**Course ID: A057490**

**iOS开发技术作业报告**



|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Name** | **黄子恒** | | **Student No.** | **20216573** |
| **Class** | **软英2101** | | **Supervisor** | **郑瑛楠** |
| **Lab Name** | **Yangzhou APP** | | | |
| **Semester** | **2022-2023 academic year Spring semester** | | | |
| **Schedule** | **Week 8 to Week 15** | | | |
| **Submission Date** | **2023-06-08** | | | |
| **Grade** |  | **Examiner** | | **郑瑛楠** |
| **Evaluation Time** | |  |

**Software College, Northeastern University**

摘 要

**Yangzhou APP**是一个旅游服务应用，旨在为游客提供全方位的扬州旅行体验。

在架构方面，本项目采用MVVM架构，其中模型负责存储核心的应用程序数据，视图定义了用户界面的显示方式，而视图模型处理用户输入并更新模型。使用Combine框架的Publisher和Subscriber模型来处理数据的变化，实现视图和模型之间的双向绑定。

在数据存储方面，该项目应用了iOS的沙盒存储技术来实现数据的本地存储。通过getDocumentsDirectory()方法获取应用文件夹的路径，并利用getImageFromCache()和loadImageInToCache()方法从缓存中高效地获取和加载图片。另外，initMyself()方法用于应用的初始化设置，而通用的加载函数load()用于加载和解析指定文件的数据。项目还使用了JSONEncoder将用户数据编码为JSON格式并存储到名为"UserInfo.json"的文件中，实现了数据的转换和持久化存储。

在界面设计方面，该项目使用SwiftUI的核心特性和技术进行界面设计。采用声明式语法与组件化设计，实现了清晰、简洁的界面描述，便于维护和扩展。数据绑定和状态管理使得界面和数据之间的同步变得简单高效。同时，通过利用SwiftUI提供的丰富组件和布局系统，能够快速构建吸引人且具有响应性的用户界面。除此之外，该应用还实现了与系统自带软件的交互功能。通过点击按钮，应用能够唤起系统地图并导航至目标位置。这个功能依赖于MapKit框架中的MKPlacemark和MKMapItem类，用于表示地图上的特定地点和标注点。通过导入MapKit框架，应用与系统地图实现了无缝集成，为用户提供导航功能。

在功能方面，本项目主要由四大部分构成：主页、登录注册页、信息页和个人资料页。登录注册页提供滑动浏览欢迎页，负责处理用户登录和注册流程，包括输入验证和数据存储。主页有模糊搜索、地图导航和卡片介绍，以及分类介绍等功能。滑动页展示图片、标题和地图导航；模糊搜索功能则监听输入变化并根据输入内容展示匹配的文章列表；地图导航使用UIViewRepresentable协议，配合SwiftUI中的UIKit的MKMapView展示地图标注和中心位置；而卡片介绍则附带喜欢和评论的交互功能。信息页为用户提供表达意见、申请活动、入园码查询和权益维护等功能。最后，个人资料页则显示用户的个人信息，以及已访问和收藏的景点列表。

**关键词：**沙盒存储技术、SwiftUI、JSONEncoder、数据绑定、状态管理

目录

[摘 要 II](#_Toc272396831)

[第1章　绪 论 1](#_Toc1885886860)

[1.1 项目背景 1](#_Toc1389309947)

[1.2 市场现状 2](#_Toc542585398)

[1.3 论文结构安排 3](#_Toc1017219024)

[第2章　相关技术 4](#_Toc282822601)

[2.1 沙盒存储 4](#_Toc1018144196)

[2.2 json数据转换 8](#_Toc799802876)

[2.3 SwiftUI技术和特性 8](#_Toc1186790359)

[2.4 与系统自带软件的交互 9](#_Toc557450377)

[第3章 架构与设计思想 11](#_Toc1744818025)

[MVVM架构 11](#_Toc1267346390)

[3.1 ViewModel的示例：UserRegistrationViewModel.swift 11](#_Toc1547965784)

[3.2 Model示例：Comment.swift 13](#_Toc2044031930)

[3.3 View示例: MapView.swift 14](#_Toc748746451)

[第4章 系统功能设计实现 17](#_Toc2074914184)

[4.1 整体设计 17](#_Toc95746855)

[4.2 主页设计典型代码讲解 17](#_Toc752140382)

[4.3 信息页设计实现 25](#_Toc1134654032)

[4.4 “我的”界面设计 27](#_Toc475530464)

[第5章 系统功能实现 30](#_Toc1453857961)

[5.1 整体设计实现 30](#_Toc921814961)

[5.2 主页设计实现 31](#_Toc997020069)

[5.3 信息页设计实现 34](#_Toc101402142)

[5.4 “我的”界面设计实现 36](#_Toc1311268523)

[第6章　总结与展望 38](#_Toc1012880547)

[参考文献 39](#_Toc1730056501)

**第1章　绪 论**

## 1.1 项目背景

扬州，作为笔者的家乡，是一个诗一般的地方。它身怀古都荣耀，是隋唐的广陵郡，是明清的扬州府；它是文人墨客笔下的“烟花三月“，是“最春天“的江南城市。在众多新城拔地而起的当今，它守着完整的江南古韵，成为最适合居住的城市之一。它拥有丰富的历史文化和自然风光，因此吸引着大量的游客前来探索和欣赏。这座城市拥有悠久的历史，可以追溯到2500多年前的春秋战国时期。在历史上，扬州曾是中国重要的政治、经济和文化中心之一，留下了许多宝贵的文化遗产。

扬州以其精美的园林和古建筑而闻名，这些景点展示了中国传统园林艺术的精髓。其中最著名的是瘦西湖，它是一座典型的江南水乡园林，以其秀美的自然景色和独特的园林设计而备受赞誉。个园、何园、二十四桥等园林也是扬州不可错过的景点。扬州还有许多历史建筑和文化遗址，如扬州城墙、广陵故城和瓜洲古镇等。这些地方见证了扬州的辉煌历史，游客可以通过参观这些古老的建筑和遗址，深入了解扬州的文化和历史。

扬州被誉为中国的美食之乡。这里有许多独特的传统菜肴，如扬州炒饭、扬州炖鱼头、文昌鸡等，以其独特的口味和精致的烹饪技艺吸引了众多美食爱好者。晨曦徐徐拉开了帷幕，带着清新降临人间。古运河畔，伴着婉转的小调、清脆的鸟鸣，人们闲散地漫步于秀美古城，袅袅炊烟唤醒了沉睡的味蕾。“早上起来日己高，只觉心里闹潮潮，茶馆里头走一遭。拌干丝，风味糕，蟹壳黄，干层糕，翡翠烧卖，三丁包；清汤面，脆火烧，龙井茶叶香气飘。“

**Yangzhou APP**是一个旅游服务应用，旨在为游客提供全方位的扬州旅行体验。对于初次来到扬州的游客来说，他们会面临诸多问题和挑战，比如不了解景点的具体信息、如何最佳规划行程、如何有序参观并出示入园码以及如何提出意见或投诉等。Yangzhou APP可以帮助游客解决这些问题。通过Yangzhou APP，游客可以轻松获取各个景点的详细信息，包括开放时间、门票价格、交通路线等，帮助他们更好地了解每个景点的特色和历史背景。扬州app还提供了便捷的入园服务。游客可以在应用中查询门票，并获得电子入园号码。

## 1.2 市场现状

1.2.1 旅游业现状

在新型冠状病毒疫情结束后，扬州旅游业迎来了新的发展机遇和挑战。疫情期间，扬州采取了严格的防控措施，保障了游客和居民的生命安全。随着疫情逐渐得到控制，扬州旅游业逐渐恢复活力。首先，扬州积极推动旅游业的复苏和发展，通过加大宣传力度和市场推广，吸引了更多的游客。旅游景点和文化遗址如瘦西湖、个园、大明寺等重新开放，吸引了大量游客前来观光和游玩。其次，扬州加大了旅游基础设施的建设和改善，提升了游客的体验感。城市道路的改造和美化、公共交通的便利化以及酒店和景区的设施升级，使得扬州成为了更加宜人和舒适的旅游目的地。此外，扬州还加强了旅游业的多元化发展，推动了旅游与文化、体育、休闲等产业的融合。举办各类文化活动、体育赛事和特色节庆，提供了更多的旅游选择和体验项目，丰富了游客的行程。然而，面对疫情后的新常态，扬州旅游业也面临着挑战。旅游行业的竞争日益激烈，需要进一步提升服务质量和创新能力，以保持吸引力。同时，全球旅游市场的不确定性和旅游需求的变化，要求扬州积极调整策略，开拓新的市场和客源，以保持可持续发展。扬州旅游业在疫情结束后取得了积极的发展势头，但仍需要不断努力和创新，以适应新的形势和挑战，实现更加可持续和繁荣的发展。

1.2.2 现有官方软件分析“我的扬州APP”

“我的扬州APP”是一个市场上现有的综合性的官方软件，旨在为扬州市民提供各种生活服务。该应用集合了金融超市、电子医保、五险一金、信用额度、电子社保、健康商城、生活圈、房产专区、实时公交、健康扬州等多个功能板块，以满足市民在金融、医疗、购物、交通等方面的需求。

然而，目前该应用并没有针对外地游客提供的专门旅游服务板块。尽管扬州作为一个历史文化名城和旅游胜地，吸引了大量的游客，但在该应用中并未提供与旅游相关的功能和信息。游客在使用“我的扬州APP”时，无法获得针对旅游景点、交通路线、特色美食等方面的详细信息和指导。

鉴于扬州旅游业的发展潜力和市场需求，我的大作业“**Yangzhou APP**”专注于旅游服务板块。这样的板块可以包括扬州各个旅游景点的介绍、开放时间、门票价格等详细信息，以帮助游客更好地规划行程和了解景点特色。此外，应用还可以提供在线查询门票、搜索服务、特色推荐等功能，以提升游客的体验和满意度。

通过**Yangzhou APP**，扬州可以更好地满足游客的需求，提升作为旅游目的地的吸引力。此举也有助于推动扬州旅游业的发展，吸引更多游客前来游览、消费，从而促进经济增长和就业机会的增加。

第2章　相关技术

## 2.1 沙盒存储

2.1.1 技术概述

iOS 中的沙盒是一个较为独立和封闭的安全体系。每个应用程序在安装时会生成唯一一个文件夹。在该文件下存在着一个常见的媒体资源、代码、配置文件等。每个应用程序都只能被允许访问自己的沙盒，不能访问其它沙盒。下面是其主要沙盒存在的目录。

- 1.Documents 目录：应用程序主要和必备的数据文件写入到这个目录下。这个目录用于存储用户数据或其它应该定期备份的信息。用户自己保存的文件在documents文件里

- 2.AppName.app 目录：这是应用程序的程序包目录，包含应用程序的本身。由于应用程序必须经过签名，所以在运行时不能对这个目录中的内容进行修改，否则会使应用程序无法启动。

- 3.Library 目录：这个目录下有两个子目录：Caches 和 Preferences

Preferences 目录：包含应用程序的偏好设置文件。您不应该直接创建偏好设置文件，而是应该使用NSUserDefaults类来取得和设置应用程序的偏好.

Caches 目录：用于存放应用程序专用的支持文件，保存应用程序再次启动过程中需要的信息。

- 4 .temp 目录：这个目录用于存放临时文件，保存应用程序再次启动过程中不需要的信息。

以及最后一个非常不常见的NSBundle。该东西系AppName.app对外封装的一个隐蔽文件，但其内部是为普通的文件夹，即为AppName.app的内容。常使用如下方法获取到对象或文件：let path = Bundle.main.path(forResource: "lover", ofType: "mp3")

2.1.2 iOS沙盒特性

沙盒是每个应用的独立存储空间，应用不能越过沙盒去访问其他存储空间。所有的非代码文件，如图像、图标、声音、映像、属性列表和文本文件等，都需要保存在沙盒中。应用在沙盒中的存储目录包括bundle container和data container，后者用于存放应用数据和用户数据。data container会被细分为各个子目录来管理数据，同时应用还可以拥有额外的container，例如icloud container。每个应用请求的数据都要经过权限检查，只有符合条件的数据才会被许可访问。以下是一个官方开发文档中描述的应用沙盒目录的示意图。

2.1.3 iOS沙盒读存方法举例

**func** getDocumentsDirectory() -> URL {

**let** paths = FileManager.default.urls(for: .documentDirectory, in: .userDomainMask)

**return** paths[0]

}

getDocumentsDirectory函数用于获取文档目录的URL。该函数没有参数，并返回一个URL类型的值。在函数内部，通过调用FileManager.default.urls(for:in:)方法，传入.documentDirectory和.userDomainMask作为参数，获取文档目录的URL数组。然后，函数返回数组中的第一个元素作为文档目录的URL。

**func** getImageFromCache(str:String) -> Image {

**let** filename = getDocumentsDirectory().appendingPathComponent("\(str).jpg")

**let** imgData = **try**! Data(contentsOf: filename)

**return** Image(uiImage: UIImage(data: imgData)!)

        //self.getImage.append(Image(uiImage: t))

 }

getImageFromCache函数是用于从缓存中获取图像的函数。它接受一个名为str的字符串参数，并返回一个Image类型的值。函数内部通过指定的文件路径URL读取数据，并将其转换为UIImage对象，最后将其转换为Image对象，并作为函数的返回值。

**func** loadImageInToCache(str:String, uploadImg:UIImage){

**let** filename = getDocumentsDirectory().appendingPathComponent("\(str).jpg")

**if** **let** data = uploadImg.jpegData(compressionQuality: 0.8) {

**try**? data.write(to: filename)

    }

    print("\(str) has load into cache")

}

loadImageInToCache函数用于将图像加载到缓存中。它接受一个名为str的字符串参数用于标识图像，以及一个uploadImg参数表示要加载的图像。函数内部将uploadImg转换为JPEG数据，并将其写入指定的文件路径URL中，完成图像的缓存加载操作。最后会打印一条提示信息表示图像已成功加载到缓存中。

**func** initMyself() -> Bool{

    print("init sucess")

**let** sanboxfilename = getDocumentsDirectory().appendingPathComponent("Comment.json")

**if** !FileManager.default.fileExists(atPath: sanboxfilename.path) {

**for** s **in** commentData {

**let** imageNameStd = s.commentImageName

**let** path = Bundle.main.path(forResource: imageNameStd, ofType: "jpg")

**let** newImage = UIImage(contentsOfFile: path!)!

                loadImageInToCache(str: imageNameStd, uploadImg: newImage)

        }

    }

    print("init sucess")

    **return** **true**

}

initMyself函数用于进行初始化操作。它打印初始化成功的提示信息，检查是否存在名为"Comment.json"的文件。如果文件不存在，则遍历commentData数组，将其中的图像加载到缓存中。最后再次打印初始化成功的提示信息，并返回true表示初始化成功。

**func** load<T: Decodable>(\_ filename: String) -> T {

**let** data: Data

**var** flag=2

**if**(flag==1){

**guard** **let** file = Bundle.main.url(forResource: filename, withExtension: **nil**)

**else** {

            fatalError("Couldn't find \(filename) in main bundle.")

        }

**do** {

            data = **try** Data(contentsOf: file)

        } **catch** {

            fatalError("Couldn't load \(filename) from main bundle:\n\(error)")

        }

    }

**else** {

**let** sanboxfilename = getDocumentsDirectory().appendingPathComponent(filename)

**if** FileManager.default.fileExists(atPath: sanboxfilename.path) {

            data = **try**! Data.init(contentsOf: sanboxfilename)

        } **else** {

**guard** **let** file = Bundle.main.url(forResource: filename, withExtension: **nil**)

**else** {

                fatalError("Couldn't find \(filename) in main bundle.")

            }

**do** {

                data = **try** Data(contentsOf: file)

            } **catch** {

                fatalError("Couldn't load \(filename) from main bundle:\n\(error)")

            }

        }

    }

**do** {

**let** decoder = JSONDecoder()

**return** **try** decoder.decode(T.**self**, from: data)

    } **catch** {

        fatalError("Couldn't parse \(filename) as \(T.**self**):\n\(error)")

    }

}

load函数是一个通用的加载函数，用于加载指定文件的数据并进行解析。函数根据条件判断加载数据的来源，然后使用JSONDecoder进行解析，并返回解析后的结果。如果在加载或解析过程中出现错误，函数将抛出相应的错误信息。

**final** **class** ImageStore {

**typealias** \_ImageDictionary = [String: Image]

**fileprivate** **var** images: \_ImageDictionary = [:]

**fileprivate** **static** **var** scale = 2

**static** **var** shared = ImageStore()

**func** image(name: String) -> Image {

**let** index = \_guaranteeImage(name: name)

**return** images.values[index]

    }

**static** **func** loadImage(name: String) -> Image {

**return** getImageFromCache(str:name)

    }

**fileprivate** **func** \_guaranteeImage(name: String) -> \_ImageDictionary.Index {

        images[name] = ImageStore.loadImage(name: name)

**return** images.index(forKey: name)!

    }

}

ImageStore是一个存储和管理图像的单例类。它包含一个私有的图像字典images，用于存储图像对象。

通过调用image(name: String)方法，可以获取指定名称的图像对象。该方法会检查字典中是否已存在该图像，若不存在则从缓存中加载图像并存储到字典中，最后返回图像对象。

静态方法loadImage(name: String)用于从缓存中加载指定名称的图像，并返回加载的图像对象。

私有方法\_guaranteeImage(name: String)用于确保指定名称的图像存在于字典中。该方法检查图像是否已存在于字典中，若不存在则从缓存中加载图像并存储到字典中，最后返回图像对象的索引。

## 2.2 json数据转换

**do**  {

            // 将player对象encod（编码）

**let** data: Data = **try** encoder.encode(**self**.userData.users)

**let** filename = getDocumentsDirectory().appendingPathComponent("UserInfo.json")

**try**? data.write(to: filename)

        } **catch** {

        }

以上代码块将self.userData.users对象进行编码，并将编码后的数据写入名为"UserInfo.json"的文件中，该文件位于应用的文档目录中。在编码和写入数据的过程中，通过try?进行异常处理，以便忽略任何错误。

具体地，在do语句块中，首先使用encoder.encode(self.userData.users)对self.userData.users对象进行编码，并将编码后的数据存储在data常量中。

接下来，通过getDocumentsDirectory()函数获取文档目录的URL，并使用appendingPathComponent(\_:)方法将文件名"UserInfo.json"连接到URL中，形成完整的文件路径URL，并将其赋值给filename常量。

最后，使用data.write(to:)方法将数据写入指定的文件路径URL中。

**2.3 SwiftUI技术和特性**

SwiftUI 是 Apple 在 2019 年 WWDC 大会上发布的一套全新的应用开发框架，用于构建所有 Apple 平台的应用界面。这套框架基于 Swift 语言开发，整合了 UIKit、AppKit 和 WatchKit 等框架的一些优点，同时引入了一些全新的设计思路和技术。

SwiftUI 使用声明式语法，这使得代码更简洁易读，更易于理解和维护。在声明式语法中，我只需要定义用户界面应该做什么，而不是如何做。例如，我可以告诉 SwiftUI 我希望文本视图在界面上的位置，以及它应该如何看起来，然后 SwiftUI 就会处理大部分细节。

此外，SwiftUI 提供了一种全新的方式来设计和构建用户界面：一切皆组件。在 SwiftUI 中，所有的视图，包括按钮、标签、滑块等都是组件。甚至应用的主界面也是一个组件。这种设计使得代码更加模块化，更易于复用和测试。

本项目使用了一些SwiftUI 的核心特性和技术，以ContentView为例子：

**声明式语法与组件化设计**：在 `ContentView` 和 `test` 这两个视图中，我们可以看到 SwiftUI 的声明式语法和组件化设计。在 `ContentView` 中，我声明了两种不同的视图状态：用户已登录和未登录，然后通过 `if-else` 语句动态选择显示哪个视图。在 `test` 视图中，我使用了很多内置的 SwiftUI 视图，如 `VStack`, `TabView`, `ForEach`, `Image`, `Text` 等，并且用自己定义的 `FormField` 和 `RequirementText` 视图来封装一些复杂的界面。这些都充分展示了 SwiftUI 的声明式语法和组件化设计思想。

**数据绑定和状态管理**：在 SwiftUI 中，数据是一等公民。在我的代码中，有许多 `@State`, `@ObservedObject`, `@EnvironmentObject` 这样的属性修饰符，这些都是 SwiftUI 的数据绑定和状态管理机制。比如在 `test` 视图中，我将 `UserregistrationViewModel` 设置为 `@ObservedObject`，然后通过 `$userRistrantionViewModel.name` 和 `$userRistrantionViewModel.password` 将数据绑定到了表单字段上，这样当用户输入信息时，ViewModel 中的数据也会自动更新。

**用户交互和逻辑处理**：在 `test` 视图中，我使用了 `Button` 视图来处理用户的点击事件。在 SwiftUI 中，处理用户交互非常直接和简洁，我只需要在 `Button` 的闭包中编写对应的逻辑就可以了。在我的代码中，我在按钮的点击事件中进行了登录和注册的逻辑处理，并通过 `alert` 视图展示了的错误信息。

**样式和布局**：在 SwiftUI 中，视图的样式和布局都是通过修饰符来实现的。在我的代码中，我使用了很多如 `.font`, `.foregroundColor`, `.bold`, `.padding`, `.frame`, `.background`, `.cornerRadius` 等修饰符来调整视图的样式和布局。

## 2.4 与系统自带软件的交互

// ...

// 打开内置地图应用并导航到目标位置

**let** placemark = MKPlacemark(coordinate: **self**.coordinate)

**let** mapItem = MKMapItem(placemark: placemark)

                        mapItem.name = article.title

                        mapItem.openInMaps(launchOptions: [MKLaunchOptionsDirectionsModeKey : MKLaunchOptionsDirectionsModeDriving])

                    }) {

                        Text("导航")

                            .foregroundColor(.white)

                            .padding()

                            .background(Color.green)

                            .cornerRadius(10)

                    }

                    .padding()

// ...

通过点击按钮来唤起系统地图并导航到目标位置。具体的操作如下：首先，根据article的ID从locations字典中获取目标位置的经纬度，并创建一个MKPlacemark对象来表示该位置。然后，创建一个MKMapItem对象，将其初始化为包含上一步创建的MKPlacemark的地点，并设置name属性为article.title。最后，调用mapItem.openInMaps(launchOptions:)方法，系统将打开内置的地图应用，并根据提供的目标位置导航信息进行导航，使用驾车导航模式。

这个操作涉及MKPlacemark和MKMapItem，它们是MapKit框架中的类，用于表示地图上的特定地点和标注点。通过导入MapKit框架，可以使用这些类来实现唤起系统地图并进行导航的功能。

第3章 架构与设计思想

## MVVM架构

架构目标是使视图和模型解耦，使视图的代码更加清晰，易于测试，并且更好地支持响应式编程。这是一个基于MVVM（Model-View-ViewModel）架构设计的项目。MVVM是一种用于解耦视图（用户界面）和模型（数据）的设计模式，它在SwiftUI中被广泛使用。

在这个项目中，我设计了以下几个核心部分：

1. 模型（Model）：模型部分位于“Models”文件夹中，包括了Article、Comment、Issue等Swift文件，和对应的json文件。这些模型负责存储核心的应用程序数据。例如，Article模型包括文章的标题，内容，作者等信息。

2. 视图（View）：视图部分位于“Views”文件夹中，包括了ProfileView、ArticleView、CommentView等Swift文件。这些视图定义了用户界面的显示方式。例如，ArticleView会显示文章的标题，内容，作者等信息。

3. 视图模型（ViewModel）：在我的项目中，我创建了一个UserRegistrationViewModel.swift文件，这是视图模型的一部分。视图模型负责处理用户输入，并根据这些输入更新模型。例如，UserRegistrationViewModel处理用户在注册表单中输入的信息，并将这些信息更新到User模型中。

通过MVVM架构，我实现了视图和模型的解耦。当模型的状态改变时，这些改变会自动反映在视图上。例如，当用户在注册表单中输入信息时，UserRegistrationViewModel会立即获取这些信息，并更新User模型。然后，这些更改会自动反映在视图上，例如在一个用户信息的概要视图中。

## 3.1 ViewModel的示例：UserRegistrationViewModel.swift

以下这段代码说明了MVVM中的ViewModel是如何工作的，它监听用户的输入，处理逻辑，更新模型，然后将结果通知给视图。

**import** Foundation

**import** Combine

**class** UserregistrationViewModel:ObservableObject {

    // 输入

    @Published **var** name = ""

    @Published **var** password = ""

    @Published **var** passwordConfirm = ""

    // 输出

    @Published **var** isNameLengthVaild = **false**

    @Published **var** isPasswordLengthVaild = **false**

    @Published **var** isPasswordCapitalLetter = **false**

    @Published **var** isPasswordComfirmValid = **false**

**private** **var** cancellabelSet:Set<AnyCancellable> = []

**init**() {

        $name

            .receive(on: RunLoop.main)

            .map { name **in**

**return** name.count >= 4

            }

            .assign(to: \.isNameLengthVaild, on: **self**)

            .store(in: &cancellabelSet)

        $password

            .receive(on: RunLoop.main)

            .map { password **in**

**return** password.count >= 8

            }

            .assign(to: \.isPasswordLengthVaild, on: **self**)

            .store(in: &cancellabelSet)

        $password

            .receive(on: RunLoop.main)

            .map { password **in**

**let** pattern = "[A-Z]"

**if** **let** \_ = password.range(of: pattern, options: .regularExpression) {

                    **return** **true**

                } **else** {

                    **return** **false**

                }

            }

            .assign(to: \.isPasswordCapitalLetter, on: **self**)

            .store(in: &cancellabelSet)

        Publishers.CombineLatest($password, $passwordConfirm)

            .receive(on: RunLoop.main)

            .map { (password, passwordConfirm) **in**

**return** !passwordConfirm.isEmpty && (passwordConfirm == password)

            }

            .assign(to: \.isPasswordComfirmValid, on: **self**)

            .store(in: &cancellabelSet)

    }

}

UserRegistrationViewModel类是一个视图模型，它在SwiftUI中作为“真实数据”的代理（或者说是数据的包装器）。这个类继承了ObservableObject，这是一种SwiftUI的特性，它允许视图监听到模型中的数据变化。

代码中，我定义了几个输入属性，例如name，password和passwordConfirm，这些属性都被标记为@Published。这意味着一旦这些值发生变化，所有依赖于这些值的视图都将被更新。换句话说，当用户在文本框中输入他们的姓名，密码或确认密码时，这些值将被更新，并触发相关视图的更新。

我也定义了一些输出属性，例如isNameLengthValid，isPasswordLengthValid，isPasswordCapitalLetter和isPasswordConfirmValid。这些属性是基于输入属性的结果，它们在输入属性发生变化时进行计算。

例如，isNameLengthValid是通过检查name属性的长度是否大于或等于4来计算的。类似地，isPasswordLengthValid检查password的长度是否大于或等于8，isPasswordCapitalLetter检查password是否包含大写字母，isPasswordConfirmValid检查passwordConfirm是否与password相同。

我使用Combine框架的Publisher和Subscriber模型来处理这些计算。这种模型允许我在数据发生变化时执行操作，例如计算输出属性的值。

当我在初始化方法中设置这些Publisher时，我将每个Publisher的结果指派（assign）给对应的输出属性，并将这个订阅关系存储在cancellableSet中。这是因为Publisher创建的订阅关系是需要被管理的，如果不保存这个订阅关系，当订阅关系超出作用域时，它就会被取消，这样就无法收到后续的值更新。

## 3.2 Model示例：Comment.swift

模型是应用程序的“真实世界”，它代表了应用程序中的数据以及相关的业务逻辑。在MVVM设计模式中，模型并不知道视图和视图模型的存在，它只关注数据和数据的处理。下面的Comment.swift文件是一个模型的示例，模型在MVVM设计模式中负责代表应用程序的数据。

**import** Foundation

**import** SwiftUI

**import** CoreLocation

**struct** Comment:Hashable, Codable, Identifiable {

**var** id: Int

**var** userId: Int

**var** articleId: Int

**var** content: String

**var** commentImageName: String

**init**(id: Int, userId:Int, articleId: Int, content: String, commentImageName: String) {

**self**.userId = userId

**self**.id = id

**self**.articleId = articleId

**self**.content = content

**self**.commentImageName = commentImageName

    }

}

**extension** Comment {

**var** commentImage: Image {

**return** ImageStore.shared.image(name:commentImageName)

    }

}

在这个例子中，Comment结构体代表了一条评论。Comment结构体遵循了Hashable，Codable，和Identifiable协议。这些协议分别使评论可以被哈希（以便在集合中进行快速查找），可以被编码和解码（以便进行数据持久化或网络传输），以及具有一个唯一标识符（以便在列表中进行唯一识别）。

Comment结构体有五个属性：id，userId，articleId，content，和commentImageName。这些属性分别代表评论的唯一标识符，评论者的用户标识符，被评论的文章标识符，评论的内容，和评论图片的名称。

此外，Comment结构体还有一个计算属性commentImage，这个属性使用了ImageStore类的shared实例来根据commentImageName获取相应的图片。这就是一个模型如何封装数据和相关的业务逻辑的例子。

## 3.3 View示例: MapView.swift

MapView是视图在MVVM设计模式中的例子。它不直接与模型进行交互，而是通过绑定的坐标来获取和更新数据。它使用UIViewRepresentable协议来创建和管理UIKit的视图，以此达到在SwiftUI中使用UIKit视图的目的。

下面这段代码展示了一个在MVVM设计模式中的视图例子：MapView。

**import** SwiftUI

**import** MapKit

**struct** MapView: UIViewRepresentable {

    @Binding **var** coordinate: CLLocationCoordinate2D

**func** makeCoordinator() -> Coordinator {

        Coordinator(**self**)

    }

**func** makeUIView(context: Context) -> MKMapView {

**let** mapView = MKMapView()

        mapView.delegate = context.coordinator

**return** mapView

    }

**func** updateUIView(\_ uiView: MKMapView, context: Context) {

        uiView.setCenter(coordinate, animated: **true**)

**let** annotation = MKPointAnnotation()

        annotation.coordinate = coordinate

        uiView.addAnnotation(annotation)

    }

**class** Coordinator: NSObject, MKMapViewDelegate {

**var** parent: MapView

**init**(\_ parent: MapView) {

**self**.parent = parent

        }

    }

}

在这个例子中，MapView结构体代表了一个地图视图。在SwiftUI中，视图是用户界面的基本组成单元，它定义了应用程序中用户看到并与之交互的部分。视图在MVVM设计模式中并不直接与模型进行交互，而是通过视图模型来获取和更新数据。

MapView遵循了UIViewRepresentable协议，这意味着它可以使用UIKit的MKMapView类来创建和管理地图视图。@Binding var coordinate是MapView的输入，它代表了地图中心的坐标。由于它是绑定的，当坐标发生改变时，MapView会被重新渲染。

makeCoordinator()方法创建并返回一个Coordinator实例，该实例是MapView的代理，负责处理和响应地图视图的事件。

makeUIView(context: Context)方法创建并返回一个MKMapView实例，并将其代理设置为上下文的协调器。这是创建视图的过程。

updateUIView(\_ uiView: MKMapView, context: Context)方法负责更新视图的状态。当绑定的坐标发生改变时，它会设置地图的中心，并在地图上添加一个标记。这是视图在响应数据变化时如何更新自身的例子。

Coordinator类是MapView的代理，它继承了NSObject并遵循了MKMapViewDelegate协议，用于处理和响应地图视图的事

第4章 系统功能设计实现

## 4.1 整体设计

## 4.2 主页设计典型代码讲解

**滑动浏览欢迎页**

ForEach(loadingData) { page **in**

                    GeometryReader { g **in**

                        VStack {

                            Spacer()

                            // 设置图片

                            Image(page.image)

                                .resizable()

                                .scaledToFit()

                            // 设置标题文字

                            Text(page.title)

                                .font(.title).bold()

                                .padding(/\*@START\_MENU\_TOKEN@\*/.all/\*@END\_MENU\_TOKEN@\*/, 20)

                            // 设置字体

                            Text(page.descrip)

                                .multilineTextAlignment(.center)

                                .padding(/\*@START\_MENU\_TOKEN@\*/.all/\*@END\_MENU\_TOKEN@\*/, /\*@START\_MENU\_TOKEN@\*/10/\*@END\_MENU\_TOKEN@\*/)

                            Spacer()

                        }

                        .opacity(Double(g.frame(in : . global).minX)/200+1)

                    }

                }

**登录注册页**

**let** onLoginFinished: (Bool) -> Void

@ObservedObject **private** **var** userRistrantionViewModel = UserregistrationViewModel()

@State **private** **var** showAlert = **false**

@EnvironmentObject **var** userData: UserData

@State **private** **var** registerAlert = **false**

**var** body: **some** View {

    VStack {

        TabView {

            ForEach(loadingData) { page **in**

                // ...

            }

            VStack {

                // ...

                Button(action: {

**let** tmp = **self**.userData.users.filter { $0.name == userRistrantionViewModel.name }

**if** tmp.count != 1 {

**self**.showAlert = **true**

                    } **else** **if** tmp[0].password != userRistrantionViewModel.password {

**self**.showAlert = **true**

                    } **else** {

**self**.userData.userId = tmp[0].id

                        onLoginFinished(**true**)

                    }

                },

                label: {

                    // ...

                })

                .alert(isPresented: $showAlert) {

                    Alert(title: Text("登录失败"), message: Text("用户名或密码不对，请检查你输入的用户名和密码"))

                }

                Button(action: {

**let** tmp = **self**.userData.users.filter { $0.name == userRistrantionViewModel.name }

**if** tmp.count == 1 {

**self**.registerAlert = **true**

                    } **else** **if** userRistrantionViewModel.name == "" {

**self**.registerAlert = **true**

                    } **else** {

**self**.userData.userId = **self**.userData.users.count

**self**.userData.users.append(User(id: **self**.userData.users.count, number: "aaa", password: userRistrantionViewModel.password, role: 1, name: userRistrantionViewModel.name, progress: ""))

                        // Save user data to a file

**let** encoder = JSONEncoder()

**do** {

**let** data: Data = **try** encoder.encode(**self**.userData.users)

**let** filename = getDocumentsDirectory().appendingPathComponent("UserInfo.json")

**try**? data.write(to: filename)

                        } **catch** {

                            // Handle error

                        }

                    }

                },

                label: {

                    // ...

                })

                .alert(isPresented: $registerAlert) {

                    Alert(title: Text("注册失败"), message: Text("用户名重复或用户名不能为空"))

                }

                Spacer()

            }

        }

        .edgesIgnoringSafeArea(.top)

        .tabViewStyle(PageTabViewStyle(indexDisplayMode: .always))

    }

}

@ObservedObject private var userRistrantionViewModel = UserregistrationViewModel()是视图模型。这个视图模型维护着用户注册的状态，包括用户的姓名和密码。使用ObservedObject修饰符，这个视图可以监听视图模型中数据的变化，并在数据发生变化时重新渲染视图。

接下来是两个@State私有变量：showAlert和registerAlert，它们用来控制警告框的显示。State是SwiftUI的一种属性封装器，它可以在视图的多次渲染中保留值，并在值发生变化时触发视图的更新。

@EnvironmentObject var userData: UserData是环境对象，它在应用程序的多个视图之间共享数据。在这个例子中，userData存储了用户的信息。

在视图的主体中，有两个按钮，一个用于登录，一个用于注册。这两个按钮都绑定了一些动作，当用户点击按钮时，这些动作将会被执行。例如，登录按钮会检查用户输入的用户名和密码是否正确，如果不正确，它将显示一个警告框。注册按钮则会检查用户名是否已经存在或者是否为空，如果满足其中任何一种情况，它也将显示一个警告框。

最后，这个视图使用了TabView和ForEach来显示多个页面，每个页面都有各自的内容和行为。

**模糊搜索**

@Binding **var** search: String

@State **private** **var** isSearching = **false**

@State **private** **var** filteredArticles: [Article] = []

@State **private** **var** searchSuggestions: [Article] = []

@State **private** **var** showSuggestions = **false**

// ...

TextField("Search...", text: $search)

    .font(.title3)

    .onChange(of: search) { value **in**

**if** !value.isEmpty {

            searchSuggestions = articleData.filter { $0.title.lowercased().contains(value.lowercased()) }

        } **else** {

            searchSuggestions = []

        }

    }

Button(action: {

**self**.filteredArticles = searchSuggestions

    isSearching = **true**

}) {

    Image(systemName: "magnifyingglass")

        .foregroundColor(.gray)

        .font(.title)

}

// ...

.sheet(isPresented: $isSearching) {

    ArticleListView(data: filteredArticles).environmentObject(UserData())

        .padding(.top, 50)

}

模糊搜索是基于一个名为 SearchBar 的 View 结构体实现的，该结构体里包含了一些状态变量和搜索框。

这个搜索框通过 TextField 实现，并且设定了当其文本变化时的回调函数。具体来说，每次 search 变量的值改变，就会触发该回调函数。这个回调函数的主要功能是更新搜索建议。

搜索建议的生成基于 articleData 数据集，这是一个 Article 类型的数组，每个 Article 包含一个标题。如果 search 变量的值（即用户输入的搜索关键字）在某个文章的标题中出现，那么这个文章就会被加入到搜索建议中。这是通过 articleData.filter 方法实现的。

这个搜索框旁边还有一个放大镜按钮，用户点击它后会触发另一个回调函数。这个回调函数会将当前的搜索建议赋值给 filteredArticles 变量，并将 isSearching 变量设为 true。

这个界面下方还包含一个列表，用于展示搜索建议。这个列表显示的是 searchSuggestions 变量的内容，也就是当前的搜索建议。当用户点击某个建议时，搜索框的内容会变为这个建议的标题。

最后，如果 isSearching 变量的值为 true，会打开一个新的 View，显示 filteredArticles 变量的内容，也就是当前的搜索结果。这个新的 View 是 ArticleListView 类型的，可以显示一个 Article 类型的数组。

**地图导航**

**import** SwiftUI

**import** MapKit

**struct** MapView: UIViewRepresentable {

    @Binding **var** coordinate: CLLocationCoordinate2D

**func** makeCoordinator() -> Coordinator {

        Coordinator(**self**)

    }

**func** makeUIView(context: Context) -> MKMapView {

**let** mapView = MKMapView()

        mapView.delegate = context.coordinator

**return** mapView

    }

**func** updateUIView(\_ uiView: MKMapView, context: Context) {

        uiView.setCenter(coordinate, animated: **true**)

**let** annotation = MKPointAnnotation()

        annotation.coordinate = coordinate

        uiView.addAnnotation(annotation)

    }

**class** Coordinator: NSObject, MKMapViewDelegate {

**var** parent: MapView

**init**(\_ parent: MapView) {

**self**.parent = parent

        }

    }

}

**评论功能**

@Environment(\.presentationMode) **var** presentationMode

@EnvironmentObject **var** userData: UserData

@State **var** card: Article

@State **var** content = ""

@State **var** secondSheet = **false**

@State **var** getImage = [Image]()

@State **var** showAlert = **false**

@State **var** getUIImage = [UIImage]()

@State **var** y = 0

@State **var** releaseAlert = **false**

**let** onCommentFinished: (String) -> Void

**func** loadImageInToCache(str: String, uploadImg: UIImage) {

**let** filename = getDocumentsDirectory().appendingPathComponent(str + ".jpg")

**if** **let** data = uploadImg.jpegData(compressionQuality: 0.8) {

**try**? data.write(to: filename)

    }

    print("\(filename) has loaded")

}

**func** upload() {

**let** imgName = "comment\(**self**.userData.comments.count)"

    loadImageInToCache(str: imgName, uploadImg: getUIImage[0])

**let** t = Comment(id: **self**.userData.comments.count, userId: **self**.userData.userId, articleId: card.id, content: **self**.content, commentImageName: imgName)

**self**.userData.comments.append(t)

**let** encoder = JSONEncoder()

**do** {

**let** data: Data = **try** encoder.encode(**self**.userData.comments)

**let** filename = getDocumentsDirectory().appendingPathComponent("Comment.json")

**try**? data.write(to: filename)

    } **catch** {

        // Handle error

    }

**self**.onCommentFinished(**self**.content)

**self**.presentationMode.wrappedValue.dismiss()

}

**var** body: **some** View {

    VStack {

        Button(action: {

**self**.releaseAlert = **true**

        }) {

            HStack {

                Image(systemName: "paperplane").font(.title)

                Text("提交")

            }

        }

        .alert(isPresented: **self**.$releaseAlert) {

**if** !**self**.getImage.isEmpty {

**return** Alert(

                    title: Text("确定发布？"),

                    primaryButton: .default(

                        Text("继续编辑")

                    ),

                    secondaryButton: .destructive(

                        Text("立即发布"),

                        action: {

**self**.upload()

                        }

                    )

                )

            } **else** {

**return** Alert(

                    title: Text("图片不能为空"),

                    message: Text("请添加图片")

                )

            }

        }

        TextField("评论", text: **self**.$content)

            .padding(20)

        VStack {

            // ...

        }

        Spacer()

    }

}

这个代码片段是一个更复杂的SwiftUI视图，它包含一个可以用于发表评论的表单。我们可以看到在@EnvironmentObject中使用了userData，这是一个跨视图共享的数据源。

@State变量包含了被视图使用的各种状态。例如，content变量用于存储评论的内容，而secondSheet，showAlert，releaseAlert则用于控制某些操作的执行和视图的显示。getUIImage数组用于存储用户选择的图片，y变量可能用于控制视图滚动的位置。

loadImageIntoCache函数用于将图片存储到缓存中，upload函数负责上传评论，包括文本内容和图片。在上传函数中，首先将图片存入缓存，然后创建一个新的评论，将其添加到userData的评论列表中，并将评论列表写入到一个名为Comment.json的文件中。

在视图的主体中，首先是一个按钮，点击这个按钮会显示一个警告框，询问用户是否确定发布评论。如果用户已经选择了图片，那么警告框将会有两个选项：一个是继续编辑，另一个是立即发布，点击立即发布将会触发上传函数。如果用户没有选择图片，警告框将会提示用户添加图片。

然后是一个文本输入框，用于输入评论的内容。最后，有一个VStack，可能用于显示用户选择的图片。

这个视图展示了在SwiftUI中如何使用表单来收集用户输入的信息，以及如何将用户输入的信息上传并保存到本地。这是在SwiftUI中创建交互式用户界面的一个很好的例子。

唤起系统相册照片选择的api，在评论中加入图片

struct ImagePicker: UIViewControllerRepresentable {

    @Environment(\.presentationMode) private var presentationMode

    let sourceType: UIImagePickerController.SourceType

    let onImagePicked: (UIImage) -> Void

    final class Coordinator: NSObject, UINavigationControllerDelegate, UIImagePickerControllerDelegate {

        @Binding private var presentationMode: PresentationMode

        private let sourceType: UIImagePickerController.SourceType

        private let onImagePicked: (UIImage) -> Void

        init(presentationMode: Binding<PresentationMode>,

             sourceType: UIImagePickerController.SourceType,

             onImagePicked: @escaping (UIImage) -> Void) {

            \_presentationMode = presentationMode

            self.sourceType = sourceType

            self.onImagePicked = onImagePicked

        }

        func imagePickerController(\_ picker: UIImagePickerController, didFinishPickingMediaWithInfo info: [UIImagePickerController.InfoKey : Any]) {

            let uiImage = info[UIImagePickerController.InfoKey.originalImage] as! UIImage

            onImagePicked(uiImage)

            presentationMode.dismiss()

        }

        func imagePickerControllerDidCancel(\_ picker: UIImagePickerController) {

            presentationMode.dismiss()

        }

    }

    func makeCoordinator() -> Coordinator {

        return Coordinator(presentationMode: presentationMode,

                           sourceType: sourceType,

                           onImagePicked: onImagePicked)

    }

    func makeUIViewController(context: UIViewControllerRepresentableContext<ImagePicker>) -> UIImagePickerController {

        let picker = UIImagePickerController()

        picker.sourceType = sourceType

        picker.delegate = context.coordinator

        return picker

    }

    func updateUIViewController(\_ uiViewController: UIImagePickerController,

                                context: UIViewControllerRepresentableContext<ImagePicker>) {

    }

}

ImagePicker是一个可以唤起系统相册照片选择器的SwiftUI视图，并在用户选择照片后将其传递给onImagePicked闭包。它使用了Coordinator类来管理委托方法并响应用户选择的操作。

ImagePicker包含了一个final class Coordinator，它是一个NSObject并实现了UINavigationControllerDelegate和UIImagePickerControllerDelegate协议。Coordinator类的作用是为ImagePicker视图提供委托方法，以响应用户选择的照片和取消选择的操作。在Coordinator类中，选择照片时，imagePickerController(:didFinishPickingMediaWithInfo:)方法会被调用，并将选择的照片传递给onImagePicked闭包。当用户取消选择时，imagePickerControllerDidCancel(:)方法会被调用，并关闭相册选择器。

makeCoordinator()方法返回一个Coordinator实例，用于管理ImagePicker视图的生命周期和响应用户选择的照片。makeUIViewController(context:)方法创建一个UIImagePickerController实例，并将其设置为sourceType类型，同时将其委托给Coordinator实例。updateUIViewController(\_:context:)方法不进行任何操作，因为ImagePicker视图不需要更新。

## 4.3 信息页设计实现

**提出建议**

Button(

                action: { **self**.suggestionIssue = **true** }, label: {

                    VStack {

                        Image("suggestion")

                            .resizable()

                            .aspectRatio(contentMode: .fit)

                            .frame(width: 80)

                            .clipped()

                        Text("表达意见")

                            .foregroundColor(Color.black.opacity(0.85))

                        .font(.system(size: 17))}

                    .frame(width: 130, height: 130)

                    .overlay(RoundedRectangle(cornerRadius:10).stroke(Color.gray, style: StrokeStyle(lineWidth: 1, dash: [10])))})

            .sheet(isPresented: **self**.$suggestionIssue,content:

                    { WriteIssueView { titleText,content **in**

**self**.userData.issues.insert(Issue(id: **self**.userData.issues.count, type: 1, title: titleText, content: content), at: 0)

**let** encoder = JSONEncoder()

**do**  {

                    // 将player对象encod（编码）

**let** data: Data = **try** encoder.encode(**self**.userData.issues)

**let** filename = getDocumentsDirectory().appendingPathComponent("Issue.json")

**try**? data.write(to: filename)

                } **catch** {

                }

            }})

**查询入园码**

**struct** CodeView: View {

    // ...

**var** body: **some** View {

        NavigationView {

            Form {

                Section(header: Text("身份证号")) {

                    TextField("请输入身份证号", text: $personID)

                }

                Section(header: Text("姓名")) {

                    TextField("请输入姓名", text: $personName)

                }

                Section(header: Text("入园码查询")) {

                    Text("\(result)")

                }

                Section {

                    Button(action: {

**let** u = **self**.userData.users.filter { $0.name == **self**.personName }

**if** u.isEmpty {

**self**.nullAlert = **true**

                        } **else** {

**let** t = u.filter { $0.number == **self**.personID }

**if** t.isEmpty {

**self**.nullAlert = **true**

                            } **else** {

**self**.result = t[0].progress

                            }

                        }

                    }) {

                        HStack {

                            Image(systemName: "magnifyingglass")

                            Text("查询")

                                .bold()

                        }

                        // ...

                    }

                    .alert(isPresented: $nullAlert) {

                        Alert(title: Text("查询失败"), message: Text("身份证号或 姓名错误"))

                    }

                }

                // ...

            }

            .navigationBarTitle(Text("查询进度"), displayMode: .inline)

        }

    }

}

其余界面功能类似提出建议，故不再赘述。

## 4.4 “我的”界面设计

**struct** ProfileView: View {

    @EnvironmentObject **var** userData: UserData

    @State **var** hero1 = **false**

    @State **var** hero2 = **false**

**func** refresh() {

**let** encoder = JSONEncoder()

**do** {

**let** data: Data = **try** encoder.encode(**self**.userData.issues)

**let** filename = getDocumentsDirectory().appendingPathComponent("Issue.json")

**try**? data.write(to: filename)

        } **catch** {

            // Handle encoding error

        }

        // ...

**do** {

**let** data: Data = **try** encoder.encode(**self**.userData.prefers)

**let** filename = getDocumentsDirectory().appendingPathComponent("Atricle.json")

**try**? data.write(to: filename)

        } **catch** {

            // Handle encoding error

        }

    }

**var** userName: String {

**self**.userData.users.filter { $0.id == userData.userId }[0].name

    }

**var** body: **some** View {

        VStack {

            ScrollView(.vertical, showsIndicators: **false**) {

                HStack {

                    Text("    Hi," + userName)

                        .bold()

                        .font(.largeTitle)

                        .multilineTextAlignment(.center)

                        .foregroundColor(.blue)

                        .padding(.leading, 20)

                        .padding(.leading, -40)

                    Spacer()

                }

                // ...

                HArticleListView(data: **self**.userData.articles.filter { **self**.userData.prefers[$0.id].likedUserId.contains(**self**.userData.userId) }) { i **in**

                    hero1.toggle()

                }

                .opacity((**self**.hero2) ? 0 : 1)

                .zIndex(5.0)

                // ...

                HArticleListView2(data: **self**.userData.articles.filter { **self**.userData.prefers[$0.id].staredUserId.contains(**self**.userData.userId) }) { i **in**

                    hero2.toggle()

                }

                .opacity((**self**.hero1) ? 0 : 1)

                .zIndex(5.0)

            }

        }

    }

}

这个页面是一个使用SwiftUI构建的用户个人页面。它主要分为两部分：在用户信息部分显示用户名称，在用户行为部分展示了用户的喜欢（心形符号表示）和想去（星形符号表示）的文章。现在，我们将详细解析这个页面的代码。

1.定义视图结构和数据 我们首先定义了一个ProfileView，这是一个遵循SwiftUI中View协议的结构体。这个View中，通过@EnvironmentObject属性包装器，我们引用了userData，它是一个用户数据的环境对象。然后，我们定义了两个State变量hero1和hero2，它们将被用于追踪用户的一些操作。

2.refresh方法 refresh方法中，我们用JSONEncoder对象将userData中的issues，comments，users，和prefers编码成JSON数据，并写入到对应的.json文件中。这是一个将数据持久化的过程。

3.userName属性 userName属性会返回当前用户的名字，通过过滤userData.users以找到当前用户的ID，然后返回其名称。

4.body属性 body属性定义了页面的UI布局。在这个页面中，最外层是一个垂直的VStack，然后是一个ScrollView，其中包含用户的个人信息和他们的行为信息。

用户个人信息部分：显示了"Hi," + 用户名的文本，文本颜色为蓝色。

用户行为部分：有两个部分，一个显示用户喜欢的文章（心形符号），另一个显示用户想去的文章（星形符号）。在每个部分下，都有一个列表显示了对应的文章。当用户点击列表中的某个文章时，对应的hero状态会被切换，同时其他部分的透明度会被设为0，以实现隐藏其他部分的效果。

这个页面在UI设计上相对简洁，便于用户查看和操作他们的信息。同时，也充分考虑了数据的持久化，使得数据在应用重新启动后仍能保留。该

第5章 系统功能实现

## 5.1 整体设计实现

5.1.1 滑动浏览欢迎页

|  |  |  |  |
| --- | --- | --- | --- |
| 登录-1（拖移项目） | 登录-7（拖移项目） | 登录-18（拖移项目） | Simulator Screenshot - iPhone 14 Pro - 2023-06-08 at 09.42.29 |

5.1.2 登录注册页

而后进入登陆与注册页面。对于登录，输入用户名和密码后。用户名栏下方有提示字样”建议至少4个字符“，当该栏输入字符不足4个时，会显示蓝紫色字样；当该栏输入字符大于等于4时，该字样会变灰并被删除线标识。对于密码栏同理，有两个提示字样分别提示至少输入8字符和1大写字符，满足条件字样会变 灰并被删除线标识。

*备注：这一功能借鉴了项目https://github.com/SwiftUIX/SwiftUIX.git*

|  |  |  |  |
| --- | --- | --- | --- |
| Simulator Screenshot - iPhone 14 Pro - 2023-06-08 at 09.48.36 | Simulator Screenshot - iPhone 14 Pro - 2023-06-08 at 09.48.43 | Simulator Screenshot - iPhone 14 Pro - 2023-06-08 at 09.48.56 | Simulator Screenshot - iPhone 14 Pro - 2023-06-08 at 09.42.46 |

用户名和密码都输入后点击登录，若用户名或密码错误会向下图最后一个展示的那样弹出报错提示，否则会正常进入系统。对于注册，输入用户名和密码后点击注册即可注册并自动登录，如果用户名已被注册会弹出如下方所示提示。

|  |  |
| --- | --- |
| Simulator Screenshot - iPhone 14 Pro - 2023-06-08 at 10.05.10 | Simulator Screenshot - iPhone 14 Pro - 2023-06-08 at 10.05.54 |

进入软件后，下方导航栏有”主页“”信息“和”我“三个按钮，以下分别介

绍。

## 5.2 主页设计实现

5.2.1 模糊搜索

|  |  |  |  |
| --- | --- | --- | --- |
| /private/var/folders/6k/tzcbt87127zf93dy7l8z8wc00000gn/T/com.kingsoft.wpsoffice.mac/picturecompress_20230608103013/output_1.pngoutput_1 | /private/var/folders/6k/tzcbt87127zf93dy7l8z8wc00000gn/T/com.kingsoft.wpsoffice.mac/picturecompress_20230608103026/output_1.pngoutput_1 | /private/var/folders/6k/tzcbt87127zf93dy7l8z8wc00000gn/T/com.kingsoft.wpsoffice.mac/picturecompress_20230608103057/output_1.pngoutput_1 | Simulator Screenshot - iPhone 14 Pro - 2023-06-08 at 10.29.26 |

5.2.2 地图导航

|  |  |
| --- | --- |
| /private/var/folders/6k/tzcbt87127zf93dy7l8z8wc00000gn/T/com.kingsoft.wpsoffice.mac/picturecompress_20230608104128/output_1.pngoutput_1 | Simulator Screenshot - iPhone 14 Pro - 2023-06-08 at 09.45.42 |

5.2.3 卡片介绍

|  |  |
| --- | --- |
| Simulator Screenshot - iPhone 14 Pro - 2023-06-08 at 09.46.17 | Simulator Screenshot - iPhone 14 Pro - 2023-06-08 at 10.45.59 |

5.2.4 评论

|  |  |  |  |
| --- | --- | --- | --- |
| Simulator Screenshot - iPhone 14 Pro - 2023-06-08 at 10.47.45 | Simulator Screenshot - iPhone 14 Pro - 2023-06-08 at 10.48.02 | /private/var/folders/6k/tzcbt87127zf93dy7l8z8wc00000gn/T/com.kingsoft.wpsoffice.mac/picturecompress_20230608105006/output_1.pngoutput_1 | Simulator Screenshot - iPhone 14 Pro - 2023-06-08 at 10.48.14 |

|  |  |  |  |
| --- | --- | --- | --- |
| Simulator Screenshot - iPhone 14 Pro - 2023-06-08 at 10.48.17 | Simulator Screenshot - iPhone 14 Pro - 2023-06-08 at 10.48.21 | Simulator Screenshot - iPhone 14 Pro - 2023-06-08 at 10.51.57 |  |

5.2.5 标记喜好

|  |  |  |  |
| --- | --- | --- | --- |
| /private/var/folders/6k/tzcbt87127zf93dy7l8z8wc00000gn/T/com.kingsoft.wpsoffice.mac/picturecompress_20230608110013/output_1.pngoutput_1 | Simulator Screenshot - iPhone 14 Pro - 2023-06-08 at 10.59.42 | /private/var/folders/6k/tzcbt87127zf93dy7l8z8wc00000gn/T/com.kingsoft.wpsoffice.mac/picturecompress_20230608110023/output_1.pngoutput_1 |  |

## 5.3 信息页设计实现

最上方是四个按钮，分别“表达意见”“申请活动”“入园码查询”“权益维护”。页面下方是已发起的服务列表，点击进入可以查看详情。

5.3.1 表达意见

|  |  |  |  |
| --- | --- | --- | --- |
| Simulator Screenshot - iPhone 14 Pro - 2023-06-08 at 11.01.08 | Simulator Screenshot - iPhone 14 Pro - 2023-06-08 at 11.04.01 | Simulator Screenshot - iPhone 14 Pro - 2023-06-08 at 11.04.05 | Simulator Screenshot - iPhone 14 Pro - 2023-06-08 at 11.04.11 |

5.3.2 申请活动

|  |  |  |  |
| --- | --- | --- | --- |
| Simulator Screenshot - iPhone 14 Pro - 2023-06-08 at 11.13.49 | Simulator Screenshot - iPhone 14 Pro - 2023-06-08 at 11.13.06 | Simulator Screenshot - iPhone 14 Pro - 2023-06-08 at 11.13.10 | Simulator Screenshot - iPhone 14 Pro - 2023-06-08 at 11.13.14 |

5.3.3 入园码查询

|  |  |  |  |
| --- | --- | --- | --- |
| Simulator Screenshot - iPhone 14 Pro - 2023-06-08 at 11.15.38 | Simulator Screenshot - iPhone 14 Pro - 2023-06-08 at 11.15.41 | Simulator Screenshot - iPhone 14 Pro - 2023-06-08 at 11.14.55 | Simulator Screenshot - iPhone 14 Pro - 2023-06-08 at 11.15.00 |

5.3.4 权益维护

|  |  |  |  |
| --- | --- | --- | --- |
| Simulator Screenshot - iPhone 14 Pro - 2023-06-08 at 11.19.21 | Simulator Screenshot - iPhone 14 Pro - 2023-06-08 at 11.21.09 | Simulator Screenshot - iPhone 14 Pro - 2023-06-08 at 11.21.13 | Simulator Screenshot - iPhone 14 Pro - 2023-06-08 at 11.21.17 |

## 5.4 “我的”界面设计实现

上方为“去过”卡片式列表，下方为收藏景点卡片式列表，该卡片可点击，点击后跳转至详情页。

|  |  |
| --- | --- |
| /private/var/folders/6k/tzcbt87127zf93dy7l8z8wc00000gn/T/com.kingsoft.wpsoffice.mac/picturecompress_20230608114245/output_1.pngoutput_1 | /private/var/folders/6k/tzcbt87127zf93dy7l8z8wc00000gn/T/com.kingsoft.wpsoffice.mac/picturecompress_20230608114250/output_1.pngoutput_1 |

第6章　总结与展望

Yangzhou APP是一个采用MVVM架构的旅游服务应用，旨在为游客提供全方位的扬州旅行体验。该应用利用iOS的沙盒存储技术实现了数据的本地存储，并通过Combine框架的Publisher和Subscriber模型实现了视图和模型之间的双向绑定。使用SwiftUI进行界面设计，利用声明式语法、组件化设计和数据绑定，实现了清晰、简洁且具有响应性的用户界面。通过与系统地图的集成，提供了导航功能。

未来，可以进一步完善和扩展Yangzhou APP的功能和特性。以下是一些可能的改进和发展方向：增加更多的旅游信息和功能：扩充应用中的景点介绍、活动信息、餐饮推荐等内容，提供更全面的旅游指南。还可以添加一些实用工具，如天气预报、语音导览等，提升用户体验；引入社交互动功能：允许用户之间的互动，例如评论、点赞、分享等。这将增加用户参与度，促进用户之间的交流和分享旅行经验；引入个性化推荐：通过分析用户的兴趣和偏好，提供个性化的旅游推荐。例如，根据用户的历史记录和收藏，推荐相关的景点、活动或餐饮场所；入在线预订功能：允许用户在线预订门票、酒店或旅行套餐等服务。与合作伙伴或第三方服务集成，提供便捷的预订体验；数据分析和改进：通过收集和分析用户行为数据，了解用户的使用习惯和偏好，进而改进和优化应用的功能和用户体验。

参考文献

1. "Swift Programming: The Big Nerd Ranch Guide" by Matthew Mathias and John Gallagher
2. "Swift in Depth" by Tjeerd in 't Veen
3. "SwiftUI by Tutorials" by raywenderlich.com Team