

架构设计文档

在 SwiftUI 中，由于引入了 Binding 绑定、Observable 观察者模式，并且得益于 Swift 引入命令式编程，以及闭包、Lambda函数等语言特性，SwiftUI 简化了传统的事件驱动 (Event Driven)，并且由传统的 MVC (Model - View -Controller) 转向了 MVVM (Model-View-ViewModel) 模式。

一、MVVM

- **Model**：MVVM 是 Domain Model 模式的实现，Model 将包括我们的数据结构以及业务逻辑和验证逻辑，即数据来源，服务器上业务逻辑操作，从后端得到传递数据。
- **View**：视图将负责定义用户在屏幕上看到的内容的结构，布局和外观。
- **ViewModel**：ViewModel 将成为视图和模型之间的桥梁，它负责视图逻辑的管理。通常，视图模型通过调用模型类中的方法与模型交互，然后，视图模型将以一种视图可以轻松使用的方式提供模型数据。

DataBinding：数据绑定是此模式的基本机制：通常通过视图本身中的**声明性语法**，使视图模型和视图保持恒定同步。这意味着用户通过视图对数据所做的更改将自动在视图模型中报告，而无需将此责任交给开发人员。

MVVM (Model-View-ViewModel) 并非一种框架，而是一种**架构模式**，一种思想，一种组织和管理代码的方法。本质上是 MVC (Model-View- Controller) 的一种改进版。

原则

在 MVVM 架构中 View 和 Model 不能直接通信，必须通过 ViewModel。ViewModel 是 MVVM 的核心，它通常要实现一个观察者，当 Model 数据发生变化时 ViewModel 能够监听并通知到对应的 View 做 UI 更新，反之当用户操作 View 时 ViewModel 也能获取到数据的变化并通知 Model 数据做出对应的更新操作。这就是 MVVM 中数据的**双向绑定**。

搜索实例

在实现搜索、歌单获取、歌词滚动显示等获取用户信息流且进行展示的功能中，我们使用了 MVVM 模式，比如在搜索中：`DataModel` 即 Model，用于构建 URL ，并且通过 API 接口进行 GET 请求并获得相应的数据：

```
class DataModel {
    private var dataTask: URLSessionDataTask?

    func loadSongs(searchTerm: String, completion: @escaping ([[Songs]] -> Void)) {
        // 取消上一次任务
        dataTask?.cancel()
        // build api: itunes.apple.com/search?term= $searchTerm &entity=song
        guard let url = buildUrl(forTerm: searchTerm) else {
            completion([])
            return
        }

        dataTask = URLSession.shared.dataTask(with: url) { data, _, _ in
            guard let data = data else {
                completion([])
                return
            }

            // TODO: use swiftJSON
            if let songResponse = try? JSONDecoder().decode(ItunesSongResponse.self, from: data) {
                completion(songResponse.songs)
            }
        }

        dataTask?.resume()
    }

    private func buildUrl(forTerm searchTerm: String) -> URL? {
        guard !searchTerm.isEmpty else { return nil }

        let queryItems = [
            URLQueryItem(name: "term", value: searchTerm),
            URLQueryItem(name: "entity", value: "song"),
        ]
        var components = URLComponents(string: "https://itunes.apple.com/search")
        components?.queryItems = queryItems
    }
}
```

```

        return components?.url
    }
}

```

而 `SongListViewModel` 即 `ViewModel`，连接 `DataModel` 和搜索页面，其本身是一个可观测的对象 `ObservableObject`，当 `Model` 中数据变化后，通过观测模式即使用 `sink()`, `store()` 等函数通知变化。

```

// Identifiable 唯一标识
// ObservableObject 观测对象
class SongViewModel: Identifiable, ObservableObject {
    let id: Int
    let trackName: String
    let artistName: String
    let artworkUrl: String
    @Published var artwork: Image?

    init(song: Songs) {
        self.id = song.id
        self.trackName = song.trackName
        self.artistName = song.artistName
        self.artworkUrl = song.artworkUrl
    }
}

```

```

import Combine
import SwiftUI
import Foundation

class SongListViewModel: ObservableObject {
    // 自动监视 KVO
    @Published var searchTerm: String = ""
    @Published public private(set) var songs: [SongViewModel] = []

    private let dataModel: DataModel = DataModel()
    private var disposables = Set<AnyCancellable>()

    init() {
        // Combine 响应式框架，用来处理随时间变化的事件
        $searchTerm
            .sink(receiveValue: loadSongs(searchTerm:)) // sink 接收值，receiveValue 收到值后执行闭包
            .store(in: &disposables) // 异步的订阅需要保存
    }

    private func loadSongs(searchTerm: String) {
        songs.removeAll()
        // Itunes API search
        dataModel.loadSongs(searchTerm: searchTerm) { songs in
            songs.forEach { self.appendSong($0) }
        }
    }

    private func appendSong(_ song: Songs) {
        let songViewModel = SongViewModel(song: song)
        // 主线程更新 UI
        DispatchQueue.main.async {
            self.songs.append(songViewModel)
        }
    }
}

```

```
}  
}
```

最后是搜索视图 SearchView:

```
import SwiftUI  
import Kingfisher  
  
struct SearchView: View {  
    @EnvironmentObject var model: Model  
    // 观测对象，如果搜索有结果自动更新其值  
    @ObservedObject var viewModel: SongListViewModel  
  
    var body: some View {  
        NavigationView {  
            VStack {  
                SearchBar(searchTerm: $viewModel.searchTerm)  
                Text("\(viewModel.songs.count)")  
                if viewModel.songs.isEmpty {  
                    Spacer()  
                    EmptyStateView(theme: $model.themeColor)  
                    Spacer()  
                } else {  
                    List(viewModel.songs) { song in  
                        SongView(song: song)  
                    }  
                    .listStyle(PlainListStyle())  
                }  
            }  
            .navigationBarTitle("Search", displayMode: .automatic)  
        }  
    }  
}
```

SwiftUI 数据绑定

- **@Binding**: 允许我们声明一个值实际上来自其他地方，并且必须在这些上下文中共享；
- **@ObservedObject** and **@EnvironmentObject**: 它们是与绑定相似的两个属性，只是它们具有更大的共享范围，可能使具有这两个标记之一的对象在应用程序的所有视图之间共享；
- **@Published** and **@ObservableObject** protocol: 允许我们创建可观察对象，这些对象在发生更改时自动宣布。可以在SwiftUI中使用所有与Observableobject协议兼容的类，并在其值更改时通知您，以便您知道何时更新视图。
- **@State**: 这是一个标签，允许我们检查Binding，ObservedObject 或 EnvironmentObject的值。如果@State标记的值更改，SwiftUI决定是否使视图无效。

也正是基于通过数据绑定的特性，我们的 NoSignal 音乐播放器可以更快地实现全局变量控制、模块化功能。

总的来说，基于 SwiftUI 的 MVVM 架构，巧妙地利用了 KVO、KVC 和观察者设计模式，我们可以设计出更多异步架构。

二、持久化数据库

在我们的 APP 进行合并 AR 与主体时，我发现 Itunes 搜索接口时而出现问题，于是我决定将所有接口换成网易云的 API，但我没想到会如此困难。

首先，网易云接口众多，如何选取我需要的，如何对接，面对海量的请求与响应，我需要先找到一个能够存储部分数据的**本地化持久数据库**，即 Core Data，并且为了用户体验，结合 SwiftUI 中新推出的 SwiftUI lifecycle 即生命周期去初始化数据。

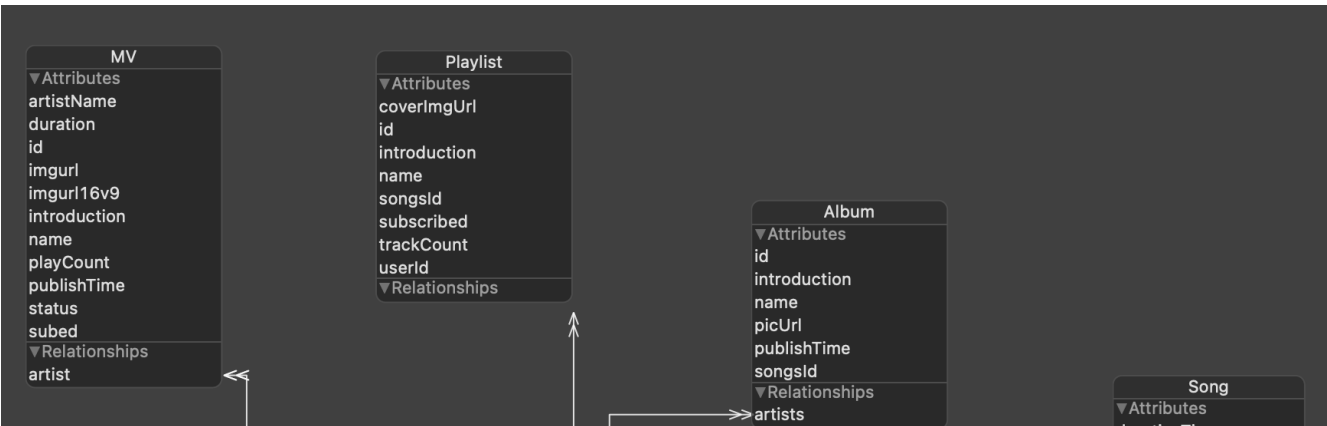
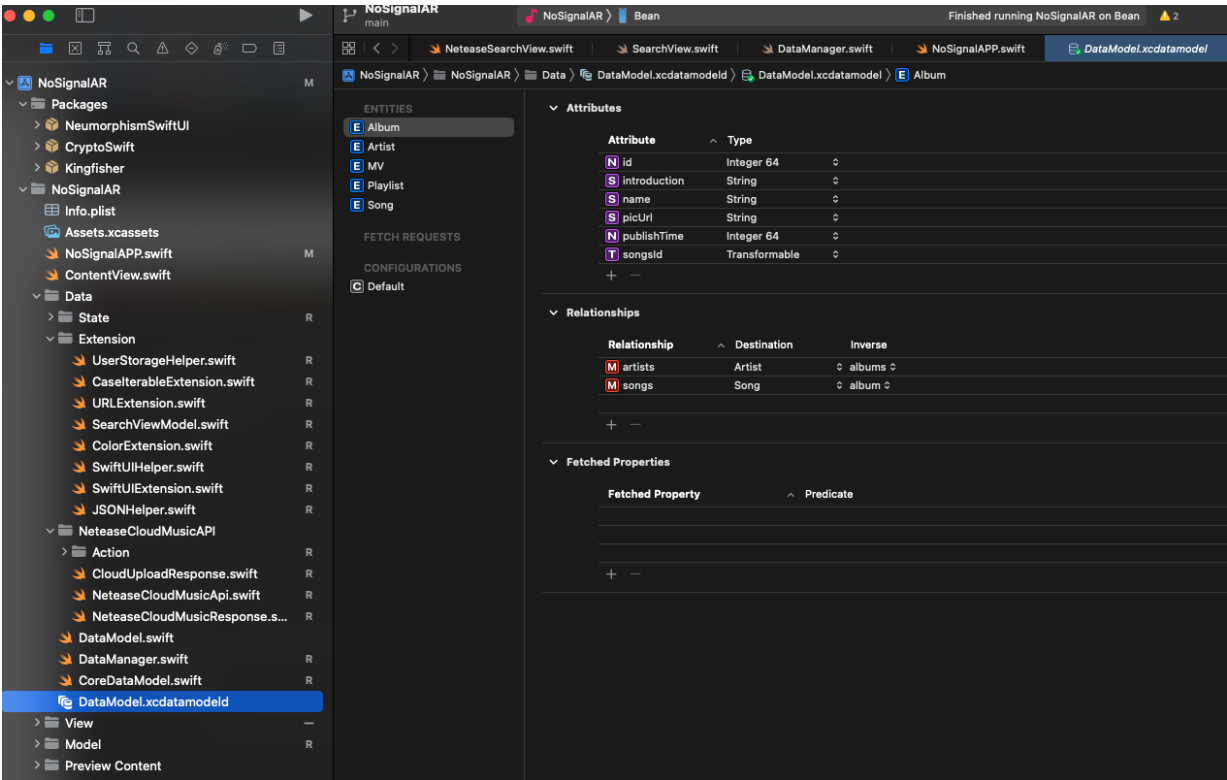
```
@main
struct NoSignalApp: App {
    // SwiftUI 生命周期
    @Environment(\.scenePhase) var scenePhase
    @UIApplicationDelegateAdaptor(AppDelegate.self) var delegate: AppDelegate
    @StateObject var store = Store.shared
    @StateObject var player = Player.shared

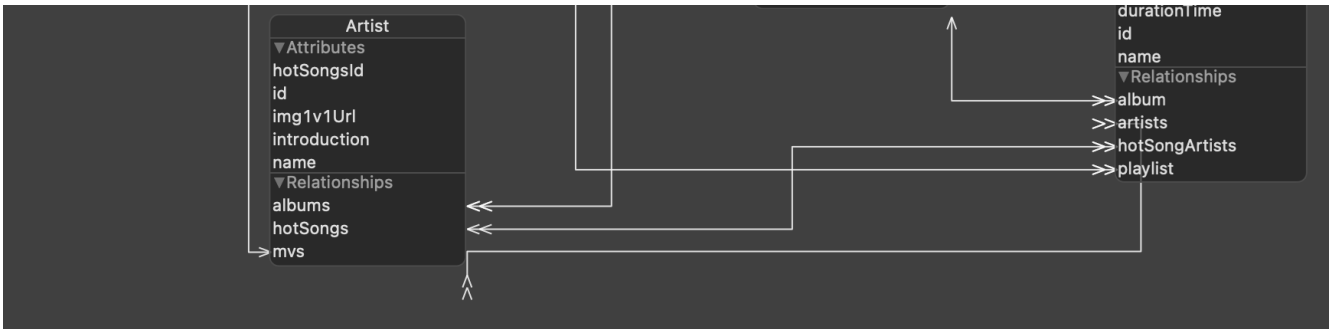
    let context = DataManager.shared.context()

    init() {
        updateSongs()
    }

    var body: some Scene {
        WindowGroup {
            // APP 启动时拉取曲库、注册 AR 组件、数据库、全局状态变量
            ContentView()
                .onChange(of: scenePhase, perform: { value in
                    if value == .active {
                        updateSongs()
                        InstrumentComponent.registerComponent()
                    }
                })
                .onAppear {
                    store.dispatch(.loginRefreshRequest)
                }
                .environmentObject(store)
                .environmentObject(player)
                .environment(\.managedObjectContext, context)
        }
    }
}
```

同时创建本地数据库 `DataModel.xcdatamodeld`，并且创建所需的 Data 数据实体：





此时就可以在 CoreDataManager 中根据上下文 `context` 进行动态地 FetchRequest，这样就不用每次打开 APP 就要登录、重新拉取用户歌单等请求，大大优化用户体验。

```

138     defer { save() }
139     _ = model.entity(context: context())
140 }
141
142 public func getAlbum(id: Int) -> Album? {
143     var album: Album? = nil
144     do {
145         let context = self.context()
146         let fetchRequest = NSFetchedRequest<NSFetchRequestResult>(entityName: "Album")
147         fetchRequest.predicate = NSPredicate(format: "%K == \(id)", "id")
148         album = try context.fetch(fetchRequest).first as? Album
149     } catch let error {
150         print("\(function): \(error)")
151     }
152     return album
153 }
154
155 public func getArtist(id: Int) -> Artist? {
156     var artist: Artist? = nil
157     let context = self.context()
158     let fetchRequest = NSFetchedRequest<NSFetchRequestResult>(entityName: "Artist")
159     fetchRequest.predicate = NSPredicate(format: "%K == \(id)", "id")
160     do {
161         artist = try context.fetch(fetchRequest).first as? Artist
162     } catch let error {
163         print("\(function): \(error)")
164     }
165     return artist
166 }
167
168 public func getMV(id: Int64) -> MV? {
169     var mv: MV? = nil
170     let context = self.context()
171     let fetchRequest = NSFetchedRequest<NSFetchRequestResult>(entityName: "MV")
172     do {
173         mv = try context.fetch(fetchRequest).first as? MV
174     } catch let error {
175         print("\(function): \(error)")
176     }
177     return mv
178 }
```

制订好 Core Data 和相应的数据实体（Swift 作为一门动态的强类型语言，对变量的类型要求非常严格）之后，就可以开始编写 API、响应的请求和响应了。

三、Redux 架构

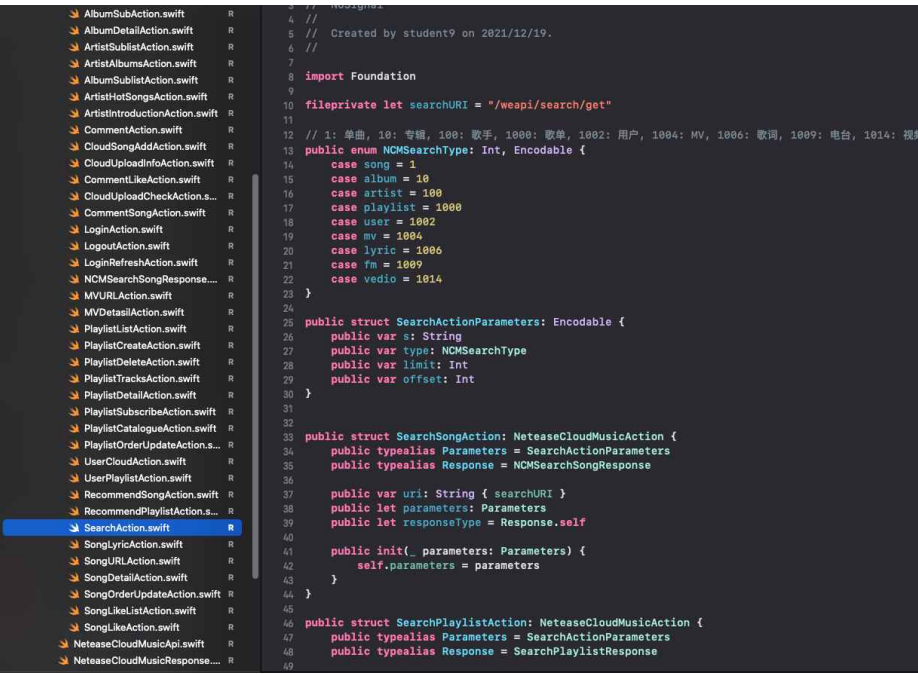
网易云接口

网易云 API 非常多，登录、注销、获取歌单、播放列表、歌手、搜索、播放和获取图片等等，还好感谢已经有前人将网易云的 API 完善且编写成文档：[Binaryify/NeteaseCloudMusicApi: 网易云音乐 Node.js API service \(github.com\)](#)。

由于项目时间原因，我只编写了其中与歌单、播放、MV、歌曲、歌词以及部分用户操作的相关接口，并抽象成动作：

```

1 //
2 // SearchAction.swift
3 // NoSignal
4 //
```





```
AlbumSubAction.swift
AlbumDetailAction.swift
ArtistSublistAction.swift
ArtistAlbumsAction.swift
AlbumSublistAction.swift
ArtistHotSongsAction.swift
ArtistIntroductionAction.swift
CommentAction.swift
CloudSongAddAction.swift
CloudUploadInfoAction.swift
CommentLikeAction.swift
CloudUploadCheckAction.swift
CommentSongAction.swift
LoginAction.swift
LogoutAction.swift
LoginRefreshAction.swift
NCMSearchSongResponse...
MVURLAction.swift
MVDetasilAction.swift
PlaylistListAction.swift
PlaylistCreateAction.swift
PlaylistDeleteAction.swift
PlaylistTracksAction.swift
PlaylistDetailAction.swift
PlaylistSubscribeAction.swift
PlaylistCatalogueAction.swift
PlaylistOrderUpdateAction...
UserCloudAction.swift
UserPlaylistAction.swift
RecommendSongAction.swift
RecommendPlaylistAction...
SearchAction.swift
SongLyricAction.swift
SongURLAction.swift
SongDetailAction.swift
SongOrderUpdateAction.swift
SongLikeListAction.swift
SongLikeAction.swift
NeteaseCloudMusicApi.swift
NeteaseCloudMusicResponse...
```

```
4 //
5 // Created by student9 on 2021/12/19.
6 //
7
8 import Foundation
9
10 fileprivate let searchURI = "/weapi/search/get"
11
12 // 1: 单曲, 10: 专辑, 100: 歌手, 1000: 歌单, 1002: 用户, 1004: MV, 1006: 歌词, 1009: 电台, 1014: 视频
13 public enum NCMSearchType: Int, Encodable {
14     case song = 1
15     case album = 10
16     case artist = 100
17     case playlist = 1000
18     case user = 1002
19     case mv = 1004
20     case lyric = 1006
21     case fm = 1009
22     case vedio = 1014
23 }
24
25 public struct SearchActionParameters: Encodable {
26     public var s: String
27     public var type: NCMSearchType
28     public var limit: Int
29     public var offset: Int
30 }
31
32
33 public struct SearchSongAction: NeteaseCloudMusicAction {
34     public typealias Parameters = SearchActionParameters
35     public typealias Response = NCMSearchSongResponse
36
37     public var url: String { searchURI }
38     public let parameters: Parameters
39     public let responseType = Response.self
40
41     public init(_ parameters: Parameters) {
42         self.parameters = parameters
43     }
44 }
45
46 public struct SearchPlaylistAction: NeteaseCloudMusicAction {
47     public typealias Parameters = SearchActionParameters
48     public typealias Response = SearchPlaylistResponse
49 }
```

但即便如此，代码量（LOC）也超过了两千行，可见如果真的要实现网易云所有的 API 相关功能，代码量将远不止这个数量级。

```
NeteaseCloudMusicAPI git:(main) x find . "(" -name "*.m" -or -name "*.mm" -or -name "*.
cpp" -or -name "*.swift" ")" -print0 | xargs -0 wc -l
```



```
103 ./Action/LoginAction.swift
60 ./Action/ArtistAlbumsAction.swift
22 ./Action/LoginRefreshAction.swift
33 ./Action/CloudUploadCheckAction.swift
27 ./Action/SongLikeListAction.swift
30 ./Action/SongLikeAction.swift
102 ./Action/CommentSongAction.swift
45 ./Action/ArtistIntroductionAction.swift
77 ./Action/ArtistHotSongsAction.swift
61 ./Action/ArtistMVAAction.swift
44 ./Action/ArtistSublistAction.swift
85 ./Action/SearchAction.swift
47 ./Action/AlbumSublistAction.swift
26 ./Action/PlaylistSubscribeAction.swift
63 ./Action/MVDetasilAction.swift
2527 total
```

所以，动作分离显得尤为重要，不能将所有的实体、请求与视图揉杂在一起，导致开发、维护以及代码阅读变得十分困难。

Redux 架构

面对如此多以及复杂的数据，如何处理以及如何维护？每个数据的状态如果发生了变化，是否会影响其他数据，会怎么影响视图？

基于 MVVM 以及观察者模式，我想到了前端中常用的 Redux 架构，也即自定义一个统一的方式管理数据状态，便于组件之间进行数据流动、并且实现 UI 视图与数据逻辑之间的分离。

响应式编程

在具体实现 Redux 架构之前，还需要了解 SwiftUI 中提供的 Combine 架构，为给定的事件源创建单个处理链，链的每个部分都是一个合并运算符，对从上一步接收到的元素执行不同的操作。在官方文档，Combine的定义为：“通过组合事件处理运算符**自定义异步事件**的处理。”

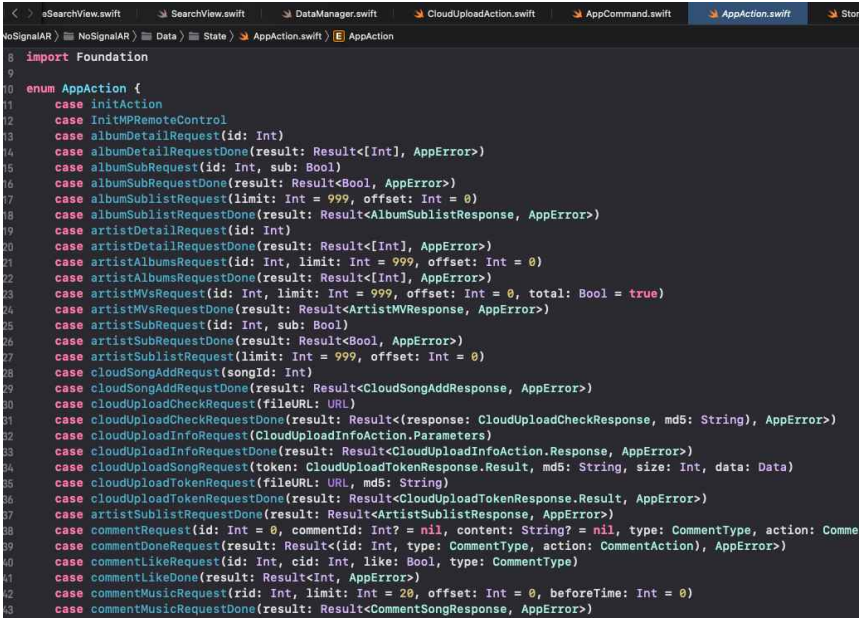
Combine 框架提供了一个**声明性**的 Swift API，用于随着时间的推移处理值。这些值可以表示多种异步事件。Combine 声明发布者公开可随时间变化的值，订阅者从发布者接收这些值。

由 Publisher（发布者）、Subscriber（订阅者）和 Operator（操作符）组成。

结合在 MVVM 中提到的搜索实例，不难理解 Combine 在 Redux 中扮演的重要角色。

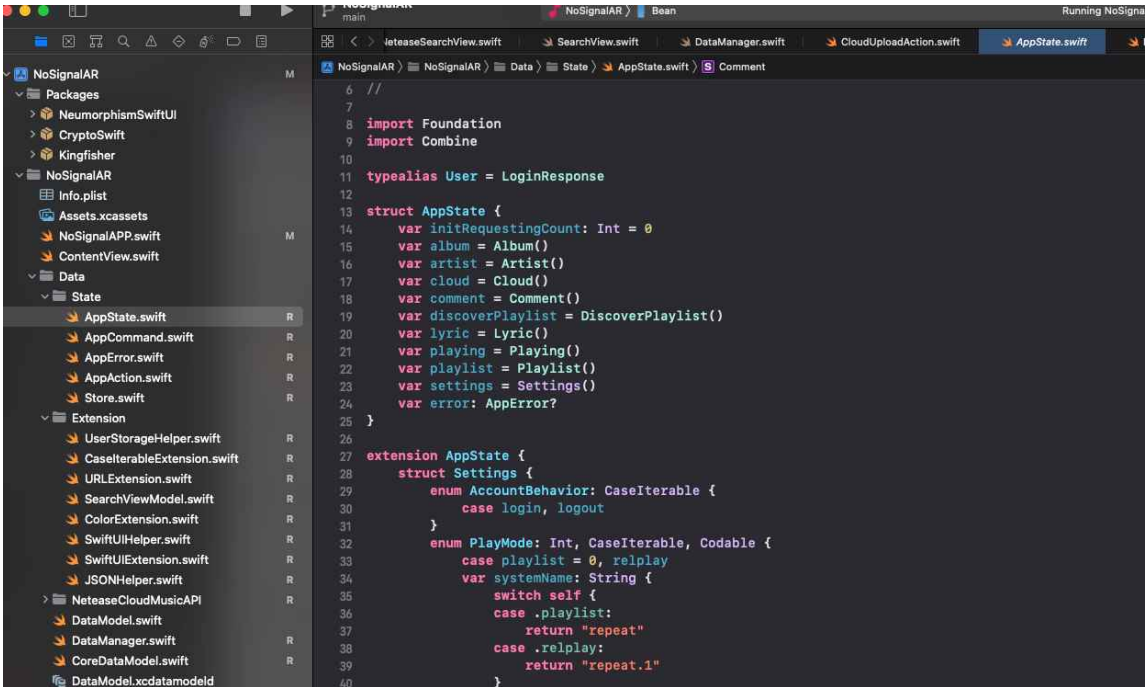
Action

数据通过上述定义的一系列动作 Action 传输，最终流入 AppState 中，并由 State 存放相关的状态，但由于网易云接口多且难以统一，我需要 `AppAction` 一个枚举结构，通过 switch 语句来帮助我管理这些动作（AppError 同样用枚举 enum 结构管理错误）：



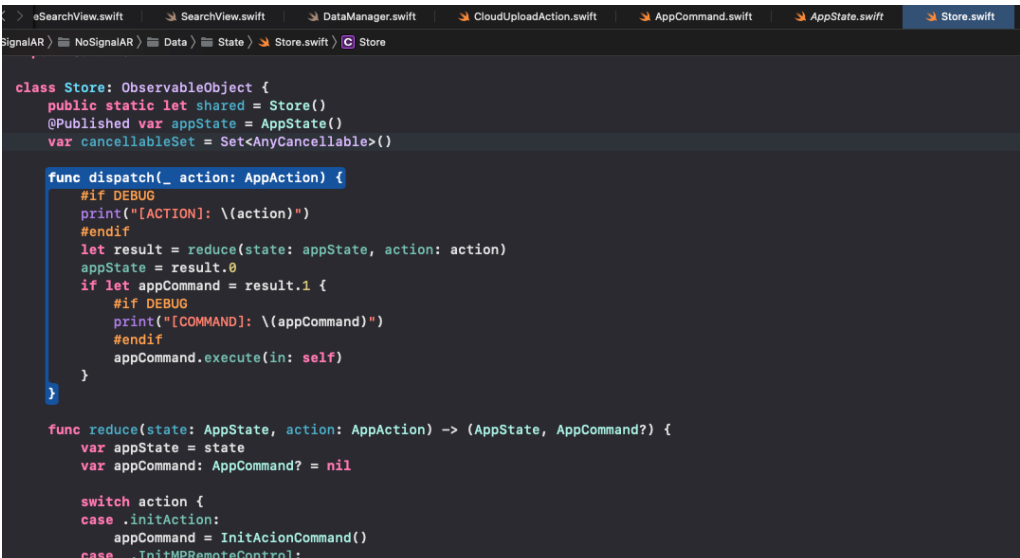
State

state 包含了所有的应用数据以及其相关状态，如歌词状态具有是否正在请求、错误以及 ViewModel，并且不同的数据使用单独的结构体进行维护，实现统一的管理。



Store

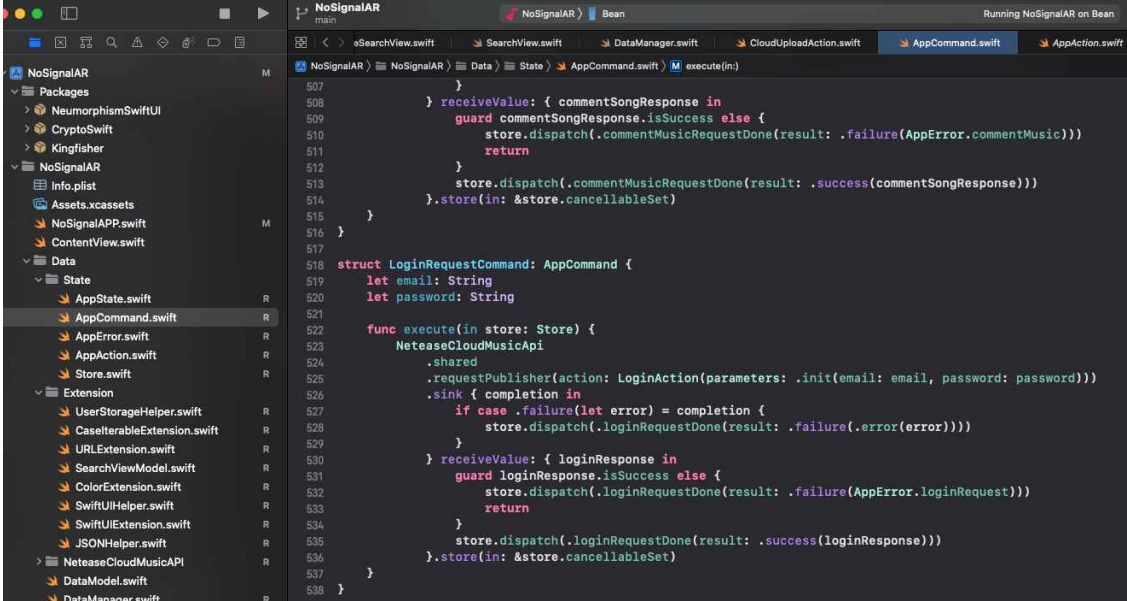
Store 存有当前的 AppState，以及 reduce 函数和分发动作的特性，其中 Reducer 接受原有的 State 和触发的动作 Action，生成新的 State，并返回执行相关额外操作的副作用 Command。



```

        appCommand = InitMPRemoteControlCommand()
    case .albumDetailRequest(let id):
        appState.album.detailRequesting = true
        appCommand = AlbumDetailRequestCommand(id: id)
    case .albumDetailRequestDone(let result):
        switch result {
        case .success: break
        case .failure(let error):
            appState.error = error
        }
        appState.album.detailRequesting = false
    case .albumSubRequest(let id, let sub):
        appCommand = AlbumSubRequestCommand(id: id, sub: sub)
    case .albumSubRequestDone(let result):
        switch result {
        case .success:
            appCommand = AlbumSubRequestDoneCommand()
        case .failure(let error):
            appState.error = error
        }
    }
}
```

对于不同的 AppComand，通过触发不同的动作执行函数，AppState 的状态进行改变（详见 execute 函数，使用 in 关键字修改当前 AppState）：



综上，结合 SwiftUI 的数据绑定、MVVM 架构以及 Combine 函数式编程，我在 SwiftUI 中实现了自己的 Redux 架构。

五、NoSignal 目录

CSS	
1	.
2	├── NoSignalAR
3	│ ├── ARModels
4	│ │ ├── gramophone.mp3
5	│ │ ├── gramophone.usdz
6	│ │ ├── symbol1.usdz
7	│ │ └── symbol2.usdz
8	│ ├── ARMusicView.swift
9	│ ├── Assets.xcassets
10	│ ├── ContentView.swift
11	│ └── Data

12				└─ DataModel.swift
13				└─ DataModel.xcdatamodeld
14				└─ DataModel.xcdatamodel
15				└─ contents
16				└─ Extension
17				└─ CaseIterableExtension.swift
18				└─ ColorExtension.swift
19				└─ CoreDataModel.swift
20				└─ DataManager.swift
21				└─ JSONHelper.swift
22				└─ SearchViewModel.swift
23				└─ SwiftUIExtension.swift
24				└─ SwiftUIHelper.swift
25				└─ URLExtension.swift
26				└─ UserStorageHelper.swift
27				└─ Info.plist
28				└─ InstrumentEntity.swift
29				└─ Model
30				└─ AudioSessionManager.swift
31				└─ CommentViewModel.swift
32				└─ DiscoverPlaylistViewModel.swift
33				└─ LyricParser.swift
34				└─ LyricViewModel.swift
35				└─ Model.swift
36				└─ NeteaseSong.swift
37				└─ Player.swift
38				└─ PlaylistViewModel.swift
39				└─ RectangleCoverView.swift
40				└─ NeteaseCloudMusicAPI
41				└─ Action
42				└─ AlbumDetailAction.swift
43				└─ AlbumSubAction.swift
44				└─ AlbumSublistAction.swift
45				└─ ArtistAlbumsAction.swift
46				└─ ArtistHotSongsAction.swift
47				└─ ArtistIntroductionAction.swift
48				└─ ArtistMVAction.swift
49				└─ ArtistSubAction.swift
50				└─ ArtistSublistAction.swift
51				└─ CloudSongAddAction.swift
52				└─ CloudUploadAction.swift
53				└─ CloudUploadCheckAction.swift
54				└─ CloudUploadInfoAction.swift
55				└─ CloudUploadTokenAction.swift
56				└─ CommentAction.swift
57				└─ CommentLikeAction.swift
58				└─ CommentSongAction.swift
59				└─ LoginAction.swift
60				└─ LoginRefreshAction.swift
61				└─ LogoutAction.swift
62				└─ MVDetasilAction.swift
63				└─ MVURLAction.swift
64				└─ PlaylistCatalogueAction.swift
65				└─ PlaylistCreateAction.swift
66				└─ PlaylistDeleteAction.swift
67				└─ PlaylistDetailAction.swift
68				└─ PlaylistListAction.swift
69				└─ PlaylistOrderUpdateAction.swift
70				└─ PlaylistSubscribeAction.swift
71				└─ PlaylistTracksAction.swift

72					RecommendPlaylistAction.swift		
73					RecommendSongAction.swift		
74					SearchAction.swift		
75					SongDetailAction.swift		
76					SongLikeAction.swift		
77					SongLikeListAction.swift		
78					SongLyricAction.swift		
79					SongOrderUpdateAction.swift		
80					SongURLAction.swift		
81					Untitled.xcworkspace		
82						contents.xcworkspacedata	
83						xcshareddata	
84							IDEWorkspaceChecks.plist
85						xcuserdata	
86							student9.xcuserdatad
87							UserInterfaceState.xcuserstate
88						UserCloudAction.swift	
89						UserPlaylistAction.swift	
90						CloudUploadResponse.swift	
91						NeteaseCloudMusicApi.swift	
92						NeteaseCloudMusicResponse.swift	
93						Response	
94							NCMSearchSongResponse.swift
95						NoSignalAPP.swift	
96						Preview\ Content	
97							Preview\ Assets.xcassets
98							Contents.json
99						State	
100						AppAction.swift	
101						AppCommand.swift	
102						AppError.swift	
103						AppState.swift	
104						Store.swift	
105						View	
106							ArtistSubListView.swift
107							CommentView.swift
108							CommonGridView.swift
109							CreatedPlaylistView.swift
110							DescriptionView.swift
111							FetchAlbumDetailView.swift
112							FetchArtistDetailView.swift
113							FetchMVDetailView.swift
114							FetchPlaylistDetailView.swift
115						Helpers	
116							BlurView.swift
117							CornerRadius.swift
118							EmptyStateView.swift
119						Library	
120							LibraryView.swift
121							SongCardView.swift
122						LyricView.swift	
123						PlayingItem	
124						Elements	
125							InvisibleRefreshView.swift
126							PlayPauseButton.swift
127							PlayingNowButtonView.swift
128						NowPlayingView.swift	
129						PlaybackFullscreenView.swift	
130						PlaybackbarView.swift	
131						PlayerControlBarView.swift	

```
132 | | | └─ PlayingNowView.swift
133 | | | └─ PlayingProgressView.swift
134 | | └─ Playlist
135 | | | └─ AppleMusicPlayListDetailView.swift
136 | | | └─ DiscoverPlaylistView.swift
137 | | | └─ PlaylistCardView.swift
138 | | | └─ PlaylistSongsManageView.swift
139 | | | └─ PlaylistView.swift
140 | | └─ RecommendPlaylistView.swift
141 | └─ PlaylistManageView.swift
142 | └─ Search
143 | | └─ Discard
144 | | | └─ AppleMusicAPI.swift
145 | | | └─ RemoteSongCardView.swift
146 | | └─ NeteaseSearchView.swift
147 | | └─ SearchSongCardView.swift
148 | | └─ SearchView.swift
149 | | └─ SongListViewModel.swift
150 | └─ Song
151 | | └─ NeteaseSongRowView.swift
152 | └─ SongListView.swift
153 | └─ SubedAlbumsView.swift
154 | └─ SubedPlaylistView.swift
155 | └─ UserInfo
156 | | └─ AboutView.swift
157 | | └─ LoginView.swift
158 | | └─ NeteaseSongCoverView.swift
159 | | └─ ProfileHeader.swift
160 | | └─ UserView.swift
161 | └─ ViewModifier.swift
162 | └─ NoSignalAR.xcodeproj
163 | | └─ project.pbxproj
164 | | └─ project.xcworkspace
165 | | | └─ contents.xcworkspacedata
166 | | | └─ xcshareddata
167 | | | | └─ IDEWorkspaceChecks.plist
168 | | | └─ xcuserdata
169 | | | | └─ a1111.xcuserdatad
170 | | | | | └─ UserInterfaceState.xcuserstate
171 | | | | └─ student9.xcuserdatad
172 | | | | | └─ IDEFindNavigatorScopes.plist
173 | | | | | └─ UserInterfaceState.xcuserstate
174 | | └─ xcuserdata
175 | | | └─ a1111.xcuserdatad
176 | | | | └─ xcschemes
177 | | | | | └─ xcschememanagement.plist
178 | | | └─ student9.xcuserdatad
179 | | | └─ xcdebugger
180 | | | | └─ Breakpoints_v2.xcbkptlist
181 | | | └─ xcschemes
182 | | | | └─ xcschememanagement.plist
183 | └─ README.md
184
185 55 directories, 170 files
```

六、NoSignal 展示

