SwiftUI 3.0 实用教程

介绍

WWDC 21 Apple 对 SwiftUI 进行了进一步更新（SwiftUI 3.0），增加了很多 API，更新了很多 API，同时也废弃了很多 API。本教程主要针对这些变化进行介绍，帮助大家能够快速过渡到 SwiftUI 3.0。Apple 官方显示本次新增内容仅适用于 **iOS 15**（个别除外），但之前的 SwiftUI 代码可以直接在 Xcode 13.x 中运行，完全兼容。

主要内容

- 开发iOS项目
- View
- Modifier
- Environment
- 跨平台
- 网络编程
- Core Data
- Accessibility

环境要求

1. 开发工具：Xcode 13 及以上。
2. 运行环境：iOS 15 / iPadOS 15 / macOS 12 / tvOS 15 / watchOS 8 及以上。

学习要求

1. 掌握 Swift 语法，尤其是 Swift 5.5 新特性。
2. 掌握 SwiftUI 1.0 和 2.0 中的各项知识。

作者

- YungFan，杨帆，高校教师，开发者。
- Email：yungfan@vip.163.com。
- GitHub：https://github.com/yungfan。
- 在线课程：https://ke.qq.com/cgi-bin/agency?aid=67223。
- 微信公众号：YungFan。
- 博客：**掘金**、**腾讯云·云社区** 和 **简书**。

勘误

各位读者在阅读本教程时，如果对其中的内容有疑义或者修改建议，欢迎联系作者。

版权

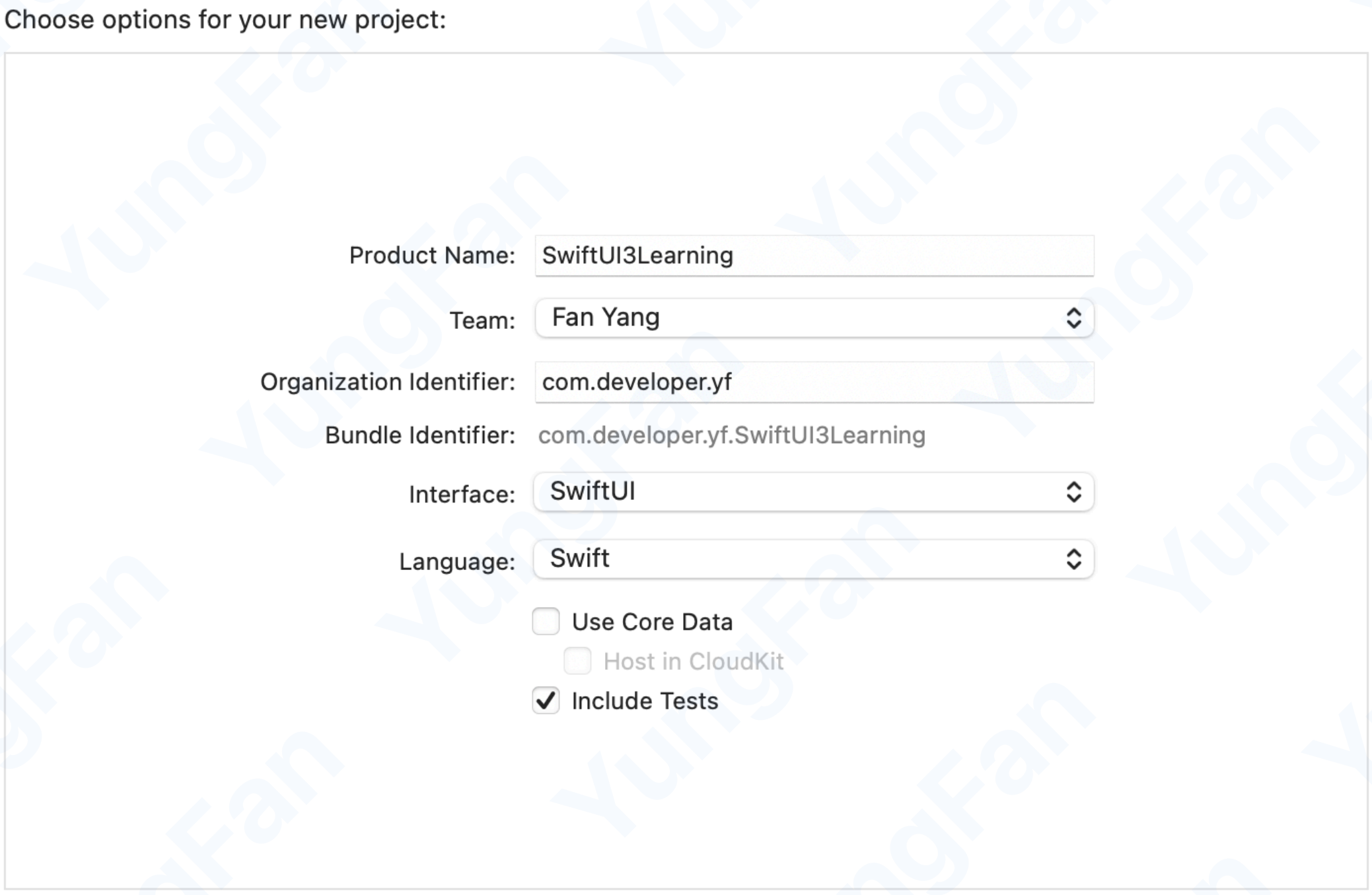
更新与修订

时间	章节	更新/修订内容
2021.10	**View**	AsyncImage—增加List中使用
	Modifier	1.refreshable—增加自定义 2.environment(.openURL)—增加案例
	Environment	增加controlSize和keyboardShortcut
2021.11	**View**	1.ControlGroup—增加自定义样式 2.TimelineView—增加汤姆猫案例 3.Canvas—增加复杂操作
	Modifier	增加monospacedDigit和contentShape
	Environment	增加controlSize和keyboardShortcut
2021.12	**开发iOS项目~new**	1.标题与内容顺序调整 2.增加创建项目变化
	View	1.Update—增加NavigationView和TabView ★ 2.List—增加设置safeAreaInset 3.Form—增加设置safeAreaInset
	网络编程	增加进一步封装
	CoreData~new	
	Accessibility~new	

开发iOS项目~new

创建项目变化

Xcode 13 创建 iOS App 时，去除了 `Life Cycle` 选项，默认即为 `SwiftUI App` 。



项目文件变化

新建项目不在有 Info.plist 文件，它被集成在：**项目 —> TARGET —> Info —> Custom iOS Target Properties**。

预览变化

可以根据指定的方向进行内容的预览。

```
struct ContentView: View {
    var body: some View {
        Text("WWDC21 SwiftUI")
            .padding()
    }
}

struct ContentView_Previews: PreviewProvider {
    static var previews: some View {
        Group {
            ContentView()
                .previewInterfaceOrientation(.landscapeLeft)

            ContentView()
```

```
                .previewInterfaceOrientation(.landscapeRight)

            ContentView()
                .previewInterfaceOrientation(.portrait)

            ContentView()
                .previewInterfaceOrientation(.portraitUpsideDown)
        }
    }
}
```

View

语法改进

得益于 Swift 5.5 中 “**在通用上下文中扩展静态成员查找**” 的新特性，SwiftUI 中的很多语法变得更简洁，如：

- `SidebarListStyle` 可以写成 `.sidebar` 。
- `RoundedBorderTextFieldStyle` 可以写成 `.roundedBorder` 。
- `SwitchToggleStyle` 可以写成 `.switch` 。
- `PlainButtonStyle` 可以变成 `.plain` 。

New

AsyncImage

用于异步加载图片，基于 URLSession，缓存使用的是 URLCache，目前不支持自定义缓存。

- 加载异步图片。

```
struct ContentView: View {
    // 图片URL
    let url = URL(string: "https://ss0.bdstatic.com/70cFuHSh_Q1YnxGkpoWK1HF6hhy/it/u=2718219500,1861579782&fm=26&gp=0.jpg")

    var body: some View {
        AsyncImage(url: url) { image in
            image
                .resizable()
                .aspectRatio(contentMode: .fit)
```

```
        } placeholder: { // 占位View
            ProgressView()
        }
    }
}
```

- 监听加载过程的不同状态。

```
struct ContentView: View {
    // 图片URL
    let url = URL(string: "https://ss0.bdstatic.com/70cFuHSh_Q1YnxGkpoWK1HF6hhy/it/u=2718219500,1861579782&fm=26&gp=0.jpg")
    // 状态发生变化时的动画
    let transaction: Transaction = .init(animation: .linear)

    var body: some View {
        AsyncImage(url: url, transaction: transaction) { phase in
            switch phase {
            case .empty: // 未加载
                ProgressView()
            case let .success(image): // 加载成功
                image
                    .resizable()
                    .aspectRatio(contentMode: .fit)
            case .failure: // 加载失败
                Image(systemName: "heart")
            default:
                EmptyView()
            }
        }
    }
}
```

- List 中使用。

```
struct ContentView: View {
    private let url = URL(string: "https://gimg2.baidu.com/image_search/src=http%3A%2F%2Fwww.eet-china.com%2Fd%2Ffile%2Fkj%2F
    var body: some View {
        List {
            ForEach(0 ..< 10) { _ in
                AsyncImage(url: url) { image in
                    image
                        .resizable()
                        .aspectRatio(contentMode: .fit)
                } placeholder: {
                    ProgressView()
                }
                .listRowInsets(.init(.zero))
            }
        }
        .listStyle(.plain)
    }
}
```

LocationButton

- 内置于 `CoreLocationUI` 框架。
- 当用户点击按钮时，能够授予一次性位置授权，但它无法获取定位的详细信息，如要获取还是需要借助于 `CoreLocation` 模块。

```
import CoreLocationUI

struct ContentView: View {
    var body: some View {
        LocationButton(.currentLocation) {
        }
        .foregroundColor(.white)
        .cornerRadius(20)
        .labelStyle(.titleAndIcon)
        .frame(width: 200, height: 44)
    }
}
```

- 搭配 CoreLocation 和 MapKit 使用。

```
import CoreLocation
import CoreLocationUI
import MapKit
import SwiftUI

class ObservableLocationManager: NSObject, ObservableObject, CLLocationManagerDelegate {
    private let locationManager = CLLocationManager()
    // 显示范围
    @Published var region = MKCoordinateRegion(center: CLLocationCoordinate2D(latitude: 39.91667, longitude: 116.41667), lati

    override init() {
        super.init()

        locationManager.delegate = self
    }

    func requestLocation() {
        locationManager.requestLocation()
    }

    func locationManager(_ manager: CLLocationManager, didUpdateLocations locations: [CLLocation]) {
        guard let location = locations.first else { return }
        region = MKCoordinateRegion(center: location.coordinate, latitudinalMeters: 500, longitudinalMeters: 500)
    }

    func locationManager(_ manager: CLLocationManager, didFailWithError error: Error) {
        print(error.localizedDescription)
    }
}

struct ContentView: View {
    @StateObject var locationManager = ObservableLocationManager()

    var body: some View {
        ZStack(alignment: .bottom) {
            Map(coordinateRegion: $locationManager.region, showsUserLocation: true)
                .edgesIgnoringSafeArea(.all)

            HStack {
                LocationButton {
                    locationManager.requestLocation()
                }
                .frame(width: 60, height: 60)
                .cornerRadius(30)
                .labelStyle(.iconOnly)
                .foregroundColor(.white)
            }
            .padding()
        }
    }
}
```

ControlGroup

一个容器视图，用于将多个相关 View 放在一起作为一个整体表示应用程序的特定部分。

- 基本使用。

```
struct ContentView: View {
    var body: some View {
        NavigationView {
            ControlGroup {
                Button("点赞") {
                }

                Button("收藏") {
                }
            }
            .frame(width: 100)
            .navigationTitle("ControlGroup")
            .toolbar {
```

```
                ControlGroup {
                    Button {
                    } label: {
                        Image(systemName: "heart")
                    }

                    Button {
                    } label: {
                        Image(systemName: "star")
                    }
                }
            }
        }
    }
}
```

- 自定义样式。

```
struct CustomControlGroupStyle: ControlGroupStyle {
    func makeBody(configuration: Configuration) -> some View {
        ControlGroup(configuration)
            .border(.red)
    }
}

struct ContentView: View {
    var body: some View {
        NavigationView {
            Text("ControlGroup")
                .navigationTitle("ControlGroup")
                .toolbar { ToolbarItem(placement: .navigationBarTrailing) {
                    ControlGroup {
                        Button {
                            print("喜欢")
                        } label: {
                            Label("喜欢", systemImage: "heart")
                        }

                        Button {
                            print("收藏")
                        } label: {
                            Label("收藏", systemImage: "star")
                        }
                    }
                    .controlGroupStyle(CustomControlGroupStyle())
                }
            }
        }
    }
}
```

TimelineView

根据提供的计划进行更新的 View，有 3 种更新计划：

- `EveryMinuteTimelineSchedule`：在每分钟开始时更新。
- `PeriodicTimelineSchedule`：以自定义的开始时间和更新间隔定期更新。
- `ExplicitTimelineSchedule`：当需要有限数量或不规则的更新规则时使用。

```
struct ContentView: View {
    var body: some View {
        VStack {
            // 默认1s更新一次
            TimelineView(.animation) { context in
                Text(context.date.formatted(.dateTime.year().month().day().hour().minute().second().locale(Locale(identifier:

                // 更新内容
            }

            // 3s更新一次
            TimelineView(.periodic(from: Date.now, by: 3)) { context in
                Text(context.date.formatted(.dateTime.year().month().day().hour().minute().second().locale(Locale(identifier:
```

```
                // 更新内容
            }

            // 每分钟更新一次
            TimelineView(.everyMinute) { context in
                Text(context.date.formatted(.dateTime.year().month().day().hour().minute().second().locale(Locale(identifier:

                // 更新内容
            }
        }
    }
}
```

- 汤姆猫案例。

```
// 动作类型
enum TomcatAction: String {
    case drink
    case eat
}

// 动作类型和图片张数
struct Tomcat {
    var name: TomcatAction
    var count: Int
}

struct ContentView: View {
    var body: some View {
        TimelineView(.periodic(from: Date.now, by: 0.1)) { context in
            // 0.1s刷新一次AnimatedView
            AnimatedView(date: context.date)
        }
    }
}

struct AnimatedView: View {
    // 第一次播放的动画
    @State private var action = Tomcat(name: .drink, count: 81)
    // 默认显示的图片
    @State private var imageName = "drink_0"
    // 图片计数
    @State private var count = 0
    // 监听日期
    let date: Date

    var body: some View {
        ZStack {
            // 切换的图片
            Image(imageName)
                .resizable()
                .onChange(of: date) { _ in
                    count = (count + 1) % action.count
                    imageName = "\(action.name.rawValue)_\(count)"
                }

            // 动作按钮
            HStack {
                VStack {
                    Button {
                        // 计数清0
                        count = 0
                        // 设置新的动作
                        action = Tomcat(name: .drink, count: 81)
                    } label: {
                        Image("drink")
                    }
                }

                Spacer()

                VStack {
                    Button {
                        count = 0
```

```
                        action = Tomcat(name: .eat, count: 40)
                    } label: {
                        Image("eat")
                    }
                }
            }
        }
        .ignoresSafeArea()
        .frame(width: UIScreen.main.bounds.width, height: UIScreen.main.bounds.height)
    }
}
```

Canvas

- 通过新的 Canvas API，SwiftUI 获得了对绘图的更多支持。
- 可以绘制文本、图片、路径等。

```
struct ContentView: View {
    var body: some View {
        VStack {
            Canvas { context, _ in // 2个参数为上下文和大小
                // 文本
                context.draw(Text("SwiftUI"), in: CGRect(x: 0, y: 0, width: 100, height: 30))
            }

            Canvas { context, _ in
                // 图片
                context.draw(Image(systemName: "heart"), in: CGRect(x: 0, y: 10, width: 30, height: 30))
            }

            Canvas { context, size in
                // 路径，Path进行fill
                context.fill(
                    Path(ellipseIn: CGRect(origin: .zero, size: size)),
                    with: .color(.green))
            }
            .frame(width: 300, height: 300)
            .border(.blue)

            Canvas { context, size in
                // 路径，对Path进行stroke
                context.stroke(
                    Path(ellipseIn: CGRect(origin: .zero, size: size)),
                    with: .color(.green),
                    lineWidth: 4)
            }
            .frame(width: 300, height: 300)
            .border(.blue)
        }
    }
}
```

- 复杂操作。

```
struct ContentView: View {
    var body: some View {
        VStack {
            // 实心1/4圆，扇形
            Canvas { context, size in
                context.clip(to: Path(CGRect(origin: .zero, size: size.applying(CGAffineTransform(scaleX: 0.5, y: 0.5)))))
                context.fill(
                    Path(ellipseIn: CGRect(origin: .zero, size: size)),
                    with: .color(.green))
            }
            .frame(width: 200, height: 200)
            .border(.blue)

            // 空心圆
            Canvas { context, size in
                context.clip(to: Path(CGRect(origin: .zero, size: size.applying(CGAffineTransform(scaleX: 1, y: 1)))))
                context.stroke(
                    Path(ellipseIn: CGRect(origin: .zero, size: size)),
                    with: .color(.green))
```

```
            }
            .frame(width: 200, height: 200)
            .border(.blue)
        }
    }
}
```

调试

新增了一个非常实用的调试工具，只需要在 body 内调用 `Self._printChanges()`，就可以查看到底是什么触发了 View 的更新。

```
class StateChangeObject: ObservableObject {
    @Published var changedTimes = 0
    var timer: Timer?

    init() {
        timer = Timer.scheduledTimer(withTimeInterval: 3, repeats: true, block: { _ in
            self.changedTimes += 1
        })
    }
}

struct ContentView: View {
    @ObservedObject var stateChangeObject = StateChangeObject()

    var body: some View {
        print(Self._printChanges()) // 输出stateChangeObject状态发生变化

        let times = stateChangeObject.changedTimes

        return Text("ContentView变化了\(times)次")
    }
}
```

Update

Color

新增了几种颜色。

```
struct ContentView: View {
    var body: some View {
        HStack {
            Circle()
                .fill(.mint)

            Circle()
                .fill(.teal)

            Circle()
                .fill(.cyan)

            Circle()
                .fill(.indigo)

            Circle()
                .fill(.brown)
        }
    }
}
```

Text

- 支持 `AttributedString`，可以直接渲染 Markdown。

```
struct ContentView: View {
    // 创建AttributedString
    var attributedString: AttributedString {
        var string = AttributedString("WWDC21 SwiftUI")
```

```
        string.foregroundColor = .red
        string.font = .title
        // 下划线，有多种类型
        string.underlineStyle = .init(pattern: .dash, color: .red)
        // 删除线，有多种类型
        string.strikethroughStyle = .init(pattern: .dot, color: .blue)
        // WWDC为蓝色
        if let range = string.range(of: "WWDC") {
            string[range].foregroundColor = .green
        }
        return string
    }

    var body: some View {
        VStack {
            // 可以直接渲染markdown，但只支持内联样式
            // 加粗
            Text("**WWDC21 SwiftUI**")

            // 斜体
            Text("*WWDC21 SwiftUI*")

            // 代码
            Text("`print('WWDC21 SwiftUI')`")

            // 删除线
            Text("~~WWDC21 SwiftUI~~")

            // 超链接
            Text("[WWDC21](https://developer.apple.com/wwdc21/)")

            Text(attributedString)
        }
    }
}
```

- 显示日期。

```
struct ContentView: View {
    @State private var date = Date.now
    // 创建AttributedString
    var dateString: AttributedString {
        // 新增的Date格式方式
        var string = date.formatted(.dateTime.year().month().day().weekday(.wide).locale(Locale(identifier: "zh_CN")).attribu
        // 针对星期做特殊处理
        let weekday = AttributeContainer.dateField(.weekday)
        // 创建属性容器
        var container = AttributeContainer()
        // 设置属性
        container.foregroundColor = .white
        container.font = .title
        // 属性替换，星期文字变为白色且字体变大
        string.replaceAttributes(weekday, with: container)
        // 属性合并，文字变为白色且字体变大
        // string.mergeAttributes(container)
        return string
    }

    var body: some View {
        Text(dateString)
    }
}
```

- 支持多种内容格式化方式。

```
struct ContentView: View {
    private let value = 12345
    private let percent = 25
    private let price = 100
    private let names = ["zhangsan", "lisi", "wangwu"]
    private let weight = Measurement(value: 100, unit: UnitMass.kilograms)

    @State private var input: String = ""
```

```
    var body: some View {
        VStack(spacing: 30) {
            // 数值
            Text(value, format: .number)
            // 百分比
            Text(percent, format: .percent)
            // 货币
            Text(price, format: .currency(code: "rmb"))
            // 数组
            Text(names, format: .list(type: .and))
            // 重量
            Text(weight, format: .measurement(width: .wide))
        }
        .padding()
    }
}
```

Image

新增 `symbolRenderingMode` 和 `symbolVariant` 修饰符，用于设置 Symbol Images 的渲染样式。

symbolRenderingMode

- 很多 SF 符号图标包含表示图标深度的多个层，这些层可以通过不同透明度的方式呈现，以突出图标的含义。
- 很多 SF 符号图标被划分为 2 种或 3 种颜色的层。它们的渲染方式取决于它们包含的层数和提供的颜色。
  - 如果只有 1 层，它将始终使用指定的第 1 种颜色，而忽略其他颜色。
  - 如果有 2 层，但只指定了 1 种颜色，则 2 层都将使用这种颜色。如果指定 2 种颜色，则每层将使用 1 种颜色。如果指定 3 种颜色，则忽略第 3 种颜色。
  - 如果有 3 层，但只指定了 1 种颜色，则 3 层都将使用这种颜色。如果指定 2 种颜色，则第 1 层使用 1 种颜色，其余 2 层使用第 2 种颜色。如果指定 3 种颜色，则每层使用 1 种颜色。

```
struct ContentView: View {
    var body: some View {
        VStack(spacing: 20) {
            // 单层颜色
            Image(systemName: "person.2.fill")
                .foregroundStyle(.purple)
                .symbolRenderingMode(.monochrome)

            // 多层渲染，透明度不同
            Image(systemName: "person.2.fill")
                .foregroundStyle(.red)
                .symbolRenderingMode(.hierarchical)

            // 多层渲染，设置不同风格，这种方式最特殊
            Image(systemName: "person.2.fill")
                .foregroundStyle(.green, .blue) // 提供2种颜色，还可以是Material、Gradient
                .symbolRenderingMode(.palette)

            // 多层渲染，很多图标是多彩的，直接渲染，如果是单色图标设置无效
            Image(systemName: "circle.hexagongrid.fill")
                .symbolRenderingMode(.multicolor)
        }
        .font(.system(size: 50))
    }
}
```

symbolVariant

一些 SF 符号还提供了多种变体，通过 `symbolVariant` 来指定这种变体样式（必须有变体该样式才会有效果）。

```
struct ContentView: View {
    var body: some View {
        VStack(spacing: 20) {
            Image(systemName: "trash")
                .symbolVariant(.slash)

            Image(systemName: "trash")
                .symbolVariant(.slash.fill)
```

```
            Image(systemName: "trash")
                .symbolVariant(.fill)

            Image(systemName: "trash")
                .symbolVariant(.square)

            Image(systemName: "trash")
                .symbolVariant(.square.fill)

            Image(systemName: "trash")
                .symbolVariant(.none)

            Image(systemName: "trash")
                .symbolVariant(.circle)

            Image(systemName: "trash")
                .symbolVariant(.circle.fill)
        }
            .foregroundStyle(.red)
            .font(.system(size: 50))
    }
}
```

TextField

支持在弹出的键盘上添加工具条。

```
struct ContentView: View {
    @State private var name: String = "zhangsan"

    var body: some View {
        NavigationView {
            TextField("请输入用户名", text: $name)
                .textFieldStyle(.roundedBorder)
                .padding()
                .toolbar {
                    ToolbarItemGroup(placement: .keyboard) { // 键盘之上添加toolbar
                        Button("拷贝") {
                        }
                        Button("剪切") {
                        }
                        Button("删除") {
                        }
                        Button("粘贴") {
                        }
                    }
                }
        }
    }
}
```

注意：也支持 SecureField 和 TextEditor。

Button

- 新增圆角矩形样式 `BorderedButtonStyle` 和圆角突出显示样式 `BorderedProminentButtonStyle` 。

```
struct ContentView: View {
    var body: some View {
        VStack(spacing: 20) {
            Button {
            } label: {
                Label("YES", systemImage: "checkmark")
            }
            .tint(.green) // 默认样式

            Button {
            } label: {
                Label("YES", systemImage: "checkmark")
            }
            .tint(.green)
            .buttonStyle(.plain) // 普通样式
```

```
        Button {
        } label: {
            Label("YES", systemImage: "checkmark")
        }
        .tint(.green)
        .buttonStyle(.borderless) // 没有边框

        Button {
        } label: {
            Label("YES", systemImage: "checkmark")
        }
        .tint(.green)
        .buttonStyle(.bordered) // 圆角

        Button {
        } label: {
            Label("YES", systemImage: "checkmark")
        }
        .tint(.green)
        .buttonStyle(.borderedProminent) // 突出显示（背景变实色）
    }
  }
}
```

- 新增 `ButtonRole` 用于描述按钮的角色。

```
struct ContentView: View {
    var body: some View {
        VStack {
            // 蓝色
            Button("取消", role: .cancel) {
                print("取消")
            }

            // 红色
            Button("删除", role: .destructive) {
                print("删除")
            }
        }
    }
}
```

Toggle

新增了一种 `ButtonToggleStyle`，显示的效果类似按钮，通过点击呈现不同的效果。

```
struct ContentView: View {
    @State private var isOn = false

    var body: some View {
        VStack {
            Toggle(isOn: $isOn, label: { Text("开关") })

            Toggle(isOn: $isOn, label: { Text("开关") })
                .disabled(isOn)
        }
        .toggleStyle(.button)
    }
}
```

Picker

默认显示的不再是滚轮样式，而是 Menu 样式。

```
struct ContentView: View {
    var items = ["黄", "紫", "红"]

    var colors: [Color] = [.orange, .purple, .red]

    @State private var currentIndex: Int = 0
```

```
    var body: some View {
        ZStack {
            colors[currentIndex]

            Picker("颜色", selection: $currentIndex) {
                ForEach(0 ..< self.items.count) { index in
                    Text(self.items[index])
                        .tag(index)
                }
            }
        }
    }
}
```

ScrollView

支持设置 `safeAreaInset` （contentInset）。

```
struct ContentView: View {
    var body: some View {
        ScrollView {
            Color
                .orange
                .frame(width: UIScreen.main.bounds.width, height: 1500)
        }
        .safeAreaInset(edge: .top) { // 顶部contentInset
            VStack {
                Text("顶部")

                Divider()
                    .frame(height: 5)
                    .background(.blue)
            }
            .background(.bar) // 选择bar
        }
        .safeAreaInset(edge: .bottom) { // 底部contentInset
            VStack {
                Divider()
                    .frame(height: 5)
                    .background(.blue)

                Text("底部")
            }
            .background(.green) // 选择颜色
        }
    }
}
```

ForEach

支持可绑定的内容遍历（兼容 iOS 14，但要求 Xcode 13）。

```
struct Person: Identifiable {
    let id = UUID()
    var name = ""
    var gender = false
}

struct ContentView: View {
    @State private var person = [Person(name: "zhangsan", gender: true), Person(name: "lisi"), Person(name: "wangwu")]

    var body: some View {
        // ForEach支持可绑定的内容遍历
        ForEach($person) { $p in
            DetailView(name: $p.name, gender: $p.gender)
        }
    }
}

struct DetailView: View {
    @Binding var name: String
    @Binding var gender: Bool
```

```
    var body: some View {
        HStack {
            Text(name)
                .frame(width: 120)

            TextField("", text: $name)
                .textFieldStyle(.roundedBorder)

            Toggle(isOn: $gender) {
            }
        }
    }
}
```

List

- 支持可绑定的内容遍历（兼容 iOS 14，但要求 Xcode 13）。

```
struct Person: Identifiable {
    let id = UUID()
    var name = ""
    var gender = false
}

struct ContentView: View {
    @State private var person = [Person(name: "zhangsan", gender: true), Person(name: "lisi"), Person(name: "wangwu")]

    var body: some View {
        // List支持可绑定的内容遍历
        List($person) { $p in
            DetailView(name: $p.name, gender: $p.gender)
        }
    }
}
struct DetailView: View {
    @Binding var name: String
    @Binding var gender: Bool

    var body: some View {
        HStack {
            Text(name)
                .frame(width: 120)

            TextField("", text: $name)
                .textFieldStyle(.roundedBorder)

            Toggle(isOn: $gender) {
            }
        }
    }
}
```

- 和 ScrollView 一样支持设置 `safeAreaInset`。

```
struct ContentView: View {
    var body: some View {
        List {
            ForEach(0 ..< 10) {
                Text("Row \($0)")
            }
        }
        .safeAreaInset(edge: .top) { // 顶部contentInset
            VStack {
                Text("顶部")

                Divider()
                    .frame(height: 5)
                    .background(.blue)
            }
            .background(.green)
        }
```

```
        .safeAreaInset(edge: .bottom) { // 底部contentInset
            VStack {
                Divider()
                    .frame(height: 5)
                    .background(.blue)

                Text("底部")
            }
            .background(.red)
        }
    }
}
```

Section

改进了构造函数。

```
struct ContentView: View {
    var body: some View {
        List {
            Section { // content
                Text("Item1")
                Text("Item2")
                Text("Item3")
            } header: { // header
                Text("Header1")
            } footer: { // footer
                Text("Footer1")
            }

            Section {
                Text("Item4")
                Text("Item5")
                Text("Item6")
            } header: {
                Text("Header2")
            } footer: {
                Text("Footer2")
            }
        }
    }
}
```

NavigationView

- iOS 15 中导航栏的 background material 被移除了，默认情况下会和内容融为一体，不再有以前的突出显示效果。

```
struct ContentView: View {
    var body: some View {
        NavigationView {
            VStack {
                Text("iOS15 导航栏")

                Spacer()
            }
            .navigationTitle("标题")
            .navigationBarTitleDisplayMode(.inline)
        }
    }
}
```

设置

如果希望导航栏跟以前一样有层次感，可以按照如下的方式对其背景进行设置。

- 方式一：借助于 `UINavigationBarAppearance` 恢复之前的样式。

```
struct ContentView: View {
    var body: some View {
        NavigationView {
            VStack {
```

```
            Text("iOS15 导航栏")
                .padding()

            Spacer()
        }
        .navigationTitle("标题")
        .navigationBarTitleDisplayMode(.inline)
        .onAppear {
            if #available(iOS 15.0, *) {
                let appearance = UINavigationBarAppearance()
                appearance.backgroundEffect = UIBlurEffect(style: .systemUltraThinMaterial)
                appearance.backgroundColor = UIColor(.green.opacity(0.4))
                UINavigationBar.appearance().standardAppearance = appearance
                UINavigationBar.appearance().scrollEdgeAppearance = appearance
            }
        }
    }
}
```

- 方式二：使用新方式，更加多样灵活。

```
struct ContentView: View {
    var body: some View {
        NavigationView {
            VStack {
                // 方式一：纯色背景
                Rectangle()
                    .frame(height: 0)
                    .background(.green.opacity(0.4))

                // 方式二：Gradient背景
                Rectangle()
                    .fill(.clear)
                    .frame(height: 0)
                    .background(LinearGradient(colors: [.purple.opacity(0.5), .green.opacity(0.5)], startPoint: .topLeading, 

                // 方式三：Material背景，后面会讲解
                Divider()
                    .background(.ultraThinMaterial)

                Text("iOS15 导航栏")
                    .padding()

                Spacer()
            }
            .navigationTitle("标题")
            .navigationBarTitleDisplayMode(.inline)
        }
    }
}
```

TabView

与 NavigationView 一样，iOS 15 中标签栏的 background material 也被移除了。

```
struct ContentView: View {
    var body: some View {
        TabView {
            Text("微信")
                .tabItem {
                    Image(systemName: "message")
                    Text("微信")
                }

            Text("通讯录")
                .tabItem {
                    Image(systemName: "person.2")
                    Text("通讯录")
                }
        }
    }
}
```

设置

如果希望标签栏跟以前一样有层次感，可以按照如下的方式对其背景进行设置。

- 方式一：借助于 `UITabBarAppearance` 恢复之前的样式。

```
struct ContentView: View {
    var body: some View {
        TabView {
            Text("微信")
                .tabItem {
                    Image(systemName: "message")
                    Text("微信")
                }

            Text("通讯录")
                .tabItem {
                    Image(systemName: "person.2")
                    Text("通讯录")
                }
        }
        .onAppear {
            if #available(iOS 15.0, *) {
                let appearance = UITabBarAppearance()
                appearance.backgroundEffect = UIBlurEffect(style: .systemUltraThinMaterial)
                appearance.backgroundColor = UIColor(.green.opacity(0.4))
                UITabBar.appearance().standardAppearance = appearance
                UITabBar.appearance().scrollEdgeAppearance = appearance
            }
        }
    }
}
```

- 方式二：使用新方式，更加多样灵活。

```
struct ContentView: View {
    var body: some View {
        TabView {
            // 每个Item需要单独设置
            VStack {
                Text("微信")
                    .padding()
                    .frame(maxHeight: .infinity) // 不能少，关键

                // 方式一：纯色背景
                Rectangle()
                    .frame(height: 0)
                    .background(.green.opacity(0.4))

                // 方式二：Gradient背景
                Rectangle()
                    .fill(.clear)
                    .frame(height: 0)
                    .background(LinearGradient(colors: [.purple.opacity(0.5), .green.opacity(0.5)], startPoint: .topLeading, 

                // 方式三：Material背景
                Divider()
                    .background(.ultraThinMaterial)
            }
            .tabItem {
                Image(systemName: "message")
                Text("微信")
            }

            // 第二个并没有效果
            Text("通讯录")
                .tabItem {
                    Image(systemName: "person.2")
                    Text("通讯录")
                }
        }
    }
}
```

Form

和 ScrollView 一样支持设置 `safeAreaInset` 。

```
struct ContentView: View {
    var body: some View {
        Form {
            Text("WWDC21 SwiftUI")
        }
        .safeAreaInset(edge: .top) { // 顶部contentInset
            VStack {
                Text("顶部")

                Divider()
                    .frame(height: 5)
                    .background(.blue)
            }
            .background(.purple)
        }
        .safeAreaInset(edge: .bottom) { // 底部contentInset
            VStack {
                Divider()
                    .frame(height: 5)
                    .background(.blue)

                Text("底部")
            }
            .background(.orange)
        }
    }
}
```

Link

支持 Text 渲染的 Markdown 链接。

```
struct ContentView: View {
    var body: some View {
        Text("[WWDC21](https://developer.apple.com/wwdc21/)")

        Text("[开发者网站](https://developer.apple.com)")
    }
}
```

Menu

之前是通过点击触发，现在同时支持点击与按压 2 种操作。

```
struct ContentView: View {
    @State private var backgroundColor: Color = .orange

    var body: some View {
        ZStack {
            backgroundColor

            Menu("菜单") { // 按压
                Button(action: {
                    backgroundColor = .purple
                }) {
                    Label("紫色", systemImage: "pencil")
                }

                Button(action: {
                    backgroundColor = .red
                }) {
                    Label("红色", systemImage: "trash")
                }

                Button(action: {
                    backgroundColor = .green
```

```
                }) {
                    Label("绿色", systemImage: "plus")
                }
            } primaryAction: { // 点击
                backgroundColor = .black
            }
        }
    }
}
```

Deprecated

- Alert：替换为 `.alert` 。

```
struct ContentView: View {
    @State private var showAlert = false

    var body: some View {
        Button("Alert") {
            self.showAlert.toggle()
        }
        .alert("温馨提示", isPresented: $showAlert) {
            Button("OK") {
            }
            // 可以有多个按钮

            // 默认有取消按钮，也可以进行替换
            Button("NO", role: .cancel) {
            }
        }
    }
}
```

- ActionSheet：替换为 `.confirmationDialog` 。

```
struct ContentView: View {
    @State private var showSheet = false

    var body: some View {
        Button("Sheet") {
            self.showSheet.toggle()
        }
        .confirmationDialog("温馨提示", isPresented: $showSheet, titleVisibility: .visible) { // titleVisibility影响标题的显示
            Button("相机") {
            }

            Button("相册") {
            }

            // 默认有取消按钮，也可以进行替换
            Button("NO", role: .cancel) {
            }
        }
    }
}
```

Modifier

New

monospacedDigit

为 Text 设置等宽字体，必须字体支持才有效果。

```
struct ContentView: View {
    var body: some View {
        // 中文
        Text("苹果发布会")

        Text("苹果发布会")
            .monospacedDigit()

        // 英文
        Text("WWDC21 SwiftUI")

        Text("WWDC21 SwiftUI")
            .monospacedDigit()

        // 数字
        Text("1234567890")

        Text("1234567890")
            .monospacedDigit()
    }
}
```

textSelection

可以选中 Text 中的文本，执行一些内置的操作，如拷贝、分享等。

```
struct ContentView: View {
    @State private var input: String = ""

    var body: some View {
        VStack(spacing: 30) {
            Text("文本不能选择")

            Text("文本可以选择")
                .textSelection(.enabled)

            // TextField默认支持
            TextField("", text: $input)
                .textFieldStyle(.roundedBorder)
        }
        .padding()
    }
}
```

foregroundStyle

- foreground 不再只能设置 Color，还能设置其他的如 Gradient、HierarchicalShapeStyle、Material。

```
struct ContentView: View {
    let gradient = Gradient(colors: [.red, .green])

    var body: some View {
        VStack {
            // 颜色
            Text("WWDC21 SwiftUI")
                .font(.title2)
                .foregroundStyle(.blue)

            // HierarchicalShapeStyle: primary、secondary、tertiary、quaternary，颜色从深到浅
            Text("WWDC21 SwiftUI")
                .font(.title2)
                .foregroundStyle(.primary)

            Text("WWDC21 SwiftUI")
                .font(.title2)
                .foregroundStyle(.secondary)

            Text("WWDC21 SwiftUI")
                .font(.title2)
                .foregroundStyle(.tertiary)

            Text("WWDC21 SwiftUI")
                .font(.title2)
                .foregroundStyle(.quaternary)

            Image(systemName: "heart")
                .resizable()
                .frame(width: 200, height: 200)
        }
        .foregroundStyle(.linearGradient(gradient, startPoint: .leading, endPoint: .trailing)) // Gradient
    }
}
```

AnyShapeStyle

新增了一个新的 ShapeStyle，用于包装不同的内部样式，对外提供统一的样式，类似于 AnyView。

```
struct ContentView: View {
    let gradient = Gradient(colors: [.red, .green])

    @State private var isChange = false
```

```
    var body: some View {
        VStack {
            Button("切换样式") {
                isChange.toggle()
            }
        }
        .foregroundStyle(isChange ? AnyShapeStyle(.black) : AnyShapeStyle(.linearGradient(gradient, startPoint: .leading, end
    }
}
```

tint

设置 View 内的 tint color，可以覆盖 View 的 accent color。

```
struct ContentView: View {
    var body: some View {
        VStack(spacing: 20) {
            Button {
            } label: {
                Label("YES", systemImage: "checkmark")
            }
            .buttonStyle(.bordered)
            .tint(.green) // 渲染色

            Button {
            } label: {
                Label("NO", systemImage: "xmark")
                    .frame(width: 200)
            }
            .buttonStyle(.bordered)
            .tint(.red)
        }
    }
}
```

textInputAutocapitalization

决定输入框（TextField、TextEditor）首字母是否大写。

```
struct ContentView: View {
    @State private var username: String = ""

    var body: some View {
        VStack(spacing: 30) {
            TextField("", text: $username)
                .textFieldStyle(.roundedBorder)
                .textInputAutocapitalization(.words) // 单词首字母大写
            // .textInputAutocapitalization(.sentences) // 句子首字母大写
            // .textInputAutocapitalization(.characters) // 每个字母都大写
            // .textInputAutocapitalization(.never) // 不大写

            TextEditor(text: $username)
                .textInputAutocapitalization(.words)
            .padding()
    }
}
```

buttonBorderShape

用于设置按钮的形状。

```
struct ContentView: View {
    var body: some View {
        VStack(spacing: 20) {
            Button {
            } label: {
                Label("YES", systemImage: "checkmark")
            }
            .padding()
```

```
            .tint(.red)
            .buttonStyle(.bordered)
            .buttonBorderShape(.automatic) // 自动

            Button {
            } label: {
                Label("NO", systemImage: "xmark")
            }
            .padding()
            .tint(.red)
            .buttonStyle(.bordered)
            .buttonBorderShape(.capsule) // 胶囊形状

            Button {
            } label: {
                Label("YES", systemImage: "checkmark")
            }
            .padding()
            .tint(.red)
            .buttonStyle(.bordered)
            .buttonBorderShape(.roundedRectangle) // 圆角矩形

            Button {
            } label: {
                Label("NO", systemImage: "xmark")
            }
            .padding()
            .tint(.red)
            .buttonStyle(.bordered)
            .buttonBorderShape(.roundedRectangle(radius: 10)) // 自定义圆角的圆角矩形
        }
    }
}
```

controlSize

重写 View 默认的大小。

```
struct ContentView: View {
    var body: some View {
        VStack(spacing: 20) {
            Button {
            } label: {
                Label("Love", systemImage: "heart")
            }
            .buttonStyle(.borderedProminent)
            .tint(.green)
            .controlSize(.large) // 默认是regular（大小）

            Button {
            } label: {
                Label("Love", systemImage: "heart")
            }
            .buttonStyle(.borderedProminent)
            .tint(.red)
            .controlSize(.regular)

            Button {
            } label: {
                Label("Love", systemImage: "heart")
            }
            .buttonStyle(.borderedProminent)
            .tint(.purple)
            .controlSize(.mini)

            Button {
            } label: {
                Label("Love", systemImage: "heart")
            }
            .buttonStyle(.borderedProminent)
            .tint(.blue)
            .controlSize(.small)
        }
    }
```

```
}
```

keyboardShortcut

可以给 View 添加快捷键操作，在模拟器测试时需要打开 **I/O —> Input —> Send Keyboard Input to Device**。

```
struct ContentView: View {
    var body: some View {
        VStack {
            Button("Command + C") {
                print("Command + C")
            }
            .keyboardShortcut("C", modifiers: [.command])

            Button("Shift + C") {
                print("Shift + C")
            }
            .keyboardShortcut("C", modifiers: [.shift])

            Button("Option + C") {
                print("Option + C")
            }
            .keyboardShortcut("C", modifiers: [.option])

            Button("Command + Option + C") {
                print("Command + Option + C")
            }
            .keyboardShortcut("C", modifiers: [.command, .option])
        }
    }
}
```

submitLabel

输入时（TextField、SecureField、TextEditor）键盘右下角按键的文字，默认为 `Return`，现在可以是 `Continue、Done、Go、Join、Next、Return、Route、Search、Send`，点击可以退掉键盘。

```
struct ContentView: View {
    @State private var username = ""
    @State private var password = ""

    var body: some View {
        VStack {
            TextField("用户名", text: $username)
                .textFieldStyle(.roundedBorder)
                .submitLabel(.done)

            SecureField("密码", text: $password)
                .textFieldStyle(.roundedBorder)
                .submitLabel(.return)
        }
        .padding()
    }
}
```

focused

- 绑定 View 的焦点，可以切换第一响应者，可用于在多个输入框之间切换。
- 同一个 View 可以有多个焦点绑定，但不能将相同的值绑定给多个 View。

```
// 定义Field
enum Field: Hashable {
    case username
    case password
}

struct ContentView: View {
    @State private var username = ""
    @State private var password = ""
    // @FocusState表示获取焦点的View，配合focused进行更新
```

```
    @FocusState private var focusedField: Field?

    var body: some View {
        VStack {
            TextField("用户名", text: $username)
                .textFieldStyle(.roundedBorder)
                .focused($focusedField, equals: .username)

            SecureField("密码", text: $password)
                .textFieldStyle(.roundedBorder)
                .focused($focusedField, equals: .password)

            Button("登录") {
                if username.isEmpty {
                    focusedField = .username
                } else if password.isEmpty {
                    focusedField = .password
                } else {
                    print(username, password)
                }
            }
        }
        .padding()
    }
}
```

onSubmit

- 按下键盘的 `Return`（或 submitLabel 指定的文字）时触发，用于 Form、TextField、Search Bar 提交数据。改造上面的案例，用 onSubmit 替换 Button。

```
enum Field: Hashable {
    case username
    case password
}

struct ContentView: View {
    @State private var username = ""
    @State private var password = ""
    @FocusState private var focusedField: Field?

    var body: some View {
        VStack {
            TextField("用户名", text: $username)
                .textFieldStyle(.roundedBorder)
                .focused($focusedField, equals: .username)

            SecureField("密码", text: $password)
                .textFieldStyle(.roundedBorder)
                .focused($focusedField, equals: .password)
        }
        .padding()
        .onSubmit {
            if username.isEmpty {
                focusedField = .username
            } else if password.isEmpty {
                focusedField = .password
            } else {
                print(username, password)
            }
        }
    }
}
```

- 通过 `submitScope` 设置触发条件。

```
struct ContentView: View {
    @State private var phone = ""

    var body: some View {
        NavigationView {
            VStack {
                TextField("手机号", text: $phone)
```

```
                .textFieldStyle(.roundedBorder)
                .submitLabel(.send)
                .submitScope(phone.count != 11) // 只有不等于11的时候提交才起作用
                .padding()
            }
            .navigationTitle("触发条件")
            .onSubmit(of: .text) { // 还可以是.search
                print("提交数据")
            }
        }
    }
}
```

privacySensitive

隐藏敏感信息，iOS 在调出后台 App 时，有些 App 如支付宝有一层模糊效果，使用它可以实现类似的效果。

- 基本使用。

```
struct ContentView: View {
    var body: some View {
        VStack {
            Text("银行卡")

            Text("1234 5678 9876 5432 10")
                .privacySensitive()
        }
    }
}
```

- 自定义显示效果。

```
struct ContentView: View {
    @Environment(\.redactionReasons) var redactionReasons

    var body: some View {
        VStack {
            if redactionReasons.contains(.privacy) {
                Color
                    .cyan
            } else {
                VStack {
                    Text("银行卡")

                    Text("1234 5678 9876 5432 10")
                }
            }
        }
        .ignoresSafeArea()
    }
}
```

EllipticalGradient

新增的椭圆渐变器，可以绘制椭圆的径向渐变。

```
struct ContentView: View {
    var body: some View {
        VStack {
            RoundedRectangle(cornerRadius: 10)
                .foregroundStyle(EllipticalGradient(colors: [.blue, .green]))
                .frame(height: 100)
                .padding()

            RoundedRectangle(cornerRadius: 10)
                .foregroundStyle(EllipticalGradient(
                    gradient: .init(colors: [.blue, .green]),
                    center: .topLeading)) // 圆心
                .frame(height: 100)
                .padding()

            RoundedRectangle(cornerRadius: 10)
```

```
                .foregroundStyle(EllipticalGradient(
                    gradient: .init(colors: [.blue, .green]),
                    center: .topLeading,
                    startRadiusFraction: 0, // 开始半径，0～1，0表示圆心，1表示直径
                    endRadiusFraction: 1)) // 结束半径，0～1，0表示圆心，1表示直径
                .frame(height: 100)
                .padding()
        }
    }
}
```

Material

可以与 background 搭配，实现 Blur Effect（模糊效果）。

- 案例一

```
struct ContentView: View {
    var body: some View {
        ZStack {
            Image(systemName: "heart.fill")
                .resizable()
                .foregroundStyle(.red)
                .frame(width: 300, height: 300)

            HStack {
            }
            .frame(width: 300, height: 300)
            .background(.ultraThinMaterial)
        }
    }
}
```

- 案例二

```
struct ContentView: View {
    var body: some View {
        ZStack {
            Color.red

            VStack {
                Label("手机", systemImage: "iphone")
                    .padding()
                    .background(RoundedRectangle(cornerRadius: 10).fill(.white))

                // 超细
                Label("手机", systemImage: "iphone")
                    .padding()
                    .background(.ultraThinMaterial, in: RoundedRectangle(cornerRadius: 10))

                // 细
                Label("手机", systemImage: "iphone")
                    .padding()
                    .background(.thinMaterial, in: RoundedRectangle(cornerRadius: 10))

                // 正常
                Label("手机", systemImage: "iphone")
                    .padding()
                    .background(.regularMaterial, in: RoundedRectangle(cornerRadius: 10))

                // 粗
                Label("手机", systemImage: "iphone")
                    .padding()
                    .background(.thickMaterial, in: RoundedRectangle(cornerRadius: 10))

                // 超粗
                Label("手机", systemImage: "iphone")
                    .padding()
                    .background(.ultraThickMaterial, in: RoundedRectangle(cornerRadius: 10))
            }
        }
    }
}
```

searchable

用于搜索，默认需要下拉一下才能出现搜索框。

```
struct ContentView: View {
    // 数据集
    let names = ["ZhangSan", "LiSi", "WangWu", "ZhaoLiu"]
    // 搜索内容
    @State private var queryString = ""
    // 过滤集
    var filteredNames: [String] {
        if queryString.isEmpty {
            return names
        } else {
            return names.filter { $0.localizedCaseInsensitiveContains(queryString) }
        }
    }

    var body: some View {
        NavigationView {
            List(filteredNames, id: \.self) { name in
                Text(name)
            }
            .navigationTitle("搜索")
        }
        // 搜索框提示文字，搜索框输入的内容，会一直出现搜索框
        // .searchable(text: $queryString, placement: .navigationBarDrawer(displayMode: .always), prompt: "搜索关键字")
        // 下拉出现搜索框
        // .searchable(text: $queryString, prompt: "搜索关键字")
        // 下拉出现搜索框，有搜索建议
        .searchable(text: $queryString, prompt: "搜索关键字", suggestions: {
            ForEach(filteredNames, id: \.self) { name in // 搜索建议
                Text(name)
                    .searchCompletion(name)
            }
        })
    }
}
```

listRowSeparator

用于设置 List 分割线的颜色或者隐藏（以前需要借助于 UITableView）。

```
struct ContentView: View {
    var body: some View {
        List {
            ForEach(0 ..< 10) {
                Text("Row \($0)")
                    // .listRowSeparatorTint(.red) // 颜色
                    .listRowSeparator(.hidden) // 隐藏
            }
        }
    }
}
```

listSectionSeparator

用于设置 Section 的分割线的颜色或者隐藏。

```
struct ContentView: View {
    var body: some View {
        List {
            Section {
                ForEach(0 ..< 5) {
                    Text("Row \($0)")
                }
            } header: {
                Text("Header1")
            } footer: {
                Text("Footer1")
            }
```

```
            Section {
                ForEach(6 ..< 10) {
                    Text("Row \($0)")
                }
            } header: {
                Text("Header2")
            } footer: {
                Text("Footer2")
            }
            .listSectionSeparatorTint(.red, edges: .bottom)

            Section {
                ForEach(10 ..< 15) {
                    Text("Row \($0)")
                }
            } header: {
                Text("Header3")
            } footer: {
                Text("Footer3")
            }
            .listSectionSeparator(.hidden, edges: [.top, .bottom])
        }
        .listStyle(.grouped) // 分组时有效果
    }
}
```

swipeActions

用于设置 List 左右侧滑时的菜单。

```
struct ContentView: View {
    var body: some View {
        List {
            ForEach(0 ..< 10) {
                Text("Row \($0)")
            // allowsFullSwipe: 默认为true, 表示滑动出现菜单以后再拉一次会自动执行第一个操作
            .swipeActions(edge: .trailing, allowsFullSwipe: true) { // 右滑
                // 菜单一
                Button("删除", role: .destructive) {
                    print("删除数据")
                }

                // 菜单二
                Button("增加") {
                    print("增加数据")
                }
                .tint(.blue)
            }
            .swipeActions(edge: .leading, allowsFullSwipe: true) { // 左滑
                Button {
                    print("收藏成功")
                } label: {
                    Label("收藏", systemImage: "star.fill")
                }
                .tint(.green)
            }
        }
    }
}
```

badge

- 设置角标。

```
struct ContentView: View {
    @State private var showBadge = false

    var body: some View {
        TabView {
            Text("微信")
```

```
            .tabItem {
                Image(systemName: "message")
                Text("微信")
            }
            .onTapGesture {
                showBadge.toggle()
            }
            .badge(showBadge ? 1 : 0) // 可以是数字，0时不显示

        Text("通讯录")
            .tabItem {
                Image(systemName: "person.2")
                Text("通讯录")
            }
            .badge("new") // 可以是字符串

        Text("发现")
            .tabItem {
                Image(systemName: "safari")
                Text("发现")
            }
            .badge(Text("10")) // 可以是Text
    }
  }
}
```

- List 中使用时，显示的效果不是传统的角标形态。

```
struct ContentView: View {
    var body: some View {
        List {
            ForEach(0 ..< 10) { index in
                Text("Row \(index)")
                    .badge(index.isMultiple(of: 2) ? Text("\(index)").foregroundColor(.red) : Text(Image(systemName: "heart")
            }
        }
    }
}
```

---

interactiveDismissDisabled

默认情况下，Modal 出来的界面即可以使用下拉手势关闭，又可以通过手动方式关闭。使用它可以设置只能通过手动方式关闭。

```
struct ModelView: View {
    @Environment(\.presentationMode) var presentationMode
    @State private var toggleOn = false

    var body: some View {
        VStack {
            Text("新界面")

            Button("手动关闭") {
                self.presentationMode.wrappedValue.dismiss()
            }

            Toggle("关闭", isOn: $toggleOn)
        }
        // .interactiveDismissDisabled() // 不能下拉关闭，只能点击按钮关闭
        .interactiveDismissDisabled(!toggleOn) // 打开开关就可以手势关闭
    }
}

struct ContentView: View {
    @State private var isModal = false

    var body: some View {
        Button("弹出新界面") {
            self.isModal = true
        }
        .sheet(isPresented: $isModal, content: {
            ModelView()
        })
    }
```

```
}
```

task

当 View 显示时可以用它执行异步操作，当 View 消失时任务会取消。

```
struct ContentView: View {
    let url = "https://www.baidu.com"
    @State private var content = "正在加载"

    var body: some View {
        NavigationView {
            ScrollView {
                Text(content)
                    .task { // task
                        guard let url = URL(string: url) else { return }
                        do {
                            let (data, _) = try await URLSession.shared.data(from: url) // 异步
                            content = String(decoding: data, as: UTF8.self)
                        } catch {
                            content = "加载失败"
                        }
                    }
            }
            .navigationTitle("搜索一下")
        }
    }
}
```

refreshable

下拉刷新，目前仅支持 List。

```
struct ContentView: View {
    @State private var names = ["zhangsan", "lisi", "wangwu"]

    var body: some View {
        List {
            ForEach(names, id: \.self) { name in
                Text(name)
            }
        }
        .refreshable {
            await loadMore()
        }
    }

    func loadMore() async {
        // 2秒延迟
        guard let url = URL(string: "https://httpbin.org/delay/2") else { return }
        let request = URLRequest(url: url)
        _ = try! await URLSession.shared.data(for: request)
        names.insert("yungfan", at: 0)
    }
}
```

1. 一旦下拉，就会调用 `refreshable` 闭包内的操作。
2. 应该在闭包内使用 await 操作进行数据的加载。
3. 在 await 操作完成之前，刷新控件会一直显示。

自定义

要使 View 获得下拉刷新的功能，需要做 3 件事：

1. 监听环境变量 `refresh` 。
2. 启动刷新操作。
3. 添加刷新状态。

```
struct RefreshView: View {
    // 环境变量
    @Environment(\.refresh) private var refresh
```

```
    // 加载状态
    @State private var isLoading = false

    var body: some View {
        // 显示刷新控件
        if let refresh = refresh {
            if isLoading {
                HStack {
                    ProgressView()
                        .padding(.horizontal, 4)
                        .tint(.primary)

                    Text("正在刷新")
                        .foregroundColor(.primary)
                }
            } else {
                Button {
                    isLoading = true
                    async {
                    // 启动刷新操作
                    await refresh()
                    isLoading = false
                    }
                } label: {
                    Text("点击刷新")
                        .foregroundColor(.primary)
                }
            }
        } else {
            Text("refresh is nil")
        }
    }
}

struct ContentView: View {
    @State private var color = Color.white

    var body: some View {
        ZStack {
            color

            RefreshView()
                .frame(width: UIScreen.main.bounds.width, height: 40)
                .refreshable {
                    let request = URLRequest(url: URL(string: "https://httpbin.org/delay/2")!)
                    _ = try! await URLSession.shared.data(for: request)

                    color = Color(red: .random(in: 0 ... 1), green: .random(in: 0 ... 1), blue: .random(in: 0 ... 1))
                }
        }
    }
}
```

Update

environment(.openURL)

- 以前 openURL 是只读的，现在可读可写，这样开发者就可以对 URL 进行自定义处理。
- 基本使用。

```
struct ContentView: View {
    let bd = URL(string: "https://www.baidu.com")!
    let wwdc21 = URL(string: "https://developer.apple.com/wwdc21/")!

    @Environment(\.openURL) var openURL

    var body: some View {
        Link("百度", destination: bd)
            // OpenURLAction
            .environment(\.openURL, OpenURLAction { _ in
                // 默认行为，效果和以前一样，这里是直接调用Safari打开
                .systemAction
```

```
        })

    Link("WWDC21", destination: wwdc21)
        .environment(\.openURL, OpenURLAction { _ in
            // handleURL(url)

            // 表示成功处理，可以点击，但不触发任何行为，需要自行处理
            .handled
        })

    Text("[开发者网站](https://developer.apple.com)")
        .environment(\.openURL, OpenURLAction { _ in
            // handleURL(url)

            // 表示不能被处理，默认无法点击，需要自行处理
            .discarded
        })

        Link("重定向", destination: bd)
            // OpenURLAction
            .environment(\.openURL, OpenURLAction { _ in
                // 重定向
                .systemAction(URL(string: "https://www.qq.com")!)
            })
    }

    func handleURL(_ url: URL) {
        openURL.callAsFunction(url) { accepted in
            print(accepted)
        }
    }
}
```

- 案例。

```
struct ContentView: View {
    @State private var turnToLogin: Bool = false
    @State private var turnToRegister: Bool = false

    var login: some View {
        ZStack {
            Color.orange

            Text("登录界面")
        }
    }

    var register: some View {
        ZStack {
            Color.green

            Text("注册界面")
        }
    }

    var body: some View {
        Text("[登录](sign-in) 或者 [注册](create-account)")
            .environment(\.openURL, OpenURLAction { url in
                switch url.absoluteString {
                case "sign-in":
                    // 登录界面
                    turnToLogin = true
                    return .handled
                case "create-account":
                    // 注册界面
                    turnToRegister = true
                    return .handled
                default:
                    return .discarded
                }
            })
            .sheet(isPresented: $turnToLogin, content: {
                self.login
            })
            .sheet(isPresented: $turnToRegister, content: {
```

```
                self.register
            })
    }
}
```

contentShape

该 Modifier 在 iOS 13 已经推出，控制 View 的可点击区域。在 iOS 15 中新增了一种构造函数，可以进一步设置 View 的交互形状（非渲染形状）。

```
// iOS 13
public func contentShape<S>(_ shape: S, eoFill: Bool = false) -> some View where S : Shape
// iOS 15
public func contentShape<S>(_ kind: ContentShapeKinds, _ shape: S, eoFill: Bool = false) -> some View where S : Shape
```

```
struct ContentView: View {
    var body: some View {
        Button("设置") {
            print("设置")
        }
        .hoverEffect(.lift) // iPadOS运行才有效果
        .contentShape(.hoverEffect, Circle()) // 有多种效果可以选择
        // .contentShape([.hoverEffect, .dragPreview], Circle())] // 可以组合使用

        Text("菜单")
            .font(.title)
            .foregroundStyle(.red)
            .contextMenu {
                Button("删除") {
                    print("删除")
                }

                Button("增加") {
                    print("增加")
                }

                Button("更新") {
                    print("更新")
                }
            }
            .contentShape(.contextMenuPreview, Circle())
    }
}
```

Deprecated

animation

- 现在使用时必须增加 value 参数，传入产生动画的那个变量。
- 使用场景：只有某个参数发生变化了才会产生动画。
- 兼容到 iOS 14。

```
struct ContentView: View {
    @State private var isScale = false

    var body: some View {
        VStack(spacing: 20) {
            Image(systemName: "paperplane")
                .scaleEffect(isScale ? 2 : 1)
                // .animation(.default) // deprecated
                .animation(.default, value: isScale) // new
                .onTapGesture {
                    isScale.toggle()
                }
        }
    }
}
```

Environment

SwiftUI 3.0 新增了很多新的环境变量用于不同的用途。

dismiss

用于替换 `presentationMode`，用于关闭 Modal 界面。

```
struct ModelView: View {
    @Environment(\.dismiss) var dismiss

    var body: some View {
        VStack {
            Text("新界面")

            Button("手动关闭") {
                dismiss()
            }
        }
    }
}

struct ContentView: View {
    @State private var isModal = false

    var body: some View {
        Button("弹出新界面") {
            self.isModal = true
        }
        .sheet(isPresented: $isModal, content: {
            ModelView()
        })
    }
}
```

isSearching与dismissSearch

- `isSearching`：用于监听用户的搜索状态。
- `dismissSearch`：用于关闭搜索，触发时：
  - isSearching 将设置为 false。
  - 清除所有搜索文本。
  - 任何搜索字段将失去焦点。

```
struct ContentView: View {
    let names = ["ZhangSan", "LiSi", "WangWu", "ZhaoLiu"]
    @State private var queryString = ""
    var filteredNames: [String] {
        if queryString.isEmpty {
            return names
        } else {
            return names.filter { $0.localizedCaseInsensitiveContains(queryString) }
        }
    }

    var body: some View {
        NavigationView {
            VStack {
                List(filteredNames, id: \.self) { name in
                    Text(name)
                }

                SearchContent()
            }
            .navigationTitle("搜索")
            .searchable(text: $queryString, prompt: "搜索关键字")
        }
    }
```

```
}

struct SearchContent: View {
    @Environment(\.isSearching) var isSearching
    @Environment(\.dismissSearch) var dismissSearch

    var body: some View {
        VStack {
            Text(isSearching ? "正在搜索" : "没有搜索")

            Button("取消") {
                dismissSearch()
            }
        }
    }
}
```

controlSize

用于读取按钮的 `controlSize` 。

```
struct ContentView: View {
    @Environment(\.controlSize) private var controlSize: ControlSize

    var body: some View {
        Button("注册") {}
            .buttonStyle(.bordered)
            .controlSize(.small)
            .buttonStyle(CustomButtonStyle())

        Button("登录") {}
            .buttonStyle(.bordered)
            .controlSize(.large)
            .buttonStyle(CustomButtonStyle())
    }
}

struct CustomButtonStyle: ButtonStyle {
    @Environment(\.controlSize) var controlSize: ControlSize

    func makeBody(configuration: Configuration) -> some View {
        configuration
            .label
            .padding(controlSize == .large ? 16 : 0) // 自定义边距
    }
}
```

keyboardShortcut

用于读取分配给 View 的快捷键。

```
struct ContentView: View {
    var body: some View {
        VStack {
            Button("Command + B") {
            }
            .keyboardShortcut("B", modifiers: [.command])

            Button("Command + C") {
            }
            .keyboardShortcut("C", modifiers: [.command])
        }
        .buttonStyle(CustomButtonStyle())
    }
}

struct CustomButtonStyle: ButtonStyle {
    @Environment(\.keyboardShortcut) var shortcut: KeyboardShortcut?

    func makeBody(configuration: Configuration) -> some View {
        configuration.label
```

```
            .font(.body.weight(shortcut == .init("B") ? .heavy : .regular)) // 如果是B，则加粗显示
            .foregroundColor(shortcut == .init("C") ? .red : .primary) // 如果是C，则文字显示红色
    }
}
```

跨平台~new

系统判断

SwiftUI 在进行跨平台开始时肯定会使用到条件 Modifier，之前的解决方案是自己写一套判断体系。得益于 Swift 5.5 中 “**条件编译支持后缀成员表达式**” 的新特性，现在可以直接在条件编译后面书写 Modifier，跨平台更加方便。

```
struct ContentView: View {
    var body: some View {
        Text("Swift 5.5")
        #if os(iOS)
            .font(.title3)
            .foregroundColor(.blue)
        #elseif os(macOS)
            .font(.title2)
            .foregroundColor(.green)
        #else
            .font(.title)
            .foregroundColor(.pink)
        #endif
    }
}
```

网络编程

Swift 5.5 推出了新的并发编程技术 — Concurrency，URLSession 和 SwiftUI 3.0 也推出了支持该特性的 API， SwiftUI 中的网络编程可以更加精简。

核心

1. 必须使用新的 URLSession API 进行网络数据的请求。
2. 通常需要在 ObservableObject 中定义异步方法进行网络数据的请求。
3. ObservableObject 需要标记为 `@MainActor` 。
4. 在 View 显示的时候使用 `task` Modifier（以前用 `onAppear` ），在其内部调用 ObservableObject 中的异步方法。
5. 需要时可以使用 `refreshable` Modifier 进行下拉刷新，它的内部可以像 `task` 一样调用异步方法。
6. 需要时可以使用 `Task` 在同步的上下文开启一个异步任务，在其内部也可以进行异步方法的调用。

头条新闻

- Model：与服务器返回的 JSON “保持一致”。

```
struct NewsModel: Codable {
    var reason: String
    var error_code: Int
    var result: Result
}

struct Result: Codable {
    var stat: String
    var data: [DataItem]
}

struct DataItem: Codable, Hashable {
    var title, date, category, author_name, url, thumbnail_pic_s: String
}
```

- Network：使用 URLSession 新增的支持 Concurrency 语法的 API，代码变得极其简单。

```
class Network {
    let decoder = JSONDecoder()
    let urlSession = URLSession.shared
    let url: URL

    init(url: String) {
        self.url = URL(string: url)!
    }

    // async异步方法
    // 注意这里用的是泛型函数
    public func get<T: Codable>() async throws -> T {
        // URLSession提供了支持URL和URLRequest的2种API
        let (data, _) = try await urlSession.data(from: url, delegate: nil)
        let object = try decoder.decode(T.self, from: data)
        return object
    }
}
```

- ViewModel：调用 Network 取出需要的数据，需要使用 `@MainActor` 标记。

```
@MainActor
class ViewModel: ObservableObject {
    @Published var news: [DataItem] = []
    var page: Int = 1

    // 异步方法中调用异步方法
    func getNews() async -> [DataItem] {
        do {
            // await调用异步方法
            // 泛型函数调用时直接通过变量的类型指定，否则需要强制转换
            let data: NewsModel = try await Network(url: "http://v.juhe.cn/toutiao/index?type=top&key=申请的key&page=\(page)&p
```

```
            return data.result.data
        } catch {
            fatalError(error.localizedDescription)
        }
    }
}
```

- List 的 Row：图片采用 AsyncImage 进行异步加载，其他部分保持不变。

```
struct NewsRow: View {
    var dataItem: DataItem

    var newsImage: some View {
        AsyncImage(url: URL(string: dataItem.thumbnail_pic_s)) { phase in
            switch phase {
            case .empty: // 未加载
                ProgressView()
                    .frame(width: 100, height: 80, alignment: .center)
            case let .success(image): // 加载成功
                image
                    .resizable()
                    .frame(width: 100, height: 80)
            case .failure: // 加载失败
                Image(systemName: "photo")
                    .resizable()
                    .frame(width: 100, height: 80)
                    .foregroundStyle(.green)
            @unknown default:
                EmptyView()
            }
        }
    }

    var body: some View {
        HStack {
            newsImage

            VStack(alignment: .leading) {
                Text(dataItem.title)
                    .font(.title3)
                    .foregroundStyle(.red)
                    .lineLimit(5)

                Spacer()

                HStack {
                    Text(dataItem.author_name)
                        .font(.subheadline)
                        .foregroundStyle(.black)

                    Spacer()

                    Text(dataItem.date.prefix(16).suffix(11))
                        .font(.subheadline)
                        .foregroundStyle(.gray)
                }
            }
            .padding(8)
        }
    }
```

- ContentView：改用 `task` 进行首次数据加载， `refreshable` 进行下拉刷新， `Task` 进行手动加载，同时实现了滑动自动加载更多数据。

```
struct ContentView: View {
    @StateObject var viewModel = ViewModel()

    var body: some View {
        NavigationView {
            List {
                ForEach(viewModel.news, id: \.self) { news in
                    NewsRow(dataItem: news)
                        .task { // 滑到最后一条数据时加载
                            if news == viewModel.news.last {
```

```
                        viewModel.page += 1
                        let moreNews = await viewModel.getNews()
                        // 追加数据
                        viewModel.news += moreNews
                    }
                }
            }
        }
        .task { // 启动加载
            viewModel.news = await viewModel.getNews()
        }
        .refreshable { // 下拉刷新
            await refreshData()
        }
        .toolbar {
            ToolbarItem(placement: .navigationBarTrailing) {
                Button { // 手动刷新
                    Task {
                        await refreshData()
                    }
                } label: {
                    Image(systemName: "arrow.clockwise")
                }
            }
        }
        .listStyle(.plain)
        .navigationBarTitle("头条新闻")
    }
}

func refreshData() async {
    print(viewModel.news.count)
    viewModel.page = 1
    let moreNews = await viewModel.getNews()
    // 如果有新数据则插入
    for item in moreNews where !viewModel.news.contains(item) {
        viewModel.news.insert(item, at: 0)
    }
}
```

天气预报

```
/// Model层
struct Weather: Codable {
    let reason: String
    let result: Result
    let error_code: Int
}

struct Result: Codable {
    let city: String
    let realtime: Realtime
    let future: [Future]
}

struct Future: Codable, Identifiable {
    let id = UUID()
    let date, temperature, weather, direct: String
    let wid: Wid
}

struct Wid: Codable {
    let day, night: String
}

struct Realtime: Codable {
    let temperature, humidity, info, wid, direct, power, aqi: String
}

/// Network层
// 一个类专门用于负责API网络请求
class Client {
```

```
    let decoder = JSONDecoder()
    let urlSession = URLSession.shared

    func getWeather<T: Codable>() async throws -> T? {
        guard let urlString = "http://apis.juhe.cn/simpleWeather/query?city=芜湖&key=申请的key".addingPercentEncoding(withAllow
        let (data, _) = try await urlSession.data(from: url, delegate: nil)
        print(data)
        do {
            let object = try decoder.decode(T.self, from: data)
            return object
        } catch {
            fatalError(error.localizedDescription)
        }
    }
}

/// Data层
@MainActor
class DataStore: ObservableObject {
    @Published var futures: [Future]? = [Future]()

    func fetchWeathers() async {
        do {
            let data: Weather? = try await Client().getWeather()
            futures = data?.result.future
        } catch {
            fatalError(error.localizedDescription)
        }
    }
}

/// View层
struct ContentView: View {
    @ObservedObject var store = DataStore()

    var body: some View {
        NavigationView {
            if let weathers = store.futures {
                List(weathers) { weather in
                    HStack {
                        Text(weather.date)

                        Text(weather.weather)
                            .foregroundStyle(.green)

                        Spacer()

                        Text(weather.temperature)
                            .foregroundStyle(.gray)
                    }
                    .font(.system(size: 18))
                }
                .task {
                    await store.fetchWeathers()
                }
                .navigationBarTitle("天气")
            }
        }
    }
}
```

进一步封装

从前面的案例可以看出，获取模型数据的步骤都是先通过 URLSession 新的 API 获取 Data，然后通过 Codable 进行转换，因此可以进一步封装，使用起来则会更加简单。

- 封装。

```
extension URLSession {
    // MARK: URL
    func decode<T: Decodable>(
        _ type: T.Type = T.self,
        from url: URL,
```

```
        keyDecodingStrategy: JSONDecoder.KeyDecodingStrategy = .useDefaultKeys,
        dataDecodingStrategy: JSONDecoder.DataDecodingStrategy = .deferredToData,
        dateDecodingStrategy: JSONDecoder.DateDecodingStrategy = .deferredToDate
    ) async throws -> T {
        let (data, _) = try await data(from: url)

        let decoder = JSONDecoder()
        decoder.keyDecodingStrategy = keyDecodingStrategy
        decoder.dataDecodingStrategy = dataDecodingStrategy
        decoder.dateDecodingStrategy = dateDecodingStrategy

        let decoded = try decoder.decode(T.self, from: data)
        return decoded
    }

    // MARK: URLRequest
    func decode<T: Decodable>(
        _ type: T.Type = T.self,
        for request: URLRequest,
        keyDecodingStrategy: JSONDecoder.KeyDecodingStrategy = .useDefaultKeys,
        dataDecodingStrategy: JSONDecoder.DataDecodingStrategy = .deferredToData,
        dateDecodingStrategy: JSONDecoder.DateDecodingStrategy = .deferredToDate
    ) async throws -> T {
        let (data, _) = try await data(for: request)

        let decoder = JSONDecoder()
        decoder.keyDecodingStrategy = keyDecodingStrategy
        decoder.dataDecodingStrategy = dataDecodingStrategy
        decoder.dateDecodingStrategy = dateDecodingStrategy

        let decoded = try decoder.decode(T.self, from: data)
        return decoded
    }
}
```

- 使用。

```
Task {
    let newsModel = try await URLSession.shared.decode(NewsModel.self, from: URL(string: "http://v.juhe.cn/toutiao/index?type
    print(newsModel.result.data)
}

Task {
    guard let urlString = "http://apis.juhe.cn/simpleWeather/query?city=芜湖&key=申请的key".addingPercentEncoding(withAllowedCh
    let request = URLRequest(url: urlString)
    let weather = try await URLSession.shared.decode(Weather.self, for: request)
    print(weather.result.future)
}
```

CoreData~new

Swift 5.5 推出了 Concurrency 的异步编程方式，针对 CoreData 推出了可以进行**分段查询**的属性包装 `@SectionedFetchRequest` 。下面改造 SwiftUI 2.0 中的案例。

- ContentView。

```
import CoreData
import SwiftUI

struct ContentView: View {
    @Environment(\.managedObjectContext) private var viewContext

    @SectionedFetchRequest<String, Item>(
        sectionIdentifier: \.sex!, // 按照某个字段进行分组
        sortDescriptors: [NSSortDescriptor(keyPath: \Item.name, ascending: true)],
        animation: .default)
    private var sectionedItem: SectionedFetchResults<String, Item> // 类型为SectionedFetchResults

    var body: some View {
        VStack {
            List {
                ForEach(sectionedItem) { section in
                    Section(header: Text(section.id)) { // 先找分组，即sectionIdentifier指定的字段
                        ForEach(section) { person in // 再设置分组内容
                            VStack {
                                Text(person.name!)

                                Text("\(person.age)")
                            }
                        }
                        .onDelete(perform: deleteItems)
                    }
                }
            }

            Button(action: addItem) {
                Text("增加")
            }

            Button(action: updateItem) {
                Text("修改")
            }
        }
    }

    private func addItem() {
        withAnimation {
            let newItem = Item(context: viewContext)
            newItem.id = UUID()
            newItem.name = "lisi"
            newItem.sex = "男"
            newItem.age = 20

            PersistenceController.shared.save()
        }
    }

    private func updateItem() {
        withAnimation {
            let newItem = sectionedItem.first?.first
            newItem?.name = "zhangsan"
            newItem?.age = 100

            PersistenceController.shared.save()
        }
    }

    private func deleteItems(offsets: IndexSet) {
        withAnimation {
            offsets.map { (sectionedItem.first?[$0])! as NSManagedObject }.forEach(viewContext.delete)
```

```
            PersistenceController.shared.save()
        }
    }
}
```

- App 入口。

```
@main
struct SwiftUI3CoreDataApp: App {
    let persistenceController = PersistenceController.shared
    @Environment(\.scenePhase) var scenePhase

    var body: some Scene {
        WindowGroup {
            ContentView()
                .environment(\.managedObjectContext, persistenceController.container.viewContext)
        }
        .onChange(of: scenePhase) { _ in
            // 生命周期发生变化时要保存数据
            persistenceController.save()
        }
    }
}
```

Accessibility~new

Swift 3.0 提供了更加便捷的 Accessibility 辅助功能 API。

accessibilityRepresentation

能够很容易地替换现有 View 为 Accessibility View。

```
struct ContentView: View {
    @State var isSelected = false
    @State var isSelected2 = false

    var body: some View {
        VStack(spacing: 20) {
            Image(systemName: isSelected ? "checkmark.rectangle" : "rectangle")
                .resizable()
                .frame(width: 40, height: 40)
                .onTapGesture {
                    isSelected.toggle()
                }
                .accessibilityRepresentation { // 单个View
                    Toggle(isOn: $isSelected) {
                        Text("选中")
                    }
                }

            Image(systemName: isSelected ? "checkmark.rectangle" : "rectangle")
                .resizable()
                .frame(width: 40, height: 40)
                .onTapGesture {
                    isSelected.toggle()
                }
                .accessibilityRepresentation { // 复杂View
                    VStack {
                        Toggle(isOn: $isSelected) {
                            Text("选中")
                        }

                        Text("复杂View")
                            .foregroundColor(isSelected ? .blue : .red)
                    }
                }
        }
    }
}
```

accessibilityShowsLargeContentViewer

在 iOS 13 中 Accessibility 引入了 Large Content Viewer，效果是会放大/高亮显示用户正在访问的 View。这个特性从 Xcode 13.2 开始，SwiftUI 也可以使用。

```
struct ContentView: View {
    var body: some View {
        VStack {
            Button("Large Content Viewer") {
            }
            .accessibilityShowsLargeContentViewer()

            Button("Large Content Viewer") {
            }
            .accessibilityShowsLargeContentViewer {
                Text("替换为一个新元素")
            }
        }
    }
}
```

总结

SwiftUI 3.0 在 1.0 和 2.0 的基础上增加了一些新的内容，同时改进了一些内容。从 WWDC 21 Sessions 中的演示案例可以看出，SwiftUI 必定会越来越重要，也会越来越完善，不可否认，只要你继续深耕 iOS 开发领域，就一定要学习 SwiftUI。希望大家要自己尝试去动手实践并不断总结经验，将 SwiftUI 1.0 ～ 3.0 的知识融会贯通。

iOS开发课程

学习 SwiftUI 需要掌握 Swift 语法，还可能需要掌握基于 UIKit 进行 iOS 开发的知识，针对这些知识作者也已经发布了相应的视频教程，详情请查看 iOS 开发系列教程。

SwiftUI系列课程

SwiftUI 1.0 + SwiftUI 2.0 + Combine + Concurrency 是在学习本教程之前必学的内容，作者也已经发布了配套的视频教程，详情请查看精通 SwiftUI 开发。

源代码

教程配套源代码下载地址。