇 **File** → **New** → **Project...**（或在歡迎視窗點「Create New Project...」）。
2. 在範本選擇畫面，平台選 **iOS**，範本選 **App**，按 **Next**。
3. 填寫專案資訊：
    - **Product Name**：`TodoList-Storyboard`
    - **Interface**：**Storyboard**（重點！不要選 SwiftUI）
    - **Language**：**Swift**
    - **Storage**：勾選 **Core Data**（Xcode 會自動幫我們產生 Core Data 的樣板程式碼）
    - **Include Tests**：勾選（我們最後會寫 UI 測試）
4. 按 **Next**，選擇專案儲存位置，按 **Create**。

小提示

「Interface: Storyboard」代表用「視覺化拖拉」的方式設計畫面，適合初學者建立對 UIKit 的直觀理解；「SwiftUI」則是 Apple 較新的宣告式介面框架。兩者觀念相通，先學會其中一種，另一種會更容易上手。

2.3 認識 Xcode 介面

打開專案後，Xcode 視窗分為幾大區域：

- 左側導覽區（**Navigator**）：顯示專案所有檔案，點檔案即可開啟。
- 中央編輯區（**Editor)**：編輯程式碼，或以視覺化方式編輯 Storyboard。
- 右側檢查器（**Inspector**）：選取 Storyboard 元件後，在這裡調整屬性（文字、顏色、字型等）。
- 上方工具列：選擇模擬器機型、按 **Run**（播放鍵）建置並執行 App（快捷鍵 `Cmd + R`）。
- 下方除錯區（**Debug Area**）：顯示 `print()` 輸出與錯誤訊息。

2.4 Xcode 幫我們產生了哪些檔案？

建立專案後，Xcode 已自動產生下列檔案，每個檔案的角色如下表：

檔案	角色
`AppDelegate.swift`	App 的「總管」：App 啟動、進入背景等系統事件都在這裡處理；勾選 Core Data 後，這裡也放 Core Data 的核心程式碼。
`SceneDelegate.swift`	管理「一個視窗（Scene）」的生命週期（前景 / 背景切換等）。
`ViewController.swift`	主畫面的控制器，我們大部分的程式都寫在這裡。
`Main.storyboard`	主畫面的視覺化設計檔。
`LaunchScreen.storyboard`	App 啟動瞬間顯示的過場畫面。
`TodoList_Storyboard .xcdatamodeld`	Core Data 的「資料模型」設計檔。
`Assets.xcassets`	放 App 圖示、圖片、顏色等資源。
`Info.plist`	App 的設定檔（例如支援的畫面方向）。

表 2.1: 專案範本檔案一覽

第 3 章

核心觀念：MVC 與 App 的運作流程

3.1 MVC 架構

UIKit App 採用 **MVC**（**Model-View-Controller**）架構，把程式分成三種角色：

觀念：MVC 三種角色

- **Model**（模型）：資料本身。在本專案中就是 Core Data 的 `TodoItem`（一筆待辦事項）。
- **View**（視圖）：使用者看到的畫面。在本專案中就是 Storyboard 裡的 Table View、儲存格與按鈕。
- **Controller**（控制器）：居中協調者。負責把 Model 的資料顯示到 View 上，並把使用者在 View 上的操作（點擊、滑動）轉成對 Model 的修改。本專案的 `ViewController` 就是這個角色。

3.2 App 從啟動到顯示畫面的流程

使用者點擊 App 圖示後，系統依序執行：

1. 系統載入 App，呼叫 `AppDelegate` 的 `didFinishLaunchingWithOptions`（App 級初始化）。
2. 系統建立 Scene（視窗），呼叫 `SceneDelegate` 的 `willConnectTo`。
3. 因為 `Info.plist` 中指定了主 Storyboard 為 `Main`，系統自動載入 `Main.storyboard`，並實例化其「初始視圖控制器（Initial View Controller）」。
4. 本專案的初始控制器是一個 **Navigation Controller**，它再載入其根控制器——我們的 `ViewController`。
5. `ViewController` 的 `viewDidLoad()` 被呼叫，我們在這裡做畫面初始化與讀取資料。

3.3 UITableView 的運作原理

待辦清單的核心元件是 `UITableView`（表格視圖）。它的運作方式和一般直覺不同——**Table View** 不會自己知道要顯示什麼，而是透過「委任（Delegation）」模式反過來問你：

觀念：Delegation（委任）模式

Table View 定義了兩份「協定（Protocol）」：

- `UITableViewDataSource`（資料來源）：Table View 問你「有幾列？」「第 3 列長什麼樣子？」你必須實作對應方法來回答。
- `UITableViewDelegate`（委任）：Table View 通知你「使用者點了第 2 列」「使用者在第 5 列向左滑動」，你實作對應方法來回應。

只要把 `ViewController` 設成 Table View 的 dataSource 和 delegate，這些問題就會傳給它。

另外，Table View 使用「儲存格重用（**Cell Reuse**）」機制：畫面上只需要十幾個儲存格，滑出畫面的儲存格會被回收、換上新資料後重新使用，因此就算清單有一萬筆資料也不會耗盡記憶體。

第 4 章

用 Storyboard 設計介面

本章在 `Main.storyboard` 中完成主畫面的設計。請在左側導覽區點開 `Main.storyboard`，進入視覺化編輯模式。

4.1 步驟一：嵌入 Navigation Controller

我們希望畫面上方有一條導覽列（顯示標題「待辦事項」和「+」按鈕），因此要把現有的 View Controller 包進一個 Navigation Controller：

1. 在 Storyboard 中點選現有的 **View Controller**。
2. 選單列選擇 **Editor → Embed In → Navigation Controller**。
3. Xcode 會自動加入一個 Navigation Controller，並把「初始視圖控制器」的箭頭移到它身上。
4. 點選 Navigation Controller 裡的 **Navigation Bar**，在右側屬性檢查器勾選 **Prefers Large Titles**，讓標題以 iOS 原生風格的大字顯示。

4.2 步驟二：設定導覽列標題與「+」按鈕

1. 點選 View Controller 上方的 **Navigation Item**，在右側屬性檢查器把 **Title** 改為「待辦事項」。
2. 打開右上角的 元件庫（**Library**，快捷鍵 **Cmd + Shift + L**），搜尋 **Bar Button Item**，拖到導覽列的右側。
3. 選取這個 Bar Button Item，在屬性檢查器中把 **System Item** 設為 **Add**，它就會變成系統標準的「+」圖示。

4.3 步驟三：加入 Table View 與 Auto Layout 約束

1. 從元件庫拖一個 **Table View** 到 View Controller 的畫面上，調整大小蓋滿整個畫面。
2. 選取 Table View，點編輯區右下角的 **Add New Constraints**（十字方框圖示），設定四邊約束：
   - **Top** 貼齊 Safe Area 上緣，間距 0
   - **Leading**（左）、**Trailing**（右）、**Bottom**（下）貼齊畫面邊緣，間距 0
3. 在屬性檢查器中，把 Table View 的 **Style** 設為 **Inset Grouped**（圓角卡片風格，更接近 iOS 原生質感）。

觀念：Auto Layout 與 Safe Area

Auto Layout 是 iOS 的自動排版系統：你不是指定元件的絕對座標，而是描述「約束（Constraints）」——例如「Table View 的左邊貼齊父視圖左邊」。這樣不管是小螢幕 iPhone 還是大螢幕 iPad，畫面都能自動調整。**Safe Area**（安全區域）則是避開瀏海、動態島與底部手勢條的區域，把內容約束在 Safe Area 內就不會被遮住。

4.4 步驟四：設計 Prototype Cell（原型儲存格）

1. 選取 Table View，在屬性檢查器把 **Prototype Cells** 數量設為 **1**，Table View 上會出現一個空白儲存格。
2. 選取這個儲存格（Table View Cell），做兩個設定：
   - **Style** 設為 **Subtitle**：儲存格會有「主標題 + 副標題」兩行文字，我們用主標題顯示事項內容、副標題顯示建立時間。
   - **Identifier** 填入 `TodoCell`：這是儲存格的「重用識別碼」，程式碼中會用這個字串向 Table View 索取儲存格，兩邊字串必須完全一致。

注意

如果程式執行時當機並出現 `unable to dequeue a cell with identifier TodoCell` 錯誤，就是 Storyboard 裡的 Identifier 和程式碼裡的字串不一致（大小寫、空格都要相同）。

4.5 步驟五：連結 IBOutlet 與 IBAction

Storyboard 上的元件要和程式碼互動，需要建立兩種連結：

- **IBOutlet**：程式碼中的「變數」指向 Storyboard 元件，讓程式可以操作它（例如叫 Table View 重新載入資料）。
- **IBAction**：Storyboard 元件觸發事件時（例如按鈕被點擊），呼叫程式碼中的「方法」。

操作方式：

1. 開啟 **Assistant Editor**（雙併排編輯器）：按住 `Option` 點擊 `ViewController.swift`，讓 Storyboard 與程式碼並排顯示。
2. 連 **Outlet**：按住 `Control`，從 Table View 拖曳到程式碼 class 內部，放開後選 **Outlet**，名稱填 `tableView`。
3. 連 **Action**：按住 `Control`，從「+」按鈕拖曳到程式碼中，Connection 選 **Action**，名稱填 `addButtonTapped`，Type 選 `UIBarButtonItem`。
4. 連 **dataSource** 與 **delegate**：按住 `Control`，從 Table View 拖曳到畫面上方的黃色 **View Controller** 圖示，在彈出選單中分別勾選 **dataSource** 與 **delegate**（要拖兩次）。

小提示

dataSource 與 delegate 也可以用程式碼設定（`tableView.dataSource = self`），但在 Storyboard 專案中習慣直接在介面上連結，程式碼更乾淨。

4.6 完成後的 Storyboard 結構

完成後，你的 Storyboard 應該包含這樣的階層：

```
Navigation Controller（初始視圖控制器）
└── ViewController（標題：待辦事項）
    ├── Navigation Item
    │   └── Bar Button Item（System Item: Add）→ IBAction:
    ↪  addButtonTapped
```

```
    └── View
        └── Table View (Style: Inset Grouped) → IBOutlet: tableView
            │                                 → dataSource /
            ↪  delegate: ViewController
            └── Prototype Cell (Style: Subtitle, Identifier:
            ↪  TodoCell)
```

第 5 章

用 Core Data 建立資料模型

5.1 為什麼需要 Core Data？

如果只把待辦事項存在陣列（Array）中，App 一關閉資料就消失了。**Core Data** 是 Apple 官方的資料持久化框架，能把物件存進裝置上的資料庫（底層是 SQLite），並提供查詢、排序、關聯等完整功能。

觀念：Core Data 三個核心角色

- **NSPersistentContainer**（持久化容器）：Core Data 的總入口，負責載入資料模型、建立底層資料庫。
- **NSManagedObjectContext**（管理物件內容，簡稱 context）：像一張「工作草稿紙」——所有新增、修改、刪除都先發生在 context 裡，呼叫 `save()` 才真正寫入資料庫。
- **NSManagedObject**（管理物件）：資料庫中的一筆資料在程式中的化身。我們的 `TodoItem` 就是它的子類別。

5.2 建立 TodoItem 實體（Entity）

1. 在左側導覽區點開 `TodoList_Storyboard.xcdatamodeld`，進入資料模型編輯器。
2. 點左下角 **Add Entity**，把新實體命名為 `TodoItem`。
3. 在 **Attributes** 區塊點「+」，新增三個屬性：

屬性名稱	型別	用途
`title`	String	待辦事項的文字內容
`isCompleted`	Boolean	是否已完成（預設 NO）
`createdAt`	Date	建立時間（用來排序與顯示）

表 5.1: TodoItem 實體的三個屬性

4. 選取 `TodoItem` 實體，在右側 **Data Model Inspector** 中確認 **Codegen** 設為 **Class Definition**——Xcode 會在建置時自動產生 `TodoItem` 類別，我們不需要手寫。

自動產生的類別大致等同於：

```
public class TodoItem: NSManagedObject {
    @NSManaged public var title: String?
    @NSManaged public var isCompleted: Bool
    @NSManaged public var createdAt: Date?
}
```

小提示

注意 `title` 與 `createdAt` 的型別是「可選型別（Optional）」`String?` 與 `Date?`——這是 Core Data 產生程式碼的慣例，表示值有可能是 `nil`（不存在）。之後在程式中讀取時，需要用 `if let` 或 `??` 安全地「解包（unwrap）」。

第 6 章

逐行解說：AppDelegate.swift

`AppDelegate.swift` 大部分由 Xcode 範本產生（因為建立專案時勾選了 Core Data）。理解它是理解 Core Data 運作的第一步。

6.1 App 啟動與 Scene 管理

```
8  import UIKit
9  import CoreData
10
11 @main
12 class AppDelegate: UIResponder, UIApplicationDelegate {
13
14     func application(_ application: UIApplication,
       ↪ didFinishLaunchingWithOptions launchOptions:
       ↪ [UIApplication.LaunchOptionsKey: Any]?) -> Bool {
15         // Override point for customization after application
           ↪ launch.
16         return true
17     }
```

第 8 行　`import UIKit`：匯入 UIKit 框架。UIKit 提供所有介面相關類別（`UIViewController`、`UITableView`、`UIButton` 等），幾乎每個 iOS 介面檔案的第一行都是它。

第 9 行　`import CoreData`：匯入 Core Data 框架，才能使用 `NSPersistentContainer` 等類別。

第 11 行　`@main`：屬性標記（attribute），告訴編譯器「App 從這個類別開始執行」。整個專案只能有一個 `@main`。

第 12 行　宣告 `AppDelegate` 類別，繼承 `UIResponder`（能回應系統事件）並遵循 `UIApplicationDelegate` 協定（定義了 App 生命週期的各個回呼方法）。

第 16–19 行

`didFinishLaunchingWithOptions`：App 啟動完成後系統呼叫的第一個方法，適合放「全 App 只需做一次」的初始化。回傳 `true` 表示啟動成功。本專案不需要額外初始化，維持範本原樣。

6.2 Core Data 堆疊（Stack）

```
37    lazy var persistentContainer: NSPersistentContainer = {
38        let container = NSPersistentContainer(name:
          ↪ "TodoList_Storyboard")
39        container.loadPersistentStores(completionHandler: {
          ↪ (storeDescription, error) in
40            if let error = error as NSError? {
41                fatalError("Unresolved error \(error),
                  ↪ \(error.userInfo)")
42            }
43        })
44        return container
45    }()
```

第 37 行　`lazy var`：「延遲初始化」變數——第一次被存取時才會執行後面的閉包來建立值。Core Data 初始化成本較高，用 `lazy` 可以避免 App 一啟動就做這件事。注意結尾第 45 行的 `()`——這是「立即執行的閉包」寫法，閉包的回傳值成為變數的值。

第 38 行　建立 `NSPersistentContainer`，參數 `name` 必須和資料模型檔名一致（`TodoList_Storyboard.xcdatamodeld` 去掉副檔名），容器靠這個名字找到模型檔。

第 39 行　`loadPersistentStores`：載入（或第一次執行時建立）底層資料庫檔案。這是非同步操作，完成後呼叫 `completionHandler` 閉包。

第 40–42 行

如果載入失敗(例如磁碟已滿)，`error` 不為 `nil`，此處以 `fatalError` 直接讓 App 終止並印出錯誤。範本註解也提醒：正式上架的 App 應改為更友善的錯誤處理。

第 44 行　把設定完成的容器回傳，成為 `persistentContainer` 的值。

6.3 儲存輔助方法 saveContext()

```
66  func saveContext () {
67      let context = persistentContainer.viewContext
68      if context.hasChanges {
69          do {
70              try context.save()
71          } catch {
72              let nserror = error as NSError
73              fatalError("Unresolved error \(nserror),
                ↪  \(nserror.userInfo)")
74          }
75      }
76  }
```

第 67 行　取得容器的 `viewContext`——主執行緒專用的 context，所有和畫面相關的資料操作都用它。

第 68 行　`hasChanges`：只有 context 上真的有未儲存的變更時才執行儲存，避免不必要的寫入。

第 69–71 行

`context.save()` 可能拋出錯誤（throws），所以要放在 `do { try ... } catch` 區塊中。`try` 標記「這行可能失敗」，失敗時跳到 `catch`。

第 72–74 行

儲存失敗時印出錯誤並終止。這個方法會在 App 進入背景時被 `SceneDelegate` 呼叫，做「最後一次保險儲存」。

第 7 章

逐行解說：SceneDelegate.swift

自 iOS 13 起，App 的「視窗」由 Scene 管理（一個 App 可以有多個視窗，例如 iPad 分割畫面）。`SceneDelegate.swift` 幾乎全部維持範本原樣，只需理解兩個重點：

```
10 class SceneDelegate: UIResponder, UIWindowSceneDelegate {
11
12     var window: UIWindow?
13
14     func scene(_ scene: UIScene, willConnectTo session:
       ↪  UISceneSession, options connectionOptions:
       ↪  UIScene.ConnectionOptions) {
15         guard let _ = (scene as? UIWindowScene) else { return }
16     }
```

第 12 行　`window`：這個 Scene 對應的視窗物件。因為我們使用 Storyboard，系統會自動建立視窗並載入 `Main.storyboard`，不需要手動寫程式。

第 15–17 行

`willConnectTo`：Scene 即將連接（顯示）時呼叫。`guard let` 是 Swift 的「提前退出」語法：嘗試把 `scene` 轉型成 `UIWindowScene`，失敗就直接 `return`。範本在此不做額外設定。

```
44     func sceneDidEnterBackground(_ scene: UIScene) {
45         (UIApplication.shared.delegate as?
           ↪  AppDelegate)?.saveContext()
46     }
```

第 44 行　`sceneDidEnterBackground`：使用者切到別的 App 或回主畫面時呼叫。

第 45 行　透過 `UIApplication.shared.delegate` 取得 AppDelegate，呼叫上一章的 `saveContext()`，確保進背景前所有未儲存的變更都寫入資料庫。`as?` 是安全轉型（失敗回傳 `nil`），配合 `?.` 可選鏈：如果轉型失敗，整行安靜地不執行、不會當機。

第 8 章

逐行解說：ViewController.swift（本教材核心）

這是我們自己撰寫的主要檔案，包含 App 的所有功能邏輯。以下依照程式碼順序，分段逐行解說。

8.1 匯入框架與類別宣告

```
8  import UIKit
9  import CoreData
10
11 class ViewController: UIViewController {
```

第 8–9 行　匯入 UIKit（介面）與 CoreData（資料儲存）兩個框架。

第 11 行　宣告 `ViewController` 類別，繼承自 `UIViewController`——所有「管理一個畫面」的類別都繼承它。Storyboard 中的畫面已透過 Identity Inspector 的 Custom Class 設定指向這個類別。

8.2 IBOutlet 與屬性宣告

```
13     // MARK: - IBOutlet
14
15     /// 與 Storyboard 中的 Table View 連結的 Outlet
16     @IBOutlet weak var tableView: UITableView!
17
```

```
18      // MARK: - 屬性
19
20      /// 存放所有待辦事項的陣列（資料來源）
21      var todoItems: [TodoItem] = []
22
23      /// Core Data 的管理物件內容（Managed Object Context），
24      /// 透過 AppDelegate 的 persistentContainer 取得
25      var context: NSManagedObjectContext {
26          (UIApplication.shared.delegate as!
            ↪  AppDelegate).persistentContainer.viewContext
27      }
```

第 13 行　`// MARK: -` 是給 Xcode 看的特殊註解：會在編輯器上方的跳轉選單中產生分節標記，方便在長檔案中快速導覽。

第 16 行　`@IBOutlet`：標記這個屬性可以連到 Storyboard 元件（IB = Interface Builder）。`weak`：弱參照，避免「循環參照」造成記憶體無法釋放（視圖層級已強持有這個 Table View，Outlet 只需弱參照即可）。結尾的 `!` 是「隱式解包可選型別」：宣告時還沒有值，但我們保證 Storyboard 載入後它一定存在。

第 21 行　`todoItems`：型別為 `[TodoItem]` 的陣列，初始為空陣列 `[]`。這是畫面顯示的資料來源——Table View 顯示幾列、每列顯示什麼，都以這個陣列為準。

第 25–27 行　`context` 是「計算屬性（computed property）」：沒有自己儲存值，每次讀取時執行大括號中的程式來取值。這裡透過 AppDelegate 拿到全 App 共用的 `viewContext`。`as!` 是強制轉型——因為我們確定 delegate 一定是 `AppDelegate`，可以放心使用。

8.3 空狀態標籤與日期格式器

```
29      /// 當清單為空時顯示的提示標籤
30      let emptyStateLabel: UILabel = {
```

```
31          let label = UILabel()
32          label.text = " 目前沒有待辦事項\n點右上角「+」新增一筆吧！"
33          label.numberOfLines = 0
34          label.textAlignment = .center
35          label.textColor = .secondaryLabel
36          label.font = .preferredFont(forTextStyle: .body)
37          return label
38      }()
39
40      /// 用來把 Date 轉成易讀字串的日期格式器
41      let dateFormatter: DateFormatter = {
42          let formatter = DateFormatter()
43          formatter.dateStyle = .medium
44          formatter.timeStyle = .short
45          formatter.locale = Locale(identifier: "zh_Hant_TW")
46          return formatter
47      }()
```

第 30 行　和 AppDelegate 的 `persistentContainer` 一樣，使用「閉包立即執行」的寫法（結尾 `()`）來初始化屬性：在閉包裡建立物件、設定好所有屬性、再回傳。這是 Swift 常見的初始化慣用寫法，比在 `viewDidLoad` 中逐行設定更整潔。

第 31 行　建立一個 `UILabel`（文字標籤）。

第 32 行　設定顯示文字，`\\n` 是換行符號。

第 33 行　`numberOfLines = 0` 表示「不限行數」，文字可自動換行。

第 34 行　文字置中對齊。`.center` 是 `NSTextAlignment.center` 的簡寫——Swift 能從屬性型別推斷列舉，所以可以省略型別名稱。

第 35 行　`.secondaryLabel` 是系統的「次要文字」語意顏色，會自動配合深色 / 淺色模式變化。

第 36 行　`preferredFont(forTextStyle:)` 使用「動態字型」：字級跟隨使用者在系統設定中的文字大小偏好，是無障礙設計的最佳實踐。

第 41–47 行

`DateFormatter` 把 `Date` 物件轉成人類可讀的字串。`dateStyle = .medium` 顯示「2026 年 7 月 15 日」、`timeStyle = .short` 顯示「下午 2:38」。`locale` 指定臺灣繁體中文格式。建立 DateFormatter 的成本不低，宣告成屬性重複使用（而不是每次都建立新的）是重要的效能習慣。

8.4 viewDidLoad：畫面初始化

```
51    override func viewDidLoad() {
52        super.viewDidLoad()
53
54        // 讓導覽列顯示大標題
55        navigationController?.navigationBar.prefersLargeTitles =
          ↪ true
56
57        // 設定空狀態提示為 Table View 的背景視圖
58        tableView.backgroundView = emptyStateLabel
59
60        // 從 Core Data 讀取已儲存的待辦事項
61        fetchTodoItems()
62    }
```

第 51 行　`override`：覆寫父類別 `UIViewController` 的方法。`viewDidLoad()` 在視圖載入完成後被系統呼叫一次，是做畫面初始化的標準位置。

第 52 行　先呼叫 `super.viewDidLoad()`，讓父類別完成它自己的初始化——覆寫生命週期方法時務必記得這行。

第 55 行　`navigationController?` 取得包住自己的導覽控制器（可選鏈：若不存在則整行跳過），開啟大標題模式。

第 58 行　把空狀態標籤設為 Table View 的 `backgroundView`——它會顯示在儲存格「後面」，之後我們用顯示 / 隱藏它來呈現空狀態。

第 61 行　呼叫自訂方法讀取資料（下一節解說）。

8.5 fetchTodoItems：從 Core Data 讀取資料

```
67      /// 從 Core Data 讀取所有待辦事項，並依建立時間排序（新的在前）
68      func fetchTodoItems() {
69          let request: NSFetchRequest<TodoItem> =
            ↪ TodoItem.fetchRequest()
70          request.sortDescriptors = [NSSortDescriptor(key:
            ↪ "createdAt", ascending: false)]
71
72          do {
73              todoItems = try context.fetch(request)
74          } catch {
75              print(" 讀取資料失敗：\(error)")
76          }
77
78          // 依資料是否為空，決定要不要顯示空狀態提示
79          emptyStateLabel.isHidden = !todoItems.isEmpty
80          tableView.reloadData()
81      }
```

第 69 行　建立「擷取請求（Fetch Request）」，泛型 `<TodoItem>` 指明「我要查詢 TodoItem 這種實體」。`TodoItem.fetchRequest()` 是 Xcode 自動產生的便利方法。

第 70 行　設定排序描述器：依 `createdAt` 欄位排序，`ascending: false` 表示遞減——最新建立的排在最上面。注意 `sortDescriptors` 是陣列，可以指定多重排序條件。

第 72–76 行　`context.fetch(request)` 執行查詢並回傳結果陣列，指派給 `todoItems`。查詢可能失敗（throws），因此包在 `do-catch` 中；失敗時用 `print` 印出錯誤方便除錯，`\\(error)` 是「字串插值」——把變數值嵌入字串。

第 79 行　一行完成空狀態切換：`todoItems.isEmpty` 為真（沒有資料）時，`!` 取反得 `false`，即 `isHidden = false`——顯示提示。有資料時則隱

藏。

第 80 行 `reloadData()`：命令 Table View 重新向 dataSource 詢問所有資料並重繪畫面。修改 **`todoItems`** 後一定要呼叫這行，畫面才會更新。

8.6 saveAndReload：儲存並更新畫面

```
83      /// 將 context 中的變更寫入永久儲存區，並重新載入畫面
84      func saveAndReload() {
85          do {
86              try context.save()
87          } catch {
88              print(" 儲存資料失敗：\(error)")
89          }
90          fetchTodoItems()
91      }
```

第 85–89 行

`context.save()` 把 context 裡累積的所有變更（新增、修改、刪除）一次寫入資料庫。同樣用 **`do-catch`** 處理可能的錯誤。

第 90 行 儲存後重新查詢並刷新畫面。把「儲存 + 刷新」包成一個方法，之後新增、完成、編輯、刪除四個功能都能重複使用——這就是 **DRY（Don't Repeat Yourself）**原則。

8.7 addButtonTapped：新增待辦事項

```
95      /// 使用者點擊右上角「+」按鈕時呼叫，跳出輸入視窗新增待辦事項
96      @IBAction func addButtonTapped(_ sender: UIBarButtonItem) {
97          let alert = UIAlertController(title: " 新增待辦事項",
98                                        message: " 請輸入事項內容",
99                                        preferredStyle: .alert)
100
101         // 在 Alert 中加入一個文字輸入框
```

```
102         alert.addTextField { textField in
103             textField.placeholder = " 例如：買牛奶"
104         }
105 
106         // 「新增」按鈕：取得輸入文字並建立新的 TodoItem
107         let addAction = UIAlertAction(title: " 新增", style:
    ↪  .default) { [weak self] _ in
108             guard let self = self,
109                   let text = alert.textFields?.first?.text?
110                       .trimmingCharacters(in:
    ↪  .whitespacesAndNewlines),
111                   !text.isEmpty else { return }
112 
113             // 在 Core Data 的 context 中建立一筆新資料
114             let newItem = TodoItem(context: self.context)
115             newItem.title = text
116             newItem.isCompleted = false
117             newItem.createdAt = Date()
118 
119             self.saveAndReload()
120         }
121 
122         // 「取消」按鈕：不做任何事
123         let cancelAction = UIAlertAction(title: " 取消", style:
    ↪  .cancel)
124 
125         alert.addAction(addAction)
126         alert.addAction(cancelAction)
127         present(alert, animated: true)
128     }
```

第 96 行　`@IBAction`：標記這個方法可連到 Storyboard 事件——「+」按鈕被點擊時系統會呼叫它。參數 `sender` 是觸發事件的元件；參數名前的底線 `_` 表示呼叫時不需寫參數標籤。

第 97–99 行

建立 `UIAlertController`（系統彈窗），`preferredStyle: .alert`

是置中對話框樣式（另一種 `.actionSheet` 是從底部升起的選單）。

第 102–104 行

`addTextField` 在彈窗中加入文字輸入框。大括號是「尾隨閉包（trailing closure）」——當閉包是最後一個參數時，可以寫在括號外面。閉包參數 `textField` 是系統建立好的輸入框，我們設定它的佔位提示文字。

第 107 行 建立「新增」按鈕。第三個參數（尾隨閉包）是按鈕被點擊時要執行的程式。`[weak self]` 是「捕捉列表」：閉包以弱參照捕捉 `self`，避免閉包與控制器互相強持有造成記憶體洩漏——在逃逸閉包中使用 **self** 時的標準防護寫法。閉包參數用 `_` 忽略（我們不需要用到該 action 物件本身）。

第 108–111 行

`guard let` 多條件安全檢查，任何一項失敗就 `return`：(1) `self` 還存在；(2) 成功取得輸入框文字並用 `trimmingCharacters` 去除前後空白與換行；(3) 處理後的文字不是空字串。這確保使用者輸入純空白時不會新增空事項。

第 114 行 **Core Data** 新增資料的關鍵語法：`TodoItem(context:)` 在指定的 context 中建立一筆新的管理物件。注意——此時資料只存在於「草稿」中，尚未寫入資料庫。

第 115–117 行

設定三個屬性：標題、未完成、建立時間為現在（`Date()` 產生當下時間）。

第 119 行 呼叫 `saveAndReload()`——這時才真正寫入資料庫並刷新畫面。

第 123 行 「取消」按鈕使用 `.cancel` 樣式（粗體顯示），不傳入閉包表示點了不做任何事、只關閉彈窗。

第 125–127 行

把兩個按鈕加入彈窗，最後用 `present(_:animated:)` 以動畫顯示彈窗。

8.8 presentEditAlert：編輯既有事項

```
132     /// 跳出輸入視窗，讓使用者修改既有事項的標題
133     func presentEditAlert(for item: TodoItem) {
```

```
134         let alert = UIAlertController(title: " 編輯待辦事項",
135                                       message: " 請修改事項內容",
136                                       preferredStyle: .alert)
137 
138         alert.addTextField { textField in
139             textField.text = item.title
140         }
141 
142         let saveAction = UIAlertAction(title: " 儲存", style:
    ↪   .default) { [weak self] _ in
143             guard let self = self,
144                   let text = alert.textFields?.first?.text?
145                       .trimmingCharacters(in:
    ↪   .whitespacesAndNewlines),
146                   !text.isEmpty else { return }
147 
148             item.title = text
149             self.saveAndReload()
150         }
151 
152         alert.addAction(saveAction)
153         alert.addAction(UIAlertAction(title: " 取消", style:
    ↪   .cancel))
154         present(alert, animated: true)
155     }
```

第 133 行　參數標籤 `for item: TodoItem`：外部呼叫時寫 `presentEditAlert(for: 某事項)`，讀起來像自然語言——這是 Swift 的 API 命名慣例。

第 139 行　與新增不同：把輸入框的初始文字設為現有的標題，讓使用者在原文字上修改。

第 148 行　**Core Data** 修改資料就是這麼簡單：直接改物件的屬性即可，context 會自動追蹤這筆物件「已被修改」，下次 `save()` 時寫入。

8.9 extension 與 UITableViewDataSource

```
155 // MARK: - UITableViewDataSource
156
157 extension ViewController: UITableViewDataSource {
158
159     /// 回傳列表的列數 = 待辦事項的數量
160     func tableView(_ tableView: UITableView, numberOfRowsInSection
        ↪ section: Int) -> Int {
161         return todoItems.count
162     }
```

第 157 行 `extension`（擴展）：在類別本體之外為它「加掛」功能。慣例上，每個協定的實作用一個獨立的 extension 包起來，讓程式碼分區清楚。這行表示「ViewController 遵循 UITableViewDataSource 協定，實作寫在這個區塊」。

第 160–162 行

DataSource 必答題之一：「這個表格有幾列？」我們回答：`todoItems` 陣列有幾個元素就有幾列。`section` 是分區編號——本 App 只有一個分區，不需要理會它。

```
164     /// 設定每一列儲存格的內容
165     func tableView(_ tableView: UITableView,
166                    cellForRowAt indexPath: IndexPath) ->
                       ↪ UITableViewCell {
167         // 從重用佇列取出識別碼為 "TodoCell" 的儲存格
168         let cell = tableView.dequeueReusableCell(withIdentifier:
            ↪ "TodoCell",
169                                                  for: indexPath)
170         let item = todoItems[indexPath.row]
171
172         // 已完成的事項：加上刪除線並變灰
173         let title = item.title ?? ""
174         if item.isCompleted {
175             let attributed = NSAttributedString(
```

```
176                 string: title,
177                 attributes: [
178                     .strikethroughStyle:
                        ↪ NSUnderlineStyle.single.rawValue,
179                     .foregroundColor: UIColor.secondaryLabel
180                 ])
181             cell.textLabel?.attributedText = attributed
182         } else {
183             cell.textLabel?.attributedText = nil
184             cell.textLabel?.text = title
185             cell.textLabel?.textColor = .label
186         }
187
188         // 副標題顯示建立時間
189         if let createdAt = item.createdAt {
190             cell.detailTextLabel?.text = dateFormatter.string(from:
                ↪ createdAt)
191         } else {
192             cell.detailTextLabel?.text = nil
193         }
194         cell.detailTextLabel?.textColor = .secondaryLabel
195
196         // 已完成的事項在右側顯示勾勾
197         cell.accessoryType = item.isCompleted ? .checkmark : .none
198
199         return cell
200     }
201 }
```

第 165–166 行

DataSource 必答題之二：「第 N 列的儲存格長什麼樣？」Table View 每次要顯示某一列時就呼叫這個方法。`indexPath` 包含 `section`（分區）與 `row`（列號）。

第 168–169 行

`dequeueReusableCell`：從「重用佇列」取出識別碼為 `TodoCell` 的儲存格（就是我們在 Storyboard 設計的 Prototype Cell）。如果佇列裡有被回

收的舊儲存格就直接重用，沒有才建立新的——這就是第三章提到的儲存格重用機制。

第 170 行　用列號從陣列取出對應的事項：第 0 列對應 `todoItems[0]`，以此類推。

第 173 行　`??` 是「nil 聚合運算子」：`item.title` 是 `String?`，若為 `nil` 則改用空字串，確保 `title` 一定有值。

第 174–181 行

若事項已完成，建立 `NSAttributedString`（帶樣式的字串）：加上單線刪除線（`.strikethroughStyle`）並把文字設為次要灰色，指派給主標題的 `attributedText`。

第 182–186 行

重用機制的重要細節：若事項未完成，必須把 `attributedText` 明確設回 `nil`、重設文字與顏色。因為這個儲存格可能是「回收再利用」的——若不清除，上一筆已完成事項的刪除線樣式會殘留在這筆未完成的事項上，造成經典的「儲存格內容錯亂」bug。

第 189–193 行

用 `if let` 解包 `createdAt`，有值就交給第八章宣告的 `dateFormatter` 轉成「2026 年 7 月 15 日下午 2:38」格式，顯示在副標題。

第 197 行　三元運算子 `條件 ? A : B`：已完成顯示系統勾勾（`.checkmark`），否則不顯示配件。

第 199 行　把設定完成的儲存格回傳給 Table View 顯示。

8.10 UITableViewDelegate：點擊與滑動操作

```
205 // MARK: - UITableViewDelegate
206
207 extension ViewController: UITableViewDelegate {
208
209     /// 點擊某一列：切換「已完成 / 未完成」狀態
210     func tableView(_ tableView: UITableView, didSelectRowAt
        ↪ indexPath: IndexPath) {
211         tableView.deselectRow(at: indexPath, animated: true)
```

```
212
213         let item = todoItems[indexPath.row]
214         item.isCompleted.toggle()
215         saveAndReload()
216     }
```

第 210 行　`didSelectRowAt`：使用者點擊某一列時，Table View 透過 delegate 通知我們。

第 211 行　`deselectRow`：取消該列的「選取反白」狀態並附帶淡出動畫——不做這行，該列會一直維持灰底。

第 213–214 行

取出被點擊的事項，`toggle()` 把布林值反轉（true 變 false、false 變 true）——一行完成「切換完成狀態」。

第 215 行　儲存並刷新，畫面上立即出現（或移除）刪除線與勾勾。

```
218     /// 向左滑動：顯示「刪除」按鈕
219     func tableView(_ tableView: UITableView,
220                    trailingSwipeActionsConfigurationForRowAt
                        ↪  indexPath: IndexPath)
221     -> UISwipeActionsConfiguration? {
222         let deleteAction = UIContextualAction(style: .destructive,
223                                               title: " 刪除") {
                                                ↪  [weak self] _, _,
                                                ↪  completion in
224             guard let self = self else { return }
225
226             // 從 Core Data 中刪除該筆資料
227             let item = self.todoItems[indexPath.row]
228             self.context.delete(item)
229             self.saveAndReload()
230             completion(true)
231         }
232         return UISwipeActionsConfiguration(actions: [deleteAction])
233     }
```

```
234
235     /// 向右滑動：顯示「編輯」按鈕
236     func tableView(_ tableView: UITableView,
237                    leadingSwipeActionsConfigurationForRowAt
                       ↪ indexPath: IndexPath)
238     -> UISwipeActionsConfiguration? {
239         let editAction = UIContextualAction(style: .normal,
240                                             title: " 編輯") { [weak
                                              ↪ self] _, _,
                                              ↪ completion in
241             guard let self = self else { return }
242             self.presentEditAlert(for:
                 ↪ self.todoItems[indexPath.row])
243             completion(true)
244         }
245         editAction.backgroundColor = .systemBlue
246         return UISwipeActionsConfiguration(actions: [editAction])
247     }
248 }
```

第 219–221 行

`trailingSwipeActions...`：設定「從尾端滑入」（在由左至右的語言環境即向左滑）出現的按鈕。回傳型別 `UISwipeActionsConfiguration?` 可為 `nil`（表示該列不提供滑動操作）。

第 222–223 行

建立情境操作按鈕：`.destructive` 樣式自動顯示為紅色，代表破壞性操作。閉包的三個參數依序是 action 本身、按鈕視圖、完成回呼——前兩個用 `_` 忽略。

第 227–229 行

Core Data 刪除資料：`context.delete(item)` 把該物件標記為刪除，`save()` 時真正從資料庫移除。之後 `saveAndReload()` 會重新查詢，畫面上該列即消失。

第 230 行　呼叫 `completion(true)` 告訴系統「操作已完成」，讓滑動選單收合。

第 232 行　把按鈕包進 `UISwipeActionsConfiguration` 回傳，陣列可放多個按鈕。

第 236–238 行

`leadingSwipeActions...`：「從前端滑入」（即向右滑）的按鈕，用來放「編輯」。

第 239–244 行

`.normal` 樣式預設灰色，第 245 行手動改為系統藍色。點擊後呼叫前面寫好的 `presentEditAlert(for:)` 彈出編輯視窗。

小提示

你可能注意到一個模式：所有資料操作都遵循同一個流程——修改 context（新增 / 改屬性 / 刪除）→ `saveAndReload()` → 畫面自動更新。掌握這個「單向資料流」心智模型，Core Data 就不再神祕。

第 9 章

建置、執行與自動化測試

9.1 在模擬器上執行

1. 在 Xcode 上方工具列的裝置選單中選擇一台模擬器（例如 **iPhone 17**）。
2. 按 **Run**（`Cmd + R`）。Xcode 會編譯程式、啟動模擬器並安裝執行 App。
3. 測試所有功能：新增幾筆事項、點擊切換完成、左滑刪除、右滑編輯。
4. 按模擬器的 Home（`Cmd + Shift + H`）把 App 收到背景，再從 Xcode 重新執行——確認資料仍然存在，驗證 Core Data 儲存成功。

也可以使用終端機以指令建置（進階；適合自動化）：

```
xcodebuild -project TodoList-Storyboard.xcodeproj \
           -scheme TodoList-Storyboard \
           -destination 'platform=iOS Simulator,name=iPhone 17' \
           build
```

9.2 撰寫 UI 自動化測試

專案的 `TodoList-StoryboardUITests` target 可以撰寫「模擬真人操作」的自動化測試。以下測試會自動點擊「+」、輸入文字、驗證清單內容，並點擊標記完成：

```
import XCTest

final class TodoList_StoryboardUITests: XCTestCase {

    override func setUpWithError() throws {
        continueAfterFailure = false
```

```
    }

    /// 完整流程測試：新增兩筆待辦事項，並將其中一筆標記為完成
    @MainActor
    func testAddAndCompleteTodoItems() throws {
        let app = XCUIApplication()
        app.launch()

        // 新增第一筆：買牛奶
        app.navigationBars.buttons["Add"].tap()
        let textField = app.alerts.textFields.firstMatch
        XCTAssertTrue(textField.waitForExistence(timeout: 3))
        textField.typeText(" 買牛奶")
        app.alerts.buttons[" 新增"].tap()

        // 新增第二筆：寫 iOS 作業
        app.navigationBars.buttons["Add"].tap()
        XCTAssertTrue(textField.waitForExistence(timeout: 3))
        textField.typeText(" 寫 iOS 作業")
        app.alerts.buttons[" 新增"].tap()

        // 確認兩筆都出現在列表上
        XCTAssertTrue(app.staticTexts[" 買牛
        ↪  奶"].waitForExistence(timeout: 3))
        XCTAssertTrue(app.staticTexts[" 寫 iOS 作業"].exists)

        // 點擊「買牛奶」把它標記為已完成
        app.staticTexts[" 買牛奶"].tap()

        sleep(1)
    }
}
```

重點語法說明：

- `XCUIApplication().launch()`：啟動被測試的 App。
- `app.navigationBars.buttons["Add"]`：以「無障礙識別」查詢介面元素

——系統的 Add 按鈕識別名稱是 `"Add"`。

- `waitForExistence(timeout:)`：等待元素出現（最多等 3 秒），處理動畫等非同步狀況。
- `XCTAssertTrue(...)`：斷言——條件不成立則測試失敗。
- `typeText(...)` 與 `tap()`：模擬打字與點擊。

在 Xcode 中打開測試檔案，點方法左側的菱形按鈕即可執行；本專案實測測試通過（**TEST SUCCEEDED**）。

9.3 常見錯誤排查

症狀	原因與解法
啟動即當機：`this class is not key value coding-compliant for the key ...`	Storyboard 的 Outlet 連到已改名或刪除的屬性。在 Storyboard 中右鍵檢查元件的連結，刪除黃色驚嘆號的失效連結後重連。
當機：`unable to dequeue a cell with identifier`	程式碼中的識別碼字串與 Prototype Cell 的 Identifier 不一致。
點「+」新增後畫面沒反應	忘記呼叫 `reloadData()`，或 dataSource 沒有連到 ViewController。
重開 App 資料消失	忘記呼叫 `context.save()`（本專案封裝在 `saveAndReload()` 中）。
畫面上儲存格樣式錯亂（刪除線出現在錯的列）	重用儲存格時沒有重設樣式，參考 `cellForRowAt` 中 else 分支的解說。

表 9.1: 常見錯誤與解法

第 10 章

總結與延伸練習

10.1 你學會了什麼

- 用 Xcode 建立 Storyboard + Core Data 的 iOS 專案。
- Storyboard 介面設計：Navigation Controller、Bar Button Item、Table View、Prototype Cell、Auto Layout 約束。
- IBOutlet / IBAction 的連結，以及 dataSource / delegate 的委任模式。
- Core Data 完整 CRUD：`TodoItem(context:)` 新增、直接改屬性修改、`context.delete()` 刪除、`NSFetchRequest` 查詢與排序、`context.save()` 儲存。
- Swift 核心語法：可選型別與解包（`?`、`!`、`if let`、`guard let`、`??`）、閉包與 `[weak self]`、計算屬性、`lazy`、extension、三元運算子、字串插值。
- UI 細節：NSAttributedString 刪除線、滑動操作按鈕、空狀態設計、動態字型。
- XCUITest 自動化 UI 測試。

10.2 延伸練習（由易到難）

1. 到期日：為 `TodoItem` 增加 `dueDate` 屬性，新增時用 `UIDatePicker` 選日期，過期事項顯示紅字。
2. 分區顯示：把清單分成「未完成」與「已完成」兩個 section（提示：實作 `numberOfSections` 與 `titleForHeaderInSection`）。
3. 搜尋：加入 `UISearchController`，用 `NSPredicate` 過濾 Core Data 查詢結果。
4. 拖曳排序：實作 `moveRowAt` 讓使用者長按拖曳調整順序（需要增加 `sortIndex` 屬性）。

5. **NSFetchedResultsController**：用它取代手動的 `fetchTodoItems()`，讓 Core Data 變更自動驅動 Table View 的插入／刪除動畫。
6. **iCloud** 同步：改用 `NSPersistentCloudKitContainer` 作為持久化容器，在多台裝置間同步資料。

恭喜完成你的第一個 iOS App！