用 Xcode 開發 iPhone App

從零開始的完整學習教材

以 `To-Do-List-Demo`（待辦事項 App）為例

本教材專為**完全沒有寫過程式**的初學者撰寫。
我們會逐行解釋專案中的每一段程式碼，
並完整說明一個 iOS App 從啟動到操作的程式流程。

技術主題：Swift · SwiftUI · SwiftData · Xcode

適用 Xcode 26 / iOS 26 / Swift 5

文件以 LaTeX（XeLaTeX）排版產生

目录

1 寫在最前面：這份教材怎麼讀

歡迎！如果你從來沒有寫過任何程式，這份教材正是為你準備的。我們會以一個真實、可以實際執行的小型 iPhone App ——**待辦事項清單（To-Do List）**——作為範例，帶你認識 iOS App 開發的完整樣貌。

這個 App 雖然小，但「五臟俱全」：它有資料的**儲存**、畫面的**呈現**、使用者的**操作**（新增、檢視、編輯、刪除），以及畫面之間的**切換**。換句話說，只要徹底搞懂這個專案，你就掌握了開發大多數 App 都會用到的核心觀念。

1.1 你會學到什麼

- iOS 開發到底需要哪些工具與語言（Xcode、Swift、SwiftUI、SwiftData）。
- 一個 Xcode 專案裡的每個檔案、每個資料夾各自負責什麼。
- Swift 程式語言的基礎語法（只挑本專案用得到的，務實學習）。
- 專案中**每一行程式碼**的意義。
- App 從「使用者點下圖示」到「畫面顯示、操作、資料儲存」的**完整流程**。
- 如何在 Xcode 中執行、預覽、除錯你的 App。

1.2 建議的閱讀順序

本教材的章節是**循序漸進**的，強烈建議你從頭往後讀：

1. 先讀第 2 節，建立「iOS 開發是什麼」的整體概念。
2. 接著第 3 節，認識專案的檔案結構。
3. 第 4 節補齊 Swift 語法基礎（看不懂程式時可隨時回來查）。
4. 第 5 到第 9 節，逐一精讀五個程式檔。
5. 第 11 節把所有檔案串起來，理解完整程式流程。
6. 最後第 12 節學會實際操作 Xcode，第 13 節做練習。

小提示

看到不懂的英文名詞先別緊張。本教材每出現一個新名詞，都會用「名詞解釋」方塊說明。書末第 14 節還有一份完整的名詞對照表可供查閱。

2 第一章　iOS 開發的基礎概念

在打開任何程式碼之前，我們先用白話文把「開發一個 iPhone App 需要哪些東西」講清楚。

2.1 什麼是 App？什麼是 iOS？

App（應用程式）就是你手機上一個一個的圖示：相機、訊息、地圖等等。**iOS** 則是 iPhone 的「作業系統」——它是手機的大管家，負責管理畫面、觸控、記憶體、儲存空間，並決定哪個 App 可以執行、可以使用哪些資源。

我們寫的 App，最終會被安裝到 iOS 上，由 iOS 負責把它顯示給使用者、把使用者的觸控傳給 App。

2.2 開發 iPhone App 需要的四樣東西

名詞解釋

Xcode：Apple 官方提供的「開發工具」（IDE，整合開發環境）。你在 Xcode 裡面寫程式、設計畫面、執行測試、把 App 跑在模擬器或真實手機上。可以把它想成是「寫 App 專用的超級記事本 + 工廠」。

名詞解釋

Swift：開發 iOS App 使用的「程式語言」。程式語言是人類與電腦溝通的文字。Swift 是 Apple 在 2014 年推出的現代語言，語法簡潔、相對好讀。本專案所有 `.swift` 檔案裡寫的都是 Swift。

名詞解釋

SwiftUI：一套用來「描述畫面長什麼樣子」的工具（稱為框架，framework）。你只要用程式碼**宣告**「我想要一個清單，裡面有文字和按鈕」，SwiftUI 就會幫你把它畫出來。這種寫法叫做**宣告式 UI**。

名詞解釋

SwiftData：Apple 的「資料儲存」框架。App 關掉後資料還要留著（例如你新增的待辦事項），就需要把資料存到手機裡。SwiftData 讓你用很少的程式碼就能做到「永久儲存」。

2.3 宣告式 UI：一個重要的思維轉變

傳統寫畫面的方式是「一步一步下指令」：先建立一個按鈕、再設定它的位置、再設定顏色、使用者點了之後手動更新畫面……這叫**命令式**。

SwiftUI 採用**宣告式**：你只描述「在某個狀態下，畫面應該長成什麼樣子」，當資料改變時，SwiftUI 會**自動**重新繪製需要更新的部分。

小提示

用做菜比喻：**命令式**像是「拿鍋、開火、倒油、放蛋、翻面……」一步步指揮；**宣告式**則像是直接說「我要一份蛋包飯」，廚房自己會處理細節。SwiftUI 就是那個聰明的廚房。

2.4 本專案使用的技術版本

本專案是用較新的 Xcode 建立的，採用了現代化的寫法：

- 使用 **SwiftUI** 而非舊的 Storyboard 來建立畫面。
- 使用 **SwiftData**（`@Model`）而非舊的 Core Data 來儲存資料。
- 使用 **檔案系統同步群組**（File System Synchronized Group），代表只要把 `.swift` 檔放進專案資料夾，Xcode 就會自動納入編譯，不需手動加入。

3 第二章　專案結構總覽

當你用 Xcode 建立一個新專案，它會自動產生一整組檔案。我們先鳥瞰整個 `To-Do-List-Demo` 專案的結構，再逐一深入。

3.1 資料夾與檔案地圖

```
To-Do-List-Demo/                          <- 專案根目錄
|-- To-Do-List-Demo.xcodeproj             <- 專案設定檔（Xcode 用）
|-- To-Do-List-Demo/                      <- App 的主要程式碼
|   |-- To_Do_List_DemoApp.swift          <- 程式進入點（App 從這裡開始）
|   |-- Item.swift                        <- 資料模型（一筆待辦事項長什麼樣）
|   |-- ContentView.swift                 <- 主畫面（待辦清單）
|   |-- AddItemView.swift                 <- 新增待辦的畫面
|   |-- ItemDetailView.swift              <- 檢視 / 編輯待辦的畫面
|   `-- Assets.xcassets/                  <- 圖片、顏色、App 圖示等資源
|-- To-Do-List-DemoTests/                 <- 單元測試
|   `-- To_Do_List_DemoTests.swift
`-- To-Do-List-DemoUITests/               <- 介面（UI）測試
    |-- To_Do_List_DemoUITests.swift
    `-- To_Do_List_DemoUITestsLaunchTests.swift
```

3.2 每個部分的職責

檔案／資料夾	負責什麼
`.xcodeproj`	記錄專案的所有設定：要編譯哪些檔案、App 名稱、支援的 iOS 版本、簽章資訊等。**你幾乎不會手動編輯它**，都是透過 Xcode 介面修改。
`To_Do_List_DemoApp.swift`	App 的「起點」。iOS 啟動 App 時，第一個執行的就是它，負責準備資料庫並顯示第一個畫面。
`Item.swift`	定義「一筆待辦事項」包含哪些欄位（主旨、描述、建立時間），以及如何被儲存。
`ContentView.swift`	App 打開後看到的**主畫面**：待辦清單，含「＋」新增按鈕與刪除功能。
`AddItemView.swift`	點「＋」後彈出的**新增畫面**，讓使用者輸入主旨與描述。
`ItemDetailView.swift`	點某一筆待辦後進入的**詳細／編輯畫面**，可修改後儲存或取消。
`Assets.xcassets`	存放 App 圖示（AppIcon）、主題色（AccentColor）等視覺資源。
`...Tests / ...UITests`	自動化測試程式。用來自動檢查 App 是否正常，初學階段可先略過。

名詞解釋

為什麼檔名是 `To_Do_List_DemoApp`（有底線）？ 專案名稱含有連字號「-」，但 Swift 的程式識別字不允許「-」，所以 Xcode 自動把「-」換成底線「_」來當作程式內部的名稱。

小提示

本專案使用「檔案系統同步群組」。這代表前面幾個畫面檔（`AddItemView.swift`、`ItemDetailView.swift`）即使是後來才新增的，只要放在 `To-Do-List-Demo/` 資料夾內，Xcode 就會**自動**把它們納入編譯，你不必做任何額外設定。

4 第三章　Swift 語法快速入門

這一章不是要把 Swift 全部教完，而是**只挑本專案用到的語法**做最務實的說明。讀程式時若遇到不懂的符號，回到本章查閱即可。

4.1 變數 var 與常數 let

程式需要「記住」資料，記住資料的盒子就叫**變數**。

```
var subject = "買牛奶"      // var：可以改變的變數
let pi = 3.14159           // let：不可改變的常數（設定後不能再變）
subject = "買麵包"          // 合法：var 可以重新賦值
// pi = 3.0                // 錯誤：let 不能再改
```

- `var`（variable）：之後可以改變內容。
- `let`（constant）：設定一次之後就固定。能用 `let` 就盡量用，比較安全。
- `=` 是「賦值」：把右邊的值放進左邊的盒子，**不是**數學上的等於。

4.2 型別 Type：資料的種類

每個資料都有「種類」，稱為型別。本專案常見的有：

型別	代表	範例
String	文字字串	" 買牛奶"
Int	整數	5、-3
Bool	真／假（是非值）	true、false
Double	小數	3.14
Date	日期與時間	現在時刻

Swift 通常能**自動推斷**型別。寫 `var subject = " 買牛奶"`，Swift 就知道 `subject` 是 `String`。也可以明寫：`var subject: String = " 買牛奶"`，冒號後面就是型別。

4.3 函式 func：把一段工作包起來

函式是一段「取了名字、可以重複使用」的程式碼。

```
func sayHello(name: String) {
    print("Hello, \(name)!")    // \(name) 會把變數的值嵌進字串
}
sayHello(name: "Nelson")        // 呼叫函式，印出 Hello, Nelson!
```

- `func` 是宣告函式的關鍵字。
- 括號內 `name: String` 是**參數**：呼叫時要提供的資料。
- 大括號 `{ }` 內是函式的**內容**（要執行的工作）。

- `\(name)` 是**字串插值**：把變數的值塞進文字裡。

4.4 結構 struct 與類別 class

這兩者都是「把一群相關的資料和功能打包在一起」的藍圖。

```
struct Point {          // 一個「點」有 x 與 y 兩個座標
    var x: Int
    var y: Int
}
let p = Point(x: 3, y: 5)    // 依藍圖造出一個實際的點
```

- `struct`（結構）：**數值型別**，複製時是「整份拷貝」。SwiftUI 的畫面（View）都是 struct。
- `class`（類別）：**參考型別**，複製時是「共用同一份」。SwiftData 的資料模型用 class。

小提示

簡單記法：在 SwiftUI 中，**畫面**用 `struct`，**要被長期儲存的資料**用 `class`（加上 `@Model`）。本專案正是這樣安排：三個畫面是 struct，`Item` 資料是 class。

4.5 Optional：可能「沒有值」的資料

Swift 用「?」表示一個資料**可能有值，也可能沒有**（沒有時稱為 `nil`）。本專案中你會在 `3...6` 等地方間接接觸到相關概念，這裡先建立印象即可：見到 `?` 就想到「這東西可能是空的」。

4.6 閉包 Closure：可以傳來傳去的一段程式

閉包是「一段沒有名字、但可以像資料一樣被傳遞」的程式碼，常用大括號 `{ }` 包起來。本專案中按鈕被點擊時要執行的動作，就是用閉包來表達：

```
Button("Save") {
    // 這整段大括號就是一個閉包：
    // 「當按鈕被按下時，要做的事」
    saveChanges()
}
```

4.7 屬性包裝器 Property Wrapper：以 @ 開頭的關鍵字

你會在本專案看到很多以 `@` 開頭的字，例如 `@State`、`@Query`、`@Model`、`@Environment`。它們叫做**屬性包裝器**，是 SwiftUI ／ SwiftData 提供的「特殊標記」，用來賦予一個變數額外的能力。

標記	白話意義
`@State`	「這是這個畫面自己擁有的狀態，變了就重畫畫面。」
`@Binding` / `@Bindable`	「這個資料是別人給我的，我可以雙向地讀和改。」
`@Environment`	「從系統環境拿一個現成的東西來用（例如資料庫、關閉畫面的能力）。」
`@Query`	「自動從 SwiftData 資料庫撈出符合條件的資料，且資料一變畫面就更新。」
`@Model`	「這個 class 是要被 SwiftData 儲存的資料模型。」
`@main`	「整個 App 從這裡開始執行。」

小提示

現在看不太懂這些 `@` 沒關係，後面逐行解說程式碼時，每一個出現的地方我們都會再次解釋它在「當下」扮演的角色。

5 第四章　程式進入點：To_Do_List_DemoApp.swift

我們從 App 的「起點」開始看。當使用者點下 App 圖示，iOS 會找到被標記為 `@main` 的地方，從那裡開始執行。

5.1 完整程式碼

```
1 import SwiftUI
2 import SwiftData
3
4 @main
5 struct To_Do_List_DemoApp: App {
6     var sharedModelContainer: ModelContainer = {
7         let schema = Schema([
8             Item.self,
9         ])
```

```
10          let modelConfiguration = ModelConfiguration(schema: schema,
                isStoredInMemoryOnly: false)
11
12          do {
13              return try ModelContainer(for: schema, configurations:
                    [modelConfiguration])
14          } catch {
15              fatalError("Could not create ModelContainer: \(error)")
16          }
17      }()
18
19      var body: some Scene {
20          WindowGroup {
21              ContentView()
22          }
23          .modelContainer(sharedModelContainer)
24      }
25  }
```

5.2 逐行解說

5.2.1 第 1–2 行：引入工具箱

```
1  import SwiftUI
2  import SwiftData
```

`import` 的意思是「把某個工具箱拿進來用」。這裡引入了兩個：**SwiftUI**（畫面）與 **SwiftData**（資料儲存）。沒有 import，下面就無法使用這些工具。

5.2.2 第 4 行：@main 標記

```
1  @main
```

告訴系統：「**整個 App 從這個結構開始執行**。」一個 App 只能有一個 `@main`。

5.2.3 第 5 行：宣告 App 主體

```
1  struct To_Do_List_DemoApp: App {
```

- `struct To_Do_List_DemoApp`：定義一個名為 To_Do_List_DemoApp 的結構。
- `: App`：冒號代表「遵循某個協定（protocol）」。這裡遵循 `App` 協定，等於宣告「我是一個 App」。遵循 `App` 就必須提供一個 `body`（見後）。

名詞解釋

協定（Protocol）：一份「規格合約」。當你說「我遵循 App 協定」，就承諾會提供 App 規定的東西（例如 `body`）。協定確保不同的東西具備一致的能力。

5.2.4 第 6–18 行：建立資料庫容器

```
1  var sharedModelContainer: ModelContainer = {
2      let schema = Schema([
3          Item.self,
4      ])
5      let modelConfiguration = ModelConfiguration(schema: schema,
           isStoredInMemoryOnly: false)
6
7      do {
8          return try ModelContainer(for: schema, configurations: [
               modelConfiguration])
9      } catch {
10         fatalError("Could not create ModelContainer: \(error)")
11     }
12 }()
```

這一整段在準備 App 的「資料庫」。我們拆開看：

- `sharedModelContainer`：一個變數，型別是 `ModelContainer`（資料容器）。可以把它想成「整個 App 的倉庫總管」，負責保管所有待辦事項。
- `Schema([Item.self])`：**綱要**。告訴資料庫「我要儲存的資料種類有哪些」，這裡只有一種：`Item`。`Item.self` 指的是「Item 這個型別本身」。
- `ModelConfiguration(... isStoredInMemoryOnly: false)`：**設定**。`isStoredInMemoryOnly: fal` 表示「資料要真正存到磁碟」，所以 App 關掉再開，資料還在。若設成 `true`，資料只放在記憶體，關掉就消失（測試時才會用）。
- `do { ... } catch { ... }`：**錯誤處理**。建立資料庫**有可能失敗**(例如磁碟壞了)。`do` 區塊內嘗試建立，若失敗就跳到 `catch` 區塊。
- `try ModelContainer(...)`：`try` 表示「這個動作可能出錯」，必須搭配 `do/catch`。
- `fatalError(...)`：若真的失敗，就讓 App 立即崩潰並印出原因。對這種「沒有資

料庫就無法運作」的情況，直接停止是合理的。

- 最外層的 `= { ... }()`：這是一個「立即執行的閉包」。大括號內準備好內容，結尾的 `()` 代表「現在就執行它」，把結果指派給 `sharedModelContainer`。

名詞解釋

ModelContainer / Schema / Configuration 三者關係：`Schema` 是「要存什麼資料」的清單；`ModelConfiguration` 是「要怎麼存」的設定；`ModelContainer` 則依據前兩者打造出實際運作的「倉庫」。

5.2.5 第 20–25 行：描述 App 的畫面

```
1 var body: some Scene {
2     WindowGroup {
3         ContentView()
4     }
5     .modelContainer(sharedModelContainer)
6 }
```

- `var body: some Scene`：`App` 協定要求提供一個 `body`，用來描述 App 的內容。`some Scene` 表示「這裡會回傳某一種場景」。
- `WindowGroup { ... }`：代表 App 的一個視窗。在 iPhone 上通常就是整個螢幕。
- `ContentView()`：建立主畫面（第六章會詳解）。這就是 App 啟動後使用者**第一眼看到**的畫面。
- `.modelContainer(sharedModelContainer)`：**關鍵的一行**。把前面建好的資料庫「注入」整個畫面階層。這之後，所有子畫面（ContentView 及其底下的畫面）都能透過 `@Environment` 取得這個資料庫來讀寫資料。

小提示

小結：這個檔案做了三件事：(1) 用 `@main` 宣告 App 起點；(2) 建立可永久儲存的資料庫；(3) 顯示 `ContentView` 作為第一個畫面，並把資料庫分享給所有畫面使用。

6 第五章　資料模型：Item.swift

這個檔案定義「一筆待辦事項」到底包含哪些資料。它是整個 App 的「資料藍圖」。

6.1 完整程式碼

```
1  import Foundation
2  import SwiftData
3
4  @Model
5  final class Item {
6      var subject: String
7      var itemDescription: String
8      var timestamp: Date
9
10     init(subject: String, itemDescription: String, timestamp: Date
           = Date()) {
11         self.subject = subject
12         self.itemDescription = itemDescription
13         self.timestamp = timestamp
14     }
15 }
```

6.2 逐行解說

6.2.1 第 1–2 行：引入工具

```
1  import Foundation
2  import SwiftData
```

`Foundation` 是最基礎的工具箱，提供 `String`、`Date` 等基本型別。`SwiftData` 提供 `@Model` 等儲存功能。

6.2.2 第 4 行：@Model 標記

```
1  @Model
```

這是**最重要的一行**。它告訴 SwiftData：「下面這個 class 是要被儲存的資料模型。」加上 `@Model` 後，SwiftData 會自動幫這個型別處理「存到磁碟、從磁碟讀出、變動追蹤」等繁瑣工作，我們完全不必自己寫資料庫程式。

6.2.3 第 5 行：宣告 Item 類別

```
1 final class Item {
```

- `class Item`：定義一個名為 Item 的類別（資料用 class，因為要被 SwiftData 追蹤與共用同一份資料）。
- `final`：表示「這個類別不允許被其他類別繼承」。能避免不必要的複雜度，也讓編譯器最佳化。

6.2.4 第 6–8 行：三個資料欄位（屬性）

```
1 var subject: String
2 var itemDescription: String
3 var timestamp: Date
```

這三行定義了「一筆待辦事項」擁有的三項資料，稱為**屬性（property）**：

- `subject: String`：待辦事項的**主旨**（標題），文字。
- `itemDescription: String`：待辦事項的**詳細描述**，文字。
- `timestamp: Date`：這筆事項的**建立時間**，日期型別。

> **⚠ 注意**
>
> 為什麼描述欄位叫 `itemDescription` 而不是 `description`？因為 Swift 裡 `description` 是一個**已被內建使用**的特殊名稱（用來描述物件）。若直接用 `description` 容易和系統行為衝突，所以這裡刻意改名為 `itemDescription` 以避開衝突。

6.2.5 第 10–14 行：初始化器 init

```
1 init(subject: String, itemDescription: String, timestamp: Date =
    Date()) {
2     self.subject = subject
3     self.itemDescription = itemDescription
4     self.timestamp = timestamp
5 }
```

`init` 是**初始化器（建構子）**：當我們要「製造一筆新的 Item」時，會呼叫它，並提供必要的初始值。

- 括號內是製造一筆 Item 所需的三項資料。

- `timestamp: Date = Date()` 中的 `= Date()` 是**預設值**：如果呼叫時沒有特別指定時間，就自動用「現在」當作建立時間。`Date()` 代表「此刻」。
- `self.subject = subject`：`self` 指「這個正在被製造的 Item 自己」。這行把外面傳進來的 `subject`（參數）存到自己的 `subject`（屬性）裡。因為兩者同名，用 `self.` 來區分「自己的屬性」與「傳入的參數」。

小提示

實際使用範例：在主畫面新增待辦時會這樣呼叫——`Item(subject: "買牛奶", itemDescription: "兩瓶")`。因為沒給 `timestamp`，它會自動用現在時間。

7 第六章　主畫面：ContentView.swift

這是 App 啟動後看到的主畫面：一份待辦清單，上方有「+」可以新增、可以左滑刪除、點任一筆可進入詳細頁。

7.1 完整程式碼

```
import SwiftUI
import SwiftData

struct ContentView: View {
    @Environment(\.modelContext) private var modelContext
    @Query(sort: \Item.timestamp, order: .reverse) private var
       items: [Item]
    @State private var isAddingItem = false

    var body: some View {
        NavigationSplitView {
            List {
                ForEach(items) { item in
                    NavigationLink {
                        ItemDetailView(item: item)
                    } label: {
                        VStack(alignment: .leading, spacing: 4) {
                            Text(item.subject)
                                .font(.headline)
                            if !item.itemDescription.isEmpty {
```

```
                                Text(item.itemDescription)
                                    .font(.subheadline)
                                    .foregroundStyle(.secondary)
                                    .lineLimit(2)
                            }
                        }
                    }
                }
                .onDelete(perform: deleteItems)
            }
            .navigationTitle("To-Do List")
            .toolbar {
                ToolbarItem(placement: .navigationBarTrailing) {
                    EditButton()
                }
                ToolbarItem {
                    Button {
                        isAddingItem = true
                    } label: {
                        Label("Add Item", systemImage: "plus")
                    }
                }
            }
            .sheet(isPresented: $isAddingItem) {
                AddItemView { subject, description in
                    addItem(subject: subject, itemDescription:
                        description)
                }
            }
        } detail: {
            Text("Select an item")
        }
    }

    private func addItem(subject: String, itemDescription: String)
        {
        withAnimation {
            let newItem = Item(subject: subject, itemDescription:
                itemDescription)
```

```
56              modelContext.insert(newItem)
57          }
58      }
59
60      private func deleteItems(offsets: IndexSet) {
61          withAnimation {
62              for index in offsets {
63                  modelContext.delete(items[index])
64              }
65          }
66      }
67  }
```

7.2 逐行解說

7.2.1 宣告畫面

```
1  struct ContentView: View {
```

定義一個名為 ContentView 的畫面。`: View` 代表「我是一個 SwiftUI 畫面」，遵循 `View` 協定就必須提供 `body`（畫面內容）。

7.2.2 三個重要的屬性

```
1  @Environment(\.modelContext) private var modelContext
2  @Query(sort: \Item.timestamp, order: .reverse) private var items: [
       Item]
3  @State private var isAddingItem = false
```

- `modelContext`：用 `@Environment` 從環境取得「資料庫的操作員」。還記得第四章把資料庫注入了環境嗎？這裡就是把它取出來用。新增、刪除資料都靠它。
- `items`：用 `@Query` 自動從資料庫**撈出所有 Item**。`[Item]` 表示「Item 的陣列（清單）」。
  - `sort: \.Item.timestamp`：依「建立時間」排序。
  - `order: .reverse`：反向排序，也就是**最新的排在最上面**。
  - `@Query` 的魔力：只要資料庫裡的資料有任何變動（新增/刪除/修改），`items` 會**自動更新**，畫面也跟著自動重畫——你完全不必手動刷新清單。

- `isAddingItem`：用 `@State` 宣告的開關，型別是 `Bool`，初始為 `false`。它記錄「現在是否正要顯示新增畫面」。設成 `true` 時，新增畫面就會彈出來。
- `private`：表示這些屬性只屬於這個畫面內部使用，外部看不到，有助於封裝與安全。

名詞解釋

`\.modelContext` 與 `\Item.timestamp` 中的反斜線是什麼？ 這叫 **KeyPath（鍵路徑）**，是一種「指向某個屬性」的寫法。`\Item.timestamp` 的意思就是「Item 的 timestamp 這個屬性」，用來告訴 `@Query` 要依哪個欄位排序。

7.2.3 畫面主體與導覽容器

```
var body: some View {
    NavigationSplitView {
        ...
    } detail: {
        Text("Select an item")
    }
}
```

- `NavigationSplitView`：一個「可以切換頁面」的容器，支援主從式版面。在 iPhone 上，它讓我們可以從清單「推進」到詳細頁，並提供返回功能。
- 第一個大括號是**主側（清單）**的內容；`detail:` 後面是**詳細側**的內容。在大畫面（iPad）上，右側預設顯示「Select an item」；在 iPhone 上，點清單項目會直接推進詳細頁。

7.2.4 清單與每一列

```
List {
    ForEach(items) { item in
        NavigationLink {
            ItemDetailView(item: item)
        } label: {
            VStack(alignment: .leading, spacing: 4) {
                Text(item.subject).font(.headline)
                if !item.itemDescription.isEmpty {
                    Text(item.itemDescription)
                        .font(.subheadline)
```

```
                            .foregroundStyle(.secondary)
                            .lineLimit(2)
                    }
                }
            }
        }
        .onDelete(perform: deleteItems)
}
```

- `List { ... }`：建立一個可滾動的清單。
- `ForEach(items) { item in ... }`：把 `items` 陣列裡的**每一筆**都拿出來，逐一產生一列。`item in` 代表「目前處理的這一筆叫做 item」。
- `NavigationLink`：把這一列變成「可點擊的連結」。點下去會**推進**到大括號裡指定的畫面——也就是 `ItemDetailView(item: item)`（把這一筆 item 交給詳細頁）。
- `label:` 後面是「這一列**顯示出來**的長相」：
  - `VStack(alignment: .leading, spacing: 4)`：垂直堆疊，靠左對齊，元素間距 4 點。
  - 第一行 `Text(item.subject).font(.headline)`：顯示主旨，字體用較大的標題樣式。
  - `if !item.itemDescription.isEmpty`：**條件判斷**——只有當描述「不是空的」時才顯示描述。`!` 是「非（不）」，`isEmpty` 是「是否為空」。
  - 描述文字使用較小字體、次要灰色（`.foregroundStyle(.secondary)`），最多顯示兩行（`.lineLimit(2)`）。
- `.onDelete(perform: deleteItems)`：為清單加上「左滑刪除」功能，刪除時呼叫 `deleteItems` 函式。

名詞解釋

`.font(.headline)` 這種「點 + 名稱」的寫法叫**修飾器（modifier）**。它像是「在原本的元件上再貼一張設定貼紙」，可以一個接一個串起來，例如先設字體、再設顏色、再設行數。

7.2.5 標題與工具列

```
.navigationTitle("To-Do List")
.toolbar {
    ToolbarItem(placement: .navigationBarTrailing) {
```

```
        EditButton()
    }
    ToolbarItem {
        Button {
            isAddingItem = true
        } label: {
            Label("Add Item", systemImage: "plus")
        }
    }
}
```

- `.navigationTitle("To-Do List")`：在畫面頂端顯示標題「To-Do List」。
- `.toolbar { ... }`：在導覽列上放按鈕。
- `EditButton()`：系統內建的「編輯」按鈕，點了會進入編輯模式，方便整理／刪除清單。
- 第二個 `ToolbarItem` 是**「＋」新增按鈕**：
  - `Button { isAddingItem = true }`：被點擊時，把開關 `isAddingItem` 設為 `true`。
  - 一旦 `isAddingItem` 變成 `true`，下面的 `.sheet` 就會偵測到並彈出新增畫面（這就是宣告式 UI 的威力：改變狀態，畫面自動反應）。
  - `Label("Add Item", systemImage: "plus")`：按鈕的長相——文字「Add Item」配上系統的「plus（加號）」圖示。`systemImage` 用的是 Apple 內建的 SF Symbols 圖示庫。

7.2.6 彈出新增畫面（sheet）

```
.sheet(isPresented: $isAddingItem) {
    AddItemView { subject, description in
        addItem(subject: subject, itemDescription: description)
    }
}
```

- `.sheet(isPresented: $isAddingItem)`：當 `isAddingItem` 為 `true` 時，從畫面底部**彈出**一個半頁卡片（sheet）。
- `$isAddingItem` 前面的 `$` 代表傳入的是「**雙向綁定**」而非單純的值。這很重要：使用者把新增畫面往下滑關閉時，sheet 會**自動**把 `isAddingItem` 改回 `false`，狀態與畫面保持同步。
- `AddItemView { subject, description in ... }`：建立新增畫面，並提供一段「儲

存時要做的事」的閉包。當使用者在新增畫面按下 Save，它會回頭呼叫這段閉包，把使用者輸入的 `subject` 與 `description` 傳回來，我們再交給 `addItem` 真正存進資料庫。

7.2.7 新增資料的函式

```
private func addItem(subject: String, itemDescription: String) {
    withAnimation {
        let newItem = Item(subject: subject, itemDescription:
            itemDescription)
        modelContext.insert(newItem)
    }
}
```

- `withAnimation { ... }`：把區塊內造成的畫面變化**加上動畫**，新項目會平順地滑入清單，而非突然出現。
- `let newItem = Item(...)`：依第五章的藍圖，製造一筆新的待辦事項。
- `modelContext.insert(newItem)`：把新事項**插入資料庫**。一旦插入，`@Query` 偵測到資料變動，`items` 自動更新，清單立刻顯示新項目。

7.2.8 刪除資料的函式

```
private func deleteItems(offsets: IndexSet) {
    withAnimation {
        for index in offsets {
            modelContext.delete(items[index])
        }
    }
}
```

- `offsets: IndexSet`：使用者左滑刪除時，系統會告訴我們「被刪掉的是第幾列(可能多列)」，這組位置就是 `offsets`。
- `for index in offsets { ... }`：**迴圈**，把每一個要刪的位置都跑過一次。
- `modelContext.delete(items[index])`：`items[index]` 取出第 `index` 筆，交給資料庫刪除。刪除後 `@Query` 同樣會自動更新清單。

小提示

本章關鍵心法：你會發現我們從頭到尾**沒有寫過「請刷新清單」的程式**。我們只是「改變資料」（插入、刪除），畫面就自動跟上。這正是 SwiftUI + SwiftData 的核心：**資料是唯一真相來源，畫面是資料的倒影**。

8 第七章　新增畫面：AddItemView.swift

點下「+」後彈出的畫面。讓使用者輸入主旨與描述，按 Save 儲存、按 Cancel 取消。

8.1 完整程式碼

```
import SwiftUI

struct AddItemView: View {
    @Environment(\.dismiss) private var dismiss

    @State private var subject = ""
    @State private var itemDescription = ""

    let onSave: (String, String) -> Void

    var body: some View {
        NavigationStack {
            Form {
                Section("Subject") {
                    TextField("Enter subject", text: $subject)
                }
                Section("Description") {
                    TextField("Enter description", text:
                        $itemDescription, axis: .vertical)
                        .lineLimit(3...6)
                }
            }
            .navigationTitle("New To-Do")
            .navigationBarTitleDisplayMode(.inline)
            .toolbar {
                ToolbarItem(placement: .cancellationAction) {
```

```
26                      Button("Cancel") {
27                          dismiss()
28                      }
29                  }
30                  ToolbarItem(placement: .confirmationAction) {
31                      Button("Save") {
32                          onSave(subject.trimmingCharacters(in: .
                               whitespacesAndNewlines),
33                                 itemDescription.trimmingCharacters(
                                       in: .whitespacesAndNewlines))
34                          dismiss()
35                      }
36                      .disabled(subject.trimmingCharacters(in: .
                           whitespacesAndNewlines).isEmpty)
37                  }
38              }
39          }
40      }
41  }
42
43  #Preview {
44      AddItemView { _, _ in }
45  }
```

8.2 逐行解說

8.2.1 屬性宣告

```
1  @Environment(\.dismiss) private var dismiss
2  @State private var subject = ""
3  @State private var itemDescription = ""
4  let onSave: (String, String) -> Void
```

- `@Environment(\.dismiss) ... dismiss`：從環境取得「**關閉這個畫面**」的能力。之後呼叫 `dismiss()` 就能把這個彈出畫面收起來。
- `subject` / `itemDescription`：兩個 `@State`，初始為空字串 `""`。它們即時記錄使用者在輸入框打了什麼。使用者每打一個字，這裡就更新。
- `let onSave: (String, String) -> Void`：這是**最關鍵的設計**。它宣告一個叫 `onSave`

的閉包屬性，型別是「接收兩個 String、不回傳東西（`Void`）」。

- 白話說：「我（新增畫面）不知道資料要怎麼存，那是別人的事。我只負責收集輸入，按 Save 時把資料**交給外面**（透過 `onSave`）。」
- 在第六章，`ContentView` 正是提供了這段 `onSave`，內容是呼叫 `addItem` 存進資料庫。
- 這種「把後續動作交給呼叫者」的設計叫**回呼（callback）**，能讓畫面更單純、更好重複使用。

8.2.2 表單與輸入框

```
NavigationStack {
    Form {
        Section("Subject") {
            TextField("Enter subject", text: $subject)
        }
        Section("Description") {
            TextField("Enter description", text: $itemDescription,
               axis: .vertical)
                .lineLimit(3...6)
        }
    }
```

- `NavigationStack`：提供頂部導覽列（標題與左右按鈕）的容器。
- `Form`：專門用來做「設定／輸入表單」的容器，會自動套用分組、底色等系統樣式。
- `Section("Subject")`：表單裡的一個「分區」，標題為 Subject。
- `TextField("Enter subject", text: $subject)`：文字輸入框。
  - 第一個參數"Enter subject" 是**提示文字（placeholder）**，輸入框空白時顯示的灰字。
  - `text: $subject` 把輸入框和 `subject` 狀態做**雙向綁定**（注意 `$`）：使用者打字會更新 `subject`，反過來 `subject` 改變也會反映到框內。
- 描述輸入框多了 `axis: .vertical` 與 `.lineLimit(3...6)`：允許**多行輸入**，高度介於 3 到 6 行之間，內容多時自動長高。

8.2.3 標題與工具列按鈕

```
.navigationTitle("New To-Do")
```

```
.navigationBarTitleDisplayMode(.inline)
.toolbar {
    ToolbarItem(placement: .cancellationAction) {
        Button("Cancel") { dismiss() }
    }
    ToolbarItem(placement: .confirmationAction) {
        Button("Save") {
            onSave(subject.trimmingCharacters(in: .
                whitespacesAndNewlines),
                    itemDescription.trimmingCharacters(in: .
                        whitespacesAndNewlines))
            dismiss()
        }
        .disabled(subject.trimmingCharacters(in: .
            whitespacesAndNewlines).isEmpty)
    }
}
```

- `.navigationTitle("New To-Do")`：標題顯示「New To-Do」。
- `.navigationBarTitleDisplayMode(.inline)`：標題用「小字、置中」的樣式（而非大標題）。
- **Cancel 按鈕**：放在 `.cancellationAction`（慣例上是左側）。點了只呼叫 `dismiss()` 關閉畫面，**不做任何儲存**，使用者輸入的內容就被丟棄。
- **Save 按鈕**：放在 `.confirmationAction`（慣例上是右側）。點了會：
  1. 呼叫 `onSave(...)` 把整理後的主旨與描述交給外面（即 `ContentView` 的 `addItem`）。
  2. 接著 `dismiss()` 關閉新增畫面，回到清單。
- `.trimmingCharacters(in: .whitespacesAndNewlines)`：把使用者輸入**頭尾的空白與換行去掉**。例如「買牛奶 」會變成「買牛奶」，避免存進一堆無意義的空白。
- `.disabled(... .isEmpty)`：當主旨「去掉空白後仍是空的」時，把 Save 按鈕**變成灰色不可點**。這確保使用者**至少要輸入主旨**才能儲存——一種基本的輸入驗證。

8.2.4 預覽

```
#Preview {
    AddItemView { _, _ in }
}
```

`#Preview` 是給 Xcode **即時預覽**用的。在 Xcode 右側的畫布（Canvas）上，你不必執行整個 App 就能看到這個畫面的長相。大括號內提供一個「空的 onSave」（`{ _, _ in }` 表示「兩個參數都用不到，什麼也不做」），只是為了讓預覽能建立這個畫面。

小提示

資料流回顧：使用者輸入 → 存在 `@State`（subject/itemDescription）→ 按 Save → 透過 `onSave` 回呼把資料交給 `ContentView` → `ContentView` 呼叫 `addItem` 存入資料庫 → `@Query` 自動更新 → 清單出現新項目。

9 第八章　詳細／編輯畫面：ItemDetailView.swift

點清單中任一筆待辦後進入的畫面。可以檢視與**編輯**主旨、描述，按 Save 儲存變更並返回、按 Cancel 放棄變更並返回。

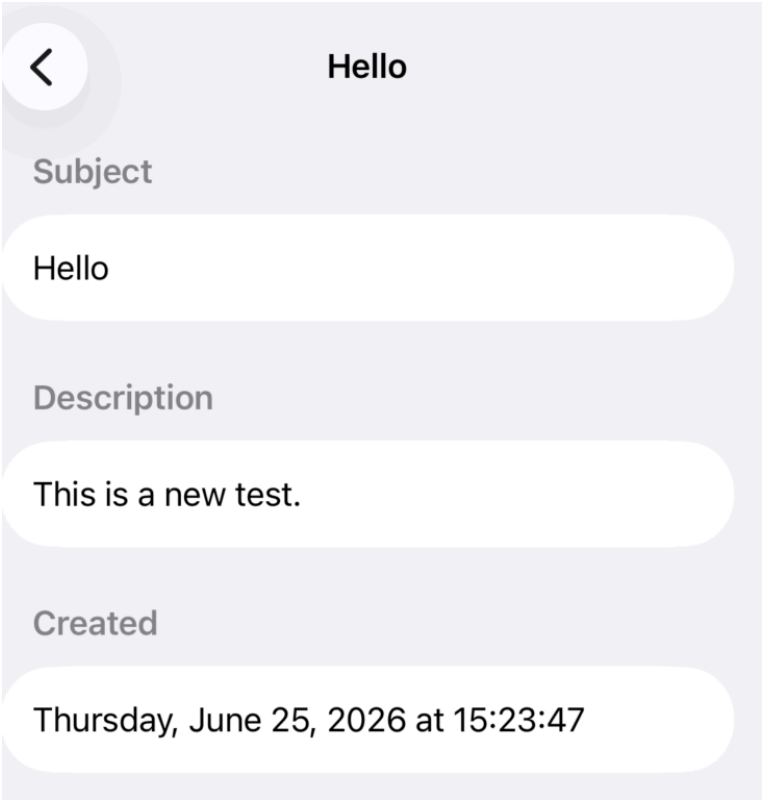

图 1: 詳細／編輯畫面：顯示主旨、描述與建立時間，頂端有 Cancel 與 Save。

9.1 完整程式碼

```
1 import SwiftUI
2
```

```
3  struct ItemDetailView: View {
4      @Environment(\.dismiss) private var dismiss
5      @Bindable var item: Item
6
7      @State private var subject = ""
8      @State private var itemDescription = ""
9
10     private var trimmedSubject: String {
11         subject.trimmingCharacters(in: .whitespacesAndNewlines)
12     }
13
14     private var trimmedDescription: String {
15         itemDescription.trimmingCharacters(in: .
               whitespacesAndNewlines)
16     }
17
18     private var canSave: Bool {
19         !trimmedSubject.isEmpty
20     }
21
22     var body: some View {
23         Form {
24             Section("Subject") {
25                 TextField("Enter subject", text: $subject)
26             }
27             Section("Description") {
28                 TextField("Enter description", text:
                       $itemDescription, axis: .vertical)
29                     .lineLimit(3...6)
30             }
31             Section("Created") {
32                 Text(item.timestamp, format: Date.FormatStyle(date:
                       .complete, time: .standard))
33             }
34         }
35         .navigationTitle(trimmedSubject.isEmpty ? item.subject :
               trimmedSubject)
36         .navigationBarTitleDisplayMode(.inline)
37         .toolbar {
```

```
38              ToolbarItem(placement: .cancellationAction) {
39                  Button("Cancel") {
40                      dismiss()
41                  }
42              }
43              ToolbarItem(placement: .confirmationAction) {
44                  Button("Save") {
45                      saveChanges()
46                      dismiss()
47                  }
48                  .disabled(!canSave)
49              }
50          }
51          .onAppear {
52              loadFromItem()
53          }
54      }
55
56      private func loadFromItem() {
57          subject = item.subject
58          itemDescription = item.itemDescription
59      }
60
61      private func saveChanges() {
62          item.subject = trimmedSubject
63          item.itemDescription = trimmedDescription
64      }
65  }
```

9.2 逐行解說

9.2.1 屬性宣告

```
1  @Environment(\.dismiss) private var dismiss
2  @Bindable var item: Item
3  @State private var subject = ""
4  @State private var itemDescription = ""
```

- `dismiss`：同前，用來返回上一頁的能力。
- `@Bindable var item: Item`：這是本檔的重點。`item` 是**從清單傳進來的那一筆待辦**。`@Bindable` 讓我們能對這個 SwiftData 物件做**雙向綁定**並直接修改它的屬性；一旦修改，變更會被 SwiftData 自動儲存。
- `subject` / `itemDescription`（`@State`）：**草稿暫存區**。注意：我們**沒有**直接綁定到 `item` 的欄位，而是先複製到這兩個草稿狀態。使用者編輯的是草稿，按 Save 才寫回 `item`。這樣「Cancel 放棄變更」才有意義。

名詞解釋

為什麼要用草稿（@State）而不是直接改 item？ 如果直接綁定 `item` 的欄位，使用者每打一個字都會立刻寫進資料庫，按 Cancel 也救不回來。改用草稿後：編輯只動草稿 → Save 才寫回 → Cancel 直接丟棄草稿、原資料毫髮無傷。這就是「可取消的編輯」標準作法。

9.2.2 計算屬性 computed property

```
private var trimmedSubject: String {
    subject.trimmingCharacters(in: .whitespacesAndNewlines)
}
private var trimmedDescription: String {
    itemDescription.trimmingCharacters(in: .whitespacesAndNewlines)
}
private var canSave: Bool {
    !trimmedSubject.isEmpty
}
```

這三個是**計算屬性**：它們不儲存固定值，而是**每次被讀取時即時算出**結果。

- `trimmedSubject`：隨時回傳「去掉頭尾空白後的主旨」。
- `trimmedDescription`：同理，去空白後的描述。
- `canSave`：回傳「現在是否可以儲存」——只要去空白後的主旨**不是空的**（`!...isEmpty`）就為 `true`。Save 按鈕會依這個值決定能不能按。

9.2.3 表單三個分區

```
Form {
    Section("Subject") {
        TextField("Enter subject", text: $subject)
    }
```

```
    Section("Description") {
        TextField("Enter description", text: $itemDescription, axis
            : .vertical)
            .lineLimit(3...6)
    }
    Section("Created") {
        Text(item.timestamp, format: Date.FormatStyle(date: .
            complete, time: .standard))
    }
}
```

- 前兩區是可編輯的主旨與描述輸入框，綁定到**草稿** `subject` / `itemDescription`。
- 第三區「Created」顯示建立時間，這是**唯讀**的（用 `Text` 而非 `TextField`）。
- `Date.FormatStyle(date: .complete, time: .standard)`：把日期格式化成「完整日期(含星期) + 標準時間」，例如圖中的 Thursday, June 25, 2026 at 15:23:47。

9.2.4 動態標題

```
.navigationTitle(trimmedSubject.isEmpty ? item.subject :
    trimmedSubject)
```

這裡用了**三元運算子** `條件? A : B`，意思是「如果條件成立就用 A，否則用 B」。

- 若草稿主旨是空的 → 顯示原本的 `item.subject`。
- 否則 → 顯示使用者正在編輯的草稿主旨。

效果是：標題會**即時**跟著使用者輸入的主旨變化。

9.2.5 工具列：Save 與 Cancel

```
ToolbarItem(placement: .cancellationAction) {
    Button("Cancel") { dismiss() }
}
ToolbarItem(placement: .confirmationAction) {
    Button("Save") {
        saveChanges()
        dismiss()
    }
    .disabled(!canSave)
}
```

- **Cancel**：只呼叫 `dismiss()` 返回。因為我們從頭到尾只改草稿、沒碰 `item`，所以「放棄變更」就是直接離開，原資料保持不變。
- **Save**：先 `saveChanges()` 把草稿寫回 `item`，再 `dismiss()` 返回。
- `.disabled(!canSave)`：主旨為空時，Save 不可點，避免存出沒有主旨的待辦。

9.2.6 畫面出現時載入資料

```
.onAppear {
    loadFromItem()
}
```

`.onAppear` 是「當這個畫面**出現在螢幕上時**」要做的事。這裡呼叫 `loadFromItem()`，把 `item` 目前的內容複製進草稿，使輸入框一打開就顯示既有資料，讓使用者在原內容上修改。

9.2.7 兩個輔助函式

```
private func loadFromItem() {
    subject = item.subject
    itemDescription = item.itemDescription
}
private func saveChanges() {
    item.subject = trimmedSubject
    item.itemDescription = trimmedDescription
}
```

- `loadFromItem()`：把真實資料 → 複製到草稿（進場時用）。
- `saveChanges()`：把草稿（去空白後）→ 寫回真實資料（按 Save 時用）。一旦寫回，SwiftData 會自動把變更存到磁碟，清單也會自動反映新主旨。

小提示

Save / Cancel 的本質差異：兩者都會 `dismiss()` 返回，差別只在 Save **多做了** `saveChanges()`。理解「編輯草稿、確認才寫回」這個模式，你就掌握了幾乎所有 App 編輯頁的設計精髓。

10 第九章　測試檔案簡介

專案附帶了測試檔。測試是「自動檢查 App 有沒有壞掉」的程式，初學階段可先了解概念即可。

10.1 單元測試 To_Do_List_DemoTests.swift

單元測試用來檢查「某一小段邏輯是否正確」。本專案目前只有一個空樣板：

```
import Testing
@testable import To_Do_List_Demo

struct To_Do_List_DemoTests {
    @Test func example() async throws {
        // 在這裡撰寫測試，用 #expect(...) 檢查預期結果
    }
}
```

- `import Testing`：Apple 新版的測試框架。
- `@testable import ...`：把我們的 App 拿進來，以便測試其內部。
- `@Test`：標記「這是一個測試項目」。將來可在裡面用 `#expect(...)` 驗證，例如「新增一筆後，數量應為 1」。

10.2 介面測試 To_Do_List_DemoUITests.swift

UI 測試會**真的啟動 App**，模擬使用者點按、輸入，檢查畫面行為。

```
func testExample() throws {
    let app = XCUIApplication()
    app.launch()          // 啟動 App
    // 之後可加入：尋找按鈕、點擊、輸入文字、驗證結果
}
```

另外 `testLaunchPerformance` 會測量 App 啟動速度，`To_Do_List_DemoUITestsLaunchTests` 則會在啟動後**自動截圖**保存，方便檢視啟動畫面。

小提示

測試雖然初學可略過，但養成「寫一點測試」的習慣，能在你日後修改程式時，自動幫你抓出「改壞了的地方」，是專業開發的重要一環。

11 第十章　完整程式流程：把所有檔案串起來

讀完每個檔案後，這一章把它們**連成一條完整的故事線**，讓你看清資料與畫面如何協同運作。

11.1 流程一：App 啟動

1. 使用者點下 App 圖示，iOS 找到 `@main` 標記的 `To_Do_List_DemoApp`。
2. App 建立 `ModelContainer`（資料倉庫），準備好 `Item` 的儲存空間。
3. App 顯示 `WindowGroup` 裡的 `ContentView`，並用 `.modelContainer(...)` 把資料庫分享給所有畫面。
4. `ContentView` 的 `@Query` 自動從資料庫撈出所有待辦，依時間**由新到舊**排好，顯示成清單。

11.2 流程二：新增一筆待辦

1. 使用者點工具列的「+」，按鈕把 `isAddingItem` 設為 `true`。
2. `.sheet` 偵測到狀態變化，從底部彈出 `AddItemView`。
3. 使用者在主旨、描述輸入框打字，內容即時存進 `AddItemView` 的 `@State`。主旨為空時 Save 為灰色不可按。
4. 使用者按 `Save`：`AddItemView` 透過 `onSave` 回呼，把（去空白後的）主旨與描述交回 `ContentView`，然後關閉自己。
5. `ContentView` 的 `addItem` 製造一筆新 `Item`，呼叫 `modelContext.insert(...)` 存進資料庫。
6. 資料庫一變，`@Query` 自動更新 `items`，清單**帶動畫**出現新項目（排在最上方，因為最新）。

11.3 流程三：檢視與編輯一筆待辦

1. 使用者點清單中某一列，`NavigationLink` 推進到 `ItemDetailView`，並把該筆 `item` 傳進去。
2. `.onAppear` 觸發 `loadFromItem()`，把 `item` 內容複製到草稿狀態，輸入框顯示既有資料。
3. 使用者修改主旨／描述，改的是**草稿**；標題即時跟著主旨變動。
4. 若按 `Cancel`：直接 `dismiss()` 返回，草稿丟棄，原資料不變。
5. 若按 `Save`：`saveChanges()` 把草稿寫回 `item`，SwiftData 自動存檔，再 `dismiss()` 返回；清單因 `@Query` 自動更新而顯示新主旨。

11.4 流程四：刪除一筆待辦

1. 使用者在清單左滑某列（或進入 EditButton 編輯模式）。
2. 觸發 `.onDelete`，呼叫 `deleteItems(offsets:)`。
3. 函式依位置從資料庫 `modelContext.delete(...)` 刪除對應的 `Item`。
4. `@Query` 自動更新，該列**帶動畫**消失。

11.5 一張圖看懂資料流

使用者操作 ⟶ **改變狀態 / 資料**（@State、modelContext）

↓

SwiftData 儲存變動 ⟶ **@Query 偵測到資料改變**

↓

SwiftUI 自動重畫相關畫面（清單、標題等自動更新）

小提示

整個 App 的靈魂：你（開發者）只負責「描述畫面」和「改變資料」；「畫面如何隨資料更新」交給 SwiftUI 自動處理。掌握這個分工，你就真正理解了現代 iOS 開發。

12 第十一章　在 Xcode 中實際操作

理解程式後，這一章帶你實際把 App 跑起來。

12.1 開啟專案

1. 在 Finder 中找到 `To-Do-List-Demo.xcodeproj`，**雙擊**開啟，Xcode 會載入整個專案。
2. 左側是**導覽器（Navigator）**，列出所有檔案；中間是**編輯區**；右側可顯示**預覽畫布（Canvas）**。

12.2 用模擬器執行 App

1. 在 Xcode 上方中央，選擇一個**模擬器**（例如 iPhone 17）。
2. 按下左上角的**執行鈕**（三角形 ▶），或按快捷鍵 `Cmd + R`。
3. Xcode 會**編譯**程式（把 Swift 翻譯成手機看得懂的指令），然後在模擬器啟動 App。

4. 你就能在模擬器中點「+」新增、點項目編輯、左滑刪除，親手驗證前面學到的每個流程。

名詞解釋

模擬器（Simulator）：在 Mac 上「假裝成一支 iPhone」的程式。不用真的手機就能測試 App，非常方便。

12.3 使用即時預覽 Canvas

打開任一畫面檔（如 `AddItemView.swift`），右側 Canvas 會顯示 `#Preview` 的即時畫面。改程式碼，預覽**立即更新**，不必每次都啟動整個 App，大幅加快開發速度。

12.4 基本除錯

- **紅色錯誤**：程式有語法錯誤無法編譯。點紅點看說明，通常會提示哪一行、缺了什麼。
- **黃色警告**：程式可以跑，但有可改進之處（例如宣告了沒用到的變數）。
- `print(...)`：在程式中插入 `print("到這裡了")`，執行時會在 Xcode 底部的**主控台（Console）**印出訊息，幫你了解程式跑到哪、變數是什麼值。
- **中斷點（Breakpoint）**：點程式行號旁可設藍色中斷點，執行到該行會暫停，讓你逐步檢查。

12.5 資料模型改變後的注意事項

注意

如果你修改了 `Item.swift` 的欄位(例如本專案從只有 `timestamp` 加上了 `subject`、`itemDescription`），舊的資料庫結構會與新模型**不相容**，可能導致 App 啟動就崩潰。最簡單的解法：在模擬器或手機上**把 App 刪除後重新安裝**，讓 SwiftData 用新結構重建資料庫。

13 第十二章　動手練習與延伸挑戰

學程式最有效的方法是**動手改**。以下練習由易到難，建議逐題嘗試。

13.1 初級練習

1. **改文字**：把主畫面標題從"To-Do List" 改成「我的待辦清單」。(提示：`.navigationTitle`)

2. **改圖示**：把新增按鈕的圖示從 `"plus"` 換成 `"plus.circle.fill"`。（提示：`systemImage`）
3. **改排序**：把清單改成「由舊到新」排序。（提示：把 `.reverse` 拿掉或改 `.forward`）

13.2 中級練習

1. **顯示時間**：在主畫面每一列的描述下方，加一行小字顯示建立時間。（提示：仿照 `ItemDetailView` 的 `Text(item.timestamp, format: ...)`）
2. **描述也要驗證**：讓 Save 在「主旨**或**描述任一為空」時都不可按。
3. **字數限制**：主旨超過 50 字時，在輸入框下方顯示紅色提醒文字。

13.3 進階挑戰

1. **完成狀態**：在 `Item` 加一個 `isDone: Bool` 欄位，清單列加上可勾選的圓圈，完成的項目顯示刪除線。
2. **搜尋功能**：在清單上方加入搜尋框，依主旨過濾待辦。（提示：`.searchable`）
3. **分類標籤**：為待辦加入分類（工作／生活／購物），並可依分類篩選。

小提示

每完成一個練習，就用 `Cmd + R` 跑跑看。出錯不要怕——讀錯誤訊息、修正、再試，這正是每個工程師每天在做的事。

14 附錄　名詞對照與速查表

14.1 核心名詞對照表

名詞	白話解釋
Xcode	Apple 的官方開發工具，用來寫程式、設計畫面、執行測試。
Swift	開發 iOS App 的程式語言。
SwiftUI	用程式碼宣告畫面長相的框架。
SwiftData	負責把資料永久儲存到裝置的框架。
View（畫面）	SwiftUI 中代表「一塊畫面」的元件，通常是 struct。
Modifier（修飾器）	以「. 名稱 (...)」串接、用來調整元件外觀或行為的設定。
@State	畫面自己擁有的狀態；改變時自動重畫。
@Binding / @Bindable	對「別人給的資料」做雙向讀寫的綁定。

名詞	白話解釋
@Environment	從系統環境取得現成能力或物件（如資料庫、dismiss）。
@Query	自動撈出 SwiftData 資料，且資料變動時畫面自動更新。
@Model	標記一個 class 為可被 SwiftData 儲存的資料模型。
ModelContainer	資料的「倉庫」，保管所有被儲存的物件。
ModelContext	對資料庫進行新增、刪除、修改的「操作員」。
Closure（閉包）	一段沒有名字、可被傳遞的程式碼，常用 {} 包住。
Callback（回呼）	把「之後要做的事」交給呼叫者處理的設計（如 onSave）。
Optional	可能有值、也可能沒有（nil）的資料，型別後加「?」。
struct / class	把資料與功能打包的藍圖；畫面用 struct，儲存資料用 class。
init（初始化器）	製造一個新物件時呼叫、用來設定初始值的函式。
KeyPath	以「\型別. 屬性」指向某個屬性的寫法。
Simulator（模擬器）	在 Mac 上模擬 iPhone 來測試 App 的工具。
Canvas（畫布）	Xcode 右側的即時預覽區，由 #Preview 驅動。

14.2 本專案修飾器速查

修飾器	作用
`.font(.headline)`	設定文字字體大小／樣式。
`.foregroundStyle(.secondary)`	設定文字顏色（次要灰）。
`.lineLimit(2)` / `(3...6)`	限制顯示行數／可變高度範圍。
`.navigationTitle(...)`	設定導覽列標題。
`.navigationBarTitleDisplayMode(.inline)`	標題用小字置中樣式。
`.toolbar { ... }`	在導覽列放置按鈕。
`.sheet(isPresented:)`	依狀態從底部彈出畫面。
`.onDelete(perform:)`	為清單加上滑動刪除。
`.onAppear { ... }`	畫面出現時執行某段程式。
`.disabled(...)`	在條件成立時讓控制項變灰不可用。
`.modelContainer(...)`	將 SwiftData 資料庫注入畫面階層。

恭喜你讀完整份教材！

你已經完整走過一個 iOS App 的資料模型、五個程式檔、以及新增／檢視／編輯／刪除的完整流程。

接下來最重要的事只有一件：**打開 Xcode，動手改、動手玩。**