Building Adaptable SwiftUI Applications for Multiple Platforms

fatbobman (东坡肘子) · Following

13 min read · Apr 26




This article is the sharing content of the author's participation in the "SwiftUI Technology Salon (Beijing Station)" event on April 20, 2023. It is based on memory and organization. For information about this event, please refer to the article "I Attended the SwiftUI Technology Salon in Beijing".

Photo by Daniel Romero on Unsplash

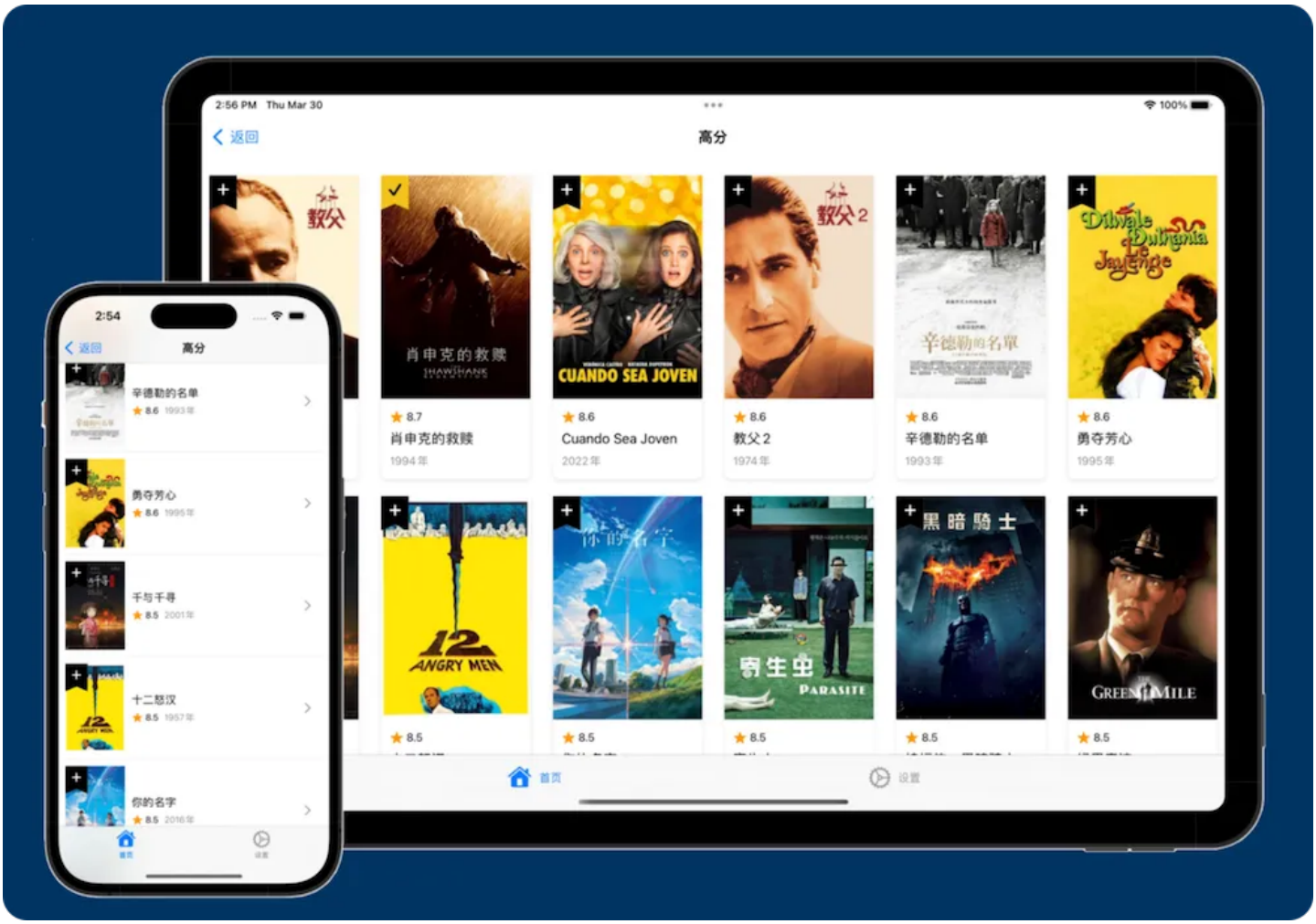
This event was conducted in the form of offline communication supplemented by live coding, so the focus and organization of the content will be significantly different from previous blog articles.

Introduction

Hello everyone, I am fatbobman（东坡肘子）. Today, I want to discuss with you the topic of creating SwiftUI applications that are adaptable to multiple platforms.

Movie Hunter

Let’s first look at an example, and then we’ll get into today’s topic.



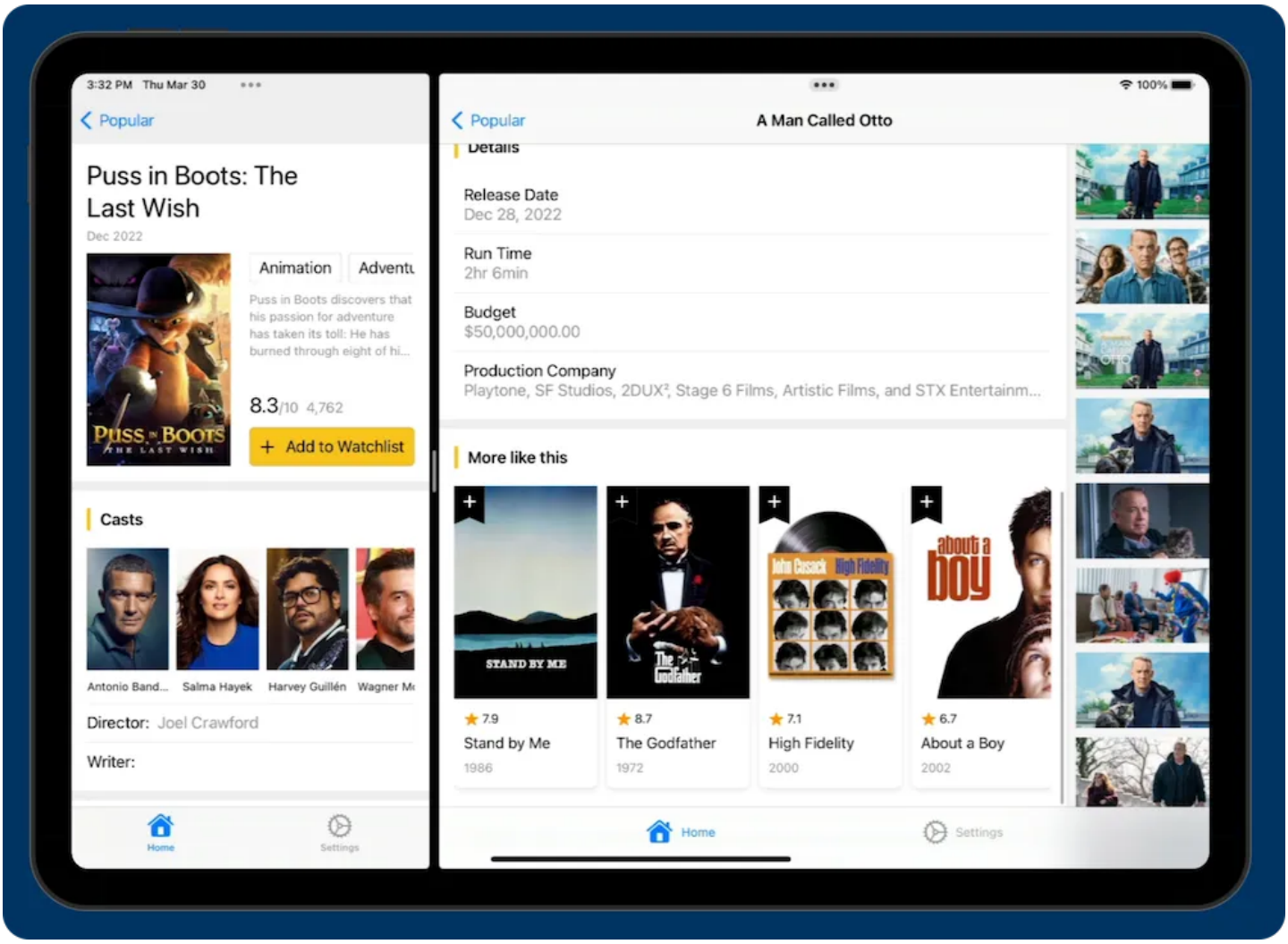
Here is a demo app I wrote for the topic of this communication — "Movie Hunter". It is 100% developed based on SwiftUI and currently supports three platforms: iPhone, iPad, and macOS.

Users can use it to browse movie information, including movies that are currently or soon to be released. They can also view movies they are interested in from various dimensions such as reputation, rating, popularity, and movie type.

"Movie Hunter" is a demo specially prepared for this communication

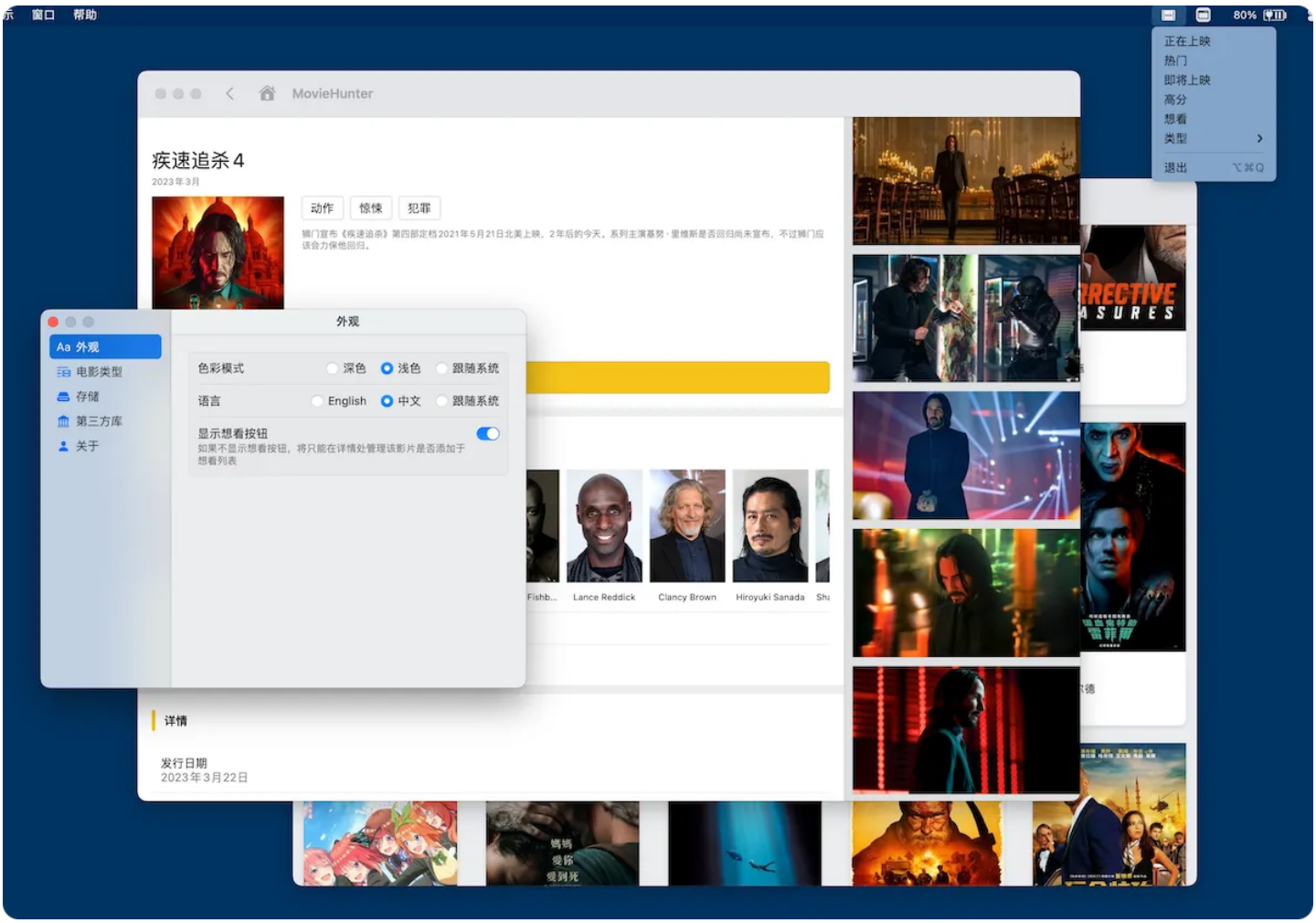
session, so only the necessary parts have been completed.

Compared to the iPhone version, the iPad version has made some adjustments to the layout to make use of the larger screen space, and also provides the ability to run multiple windows, allowing users to operate independently in each window.



The Mac version has undergone more adaptation to fit the macOS style, such as using a settings view that conforms to Mac specifications,

supporting pointer hovering response, menu bar icons, and supporting the creation of new windows and direct jumping to specific movie categories (based on data-driven WindowGroup).



Due to time constraints, we will not discuss the complete adaptation process of the application in this communication, but rather focus on two points that I personally consider to be important but easily overlooked.

Compatibility

Unlike the “Write once, run anywhere” approach advocated by many cross-

platform frameworks, Apple's approach to SwiftUI is "Learn once, apply anywhere."

In my understanding, SwiftUI is more like a programming philosophy. Once you have mastered it, you will have the ability to develop on different platforms within the Apple ecosystem for a long time. From another perspective, the code written in SwiftUI can run on most platforms, but some parts can only run on specific platforms, and often these parts have platform-specific features that best reflect the characteristics and advantages of the platform.

By setting certain compatibility restrictions, SwiftUI forces developers to consider the differences in platform characteristics and make targeted adjustments based on these differences when doing multi-platform adaptation.

However, if developers cannot understand this "limitation" of SwiftUI and prepare in advance, it may bring hidden dangers and unnecessary workload for multi-platform development in the future.

Take the iPad version of "Movie Hunter" as an example. In iPad, users can adjust the size of the application window. In order to make the layout more suitable for the current window state, we usually use environment values in the view to make judgments:

```
@Environment(\.horizontalSizeClass) var sizeClass
```

Adjust the layout dynamically based on the current state of sizeClass, whether it is compact or regular.

If your application is only intended to be adapted for iPadOS, this approach is entirely correct. However, for the “Movie Hunter” application, because it needs to be adapted for macOS later, using this method will cause problems.

Because the environment value of horizontalSizeClass cannot be used in macOS, UserInterfaceSizeClass is a concept unique to iOS (iPadOS). The more we rely on this environment value in the view code, the more adjustments we will have to make in the future.

```
@available(macOS, unavailable)
@available(tvOS, unavailable)
@available(watchOS, unavailable)
public var horizontalSizeClass: UserInterfaceSizeClass?
```

To avoid repetitive code adjustments when adapting to other platforms, we can create a custom environment variable that can be used for all platforms that need to be adapted, similar to horizontalSizeClass (via an environment variable).

First, create an enumeration type called DeviceStatus:

```
public enum DeviceStatus: String {
  case macOS
  case compact
  case regular
}
```

Within this enumeration type, we have added the macOS enum in addition to the two window states that appear in iOS.

Next, create an environment value of type DeviceStatus:

```
struct DeviceStatusKey: EnvironmentKey {
  #if os(macOS)
    static var defaultValue: DeviceStatus = .macOS
  #else
    static var defaultValue: DeviceStatus = .compact
  #endif
}

public extension EnvironmentValues {
  var deviceStatus: DeviceStatus {
    get { self[DeviceStatusKey.self] }
    set { self[DeviceStatusKey.self] = newValue }
  }
}
```

By using the conditional compilation statement `#if os(macOS)`, in macOS,

the environment value is set to the corresponding option. We also need to create a View Modifier so that we can stay informed about the current window state in iOS:

```
#if os(iOS)
  struct GetSizeClassModifier: ViewModifier {
    @Environment(\.horizontalSizeClass) private var sizeClass
    @State var currentSizeClass: DeviceStatus = .compact
    func body(content: Content) -> some View {
      content
        .task(id: sizeClass) {
          if let sizeClass {
            switch sizeClass {
            case .compact:
              currentSizeClass = .compact
            case .regular:
              currentSizeClass = .regular
            default:
              currentSizeClass = .compact
            }
          }
        }
        .environment(\.deviceStatus, currentSizeClass)
    }
  }
#endif
```

When the horizontalSizeClass of the view changes, update our custom deviceStatus in a timely manner. Finally, combine the code for different platforms through a View Extension:

```
public extension View {
```

```
    @ViewBuilder
    func setDeviceStatus() -> some View {
      self
      #if os(macOS)
      .environment(\.deviceStatus, .macOS)
      #else
      .modifier(GetSizeClassModifier())
      #endif
    }
  }
```

Apply setDeviceStatus to the root view:

```
ContentView:View {
    var body:some View {
      RootView()
          .setDeviceStatus()
    }
}
```

Now we have the ability to understand the current window status on iPhone, iPad, and macOS.

```
@Environment(\.deviceStatus) private var deviceStatus
```

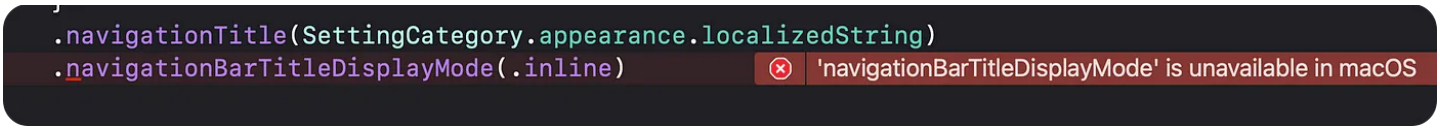
If we need to adapt to more platforms in the future, we only need to adjust the settings of the custom environment value. Although we still need to

adjust the view code, the amount of modification will be greatly reduced compared to horizontalSizeClass.

> setDeviceStatus is not only applicable to the root view, but at least it should be used at the widest view of the current application. This is because horizontalSizeClass only represents the horizontal size category of the current view, which means that if horizontalSizeClass is obtained in a view with a limited horizontal size (such as the Sidebar view of NavigationSplitView), regardless of the window size of the application, the sizeClass of the current view can only be compact. We create deviceStatus for the purpose of observing the window status of the current application, so it must be applied to the widest point.

In SwiftUI, in addition to environment values, another part that has more “restrictions” on platforms is the view’s modifier.

For example, when preparing to adapt the macOS version of “Movie Hunter” (the adaptation of the iPad version has been completed), when adding the macOS destination and compiling, you will find that Xcode has many errors like the following:



This is because certain View Modifiers are not supported on macOS. For the above error message, we can simply use conditional compilation

statements to shield it.

```
#if !os(macOS)
    .navigationBarTitleDisplayMode(.inline)
#endif
```

However, if there are many similar problems, we can adopt a one-time solution.

In “Movie Hunter”, navigationBarTitleDisplayMode is a frequently used Modifier. We can create a View Extension to handle compatibility issues on different platforms:

```
enum MyTitleDisplayMode {
    case automatic
    case inline
    case large
    #if !os(macOS)
        var titleDisplayMode: NavigationBarItem.TitleDisplayMode {
            switch self {
            case .automatic:
                return .automatic
            case .inline:
                return .inline
            case .large:
                return .large
            }
        }
    #endif
}
```

```
extension View {
    @ViewBuilder
    func safeNavigationBarTitleDisplayMode(_ displayMode: MyTitleDisplayMode) -
        #if os(iOS)
            navigationBarTitleDisplayMode(displayMode.titleDisplayMode)
        #else
            self
        #endif
    }
}
```

Use directly in the view:

```
.safeNavigationBarTitleDisplayMode(.inline)
```

Preparing some compatibility code in advance can greatly improve the efficiency of future development if you plan to introduce your app to more platforms. This approach not only solves cross-platform compatibility issues but also has other benefits:

- Improves the cleanliness of the code in views (reduces the use of conditional compilation statements)
- Improves the compatibility of SwiftUI between different versions

Of course, to create and use such code, the prerequisite is that developers must have a clear understanding of the “limitations” of SwiftUI in different platforms (the characteristics, advantages and handling methods of each

platform). Blindly using these compatibility codes may undermine the hard work of SwiftUI creators and prevent developers from accurately reflecting the characteristics of different platforms.

Source of Truth

After discussing compatibility, let's talk about another issue that is often overlooked in the early stages of building cross-platform applications: source of truth (data dependencies).

When we port "Movie Hunter" from iPhone to iPad or Mac, in addition to the larger available screen space, another significant change is that users can open multiple windows simultaneously and independently operate "Movie Hunter" in different windows.

However, if we directly run the iPhone version of "Movie Hunter" that has not been adapted to multiple screens on an iPad, we will find that although multiple "Movie Hunter" windows can be opened at the same time, all operations are synchronized, meaning that the operation performed in one window will also be reflected in another window. This loses the purpose of having multiple windows.

Why does this situation occur?

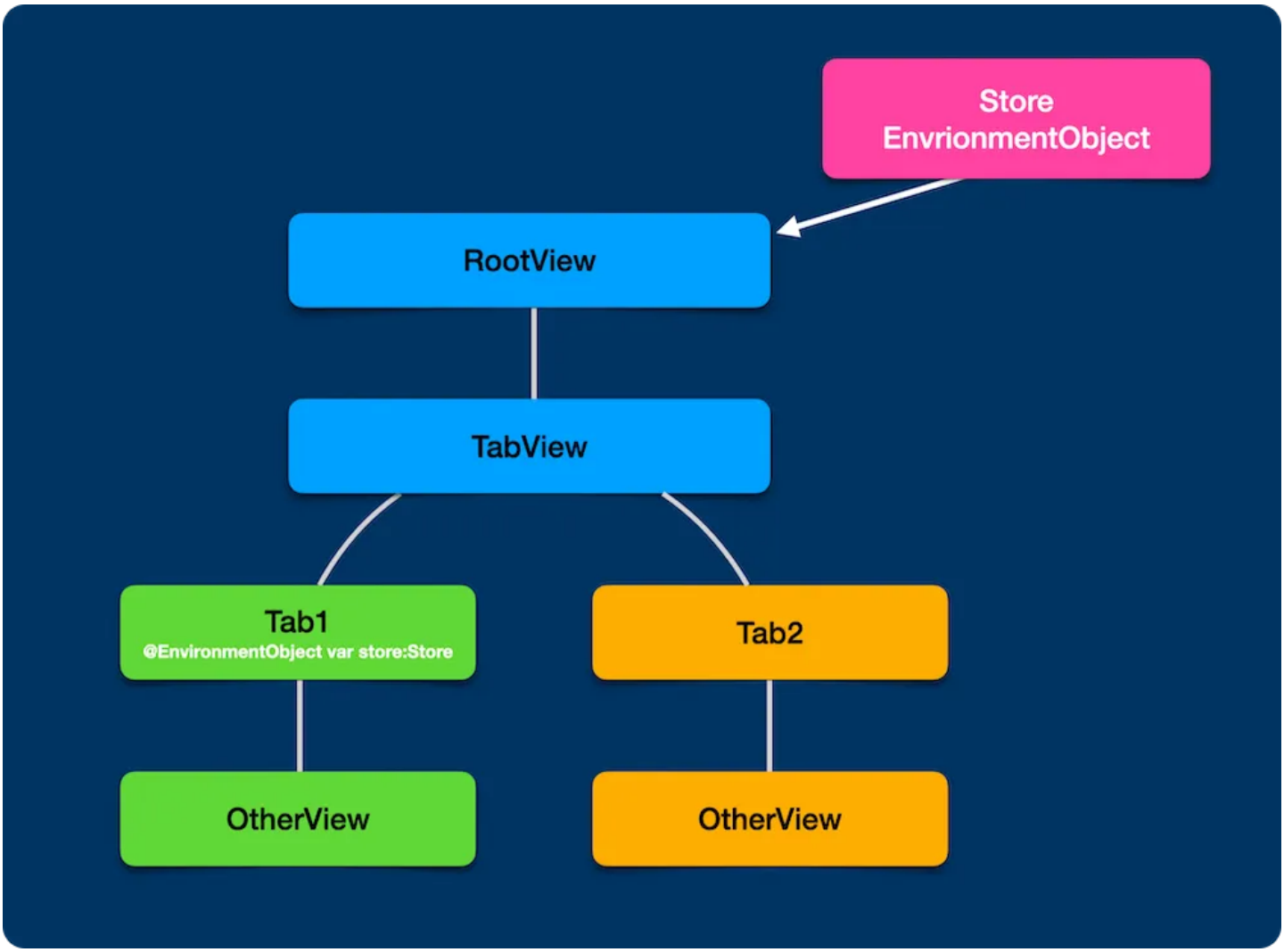
We all know that SwiftUI is a declarative framework. This means that developers can construct views declaratively, and scenes (corresponding to independent windows) or even the entire app are created based on declarative code.

```
@main
struct MovieHunterApp: App {
```

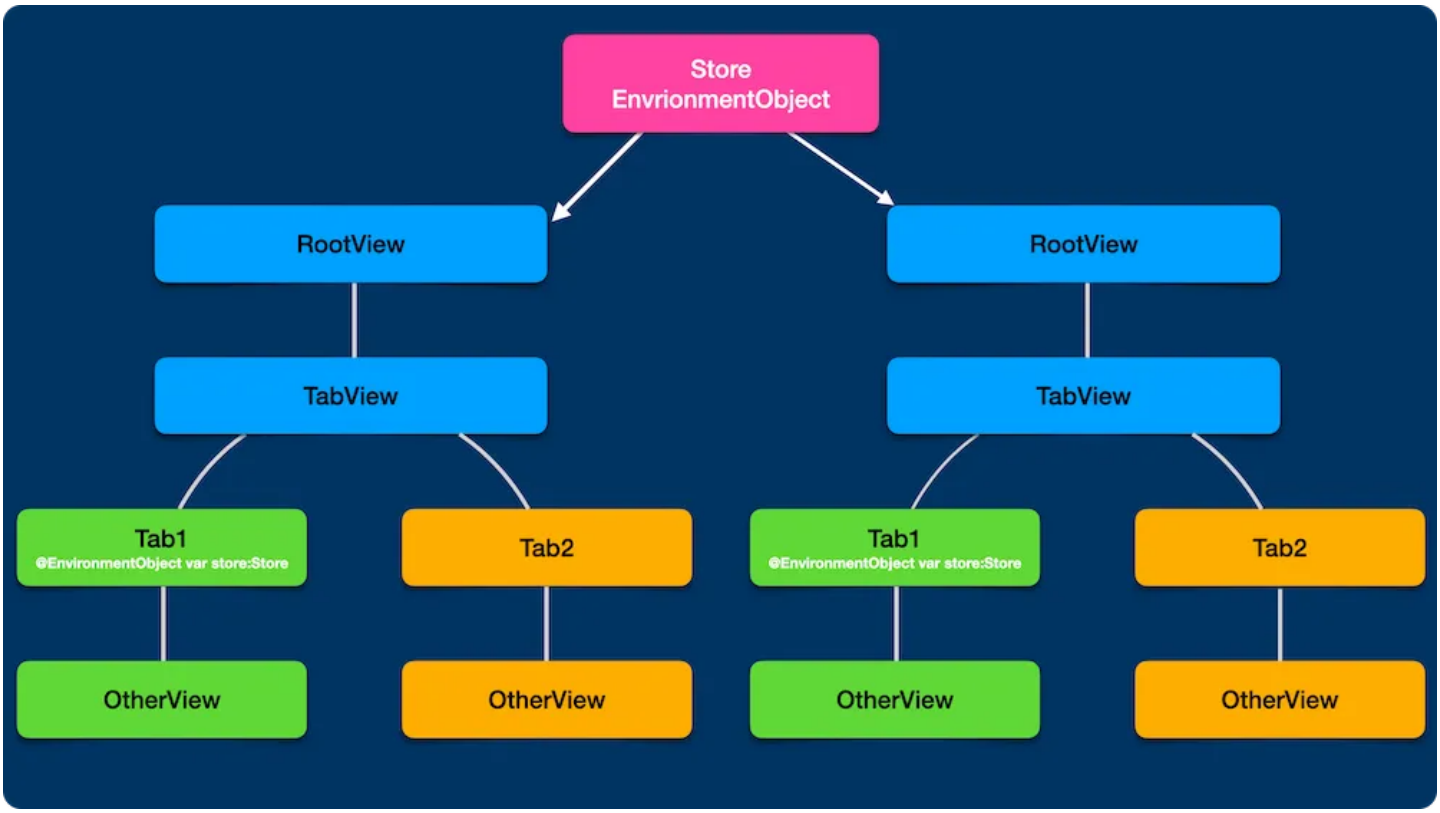
```
        @StateObject private var store = Store()
        var body: some Scene {
            WindowGroup {
                ContentView()
                    .environmentObject(store)
            }
        }
    }
```

Within the SwiftUI project template created in Xcode, WindowGroup corresponds to a scene declaration. As iPhone only supports single window mode, we typically don't pay too much attention to its existence. However, in systems that support multiple windows such as iPadOS and macOS, it represents that each time a new window is created (in macOS, created through the "New" option in the menu), it will strictly follow the declaration of WindowGroup.

In "MovieHunter", we created an instance of Store (a unit that saves application state and handles primary logic) at the App location, and injected it into the root view through `.environmentObject(store)`. The information injected through `environmentObject` or `environment` can only be used in the view tree created for the current scene.



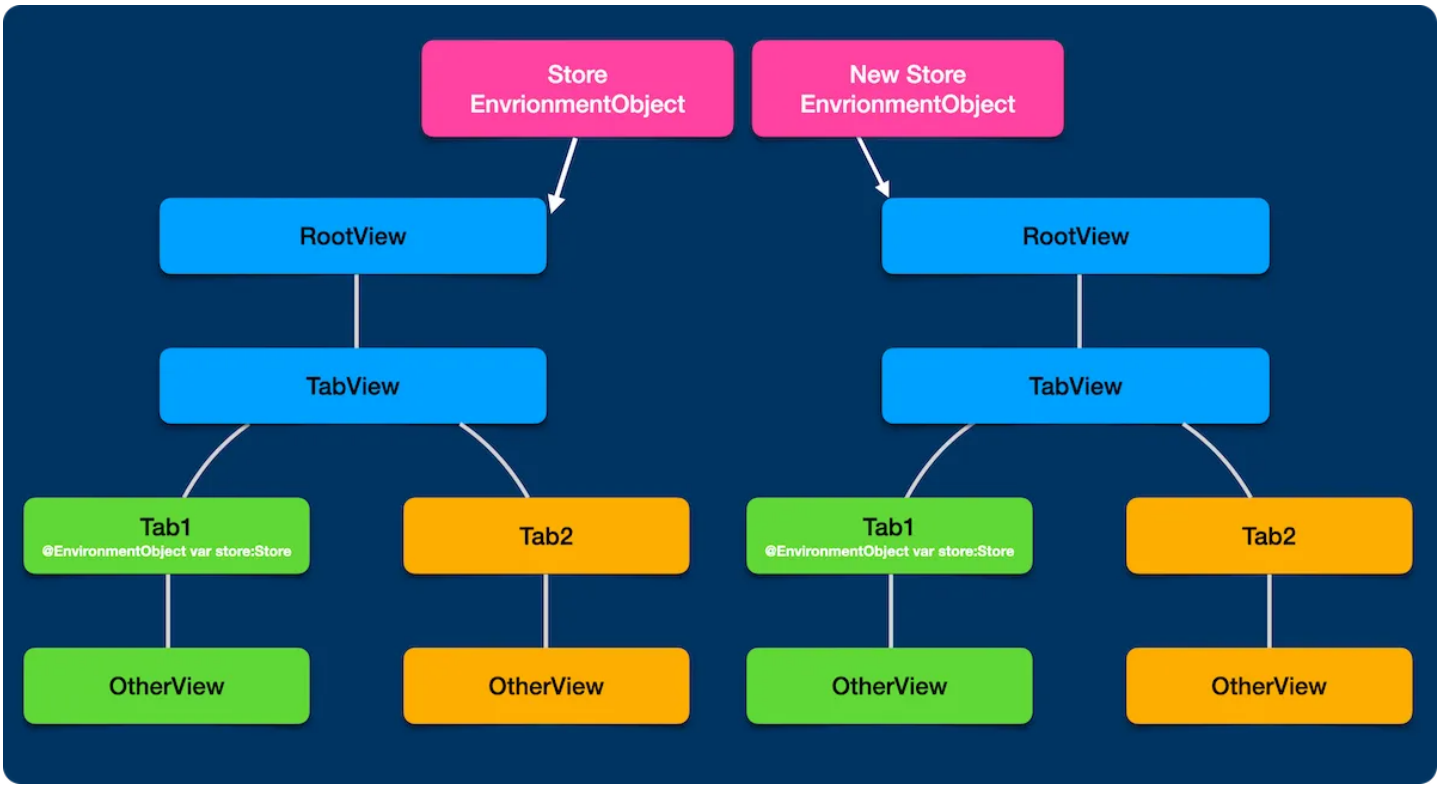
Although the system creates a new view tree for each new scene (new window), since the same Store instance is injected into the root view of the new scene, the application state obtained in different windows is completely consistent even though the scenes are different.

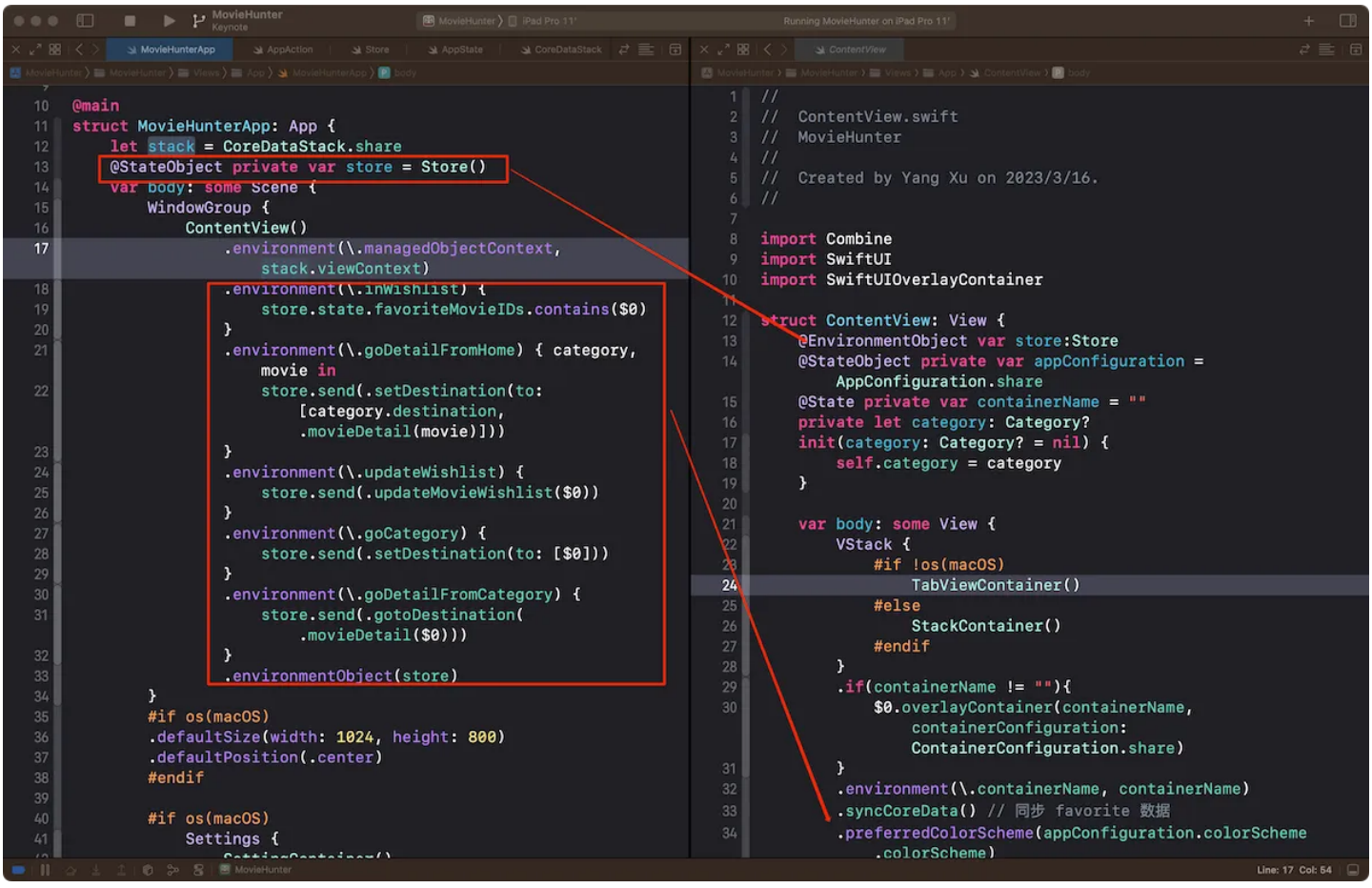


As “Movie Hunter” uses programmatic navigation, the state of the view stack and TabView are saved in the Store, so there may be situations where operations are synchronized.

Therefore, if we plan to introduce the application to a multi-window platform, it is best to consider this situation in advance and think about how to organize the application’s state.

For the current state configuration of “Movie Hunter,” we can solve the above problem by moving the location of creating the Store instance to the scene (moving the code related to Store in MovieHunterApp to ContentView).


Store
EnvrionmentObject
New Store
EnvrionmentObject
RootView
RootView
TabView
TabView
Tab1
@EnvironmentObject var store:Store
Tab2
Tab1
@EnvironmentObject var store:Store
Tab2
OtherView
OtherView
OtherView
OtherView


@main
struct MovieHunterApp: App {
let stack = CoreDataStack.share
@StateObject private var store = Store()
var body: some Scene {
WindowGroup {
ContentView()
.environment(\.managedObjectContext,
stack.viewContext)
.environment(\.inWishlist) {
store.state.favoriteMovieIDs.contains($0)
}
.environment(\.goDetailFromHome) { category,
movie in
store.send(.setDestination(to:
[category.destination,
.movieDetail(movie)]))
}
.environment(\.updateWishlist) {
store.send(.updateMovieWishlist($0))
}
.environment(\.goCategory) {
store.send(.setDestination(to: [$0]))
}
.environment(\.goDetailFromCategory) {
store.send(.gotoDestination(
.movieDetail($0)))
}
.environmentObject(store)
}
#if os(macOS)
.defaultSize(width: 1024, height: 800)
.defaultPosition(.center)
#endif
#if os(macOS)
Settings {
//
// ContentView.swift
// MovieHunter
//
// Created by Yang Xu on 2023/3/16.
//
import Combine
import SwiftUI
import SwiftUIOverlayContainer
struct ContentView: View {
@EnvironmentObject var store:Store
@StateObject private var appConfiguration =
AppConfiguration.share
@State private var containerName = ""
private let category: Category?
init(category: Category? = nil) {
self.category = category
}
var body: some View {
VStack {
#if !os(macOS)
TabViewContainer()
#else
StackContainer()
#endif
}
.if(containerName != ""){
$0.overlayContainer(containerName,
containerConfiguration:
ContainerConfiguration.share)
}
.environment(\.containerName, containerName)
.syncCoreData() // 同步 favorite 数据
.preferredColorScheme(appConfiguration.colorScheme
.colorScheme)

However, creating independent Store instances in every scene is not suitable for all situations. In many cases, developers only want to maintain one Store instance in their application. I will demonstrate this scenario through another simple application.

> I believe many readers may not agree with creating an independent Store instance in every scene. As for whether this approach is correct and in line with the current popular Single Source of Truth concept, we will continue to discuss it later.

This is an extremely simple demo — SingleStoreDemo. It only has one Store instance and supports multiple windows. Users can independently switch TabViews in each window, and the state of TabView is held by the unique Store instance. Click the “Hit Me” button in any tab in any window to increase the click count. The click count is displayed at the top of the window.

When designing the state of this app, we need to consider which states are global to the entire application and which states are limited to the current context (window).

```
struct AppReducer: ReducerProtocol {
    struct State: Equatable {
        var sceneStates: IdentifiedArrayOf<SceneReducer.State> = .init()
        var hitCount = 0
    }
}

struct SceneReducer: ReducerProtocol {
    struct State: Equatable, Identifiable {
```

```
        let id: UUID
        var tabSelection: Tab = .tab1
    }
}
```

In the total state of the application, in addition to serving the global `hitCount`, we also separate the state of each scene for possible multi-scenario needs. And we use IdentifiedArray to manage the state of different scenes.

When a scene is created, its own state data is created in the App State through the code in onAppear, and when the scene is deleted, the state of the current scene is cleared through the code in onDisappear.

```
.onAppear {
    viewStore.send(.createNewScene(sceneID)) // create new scene state
}
.onDisappear {
    viewStore.send(.deleteScene(sceneID)) // delete current scene state
}
```

As a result, the requirement for independent operation of multiple windows is achieved through a single Store instance.

It should be noted here that for some unknown reason (perhaps related to the seed of random numbers), if the root view is created through the same scene declaration and @State is used to create UUIDs or random numbers,

even in different windows created at different times, the values of UUIDs or random numbers are completely the same. This makes it impossible to create different sets of state for different scenes (the current scene state uses UUID as the identifier). To avoid this situation, new UUIDs or random numbers need to be generated in onAppear.

```
.onAppear {
    sceneID = UUID()
    ...
}
```

> This problem also appeared in the scene where overlayContainer was created in “Movie Hunter” (used to display full-screen movie stills), and was solved using the above method.

Although SingleStoreDemo uses TCA as the data flow framework, this does not mean that TCA has any special advantages in implementing similar requirements. In SwiftUI, as long as developers understand the relationship between state, declaration, and response, they can organize data in any form they want. Whether it is to unify the management of state or disperse it in different views, each has its own advantages and meanings. In addition, SwiftUI itself provides developers with many property wrapper types specifically designed to handle multi-scene mode, such as: @AppStorage, @SceneStorage, @FocusedSceneValue, @FocusedSceneObject, etc.

Going back, let's take another look at the implementation of multiple Store instances in "Movie Hunter". Is there no need for application-level (global) state requirements?

Of course not. In "Movie Hunter", most of the application-level states are managed by @AppStorage, while some other global states are maintained through Core Data. This means that although "Movie Hunter" adopts the external form of creating a separate Store instance for each scene, the underlying logic is essentially no different from the TCA implementation of SingleStore.

I believe that developers should adopt appropriate means according to their needs, rather than being confined to a specific data flow theory or framework.

Finally, let's talk about another source of truth-related issue encountered when adapting "Movie Hunter" to macOS.

In order to make "Movie Hunter" more compliant with the standards of macOS applications, we moved the views to menu items and removed TabView in the macOS code.

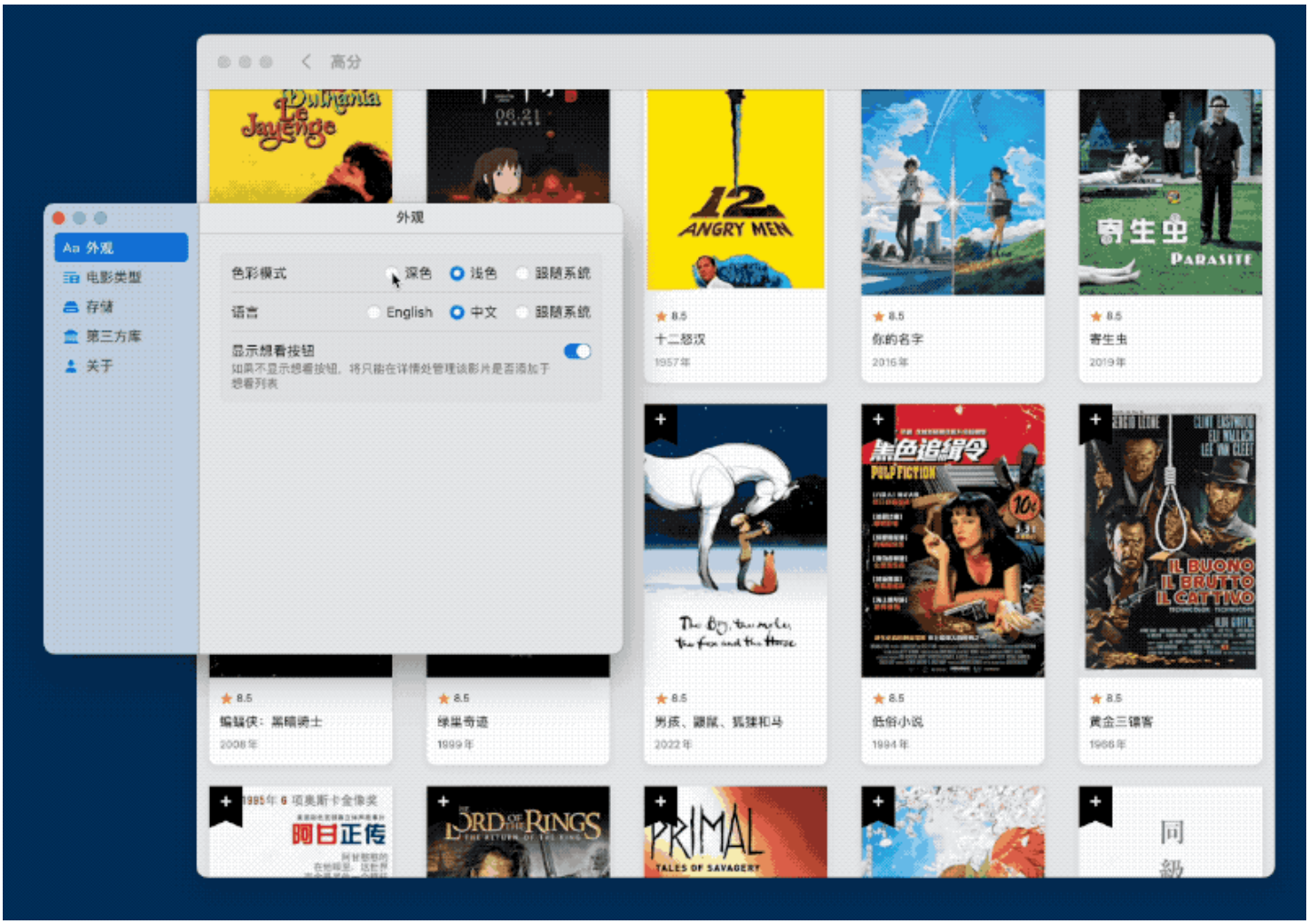
```
@main
struct MovieHunterApp: App {
    let stack = CoreDataStack.share
    @StateObject private var store = Store()
    var body: some Scene {
```

```
        WindowGroup {
         ...
        }

        #if os(macOS)
            Settings {
                SettingContainer() // 声明设置视图
            }
        #endif
    }
}

// ContentView
VStack {
    #if !os(macOS)
        TabViewContainer()
    #else
        StackContainer()
    #endif
}
```

After making these changes, you will find that we can only change the color mode and language of the movie information window in the settings, and the settings view will not change together like on the iPhone and iPad.



This is because, in macOS, using Settings to declare the Settings window also creates a new scene, which creates an independent view tree. In iOS, we change the color and language by modifying the environment values in the root view (ContentView), which does not affect the Settings scene in macOS. Therefore, in macOS, we need to adjust the environment values for the Settings view separately.

```
struct SettingContainer: View {
    @StateObject var configuration = AppConfiguration.share
    @State private var visibility: NavigationSplitViewVisibility = .doubleColum
```

```
        var body: some View {
            NavigationSplitView(columnVisibility: $visibility) {
              ...
            } detail: {
               ...
            }
            #if os(macOS)
            .preferredColorScheme(configuration.colorScheme.colorScheme)
            .environment(\.locale, configuration.appLanguage.locale)
            #endif
        }
    }
```

> It is precisely because @AppStorage is used to manage global state that the adaptation work of the settings window can be easily completed without introducing a Store instance.

Summary

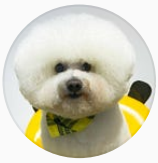
Compared to adjusting view layout for different platforms, the issue discussed today may not be as prominent and is easily overlooked.

However, as long as these points are understood and planned for in advance, the adaptation process will be smoother. Developers can then devote more energy to creating unique user experiences for different platforms.

That concludes today's discussion. Thank you for listening and I hope it has been helpful to you.

I hope this article can be helpful to you. You are also welcome to communicate with me through Twitter, Discord channel, or the message board of my blog.

Swiftui Swift IOS App Development IOS Development Mobile App Development



Written by fatbobman (东坡肘子)

Following

109 Followers

As a developer enthusiast, I enjoy fitness, wellness, and drinking tea. My energy is mainly focused on taking care of my furry pets.

More from fatbobman (东坡肘子)

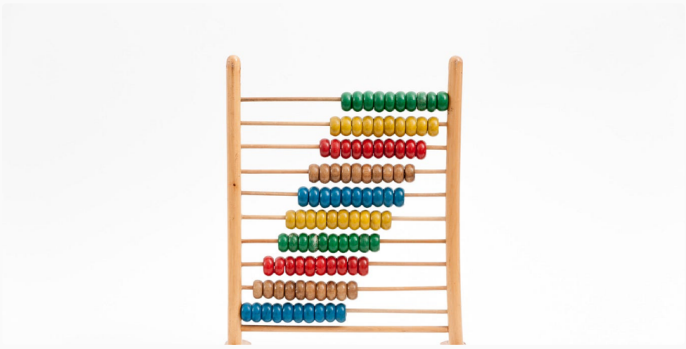
fatbobman (东坡肘子) in ITNEXT

Ask Apple 2022 Q&A Related on Core Data (Part 1)

This article compiles some of the Q&A related to Core Data from Ask Apple 2022,...

18 min read · May 7

20



fatbobman (东坡肘子) in ITNEXT

Count Queries in Core Data: The Master Guide

This article will introduce several methods for querying and using count in Core Data,...

9 min read · May 4

10

fatbobman (东坡肘子)

Ask Apple 2022 Q&A Related to SwiftUI (Part 1)

This Q&A article covers a range of topics related to SwiftUI in the Ask Apple 2022...

15 min read · May 3

16

fatbobman (东坡肘子)

SwiftUI's StateObject and ObservedObject: The Key...

This article will introduce the similarities, differences, principles, and precautions...

6 min read · Apr 10

7

See all from fatbobman (东坡肘子)

Recommended from Medium

fatbobman (东坡肘子)

Core Data with CloudKit: Syncing Local Database to iCloud Private...

In this article, we have explored how to synchronize a local database with an iCloud...

11 min read · Mar 17

3

Imad Ali Mohammad

10 Swift Coding Tips—ONE Liner

Swift is an incredibly powerful and expressive programming language that...

3 min read · Apr 27

73

Lists

Staff Picks
300 stories · 62 saves

Stories to Help You Level-Up at Work
19 stories · 19 saves

Self-Improvement 101
20 stories · 44 saves

Productivity 101
20 stories · 42 saves

Hemal Asanka

Making API calls with iOS Combine Future Publisher

What Is the Combine Framework?

8 min read · May 5

12

Umut SERIFLER

Drag&Drop in SwiftUI

Moving content from one part of an app to another, or from one app to another is...

5 min read · Jan 20

44

Mobile@Exxeta

SwiftGen—How to neatly get rid of magic strings in iOS projects

It's worth a try

8 min read · May 8

21

Arangott Ramesh Chandran

How to Use VisionKit in SwiftUI for Text and Barcode Scanning on...

Introducing DataScannerViewController

9 min read · May 5

1