



SWIFTER

SwiftUI 与
Combine 编程
第二版

王巍 (@onevcat)

2.0 (2020 年 10 月)

© 2019~ ObjC 中国

版权所有

ObjC 中国

在中国地区独家翻译和销售授权

获取更多书籍或文章, 请访问 <https://objccn.io>

电子邮件: mail@objccn.io

1	简介	8
	前置要求 9	
	适合的读者人群 10	
	章节结构和推荐阅读方式 10	
	声明式的 UI 构建方式 13	
	SwiftUI 和 Combine 简介 17	
	练习 19	
2	你好，SwiftUI	22
	计算器 app 实例 22	
	创建项目和 Hello World 23	
	使用 Modifier 描述 Text 及 Button 27	
	基本布局和 Stack 容器 35	
	预览多尺寸以及适配 53	
	总结 55	
	练习 56	
3	数据状态和绑定	62
	Calculator 模型 63	
	@State 数据状态驱动界面 69	
	操作回溯和数据共享 79	
	总结 96	
	练习 97	

4	真实世界的 SwiftUI	101
	示例 app: PokeMaster 101	
	开始项目 103	
	创建列表 105	
	创建弹出面板 131	
	创建设置界面 145	
	总结 151	
	练习 152	
5	Combine 和异步编程	157
	异步编程简介 157	
	Combine 基础 160	
	总结 176	
	练习 176	
6	Publisher 和常见 Operator	182
	准备 182	
	基础 Publisher 和 Operator 185	
	错误处理 199	
	Publisher 的类型系统 209	
	总结 213	
	练习 214	
7	响应式编程边界	219
	Subject 行为 219	

响应式和指令式的桥梁 229
Foundation 中的 Publisher 232
订阅和绑定 238
Cancellable, AnyCancellable 和内存管理 246
总结 248
练习 248

8 SwiftUI 架构 255

UI 调整 255
Swift UI 的架构方式 258
通过 Binding 改变状态 260
通过 Action 改变状态 264
Command 和异步操作 275
总结 295
练习 296

9 SwiftUI 中的 Combine 301

复合状态驱动 UI 301
实际的网络请求 312
异步图片和图片缓存 327
总结 330
练习 331

10 手势处理和导航 339

手势处理 339
导航层级 351

支持 URL Scheme 364

总结 369

练习 370

11 用户体验和布局进阶 373

自定义绘制和动画 373

布局和对齐 387

总结 408

练习 408

简介

1

2019 年春夏之交的这场 WWDC，注定会成为深深烙印在 Apple 开发者们记忆中的一场盛会。自从 2014 年 Apple 发布 Swift 以来，还没有哪一次的 WWDC 能汇集到这么多人的目光。在这个充满新意和激情的舞台中央，被开发者们给予热情欢呼的主角毫无疑问只有一个，那就是 SwiftUI。

SwiftUI 作为 Apple 在自家平台使用 Swift 语言打造的首个重量级系统框架，将为这个平台上用户界面的构建方式带来革命性的转变。它摒弃了从上世纪八十年代开始就一直被使用的指令式 (imperative) 编程的方式，转而投向声明式 (declarative) 编程的阵营，这提高了我们解决问题时所需要着手的层级，从而让我们可以将更多的注意力集中到更重要的创意方面。

SwiftUI 充分利用了 Swift 先进简洁的语法，提供了一套完整而优美的领域特定语言 (Domain-Specific Language, DSL) 来描述 UI；在漂亮的外表背后，崭新的响应式编程框架 Combine 和早已深深植入于 Swift 中的函数式编程思想方式，两者相得益彰，共同驱动了 SwiftUI 的数据流向；在底层，成熟的 iOS 系统及强大的 UIKit 则保证了 SwiftUI 的高效渲染以及与现有特性的无缝衔接；而通用的语法和基础类型，有效降低了跨平台的难度，模糊了不同目标平台之间的界线，让开发者们更容易把 iOS app 带到 macOS，甚至带到 web 中去。

在本书里，我们会涉及到上面所提到的各个方面。你可以学习到如何使用 SwiftUI 来构建一个现代的 iOS app，包括使用 SwiftUI 提供的控件，配置合理的架构和数据流，按照函数响应式编程 (Functional reactive programming, FRP) 的方式来组织逻辑代码与 UI 代码，以及让 SwiftUI app 与现有的 UIKit 框架有机结合。

书中将提供一些引导教程，指导你动手实践和尝试不同的方案。在本书最后，你应该能够掌握使用 SwiftUI 和 Combine 开发 iOS app 的基本知识，并依靠教程中所积累的实际经验，独自进行 SwiftUI app 的开发。

前置要求

阅读本书，并且跟随实践书中的示例程序，你需要以下技能和环境：

- **至少入门级别的 SwiftUI 知识：**想要基于 SwiftUI 进行开发，我们只能使用 SwiftUI 语言，但本书篇幅有限，我们**并不会**在本书中专门介绍 SwiftUI 语言的基础语法和思想。对于某些程序语法上难以理解的部分，我们会进行说明，也会对 SwiftUI 和 Combine 框架中新加入的特定用法作出解释。但是构建项目时的其他部分的代码，需要你自己进行理解。如果你完全没有接触过 Swift，推荐你可以先阅读 [The Swift Programming Language](#) 中的 [教程部分](#) 的内容，你也可以找到这个教程的 [完整的中文翻译版本](#)。除此之外，我也推荐你了解一些 UIKit 开发的基础知识，比如 Model-View-Controller (MVC) 架构和一些 UIKit 控件的基本使用等。如果你已经是 Apple 平台的开发者，那么这个要求应该没有问题；如果你是从其他平台慕 SwiftUI 之名而来，也请放心：UIKit 的知识可以帮助你更好地理解 SwiftUI 中的一些概念，但它们并不是必须的。
- **运行 macOS 10.15.4 或之后版本的 mac 电脑：**Xcode 12 支持的最低系统版本是 macOS 10.15.4，如果你的系统版本过低，可能需要先进行升级（记得备份重要的资料）。我们推荐安装最新的 macOS 系统，因为随着 SwiftUI 的进化，会有一些在最新的系统和 SDK 中才能使用的 SwiftUI 功能。比如 SwiftUI 预览，它可以让你在 Xcode 中在编写代码的同时，实时展示结果。这将节省非常多的时间，并提供最好的 SwiftUI 开发体验。
- **Xcode 11 或之后的版本：**Xcode 提供了 Apple 平台的标准开发环境，从 Xcode 11 开始，它所提供的 SDK 中才包含有 SwiftUI 和 Combine 框架。你可以在 Mac App Store 搜索 Xcode 或者到 Apple Developer 网站进行下载。本书当前版本基于 Xcode 12，如果没有特殊理由，你应该始终选择最新的 Xcode 版本。

适合的读者人群

如上所述，本书面向的主要群体是对新一代 Apple 平台 UI 构建技术感兴趣的读者。

大致上目标读者的画像可以分为两类：

- **现有的 iOS 开发从业者**：你已经在日常工作中使用 UIKit 构建 app (不论是使用 Swift 还是 Objective-C)，在 SwiftUI 这一新技术出现时，你希望能够了解和学习这种不同以往的 UI 构建方式，并研究今后迁移到使用 SwiftUI 的可行性。SwiftUI 和 Combine 会为你带来全新的思考方式，就算暂时还没有办法使用这些新技术，它们也将成为重要的知识储备并帮助你改善现有的程序设计。
- **希望学习 iOS 开发的其他平台的开发者**：你可能会拥有一些响应式框架或者声明式编程的经验 (比如使用 React, Flutter 或者各类 Rx 框架) 并开发过一些客户端的项目；你可能单纯只是对 Swift 语言或者 iOS 开发感兴趣，而实际上做的是完全不相关的后端、运维、或者甚至产品经理的工作；你也可能只是刚刚进入到开发者的队伍还没有任何项目经验。相比于 iOS 开发中传统的 UIKit, SwiftUI 要简单得多。也就是说，SwiftUI 可能会是一个完美的入门 iOS 开发的契机。不过，我还是建议你先对 Swift 有部分了解后再继续。

如果你对 Swift 有兴趣，我们也准备了一些其他书籍，其中涵盖了关于函数式开发，语言进阶，优化以及 app 架构等很多方面的话题。你可以在「ObjC 中国」的 [在线商店](#) 找到所有这些图书：

<https://objccn.io/products/>

章节结构和推荐阅读方式

按照话题，本书被由浅入深地分为三个部分。

第一部分，SwiftUI 初步

在这部分中，我们首先会通过创建一个计算器 app 来介绍 SwiftUI 的基本概念，包括：通过使用 View 来描述 UI；基本控件和容器的使用；利用 modifier 来配置 UI；调整布局等内容。这些步骤让我们可以得到一个不错的用户界面。接下来，我们会实际使用事件和模型数据让计算器正常工作。

在完成这个相对简单的例子后，我们会构建一个更贴近于实际，也更复杂的例子。其中会涉及到 iOS app 中的 Navigation (导航)，List (列表)，动画以及其他一些 iOS 上常见的交互。通过这个复杂例子，你可以学习到如何使用 SwiftUI 构建出一个常见的 iOS app。

第二部分，Combine 框架

Combine 框架非常适合用来配合声明式的 SwiftUI，创建出稳定和单向的数据流动。相比于传统的 MVC 架构，使用在 Model 和 View 间使用数据绑定，并将底层事物交由系统处理，可以大大简化开发工作流程。我们会从 FRP 的基础开始，介绍 Combine 框架的基础概念，包括 Publisher, Subscriber, 各类 Operator 等。

Combine 在背后驱动了 SwiftUI 的数据流动。虽然就算不了解 Combine，我们也能写出合理的 SwiftUI app，但是对它进行了解，有助于我们更好地理解架构 app 的方式。这部分内容偏向于理论，它将为后面的实践内容提供基础和背景知识。

第三部分，综合实践

在最后一部分中，我们会使用之前准备的 app 的基本框架，以及我们学习到的 Combine 的相关知识，来完成示例 app 的实现。我们会使用响应式编程的方式实际地从网络 API 获取数据，并将它们绑定到 SwiftUI 上。你可以学习到如何将 Combine 框架中的概念运用到日常开发里，以此改善代码设计。

SwiftUI 还处于早期阶段，有很多不足之处。为了打造一款优秀的 app，我们有时候也需要和 UIKit 混用。SwiftUI 提供了展示 UIKit 内容的接口，可以让我们在遇到无法解决的问题时回滚到 UIKit。另外，像是错误处理、用户状态存储等话题，也会涵盖在本章中。

如何阅读本书

本书的章节之间有较强的相关性，即使你有 UIKit 开发的基础，或者有 RxSwift 等 FRP 的知识，也还是建议从头开始通读本书。书中内容假定你对 SwiftUI 几乎没有过接触和经验，并会以详细的示例引领你一步步在构建 app 中掌握必要知识。实际动手会对理解本书内容有很大帮助，因此最好是能够一边打开 Xcode 敲写代码一边阅读本书。

在每章结束后，我们为读者准备了若干的练习题以及一些挑战项目。挑战项目的内容一般是对于示例 app 的代码不足之处的一些补完任务，为了不限制思路，我们并没有提供标准答案：实践才是检验真理的途径。如果能够完成这些可选内容，相信会对加深理解大有裨益。

需要特别说明的是，书中为示例 app 提供的实现方式只是一个可以完成任务的参考。在本书写作时，SwiftUI 还是非常年轻的框架，大家对 SwiftUI 的热情探索才刚刚开始。在许多问题上，SwiftUI 框架本身还有欠缺，相关的最佳实践也都还没有形成，因此，书中提供的方式不一定是优秀的，甚至有时候还会显得笨拙。我十分推荐在每个小任务完成后，读者能够自己再进行思考，尝试去寻找出更优秀的解决方案。古有云，“弟子不必不如师，师不必贤于弟子”，阅读书籍亦是如此。笔者本身水平有限，书中也难免会有谬误，如有发现，还请不吝赐教，让我们加深理解，一起进步。

欢迎你到本书的公共 GitHub 仓库的 [issue 页面](#)自由发表你的看法，或者讨论各类问题。

接下来，我想先对本书的几个重要话题先做一些概要性的说明介绍。

声明式的 UI 构建方式

指令式编程

SwiftUI 是一个声明式的 UI 开发方式。在能够进一步之前，我们最好先弄清指令式和声明式两种编程方式的区别。

C, C++ 和部分更早期的语言遵从指令式编程的范式。一般来说，指令式编程支持三种语句：

1. **运算语句**：将某个值保存到变量中，计算某个表达式的结果，或者方法调用。
比如 `let a = 1 + 2` 就是一个标准的运算语句。
2. **循环语句**：在特定条件下反复运行某些语句，比如 `for` 和 `while`。
3. **条件语句**：如果某些条件成立时才运行某个区块的代码，否则就将省去。比如 `if` 和 `switch` 都是条件语句。

通过组合这些语句，我们在指令式编程中逐条指示计算机如何工作。早期的指令式编程语言都是针对计算机本身的机器或汇编语言。在这些语言中，指令的设计贴近计算机的运算硬件设计，这让硬件的运行更容易和高效。在随后的早期高级语言中，对这些指令语句的映射往往成为了设计语言的主流方向。虽然这有利于编译器的编写和最终的运行效率，但同时也阻碍了复杂程序的设计。

汇编距离我们有些遥远了，让我们回到 Swift，来看一个典型的指令式编程的例子。比如我们有一个学生系统，记录了学生姓名，并用一个字典保存各科目的考试成绩：

```
struct Student {  
    let name: String  
    let scores: [科目: Int]  
}
```

```
enum 科目: String, CaseIterable {
    case 语文, 数学, 英语, 物理
}
```

假设我们有一些学生的数据：

```
let s1 = Student(
    name: "Jane",
    scores: [.语文: 86, .数学: 92, .英语: 73, .物理: 88]
)
let s2 = Student(
    name: "Tom",
    scores: [.语文: 99, .数学: 52, .英语: 97, .物理: 36]
)
let s3 = Student(
    name: "Emma",
    scores: [.语文: 91, .数学: 92, .英语: 100, .物理: 99]
)

let students = [s1, s2, s3]
```

我们现在想要检查 `students` 里的学生的平均分，并输出第一名的姓名。使用指令式的方式，依靠运算，循环和条件语句，可以给出下面这种解决方案：

```
var best: (Student, Double)?
for s in students {
    var totalScore = 0
    for key in 科目.allCases {
        totalScore += s.scores[key] ?? 0
    }
    if totalScore > best?.totalScore {
        best = (s, totalScore)
    }
}
```

```
}

let averageScore = Double(totalScore) / Double(科目.allCases.count)

if let temp = best {

    if averageScore > temp.1 {

        best = (s, averageScore)

    }

} else {

    best = (s, averageScore)

}

}

if let best = best {

    print("最高平均分: \(best.1), 姓名: \(best.0.name)")

} else {

    print("students 为空")

}
```

如果第一次读这段代码的话，想要了解它到底做了什么或者到底最后会得到怎样的结果，可能必须要仔细阅读并理解每一行指令。代码行数与 bug 多少往往是正相关，错误很可能隐藏在某个循环或者判断里，这种开发方式为代码的正确性和后续维护带来了巨大的挑战。

我们有什么办法可以减轻开发者的负担，让计算机更加“智能”地为我们解决问题呢？

声明式编程

声明式的编程范式正好站在指令式的对面：如果说指令式是教会计算机“怎么做”，那么声明式就是告诉计算机要“做什么”。指令式编程是描述过程，期望程序执行以得到我们想要的结果；而声明式编程则是描述结果，让计算机为我们考虑和组织出具体过程，最后得到被描述的结果。

现代语言中，一般使用函数式编程或者 DSL 的方式来实现声明式的编程方式。

我们不会在本书中再详述关于什么是函数式编程的概念，如果你对函数式编程的相关话题感兴趣，笔者推荐另一本《函数式 Swift》的图书，你可以在 [ObjC 中国的网站](#) 上找到相关资料。对于上面的例子，使用函数式编程的方式，可以将它改写为：

```
func average(_ scores: [科目: Int]) → Double {  
    return Double(scores.values.reduce(0, +)) /  
        Double(科目.allCases.count)  
}  
  
let bestStudent = students  
.map { ($0, average($0.scores)) }  
.sorted { $0.1 > $1.1 }  
.first
```

在这段代码中，我们首先将 `students` 映射为了 (`Student`, 平均分) 的数组，然后对平均分按降序进行排序，最后取出排序后的首个元素。在这个过程中，我们仅仅是用语句描述了我们想要的结果，例如：按规则进行映射、对元素进行排序等。我们并不关心代码在底层具体是如何操作数组的，而只关心这段代码能够得到我们所描述的结果。

另一种经常用来实现声明式编程的方法是领域特定语言 (DSL)，其中一个典型的代表是 [SQL](#)。SQL 被用在关系数据库中，专门用在结构化查询这一特定领域，它通过描述期望的结果来对数据库进行查询。上面的例子在 SQL 中的对应语句如下：

```
select name, avg(score) as avg_score  
from scores group by name order by avg_score;
```

不论是使用函数式的方式，还是使用 DSL 的方式，我们都能够比较轻松地阅读代码，更快速地理解代码的意图。指令式编程更偏向于是“写给计算机的语言”，而相对地，

声明式编程则更偏向于“写给人看的语言”。将具体的步骤和工作交给底层，同时也最大限度避免了由于开发者的错误而造成的 bug。

声明式的 UI

使用声明式的编程方式来进行用户界面开发，在近年来是颇为热门和受到欢迎的实践方式。当前流行的声明式 UI 的思想，可以追溯到 Elm 语言的设计。在之后，React 的 Component 和 Flutter 的 Widget 都或多或少继承了这种思想，这类声明式 UI 都有如下特点：

1. 代表 UI 层的 View，一般来说并不是真实负责渲染的传统意义的视图层级，而是一个“虚拟的”对 View 组织关系的描述（声明）。
2. 决定 UI 的用户状态 State 被存储在某个或某几个对象中。
3. 用一个函数描述 View，这个函数的输入参数是 State，即 $\text{View} = f(\text{State})$ 。
4. 框架在 State 改变时，调用上述函数获取对应新的 State 的 View，并与当前的 View 进行差分计算，并重新渲染更改的部分。

一般来说， $\text{View} = f(\text{State})$ 中的函数 f 是纯函数，也就是对于某个特定的输入 State，所对应的 View 是确定的，不随其他变量而改变。我们可以单纯地通过控制和改变 State 来得到确定的 UI，这是使用声明式的方法来构建 UI 的基础。

如果你对这种方式的 UI 编写还心存疑虑或颇有不解，那很正常。本书的主要任务就是通过一些实例打消你的这种疑虑，让你能够熟悉和掌握声明式的 UI 编程方式。在接下来的几章中，我们将会多次看到上述的四个特点，并依次逐渐展开详述每个部分的内容，让你获得一些基本的声明式 UI 编程经验。

SwiftUI 和 Combine 简介

现在，可以来谈一谈本书的两个主角了。

SwiftUI 和 Combine 都是在 WWDC 2019 上 Apple 公布的开发框架，它们都是由纯 Swift 编写的。前者是一个声明式 UI 的用户界面开发框架，后者是基于响应式编程，用于处理数据流的框架。SwiftUI 依赖 Combine 来进行背后的数据处理部分的工作。得益于 Swift 5 所带来的 ABI 稳定，现在 Apple 可以放手在 iOS 和 macOS 等系统中内置 Swift 运行环境，这也使得 Apple 自己使用 Swift 编写系统级别的开发框架成为可能。SwiftUI 和 Combine 正是在这个背景下所诞生的首批 Swift 系统级框架。

在 Swift 诞生之前，Apple 平台开发所使用的 Objective-C 语言已经有快三十年的历史了，大家也已经早已习惯了 AppKit 和 UIKit 所提供的一套编程范式。不论是在 macOS 上还是在 iOS 上，使用 Model-View-Controller (MVC) 的架构，配合 Target-Action 或者 protocol-delegate 模式来交换信息，使用 Key-Value Observing (KVO) 或者 Key-Value Coding (KVC) 来灵活地监测变化和读写属性，已经是大部分 Apple 平台开发者驾轻就熟的事情了。这一套“成熟”的工具在过去很好地服务了开发者，并帮助我们创造了无数美好的 app。但是，随着移动端的不断发展，现代开发的节奏越来越快，项目的复杂度也越来越高。像是 React Native 和 Flutter 这样的移动端跨平台方案大行其道，它们在开发时可以进行热重载，不需要重新编译和运行就可以看到 UI 部分修改的结果，这大大加速了开发进度。而两者声明式 UI 的编写方式和严格的数据流动方向，也大幅减轻了开发者的思考负担。由于这些先进特性所带来的良好体验，以及一次编写，多平台运行的独特魅力，导致了越来越多的 native 开发者“流失”到这些平台，这对 Apple 自身的生态也是一种威胁。

因此，SwiftUI 和 Combine 作为首批系统级别的 Swift 框架，承担了双重任务。

首先，它们需要作为 Swift 语言的推广，让大家看到这门语言与 Apple 生态的结合究竟能够产生怎样的优势。Swift 语言虽然是开源项目，但毫无疑问 Apple 依然主导着这门语言的发展方向。Swift 5.1 中许多特性，都是为了能更简洁地使用 SwiftUI 和 Combine 而新增的。相比于其他竞品，由于编程语言上的完全控制，SwiftUI 和 Combine 在使用上来说都比其他方案更加简明，可读性也更高。这对于吸引新人加入到 Swift 开发生态圈至关重要。

其次，这两个框架还担负着 Apple 平台编程范式转变的重要任务。越来越多的案例证明，声明式和响应式可以有效加速进度，并减少开发者出错的可能性。而 Objective-C 积重难返，想要承担起这样的转变，可能需要花费更多的时间和精力。在如今 React Native、Flutter 以及微信小程序的“围杀”下，如何为开发者提供更好的创作环境，持续地吸引开发者留在自己的平台，是 Apple 不得不认真考虑的问题。

相对于高度成熟的 UIKit 和 AppKit 来说，SwiftUI 还十分年轻，它周围的生态也远远没有达到成熟。不过，SwiftUI 在 Apple 平台上是与 UIKit 和 AppKit 兼容的：你可以在已有的 app 中加入某些使用 SwiftUI 制作的界面，也可以在 SwiftUI 中使用任何已有的 UIKit 和 AppKit 控件。这让开发者们不需要冒险一次性地迁移到新的平台，而可以对新事物“渐进式”地一点点尝试，也可以在任何时候“返回”到熟悉和完备的解决方案。拥有这个保证后，开发者就完全可以放心地将 SwiftUI 引入到项目中，而不用担心最终会有什么需求无法实现。

练习

1. 区分指令式和声明式

下列语句分别是哪种思想的体现？是指令式还是声明式？

- a. 去超市买一只鸡，然后用微波炉 600W 加热三分钟以后装盘。
- b. 将文件夹中尺寸大于 1MB 的文件按修改日期排序并打印它们的文件名。
- c. 画一组圆心在 (0, 0)，半径以 5 逐渐增加的圆。
- d. 将画笔移动到 (0, 0)，从 (0, 0) 到 (1, 1) 画一条线。

2. 关于 SwiftUI

下面关于 SwiftUI 的说法，不正确的一项是什么？

- a. SwiftUI 是 Apple 推出的声明式 UI 开发框架，它由 Swift 开发。
- b. SwiftUI 可以和现有的 UIKit 和 AppKit 兼容，因此现有 app 可以逐步迁移到 SwiftUI。
- c. SwiftUI 使用了大量 Swift 5.1 的特性来简化语法和降低学习难度。
- d. 在 SwiftUI 中，我们使用 MVC 和 Target-Action 等开发范式来组织数据流动。

3. 实践指令式和声明式处理问题

对本章代码中的 `students` 数组，分别使用指令式和声明式（通过函数式编程）的方式完成下列任务：

1. 计算所有学生的语文成绩平均分，并将其打印出来。
2. 统计各个科目的及格率（60 分以上及格），并将及格率结果进行排序。

你好，SwiftUI

2

脱离了实践的学习将会是无源之水，无本之木。在本章中，我们将会通过一个具体的案例，从 Hello World 开始，引领你逐步构建一个完整的 demo app 的 UI 部分。

计算器 app 实例

在本章和下一章中，我们的第一个大目标是完成一个和 iOS 上的计算器看上去类似的示例 app，设计效果图如下：



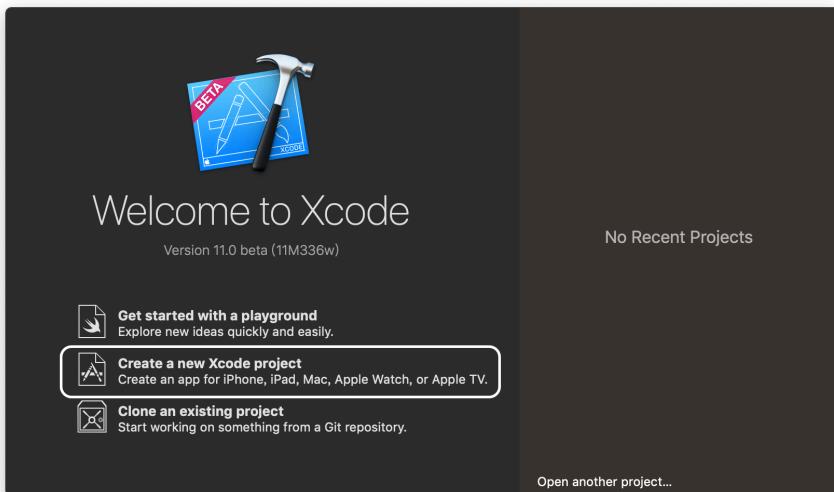
这一章里，我们会专注于使用 SwiftUI 的 DSL 来描述这个用户界面的构成要件，并完成 UI 样式和布局等方面的内容。SwiftUI DSL 的基础是 View，它基于 View 来对

UI 进行描述，比如 UI 的内容、应有的样式以及控件触发时的响应等。我们在本章中会从零开始，逐渐看到 SwiftUI 的基本使用方式和一些核心概念。

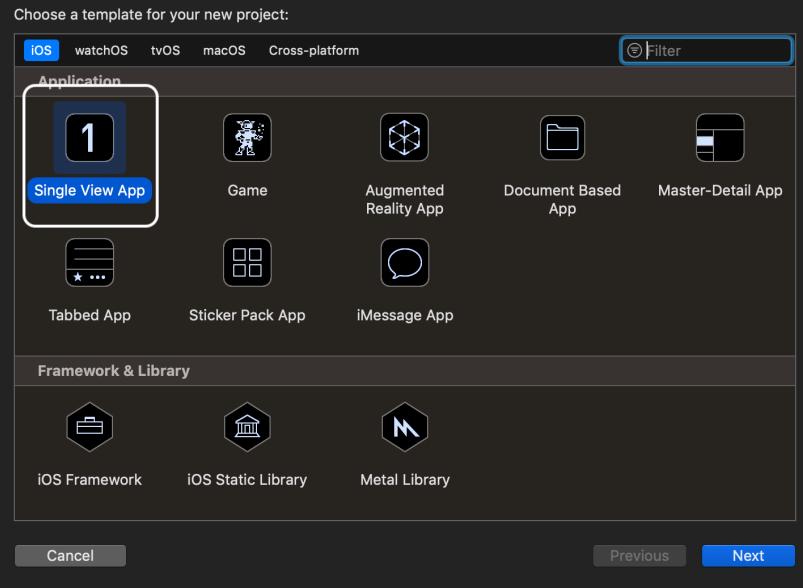
在下一章关于数据状态和绑定的话题中，我们会进一步在 UI 的基础上为计算器 app 添加 model 和绑定事件，让它成为一个可以使用的实际的 app。

创建项目和 Hello World

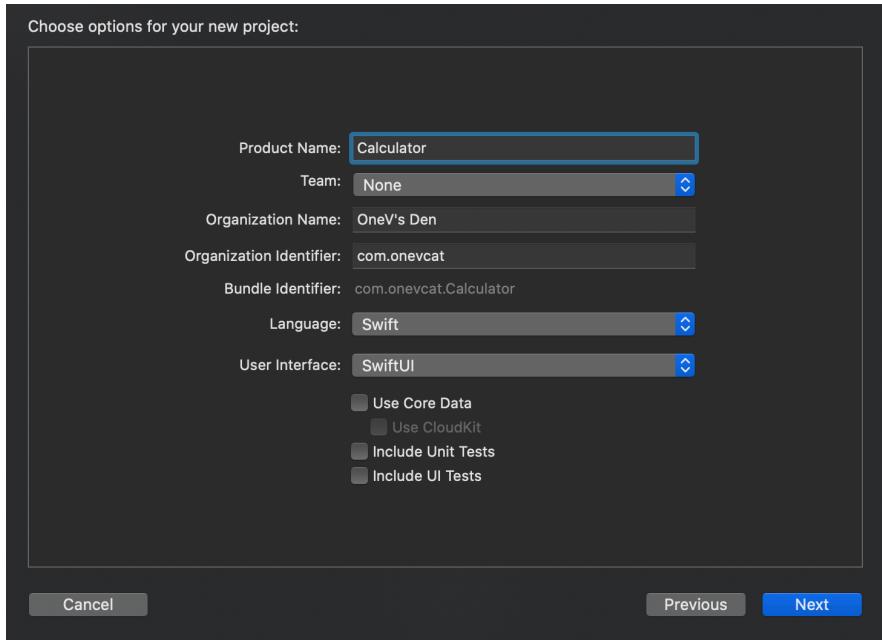
打开 Xcode，选择“Create a new Xcode project”：



选择 iOS tab 下的 Single View App 模板，Xcode 将为我们创建一个单页的 iOS app。



接下来，将项目命名为 Calculator，并且选择“Use SwiftUI”作为用户界面的构建方式，让 Xcode 使用 SwiftUI 来创建第一个画面。



点击 Next 并选择合适的位置后，我们可以得到一个使用 SwiftUI 作为界面开发方式的新项目。在项目导航中找到 ContentView.swift，它的内容如下：

```
import SwiftUI

struct ContentView : View {
    var body: some View {
        Text("Hello World")
    }
}

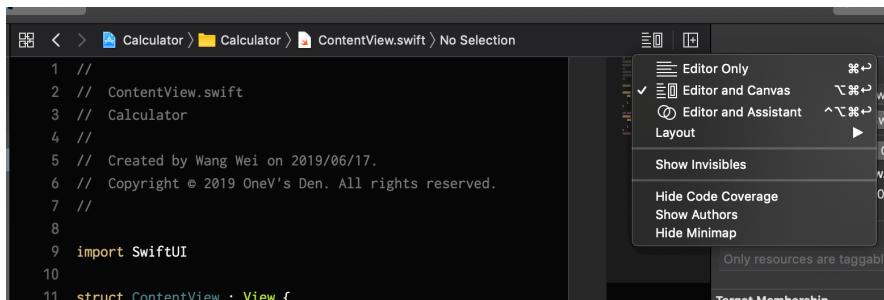
struct ContentView_Previews : PreviewProvider {
    static var previews: some View {

```

```
ContentView()  
}  
}
```

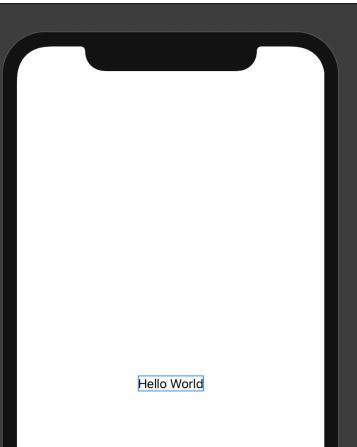
上面代码中 `ContentView` 所定义的是实际的 UI，`ContentView_Previews` 则是一个满足了 `PreviewProvider` 的 dummy 界面。如果你的操作系统是 macOS 10.15 或以上的话，你可以使用编辑器右上角的切换编辑器视图按钮，选中“Editor and Canvas”(快捷键 Option + Command + 回车)，并单击右上角的 Resume 按钮。`Xcode` 预览界面将会读取 `ContentView_Previews` 的内容，并渲染在编辑器右侧的实时预览栏中。它就是 `ContentView_Previews` 的 `previews` 属性所返回的 `View`，也就是 `ContentView` 的内容。默认情况下，`Xcode` 会使用你当前选取的 iOS 模拟器作为预览尺寸，为了能让你得到的内容和书中一致，建议你在这个例子中选择 iPhone XR 模拟器作为目标设备。

如果你还在使用 macOS 10.15 之前的版本，`Xcode` 预览的功能将不能工作。你需要将项目编译到模拟器或者真机中才能确认界面情况。



如果一切顺利，那么在 `Xcode` 编辑器的右侧，应该可以看到实时的预览。`ContentView` 现在只是一个在屏幕正中的文本：

```
1 // ContentView.swift
2 // Calculator
3 //
4 // Created by Wang Wei on 2019/06/17.
5 // Copyright © 2019 OneV's Den. All rights reserved.
6 //
7 //
8
9 import SwiftUI
10 import Combine
11
12 struct ContentView : View {
13     var body: some View {
14         Text("Hello World")
15     }
16 }
17
18 #if DEBUG
19 struct ContentView_Previews : PreviewProvider {
20     static var previews: some View {
21         ContentView()
22     }
23 }
24 #endif
25
```



此时，如果你尝试编辑 `Text("Hello World")` 中的字符串的内容，比如换为 `Text("Hello SwiftUI")`，应该可以在预览栏中实时看到文本的变化。

Xcode 预览使用了动态替换 `body` 属性的特性，但是它有一些局限：当 `body` 以外的部分被改变，导致 `ContentView` 需要整个重新编译时（比如在 `ContentView` 中随意加入一个存储属性 `var value = 1`），你必须再次点击 `Resume` 按钮才能重新开始预览。默认情况下刷新预览的快捷键是 `Option + Command + P`。

使用 Modifier 描述 Text 及 Button

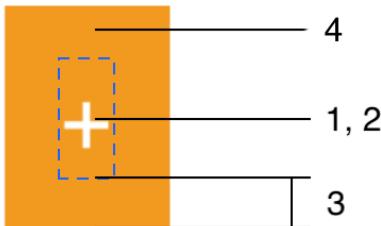
基本 Modifier

除了改变 `Text` 的文本内容外，我们会通过在 `Text` 声明之后使用方法调用的方式，来定义文本样式。让我们从单个的计算器按钮入手，比如加号键。将 `ContentView` 的 `body` 部分替换为以下内容：

```
var body: some View {
```

```
Text("+")
    .font(.title)           // 1
    .foregroundColor(.white) // 2
    .padding()              // 3
    .background(Color.orange) // 4
}
```

font, foregroundColor, padding 和 background 各自定义了 Text 的一项属性：



1. 将文本字体设置为 `Font.title`。除了 `.title` 以外，SwiftUI 还预定义了一系列字体，比如 `.headline`, `.body` 等。你可以在 Apple 提供的[人机交互指南 \(HIG\)](#) 的[排版](#)页面中找到每种预定义字体所对应的具体大小。如果你想要具体按数值指定字号的话，可以使用 `.font(.system(size: 48))` 的方式；除了系统提供的字体，你也可以通过指定名字，来使用自定义字体：
`.font(.custom("Copperplate", size: 48))`。

相对于使用 `.system` 或者 `.custom` 字体，使用预定义的字体可以让我们不需要额外的工作就能适配 Dynamic Type 特性。除非有特别的字体尺寸偏好，否则对于需要跟随用户字体大小设置进行缩放的内容，最好都应该按照 HIG 的定义选择预定义的字号。

2. 使用 `foregroundColor` 设定文本的颜色。除了像是 `Color.white` 或者 `Color.red` 这样的系统颜色，你也可以使用 `Color(_:red:green:blue:)` 以提供色彩空间和 RGB 值来设置颜色。为了更方便颜色的管理，`Color` 还提供了一个以颜色名字字符串从 `Assets.xcassets` 中加载颜色的方法，我们马上会看到相关例子的应用。
3. `padding` 将把当前的 `View` 包裹在一个新的 `View` 里，并在四周填充空白部分。在上面例子中，我们没有给定参数，这会在文本的四周都填上系统默认尺寸的空白。我们可以为 `padding` 指定需要填充的方向以及大小，比如：
`.padding(.top, 16)` 将在上方填充 16 point 的空白，`.padding(.horizontal, 8)` 在水平方向（也即 `[.leading, .trailing]`）填充 8 point。
4. 最后，使用 `background` 为文本指定背景。这个方法接受的参数只需要满足 `View`，也就是说，除了像例子中设定一个 `Color` 以外（`Color` 也是一个遵守 `View` 协议的类型），你也可以将任意的 `View` 作为背景元素进行设置。

在 SwiftUI 中，上面这四次调用，都被称为 `View` 的 `modifier`。一个 `view modifier` 作用在某个 `View` 上，并生成原来值的另一个版本。按照这个定义，大致来说，`view modifier` 分为两种类别：

- 像是 `font`, `foregroundColor` 这样定义在具体类型（比如例中的 `Text`）上，然后返回同样类型（`Text`）的原地 `modifier`。
- 像是 `padding`, `background` 这样定义在 `View extension` 中，将原来的 `View` 进行包装并返回新的 `View` 的封装类 `modifier`。

原地 modifier 一般来说对顺序不敏感，对布局也不关心，它们更像是针对对象 View 本身的属性的修改。而与之相反，封装类的 modifier 的顺序十分重要。我们可以尝试把上例改为：

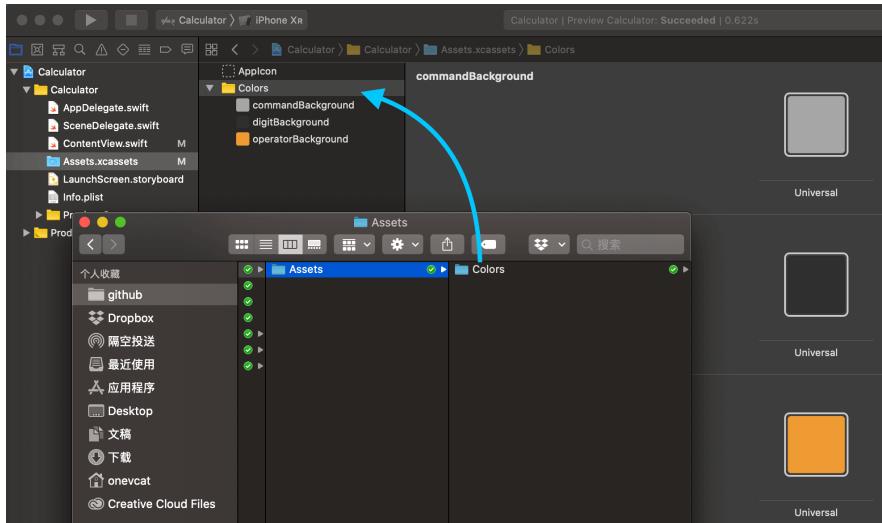
```
Text("+")
    .font(.title)
    .foregroundColor(.white)
    .background(Color.orange) // 1
    .padding()
    .background(Color.blue) // 2
```

和原例相比，我们在颠倒了 .padding() 和 .background(Color.orange) 的顺序，为了清晰说明 modifier 的顺序对结果的影响，我们还在最后加上了另一个蓝色的背景。预览结果，由 “// 1” 导致的在 “+” 文本周围的内层橙色背景并不会包括 padding 的影响，而在更外层蓝色背景则是作用在 padding 之后的结果上的。SwiftUI 的 modifier 所造成的布局影响是严格按照顺序执行的。

我们会在本书最后一章中详细说明造成这种现象的原因。现在，先让我们专注于其他更重要的部分。

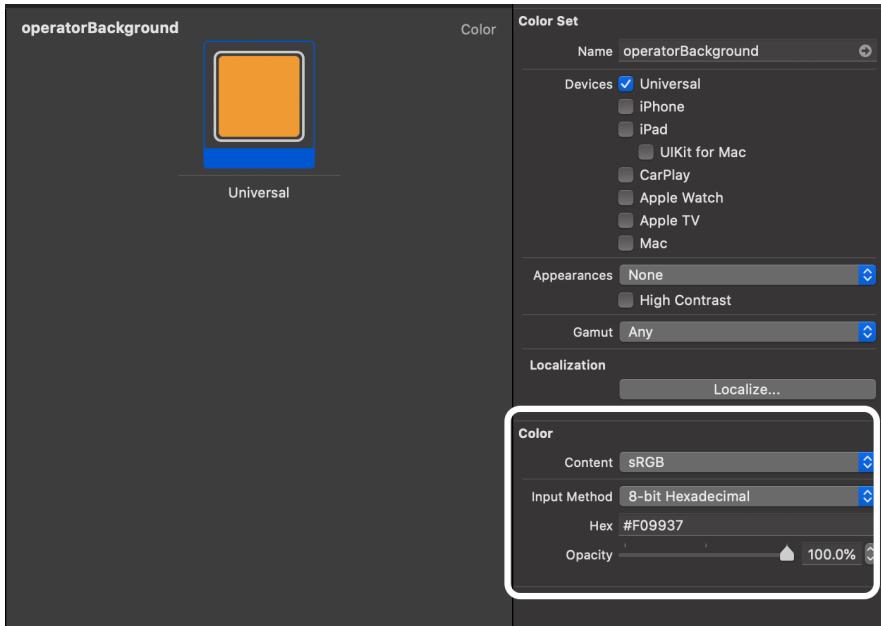
添加和使用颜色

在继续之前，我们再做一点准备工作，为项目设定必须的颜色。打开本书源码的 2.Hello-SwiftUI 文件夹，在 Assets 下，将 “Colors” 整个拖到 Calculator 项目的 Assets.xcassets 中去：



Colors 定义了三个颜色，它们是计算器不同类型按钮的颜色。通过在 Assets 文件中定义颜色，我们可以使用 `Color("colorName")` 的方式从定义中创建颜色。这么做主要有两个好处：

- 更简单的自适配颜色：在同一个颜色名字的定义中，我们可以指定多个具体颜色，并一一设定它们的使用环境：比如在 iPhone 和 iPad 上使用不同颜色，在亮色和暗色设置上使用不同颜色等等。在实际使用代码初始化时，同样的名字在不同的条件下会按照预先的设置生成颜色。
- 可以让我们直接使用十六进制的颜色值表示。很多时候，设计师都会偏向标注十六进制颜色值，比如 `#F09937`，而不是通过 RGB 给出。使用 `Assets.xcassets` 管理颜色时，可以直接通过 Hex 值设定：



现在，让我们更进一步，将刚才定义的 Text 变为：

```
Text("+")
    .font(.system(size: 38)) // 1
    .foregroundColor(.white)
    .frame(width: 88, height: 88) // 2
    .background(Color("operatorBackground")) // 3
    .cornerRadius(44) // 4
```

1. 因为计算器按钮的字体需要足够大，且不存在适配 Dynamic Type 的需要，因此我们使用固定大小的字号。
2. padding 虽然可以在字符和边界之间加入空隙，但是计算器的每个按钮都是同等大小。我们使用 frame 这个 modifier 来定义某个 View 的尺寸。这个大小是按照 iPhone X 的屏幕设定的，可能会不太合适小尺寸的屏幕 (比如 iPhone)

SE)。我们先忽略掉这个问题，在本章最后我们会再回头处理小屏幕适配的问题。

3. “operatorBackground” 是我们刚刚添加到 Assets 中的颜色。
4. cornerRadius 是另一个非常常用的 modifier，它通过包装的方式为 View 添加圆角，并返回新的 View。

现在，我们准备好了加号按钮的文本值和样式，接下来让我们为它添加按钮行为吧。

按钮及自定义 View

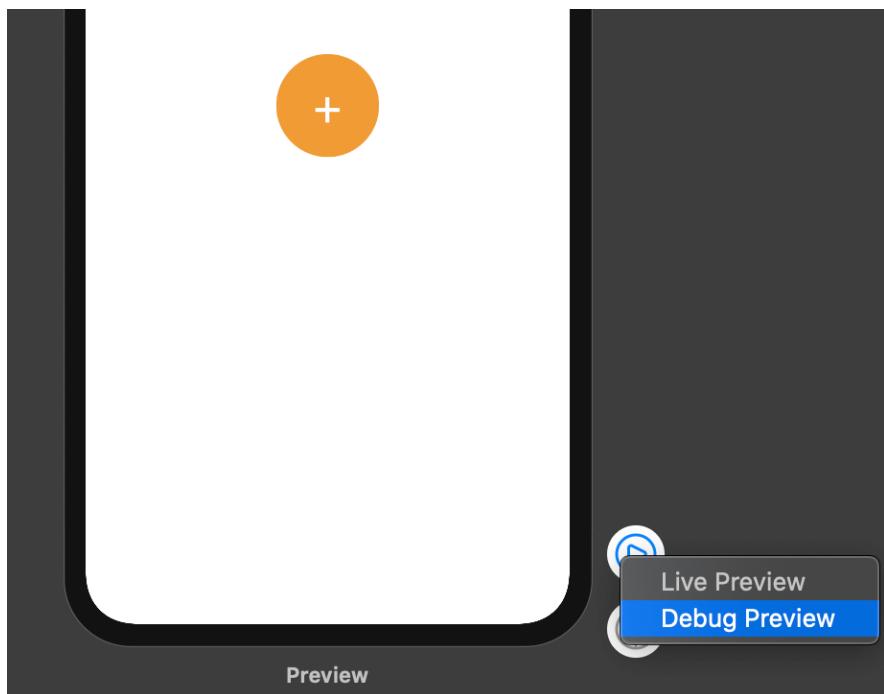
SwiftUI 中初始化按钮非常简单，调用 Button.init(action:label:) 就行了。第一个参数 action 定义了按钮触发时的行为，第二个参数 label 接受一个闭包，它是一个 ViewBuilder，在底层，ViewBuilder 使用了所谓的 function builder 技术，来把 DSL 的 View 描述转换为实际的 View 对象。在本章中，我们先跳过这些细节和底层，来直接看看按钮的使用。

将上一节里 Text 部分的代码用 Button 包装起来：

```
Button(action: {                                     // 1
    print("Button: +")
}) {                                              // 2
    Text("+")
        .font(.system(size: 38))
        .foregroundColor(.white)
        .frame(width: 88, height: 88)
        .background(Color("operatorBackground"))
        .cornerRadius(44)
}
```

1. 通过提供 action 闭包，我们可以定义 Button 按下时所要触发的行为。例子中我们先简单地向控制台打印一段字符串。
2. 定义 Button 的视觉内容，我们直接使用了上节中的 Text。

在 Xcode 预览看起来，这段代码和上一节的 Text 没有任何区别。除了静态预览以外，Xcode 也提供了运行的实时预览。我们可以点击预览栏中设备框右下角外侧的开始按钮，就可以与预览界面进行实际交互了。实际尝试可以发现，这个 Button 现在已经可以响应点击，并正确地在按下时高亮。但是这样的实时预览在点击时并不会按预想将“Button: +”打印到控制台。为了在实时的交互预览中进行控制台输出，我们可以右键点击开始按钮，并选择“Debug Preview”选项：



当然，选择合适的 iOS 模拟器或者直接在设备上运行，也可以进行按钮交互及看到控制台输出。但是 Xcode Previews 能够实时反映出对 UI 的修改。在本书之后的部分，如无特殊说明，控制台输出和实时运行，检查结果等操作，都默认是在预览模式下进行。

基本布局和 Stack 容器

Stack 堆叠

我们已经做好了一个计算器按钮，接下来的任务是将这个按钮当作样板，弄出整行的按钮。以“1”，“2”，“3”，“+”这一行按钮为例。想要将若干个 View 并列在同一行，可以使用 HStack，它是水平堆栈 (Horizontal Stack) 的缩写。把上面的 Button 复制三次，修改 Text 的文本、背景颜色和按钮触发时的输出，然后用 HStack 包装起来：

```
var body: some View {
    HStack {
        Button(action: {
            print("Button: 1")
        }) {
            Text("1")
                .font(.system(size: 38))
                .foregroundColor(.white)
                .frame(width: 88, height: 88)
                .background(Color("digitBackground"))
                .cornerRadius(44)
        }
        Button(action: {
            print("Button: 2")
        }) {
            Text("2")
                .font(.system(size: 38))
                .foregroundColor(.white)
                .frame(width: 88, height: 88)
                .background(Color("digitBackground"))
                .cornerRadius(44)
        }
        Button(action: {
            print("Button: 3")
        }) {
            Text("3")
                .font(.system(size: 38))
                .foregroundColor(.white)
                .frame(width: 88, height: 88)
                .background(Color("digitBackground"))
                .cornerRadius(44)
        }
    }
}
```

```
}) {  
    Text("2")  
        .font(.system(size: 38))  
        .foregroundColor(.white)  
        .frame(width: 88, height: 88)  
        .background(Color("digitBackground"))  
        .cornerRadius(44)  
}  
  
Button(action: {  
    print("Button: 3")  
}) {  
    Text("3")  
        .font(.system(size: 38))  
        .foregroundColor(.white)  
        .frame(width: 88, height: 88)  
        .background(Color("digitBackground"))  
        .cornerRadius(44)  
}  
  
Button(action: {  
    print("Button: +")  
}) {  
    Text("+")  
        .font(.system(size: 38))  
        .foregroundColor(.white)  
        .frame(width: 88, height: 88)  
        .background(Color("operatorBackground"))  
        .cornerRadius(44)  
}  
}
```

```
}
```

当然啦，大概只有按照代码行数来计算工作量的公司里我们会采用这样的写法，正常思维的人类肯定会选择进行一定的抽象，来避免重复代码。

我们将这个 Button 提取为一个新的 View，把它重命名为 CalculatorButton。只要把需要的参数提取出来，CalculatorButton 可以变成通用的类型：

```
struct CalculatorButton : View {

    let fontSize: CGFloat = 38
    let title: String
    let size: CGSize
    let backgroundColorName: String
    let action: () → Void

    var body: some View {
        Button(action: action) {
            Text(title)
                .font(.system(size: fontSize))
                .foregroundColor(.white)
                .frame(width: size.width, height: size.height)
                .background(Color(backgroundColorName))
                .cornerRadius(size.width / 2)
        }
    }
}
```

这样一来，ContentView 的 body 里可以简化为：

```
var body: some View {
    HStack {
        CalculatorButton(
            title: "1",
            size: CGSize(width: 88, height: 88),
            backgroundColorName: "digitBackground")
        {
            print("Button: 1")
        }
        CalculatorButton(
            title: "2",
            size: CGSize(width: 88, height: 88),
            backgroundColorName: "digitBackground")
        {
            print("Button: 2")
        }
        CalculatorButton(
            title: "3",
            size: CGSize(width: 88, height: 88),
            backgroundColorName: "digitBackground")
        {
            print("Button: 3")
        }
        CalculatorButton(
            title: "+",
            size: CGSize(width: 88, height: 88),
            backgroundColorName: "operatorBackground")
        {
            print("Button: +")
        }
    }
}
```

```
    }
}
}
```

计算器按钮 Model

虽然有一定简化，不过想要用上面的 `CalculatorButton` 逐个定义计算器的所有按钮，还是会很麻烦。我们可以使用循环的方式让事情变简单一些。在这一节里，我们来定义一个代表计算器按钮的 `model` 类型，并用这个 `model` 和 SwiftUI 中的“循环”语句来简化 UI 构建的过程。

在项目中新建一个 Swift 文件，将它命名为 `CalculatorButtonItem.swift`，添加如下代码：

```
enum CalculatorButtonItem {
```

```
    enum Op: String {
        case plus = "+"
        case minus = "-"
        case divide = "÷"
        case multiply = "×"
        case equal = "="
    }
}
```

```
    enum Command: String {
        case clear = "AC"
        case flip = "+/-"
        case percent = "%"
    }
}
```

```
case digit(Int)
case dot
case op(Op)
case command(Command)
}
```

我们可以粗略地把计算器上的按钮分为四大类：代表从 0 至 9 的数字 digit，小数点 dot，加减乘除等号这样的操作 (Op)，以及清空、符号翻转等这类命令 (Command)。接下来，可以在 extension 里追加定义必要的外观：

```
extension CalculatorButtonItem {
    var title: String {
        switch self {
            case .digit(let value): return String(value)
            case .dot: return "."
            case .op(let op): return op.rawValue
            case .command(let command): return command.rawValue
        }
    }

    var size: CGSize {
        CGSize(width: 88, height: 88)
    }

    var backgroundColorName: String {
        switch self {
            case .digit, .dot: return "digitBackground"
            case .op: return "operatorBackground"
            case .command: return "commandBackground"
        }
    }
}
```

```
    }
}
}
```

严格来说，可能将上面的 extension 提取出来，遵照 MVVM 的架构模式将它定义成一个新的 View Model 类型会更好。但是对于 CalculatorButton 来说，并不存在数据变化和 UI 同步的问题，所以在这里为了简洁，我们直接把外观也定义在了同一个类型里。

现在，我们可以进一步简化 ContentView 了。在 ContentView 中添加存储属性，来定义这一行按钮：

```
struct ContentView: View {
    let row: [CalculatorButtonItem] = [
        .digit(1), .digit(2), .digit(3), .op(.plus),
    ]
    // ...
}
```

接下来，重写 body 部分，将内容替换为：

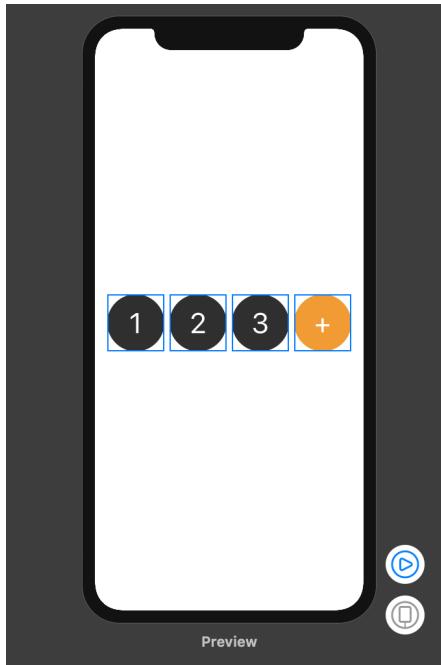
```
var body: some View {
    HStack {
        ForEach(row, id: \.self) { item in
            CalculatorButton(
                title: item.title,
                size: item.size,
                backgroundColorName: item.backgroundColorName)
        }
    }
}
```

```
        {
            print("Button: \(item.title)")
        }
    }
}
```

上面的代码可能无法编译，我们需要对此进行一些解释。`ForEach` 是 SwiftUI 中一个用来列举元素，并生成对应 View collection 的类型。它接受一个数组，且数组中的元素需要满足 `Identifiable` 协议。如果数组元素不满足 `Identifiable`，我们可以使用 `ForEach(_:_id:)` 来通过某个支持 `Hashable` 的 key path 获取一个等效的元素是 `Identifiable` 的数组。在我们的例子中，数组 `row` 中的元素类型 `CalculatorButtonItem` 是不遵守 `Identifiable` 的。为了解决这个问题，我们可以为 `CalculatorButtonItem` 加上 `Hashable`，这样就可以直接用 `ForEach(row, id: \.self)` 的方式转换为可以接受的类型了。在 `CalculatorButtonItem.swift` 文件最后，加上一行：

```
extension CalculatorButtonItem: Hashable {}
```

在预览中，我们应该可以看到这一行按钮被正确渲染了。



布局计算和调整

接下来我们要把计算器所有的按钮，以及显示计算结果的文本都构建出来。

首先，还是让我们进行一些重构工作。用和上面提取 `CalculatorButtonItem` 同样的方法，新建一个 `CalculatorButtonRow`，并将 `ContentView` 中的内容都移动到新的类型中：

```
struct CalculatorButtonRow : View {
    let row: [CalculatorButtonItem]
    var body: some View {
        HStack {
            ForEach(row, id: \.self) { item in
                CalculatorButton(
                    title: item.title,
                    value: item.value,
                    color: item.color
                )
            }
        }
    }
}
```

```
        title: item.title,
        size: item.size,
        backgroundColorName: item.backgroundColorName)
    {
        print("Button: \(item.title)")
    }
}
}
}
```

CalculatorButtonRow 可以接受任意的 [CalculatorButtonItem]，并将其中的元素显示为一行按钮。这样一来，ContentView 的 body 就十分简单了：

```
var body: some View {
    CalculatorButtonRow(row: [
        .digit(1), .digit(2), .digit(3), .op(.plus)
    ])
}
```

为了显示多行按钮，我们需要将它们在竖直方向堆叠起来，这需要用到 VStack。它和我们已经看到的 HStack 类似，只不过方向有所不同。

```
var body: some View {
    VStack(spacing: 8) { // 1
        CalculatorButtonRow(row: [
            .command(.clear), .command(.flip),
            .command(.percent), .op(.divide)
        ])
        CalculatorButtonRow(row: [

```

```

    .digit(7), .digit(8), .digit(9), .op(.multiply)
  ])
CalculatorButtonRow(row: [
  .digit(4), .digit(5), .digit(6), .op(.minus)
])
CalculatorButtonRow(row: [
  .digit(1), .digit(2), .digit(3), .op(.plus)
])
CalculatorButtonRow(row: [
  .digit(0), .dot, .op(.equal)
])
// 2
}
}

```

1. 在使用 VStack 时，我们为它明确指定了每个元素的间距为 8。HStack 也可以接受类似的参数，在我们的例子里，对按钮之间的水平间距我们使用了默认值。如果不加指定，SwiftUI 会根据 View 所在的环境为我们挑选默认值。这里的 VStack 间距默认值为 0，不满足我们的要求，因此我们进行了具体指定。
2. 计算器的最后一行只有三个按钮，因此尺寸不太正确。我们会在之后修正这个问题。

重复五次 CalculatorButtonRow 也挺蠢的，我们可以同样用 ForEach 来简化。提取一个新的类型 CalculatorButtonPad，把所有按钮的信息写到这个类型中去，然后同样地用 ForEach 绘制每一行按钮：

```

struct CalculatorButtonPad: View {
  let pad: [[CalculatorButtonItem]] = [
    [.command(.clear), .command(.flip),
     .command(.percent), .op(.divide)],

```

```
[.digit(7), .digit(8), .digit(9), .op(.multiply)],
[.digit(4), .digit(5), .digit(6), .op(.minus)],
[.digit(1), .digit(2), .digit(3), .op(.plus)],
[.digit(0), .dot, .op(.equal)]  
]  
  
var body: some View {  
    VStack(spacing: 8) {  
        ForEach(pad, id: \.self) { row in  
            CalculatorButtonRow(row: row)  
        }  
    }  
}
```

现在，我们把 ContentView 的 body 中的内容替换为简单的 CalculatorButtonPad()，Xcode 预览将为我们显示完整的计算器按键盘面：

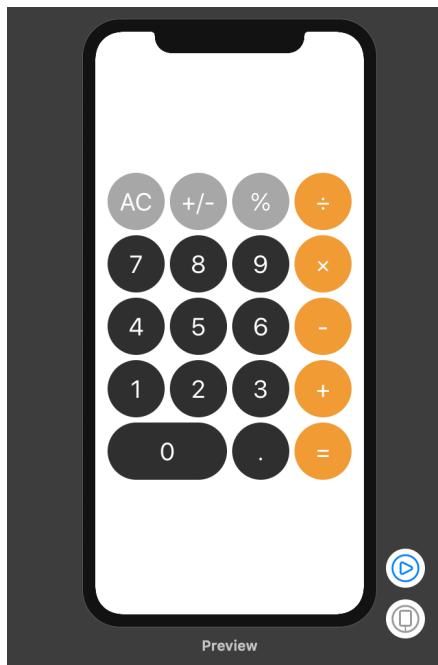
```
// ContentView.swift  
var body: some View {  
    CalculatorButtonPad()  
}
```

对于最后一行的按键 0，我们希望它占有两个按钮外加一个单位间距的宽度。在 CalculatorButtonItem 中，更改 extension 里的 size 属性就可以做到这一点：

```
extension CalculatorButtonItem {  
    // ...  
  
var size: CGSize {
```

```
if case .digit(let value) = self, value == 0 {  
    return CGSize(width: 88 * 2 + 8, height: 88)  
}  
return CGSize(width: 88, height: 88)  
}  
  
// ...  
}
```

它对按钮为 `.digit(0)` 的情况进行了匹配，并返回更宽的按钮尺寸。在这一切完成后，Xcode 预览中现在可以看到居中且大小合适，样式正确的计算器按钮了：

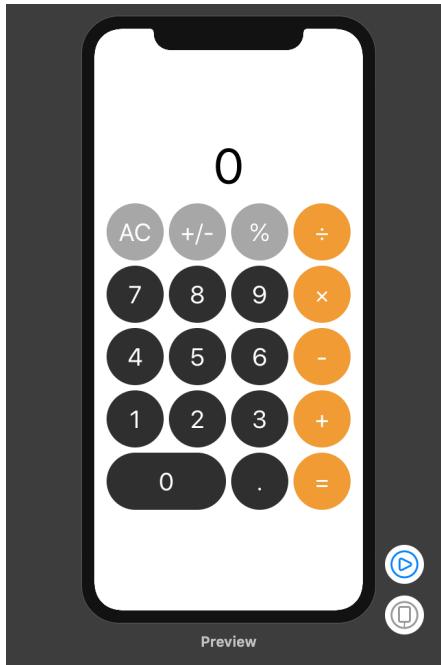


使用 frame 或 Spacer 占满屏幕

最后，让我们来完成这个计算器的 UI 部分剩余工作。我们还需要在 CalculatorButtonPad 上方添加一个用于显示输入数字和计算结果的 Text。再次使用一个 VStack，并将 Text 和 CalculatorButtonPad 都包含进去，并设置合适的间距及字体样式：

```
var body: some View {
    VStack(spacing: 12) {
        Text("0").font(.system(size: 76))
        CalculatorButtonPad()
    }
}
```

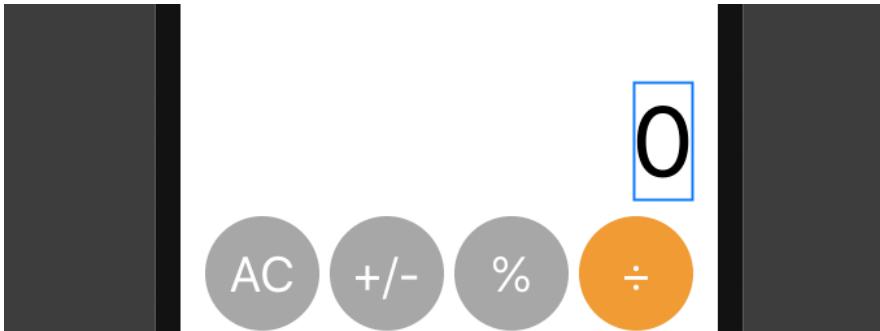
VStack 在默认情况下会将其中的 View 居中对齐，上面的代码将使得 Text 也位于正中，这并不是我们想要的结果：



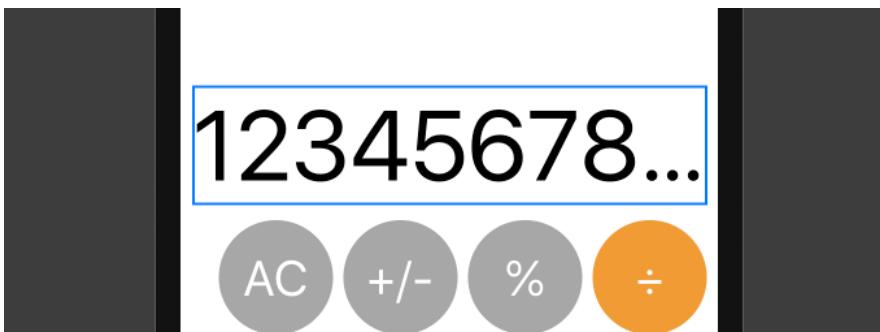
VStack 可以接受一个 alignment 参数，来指定其中 View 的对齐方式：

```
 VStack(alignment: .trailing, spacing: 12) {  
     Text("0").font(.system(size: 76))  
     CalculatorButtonPad()  
 }
```

它可以将 Text 置于最右侧，如图所示：



不过值得注意的是，这有时候可能不是我们想要的结果，因为不仅是 Text，其实 CalculatorButtonPad 也被靠右对齐了。只不过是因为在初始状态下 (Text 文本为“0”时)，CalculatorButtonPad 的宽度较 Text 要宽，外层 VStack 的宽度由内部 CalculatorButtonPad 的宽度决定，因此，尾端对齐从视觉上来说，只对 Text 产生效果，所以看不出端倪。上图中 0 周围的蓝框表明了该 Text 所占的大小，它是一个随着文本内容变动的尺寸。当我们尝试在 Text 里输出比较长的内容，Text 本身的宽度将根据文本内容发生变化。在 Text 横向上超过按钮面板的宽度后，VStack (也就是 Text 的 Container View) 的宽度跟随 Text 变动，这时下方的计算器按钮执行右侧对齐，发生偏移，这显然不是我们期望的结果。



更正确的做法不是将 VStack 的内容全部靠右对齐，而是只希望 Text 的内容靠右。为此，我们需要给 Text 设定一个固定的宽度。在上面设置 CalculatorButton 的尺寸

的时候，我们已经使用过 `frame` 了。对 `Button` 来说，`.frame(width: 88, height: 88)` 这样的固定尺寸就可以了，但是对于这里的 `Text`，我们希望它能占满整行的宽度。

`frame modifier` 还有另一个版本：

```
func frame(  
    minWidth: CGFloat? = nil,  
    idealWidth: CGFloat? = nil,  
    maxWidth: CGFloat? = nil,  
    minHeight: CGFloat? = nil,  
    idealHeight: CGFloat? = nil,  
    maxHeight: CGFloat? = nil,  
    alignment: Alignment = .center  
) → Self.Modified<_FlexFrameLayout>
```

这个函数所有参数都是可选的，它指定了 `View` 可以使用的可变高宽范围。在默认情况下，`View` 的尺寸会根据其内容以及它的子 `View` 的内容自动适配：比如在上面例子中，当 `Text` 为 0 时，`Text` 的宽度只有 0 的宽度；当文本内容变长时，`Text` 的宽度也随之变化。我们可以使用 `frame` 改变这一行为，将上面的代码改为：

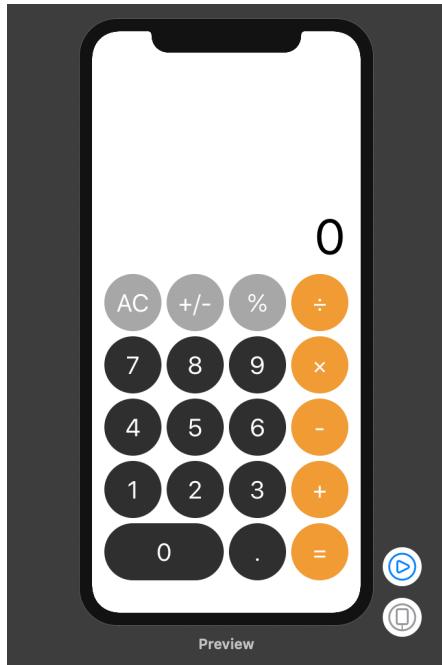
```
 VStack(spacing: 12) {  
     Text("0")  
         .font(.system(size: 76))  
         .frame(  
             minWidth: 0,  
             maxWidth: .infinity,  
             alignment: .trailing)  
     CalculatorButtonPad()  
 }
```

frame 中的 minWidth 和 maxWidth 为 Text 的宽度定义了范围，这会“提示”Text 不必遵守内容尺寸，而是去适应容器的尺寸。将 .infinity 传递给 maxWidth，表示不对最大宽度进行限制，这种情况下 Text 会尽可能占据它的容器的宽度，变为全屏宽。frame 还提供了一个 alignment 参数，让我们可以设定其对齐，在这里我们指定 .trailing 让文本靠右。

接下来一个问题是，现在文本和按键在水平方向上依然是居中的，在屏幕顶部和底部存在大量空白。最终我们想要的效果应该是按键位于屏幕底部。举一反三，我们可以通过类似的方法，给 VStack 添加 frame 并设置 maxHeight 和 alignment 来达到我们的目的。不过，我们有更简单的方法来把屏幕占满。SwiftUI 允许我们定义可伸缩的空白：Spacer，它会尝试将可占据的空间全部填满。在我们的 body 中，可以加入一个 Spacer 来把 VStack 的上半部分全部填满。同时，为了美观，我们也可以为 Text 和 CalculatorButtonPad 添加一些必要的 padding 并且限制 lineLimit 为一行：

```
VStack(spacing: 12) {  
    Spacer()  
    Text("0")  
        .font(.system(size: 76))  
        .minimumScaleFactor(0.5)  
        .padding(.trailing, 24)  
        .lineLimit(1)  
        .frame(minWidth: 0, maxWidth: .infinity, alignment: .trailing)  
    CalculatorButtonPad()  
        .padding(.bottom)  
}
```

至此，我们使用 SwiftUI 完整构建了一个计算器 app 的界面：



预览多尺寸以及适配

在界面开发过程中，Xcode 预览可以给我们即时的反馈，该预览是通过对对应文件中的 PreviewProvider 进行提供的。在计算器 ContentView 的例子里，对应代码如下：

```
struct ContentView_Previews : PreviewProvider {
    static var previews: some View {
        ContentView()
    }
}
```

示例中，对于按钮尺寸和字体大小等，我们是通过硬编码的方式写死在程序中的。这在我们预览所使用的 iPhone XR 屏幕上表现良好，但是如果在小屏幕上，可能就不尽人意了。通过修改 `previews`，我们可以同时预览多个屏幕尺寸。将 `previews` 改写为：

```
static var previews: some View {  
    Group {  
        ContentView()  
        ContentView().previewDevice("iPhone SE (2nd generation)")  
    }  
}
```

这时，在原来的预览设备下方，Xcode 将为我们追加显示 UI 在小屏 iPhone SE 的情况。可以看到原尺寸的 UI 在小屏幕下是不可接受的：



最简单粗暴的做法，是把整个界面进行缩放。View 提供了 `.scaleEffect` modifier 来进行缩放。我们所设计的界面是基于 414 宽度的，在 `ContentView.swift` 的最上方添加一个按照当前屏幕宽度计算的比例：

```
let scale: CGFloat = UIScreen.main.bounds.width / 414
```

然后将 VStack 按照 scale 缩放即可：

```
var body: some View {
    VStack(spacing: 12) {
        // ...
    }
    .scaleEffect(scale)
}
```

不过，正如上面所说，这只不过是一种简单粗暴的做法，而且是会带来很多问题的。在本章的练习中我们会看到并尝试解决其中一个问题。在具体实践里，对不同设备的适配可能会更加复杂，所需要采取的策略也会随设计的不同而改变，因此并没有什么“银弹”和“真理”。但是这并不意味着我们束手无策：正确理解常见的 SwiftUI 布局的方式和尺寸计算的原则，会对实现各种不同界面带来坚实的理论支持和指导。为了达到这个目的，需要多多练习，并尝试思考和解释为何会得到预览的结果。

总结

本章着重于介绍了 `Text` 和 `Button` 这两种 UI 中的重要元素，并通过组合 `HStack` 及 `VStack` 等 container 进行了简单的界面布局。这些控件配合上各式各样的 modifier，构成了使用 SwiftUI 制作用户界面的基本元素。

本章中我们没有涉及到任何有关数据的内容，也没有像是列表、动画或者手势等界面构建时的常见话题。在这一章里，我们尝试把注意力集中 SwiftUI 的基本使用和布

局上。我们一步步看到了如何从一个最简单的文本开始，将 View 进行层级抽象，并用简单的基本元素构建出相对复杂的界面。同时，我们保持了顶层 ContentView 的简单定义，这让今后的修改和扩展都相对轻松。

练习

1. Modifier 顺序的影响

在 CalculatorButton 中，我们使用了一系列的 modifier 来定义按钮和文本的样式：

```
var body: some View {
    Button(action: action) {
        Text(title)
            .font(.system(size: fontSize))
            .foregroundColor(.white)
            .frame(width: size.width, height: size.height)
            .background(Color(backgroundColorName))
            .cornerRadius(size.width / 2)
    }
}
```

我们可以调换 background 和 cornerRadius 的顺序吗？试试看它会造成什么影响，并且对结果进行解释说明。另外，我们可以调换 font 和 foregroundColor 的顺序吗？它又会有怎么样的影响？

2. 字体颜色调整

可能你已经注意到了“AC”，“+/-”和“%”这三个按钮文本的颜色与本章一开始的设计稿不一致。请在 Assets.xcassets 的 Colors 中添加颜色，将它设置为 #353535，并将它设定为 CalculatorButtonItem.Command 按钮的文本颜色。

3. 重新绘制 CalculatorButton

在 SwiftUI 中，达到目的所能采用的手段往往不止一个。在示例中，我们使用了 Text 并通过 background 和 cornerRadius 等 modifier 的方式来绘制 CalculatorButton。在这个练习里，让我们尝试使用另外的方式。在 SwiftUI 中，我们可以用 RoundedRectangle 来绘制圆角矩形，使用 ZStack 将若干个 View 在垂直于屏幕的纵深方向进行堆叠。这些特性也正好满足 CalculatorButton 布局的需求。请尝试用 ZStack，RoundedRectangle 和 Button，在保持样式一致的前提下，重新绘制 CalculatorButton。

提示：你可能需要使用 frame modifier 来限定每个按钮的尺寸，并使用 .fill 来填充圆角矩形。当然，除了 ZStack 之外，像是 .overlay 或者 .background 这些 modifier，也能接受任意的 View 并达成同样的效果。

4. 使用 frame 和 Spacer 进行填充

本章示例中，最终选择了如下布局方式：

```
 VStack(spacing: 12) {  
     Spacer() // 1  
     Text("0")  
     // ...  
     // 2  
     .frame(minWidth: 0, maxWidth: .infinity, alignment: .trailing)  
     CalculatorButtonPad()  
     .padding(.bottom)  
 }
```

1. 我们用 Spacer 的方法完成将整个界面推至屏幕底部。

- 用 `frame` 方法对用于显示结果的 `Text` 进行了右对齐。

这两种方式在某种程度上来说是“等效的”，请尝试改变原来的代码，使用 `frame` 的方式完成界面下推，使用 `Spacer` 的方式实现文本右对齐。同时，请解释一下两种“占满界面”的方式有什么异同？它们分别适用于怎么样的场景？(比如多个 `Spacer` 时的行为，给定不同参数时 `frame` 表现的差异等。)

5. 验证 Dark Mode

iOS 13 中增加了 Dark Mode (深色模式) 的功能，SwiftUI 对其提供了很好的支持。在模拟器中运行 app，并将其调整为 Dark Mode，观察颜色所发生的变化。你可以通过运行时调试工具栏上的“Environment Overrides”工具来在颜色模式之间进行切换：

```
.padding(.trailing, 24 * scale)
```

Environment Overrides

Interface Style



Light



Dark

Text



Dynamic Type



Accessibility



- Increase Contrast
- Reduce Transparency
- Bold Text
- Reduce Motion
- On/Off Labels
- Button Shapes
- Grayscale
- Smart Invert
- Differentiate Without Color

```
var body: some View {
```

现有情况下 app 是不是已经能很好地适配 Dark Mode？如果我们为 ContentView 中的 Text 设定一个 foregroundColor 的话（比如 .black），它是否还能适配 Dark Mode？将这个颜色改为 .primary 呢？

如果想要在 Dark Mode 下更改按钮颜色，我们应该如何做？请通过 Assets.xcassets 中 Colors 的设置，为各个按钮在深色模式下设定不同的颜色。

小技巧：你可以在预览中使用在 ContentView() 后面添加 environment(\.colorScheme, .dark) 来快速检查深色模式下的 UI。

6. iPad 的预览以及调整

我们在预览中确认了 iPhone XR 和 iPhone SE 的情况。示例中使用了 scaleEffect 来将 iPhone XR 的界面缩小成了 iPhone SE，看似是解决了适配问题，但是这真的是可行方案吗？请尝试一下在预览中添加一个大屏幕的设备，比如“iPad Air (3rd generation)”，通过 scaleEffect 得到的 UI 还能正确工作吗？

整体使用 scaleEffect 进行缩放，会在布局时带来困难，因为它只是对视觉上进行了缩放，而布局还是依照原有的尺寸进行。相对于整体的 scale 操作，可能对单个具体元素的 frame 进行缩放效果会好一些，请你尝试调整示例中的代码，让界面在 iPad 上也能正常显示。

当然，如果我们在上面提到过的，SwiftUI 中解决问题的方法不止一种，因此可能的答案也不止一个。有余力的话，你可以尽可能尝试多种解决方案，并多思考它们的异同。

数据状态和绑定

3

上一章里，我们完成了计算器 app 的界面。整个 app 只有一个页面，在按下 CalculatorButton 后打印了当前按钮的 title：

```
CalculatorButton(  
    title: item.title,  
    size: item.size,  
    backgroundColorName: item.backgroundColorName,  
    foregroundColor: item.foregroundColor)  
{  
    print("Button: \(item.title)")  
}
```

到现在为止，我们还没有涉及到如何使用数据让 app 界面真正能被使用。在 SwiftUI 里，用户界面是严格被数据驱动的：在运行时，任何对于界面的修改，都只能通过修改数据来达成，而不能直接对界面进行调整和操作。相比于传统的 UIKit 或 AppKit，这在一定程度上对灵活性进行了限制，强制了我们必须使用更合理的数据处理方式。不过另一方面，它也规范了 SwiftUI 中数据流动的方式，让开发者更不容易犯错。在本章中，首先我们会为计算器 app 添加基本的运算逻辑，并将其作为 model 使界面正确工作。之后，我们会深入探索 SwiftUI 里 @State, @ObservedObject 和 @EnvironmentObject 等几个和数据传递相关的方式之间的异同及选择。基于这些讨论，我们会尝试实现一个计算器的“历史回溯”，让这个 app 能在各个结果之间跳转，并具有撤销和重做功能。通过这些例子，我们会看到在 SwiftUI 中数据传递和使用的一般方式。

你可以在随书附带的源码中找到位于“3.Data-and-Binding”文件夹下的“Calculator-Starter”项目作为起始，这可以保证之后你的实践能与书中内容对应。

Calculator 模型

除了新增了一个还未使用的 `CalculatorBrain.swift` 以外，Starter 项目中绝大部分内容和上一章的最终内容一样。`CalculatorBrain` 是计算器 app 的逻辑部分的实现，本书想要专注于 SwiftUI 的内容，因此我们不会逐行讲解这个类型的具体实现，也不关心计算器本身是如何运作的。但是，在继续之前，我还是想要借此机会讨论一下数据流动的方式。

使用 enum 定义 app 状态

`CalculatorBrain.swift` 中所定义的核心是 `CalculatorBrain` 这个枚举类型，它由四个带有关联值的成员组成，代表了一个计算器可能处于的四种状态：

```
enum CalculatorBrain {
    case left(String) // 1
    case leftOp(
        left: String,
        op: CalculatorButtonItem.Op
    ) // 2
    case leftOpRight(
        left: String,
        op: CalculatorButtonItem.Op,
        right: String
    ) // 3
    case error // 4
}
```

一个计算器的输入算式的操作可以概括为“左侧数字 + 计算符号 + 右侧数字 + 计算符号或等号”。将输入的阶段进行总结，可以得到计算器必然处于下面的某一个状态：

1. 计算器正在输入算式左侧数字，这个状态将在用户按下计算操作按钮 (加减乘除号) 后改变为下一个状态。
2. 计算器输入了左侧数字和计算符号，等待开始输入右侧数字。
3. 计算器已经输入了左侧数字，计算符号，和部分右侧数字，并在等待更多右侧数字的输入。
4. 输入或计算结果出现了错误，无法继续。比如发生了“除以 0”的操作。

假设当前状态处于 `.left("3")`，这表示用户已经按下了数字“3”，接下来的输入可能会是数字，小数点或者运算符号。数字 `4` 会让状态变为 `.left("34")`，而小数点将会让状态变为 `.left("3.")`；运算符 `+` 则会让状态变为 `leftOp(left: 3, op: .plus)`。其他的状态间的转换亦可推理得出。

纯函数式事件响应循环

这些状态最终需要表示在计算器 app 的 UI 上，并且需要根据用户输入来变化。

`CalculatorBrain` 类型对外提供一个用于显示结果字符串的 `output` 属性和一个用来接受输入的按钮数据的 `apply(item: CalculatorButtonItem)` 方法。前者是一个字符串：

```
var output: String
```

它对 `CalculatorBrain` 当前所处的状态进行格式化，然后返回需要在界面上显示的字符串。我们使用一个 `NumberFormatter` 来将结果的数字小数点后位数限定在八位以内：

```
var formatter: NumberFormatter = {
  let f = NumberFormatter()
  f.minimumFractionDigits = 0
  f.maximumFractionDigits = 8
```

```
f.numberStyle = .decimal  
return f  
}()
```

举例来说，当 CalculatorBrain 处于正在输入左侧数字的 left 状态时，output 就只需要的是当前的 left 数字：

```
var output: String {  
    let result: String  
    switch self {  
        case .left(let left): result = left  
        // 其他 case  
        // ...  
    }  
    guard let value = Double(result) else {  
        return "Error"  
    }  
    return formatter.string(from: value as NSNumber)!  
}
```

另一个可供外部使用的是 apply(item:) 方法。它接受按钮事件 CalculatorButtonItem，返回新的 CalculatorBrain 状态：

```
func apply(item: CalculatorButtonItem) → CalculatorBrain {  
    switch item {  
        case .digit(let num):  
            return apply(num: num)  
        case .dot:  

```

```
    return apply(op: op)

  case .command(let command):
    return apply(command: command)

  }
}

}
```

这是一个和状态有关的事件循环：在函数中，我们按照输入的 CalculatorButtonItem 类型的不同，分别进行处理；这些输入作用在 self 上，生成新的状态，并等待下一个输入。这里的 apply(item:) 所得到的结果，除了和输入的 CalculatorButtonItem 有关外，也和 self 有关，因此它不是严格意义上的纯函数。不过，因为 self 本身是个 enum，它不再含有任何更多的存储属性，因此也就不存在更多的内部状态。我们可以很容易地创建一个新的类型，并用纯函数来替代它：

```
typealias CalculatorState = CalculatorBrain

typealias CalculatorStateAction = CalculatorButtonItem

struct Reducer {
  static func reduce(
    state: CalculatorState,
    action: CalculatorStateAction
  ) -> CalculatorState
  {
    return state.apply(item: action)
  }
}
```

现在，Reducer.reduce(state:action:) 就是一个纯函数了：一旦输入的 CalculatorBrain 和 CalculatorButtonItem 确定，接下来的状态也即确定。

如果你对函数式编程不熟悉，可能会对“纯函数”的概念比较陌生。纯函数指的是，返回值只由调用时的参数决定，而**不依赖于任何系统状态，也不改变**其作用域之外的变量状态的函数。我们强调纯函数，是因为其中不存在复杂的依赖关系，理解起来非常简单。而其优秀的特性，也让我们始终可以通过测试来确保逻辑正确。

纯函数，或者像 `apply(item:)` 这样可变依赖很少的类似纯函数的方式，和 SwiftUI 配合得天衣无缝。SwiftUI 强调 `single source of truth`，一个稳定可测试的 model 是 app 状态清晰的最重要的保证，也是在 SwiftUI 中传递状态时的首先需要考虑的方向。

Store, Action, Reducer

你可能注意到了，在上面的例子中，我们特意添加了几个 typealias，把 `CalculatorBrain` 重命名成了 `CalculatorState`，把 `CalculatorButtonItem` 称为 `CalculatorStateAction`。如果你对 Redux 或者 Flux，甚至是它们的思想来源 Elm 有了解的话，会对这些命名感到十分亲切。

以 Redux 为代表的状态管理和组件通讯架构，在近来的前端开发中很受欢迎。它的基本思想和步骤如下：

1. 将 app 当作一个状态机，状态决定用户界面。
2. 这些状态都保存在一个 Store 对象中，被称为 State。
3. View 不能直接操作 State，而只能通过发送 Action 的方式，间接改变存储在 Store 中的 State。
4. Reducer 接受原有的 State 和发送过来的 Action，生成新的 State。
5. 用新的 State 替换 Store 中原有的状态，并用新状态来驱动更新界面。

用图可以将这个过程表示为：

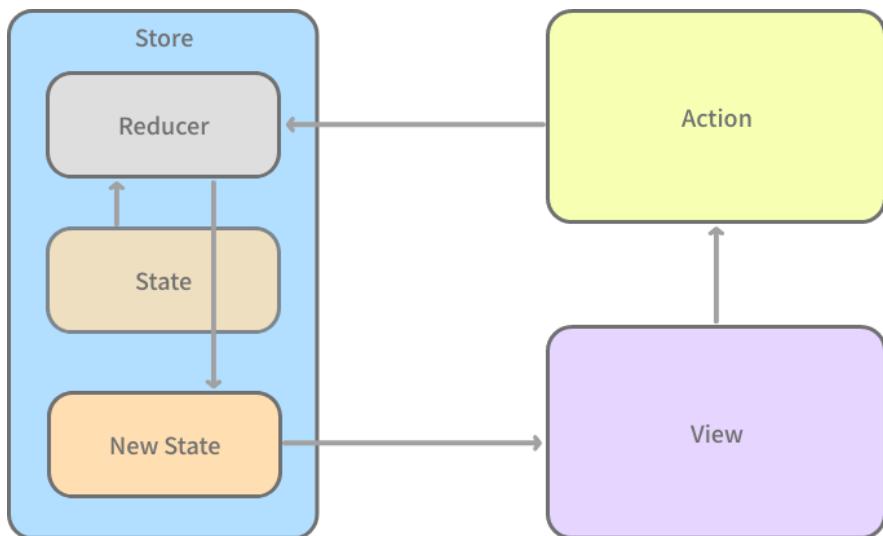


Figure 3.1: Redux 架构

这套架构方式在大中型的 app 中会让数据交互和状态管理十分清晰，但是它也引入了不少抽象概念。和 Swift 中的 protocol 为类型系统增加的复杂度类似，类 Redux 的架构在数据流动时也增加了一些额外的步骤。在大中型项目里，这么做可以硬性规范数据流动的方式，让 app 状态统一，但是在一些小项目里，这样的架构并不能起到太大作用，引入额外的复杂度得不偿失。

在本章中我们看到的 Calculator 就是这样一个小项目，它既没有异步 API 带来状态同步问题，也没有视图层需要从各个来源获取数据的问题。所以我们在这里先使用最直接的 CalculatorBrain，而不去引入更复杂的架构管理这把牛刀。不过，在本书的后半部分的示例项目中，我们会看到使用该架构管理 app 状态的实际例子。

另外需要说明的是，类 Redux 的架构并不是 SwiftUI 所必要的架构，也不是唯一的处理复杂数据的架构。它只不过是前端领域里类似范畴中一个较成熟，也很适合 SwiftUI 的方案。Apple 以及开发者社区中，现在对处理 SwiftUI 数据流应该采用怎样的架构，还并没一个统一的结论。

@State 数据状态驱动界面

CalculatorBrain 中已经包含了计算器 app 里 model 所需要的全部操作。我们现在要做的就是将它与 UI 部分关联起来。

@State

使用 `@State` 是最简单的关联方式。在 `ContentView` 中，加入一个 `brain` 属性：

```
struct ContentView : View {  
  
    @State private var brain: CalculatorBrain = .left("0")  
  
    var body: some View {  
        // ...  
    }  
}
```

和一般的存储属性不同，`@State` 修饰的值，在 SwiftUI 内部会被自动转换为一对 `setter` 和 `getter`，对这个属性进行赋值的操作将会触发 `View` 的刷新，它的 `body` 会被再次调用，底层渲染引擎会找出界面上被改变的部分，根据新的属性值计算出新的 `View`，并进行刷新。

将 `ContentView` 的 `body` 替换为下面的代码：

```
var body: some View {
    VStack(spacing: 12) {
        Spacer()
        Text(brain.output) // 1
            .font(.system(size: 76))
            .minimumScaleFactor(0.5)
            .padding(.trailing, 24 * scale)
            .frame(
                minWidth: 0,
                maxWidth: .infinity,
                alignment: .trailing
            )
        Button("Test") { // 2
            self.brain = .left("1.23")
        }
        CalculatorButtonPad()
            .padding(.bottom)
    }
}
```

1. 将表示结果的 Text 的字符串值替换为实际的输出 brain.output。
2. 为了方便测试，我们添加了一个按钮，在点击时，它对 brain 的状态值进行直接设置。

运行 app，由于一开始 brain 的值为 .left("0")，因此初始显示为“0”。在按下“Test”按钮后，brain 被设置为新的值 .left("1.23")。这一状态值的改变将触发 UI 刷新，让界面上的结果值变为“1.23”。

@Binding

在 app 中，计算器的按钮事件被定义在 CalculatorButton 中：

```
struct CalculatorButton : View {  
    // ...  
    let action: () → Void  
    var body: some View {  
        Button(action: action) {  
            Text(title)  
            // ...  
        }  
    }  
}
```

CalculatorButton 被层层包围，“深埋”在 CalculatorButtonPad 和 CalculatorButtonRow 中。它没有办法直接访问和改变顶层 ContentView 中的 brain 属性。@State 属性值仅只能在属性本身被设置时会触发 UI 刷新，这个特性让它非常适合用来声明一个值类型的值：因为对值类型的属性的变更，也会触发整个值的重新设置，进而刷新 UI。不过，在把这样的值在不同对象间传递时，状态值将会遵守值语义发生复制。所以，即使我们将 ContentView 里的 brain 通过参数的方式层层向下，传递给 CalculatorButtonPad 和 CalculatorButtonRow，最后在按钮事件中，因为各个层级中的 brain 都不相同，按钮事件对 brain 的变更也只会作用在同层级中，无法对 ContentView 中的 brain 进行改变，因此顶层的 Text 无法更新。

@Binding 就是用来解决这个问题的。和 @State 类似，@Binding 也是对属性的修饰，它做的事情是将值语义的属性“转换”为引用语义。对被声明为 @Binding 的属性进行赋值，改变的将不是属性本身，而是它的引用，这个改变将被向外传递。

修改 CalculatorButtonPad 和 CalculatorButtonRow 的定义，分别为它们加上 @Binding 的 brain：

```
struct CalculatorButtonPad: View {
    @Binding var brain: CalculatorBrain
    // ...
}

var body: some View {
    // ...
}

struct CalculatorButtonRow : View {
    @Binding var brain: CalculatorBrain
    // ...
}

var body: some View {
    // ...
}
```

接下来，把 ContentView 中的 brain 通过 CalculatorButtonPad 的初始化方法进行传递；类似地，在 CalculatorButtonPad 中也通过初始化方法将 binding 传递到 CalculatorButtonRow 里：

```
struct ContentView : View {
    var body: some View {
        // ...
        CalculatorButtonPad(brain: $brain)
            .padding(.bottom)
```

```
        }
    }

struct CalculatorButtonPad: View {
    var body: some View {
        VStack(spacing: 8) {
            ForEach(pad, id: \.self) { row in
                CalculatorButtonRow(row: row, brain: self.$brain)
            }
        }
    }
}
```

在传递 brain 时，我们在它前面加上美元符号 \$。在 Swift 5.1 中，对一个由 @ 符号修饰的属性，在它前面使用 \$ 所取得的值，被称为**投影属性** (projection property)。有些 @ 属性，比如这里的 @State 和 @Binding，它们的投影属性就是自身所对应值的 Binding 类型。不过要注意的是，并不是所有的 @ 属性都提供 \$ 的投影访问方式。我们会在下一节中看到更详细的说明。在这里，你只需要知道 \$brain 的写法将 brain 从 State 转换成了引用语义的 Binding，并向下传递。这样一来，底层 CalculatorButtonRow 中对 brain 的修改，将反过来影响和设置最顶层 ContentView 中的 @State brain。

现在，在 CalculatorButtonRow 的 body 中，对 CalculatorButton 的点击行为进行修改，我们把被点击的 CalculatorButtonItem 传递给当前状态，然后将新状态设置给 brain：

```
struct CalculatorButtonRow : View {
    // ...
    @Binding var brain: CalculatorBrain
```

```
var body: some View {
    HStack {
        ForEach(row, id: \.self) { item in
            CalculatorButton(
                title: item.title,
                size: item.size,
                backgroundColorName: item.backgroundColorName,
                foregroundColor: item.foregroundColor
            )
        }
    }
}
```

现在运行计算器 app，并尝试输入一些数字和算式吧，一切应该已经正常工作，我们的第一个 SwiftUI app 也完整出炉了！

propertyWrapper

在继续旅程之前，我想先对上面提到过多次的 @ 属性做一些更正式的说明。在 Swift 中，这一特性的正式名称是属性包装 (Property Wrapper)。不论是 @State，@Binding，或者是我们在下一节中将要看到的 @ObjectBinding 和 @EnvironmentObject，它们都是被 @propertyWrapper 修饰的 struct 类型。以 State 为例，在 SwiftUI 中 State 定义的关键部分如下：

```
@propertyWrapper
public struct State<Value> :
```

```
DynamicViewProperty, BindingConvertible  
{  
    public init(initialValue value: Value)  
    public var value: Value { get nonmutating set }  
  
    public var wrappedValue: Value { get nonmutating set }  
    public var projectedValue: Binding<Value> { get }  
}
```

init(initialValue:), wrappedValue 和 projectedValue 构成了一个 propertyWrapper 最重要的部分。在 @State 的实际使用里：

```
struct ContentView : View {  
  
    // 1  
    @State  
    private var brain: CalculatorBrain = .left("0")  
  
    var body: some View {  
        VStack(spacing: 12) {  
            Spacer()  
            Text(brain.output) // 2  
            // ...  
            CalculatorButtonPad(brain: $brain) // 3  
            // ...  
        }  
    }  
}
```

1. 由于 `init(initialValue:)` 的存在，我们可以使用直接给 `brain` 赋值的写法，将一个 `CalculatorBrain` 传递给 `brain`。我们可以为属性包装中定义的 `init` 方法添加更多的参数，我们会在接下来看到一个这样的例子。不过 `initialValue` 这个参数名相对特殊：当它出现在 `init` 方法的第一个参数位置时，编译器将允许我们在声明的时候直接为 `@State var brain` 进行赋值。
2. 在访问 `brain` 时，这个变量暴露出来的是 `CalculatorBrain` 的行为和属性。对 `brain` 进行赋值，看起来也就和普通的变量赋值没有区别。但是，实际上这些调用都触发的是属性包装中的 `wrappedValue`。`@State` 的声明，在底层将 `brain` 属性“包装”到了一个 `State<CalculatorBrain>` 中，并保留外界使用者通过 `CalculatorBrain` 接口对它进行操作的可能性。
3. 使用美元符号前缀 (`$`) 访问 `brain`，其实访问的是 `projectedValue` 属性。在 `State` 中，这个属性返回一个 `Binding` 类型的值，通过遵守 `BindingConvertible`，`State` 暴露了修改其内部存储的方法，这也就是为什么传递 `Binding` 可以让 `brain` 具有引用语义的原因。

希望这些解释能让你对于 SwiftUI 中数据传递的方式有一些更深入的理解。对于第三方开发者来说，属性包装特性给了我们一个机会，可以在一定程度上简化语言的模板代码，并且通过“标注”的方式来改变特性。它与自定义 `getter` 和 `setter` 做的事情相似，只不过功能更强大，且不需要到处重复去写一样的代码。

举个和我们的 app 不相关的例子。比如，我们可以写一段代码，来进行货币间的转换：

```
@propertyWrapper struct Converter {  
    let from: String  
    let to: String  
    let rate: Double  
  
    var value: Double
```

```
var wrappedValue: String {
    get { "\($from) \($value)" }
    set { value = Double(newValue) ?? -1 }
}

var projectedValue: String {
    return "\($to) \($value * rate)"
}

init(
    initialValue: String,
    from: String,
    to: String,
    rate: Double
)
{
    self.rate = rate
    self.value = 0
    self.from = from
    self.to = to
    self.wrappedValue = initialValue
}
}
```

Converter 提供的 init 方法除了 initialValue 外，还接受 from, to 和 rate，它们分别代表转换货币的名称和汇率。属性中 wrappedValue 是 String 类型，表示我们希望包装一个字符串：它提供的 setter 负责将字符串转换为数字并存储(当用户输入无

法转换为数字时，用 -1 代表错误)，`getter` 则输出源货币的信息。而访问 `projectedValue` 则能得到转换后的货币信息。

我们可以通过定义来使用这些属性包装，也可以像普通的属性那样，为声明的值进行赋值：

```
@Converter(initialValue: "100", from: "USD", to: "CNY", rate: 6.88)
```

```
var usd_cny
```

```
@Converter(initialValue: "100", from: "CNY", to: "EUR", rate: 0.13)
```

```
var cny_eur
```

```
print("\(usd_cny) = \(usd_cny)")
```

```
print("\(cny_eur) = \(cny_eur)")
```

```
// 输出:
```

```
// USD 100.0 = CNY 688.0
```

```
// CNY 100.0 = EUR 13.0
```

```
usd_cny = "324.3"
```

```
// USD 324.3 = CNY 2231.184
```

几乎所有依赖 `getter` 和 `setter`，并需要多次重复同样代码的地方，都可以用属性包装的方式得到更好的解决方式。比如通过属性对 `UserDefault` 或者 `Keychain` 进行读写，对某个字符串进行格式化或者去前后段空白，为属性读写加锁等等。相关话题距离 SwiftUI 的主题有点偏离了，现在让我们回到 SwiftUI 中的数据流动这一话题。

操作回溯和数据共享

`@State` 非常适合 `struct` 或者 `enum` 这样的值类型，它可以自动为我们完成从状态到 UI 更新等一系列操作。但是它本身也有一些限制，我们在使用 `@State` 之前，对于需要传递的状态，最好关心和审视下面这两个问题：

1. 这个状态是属于单个 `View` 及其子层级，还是需要在平行的部件之间传递和使用？`@State` 可以依靠 SwiftUI 框架完成 `View` 的自动订阅和刷新，但这是有条件的：对于 `@State` 修饰的属性的访问，只能发生在 `body` 或者 `body` 所调用的方法中。你不能在外部改变 `@State` 的值，它的所有相关操作和状态改变都应该是和当前 `View` 挂钩的。如果你需要在多个 `View` 中共享数据，`@State` 可能不是很好的选择；如果还需要在 `View` 外部操作数据，那么 `@State` 甚至就不是可选项了。
2. 状态对应的数据结构是否足够简单？对于像是单个的 `Bool` 或者 `String`，`@State` 可以迅速对应。含有少数几个成员变量的值类型，也许使用 `@State` 也还不错。但是对于更复杂的情况，例如含有很多属性和方法的类型，可能其中只有很少几个属性需要触发 UI 更新，也可能各个属性之间彼此有关联，那么我们应该选择引用类型和更灵活的可自定义方式。

在本节中，我们将会设计和实现一种机制，用它来记录计算器的按键操作，并且提供 UI 让用户可以返回到任意次之前的状态。这就是一个不太适合使用 `@State` 的例子：会有多个 `View` 共享和更改模型数据，而且当前的输入状态和历史操作的输入记录之间是紧密联系的。对于这样的不适合选择 `@State` 的情况（往往这是实际数据传递中更普遍的情况），`ObservableObject` 和 `@ObservedObject` 是解决的方案。

`ObservableObject` 和 `@ObjectBinding`

如果说 `@State` 是全自动驾驶的话，`ObservableObject` 就是半自动，它需要一些额外的声明。`ObservableObject` 协议要求实现类型是 `class`，它只有一个需要实现的

属性：`objectWillChange`。在数据将要发生改变时，这个属性用来向外进行“广播”，它的订阅者（一般是 View 相关的逻辑）在收到通知后，对 View 进行刷新。

创建 `ObservableObject` 后，实际在 View 里使用时，我们需要将它声明为 `@ObservedObject`。这也是一个属性包装，它负责通过订阅 `objectWillChange` 这个“广播”，将具体管理数据的 `ObservableObject` 和当前的 View 关联起来。

重构计算器模型

作为第一步，让我们先来将 `CalculatorBrain` 作为属性，放到一个 `ObservableObject` 中。

新建文件，取名为 `CalculatorModel.swift`，并定义满足 `ObservableObject` 协议的 `CalculatorModel`：

```
class CalculatorModel: ObservableObject {  
    let objectWillChange = PassthroughSubject<Void, Never>()  
}
```

一般情况下，我们使用一个 `PassthroughSubject` 实例作为 `objectWillChange` 的值。在后面几章关于 `Combine` 的内容中，我们会更详细地介绍包括 `PassthroughSubject` 在内的知识。在这里我们只需要知道，`PassthroughSubject` 提供了一个 `send` 方法，来通知外界有事件要发生了（此处的事件即驱动 UI 的数据将要发生改变）。

在 `CalculatorModel` 里添加 `CalculatorBrain`：

```
class CalculatorModel: ObservableObject {  
  
    let objectWillChange = PassthroughSubject<Void, Never>()
```

```
var brain: CalculatorBrain = .left("0") {
    willSet { objectWillChange.send() }
}
}
```

然后在 ContentView 里将 @State 的内容都换成对应的 @ObservedObject 和 CalculatorModel:

```
struct ContentView : View {
    // @State private var brain: CalculatorBrain = .left("0")
    @ObservedObject var model = CalculatorModel() // 1

    var body: some View {
        VStack(spacing: 12) {
            Spacer()
            Text(model.brain.output) // 2
            // ...
            CalculatorButtonPad(brain: $model.brain) // 3
            // ...
        }
    }
}
```

1. model 现在是一个引用类型 CalculatorModel 的值，使用 @ObservedObject 将它和 ContentView 关联起来。当 CalculatorModel 中的 objectWillChange 发出事件时，body 会被调用，UI 将被刷新。
2. brain 现在是 model 的属性。
3. CalculatorButtonPad 接受的是 Binding<CalculatorBrain>。model 的 \$ 投影属性返回的是一个 Binding 的内部 Wrapper 类型，对它再进行属性访问 (这里

的 .brain)，将会通过动态查找的方式获取到对应的 Binding<CalculatorBrain>。

CalculatorButtonPad 通过和 @State 时同样的方式，将 brain 的 Binding 传递给 CalculatorButtonRow，并在按下按钮时重新设置状态值。这个对 model.brain 的设置，触发了 CalculatorModel 中 brain 的 willSet，并通过 objectWillChange 把事件广播出去。订阅了这个事件的 ContentView 在收到变更通知后，进行 UI 刷新。

编译和运行 app，会发现重构后的 app 行为上看起来没有任何变化。但我们需要知道，在表象之下，计算器 app 数据变更和 UI 更新的机制已经和之前不一样了。

使用 @Published 和自动生成

在 ObservableObject 中，对于每个对界面可能产生影响的属性，我们都可以像上面 brain 的 willSet 那样，手动调用 objectWillChange.send()。如果在 model 中有很多属性，我们将需要为它们一一添加 willSet，这无疑是非常麻烦，而且全是重复的模板代码。实际上，如果我们省略掉自己声明的 objectWillChange，并把属性标记为 @Published，编译器将会帮我们自动完成这件事情。将 CalculatorModel 重写为：

```
class CalculatorModel: ObservableObject {  
    @Published var brain: CalculatorBrain = .left("0")  
}
```

在 ObservableObject 中，如果没有定义 objectWillChange，编译器会为你自动生成它，并在被标记为 @Published 的属性发生变更时，自动去调用 objectWillChange.send()。这样就省去了我们一个个添加 willSet 的麻烦。关于 @Published 的具体行为和作用，我们在后面关于 Combine 的章节会详细介绍。

添加回溯操作模型

为了实现回溯操作，我们要将每一步的 CalculatorButtonItem 记录下来。在 CalculatorModel 中添加一个数组：

```
class CalculatorModel: ObservableObject {  
    // ...  
    @Published var history: [CalculatorButtonItem] = []  
}
```

每次通过按钮对 brain 的操作都应该被记录下来，因此添加一个方法，来同时完成对 brain 和 history 的维护：

```
class CalculatorModel: ObservableObject {  
    // ...  
  
    func apply(_ item: CalculatorButtonItem) {  
        brain = brain.apply(item: item)  
        history.append(item)  
    }  
}
```

回到 ContentView 中，现在我们需要使用 model，而不是 \$model.brain。将 CalculatorButtonPad 和 CalculatorButtonRow 中的 brain 属性删去，换为对 model 的引用：

```
struct CalculatorButtonPad: View {  
    // @Binding var brain: CalculatorBrain  
    var model: CalculatorModel
```

```
// ...
}

struct CalculatorButtonRow : View {
    let row: [CalculatorButtonItem]
    // @Binding var brain: CalculatorBrain
    var model: CalculatorModel

    // ...
}
```

最后，更新对应的初始化方法，修改为传递 CalculatorModel 的形式：

```
struct ContentView : View {
    @ObservedObject var model = CalculatorModel()
    var body: some View {
        VStack(spacing: 12) {
            Spacer()
            Text(model.brain.output)
            // ...
            CalculatorButtonPad(model: model) // 修改部分
            // ...
        }
    }
}

struct CalculatorButtonPad: View {
    var model: CalculatorModel
```

```
// ...
var body: some View {
    VStack(spacing: 8) {
        ForEach(pad, id: \.self) { row in
            // 修改部分
            CalculatorButtonRow(row: row, model: self.model)
        }
    }
}
```

最后，在按钮事件中，直接使用 model 的 apply 方法，而不是去重新设置 brain：

```
struct CalculatorButtonRow : View {
// ...
var body: some View {
    CalculatorButton( /* */ )
{
    // self.brain = self.brain.apply(item: item)
    self.model.apply(item)
}
}
```

最后，为了说明 history 确实在操作过程中被正确更新了，我们在 ContentView 里表示结果的 Text 上方添加一个按钮：其文本为操作履历中记录的操作数量，点击后在控制台输出记录的每一步操作：

```
// ContentView
var body: some View {
```

```
 VStack(spacing: 12) {
    Spacer()
    Button("操作履历: \(model.history.count)") {
        print(self.model.history)
    }
    Text(model.brain.output)
    // ...
    CalculatorButtonPad(model: model)
    // ...
}
```

运行 app，随着按下计算器按钮，不仅原来的结果显示 Text 会变化，每次按下时“操作履历”的计数也会增加。点击操作履历按钮，可以在控制台看到如 [1, 2, 3, +, 4, =] 这样的输出。

操作回溯界面

接下来，我们要创建一个新的界面，用它来显示已经输入的历史状态，当前显示的 Output 值，以及一个用来回溯输入历史的滑块：

履历 98-3+2×4
显示 4



修改模型

我们希望滑动滑块时，履历可以按照输入历史进行回溯，并把计算结果显示在这个回溯界面和原来的计算器输出上。为了达到这些目的，我们首先需要在 CalculatorModel 上添加一些辅助属性：

```
class CalculatorModel: ObservableObject {  
    // ...  
  
    // 1  
    var historyDetail: String {  
        history.map { $0.description }.joined()  
    }  
  
    // 2
```

```
var temporaryKept: [CalculatorButtonItem] = []

// 3

var totalCount: Int {
    history.count + temporaryKept.count
}

// 4

var slidingIndex: Float = 0 {
    didSet {
        // 5
        // 维护 `history` 和 `temporaryKept`
    }
}

}

}
```

1. historyDetail 将 history 数组中所记录的操作步骤的描述连接起来，作为履历的输出字符串。
2. 在回溯操作时，除了维护 history 并让 historyDetail 反映当前的历史步骤的同时，我们也希望保留那些“被回溯”的操作，这样我们可以还能使用滑块恢复这些操作。用 temporaryKept 来暂存这些操作。
3. 滑块的最大值应当是 history 和 temporaryKept 两个数组元素数量的和。
4. 使用 slidingIndex 表示当前滑块表示的 index 值，这个值应该是 0 到 totalCount 之间的一个数字。事实上我们应该选用 Int 作为 slidingIndex 的类型，但是 SwiftUI 中代表滑块的 Slider 只接受浮点数的值。我们想要将 model 的这个属性绑定到 UI 上，需要有符合的类型。
5. slidingIndex 的 didSet 会在滑块值变动时被调用，在这里我们需要根据当前回溯的位置决定 history 和 temporaryKept 的内容。

在 CalculatorModel 最后添加下面的方法：

```
func keepHistory(upTo index: Int) {
    precondition(index ≤ totalCount, "Out of index.")

    let total = history + temporaryKept

    history = Array(total[..
```

它所做的就是根据 index 把 history 和 temporaryKept 的元素重新分配。然后根据 history 重新计算当前 brain 的状态。在 slidingIndex 的 didSet 中，调用这个方法：

```
var slidingIndex: Float = 0 {
    didSet {
        keepHistory(upTo: Int(slidingIndex))
    }
}
```

最后，当我们使用 apply(_) 添加一个按键操作时，需要将回溯时暂时使用的 temporaryKept 清空，并把 slidingIndex 设置到最后一步：

```
func apply(_ item: CalculatorButtonItem) {
    brain = brain.apply(item: item)
```

```
    history.append(item)

    temporaryKept.removeAll()
    slidingIndex = Float(totalCount)
}

}
```

回溯界面

创建一个新的 HistoryView 类型，作为回溯履历用的界面：

```
struct HistoryView: View {
    @ObservedObject var model: CalculatorModel
    var body: some View {
        VStack {
            // 1
            if model.totalCount == 0 {
                Text("没有履历")
            } else {
                HStack {
                    Text("履历").font(.headline)
                    Text("\(model.historyDetail)").lineLimit(nil)
                }
                HStack {
                    Text("显示").font(.headline)
                    Text("\(model.brain.output)")
                }
            }
            // 2
            Slider(
                value: $model.slidingIndex,
                in: 0 ... Float(model.totalCount),
```

```
        step: 1
    )
}

}.padding()
}

}
}
```

1. 在 VStack，或者是其他的类似的 View 构造的函数中，我们可以使用部分的条件语句，比如 if 或者 if else。这涉及到 ViewBuilder 这个特殊的 struct 和 Swift 内部使用的 @_functionBuilder 修饰符。我们在这里不会详细展开这方面的内容，你需要记住在类似的声明 View 的 DSL block 中，并不是所有的标准 Swift 语法都被支持。
2. 我们之前没有接触过 Slider，它是 SwiftUI 中提供的默认外观的滑动条控件，它最主要的特点是接受一个 Binding 值来显示当前滑动值，用户通过滑动操作的设置的新值，也通过 Binding 反过来设定被包装的底层变量。在这里，\$model.slidingIndex 作为 Binding 被绑定到了控件上。用户的滑动操作将直接设定 slidingIndex，并触发它的 didSet。

使用按钮显示回溯界面

最后我们要做的是，在计算器主界面点击“操作履历”按钮时，展示出 HistoryView 的界面。为此，我们对 ContentView 的操作履历按钮进行一些改造即可：

```
struct ContentView : View {
    // ...
    @State private var editingHistory = false

    var body: some View {
        VStack(spacing: 12) {
```

```
Spacer()

Button("操作履历: \$(model.history.count)") {

    self.editingHistory = true

}.sheet(isPresented: self.$editingHistory) {

    HistoryView(model: self.model)

}

// ...

}

}

}

.sheet 调用将会在它的 isPresented 为 true 的时候以 modal 的方式展示一个在尾随闭包中定义的 View (这里就是 HistoryView)。为了追踪这个 isPresented，我们需要在 ContentView 添加一个 @State: editingHistory。当 Button 被按下后，editingHistory 的值被设为 true，触发 modal 行为和 HistoryView 的展示。当用户使用手势关闭 HistoryView 时，SwiftUI 会通过 self.$editingHistory 这个 Binding 把值设回 false。
```

编译和运行 app，尝试进行一些输入，并打开历史回溯面板试试看吧。现在我们有了一个可以在任意输入状态之间进行“穿越”的强力计算器了！

使用 `@EnvironmentObject` 传递数据

作为本章的结尾，我们会对 `@EnvironmentObject` 这个看起来不怎么“友善”的修饰词进行一些讨论。

在我们的例子中，存在着一个 model 的传递链。CalculatorModel 是一个 class，这也代表了它是一个引用类型。我们在 ContentView 的变量声明里创建了这个 model：

```
struct ContentView : View {  
  
    @ObservedObject var model = CalculatorModel()  
    // ...  
}
```

为了让除 ContentView 以外的其他 View (比如 CalculatorButtonPad, CalculatorButtonRow 和 HistoryView 等) 也能访问到同样的模型，我们现在通过它们的初始化方法将 model 进行传递。这在传递链条比较短，或者是链条上每个 View 都需要 model 时是相对合理的。但是，在很多时候实际情况会不同，比如计算器例子中 CalculatorButtonPad 其实完全不需要知道 model 的任何信息，它做的仅仅是把这个值向下传递。在 SwiftUI 中，View 提供了 environmentObject(_) 方法，来把某个 ObservableObject 的值注入到当前 View 层级及其子层级中去。在这个 View 的子层级中，可以使用 @EnvironmentObject 来直接获取这个绑定的环境值。

比如，我们可以将 ContentView 里的 @ObservedObject model 换为 @EnvironmentObject：

```
struct ContentView : View {  
  
    @EnvironmentObject var model: CalculatorModel  
    // ...  
}
```

类似地，在 CalculatorButtonRow 里，也可以修改 model 的定义：

```
struct CalculatorButtonRow : View {  
  
    @EnvironmentObject var model: CalculatorModel  
    // ...  
}
```

在对应的 View 生成时，我们不需要手动为被标记为 `@EnvironmentObject` 的值进行指定，它们会自动去查询 View 的 Environment 中是否有符合的类型的值，如果有则使用它们，如没有则抛出运行时的错误。

由于 `model` 将通过 `@EnvironmentObject` 传递，而不是经由初始化方法传递，所以 `CalculatorButtonPad` 完全不再需要 `CalculatorModel`，可以将它从 `CalculatorButtonPad` 的属性声明中去掉：

```
struct CalculatorButtonPad: View {  
    // var model: CalculatorModel  
  
    // ...  
}
```

和 `@ObservedObject` 不同，`@EnvironmentObject` 不会在类型中自动创建变量，因此 `CalculatorButtonRow` 和 `CalculatorButtonPad` 的初始化方法不再会有 `model` 参数，将它们从对应的地方去掉：

```
struct CalculatorButtonPad: View {  
    // ...  
  
    var body: some View {  
        // ...  
        // CalculatorButtonRow(row: row, model: model)  
        CalculatorButtonRow(row: row)  
    }  
}
```

```
struct ContentView: View {  
    // ...  
  
    var body: some View {
```

```
// ...
// CalculatorButtonPad(model: model)
CalculatorButtonPad()

}

}
```

最后，我们在 SceneDelegate.swift 文件中创建 ContentView 的地方，通过 environmentObject 把通用的 CalculatorModel 添加上去：

```
window.rootViewController = UIHostingController(
    rootView: ContentView().environmentObject(CalculatorModel())
)
```

为了让 ContentView 的预览也能保持工作，你还需要为在ContentView_Previews 中也用同样的方式添加上 environmentObject：

```
struct ContentView_Previews: PreviewProvider {
    static var previews: some View {
        ContentView().environmentObject(CalculatorModel())
    }
}
```

运行 app，一切应该保持不变。和上面将 @State 重构为 ObservedObject 时一样，在表面上，我们看不到任何行为上的变化，但是实际上，数据流动的方式已经发生了改变。

可能一开始你会认为 @EnvironmentObject 和“臭名昭著”的单例很像：只要我们在 View 的层级上，不论何处都可以访问到这个环境对象。看似这会带来状态管理上的困难和混乱，但是 Swift 提供了清晰的状态变更和界面刷新的循环，如果我们能选择

正确的设计和架构模式，完全可以避免这种风险。使用 `@EnvironmentObject` 带来很大的便捷性，也避免了大量不必要的属性传递，这会为之后代码变更带来更多的好处。

总结

本章中，我们看到了 SwiftUI 中的几种处理数据和逻辑的方式。根据适用范围和存储状态的复杂度的不同，需要选取合适的方案。`@State` 和 `@Binding` 提供 View 内部的状态存储，它们应该是被标记为 `private` 的简单值类型，仅在内部使用。

`ObservableObject` 和 `@ObservedObject` 则针对跨越 View 层级的状态共享，它可以处理更复杂的数据类型，其引用类型的特点，也让我们需要在数据变化时通过某种手段向外发送通知（比如手动调用 `objectWillChange.send()` 或者使用 `@Published`），来触发界面刷新。对于“跳跃式”跨越多个 View 层级的状态，`@EnvironmentObject` 能让我们更方便地使用 `ObservableObject`，以达到简化代码的目的。

随着经验的积累，你会逐渐形成对于某个场景下应该使用哪种方式来管理数据和状态的直觉。在此之前，如果你纠结于选择使用哪种方式的话，从 `ObservableObject` 开始入手会是一个相对好的选择：如果发现状态可以被限制在同一个 View 层级中，则改用 `@State`；如果发现状态需要大批量共享，则改用 `@EnvironmentObject`。

本章中示例的完整代码包含在随书的“3.Data-and-Binding/Calculator-Finished”中，你可以在此基础上进行确认以及完成下面的练习。在此之后，我们会结束计算器的这个简单例子，并在下一章开启一个更加复杂、也更加贴近于真实的 iOS 开发的例子。在那里，我们将逐渐看到像是界面导航，列表，手势，动画以及网络请求等更综合的内容。

练习

1. 使用 @State 控制 Alert 弹窗

在本章例子中，我们使用了 `editingHistory` 来控制操作履历界面的显示。现在我们希望你能添加另外一个叫做 `showingResult` 的 `@State`，并用它来控制一个 `Alert`：当用户点击计算器界面的输出文本时，我们希望弹出一个对话框，并显示当前的输入算式和最后的结果，类似这样：



提示：可以使用 `onTapGesture` 为 `Text` 添加点击手势的识别，并通过 `alert(isPresented:content:)` 来显示一个弹窗 `Alert`。

你能在这个基础上试试看把 Alert 的按钮增加到两个 (“复制” 和 “取消”) 吗？并且在点击 “复制” 按钮时将计算的结果保存在 iOS 系统的剪贴板中吗 (请查阅 UIPasteboard 相关的文档和用法)？

2. 关闭回溯界面

在历史回溯界面中，我们使用 iOS 13 开始默认提供的下拉关闭的手势来返回计算器界面。请尝试在回溯界面里添加一个 “关闭” 按钮，当用户点击这个按钮时，关闭历史回溯界面。

3. (选做) 修正计算器模型中的 bug

现在计算器的 CalculatorBrain 中还存在一个 bug，那就是当我们输入算式和等号得到计算结果后，如果我们继续输入下一个数字，会发生错误。举个例子，我们顺次输入了 “ $1 + 2 =$ ” 之后，界面显示 “3”。此时如果我们想要计算 “ $4 + 5$ ” 的话，正确的计算器中按下 “4” 后界面应该显示 “4”，并等待继续输入。但是现在我们的计算器中将会显示为 “3.04”，这显然是不对的。请您尝试阅读并理解 CalculatorBrain 的代码，定位这个 bug 产生的原因，并修复它。

4. 内嵌显示 HistoryView

现在我们使用了 modal 的方式来展示 HistoryView。也许我们会更喜欢直接把算式和控制滑块内嵌在计算器界面中。请尝试直接在计算器输出结果的 Text 上方替代原先的 “操作履历” 按钮，而直接显示履历内容和履历回溯部分滑块的 UI。

5. objectWillChange 的触发机制思考

当前的 CalculatorModel 里，brain 和 history 都被标记为了 @Published，对它们进行设置会触发 objectWillChange.send()。在 apply(_:) 或者 keepHistory(upTo:) 方法中，我们既对 brain 进行了设置，又对 history 进行了设置，这会导致 objectWillChange.send() 被触发多次吗？如果是，那这又会导致 ContentView 的

`body` 被计算多次吗？有没有必要去掉 `@Published` 为我们自动生成的代码，而只在 `apply(_:)` 或者 `keepHistory(upTo:)` 中手动调用一次 `objectWillChange.send()`? 这会带来性能的提升吗?

真实世界的 SwiftUI

4

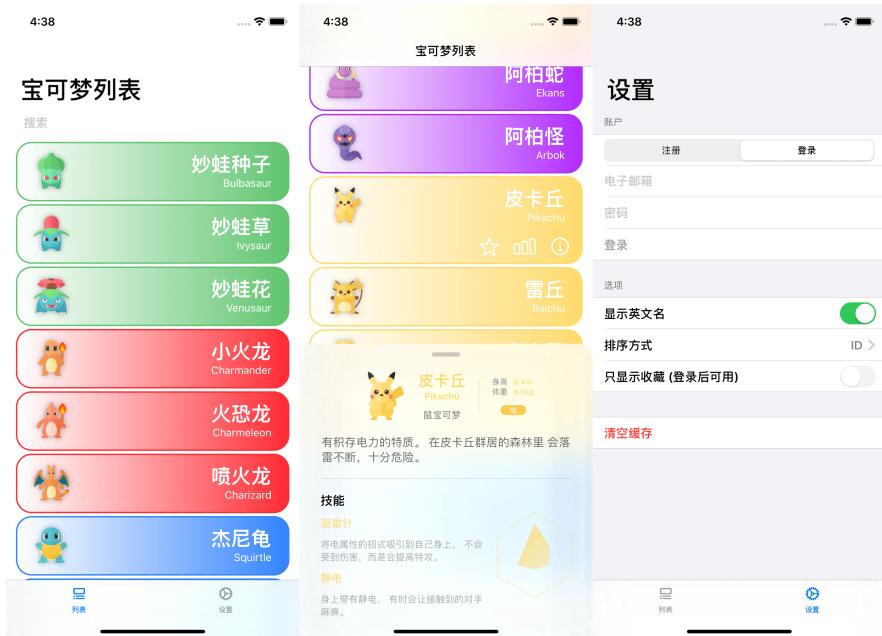
前两章的例子是我们的第一个 SwiftUI 项目，我们介绍了一个计算器界面的构建方式，看到了 HStack, VStack 和 Button 及 Text 这些基础控件的使用方式。在数据方面，则通过 @State, @Binding, Observable/@ObservedObject 和 @EnvironmentObject 按照不同层级来组织数据和状态的变更。

不论是 UI 方面，还是状态数据管理方面，计算器例子都非常简单。而实际在制作 app 时，我们会面临远比其复杂的情况。在本书接下来的部分，我们会通过一个更贴近真实世界的例子，来对使用 SwiftUI 构建 app 做进一步讲解。本章中，我们会先介绍这个示例 app 的一些基本信息，让你对 app 的各个部件和交互方式有一个整体的理解。在此基础上，去完成其中一些相对简单的 UI 构建。

示例 app：PokeMaster

本章以及后续章节中“精灵宝可梦”中英文专有名称、Pokémon形象、设计以及相关数据均为 The Pokémon Company 及其相关授权单位所有。本书中作为示例使用的宝可梦图片和资源素材为 [Artificial Design](#) 及 [Shannon E. Thomas](#) 的创作。相关数据由 [PokéAPI](#) 提供。本书对上述内容的使用目的在于提供非涉权方向的软件开发教学，读者在学习目的之外对使用这些内容需要遵守相应的使用条款。

我们要实际做的 app 名字叫做 PokeMaster，它是一个精灵宝可梦展示 app，主要界面如下：



和大部分 iOS app 类似，在最外层是一个基于 Tab 的导航结构。

第一个 tab 是一个宝可梦的列表，在每一行里我们展示了宝可梦的图片和中英文名字。点击某行将使它展开，并在下方显示“最爱”，“属性”和“详细信息”三个按钮。点击“属性”按钮时，从屏幕下方弹出一个包含宝可梦基本信息和描述的面板。“详细信息”按钮将通过 navigation push 的方式显示一个对应的网页。

第二个 tab 是常见的设置表单，其中包括了用户注册和登录的功能 (登录后才能将宝可梦添加到“最爱”)，一些控制列表显示和排序的选项，以及一个清空缓存的按钮。

在本章中，我们会集中展示如何创建列表，弹出面板和设置页面的基本结构。为了从最简单的案例开始，我们会先选择使用本地的示例数据来创建和调试界面。本章里这些内容将是独立存在的，它们不涉及实际的网络请求或者数据流动，也不会有交互发生。从下一章开始，我们会花一些篇幅介绍 Combine 框架。当我们对 Combine

有足够的认识后，我们会再次回到这个示例 app，并提出一种 SwiftUI 的架构方式，将 app 状态添加进来，并将这些页面组织成完整的 app。

在这个过程中，我们也会涉及到其他方面，比如自定义绘制，View 动画，路径动画，手势处理，URL Scheme 处理等 iOS 开发中常见的问题，并讨论它们在 SwiftUI 中对应的解决方案。

开始项目

你可以在随书源码的“4.SwiftUI-in-Real-World/PokeMaster-Starter”中找到本章的起始项目。PokeMaster 最终会直接从 [PokéAPI](#) 获取数据并进行展示，但是在构建 UI 阶段，我们更倾向于在本地完成所有操作。我们获取了 id 为 1~30 的宝可梦的数据，并将它们的 JSON 返回保存到了“JSON/pokemon”文件夹下。例如，id 编号 25 的皮卡丘可以通过访问 <https://pokeapi.co/api/v2/pokemon/25> 获得，它的内容被保存为“pokemon-25.json”。

在“Model/DataModel”文件夹下，Pokemon.swift 文件是对应这个 JSON 文件的 Codable 模型。我们只对一部分感兴趣的内容进行了解析，它们包括 types (比如 “poison”，“electric” 等表示宝可梦类型)，abilities (技能)，stats (HP，攻击，防御等属性值) 和 species 等。这些条目并不包含具体的详细内容，而是一个包含目标 URL 的结构。比如以 ID 25 的皮卡丘为例：

```
{
  "abilities": [
    {
      "ability": {
        "name": "lightning-rod",
        "url": "https://pokeapi.co/api/v2/ability/31/"
      },
      "is_hidden": true,
      "slot": 3
    }
  ]
}
```

```
        },
        {
            "ability": [
                {
                    "name": "static",
                    "url": "https://pokeapi.co/api/v2/ability/9/"
                },
                "is_hidden": false,
                "slot": 1
            ]
        ],
        ...
    }

    "species": [
        {
            "name": "pikachu",
            "url": "https://pokeapi.co/api/v2/pokemon-species/25/"
        }
    ]
}
```

如果我们还需要子条目内的额外信息，我们还需要访问对应 url 中的内容。单对于构建列表来说，只有 Pokemon 类型是不够的：中英文名字和种类的颜色等信息被存储在 species 里，使用类似的手段，我们把前三十个宝可梦的 species JSON 也下载到了本地，并存放在“JSON/species”中。类似地，“Model/DataModel”中的 PokemonSpecies.swift 文件则定义了 species 的数据模型。

对于 abilities 的处理略有不同，(在写作时) 宝可梦系列的技能有 293 个之多，而且它们并非是和宝可梦的 id 一一对应的，因此我们在“JSON/abilities”中只预存了编号为 34 和 65 的技能，这是编号前三的宝可梦(妙蛙种子，妙蛙草，妙蛙花)的技能，对应的模型文件是 Ability.swift。

我们可以选择直接读取 Pokemon，PokemonSpecies 和 Ability 这样的数据 model 内容来组合构建 UI。不过，更多时候我们会希望 UI 上需要显示的内容能和某个中间类型的属性一一对应，而不是在 View 中再去对数据 model 做变形和计算。在开发

中，对于这种“驱动 View 的 Model”中间类型，一般被叫做 ViewModel。在“Model/ViewModel”文件夹下，我们定义了两个 ViewModel：PokemonViewModel 和 AbilityViewModel。前者是对 Pokemon 和 PokemonSpecies 的组合，它提供宝可梦列表每行显示所需要的全部内容；后者是对 Ability 的封装，我们会在稍后构建弹出面板时使用到技能信息。

最后，在“Utils”文件夹下包含了一些辅助方法：FileHelper 帮助在磁盘或者 app bundle 中读写 JSON 文件；Sample.swift 中通过 FileHelper 定义的 loadBundledJSON 来从本地加载 Data Model 和 View Model。

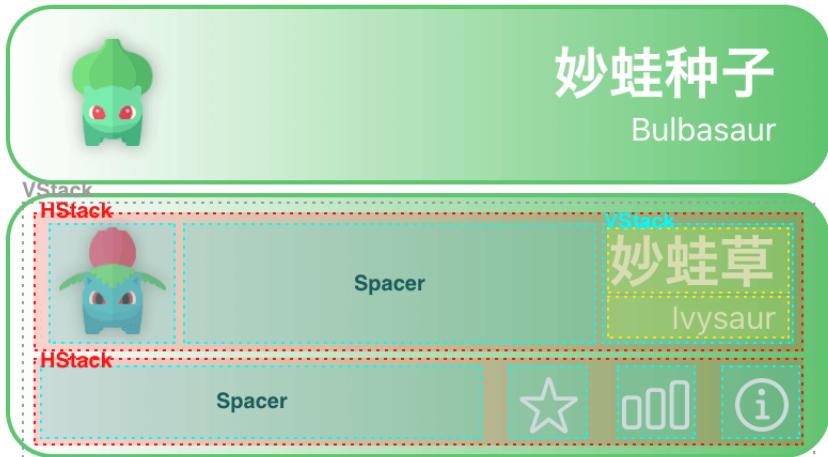
为了后续方便，我们也把图像素材和颜色定义预先放置在了“Assets.xcassets”里。在这个完整示例 app 的最后，我们会介绍如何在运行时实时地下载和使用图片，并通过 Swift Package Manager 使用第三方框架来简化我们的开发。不过一开始，我们会直接通过对应的 Image 和 Color 初始化方法从本地加载它们，我们会在后面的例子中看到这方面的使用。

创建列表

第一个任务是创建用来展示宝可梦信息的列表，而想要创建这个列表，我们需要分析和创建列表中的每一个 cell。

Cell 布局

绝大部分 cell 都可以使用嵌套 HStack 和 VStack 的方式完成，在我们的例子中也不例外。拿到设计稿后，我们可以先对它进行分析：



以 SwiftUI 模板新建一个文件，命名为 PokemonInfoRow.swift，并按照我们对 Stack 嵌套的分析，将其中 struct PokemonInfoRow 的内容进行替换，以构成 cell 的基础结构：

```
struct PokemonInfoRow: View {
    let model = PokemonViewModel.sample(id: 1)
    var body: some View {
        VStack {
            HStack {
                Image("Pokemon-\(model.id)")
                Spacer()
            }
            VStack(alignment: .trailing) {
                Text(model.name)
                Text(model.nameEN)
            }
        }
        HStack {
```

```
Spacer()  
Button(action: {}) {  
    Text("Fav")  
}  
Button(action: {}) {  
    Text("Panel")  
}  
Button(action: {}) {  
    Text("Web")  
}  
}  
.background(Color.green)  
}  
}
```

除了 `Image` 之外，其他部分我们在计算器的示例中应该都已经有接触过了。在这里，我们把预先放置在 `Assets.xcassets` 中的图片名作为参数传入给 `Image`，它将加载并按照图片本身的大小进行显示。

在预览中，我们应该可以看到一个“粗糙版本”的 `cell` 布局，其中文本和图片都不包含任何样式：



接下来，我们逐步为这个 cell 添加各种 modifier，来达到最终效果。

首先处理图片，将 `Image("Pokemon-\(model.id)")` 替换为：

```
Image("Pokemon-\(model.id)")  
    .resizable() // 1  
    .frame(width: 50, height: 50)  
    .aspectRatio(contentMode: .fit) // 2  
    .shadow(radius: 4) // 3
```

1. 默认情况下，SwiftUI 中图片绘制会优先遵从图片本身的大小。如果我们想要图片可以按照所在的 frame 缩放，需要添加 `resizable()`。
2. 图片的原始尺寸比例和使用 `frame(width:height:)` 所设定的长宽比例可能有所不同。`aspectRatio` 让图片能够保持原始比例。不过在本例中，缩放前的图片长宽比也是 1:1，所以预览中不会有什么变化。
3. 为图片增加一些阴影的视觉效果。

接下来是宝可梦的中英文名字。为 `VStack` 中的两个 `Text` 加上字体样式和颜色：

```
 VStack(alignment: .trailing) {
    Text(model.name)
        .font(.title)
        .fontWeight(.black)
        .foregroundColor(.white)
    Text(model.nameEN)
        .font(.subheadline)
        .foregroundColor(.white)
}
```

接下来，为这个外层的 HStack 加上一些 padding：

```
HStack {
    Image("Pokemon-\(model.id)")

    ...
    Spacer()
    VStack(alignment: .trailing) {
        Text(model.name)
        ...
        Text(model.nameEN)
        ...
    }
}
.padding(.top, 12) // 暂时只处理顶部间距
```

除了左右间距以外，Cell 中的第一个 HStack 已经和设计几乎吻合了：

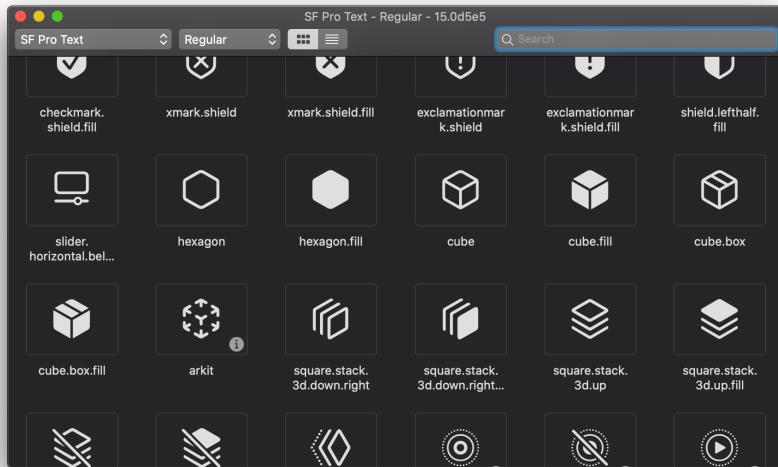


使用内置图标库 (SF Symbols)

对于第二个 HStack 中的三个按钮，我们计划使用 [SF Symbols 图标库](#) 来作为按钮图片。

SF Symbols 是从 iOS 13 和 macOS 10.15 开始内置于系统中的字符图标库，它提供了上千种常见的线条图标，而且我们可以任意地为它们设置尺寸，颜色等属性。

Apple 甚至准备了[专门的 app](#) 来帮助你查看可用的符号：



SwiftUI 的 `Image` 提供了 `Image(systemName:)` 来通过符号名生成对应图片。这种“图片”相对特殊，它的行为更接近于字体（实际上 SF Symbols 就是 San Francisco 字体的一部分），我们可以用 `.font` 和 `.foregroundColor` 等来对它进行设置。

对于第二行 `HStack` 中的三个按钮，给它们加上合适的按钮图标，同时对 `HStack` 的 `spacing` 进行一些修改：

```
HStack(spacing: 20) { // 1
    Spacer()
    Button(action: { print("fav") }) {
        Image(systemName: "star") // 2
            .font(.system(size: 25)) // 3
            .foregroundColor(.white)
            .frame(width: 30, height: 30) // 4
    }
}
```

```
Button(action: { print("panel") }) {
    Image(systemName: "chart.bar")
        .font(.system(size: 25))
        .foregroundColor(.white)
        .frame(width: 30, height: 30)
}

Button(action: { print("web") }) {
    Image(systemName: "info.circle")
        .font(.system(size: 25))
        .foregroundColor(.white)
        .frame(width: 30, height: 30)
}

.padding(.bottom, 12) // 5
```

1. 按照设计，我们需要在按钮之前添加一定的间距。
2. 使用 `Image(systemName: "star")` 来加载系统内置的 SF Symbol。
3. 可以使用 `.font` 来控制显示的大小。
4. `.font(.system(size: 25))` 虽然可以控制图片的显示尺寸，但是它并不会改变 `Image` 本身的 `frame`。默认情况下的 `frame.size` 非常小，这会使按钮的可点击范围过小，因此我们使用 `.frame(width:height:)` 来指定尺寸。因为加载后的 SF Symbol 是 `Image`，配合 `frame` 使用上面处理图像时提到的 `resizable` 和 `padding` 来指定显示范围和可点击范围也是可以的，但直接设置 `font` 和 `frame` 会更简单一些。
5. 在整个 `HStack` 底部添加了 `padding`。

View Modifier

注意到，三个 Button 的设置部分是完全重复的。我们可以像之前章节那样把重复部分提取出来，形成一个新的 View 类型。不过，我们还有另一种方法，来避免这种对 View 的重复设置，那就是创建自定义的 ViewModifier。

ViewModifier 是 SwiftUI 提供的一个协议，它只有一个要求实现的方法：

```
public protocol ViewModifier {  
    func body(content: Self.Content) -> Self.Body  
}
```

我们在 body 中对输入的 Content 进行一番配置后，返回一个 View。举个例子，新创建一个 ToolButtonModifier，将上面对 SF Symbol 的配置复制过来：

```
struct ToolButtonModifier: ViewModifier {  
    func body(content: Content) -> some View {  
        content  
            .font(.system(size: 25))  
            .foregroundColor(.white)  
            .frame(width: 30, height: 30)  
    }  
}
```

现在，我们可以把 Button 中的 View 配置换为更简单的形式了：

```
Button(action: { print("fav") }) {  
    Image(systemName: "star")  
    .modifier(ToolButtonModifier())  
}
```

我们的示例 app 相对简单，ViewModifier 可能没有太多用武之地。特别是本例中，Button 们都具有统一的行为和外观，所以使用新建类型的方式可能更好。不过，由于 ViewModifier 可以跨越页面并作用在任意 View 上，因此在大型项目中，合理使用 ViewModifier 来减少重复和维护难度会是很常见的做法。

渐变背景填充

接下来，在最外层 VStack 上设置一个合理的 frame，并为水平方向添加 padding：

```
VStack {  
    HStack {  
        // 图片, 名字  
    }  
    .padding(.top, 12)  
  
    HStack(spacing: 20) {  
        // 按钮  
    }  
    .padding(.bottom, 12)  
}  
.frame(height: 120)  
.padding(.leading, 23)  
.padding(.trailing, 15)  
.background(Color.green)
```



效果已经和我们最终的要求比较接近了。本小节中我们来解决背景渐变颜色的问题。

当前 `.background()` 接受的是单一颜色 `Color.green`。实际上这个方法可以接受任意遵守 `View` 协议的值。比如我们所需要的一个圆角矩形的形状：

```
 VStack {  
}  
// ...  
.background(  
    RoundedRectangle(cornerRadius: 20)  
)
```

和常见的一些形状 (比如圆形 `Circle`, 矩形 `Rectangle`) 一样, `RoundedRectangle` 也是一个满足 `Shape` 协议的类型, 而 `Shape` 协议本身也满足 `View`, 它是一个基于 `Path` 的绘图协议。关于 SwiftUI 中的自定义绘制和动画, 我们稍后会有专门的话题。现在, 我们只需要知道 `Shape` 是一种特殊的 `View`, 它提供了一些额外的方法。比如可以用 `fill` 来为 `Shape` 指定填充颜色:

```
RoundedRectangle(cornerRadius: 20)
```

```
.fill(Color.green)
```

在 SwiftUI 中，创建渐变非常简单，分为两个步骤：

```
// 1  
let gradient = Gradient(colors: [.white, model.color])  
// 2  
let gradientStyle = LinearGradient(  
    gradient: gradient,  
    startPoint: .leading,  
    endPoint: .trailing)
```

1. 创建一个代表渐变数据的模型：Gradient，它基本就是一系列颜色。如果我们不明确指定颜色位置的话，输入的颜色将在渐变轴上等分。比如例子中的渐变起始位置和终止位置上分别是白色和宝可梦模型中定义的颜色。
2. 选取一个渐变 style，比如线性渐变 (LinearGradient)，径向渐变 (RadialGradient) 或角度渐变 (AngularGradient)。将 1 中定义的 Gradient 和合适的渐变参数传入，就可以得到一个可适用于 Shape 上的渐变了。

将上面定义的渐变适用于背景填充：

```
.background(  
    RoundedRectangle(cornerRadius: 20)  
        .fill(  
            LinearGradient(  
                gradient: Gradient(colors: [.white, model.color]),  
                startPoint: .leading,  
                endPoint: .trailing  
            )  
        )
```

```
)  
)
```



妙蛙种子

Bulbasaur



最后，在背景部分我们还需要一个带颜色外框。对于同样的 RoundedRectangle，使用 `stroke` 来获取它的轮廓，并将轮廓和渐变背景用 ZStack 堆叠起来。同时，我们可以为整个 View 在水平方向上添加一些 padding：

```
.background(  
    ZStack {  
        RoundedRectangle(cornerRadius: 20)  
            .stroke(model.color, style: StrokeStyle(lineWidth: 4))  
        RoundedRectangle(cornerRadius: 20)  
            .fill(  
                LinearGradient(  
                    gradient: Gradient(colors: [.white, model.color]),  
                    startPoint: .leading,  
                    endPoint: .trailing  
    )
```

```
)  
}  
)  
.padding(.horizontal)
```

我们现在基本得到一个满足设计要求的 cell 了：



View 动画

View 状态切换

我们制作了带有按钮的“展开”版本，将按钮隐藏并减小 frame 的高度，我们就可以得到非展开的普通状态下的 cell。为了控制 cell 的展开状态，我们可以在 PokemonInfoRow 中引入一个 @State 来控制 cell 的展开状态，我们顺便修改一下 model 的数据来源，让它接受外部输入：

```
struct PokemonInfoRow: View {  
    // let model = PokemonViewModel.sample(id: 1)  
    let model: PokemonViewModel
```

```
@State var expanded: Bool  
  
// ...  
}
```

在 PokemonInfoRow_Previews 中，我们希望同时观测若干个 PokemonInfoRow 的表现，比如：

```
struct PokemonInfoRow_Previews: PreviewProvider {  
    static var previews: some View {  
        VStack {  
            PokemonInfoRow(model: .sample(id: 1), expanded: false)  
            PokemonInfoRow(model: .sample(id: 21), expanded: true)  
            PokemonInfoRow(model: .sample(id: 25), expanded: false)  
        }  
    }  
}
```

根据 expanded 的值，我们对布局进行一些调整：

```
VStack {  
    HStack {  
        // 图片, 名字  
    }  
    .padding(.top, 12)  
    Spacer() // 1  
    HStack(spacing: expanded ? 20 : -30) { // 2  
        // 按钮  
    }  
    .padding(.bottom, 12)
```

```
.opacity(expanded ? 1.0 : 0.0) // 3
.frame(maxHeight: expanded ? .infinity : 0) // 4
}
.frame(height: expanded ? 120 : 80) // 5
.background(
    // ...
)
.padding(.horizontal)
.onTapGesture { // 6
    self.expanded.toggle()
}
```

1. 在两个 HStack 之间插入了一个 Spacer。这是为了让宝可梦图片名字的 HStack 相对固定，这样在之后的展开/收缩动画中它不会任意移动。其实通过精确计算 frame 高度也可以达到同样的效果，不过加入这个 Spacer 让我们可以用灵活的方式应对今后 cell 高度变化所带来的变动。
2. 非展开状态下，将按钮的 HStack 间距设定为 -30。因为在非展开时我们直接隐藏了按钮行，所以暂时看不出效果，这是为了之后的动画所做的设定。
3. 通过设定透明度来隐藏按钮。
4. 当 expanded 为 true 时，设定按钮的 HStack 填满剩余高度；false 时，将 frame 高度设为 0，它将不占用外部的布局的高度。
5. 当非展开状态时，将 cell 高度设为 80。
6. 为这个 cell 添加了一个点击的手势，在触发时将 expanded 状态进行翻转。Tap Gesture 是最简单的手势操作，我们在后面更详细地介绍其他手势。

完成这一系列设定后，Preview 画面中应该会显示两种不同状态的 cell。如果我们点击运行按钮，应该可以通过点按 cell 进行展开切换：



妙蛙种子

Bulbasaur



烈雀

Spearow



皮卡丘

Pikachu

隐式动画和显式动画

现在切换还有些生硬，让我们来为它添加一些动画效果。SwiftUI 中的动画有两种类型：隐式动画和显式动画。

通过 View 上的 `animation` 修饰，就可以在 View 中支持动画的属性发生变化时自动为整个 View 添加上动画支持了。这种动画并没有指明运行时机，它会在 View 中属性改变时自动触发。在手势部分的代码前面添加上 `animation`：

```
 VStack {  
 // ...  
 }
```

```
.frame(height: expanded ? 120 : 80) // 5  
.background(  
    // ...  
)  
.padding(.horizontal)  
.animation(.default) // 隐式动画  
.onTapGesture {  
    self.expanded.toggle()  
}
```

此时点击 cell 导致的展开/收缩将以默认的动画 (Animation.default, 实质上是一个 easeInOut 动画) 呈现。除了最简单的 .default 动画, 我们也可以选择其他动画方式, 并为它设置持续时间, 动画延迟, 是否重复等。结合这些, 我们可以设计出一些“令人难忘”的动画效果。比如:

```
.animation(  
    Animation  
        .linear(duration: 0.5)  
        .delay(0.2)  
        .repeatForever(autoreverses: true)  
)  
)  
// 毫无意义的动画, 请不要用在实际项目里!
```

隐式动画的作用范围很大: 只要这个 View 甚至是它的子 View 上的可动画属性发生变化, 这个动画就将适用。很多时候这并不是我们想要的效果, 比如说例子中的 cell, 我们只希望它在被点击时进行动画, 而不需要它在初始加载显示时以动画方式呈现。这种情况下, 显式的动画可能更为合适。

显式动画通过明确的 `withAnimation` 调用触发，我们可以将改变 app 状态的操作放在 `withAnimation` 的闭包中，这时由闭包中状态变化所触发的 View 变化，将以动画形式呈现。比如，将上面的 `.animation` 语句去掉，转而在 `onTapGesture` 中使用显式动画：

```
 VStack {  
    // ...  
}  
// ...  
.padding(.horizontal)  
.onTapGesture {  
    withAnimation {  
        self.expanded.toggle()  
    }  
}
```

`withAnimation` 默认也将使用 `.default` 动画，当然，我们也可以创建效果更好的动画，比如一个模拟弹簧给出力学曲线，并让动画显得生动和自然的弹性动画。你不妨运行预览界面实际试试看！

```
 VStack {  
    // ...  
}  
// ...  
.padding(.horizontal)  
.onTapGesture {  
    withAnimation(  
        .spring(  
            response: 0.55,  
            dampingFraction: 0.425,  
    )  
}
```

```
        blendDuration: 0
    )
)
{
    self.expanded.toggle()
}
}
```

List 和 ScrollView

对 cell 应该已经很满意了，接下来就是把它们放到列表中。我们新建一个 SwiftUI View 文件 PokemonList.swift 来构建这个列表。

```
struct PokemonList: View {
    var body: some View {
        List(PokemonViewModel.all) { pokemon in
            PokemonInfoRow(model: pokemon, expanded: false)
        }
    }
}
```

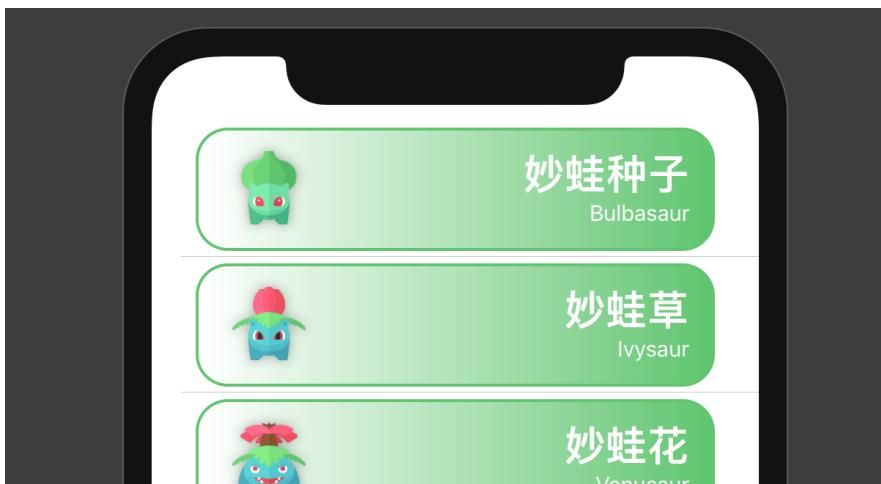
List 是最传统的构建列表的方式。它接受一个数组，数组中的元素需要遵守 Identifiable 协议。该协议只有一个要求，那就是用来辨别某个值的 id：

```
protocol Identifiable {
    associatedtype ID : Hashable
    var id: Self.ID { get }
}
```

这个协议除了要求作为 id 的 ID 类型满足 `Hashable` 外，并没有太多额外约束。一般来说，`Int`, `String`, `UUID` 或者甚至 `URL` 都是很好的备选方案。只要我们能够确保 id 可以用来区分不同的值，并对同一个值保持稳定就可以了。`PokemonViewModel` 已经是满足 `Identifiable` 的类型了，它的 id 就是所包装的 `Pokemon` 的 id，这样的定义满足要求，使得我们可以直接把所有可用的 `PokemonViewModel` 传递给 `List`。

```
struct PokemonViewModel: Identifiable, Codable {  
    let pokemon: Pokemon  
    var id: Int { pokemon.id }  
  
    // ...  
}
```

在预览中可以看到包含宝可梦 cell 的列表了：



现在我们遇到一点困难：`List` 的默认格式会为 `cell` 添加额外的默认 `padding`，所以左右的间距会有一些奇怪，但我们可以调整 `PokemonInfoRow` 的 `padding` 来修正这个问题。不过真正的问题在于，列表在每个 `cell` 之间添加了默认的分隔线。当

前 (Xcode 12 中) 的 SwiftUI API 并没有提供可以移除或者编辑这些分隔线的 API。我们可以通过桥接一个 UIKit 中的 UITableView 来绕过这个问题，或者使用一些“富有技巧性”的 List 变形来隐藏它们，但我不认为这是合理的解决方案。也许在未来的版本，Apple 会添加编辑 List 分隔线的 API，在此之前，我们可以选择使用 ScrollView 来继续我们的布局。

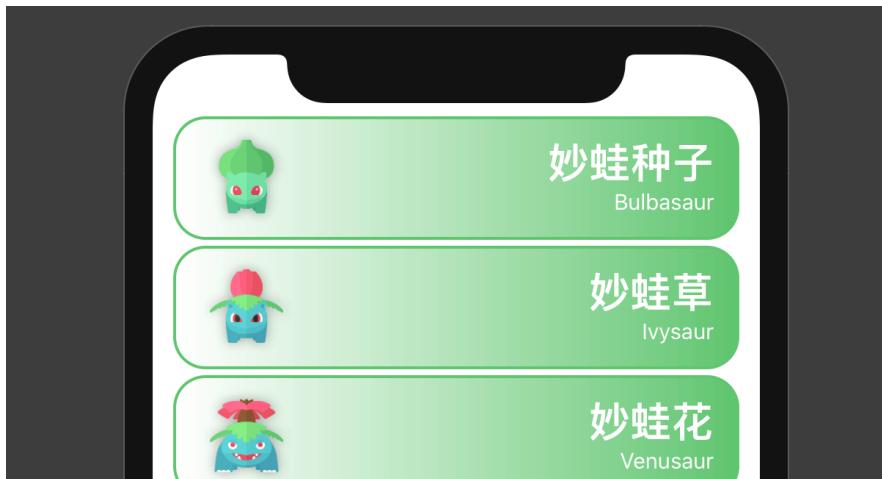
和 List 不同，ScrollView 并没有一个接受数组数据的初始化方法。我们需要使用 DSL 来描述它的内容。SwiftUI 提供了 ForEach 来帮助我们通过数组构建一系列 View。将 PokemonList 原来 body 中的方法换为：

```
struct PokemonList: View {
    var body: some View {
        ScrollView {
            ForEach(PokemonViewModel.all) { pokemon in
                PokemonInfoRow(model: pokemon, expanded: false)
            }
        }
    }
}
```

在 Xcode 12 / iOS 14 中，Apple 引入了 LazyVStack 和 LazyHStack 以仅在需要展示 View 的时候才创建 View。这是一个简单而有效的优化手段，仅需要在 Cell 外层包一层 LazyVStack 即可。那么现在 PokemonList 的 body 将会变成这个样子：

```
struct PokemonList: View {
    var body: some View {
        ScrollView {
            LazyVStack {
                ForEach(PokemonViewModel.all) { pokemon in
                    PokemonInfoRow(model: pokemon, expanded: false)
                }
            }
        }
    }
}
```

```
        }  
    }  
}  
}  
}
```



维护展开状态

在预览中运行，并尝试点击各个 cell，我们会发现现在可以同时展开多个 cell。这可能算不上问题，但我们的 app 预想设计是每次只能展开一个 cell。在展开另外的 cell 时，我们希望将现在已经处于展开状态的 cell 恢复原状。这需要我们维护一个状态，来暂存当前已经展开的 cell。

通过把 cell 的选中逻辑向外移到 PokemonList 里来解决这个问题。

首先，将 PokemonInfoRow 中的 @State 拿掉，并把点击手势移除，让 cell 成为一个不带状态的纯粹的 View：

```
struct PokemonInfoRow: View {
    let model: PokemonViewModel
    let expanded: Bool

    var body: some View {
        VStack {
            // ...
        }
        // 移除 .onTapGesture
        // .onTapGesture {
        //     withAnimation(
        //         .spring(
        //             response: 0.55,
        //             dampingFraction: 0.425,
        //             blendDuration: 0
        //         )
        //     )
        //     {
        //         self.expanded.toggle()
        //     }
        // }
    }
}
```

然后，修改 PokemonList，为它添加 @State 和手势操作：

```
struct PokemonList: View {
    // 1
```

```
@State var expandingIndex: Int?  
  
var body: some View {  
    ScrollView {  
        LazyVStack {  
            ForEach(PokemonViewModel.all) { pokemon in  
                // 2  
                PokemonInfoRow(  
                    model: pokemon,  
                    expanded: self.expandingIndex == pokemon.id  
                )  
                .onTapGesture {  
                    // 3  
                    withAnimation(  
                        .spring(  
                            response: 0.55,  
                            dampingFraction: 0.425,  
                            blendDuration: 0  
                        )  
                )  
            }  
            {  
                if self.expandingIndex == pokemon.id {  
                    self.expandingIndex = nil  
                } else {  
                    self.expandingIndex = pokemon.id  
                }  
            }  
        }  
    }  
    // ...
```

}

1. 用 `expandingIndex` 追踪选中 cell 的 `index`, `nil` 表示没有任何 cell 被选中。在 `PokemonList` 中添加 `@State` 将会使该属性被修改时触发 `ScrollView` 的计算和重绘, 以保证尺寸正确。
2. `PokemonInfoRow` 的 `expanded` 现在需要通过比较 `pokemon.id` 和存储的 `expandingIndex` 来得出。两者相同, 表示 cell 需要被展开。
3. 将 `onTapGesture` 从 cell 移出来, 并且添加上“点击已展开的 cell, 则将其缩回”的逻辑。

现在可以在预览中确认最后的列表效果了。如果愿意的话, 你也可以在 `SceneDelegate.swift` 中将 `ContentView()` 换为 `PokemonList()`, 这样你就可以在 iOS 模拟器或者真实设备中确认 UI 的表现了。

小结

`PokemonInfoRow` 是很经典的日常开发中可能遇到的 cell 的情况。对于绝大多数 cell, 我们应该都可以通过灵活组合 `VStack`, `HStack` 甚至是部分的 `ZStack` 来完成。布局方面, 利用 Xcode 预览的便利性来确认对齐方式和调整 `padding`, 可以很快接近我们的设计原稿。

如果遇到对齐方式和 `padding` 需要多次定义, 或者最后看起来构建非常复杂的话, 一般来说这是布局考虑不周的信号。尝试将重复的 `padding` 提取到外层, 甚至是重新思考布局的嵌套方式, 有时候能让代码大幅简化。

在这一小节中, 我们还看到了基本的 `List`、`ScrollView` 和 `LazyVStack` 的使用, 它们作为 cell 的容器负责展示数据, 是绝大多数“业务类”app 的核心部分。因此, 我们花了比较多的篇幅详细从 cell 的布局构思开始分析。在 SwiftUI 中创建这类业务 View 的过程和技术都大同小异, 因此在本书后面的部分, 我们不会再这样一步步详细说明 View 的构建, 而是只对关键部分和新的知识点加以说明。

创建弹出面板

接下来我们来创建弹出面板的部分。

1

2  妙蛙种子
Bulbasaur
种子宝可梦

身高 0.7m
体重 6.9kg

草 毒

3 Bulbasaur can be seen napping in bright sunlight. There is a seed on its back. By soaking up the sun's rays, the seed grows progressively larger.

4

技能 5

叶绿素
晴朗天气时，速度会提高。

茂盛
HP减少的时候，草属性的招式威力会提高。

布局

这是一个很明显的 VStack 布局：

1. 最上面一个暗示面板可以拖动的灰色 indicator。
2. 包含宝可梦图标，名字，种类，身高体重以及属性种类的 Header View。这个 Header View 自身又可以由 HStack 和 VStack 嵌套而成。
3. 宝可梦的描述。
4. 分隔线。
5. 宝可梦的技能列表。

创建一个新的 SwiftUI View 文件，命名为 PokemonInfoPanel.swift。首先我们把需要的 View Model 添加进去：

```
struct PokemonInfoPanel: View {  
  
    let model: PokemonViewModel  
  
    // 1  
    var abilities: [AbilityViewModel] {  
        AbilityViewModel.sample(pokemonID: model.id)  
    }  
  
    // ...  
}  
  
struct PokemonInfoPanel_Previews: PreviewProvider {  
    static var previews: some View {  
        PokemonInfoPanel(model: .sample(id: 1)) // 2  
    }  
}
```

1. 技能 [AbilityViewModel] 是和 PokemonViewModel 对应的，我们暂时用 sample(pokemonID:) 来避免意外地获取到不正确的技能。不过要记住，在本地我们暂时只有两个技能的 JSON，因此任何尝试加载不存在的技能的行为，会造成崩溃。我们会在后面的章节实际使用 API 返回的数据，来修正这个问题。
2. 为了使 Xcode 预览继续工作，需要传入合适的 sample data。

Top Indicator

最上面的灰色 Indicator 很简单，在 PokemonInfoPanel 中进行修改：

```
struct PokemonInfoPanel: View {  
    var topIndicator: some View {  
        RoundedRectangle(cornerRadius: 3)  
            .frame(width: 40, height: 6)  
            .opacity(0.2)  
    }  
  
    var body: some View {  
        VStack(spacing: 20) {  
            topIndicator  
  
            // 更多内容  
        }  
    }  
}
```

Header View

Header 是面板部分中最复杂的组件，以尽量减小组件为原则，我们为它创建一个新的类型：

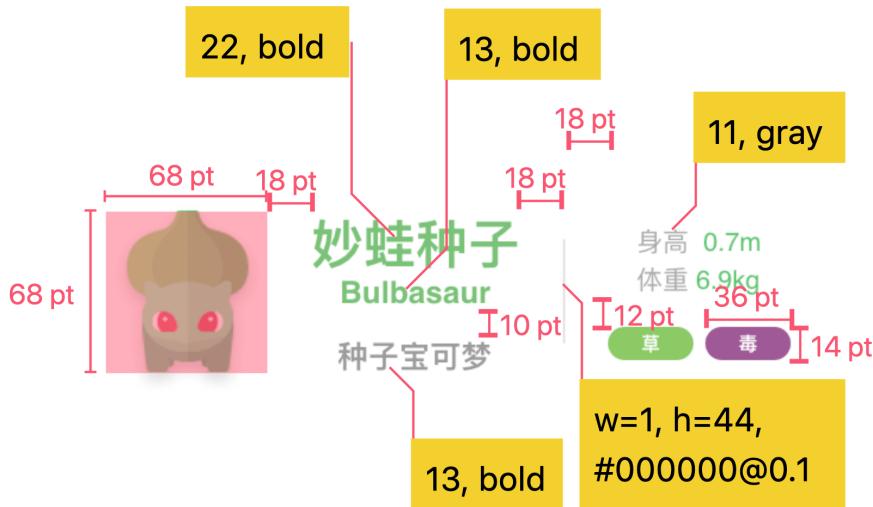
```
extension PokemonInfoPanel { // 1
    struct Header: View {
        let model: PokemonViewModel

        var body: some View {
            HStack(spacing: 18) {
                pokemonIcon
                // 2
                // nameSpecies
                // verticalDivider
                // VStack(spacing: 12) {
                //     bodyStatus
                //     typeInfo
                // }
            }
        }
    }

    var pokemonIcon: some View {
        Image("Pokemon-\(model.id)")
            .resizable()
            .frame(width: 68, height: 68)
    }
}
```

1. 相比于新建一个顶级的 `PokemonInfoPanelHeader`，将 `Header` 定义在 `PokemonInfoPanel` 的 extension 中，可以利用上下文获得更简洁明确的命名。

2. 我们给出了最外层的 HStack 和第一个要素 `pokemonIcon` 的实现。在下面，我们还给出了完整的设计图和余下的一些部件的定义（以及它们需要使用的 `ViewModel` 中属性的名字）。作为练习，请你尝试自行完成 `Header` 的实现。



// Header 中欠缺的内容

```
var nameSpecies: some View {  
    // model.name - 宝可梦中文名 (妙蛙种子)  
    // model.nameEN - 宝可梦英文名 (Bulbasaur)  
    // model.color - 主题色  
    // model.genus - 宝可梦种类 (种子宝可梦)  
}
```

```
var verticalDivider: some View {  
    // ...  
}
```

```
var bodyStatus: some View {
    // model.height - 身高
    // model.weight - 体重
    // model.color - 主题色
}

var typeInfo: some View {
    // model.types - 宝可梦属性
    // model.types[i].color - 属性颜色
    // model.types[i].name - 属性名字 (草, 毒)
}
```

SwiftUI 的特点之一是，我们往往有很多方法来对同样的 UI 进行描述，因此只要能满足设计稿要求，上面的实现并不唯一。如果你遇到困难，可以参考源码文件夹中本章 Finished 下的 PokemonInfoPanelHeader.swift 的实现。

宝可梦描述文本

描述文本相对简单，直接在 PokemonInfoPanel 中添加一个计算属性：

```
struct PokemonInfoPanel: View {
    var pokemonDescription: some View {
        Text(model.descriptionText)
            .font(.callout)
            .foregroundColor(Color(hex: 0x666666))
            .fixedSize(horizontal: false, vertical: true)
    }
}
```

`pokemonDescription` 中最后一行的 `fixedSize` 修饰符用来告诉 SwiftUI 保持 View 的理想尺寸，让它不被上层 View “截断”。对于 `Text`，默认情况下是可以显示多行文本，而不会被截断或者限制。但是，在某些情况下 `Text` 的行为会被改变，而让本应多行的文本被显示为单行（比如在 Xcode 11.1 中，发生拖拽时文本就无法完全显示，这大概率是 SwiftUI 的 bug），通过 `.fixedSize(horizontal: false, vertical: true)`，可以在竖直方向上显示全部文本，同时在水平方向上保持按照上层 View 的限制来换行。

默认情况 `Text` 会显示全部文本，如果想要限制行数（比如显示两行，其余用 ... 截断），可以设置 `.lineLimit(2)`。

技能列表

处理技能列表的方式和 `Header` 类似，我们创建一个新的 `PokemonInfoPanelAbilityList.swift` 文件，将 `AbilityList` 作为子类型嵌套至 `PokemonInfoPanel` 中：

```
extension PokemonInfoPanel {
    struct AbilityList: View {
        let model: PokemonViewModel
        let abilityModels: [AbilityViewModel]?

        var body: some View {
            VStack(alignment: .leading, spacing: 12) {
                Text("技能")
                    .font(.headline)
                    .fontWeight(.bold)
                if abilityModels != nil {
                    ForEach(abilityModels!) { ability in
                        Text(ability.name)
                    }
                }
            }
        }
    }
}
```

```
.font(.subheadline)
.foregroundColor(self.model.color)
Text(ability.descriptionText)
    .font(.footnote)
    .foregroundColor(Color(hex: 0xAAAAAA))
// 1
.fixedSize(horizontal: false, vertical: true)
}
}
}
// 2
.frame(maxWidth: .infinity, alignment: .leading)
}
}
}
```

1. `fixedSize` 的目的和 `pokemonDescription` 中相同。
2. 将技能列表的文本完全左对齐，并占满可用空间。这是设计要求，我们可以在外层控制 `padding` 来调整具体的位置。

组合起来

在 `PokemonInfoPanel` 的 `body` 中将上面各个部件组合起来，并且加上 `padding`，外围的圆角等：

```
var body: some View {
    VStack(spacing: 20) {
        topIndicator
        Header(model: model)
```

```
pokemonDescription

Divider()

AbilityList(
    model: model,
    abilityModels: abilities)

}

.padding(
    EdgeInsets(
        top: 12,
        leading: 30,
        bottom: 30,
        trailing: 30
    )
)

.background(Color.white)
.cornerRadius(20)
.fixedSize(horizontal: false, vertical: true)
}
```

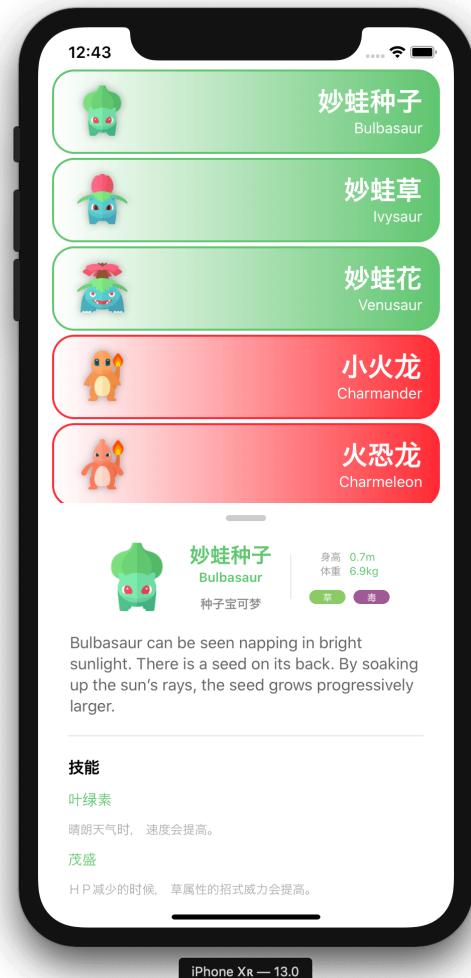
为了简单验证一下这个弹出面板的外观，我们可以在 PokemonList 里通过 overlay 的方式把它加到宝可梦列表上方：

```
struct PokemonList: View {
    // ...
    var body: some View {
        ScrollView {
            // ...
        }.overlay( // 1
            VStack {
```

```
        Spacer()  
        PokemonInfoPanel(model: .sample(id: 1))  
    }.edgesIgnoringSafeArea(.bottom) // 2  
)  
}  
}
```

1. `overlay` 在当前 View (`ScrollView`) 上方添加一层另外的 View。它的行为和 `ZStack` 比较相似，只不过 `overlay` 会尊重它下方的原有 View 的布局，而不像 `ZStack` 中的 View 那样相互没有约束。不过对于我们这个例子，`overlay` 和 `ZStack` 的行为没有区别，这里选择 `overlay` 纯粹是因为嵌套少一些，语法更简单。
2. iPhone X 系列的设备引入了 `safe area` 的概念，SwiftUI 中自定义的一般 View 是以 `safe area` 为布局边界的。如果我们想要弹出面板覆盖屏幕底部，需要明确指出忽略 `safe area`。

运行 app 展示列表，或者在 Xcode 预览中可以确认情况：



包装 UIView 类型

接下来，我们要为面板添加半透明的模糊效果。这在 iOS 系统中是非常常用的特效，UIKit 中，我们可以使用 `UIVisualEffectView`，并把其他的 View 添加到它的上方，

来实现背景模糊的效果。但不幸的是，当前 SwiftUI 中并没有直接提供类似的功能。在本书写作时，SwiftUI 还处在非常初期的阶段，难免会出现无法实现的效果或者在 SwiftUI 中无法绕过的问题。遇到这样的情况时，最简单的解决方案是把 UIKit 中已有的部分进行封装，提供给 SwiftUI 使用。

UIViewRepresentable 协议提供了在 SwiftUI 中封装 UIView 的功能。这个协议要求我们实现两个方法：

```
protocol UIViewRepresentable : View
    associatedtype UIViewType : UIView
    func makeUIView(context: Self.Context) -> Self.UIViewType
    func updateUIView(
        _ uiView: Self.UIViewType,
        context: Self.Context
    )
    // ...
}
```

makeUIView(context:) 需要返回想要封装的 UIView 类型，SwiftUI 在创建一个被封装的 UIView 时会对其调用。updateUIView(_:context:) 则在 UIViewRepresentable 中的某个属性发生变化，SwiftUI 要求更新该 UIKit 部件时被调用。创建 BlurView，让它遵守 UIViewRepresentable：

```
import SwiftUI
import UIKit

struct BlurView: UIViewRepresentable {
    // 1
    let style: UIBlurEffect.Style
```

```
func makeUIView(  
    context: UIViewRepresentableContext<BlurView>  
) -> UIView  
{  
    let view = UIView(frame: .zero)  
    view.backgroundColor = .clear  
  
    let blurEffect = UIBlurEffect(style: style)  
    let blurView = UIVisualEffectView(effect: blurEffect)  
  
    // 2  
    blurView.translatesAutoresizingMaskIntoConstraints = false  
    view.addSubview(blurView)  
    NSLayoutConstraint.activate([  
        blurView.heightAnchor  
            .constraint(equalTo: view.heightAnchor),  
        blurView.widthAnchor  
            .constraint(equalTo: view.widthAnchor)  
    ])  
    return view  
}  
  
// 3  
func updateUIView(  
    _ uiView: UIView,  
    context: UIViewRepresentableContext<BlurView>) {  
}
```

```
}
```

1. 为了更好的泛用性，我们将控制模糊样式的 `UIBlurEffect.Style` 作为成员变量使用。这样在 SwiftUI 层可以通过控制 `style` 来决定需要什么样的模糊效果。
2. 在 `UIView` 布局方面，`UIKit` 中的 Auto Layout 依然有效，按照传统的方式将 `UIVisualEffectView` 添加到上层 view 上即可。最外层返回的 view 的布局将由 SwiftUI 接手。
3. 对于模糊背景的需求，我们不需要关心更新的问题。所以把 `updateUIView` 留空。

为了使用起来方便，我们可以创建一个 View 的 extension，把 `BlurView` 的使用包装起来：

```
extension View {  
    func blurBackground(style: UIBlurEffect.Style) → some View {  
        ZStack {  
            BlurView(style: style)  
            self  
        }  
    }  
}
```

现在，我们可以把 `PokemonInfoPanel` 中的背景替换为 `BlurView`：

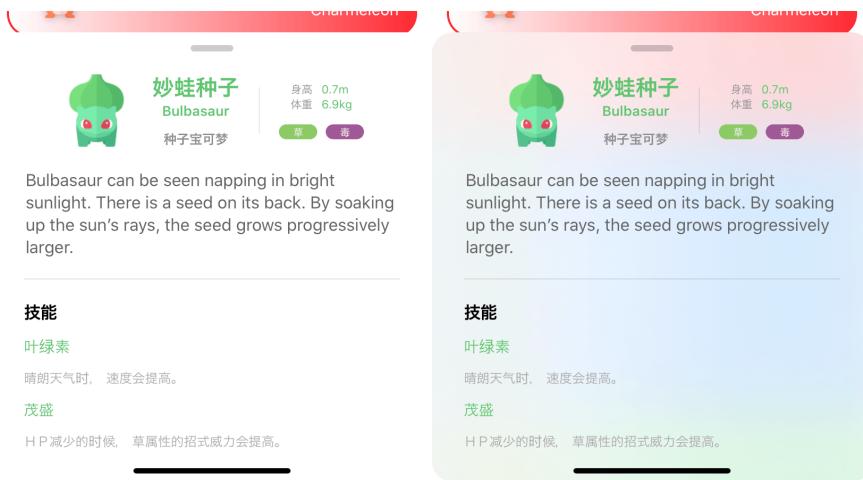
```
struct PokemonInfoPanel: View {  
    // ...  
    var body: some View {  
        VStack(spacing: 20) {  
            // ...  
    }  
}
```

```

        }
        // ...
        //.background(Color.white)
        .blurBackground(style: .systemMaterial)
        // ...
    }
}

```

运行 app，我们可以看到漂亮的毛玻璃模糊效果了(作为对比，左侧是原来的白色背景)：



创建设置界面

最后我们来看看设置界面的处理。设置界面是一个经典的 iOS 样式的表单，在 SwiftUI 中，Form 类型是用来专门处理这类需求的。

我们当然可以自己从头构建表单，在声明式的 SwiftUI 中，这并不是太困难的事情。不过利用 Form 能让我们事半功倍。我们只需要把合适的控件扔到 Form 中去，就可

以得到标准化的样式。这个样式在不同平台(比如iOS或者macOS)中会按照该平台的特点进行渲染。在iOS中它是一个分组列表(grouped list)。

本节里，我们要实现的设置界面是这样的：



Settings 模型

创建新的 SwiftUI View 文件 SettingView.swift。虽然这里我们不会涉及状态处理，不过为了更贴近实际，我们还是先来定义驱动这个 View 的数据。新建 Settings 类型，让它满足 ObservableObject：

```
class Settings: ObservableObject {
    enum AccountBehavior: CaseIterable {
        case register, login
    }

    enum Sorting: CaseIterable {
        case id, name, color, favorite
    }

    @Published var accountBehavior = AccountBehavior.login
    @Published var email = ""
    @Published var password = ""
    @Published var verifyPassword = ""

    @Published var showEnglishName = true
    @Published var sorting = Sorting.id
    @Published var showFavoriteOnly = false
}
```

除此之外，我们对 AccountBehavior 和 Sorting 两个枚举类型再定义一个 text 辅助属性，来表示它们显示在 UI 中的文本：

```
extension Settings.Sorting {
    var text: String {
```

```
        switch self {
            case .id: return "ID"
            case .name: return "名字"
            case .color: return "颜色"
            case .favorite: return "最爱"
        }
    }
}

extension Settings.AccountBehavior {
    var text: String {
        switch self {
            case .register: return "注册"
            case .login: return "登录"
        }
    }
}
```

模型中的每个成员和 UI 里的每个项目一一对应。接下来，我们要把这些模型放到 Form 中合适的控件里。首先，在 SettingView 里初始化模型：

```
struct SettingView: View {
    @ObservedObject var settings = Settings()
    var body: some View {
        Form {
            // 更多内容
        }
    }
}
```

Picker, TextField

对于第一个 section，它包括切换“注册/登录”的选项按钮，以及输入邮箱和密码的文本框。在 SettingView 中添加一个计算属性来代表这个帐号相关的 section：

```
var accountSection: some View {
    Section(header: Text("账户")) {
        // 1
        Picker(
            selection: $settings.accountBehavior,
            label: Text(""))
        {
            ForEach(Settings.AccountBehavior.allCases, id: \.self) {
                Text($0.text)
            }
        }
        // 2
        .pickerStyle(SegmentedPickerStyle())
        // 3
        TextField("电子邮箱", text: $settings.email)
        SecureField("密码", text: $settings.password)
        // 4
        if settings.accountBehavior == .register {
            SecureField("确认密码", text: $settings.verifyPassword)
        }
        Button(settings.accountBehavior.text) {
            print("登录/注册")
        }
    }
}
```

```
}
```

1. 对于“从多个值中选择一个”的情景，Picker 类型是最合适的。我们需要提供一个 Binding 作为当前被选择的对象，并在 Picker 的构建中指明可能的选项。
2. Picker 提供了多种可选择的样式，在 Form 中它默认是以 Push 导航方式在新画面中询问用户的选择。这里我们使用 SegmentedPickerStyle 来得到分段式的选项 UI。
3. TextField 和 SecureField 分别对应普通的文本输入框和密码输入框(输入显示为黑圆点)。
4. 确认密码的输入框只在注册时需要显示。

更新 SettingView 的 body，把 accountSection 添加进去：

```
var body: some View {
    Form {
        accountSection
        // optionSection
        // actionSection
    }
}
```

由于篇幅，我们没有给出 optionSection 和 actionSection 的代码，希望你可以将后两个 section 作为练习并自行实现。你可以用类似 accountSection 的方法，把合适的控件组织在 Section 中。对于“排序方式”和“清空缓存”，我们已经看到过 Picker 和 Button 的使用方式了，对于“显示英文名”和“只显示收藏”这两个选项，在 SwiftUI 中可以用 Toggle 来描述控制切换的开关。

嵌套 Navigation View

如果你在实现上面的练习时运行并确认过，也许你已经发现了两点问题：

1. 界面顶部需要“设置”标题，但是现在只有表单本身。
2. “排序方式”无法点击或进行设置，整个选项都是灰色的。

我们需要一个 `NavigationView` 来显示导航标题，并且让使用 `Push` 导航的“排序方式”按钮能够把选项页面推入到导航栈。新建 `SettingRootView.swift`，我们会在这里定义一个 `NavigationView`，它的唯一目的就是把 `SettingView` 放到导航栈中，并为它设置标题：

```
struct SettingRootView: View {  
    var body: some View {  
        NavigationView {  
            SettingView().navigationBarTitle("设置")  
        }  
    }  
}
```

现在，试试看修改 `SceneDelegate` 来显示 `SettingRootView`，一切都应该正常工作了。

总结

在这一章中，我们介绍了示例 app `PokeMaster` 的基本情况，并趁热打铁集中将这个 app 中涉及到的 UI 部分都完成了。通过这些例子，可以看到，使用 `SwiftUI` 对界面进行声明式的开发，能够以少量代码很高效地构建出合理的 UI。

我们刻意避开了状态和数据的维护，也是为了说明这些 UI 描述语句是可以和数据分离的。在构建 UI 时，不需要太关心数据流动的问题，这为使用分布式的开发方式来提高效率这一做法提供了保障。

本章完整的源码可以在“4.SwiftUI-in-Real-World/PokeMaster-Finished”中找到。如果你在阅读时遇到了任何疑惑和困难，可以参考源码进行实验。

在之后的章节中，我们会逐渐把状态和用户操作填入到这个 UI 架子中，让它变为一个实际可用的 app。这些内容包括对用户输入的判定和响应，通过网络请求获取数据和填充 UI，不同界面的关系和联动更新等。另一方面，我们也会更进一步打磨交互细节，介绍一些像是手势，绘制等更深入的话题。

不过在此之前，我建议你认真完成本章练习。特别是对于界面实现，通过自己动手获取的经验才能帮你更加驾轻就熟地应对新的问题。

练习

1. 完成 PokemonInfoPanel.Header 和 SettingView 的实现

在Header的部分，我们只给出了 `pokemonIcon` 的实现，而把 `nameSpecies`, `verticalDivider`, `bodyStatus` 和 `typeInfo` 作为练习留给了读者。同样，在构建设置界面表单时，`optionSection` 和 `actionSection` 也留作了练习。希望你已经完成了这些部分，不过如果你在阅读时选择了先跳过的话，现在是另一次实践的机会！

2. 验证 BlurView 的样式和 View update 机制

`BlurView` 满足 `UIViewRepresentable`，它是一个特殊的包装了 `UIView` 的 `SwiftUI.View`。在本章里，我们忽略了 `updateUIView(_:context:)` 并简单地把它留空。在这个练习中我们来看看它的用法。

首先，在 BlurView 的两个方法中加上一句打印语句，来确定它们的调用时机：

```
func makeUIView(  
    context: UIViewRepresentableContext<BlurView>  
) → UIView  
{  
    print("makeUIView")  
  
    // ...  
}  
  
func updateUIView(  
    _ uiView: UIView,  
    context: UIViewRepresentableContext<BlurView>)  
{  
    print("Update")  
}
```

接下来，在 PokemonInfoPanel 中进行一些修改：

```
struct PokemonInfoPanel: View {  
    // 1  
    @State var darkBlur = false  
  
    var body: some View {  
        VStack(spacing: 20) {  
            // 2  
            Button(action: {  
                self.darkBlur.toggle()  
            }) {  
                Text("切换模糊效果")  
            }  
        }  
    }  
}
```

```
        }

        topIndicator
        // ...

    }
// ...
// 3

.blurBackground(
    style: darkBlur ? .systemMaterialDark
    : .systemMaterial)
}

}
```

1. 添加一个 @State darkBlur，用来指示我们希望展示的模糊底色的类型：
2. 在 VStack 最上方新加一个按钮，来切换 darkBlur。
3. 按照 darkBlur 来选择使用 .systemMaterial 还是 .systemMaterialDark 作为模糊效果。

运行 app，切换到面板界面，观察控制台的输出情况。尝试点击按钮来切换效果，控制台输出有什么变化么？请尝试总结一下原因。

在 BlurView 中明确地添加初始化方法，并且也进行输出：

```
struct BlurView: UIViewRepresentable {

    let style: UIBlurEffect.Style

    init(style: UIBlurEffect.Style) {
        print("Init")
        self.style = style
    }

    func makeUIView(context: Context) -> UIView {
        let view = UIView()
        view.backgroundColor = .red
        return view
    }

    func updateUIView(_ uiView: UIView, context: Context) {
    }
}
```

```
}

// ...

}
```

试想一下显示面板和点击“切换模糊效果”时，输出分别应该是怎样？并验证一下。

最后，根据上面所展示的 BlurView 的声明周期和各方法调用的情况，重构 BlurView 并补写 updateUIView(_:context:) 方法，让 BlurView 能够依照 darkBlur 的设定，在两种不同风格的模糊效果之间进行切换。

3. 将 PokemonList 也放到导航中

我们已经把设置界面放到 NavigationView 里面了。按照设计，其实我们也需要把 PokemonList 也放到导航栈中。仿照 SettingRootView 的方式，尝试新建一个 PokemonRootView，并设定它的标题“宝可梦列表”。

4. 添加搜索文本框

PokemonList 中列表的最上部还需要一个搜索文本框，用来按照名字对列表中的宝可梦进行过滤，请尝试在 ScrollView 中添加这样一个文本框。你可以自行按照喜好设置它的样式及间距，并且暂时不用关心对输入数据的处理等工作。我们在后续章节会针对搜索功能布置你更多的任务和练习。

Combine 和异步 编程

5

异步编程简介

到现在为止，我们已经使用 SwiftUI 搭建了一个完整的计算器 app，并且为另一个更加复杂一些的 PokeMaster app 起了个头。在计算器的示例中，所有的计算和操作都是立即完成和返回的；在上一章的 PokeMaster 里，我们只使用了本地的 dummy 数据和状态来进行操作。这两种方式都只是使用同步的方式对状态进行读取和更新，我们还没有接触到程序设计领域一个重要而又复杂的话题：异步程序。

为了能在最后一部分中将所有内容串联起来，在这部分里，让我们先从 SwiftUI 的实践世界中抽出身来，回归到理论，介绍一下藏身在 SwiftUI 后面，负责用来处理数据的 Combine 框架。

传统的异步编程方式

相比于同步程序而言，其实异步操作和它产生的状态是客户端开发中更常见的内容。比如说，对于一个 iOS app 来说，下面的任务一般都使用异步的编程方式：

- 请求某个网络 API，并获取返回的 JSON 数据
- 等待响应用户点击某个按钮
- 将图片数据传递给某个机器学习的模型，识别其中的文字
- 从网络下载高清图片并保存到硬盘
- 进行 UI 动画并在动画结束时触发某段代码

这些操作的都有一个共同的特点，那就是操作本身会耗费很多时间。客户端开发的第一原则是**不能让用户界面卡住**，在处理这些耗时操作时，我们绝不能让 UI 去等待操作完成，它们需要和程序的其他部分“和平相处”。传统的 Cocoa 和 UIKit 提供了一系列的异步 API，它们往往以下面某种形式出现：

- **闭包回调**: 在调用需要耗时的方法时，提供一个闭包用以接收完成回调。耗时方法本身会被放到主线程之外执行；在执行完毕后，调用这个闭包来通知调用者任务已经完成。这是现代 Cocoa 开发中最常见的异步方式，比如 URLSession 就提供了闭包的网络请求方法：`dataTask(with:completionHandler:)` 中的 `completionHandler` 就是这样的闭包。
- **Delegate**: Protocol-Delegate 是 Cocoa 中的另一种常见设计模式。我们通过定义一个 protocol 和其中的方法，来描述异步行为可能产生的结果。在使用时，创建一个满足该 protocol 的 delegate，并把它设置为异步行为结果的接收者。在异步行为完成后，被调用方检查 delegate 是否存在，并尝试调用对应的方法来通知异步行为完成。像是 UITableView 中相关的 UITableViewDataSource 和 UITableViewDelegate 都是这种模式的体现。另外，URLSession 除了提供基于闭包回调的方法以外，也存在基于 protocol-delegate 的另一套 API。
- **Notification**: 在异步操作完成时，可以通过 NotificationCenter 的相关 API 发送一个通知 (Notification)，如果有观察者注册想要接收该通知，那么这个通知将被传递给观察者并调用相关代码。这是 Cocoa 中的另一种常见模式 — 观察者 (observer) 模式。大部分 UI 相关的事件，比如键盘显示或消失、文本框中内容的变更等，都提供了可订阅的通知。

这三种异步 API 的方式各自有自己的使用场景，并且互为补充：如果事件比较简单，不需要关心过程细节，只需要响应结果的话，闭包回调提供了最简单的异步方案；如果希望能控制更多细节，或者需要关心多种异步事件时，闭包 API 会存在一些设计上的困难，这时选择 delegate 会更方便；对于那些触发时机不确定的，可能长期存在的行为所对应的事件，使用监听通知的方式最为合适。

响应式异步编程模型

在上面我们不止一次提到了**事件**这个词，它是整个客户端异步编程中最核心的概念。异步编程的关键就在于，我们如何处理这些事件，并通过响应这些事件来设定程序状态，最终影响用户界面和体验。

如果程序相对简单，比如只有一两个异步操作和状态时，对这些状态的管理不会特别困难。但是现实中，我们往往需要混合使用多个不同类型的异步 API，而有时候这些不同的异步 API 会操作和设置同一个状态。我们无法确定这些异步 API 的调用顺序，这进一步使得状态的维护变得异常困难，并很快超出普通人类的理解，而这正是大多数客户端 bug 的来源。

在遇到一系列相关联的繁琐复杂的具体问题时，一种常见的解决方式是寻找这些问题的共同点，并使用更上层的抽象来总结出对各个问题都通用的方案，对其进行简化。在异步编程中，不论采用闭包，`delegate` 还是 `notification`，实际上都是在当前的时间节点上预先设置好特定的逻辑，去处理未来会发生的事件。抛开不同 API 的定义所产生的表象，异步编程的本质是响应未来发生的**事件流**。那么，我们其实完全可以用一种通用的方式来“抹去”不同异步 API 的区别，让**事件发生**这一核心概念暴露出来。在异步操作中某个事件发生时，把这个事件和与其相关的数据“**发布**”出来。而对这个事件感兴趣的代码可以订阅这个事件，来进行后续操作。

一开始听起来，这种方式在接收事件方面和 `Notification` 的方式很像，但是随后对于接收到的数据的处理，是这套抽象方法与 `Notification` 的根本区别所在。在本书一开始的简介，以及前面几章关于 SwiftUI 的基本概念中，我们已经多次强调在声明式 UI 中，用户界面只是状态的函数，即：`View = f(State)`。既然 UI 是“被动地”响应状态的变化，那么我们是不是可以将状态变化也作为“事件”来看待呢？答案是肯定的，我们可以将“状态变化”看作是被发布出来的异步操作的事件，订阅这个事件，并对订阅了事件的 `View` 根据更新后的状态进行绘制，这就是 SwiftUI 的核心逻辑。

不过，异步 API 的事件流所带有的数据，一般并不会恰好就是驱动 UI 所需要的那些状态：有可能这些事件所带有的数据需要通过一些变形和处理，才能符合我们的 UI 状态需求；有可能有一些事件是多余的，不应该用来改变 UI 状态，需要被舍弃掉；有可能我们希望“暂存”某个事件的数据，然后等待另外的事件数据，并把它们合并起来，再去改变状态。不论采用何种方式，这都意味着，我们需要对事件数据进行某种操作或者变形。



至此，我们可以总结一下响应式异步编程的抽象和特点：异步操作在合适的时机发布事件，这些事件带有数据。接下来，我们使用一个或多个操作来处理这些事件以及内部的数据。在末端，会有一个订阅者来“消化”处理后的事件和数据，并进一步驱动程序的其他部分（比如 UI 界面）的运行。上面这些对于事件和数据的操作，以及末端的订阅，都是在事件发生之前完成的。在一开始的时候，我们就将这些描述清楚，之后它便可以以预设的方式响应源源不断发生的事件流。

Combine 基础

Apple 在 Combine 框架的文档首页上给这个框架如下定义：

通过对事件处理的操作进行组合 (combine)，来对异步事件进行自定义处理（这也正是 Combine 框架的名字的由来）。Combine 提供了一组声明式的 Swift API，来处理随时间变化的值。这些值可以代表用户界面的事件，网络的响应，计划好的事件，或者很多其他类型的异步数据。

这些定义符合上面我们提到的响应式异步编程模型的抽象，Combine 为我们在 Apple 平台上处理异步编程提供了统一的实现。在响应式异步编程中，一个事件及其对应的数据被发布出来，最后被订阅者消化和使用。期间这些事件和数据需要通过一系列操作变形，成为我们最终需要的事件和数据。Combine 中最重要的角色有三种，恰好对应了这三种操作：负责发布事件的 Publisher，负责订阅事件的 Subscriber，以及负责转换事件和数据的 Operator。



响应式编程并不算什么特别新鲜的概念，在 2012 年微软的.NET Framework 3.5 开始，就已经可以通过插件的方式使用基于 C# 的响应式编程了；Java 的 [RxJava](#)，JavaScript 的 [rxjs](#) 也都是各自语言平台上非常常用的响应式扩展。在 iOS 开发中，Objective-C 时代有 ReactiveCocoa，Swift 时代有 RxSwift，虽然它们的具体实现略有差异，但是核心思想都是一致的。一部分开发者已经从响应式编程中获益良多，而 Combine 框架的出现，从系统层级对响应式编程提供了完备的支持。

在本章中，我们会对这三种角色和对应的 protocol 及方法进行一些概括性的说明，这可以让你在一个相对高的层次上俯瞰整个 Combine 框架的设计。在之后的几章中，我们会挑选一些典型的角色类型，并结合图示和案例，仔细地对各个角色的职能和实践中的用法作为解释说明。

Publisher

事件发布

在 Combine 中，我们使用 Publisher 协议来代表事件的发布者。Swift 提倡使用[面向协议编程的方式](#)，Combine 中包括 Publisher 在内的一系列角色都使用协议来进

行定义，也正是这一思想的具体体现。协议本身定义了一个 Publisher 应该具备的能力，而它们的实现则由遵守 Publisher 协议的具体类型所提供。在 Combine 框架中，已经有一系列框架自带的发布者类型，它们大部分被定义在了 Publishers 这个 enum 之中。我们会在下一章仔细研究一些常用的 Publisher 类型的行为特点，在那之前，让我们先来看一看 Publisher 共通的部分特性。

Publisher 协议中所必须的内容十分简单，它包括两个关联类型 (associatedtype) 以及一个 receive 方法：

```
public protocol Publisher {  
    associatedtype Output  
    associatedtype Failure : Error  
    func receive<S>(subscriber: S) where  
        S : Subscriber,  
        Self.Failure == S.Failure,  
        Self.Output == S.Input  
}
```

Publisher 最主要的工作其实有两个：发布新的事件及其数据，以及准备好被 Subscriber 订阅。

Output 定义了某个 Publisher 所发布的值的类型，Failure 则定义可能产生的错误的类型。随着时间的推移，事件流也会逐渐向前发展。对应 Output 及 Failure，Publisher 可以发布三种事件：

1. **类型为 Output 的新值**：这代表事件流中出现了新的值。
2. **类型为 Failure 的错误**：这代表事件流中发生了问题，事件流到此终止。
3. **完成事件**：表示事件流中所有的元素都已经发布结束，事件流到此终止。

对于第一种事件，Publisher 会直接将新的值发布出来。后两种事件在 Combine 中则使用 `Subscribers.Completion` 来描述，`Completion` 是一个含有两个成员的 enum，其中成员类型为 `.failure(Failure)` 以及 `.finished`。我们在本书后面的部分，会使用 `output`, `failure` 和 `finished` 来描述这三种事件。

为了比较容易地说明事件流，我们在本书中会大量使用下面这样的事件图。图中从左向右的横轴表示时间，圆圈表示每个 `output` 的值，时间轴最后的竖线代表了 `finished` 事件。下面的图示中，Publisher 依次发布了 20, 40, 60, 80 和 100 五个值，并在之后正常完成：



有限事件流和无限事件流

虽然 Publisher 可以发布三种事件，但是它们并不是必须的。一个 Publisher 可能发出一个或多个 `output` 值，也可能一个值都不发出；Publisher 有可能永远不会停止终结，也有可能通过 `failure` 或者 `finished` 事件来表明不再会发出新的事件。我们将最终会终结的事件流称为**有限事件流**，而将不会发出 `failure` 或者 `finished` 的事件流称为**无限事件流**。

有限事件流最常见的一个例子是网络请求的相关操作：发出网络请求后，可以把每次接收到数据的事件看作一个 `output`。在请求的所有数据都被返回时，整个操作正常结束，`finished` 事件被发布。如果在过程中遭遇到错误，比如网络连接断开或者连接被服务器关闭等，则发布 `failure` 事件。不论是 `finished` 或者是 `failure`，都表明这次请求已经完成，将来不会有更多的 `output` 发生。

上一节的图示中，100 被发出后，紧接的就是 `finished` 事件，它是一个有限事件流。

而无限事件流则正好相反，这类 Publisher 永远不会发出 failure 或者 finished。一个典型的例子是 UI 操作，比如用户点击某个按钮的事件流：如果将按钮看作是事件的发布者，每次按钮点击将发布一个不带有任何数据的 output。这种按钮操作并没有严格意义上的完结：用户可能一次都没有点击这个按钮，也可能无限次地点击这个按钮，不论用户如何操作，你都无法断言之后不会再发生任何按钮事件。

下图是一个从 1 开始每秒加一，并将结果作为 output 发布的 Publisher。因为正整数是无穷的，所以这个 Publisher 并不会完结，因此在时间轴的最后并没有代表 finished 事件的竖线的存在。



当然，因为计算机中 Int 本身是有界的，所以上图的 Publisher 只是在理论上是无限事件流，而实际上会遇到 Int 越界而导致异常，并不能真正意义上做到“无限”。不过，在 iOS app 开发中，像是按钮事件，屏幕旋转，或者文本框输入之类的 UI 事件，只要相应的控件和 UI 还存活，它就是真正的无限事件流。

对有限事件流和无限事件流进行区分，可以帮助我们理解 Publisher 的生命周期。使用一个正则表达式来表示 Publisher 的所能发布的事件的话，可以写作 `output*(failure|finished)?`。每一个事件都是状态变化的信号，对这种信号进行操作或者组合，这正是 Operator 角色所要做的工作。

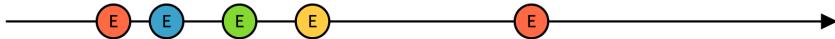
Operator

客户端的响应式编程中，由状态驱动 UI 是最核心的思想。不过，异步 API 的 Publisher 所提供的事件和数据，往往并不能够直接用来驱动决定 UI 的状态。举个简单的例子，在上面的按钮事件中，每次在按钮按下的时候我们只发送了 output 事件，而不带有任何数据。如果我们想在一个 Text 上表示某个按钮被按下的总次数的

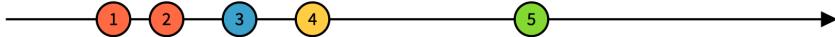
话，就需要对这些 output 事件进行统计。在 Combine 中，我们可以使用 scan 来完成累加的工作。scan 让我们提供一个暂存值，每次事件发生时我们有机会执行一个闭包来更新这个暂存值，并准备好在下一次事件时使用它。同时，这个暂存值也将被作为新的 Publisher 事件被发送出去：

```
let buttonClicked: AnyPublisher<Void, Never>
buttonClicked
    .scan(0) { value, _ in value + 1 }
```

如果我们用 E 的圆圈来表示按钮点击事件，上面的代码将把 buttonClicked 转换为一个发布点击次数的新的 Publisher，它所发布的数据的类型是 Int。



```
scan(0) { value, _ in value + 1 }
```



然而，如果我们想要直接驱动 Text，我们所需要的其实是一个 String 类型。

Combine 中 Publisher 上的 map 操作可以为我们完成事件中数据类型的转换。刚才通过 scan 所得到的是一个新的 Publisher，所以我们可以简单地在 scan 操作后添加 map 将 Int 数据转换为实际需要的 String：

```
let buttonClicked: AnyPublisher<Void, Never>
buttonClicked
    .scan(0) { value, _ in value + 1 }
    .map { String($0) }
```

和 `scan` 类似，`map` 返回的也是一个新的 `Publisher`。在 `Combine` 框架中，类似的用来操作数据和事件的函数还有很多，它们大多数都以函数式的形式出现，来对原有的 `Publisher` 进行变形等逻辑操作。这些操作符在响应式异步编程中担任的就是 `Operator` 的角色。

如果你对函数式和声明式编程还不熟悉，一开始对这种写法可能会有些不习惯。不过，它们都是一些基础单元，其作用也都很容易理解。你可以很容易地把小的基础单元分离出来，进行各种实验和学习。而另一方面，在实践中将各类基础单元的 `Operator` 高度组合后，往往可以很优雅地实现那些原本使用传统指令式编程时难以完成的任务。

在响应式编程中，绝大部分的逻辑和关键代码的编写，都发生在数据处理和变形中。每个 `Operator` 的行为模式都一样：它们使用上游 `Publisher` 所发布的数据作为输入，以此产生的新的数据，然后自身成为新的 `Publisher`，并将这些新的数据作为输出，发布给下游。通过一系列组合，我们可以得到一个响应式的 `Publisher` 链条：当链条最上端的 `Publisher` 发布某个事件后，链条中的各个 `Operator` 对事件和数据进行处理。在链条的末端我们希望最终能得到可以**直接驱动** UI 状态的事件和数据。这样，终端的**消费者**可以直接使用这些准备好的数据，而这个消费者的角色由 `Subscriber` 来担任。

Subscriber

和 `Publisher` 类似，`Combine` 中的 `Subscriber` 也是一个抽象的协议，它定义了某个类型想要成为订阅者角色时所需要满足的条件：

```
public protocol Subscriber {  
    associatedtype Input  
    associatedtype Failure : Error  
  
    func receive(subscription: Subscription)
```

```
func receive(_ input: Self.Input) → Subscribers.Demand
func receive(completion: Subscribers.Completion<Self.Failure>)
}
```

定义中 Input 和 Failure 分别表示了订阅者能够接受的事件流数据类型和错误类型。想要订阅某个 Publisher，Subscriber 中的这两个类型必须与 Publisher 的 Output 和 Failure 一致。

Combine 中也定义了几个比较常见的 Subscriber，我们承接上面的按钮的例子来进行说明。在上面，我们通过 scan 和 map，对 buttonClicked 进行了变形，将它从一个不含数据的按钮事件流，转变为了以 String 表示的按钮按下次数的计数。如果我们想要订阅和使用这些值，可以使用 sink：

```
let buttonClicked: AnyPublisher<Void, Never>
buttonClicked
    .scan(0) { value, _ in value + 1 }
    .map { String($0) }
    .sink { print("Button pressed count: \"\($0)\"") }
```

因为 buttonClicked 是一个无限事件流，所以我们在上面只对 output 值进行了打印。
sink 方法完整的函数签名如下：

```
func sink(
    receiveCompletion:
        @escaping ((Subscribers.Completion<Self.Failure>) → Void),
    receiveValue:
        @escaping ((Self.Output) → Void)
) → AnyCancellable
```

你可以同时提供两个闭包，`receiveCompletion` 用来接收 `failure` 或者 `finished` 事件，`receiveValue` 用来接收 `output` 值。

`sink` 可以充当一座桥梁，将响应函数式的 `Publisher` 链式代码，终结并桥接到基于闭包的指令式世界中来。如果你不得不在指令式的世界中进行一些操作，或者只是以学习和验证为目的，那么使用 `sink` 无可厚非。但是如果你是想要让数据继续在 SwiftUI 的声明式的世界中来驱动 UI 的话，另一个 `Subscriber` 可能会更为简洁常用，那就是 `assign`。

和通过 `sink` 提供闭包，可以执行任意操作不同，`assign` 接受一个 `class` 对象以及对象类型上的某个键路径 (`key path`)。每当 `output` 事件到来时，其中包含的值就将被设置到对应的属性上去：

```
class Foo {  
    var bar: String = ""  
}  
  
let foo = Foo()  
  
let buttonClicked: AnyPublisher<Void, Never>  
buttonClicked  
.scan(0) { value, _ in value + 1 }  
.map { String($0) }  
.assign(to: \.bar, on: foo)
```

这样的 `Subscriber` 让我们可以彻底摆脱指令式的写法，直接将事件值“绑定”到具体的属性上。`assign` 方法的具体定义如下，它要求 `keyPath` 满足 `ReferenceWritableKeyPath`：

```
func assign<Root>(  
    keyPath: ReferenceWritableKeyPath<Root>,  
    value: WritableKeyPath<Root, Value>)
```

```
to keyPath: ReferenceWritableKeyPath<Root, Self.Output>,
on object: Root
) → AnyCancellable
```

也就是说，只有那些 class 类型的实例中的属性能被绑定。在 SwiftUI 中，代表 View 对应的模型的 ObservableObject 接口只能由 class 修饰的类型来实现，这也正是 assign 最常用的地方。

其他角色

Publisher, Operator 和 Subscriber 三者组成了从事件发布，变形，到订阅的完整链条。在建立起事件流的响应链后，随着事件发生，app 的状态随之演变，这些是响应式编程处理异步程序的 Combine 框架的基础架构。

除此之外，Combine 框架中还有两个比较重要的概念，那就是 Subject 和 Scheduler。它们和 Publisher 及 Subscriber 一样，都是通过 protocol 的方式来对抽象概念进行描述。本章中我们会简单介绍一些最常用的相关类型。

Subject

Subject 本身也是一个 Publisher：

```
public protocol Subject : AnyObject, Publisher {
    func send(_ value: Self.Output)
    func send(completion: Subscribers.Completion<Self.Failure>)
}
```

从定义可以看到，Subject 暴露了两个 send 方法，外部调用者可以通过这两个方法来主动地发布 output 值、failure 事件或 finished 事件。如果说 sink 提供了由函数响应式向指令式编程转变的路径的话，Subject 则补全了这条通路的另一侧：它让你可以将传统的指令式异步 API 里的事件和信号转换到响应式的世界中去。

Combine 内置提供了两种常用的 Subject 类型，分别是 PassthroughSubject 和 CurrentValueSubject。PassthroughSubject 简单地将 send 接收到的事件转发给下游的其他 Publisher 或 Subscriber：

```
let publisher1 = PassthroughSubject<Int, Never>()
print("开始订阅")
publisher1.sink(
    receiveCompletion: { complete in
        print(complete)
    },
    receiveValue: { value in
        print(value)
    }
)

publisher1.send(1)
publisher1.send(2)
publisher1.send(completion: .finished)

// 输出:
// 开始订阅
// 1
// 2
// finished
```

PassthroughSubject 并不会对接受到的值进行保留，当订阅开始后，它将监听并响应接下来的事件。比如在上面的代码中，订阅开始后直到第一次 publisher.send(1) 发生时，receiveValue 的闭包才会被调用，并接收发送的值。如果我们调整一下

`sink` 订阅的时机，将它延后到 `publisher.send(1)` 之后，那么订阅者将会从 2 的事件开始进行响应：

```
let publisher2 = PassthroughSubject<Int, Never>()

publisher2.send(1)

print("开始订阅")
publisher2.sink(
    receiveCompletion: { complete in
        print(complete)
    },
    receiveValue: { value in
        print(value)
    }
)

publisher2.send(2)
publisher2.send(completion: .finished)

// 输出:
// 开始订阅
// 2
// finished
```

和 `PassthroughSubject` 不同，`CurrentValueSubject` 则会包装和持有一个值，并在设置该值时发送事件并保留新的值。在订阅发生的瞬间，`CurrentValueSubject` 会把当前保存的值发送给订阅者。举例来说，如果把上例中的 `PassthroughSubject` 换成 `CurrentValueSubject`：

```
let publisher3 = CurrentValueSubject<Int, Never>(0)

print("开始订阅")
publisher3.sink(
    receiveCompletion: { complete in
        print(complete)
    },
    receiveValue: { value in
        print(value)
    }
)

publisher3.value = 1
publisher3.value = 2
publisher3.send(completion: .finished)

// 输出:
// 开始订阅
// 0
// 1
// 2
// finished
```

开始订阅后，publisher 里当前存储的 0 被立即发送给了 receiveValue，接下来对 publisher.value 的每次设置也都触发了订阅者的响应。

Scheduler

如果说 Publisher 决定了发布怎样的 (what) 事件流的话，Scheduler 所要解决的就是两个问题：在什么地方 (where)，以及在什么时候 (when) 来发布事件和执行代码。

关于 where

在更新 UI 时，我们需要保证相关操作发生在主线程。因为异步 API 往往涉及到在不同线程间切换，这个问题就显得尤为重要。比如，使用 URLSession 进行网络请求，默认情况下异步回调方法会在后台线程被调用，如果这时候需要根据数据更新 UI，最简单的方式就是将它 Dispatch 到 main queue 中去：

```
URLSession.shared.dataTask(  
    with: URL(string: "https://example.com")!)  
{  
    data, _, _ in  
    if let data = data,  
        let text = String(data: data, encoding: .utf8)  
    {  
        DispatchQueue.main.async {  
            // 在 main queue 中执行 UI 更新  
            textView.text = text  
        }  
    }  
}.resume()
```

异步的响应式编程中也一样，如果前序的 Publisher 是在后台线程进行操作，那么在订阅时，当状态的变化会影响 UI 时，我们需要将接收事件的线程切换到主线程。Combine 里提供了 receive(on:options:) 来让下游在指定的线程中接收事件。比如，对于后台线程的网络请求返回，可以通过这样的方式在 main runloop 中进行处理：

```
URLSession.shared  
.dataTaskPublisher(for: URL(string: "https://example.com")!)  
.compactMap { String(data: $0.data, encoding: .utf8) }  
.receive(on: RunLoop.main)
```

```
.sink(receiveCompletion: { _ in  
  
}, receiveValue: {  
    textView.text = $0  
})
```

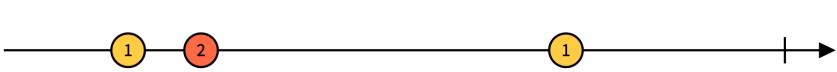
RunLoop 就是一个实现了 Scheduler 协议的类型，它知道要如何执行后续的订阅任务。如果没有 receive(on: RunLoop.main) 的话，sink 的闭包将会在后台线程执行，这在更新 UI 时将带来问题。

关于 when

Scheduler 的另一个重要工作是为事件的发布指定和规划时间。默认情况下，被订阅的 Publisher 将尽可能快地把事件传递给后续的处理流程，不过有些情况下，我们会希望改变事件链的传递时间，比如加入延迟或者等待空闲时再进行传递，这些延时也是由 Scheduler 负责调度的。

比较常见的两种操作是 delay 和 debounce。delay 简单地将所有事件按照一定事件延后。debounce 则是设置了一个计时器，在事件第一次到来时，计时器启动。在计时器有效期间，每次接收到新值，则将计时器时间重置。当且仅当计时窗口中没有新的值到来时，最后一次事件的值才会被当作新的事件发送出去。

它们的时序图分别如下所示：



```
delay(for: .seconds(2), scheduler:  
RunLoop.main)
```

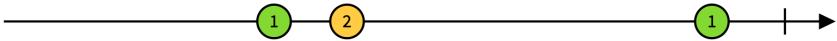


Figure 5.1: `delay` 操作的时序表示



```
debounce(for: .seconds(1), scheduler:  
RunLoop.main)
```

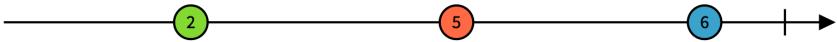


Figure 5.2: `debounce` 操作的时序表示

在 Operator 的介绍中，我们看到过 `map` 和 `scan` 是如何作用于 Publisher 来改变事件值的。这里的 `delay` 和 `debounce` 方法，包括上面的 `receive(on:options:)`，看起来和 `map`, `scan` 这类 Operator 并没有什么不同：它们都是 Publisher 上的扩展方法，并返回一个新的 Publisher。区别在于，`receive(on:options:)`, `delay` 和 `debounce` 所接受的参数包括一个 Scheduler 实例。它们所要解决的不是如何变更事件的值，而是负责更改时间或者线程相关的内容，是调用机制的管理者。

总结

本章中，我们介绍了响应式异步编程的基本概念，以及 Combine 框架中的几个重要角色。Combine 和其他响应式编程框架十分类似：Publisher 负责在源头发布事件，通过若干个 Operator 进行转换，最终得到可供 Subscriber 消化和使用的状态。使用这种统一的抽象方式来处理纷杂的异步操作和状态，可以在很大程度上维持代码的简洁。

想要完成从指令式处理异步 API 的思维模式向响应式的编程方式的转变，最关键的部分在于建立新的习惯，用不一样的思考方式来处理问题。你需要将框架中每个部分的职责明确地理解清楚，尽量保持在响应式的世界中处理问题，并使用组合的方式像搭积木一样构建出稳定的逻辑和数据流。

本章作为概览性的章节，出现了不少新概念。但由于篇幅限制，我们无法在这里完全展开。你在阅读完本章后，可能对如何具体使用这些知识还没有概念，打开代码编辑器也依然不知道该如何下手，这非常正常。在接下来几章中，我们会更详细地结合用例来说明 Combine 的工作流程，以及如何在一个真实的 app 中使用这些工具。不过在此之前，从本章内容中我们希望你至少能对 Combine 框架中每个角色大致的责任范围，以及整体上响应式编程的工作流程有基本认识。你可以通过下面的练习，来检测你是否掌握了这方面的知识。

练习

1. Combine 组件概念

关于 Combine 中 Publisher，Operator 和 Subscriber 的说法，下面哪一项是正确的？不正确的项目的原因是什么？

- a. 它们三者都是由 protocol 所定义的概念，它们结合起来可以组成完整的响应链。
- b. 除了 Combine 框架内置的一些 Publisher 和 Subscriber 以外，我们也可以根据需要自行创建新的 Publisher 和 Subscriber 类型。
- c. 之所以可以通过链式调用的方式使用 Operator 进行数据和事件转换，是因为每个 Operator 操作都返回了一个 Subscriber。
- d. 在 Subscriber 中，我们可以使用 sink 来代替 assign 的工作，反过来也能使用 assign 来实现 sink，它们是等价的。

2. 下面哪一项不是 Combine 框架的特点？

- a. 在编译期间，Combine 为我们提供了强类型保证，Output 和 Input 类型不一致的 Publisher 和 Subscriber 不能一起工作，必须经过变换。
- b. Combine 可以提供统一的异步编程方式，让我们可以忽略掉闭包回调、Delegate 或者 Notification 的区别，专注于处理事件流和数据。
- c. 已经存在 RxSwift 和一些其他类似的框架帮助我们进行异步的响应式编程，Combine 的核心概念和它们十分类似。
- d. Combine 中的核心是对事件流的处理，事件流都有始有终，每次事件发生时通过 output 将数据发布出去，结束时用 failure 或者 finished 事件表示。

3. 关于 PassthroughSubject 和 CurrentValueSubject

关于 PassthroughSubject 和 CurrentValueSubject，思考下面代码的输出分别是什么？尝试实际验证一下你的想法是否正确，并作出解释。

a.

```
let publisher = PassthroughSubject<Int, Never>()
```

```
publisher.send(1)
publisher.send(2)
publisher.send(completion: .finished)

print("开始订阅")
publisher.sink(
    receiveCompletion: { complete in
        print(complete)
    },
    receiveValue: { value in
        print(value)
    }
)

publisher.send(3)
publisher.send(completion: .finished)
```

b.

```
let publisher = CurrentValueSubject<Int, Never>(0)

publisher.value = 1
publisher.value = 2
publisher.send(completion: .finished)

print("开始订阅")
publisher.sink(
    receiveCompletion: { complete in
```

```
        print(complete)
    },
    receiveValue: { value in
        print(value)
    }
)

c.

let publisher = CurrentValueSubject<Int, Never>()

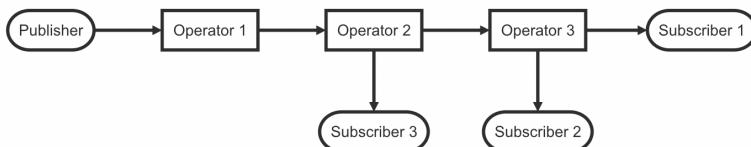
print("开始订阅")
publisher.sink(
    receiveCompletion: { complete in
        print(complete)
    },
    receiveValue: { value in
        print(value)
    }
)

publisher.value = 1
publisher.value = 2
publisher.send(3)
publisher.send(completion: .finished)

print("--- \(publisher.value) ---")
```

4. 下面的线框图中关于 Combine 的工作方式，正确的是哪一个？

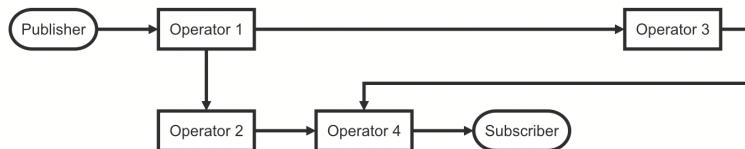
a.



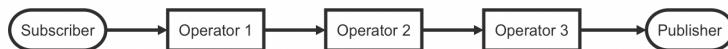
b.



c.



d.



Publisher 和常 见 Operator

6

上一章中，我们已经概括性地涉及了 Combine 框架中各个部件的作用，以及它们是如何协作来处理异步事件的。事件流的源头是 Publisher，之后通过各种 Operator 的处理，所得到的也是 Publisher。最后，Subscriber 是针对某个 Publisher 进行订阅。毫无疑问，Publisher 是 Combine 框架的核心。

本章会着眼于 Publisher 的实际使用。我们将通过一些例子，由浅入深地了解几类常用的 Publisher 和 Operator。通过变换和组合，使用 Combine 框架所提供的内建 Publisher 我们可以进行丰富的逻辑表达以满足绝大部分的要求。

对于刚刚进入响应式编程的开发者来说，在指令式世界和响应式世界之间的切换，是需要面对的第一个重要课题。在本章后半部分，我们会对 Subject 做进一步的补充性说明。

在本章最后，我们还会探究 Publisher 协议和扩展的内容，来看看 Publisher 背后到底是如何工作的。

准备

在开始本章内容之前，强烈建议你确认已经完成了上一章的练习题。在本章中，我们会在 Xcode 的 Playground 里进行编码。你可以在本书附带的源码中找到 CombineExample-Start.playground 文件，其中包含了一些辅助类的代码，用来帮助打印和确认事件流。在 Playground 的 Sources/Utils.swift 文件里 (使用 Command + 1 可以打开项目导航的文件面板)，我们能够看到这些辅助方法和定义：

```
public enum SampleError: Error {
    case sampleError
}

public func check<P: Publisher>(
    _ title: String,
```

```
publisher: () -> P
) -> AnyCancellable
{
    print("----- \u2022(title) -----")
    defer { print("") }
    return publisher()
        .print()
        .sink(
            receiveCompletion: { _ in },
            receiveValue: { _ in }
        )
}
```

我们使用 `check` 来检查某个 Publisher，它将打印输入的 title 信息，然后通过 sink 来订阅这个 Publisher，这可以让输入的 publisher 开始发送和产生值。在订阅之前，为 publisher 添加了一个 `.print()`，这是一个 Combine 内建的 Operator，它会在该 publisher 发送事件时，将具体事件的内容打印到控制台，方便调试观察。

我们先来看看 `check` 的工作方式。打开 Playground 的 Basic 页面，添加如下代码并执行：

```
check("Empty") {
    Empty<Int, SampleError>()
}

// 输出:
// ----- Empty -----
// receive subscription: (Empty)
// request unlimited
// receive finished
```

本书示例代码中的 CombineExample-Start.playground 文件为了提高执行速度，选用了 macOS 作为目标平台。如果你还在使用 macOS 10.15 之前的版本，会出现编译或者运行错误，这是由于旧版本的 macOS 上没有 Combine 框架所导致的。这种情况下，你可以在 Playground 页面的 Inspector 边栏中将 Playground Settings 里的 Platform 选项从 macOS 修改为 iOS。(选中任意一个 Playground 的实际代码页面，使用 Option + Command + 1 可以打开 Inspector。)

Empty 是一个最简单的发布者，它只在被订阅的时候发布一个完成事件 (receive finished)。这个 publisher 不会输出任何的 output 值，只能用于表示某个事件已经发生。

如果我们想要 publisher 在完成之前发出一个值的话，可以使用 Just，它表示一个单一的值，在被订阅后，这个值会被发送出去，紧接着是 finished：

```
check("Just") {
    Just(1)
}

// 输出:
// ----- Just -----
// receive subscription: (Just)
// request unlimited
// receive value: (1)
// receive finished
```

在 Just 的输出中，和 Empty 相比，多了一个 “receive value: (1)” 的 output 事件。除了 output 和 finished 以外，我们还看到两者都有两行额外的内容：“receive subscription” 和 “request unlimited”。Publisher 在接收到订阅，并且接受到请求

要求提供若干个事件后，才开始对外发布事件（在我们的 Playground 例子中，check 函数中的 sink 负责订阅和请求）。接受订阅和接受请求，也是 Publisher 生命周期的一部分。Publisher 和 Subscriber 的调用关系如下：



在 Just 的例子里，四个输出分别对应上图的 2, 3, 4, 6 四个事件。如果你在 Playground 中能够成功运行这些例子并输出正确结果的话，你应该已经准备好对于 Publisher 更深入的探索了。关于上图中的各种事件和对应方法，我们会在本书之后创建自定义 Publisher 和 Subscriber 的部分看到更详细的解释，现在先让我们跳过这些内容，来看看 Publisher 和 Operator 具体的工作方式。

基础 Publisher 和 Operator

本节中，我们将介绍几种基础的内建 Publisher，这可以让你对 Publisher 的特性有一个直观的认识。

序列 Publisher 及其操作

Just 涉及的是单个值，如果我们对一连串的值感兴趣的话，可以使用的是 Publishers.Sequence。顾名思义，Publishers.Sequence 接受一个 Sequence：它可以是一个数组，也可以是一个 Range。在被订阅时，Sequence 中的元素被逐个发送出来：

```
check("Sequence") {
    Publishers.Sequence<[Int], Never>(sequence: [1, 2, 3])
}

// 输出:
// ----- Sequence -----
// receive subscription: ([1, 2, 3])
// request unlimited
// receive value: (1)
// receive value: (2)
// receive value: (3)
// receive finished
```

为了方便使用，Sequence 提供了一个辅助属性来从一个序列中获取 Publisher。上面的代码可以等效写为：

```
check("Array") {
    [1, 2, 3].publisher
}
```

Apple 为 Swift 标准库和 Foundation 中的很多类型都添加了类似的获取 CombinePublisher 的简便方法，在后续我们会看到更多类似的例子。

元素变形

在 Publisher 上使用 Operator，可以对事件中的数据进行变形。对于普通 Array 来说，我们可以使用 map 对其中的元素进行变形，比如：

```
[1, 2, 3].map { $0 * 2 }  
// [2, 4, 6]
```

在 Publisher 上也存在一个 map 函数，我们可以通过类似的方式，对 output 的元素进行变形：

```
check("Map") {  
    // 注意我们是在 `Publisher` 上调用了 `map`  
  
    [1,2,3]  
        .publisher  
        .map { $0 * 2 }  
    }  
  
    // 输出：  
    // ----- Map -----  
    // receive subscription: ([2, 4, 6])  
    // request unlimited  
    // receive value: (2)  
    // receive value: (4)  
    // receive value: (6)  
    // receive finished
```

对 Publisher 进行 map 操作的结果看起来和 Array 的 map 很类似，它们都具有“对元素进行某种方式的变形操作”这一共性，因此它们使用了同样的方法名。Array 存储的是当前存在的一组元素，对它们进行 map 将**同步地**把这些元素进行变形。而

Publisher “存储的”是未来的一组元素，map 操作则是**异步地**在未来 output 事件发生时再进行变形。

如果你注意到上面输出的 receive subscription: ([2, 4, 6])，可能会有疑惑：map 操作到底是在何时发生的。在本章最后，我们会介绍操作符熔合的内容。确实，实际上对于 Array Publisher 来说，map 并不是在“未来”才进行变形。所以严格来说，这里在 Array Publisher 语境下，“在未来 output 事件发生时再进行变形”这一说法并不准确。但是对于一般性的 Publisher，这个结论是正确的。

虽然在上面的例子里，我们是对一个 Publishers.Sequence 进行 map 操作，但是为了避免混淆，我想要特别做一点说明：Publisher 的 map 操作和这个 Publisher 是否是一个 Sequence 的 publisher 无关，对于任意的 Publisher，它都是若干个未来可能被发布的值，我们都可以使用 map 来对它将会发布的值进行变换：

```
check("Map Just") {  
    Just(5)  
    .map { $0 * 2 }  
}  
  
// 输出：  
// ----- Map Just -----  
// receive subscription: (Just)  
// request unlimited  
// receive value: (10)  
// receive finished
```

聪明的读者应该已经想到了，既然 Publisher 实际上是事件的序列，那么那些适用于普通序列 (Sequence，或者说 Array) 的各类函数式方法，比如 filter，contains 等，是不是都适用于 Publisher 呢？答案是肯定的：大部分对于 Sequence 的操作，都可

以在 Publisher 中找到对应的版本，我们在本章最后的练习中会有机会进行一些实践。现在先让我们来看几个 Publisher 特有的操作。

reduce 和 scan

Array 的 reduce 方法可以将数组中的元素按照某种规则进行合并，并得到一个最终的结果：

```
[1,2,3,4,5].reduce(0, +)  
// 15
```

上面的调用中，以 0 为初始输入值，每次处理输入数组中的一个元素，将它与初值相加，并把结果作为新的输入值，去处理下一个元素。输入值的变化历经“1, 3, 6, 10, 15”这几个值，最终输出结果 15。

Publisher 的 reduce 和它很类似：

```
check("Reduce") {  
    [1,2,3,4,5].publisher.reduce(0, +)  
}
```

```
// 输出:  
// ----- Reduce -----  
// receive subscription: (Once)  
// request unlimited  
// receive value: (15)  
// receive finished
```

[1,2,3,4,5].publisher 将发布五个 output 值，当序列中值耗尽时，它将发布 finished。而经过 reduce 变形后，新的 Publisher 只会在接到上游发出的 finished

事件后，才会将 reduce 后的结果发布出来。而紧接这个结果，则是新的 reduce Publisher 的结束事件。

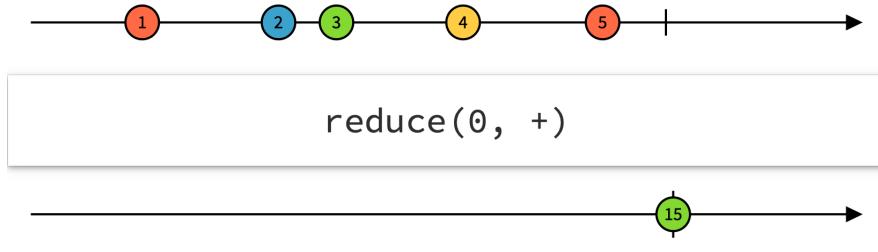


Figure 6.1: Reduce Publisher

有些情况下，除了最终的结果，我们也有可能会想要把中途每一步的过程保存下来。在 Array 中，这种操作一般叫做 scan。这个方法在标准库中并不存在，不过我们可以很容易地添加一个：

```
extension Sequence {  
    public func scan<ResultElement>(  
        _ initial: ResultElement,  
        _ nextPartialResult: (ResultElement, Element) -> ResultElement  
    ) -> [ResultElement] {  
        var result: [ResultElement] = []  
        for x in self {  
            result.append(nextPartialResult(result.last ?? initial, x))  
        }  
        return result  
    }  
}
```

调用该方法的方式和 reduce 几乎相同：

```
[1,2,3,4,5].scan(0, +)  
// [1, 3, 6, 10, 15]
```

我们在上一章中，已经看到如何使用 scan 将点击按钮的事件转变为累积的点击次数。在 Publisher 中，这类一边进行重复操作，一边将每一步中间状态发送出去的场景十分普遍。因此，Combine 内置提供了 scan 这个 Operator。对 [1,2,3,4,5] 用累加方式进行 scan，输出如下：

```
check("Scan") {  
    [1,2,3,4,5].publisher.scan(0, +)  
}  
  
// ----- Scan -----  
// receive subscription: ([1, 3, 6, 10, 15])  
// request unlimited  
// receive value: (1)  
// receive value: (3)  
// receive value: (6)  
// receive value: (10)  
// receive value: (15)  
// receive finished
```

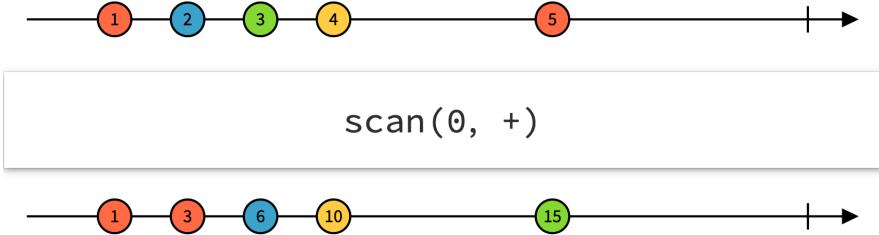


Figure 6.2: Scan Publisher

实践中，`scan` 一个最常见的使用场景是在某个下载任务执行期间，接受 `URLSession` 的数据回调，将已接收到的数据量做累加来提供一个下载进度条的界面。

`compactMap` 和 `flatMap`

`map` 在函数式编程中的地位十分重要，以至于它还有一些其他的变形。其中 `compactMap` 和 `flatMap` 是比较重要的两个。

`compactMap` 比较简单，它的作用是将 `map` 结果中那些 `nil` 的元素去除掉，这个操作通常会“压缩”结果，让其中的元素数减少，这也正是其名字中 **compact** 的来源：

```
check("Compact Map") {
    ["1", "2", "3", "cat", "5"]
        .publisher
        .compactMap { Int($0) }
}

// 输出:
// ----- Compact Map -----
// receive subscription: ([1, 2, 3, 5])
```

```
// request unlimited
// receive value: (1)
// receive value: (2)
// receive value: (3)
// receive value: (5)
// receive finished
```

输入的 "cat" 在使用 Int.init 转换时，由于其不是有效整型数字，将会得到 nil。使用 compactMap 可以将这个 “例外” 排除在最终结果之外。在语义上，compactMap 和下面接连使用 filter 和 map 的方式是等效的：

```
check("Compact Map By Filter") {
    ["1", "2", "3", "cat", "5"]
        .publisher
        .map { Int($0) }
        .filter { $0 != nil }
        .map { $0! }
}
```

相比于 compactMap 的简单明了，flatMap 则要复杂许多。map 及 compactMap 的闭包返回值是单个的 Output 值。而与它们不同，flatMap 的变形闭包里需要返回一个 Publisher。也就是说，flatMap 将会涉及两个 Publisher：一个是 flatMap 操作本身所作用的外层 Publisher，一个是 flatMap 所接受的变形闭包中返回的内层 Publisher。flatMap 将外层 Publisher 发出的事件中的值传递给内层 Publisher，然后汇总内层 Publisher 给出的事件输出，作为最终变形后的结果。

这么说可能有些抽象，我们通过几个例子来进行解释。考虑下面的代码：

```
check("Flat Map 1") {
    [[1, 2, 3], ["a", "b", "c"]]
}
```

```
.publisher  
.flatMap {  
    $0.publisher  
}  
}
```

组成外层 Publisher 的是一个数组的数组，它含有两个元素，分别是 [1, 2, 3] 和 ["a", "b", "c"]。在被订阅后，这个外层 Publisher 会发送两个 Output 事件 (两个事件的值分别是 [1, 2, 3] 和 ["a", "b", "c"])，每个事件的值被 flatMap 传递到内层，并通过 \$0.publisher 生成新的 Publisher 并返回。内层 Publisher 实际上是 [1, 2, 3].publisher 和 ["a", "b", "c"].publisher，它们发送的值将被作为 flatMap 的结果，被“展平”(flatten) 后发送出去。因此，上面代码的输出结果是：

```
// ----- Flat Map 1 -----  
  
// receive subscription: (FlatMap)  
  
// request unlimited  
  
// receive value: (1)  
// receive value: (2)  
// receive value: (3)  
// receive value: (a)  
// receive value: (b)  
// receive value: (c)  
// receive finished
```

通过这种由外层 Publisher 提供数据源，内层 Publisher 提供控制方式的方法，可以将两个甚至是多个 Publisher 的行为关联起来，形成具有更复杂逻辑的全新 Publisher。

当内层的 Publisher 的逻辑相对复杂时，flatMap 理解起来也会变得复杂。例如下面的例子中：

```
check("Flat Map 2") {
    ["A", "B", "C"]
        .publisher
        .flatMap { letter in
            [1, 2, 3]
                .publisher
                .map { "\($letter)\($0)" }
        }
    }
}

// 输出:
// ----- Flat Map 2 -----
// receive subscription: (FlatMap)
// request unlimited
// receive value: (A1)
// receive value: (A2)
// receive value: (A3)
// receive value: (B1)
// receive value: (B2)
// receive value: (B3)
// receive value: (C1)
// receive value: (C2)
// receive value: (C3)
// receive finished
```

外层 ["A", "B", "C"] 里的每个元素，作为 flatMap 变形闭包的输入，参与到了内层 Publisher 的 map 计算。内层 Publisher 逐次使用 [1, 2, 3] 里的元素，和输入进来的 letter (也就是 “A”, “B” 和 “C”) 进行拼接后作为新事件发出。

在上一章介绍 Publisher 和 Subscriber 的时候，我们已经提到过

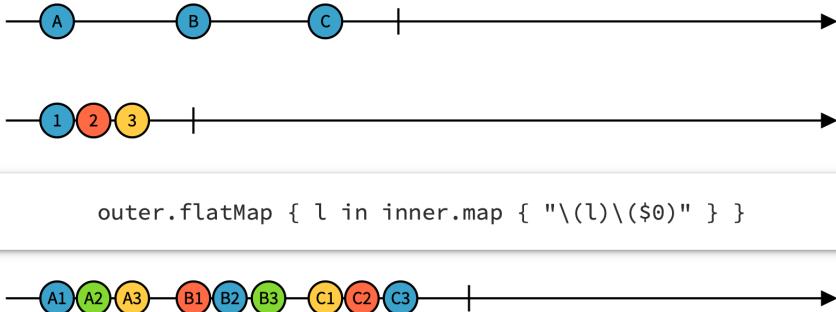


Figure 6.3: flatMap 作用于 Publisher

整个过程看起来和 for 循环内嵌套另一个 for 循环很相似，不过我们需要记住的是，在实际的 Publisher 中，这些事件的发生可能是异步的，它们会带有时序信息。在外层 Publisher 发出事件的时序上出现不同时，flatMap 的结果可能也会有很大不同，并且变得更加复杂。我们会在练习中看到一个这方面的例子并强化你的理解。

一开始你可能会不习惯这种 Publisher 嵌套来构建逻辑的方式，但是很快你将会看到，这类操作在异步编程中非常常见。将多个 Publisher 进行合并，形成一个新的 Publisher 的操作，其核心目的在于“降维”。Publisher 的核心是事件流，每个 Publisher 都维护了一个独立的事件流。在真实世界里的情况，往往会有多个 Publisher 协同工作。每多一个事件流，就会让我们的思考维度加一，如果这些事件流之间还有状态共享关系的话，它们之间的关系会变得非常复杂。通过将多个事件流合并和降维，我们可以明确地保持只处理单个事件流，而忽略掉不必要的实现细节，这与声明式编程的思想不谋而合，也是这套方式能简化程序状态的核心原因。

包括我们这里看到的 flatMap，以及马上会在本章稍后部分看到的 zip，combineLatest 都属于这类“降维”操作。

比如用户通过键盘在搜索栏里输入了新的搜索关键字，触发了文字变化的新事件；这些文字被发送给服务器，变成了网络请求的事件；最后，网络返回的数据被解码变形成为对应的 model 类型，再去改变 UI 里的搜索结果文本。每一个事件流都有它们各自的输入和输出，但是对于开发者来说，我们其实只关心“用户输入，导致 UI 中搜索结果变更”这一个事件流。flatMap 的目的，正是将这类多个异步操作展平为单个事件流，减轻开发者的心智负担。

removeDuplicates

在处理“序列”上，相较于 Array 所提供的方法，Publisher 还有一个很方便的操作，removeDuplicates。在一个不断发送 Output 值的事件流中，可能会存在连续多次发送相同事件的情况。有时，我们会想要过滤掉这些重复的事件，进而避免无谓的额外开销。移除连续出现的重复事件值，正是 removeDuplicates 所提供的：

```
check("Remove Duplicates") {
    ["S", "Sw", "Sw", "Sw", "Swi",
     "Swif", "Swift", "Swift", "Swif"]
        .publisher
        .removeDuplicates()
}

// 输出:
// ----- Remove Duplicates -----
// receive subscription: (["S", "Sw", "Swi", "Swif", "Swift", "Swif"])
// request unlimited
// receive value: (S)
```

```
// receive value: (Sw)
// receive value: (Swi)
// receive value: (Swif)
// receive value: (Swift)
// receive value: (Swif)
// receive finished
```

上例中，“Sw”连续出现了三次，“Swift”出现了两次，而经过移除操作后，我们得到的是一系列没有重复的字符串事件。`removeDuplicates`经常被用来减少那些非常消耗资源的操作，比如由事件触发造成的网络请求或者图片渲染。如果当作为源头的数据没有改变时，所预期得到的结果也不会变化的话，那么就没有必要去重复这样操作。在源头将重复的事件移除，可以让下游的事件流也变得简单。

到现在为止，我们已经介绍了几个相对重要的，且在普通 Array 中不存在的操作。Publisher 的事件序列在普通元素序列的基础上增加了一个时间维度，但它们在“有序元素”这一特性上是一致的，这让大多数针对普通序列的操作可以等价对应到 Publisher 的事件序列上。

由于篇幅有限，还有部分的序列操作我们没有提及。练习中会涉及一些其他的常见序列操作，你可以结合示例进行练习和学习。Publisher 中能进行的所有变形操作，都在 Apple 的 [Publisher 文档](#)中进行了记载，必要时你也可以一一对照。

本章中我们都使用了 `Array.publisher` 的方法，获取了“同步”的 Publisher 来作为示例，这是一种为了进行简单例证而进行的假设，实际情况中大多数事件流都会伴随时间维度。在下一章里，我们会结合时序来对 Publisher 和 Operator 进行更深入的探索。不过在此之前，我们会先把 Operator 的话题暂时搁置，来看看异步编程中另一个很重要的话题，错误处理。

错误处理

转换错误类型

在上面的基于序列的例子中，Publisher 总是在发布一些值后，以 .finished 事件作为正常结束。但实际上，Publisher 的结束事件有两种可能：代表正常完成的 .finished 和代表发生了某个错误的 .failure，两者都表示 Publisher 不再会有新的事件发出：

```
extension Subscribers {  
    public enum Completion<Failure> where Failure : Error {  
        case finished  
        case failure(Failure)  
    }  
}
```

Combine 中提供了 Fail 这个内建的基础 Publisher，它所做的事情就是在被订阅时发送一个错误事件：

```
check("Fail") {  
    Fail<Int, SampleError>(error: .sampleError)  
}  
  
// 输出：  
// ----- Fail -----  
// receive subscription: (Empty)  
// request unlimited  
// receive error: (sampleError)
```

在上一章中，我们提到过 Subscriber 在订阅上游 Publisher 时，不仅需要保证 Publisher.Output 的类型和 Subscriber.Input 的类型一致，也要保证两者所接受的 Failure 也具有相同类型。

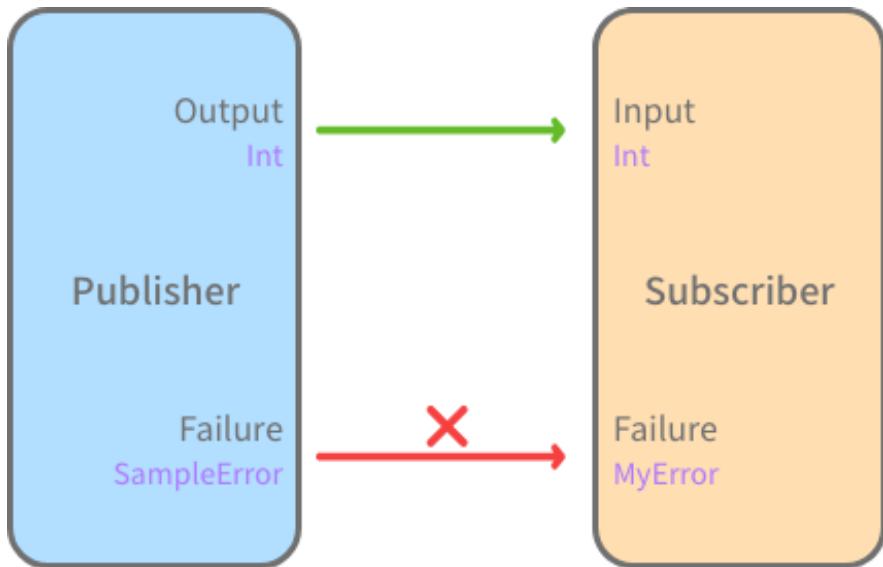


Figure 6.4: Publisher 的 Failure 类型与 Subscriber 不一致

如果 Publisher 在出错时发送的是 SampleError，但订阅方声明只接受 MyError 时，就算实际上 Publisher 只发出 Output 值而从不会发出 Failure 值，我们也无法使用这个 Subscriber 去接收一个类型不符的 Publisher 的事件。

在这种情况下，我们可以通过使用 mapError 来将 Publisher 的 Failure 转换成 Subscriber 所需要的 Failure 类型：

```
check("Map Error") {  
    Fail<Int, SampleError>(error: .sampleError)
```

```
.mapError { _ in MyError.myError }  
}
```

```
// ----- Map Error -----  
// receive subscription: (Empty)  
// request unlimited  
// receive error: (myError)
```

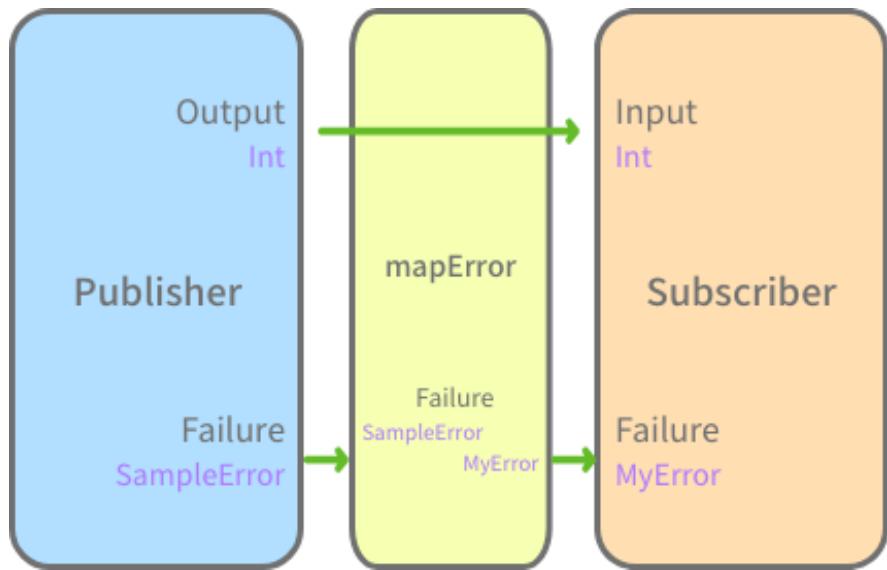


Figure 6.5: 使用 mapError 转换错误类型

map 对 Output 进行转换，mapError 对 Failure 进行转换，就是这么简单。

抛出错误

有时候，Operator 在对上游 Output 的数据进行处理时，可能会遇到发生错误的情况。比如说尝试将一个字符串转换为 Int 时，我们并不总是能得到结果 Int。有时候我们可以用上面提到的 compactMap 来把这种结果过滤掉，但是有些时候，我们会希望不要放过这种“例外”，而是让事件流以明确的错误作为结束，来表明输入数据出现了问题。

Combine 为 Publisher 的 map 操作提供了一个可以抛出错误的版本，tryMap。使用 tryMap 我们就可以将这类处理数据时发生的错误转变为标志事件流失败的结束事件：

```
check("Throw") {
    ["1", "2", "Swift", "4"].publisher
        .tryMap { s → Int in
            guard let value = Int(s) else {
                throw MyError.myError
            }
            return value
        }
}

// 输出:
// ----- Throw -----
// receive subscription: (TryMap)
// request unlimited
// receive value: (1)
// receive value: (2)
// receive error: (myError)
```

上面的例子中，“Swift”这个字符串是无法被转换为 Int 值的，当问题发生时，我们抛出了一个自定义的 myError 错误。这导致整个事件流以错误结果终止，接下来的“4”也不再会被计算和发布。

除了 tryMap 以外，Combine 中还有很多类似的以 try 开头的 Operator，比如 tryScan，tryFilter，tryReduce 等等。当你有需求在数据转换或者处理时，将事件流以错误进行终止，都可以使用对应操作的 try 版本来进行抛出，并在订阅者一侧接收到对应的错误事件。

从错误中恢复

不管是什么语言或者框架，错误处理总是很考验细节的地方。大多数情况下我们可能会以某种形式把错误反馈给用户，用弹框或者文本告诉他们某个地方可能出了问题。不过有些时候，我们也可能会选择使用默认值来让事件流从错误中“恢复”。

在 Combine 里，有一些 Operator 是专门帮助事件流从错误中恢复的，最简单的是 replaceError，它会把错误替换成一个给定的值，并且立即发送 finished 事件：

```
check("Replace Error") {
    ["1", "2", "Swift", "4"].publisher
        .tryMap { s → Int in
            guard let value = Int(s) else {
                throw MyError.myError
            }
            return value
        }
        .replaceError(with: -1)
}

// 输出:
```

```
// ----- Replace Error -----
// receive subscription: (ReplaceError)
// request unlimited
// receive value: (1)
// receive value: (2)
// receive value: (-1)
// receive finished
```

如果我们想要在事件流以错误结束时被转为一个默认值的话，`replaceError` 就会很有用。`replaceError` 会将 Publisher 的 Failure 类型抹为 Never，这正是我们使用 `assign` 来将 Publisher 绑定到 UI 上时所需要的 Failure 类型。我们可以用 `replaceError` 来提供这样一个在出现错误时应该显示的默认值。

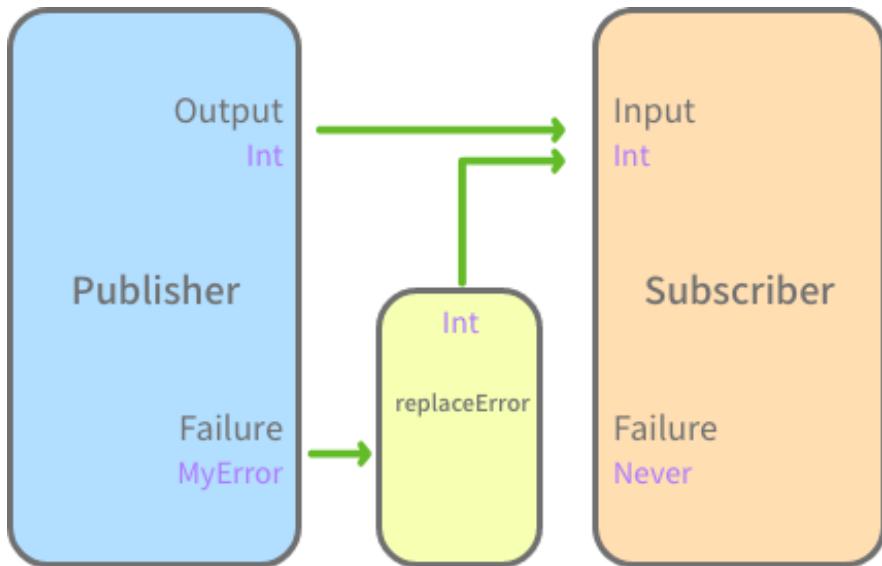


Figure 6.6: `replaceError` 操作

`replaceError` 在错误时接受单个值，另一个操作 `catch` 则略有不同，它接受的是一个新的 Publisher，当上游 Publisher 发生错误时，`catch` 操作会使用新的 Publisher 来把原来的 Publisher 替换掉。举个例子：

```
check("Catch with Just") {
    ["1", "2", "Swift", "4"].publisher
        .tryMap { s → Int in
            guard let value = Int(s) else {
                throw MyError.myError
            }
            return value
        }
        .catch { _ in Just(-1) }
}
```

```
// ----- Catch with Just -----
// receive subscription: (Catch)
// request unlimited
// receive value: (1)
// receive value: (2)
// receive value: (-1)
// receive finished
```

看上去输出和上面的 `replaceError` 没有区别，但是记住在 `catch` 的闭包中，我们返回的是 `Just(-1)` 这个 Publisher，而不仅仅只是 `Int` 的 -1。实际上，任何满足 `Output == Int` 和 `Failure == Never` 的 Publisher 都可以作为 `catch` 的闭包被返回，并替代原来的 Publisher：

```
check("Catch with Another Publisher") {
    ["1", "2", "Swift", "4"].publisher
```

```
.tryMap { s → Int in
    guard let value = Int(s) else {
        throw MyError.myError
    }
    return value
}
.catch { _ in [-1, -2, -3].publisher }
```

```
// 输出:
// ----- Catch with Another Publisher -----
// receive subscription: (Catch)
// request unlimited
// receive value: (1)
// receive value: (2)
// receive value: (-1)
// receive value: (-2)
// receive value: (-3)
// receive finished
```

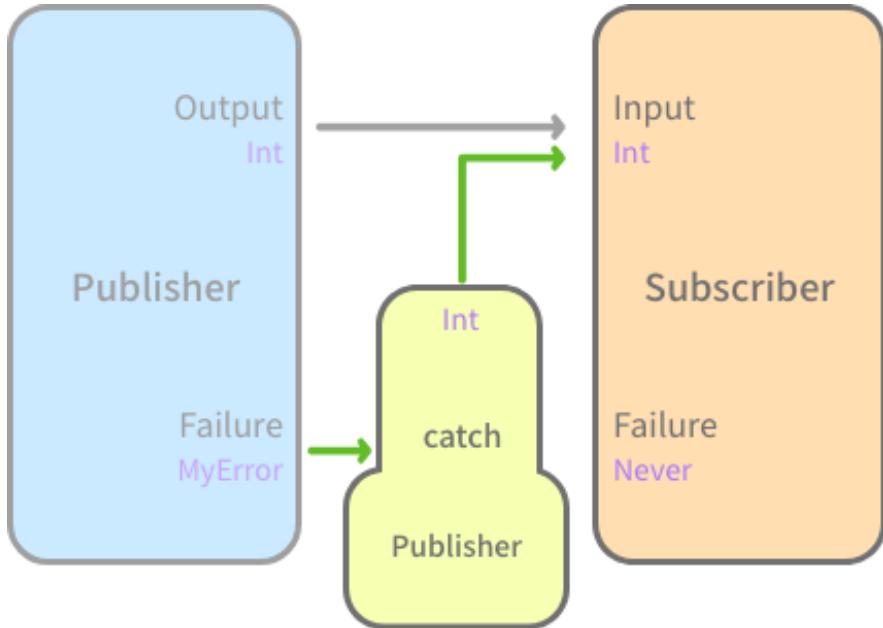


Figure 6.7: 使用 Publisher 的 catch 操作

注意在上图中，当错误发生后，原本的 Publisher 事件流将被中断，取而代之，则是由 catch 所提供的事件流继续向后续的 Operator 及 Subscriber 发送事件。原来 Publisher 中的最后一个元素 “4”，将没有机会到达。

如果我们将 (由 ["1", "2", "Swift", "4"] 构成) 原 Publisher 看作是用户输入，将结果的 Int 看作是最后输出，那么像上面那样的方式使用 replaceError 或者 catch 的话，一旦用户输入了不能转为 Int 的非法值 (如 “Swift”)，整个结果将永远停在我们给定的默认恢复值上，接下来的任意用户输入都将被完全忽略。这往往不是我们想要的结果，一般情况下，我们会想要后续的用户输入也能继续驱动输出，这时候我们可以靠组合一些 Operator 来完成所需的逻辑：

```
check("Catch and Continue") {
```

```
[ "1", "2", "Swift", "4" ].publisher
    .flatMap { s in
        return Just(s)
    }
    .tryMap { s → Int in
        guard let value = Int(s) else {
            throw MyError.myError
        }
        return value
    }
    .catch { _ in Just(-1) }
}
}

// 输出:
// ----- Catch and Continue -----
// receive subscription: (FlatMap)
// request unlimited
// receive value: (1)
// receive value: (2)
// receive value: (-1)
// receive value: (4)
// receive finished
```

这样一来，即使发生了“Swift”无法转换为 Int 的错误，最终的“4”依然能被正确处理和出现在输出。在响应式异步编程中，这种使用 flatMap 进行“包装”的手法是很常见的。如果你不能理解上面的代码为什么工作，我建议可以返回去阅读 flatMap 的相关内容，以及对 tryMap 和 catch 的解释。另外，在每个 Operator 之后，你都可以加上 print("TAG") 来将对应 Publisher 发送的事件添加一个标识 Tag 后打印出来，这可以帮助你“逐步”理解到底发生了什么：

```
check("Catch and Continue") {
    ["1", "2", "Swift", "4"].publisher
        .print("[ Original ]")
        .flatMap { s in
            return Just(s)
        }
        .tryMap { s → Int in
            guard let value = Int(s) else {
                throw MyError.myError
            }
            return value
        }
        .print("[ TryMap ]")
        .catch { _ in
            Just(-1).print("[ Just ]") }
        .print("[ Catch ]")
    }
}
```

Publisher 的类型系统

嵌套的泛型类型和类型抹消

从上一章开始，我们一直都在谈论 Combine 中最重要的两个角色：Publisher 和 Operator，也在不断强调 Combine 编程中开发者的核心工作就是组合各种 Operator 来生成满足最终逻辑的 Publisher。但是，如果你翻阅 Combine 框架的文档，你只能找到 Publisher，Subscriber 或者是 Subject 的定义，却没有哪个类型或者协议叫做 Operator。那我们一直提到的 Operator 到底是什么呢？

Operator 其实是 Publisher 的 extension 中所提供的一些方法，比如 flatMap 或者 map，它们都作用于 Publisher。如果你注意观察，会发现这些方法返回的也是 Publisher，不过类型不尽相同：

```
let p1 = [[1, 2, 3], [4, 5, 6]]  
.publisher  
.flatMap { $0.publisher }  
  
// p1 : Publishers.FlatMap<  
//     Publishers.Sequence<[Int], Never>,  
//     Publishers.Sequence<[[Int]], Never>  
// >  
  
let p2 = p1.map { $0 * 2 }  
// p2 : Publishers.Map<  
//     Publishers.FlatMap<  
//     Publishers.Sequence<[Int], Never>,  
//     Publishers.Sequence<[[Int]], Never>  
//     >,  
//     Int  
// >
```

上面两个操作返回的结果类型分别是 Publishers.FlatMap 和 Publishers.Map，两者都满足 Publisher 协议。我们其实完全可以通过调用它们的初始化方法来构筑同样的 Publisher。比如下面的例子中，p1_，p2_ 分别和 p1，p2 等效：

```
let p1_ = Publishers.FlatMap(  
    upstream: [[1, 2, 3], [4, 5, 6]].publisher,  
    maxPublishers: .unlimited)  
{
```

```
$0.publisher  
}  
  
let p2_ = Publishers.Map(upstream: p1_) { $0 * 2 }
```

flatMap, map, 以及其他各种我们称之为 Operator 的方法，其实都只是对应的 Publishers 类型初始化方法的简便调用方式。除此之外，它们作用于 Publisher，也返回 Publisher 的特性，让我们最终能够通过简洁易读的链式连续调用来构建发布者转换逻辑。

Publisher 中存在 Output 和 Failure 两个泛型的关联类型，多次的嵌套会让类型变得非常复杂。以上面例子中的 p2 类型为例：在外层是一个 Publishers.Map，内一层嵌套了 Publishers.FlatMap，而再其中则又嵌套了两个代表 flatMap 操作待展平的 Publishers.Sequence。仅仅是两三次 Operator 操作，就足以让得到的 Publisher 类型非常复杂，难以阅读，而现实中往往会经过更多的操作才能得到合适的 Publisher。

但就订阅者来说，只需要关心 Publisher 的 Output 和 Failure 两个类型就能顺利订阅，它们并不需要具体知道这个 Publisher 是如何得到、如何嵌套的。Combine 提供了 `eraseToAnyPublisher` 来帮助我们对复杂类型的 Publisher 进行类型抹消，这个方法返回一个 AnyPublisher：

```
let p3 = p2.eraseToAnyPublisher()  
// p3: AnyPublisher<Int, Never>
```

Combine 中的其他角色也大都提供了类似的抹消后的类型，比如 AnySubscriber 和 AnySubject 等。在大多数情况下我们都只会关注某个部件所扮演的角色，也即，它到底是一个 Publisher 还是一个 Subscriber。一般我们并不关心具体的类型，因为对 Publisher 的变形往往都伴随着类型的变化。通过类型抹消，可以让事件的传递和订阅操作变得更加简单，对外的 API 也更加稳定。

操作符熔合

可能你已经发现了，有时候 map 操作的返回结果的类型并不是 Publishers.Map，比如下面的两个例子：

```
[1, 2, 3].publisher.map { $0 * 2 }  
// Publishers.Sequence<[Int], Never>  
  
Just(10).map { String($0) }  
// Just<String>
```

这是由于 Publishers.Sequence 和 Just 在各自的扩展中对默认的 Publisher 的 map 操作进行了重写。由于 Publishers.Sequence 和 Just 这样的类型在编译期间我们就能确定它们在被订阅时就会同步地发送所有事件，所以可以将 map 的操作直接作用在输入上，而不需要等待每次事件发生时再去操作。这种将操作符的作用时机提前到创建 Publisher 时的方式，被称为操作符熔合 (operator fusion)。我们完全可以想象 Publishers.Sequence 上 map 操作的实现：

```
extension Publishers.Sequence {  
    public func map<T>(_ transform: (Elements.Element) → T)  
        → Publishers.Sequence<[T], Failure>  
    {  
        return Publishers.Sequence(sequence: sequence.map(transform))  
    }  
}
```

这样做可以避免 transform 的 @escaping 的要求，避免存储这个变形闭包，让所得得到的 Publisher 更加高效，同时也让后续 Publisher 变形类型能简单一些。这些优点在多次变形的时候尤为突出。

总结

在本章中，我们从最简单的 Just 开始，以 Publishers.Sequence 为例子，着重介绍了一些常见 Operator 操作。

在学习 Combine 框架时，只有确实理解了每个 Operator 的作用和行为特点，才能进一步理解各种 Operator 在组合后所形成的逻辑。最终，依靠这些小块知识和常见模式，按照需求写出合适的组合逻辑。毫无疑问，如果没有各个 Operator 的知识基石，是不可能构建出一套异步响应式的逻辑大厦的。

本章中的 Publishers.Sequence 在发送数据时并没有时序上的考虑，所有的事件都是即时发出的。但你需要铭记于心，这往往并不是实际情况：在异步编程中，更多的时候事件是随着时间流逝而发生的。不过本章中的 Operator 们并没有局限在同步的 Publishers.Sequence 上，它们对于任意的 Publisher 都是广泛适用的。

在下一章中，我们将把注意力放到“时序”上，来看看真实情况的 Publisher 和 Operator 是什么样子的，在那里，我们会进一步涉及到几个关于时序的 Operator。另一方面，在很多时候，程序中会有在函数响应式编程和传统指令式编程之间进行切换的需要。这将依赖于 Subject 的使用以及那些 Foundation 框架为我们提供的 Publisher。而在订阅发生时，内存管理也是一个重要的话题。

不过，在开始下一章之前，请检查一下练习部分的题目。除了确认你对本章内容完全理解之外，练习中也包含了一些我们在本章中没有介绍的 Operator。

练习

1. filter 和 contains

我们已经看到，对于任意 Publisher，我们总是可以把它想象为一个“异步数组”，它表示随着时间发生的一系列值，因此 Sequence 里的 map 等操作对它都适用。除了本章介绍到的操作之外，序列常用的操作还有很多，比如 filter 利用闭包来将不满足条件的元素过滤掉：

```
check("Filter") {
    [1,2,3,4,5].publisher.filter { $0 % 2 == 0 }
}

// ----- Filter -----
// receive subscription: ([2, 4])
// request unlimited
// receive value: (2)
// receive value: (4)
// receive finished
```

或者使用 contains：

```
check("Contains") {
    [1,2,3,4,5].publisher
        .print("[Original]")
        .contains(3)
}

// 输出略
```

在 `contains` 中，我们为原始的 `publisher` 加上了 `print` 语句，来把中间过程也打印出来。请你观察输出，看看原始的序列 `Publisher` 发生了什么，它们对应了怎样的事件。尝试将 `contains(3)` 修改成序列中不存在的数字，比如 `contains(10)`，输出有什么变化？它和 `filter` 相比有什么特点？普通的 `Sequence` 中也是如此吗？

2. 更多的变形操作

除了 `filter` 和 `contains` 外，还有很多其他常见 Operator，比如 `prefix`, `drop`, `replaceNil`, `replaceEmpty`, `min`, `max`, `allSatisfy`, `collect` 等。请你参照 Apple 提供的 [Publisher 文档](#)，具体了解一下每个操作的涵义，并且在 Playground 中实际进行验证。

3. 使用 `merge` 整合事件流

Combine 中有不少处理两个或多个事件流，并将它们合并到一起的操作符。`merge` 操作就是其中之一：它将两个事件流进行合并，在对应的时间完整保留两个事件流的全部事件：

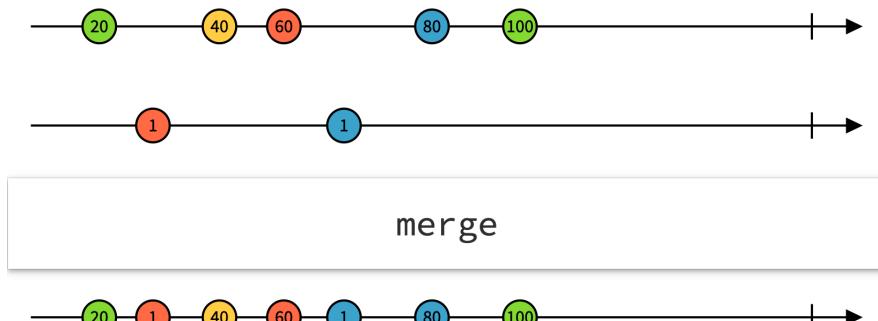
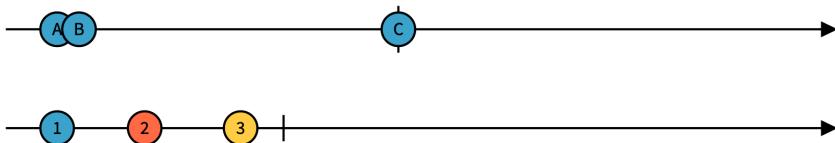


Figure 6.8: `merge` 操作

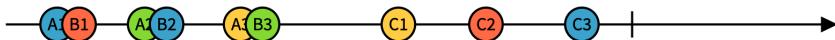
在本章的提供的 Playground 文件的 Utils.swift 中，我们为 Publisher 定义了一个 timerPublisher 操作符，它可以作用于 [Int: Value] 类型的值，其中的 Value 元素将在被订阅后第 Int 秒被发出。比如 [1: "A", 2: "B", 3: "C"].timerPublisher 将分别在第 1, 2, 3 秒输出 [1.0] - A, [2.0] - B 和 [3.0] - C。请尝试使用 timerPublisher 来验证 merge 的行为。

4. 深入理解 flatMap

我们在本章中看到过 flatMap，它由外层和内层两个 Publisher 构成，外层 Publisher 提供数据，内层 Publisher 提供变形方式并控制输出。在其他一些函数响应式框架中，这个操作符有时候被叫做 mergeMap：也就是，外层 Publisher 的元素以内层 Publisher 为基准进行 map，然后再把各结果 merge。下图是一个 flatMap 更一般的时序图例：

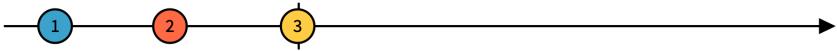
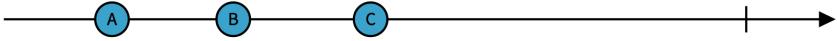


```
outer.flatMap { l in inner.map { "\$(l)\$(\$0)" } }
```



请指出上图中 flatMap 的工作方式，为什么输出的部分是这样的事件流？map 和 merge 在图中是如何体现的？

如果是下图的输入的话，结果会是什么？请尝试在输出轴上进行事件标注，并尝试使用上一个练习中的 timerPublisher 来验证结果。



```
outer.flatMap { l in inner.map { "\$(l)\$(\$0)" } }
```



响应式编程边界

7

在前两章我们花了不少篇幅介绍了 Combine 中最重要的部分：Publisher 和 Operator，以及如何使用常见的 Operator 来构建出我们最终想要的逻辑。这是响应式编程的主轴和重点，是我们编写类似程序时的核心部分。但是，距离实际在开发中使用响应式编程，我们还有两个重要的问题没有解决，它们涉及的核心话题是响应式编程的边界在哪里；或者说，响应式的程序要如何与剩余部分互动。

第一个问题：最初的 Publisher 从何而来？在前面的章节里，我们使用的都是由一个数组生成的 Publishers.Sequence 类型进行举例。这样的 Publisher 用来做学习示例自然是再适合不过了，但是实际的 app 开发中应该鲜有类似情况。Combine 框架真正有意义的使用情景是那些涉及异步操作，会产生事件流的地方。像是网络请求，用户输入等等。Combine 框架中为我们提供了 Subject 角色，来把传统指令式编程转换到响应式世界。Foundation 也提供了一系列便利的方式，来获取初始 Publisher。我们在本章中会看到它们的使用方式。

如果说第一个问题是“从哪里来”，那么第二个问题可以归结为“到哪里去”：经过 Operator 转换的 Publisher 最终需要被 Subscriber 订阅，并用来驱动 app 逻辑或者 UI。另外，关于 Publisher 共享和内存利用等方面，我们也还要做一些特别的说明。

Subject 行为

让我们现在解决第一个问题：最初的 Publisher 从何而来。我们知道 Publisher 是一个协议，我们可以通过自己创建一个类型，去满足 Publisher 协议，按照要求发送各种事件，来得到自定义的 Publisher。不过更多的时候，如果只是为了将既有的指令式程序进行转换的话，我们可以使用 Subject。

Subject 也是 Combine 框架中的一个协议，它为我们提供了从外界发送数据的方式。在 [Combine 理论的第一章](#) 里，我们已经介绍过关于 Subject 角色的定义和基本类型了。首先，遵守该协议的类型也需要遵守 Publisher。除此之外，它还暴露出一些 send 方法，可以让外界通过它们来发送事件：

```
public protocol Subject : AnyObject, Publisher {  
    func send(_ value: Self.Output)  
    func send(completion: Subscribers.Completion<Self.Failure>)  
    func send(subscription: Subscription)  
}
```

外界每次调用 `send(_)` 或者 `send(completion:)`，对应的 `Subject` 就会向外发布一个事件：

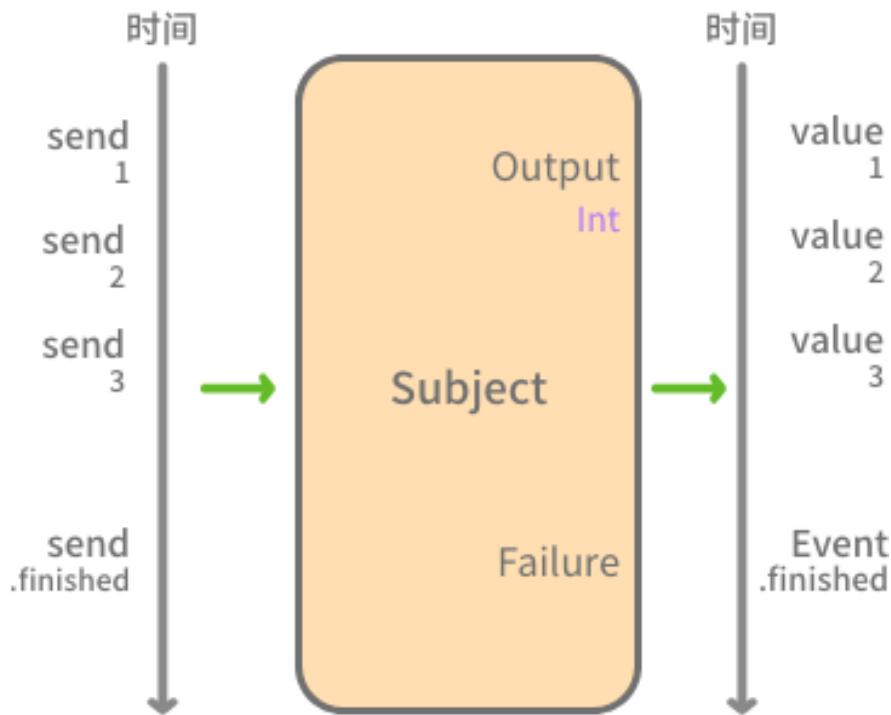


Figure 7.1: 一般的 `Subject` 行为

Combine 内部预先已经定义了很多常用的 Publisher，比如 Publishers.Map 和 Just 等。和 Publisher 一样，框架里也预先定义了两种最常用的 Subject，它们分别是 PassthroughSubject 及 CurrentValueSubject。

在前面我们已经看到过 Subject 的用法了：PassthroughSubject 简单地将 send 输入的内容如实反馈，而 CurrentValueSubject 则保留一个最后的值，并在被订阅时将这个值作为事件发送。可以用时序图来表示两种 Subject 的行为。

PassthroughSubject 在订阅时不产生值，并将之后的值进行传递。

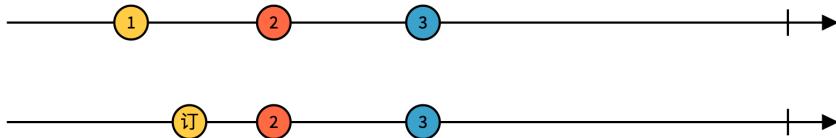


Figure 7.2: PassthroughSubject

CurrentValueSubject 则在订阅时立即将上次的值发送给订阅者。

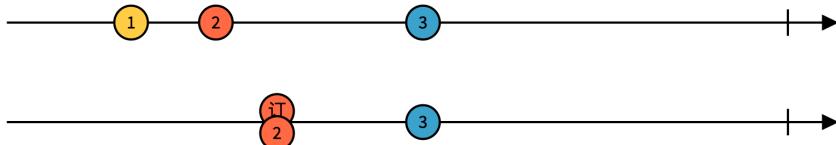


Figure 7.3: CurrentValueSubject

时序

有了 Subject 这个从外部发送事件的方式，我们就可以很容易地构建出按照时序发布事件的 Publisher 了。例如，下面的 subject 将每隔一秒顺次输出 1, 2 和 .finished

```
let s1 = check("Subject") {
    () -> PassthroughSubject<Int, Never> in
    let subject = PassthroughSubject<Int, Never>()
    delay(1) {
        subject.send(1)
        delay(1) {
            subject.send(2)
            delay(1) {
                subject.send(completion: .finished)
            }
        }
    }
    return subject
}
```

Subject 提供了自由控制 Publisher 行为的机会，我们可以精确地控制每个事件的时序。比如，上一章中练习题中 merge 图示，可以使用 PassthroughSubject 表示如下：

```
let subject_example1 = PassthroughSubject<Int, Never>()
let subject_example2 = PassthroughSubject<Int, Never>()

check("Subject Order") {
    subject_example1.merge(with: subject_example2)
}
```

```
subject_example1.send(20)
subject_example2.send(1)
subject_example1.send(40)
subject_example1.send(60)
subject_example2.send(1)
subject_example1.send(80)
subject_example1.send(100)
subject_example1.send(completion: .finished)
subject_example2.send(completion: .finished)
```

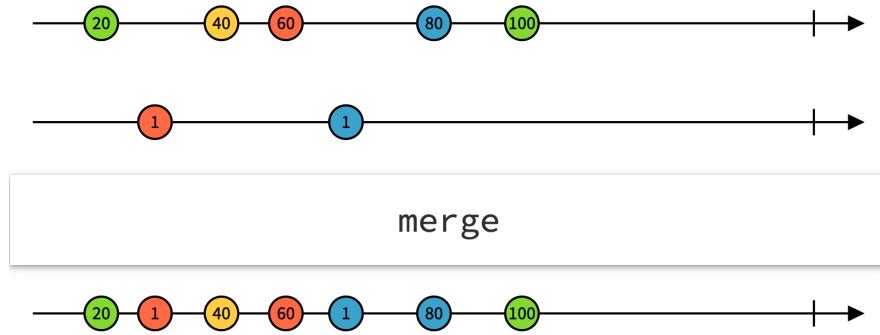


Figure 7.4: 上例中的 merge 操作

Subject 严格遵守时序的特性，让我们有机会同时验证那些对时序敏感的多个 Publisher 的组合。

zip

在布尔逻辑中，and 及 or 可以“合并”多个布尔值。我们在 Publisher 的世界中也已经看到过一些操作两个或多个 Publisher 的操作符，比如 merge 和 flatMap：前者简单地将两者合并，后者则是外层提供输入，内层转换输出。Publisher 上还有一些其他的操作，比如 zip 和 combineLatest，能让我们在**时序**上对控制多个 Publisher 的结果进行类似 and 和 or 的合并，它们在构建复杂 Publisher 逻辑时也十分有用。

对普通的 Sequence，Swift 标准库中也提供了 zip 操作，它将 Sequence<Element1> 和 Sequence<Element2> 的两个序列合并成一个元素类型为多元组 (Element1, Element2) 的序列。zip 将从两个序列中取出 index 相同的元素，把它们组合为多元组，然后放到返回的序列中去：

```
zip([1, 2, 3, 4, 5], ["A", "B", "C", "D"])
// [(1, "A"), (2, "B"), (3, "C"), (4, "D")]
```

虽然上例中我们把结果写成了数组的样式，但实际上，对于 Sequence 进行例程中的 zip 操作，返回的并不是数组，而是一个特殊的 Zip2Sequence。这个类型只是简单地持有了输入的两个 Sequence，并且提供合适的 Iterator 来对持有的序列迭代并产生新的元素。关于 Swift 中序列的更深入的内容，不属于本书的范畴。如果你对这部分内容感兴趣的话，推荐阅读我们的另一本图书：[《Swift 进阶》](#)。

Publisher 中的 zip 和 Sequence 的 zip 相类似：它会把两个（或多个）Publisher 事件序列中在同一 **index** 位置上的值进行合并，也就是说，Publisher1 中的第一个事件和 Publisher2 中的第一个事件结对合并，Publisher1 中的第二个事件和 Publisher2 中的第二个事件合并，以此类推：

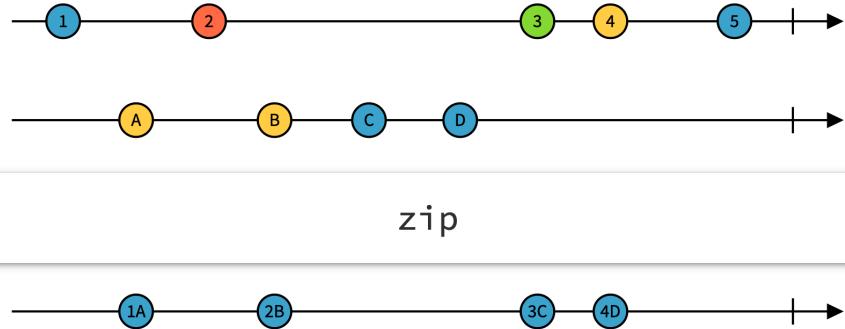


Figure 7.5: `zip` 操作

比如上图中，Publisher1 中第一个位置值为 1，在等待到 Publisher2 的第一个值 A 到来时，两者进行合并，在输出的 Publisher 中形成 (1, A)。两个 Publisher 第二个位置的 2 和 B 也同理。在 Publisher2 中 C 和 D 连续发生时，由于 Publisher1 里对应序列位置的事件还没有发生，因此并不会输出在最终的 Publisher 中，直到 Publisher1 里的 3 和 4 被发布。最后 5 在 Publisher2 里并没有对应位置的 Output 事件，因此它被最终的 Publisher 忽略了。

```

let subject1 = PassthroughSubject<Int, Never>()
let subject2 = PassthroughSubject<String, Never>()

check("Zip") {
    subject1.zip(subject2)
}

subject1.send(1)
subject2.send("A")
subject1.send(2)

```

```
subject2.send("B")
subject2.send("C")
subject2.send("D")
subject1.send(3)
subject1.send(4)
subject1.send(5)

// 输出:
// ----- Zip -----
// receive subscription: (Zip)
// request unlimited
// receive value: ((1, "A"))
// receive value: ((2, "B"))
// receive value: ((3, "C"))
// receive value: ((4, "D"))
```

zip 在时序语义上更接近于“当...且...”，**当** Publisher1 发布值，**且** Publisher2 发布值时，将两个值合并，作为新的事件发布出去。在实践中，zip 经常被用在合并多个异步事件的结果，比如同时发出了多个网络请求，希望在它们**全部完成**的时候把结果合并在一起。

combineLatest

至今为止，我们看到的 Operator 或多或少都能在普通的 Sequence 中找到类似的操作。不过 Publisher 带有时序的特点，注定了会有一些不同的操作，combineLatest 是一个很典型的例子，和 zip 相反，它的语义接近于“当...或...”，**当** Publisher1 发布值，**或者** Publisher2 发布值时，将两个值合并，作为新的事件发布出去。

不论是哪个输入 Publisher，只要发生了新的事件，`combineLatest` 就把新发生的事件值和另一个 Publisher 中当前的最新值合并。听起来会有点复杂？其实结合时序图来看的话并不难。`combineLatest` 操作的图示如下：

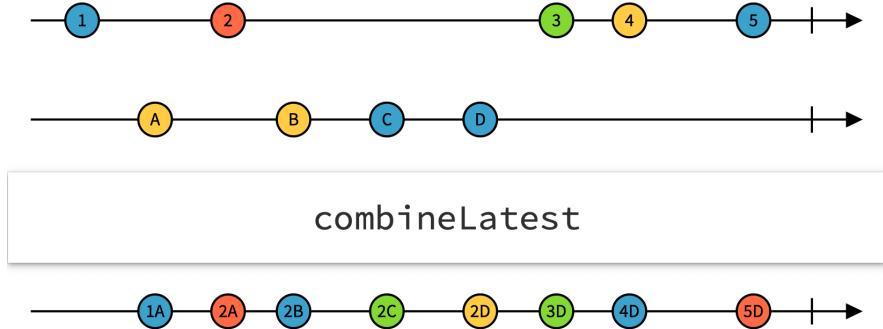


Figure 7.6: `combineLatest` 操作

上图中，除了首个元素 1 以外，其余的每个事件值，不论是从哪个 Publisher 发出的，都会被用来进行组合和输出。比如在 Publisher2 输出 A 时，`combineLatest` 将使用这个 A 和 Publisher 的当前最终值 (latest) 1 组合出结果 Publisher 的事件值 1A。同样，随后 Publisher1 发布的 2 也和 Publisher2 的最终值 A 组合得到 2A。这个过程持续下去，直到两个 Publisher 都结束。

```
let subject3 = PassthroughSubject<String, Never>()
let subject4 = PassthroughSubject<String, Never>()
```

```
check("Combine Latest") {
    subject3.combineLatest(subject4)
}
```

```
subject3.send("1")
subject4.send("A")
subject3.send("2")
subject4.send("B")
subject4.send("C")
subject4.send("D")
subject3.send("3")
subject3.send("4")
subject3.send("5")

// 输出:
// ----- Combine Latest -----
// receive subscription: (CombineLatest)
// request unlimited
// receive value: ((1, "A"))
// receive value: ((2, "A"))
// receive value: ((2, "B"))
// receive value: ((2, "C"))
// receive value: ((2, "D"))
// receive value: ((3, "D"))
// receive value: ((4, "D"))
// receive value: ((5, "D"))
```

在实践中，`combineLatest` 被用来处理多个可变状态，在其中某一个状态发生变化时，获取这些全部状态的最新值。比如你的 UI 上有多个 `TextField`，你可能想要在其中某一个值变动时获取到所有 `TextField` 中的值并对它们进行检查（没错，我说的就是用户注册）。

对于 zip 和 combineLatest，它们有一个共同特点，那就是结合后的新 Publisher 所发出的是数据的多元组。对这两种操作，一种常见的模式是将结果的发出多元组数据的 Publisher 沿着响应链继续传递，使用我们之前看到过的各类 Operator 来获取能实际驱动 UI 和 app 状态的 Publisher。

响应式和指令式的桥梁

一般来说，开发者入门时主要还是学习更符合直觉的指令式的编程方式，而在 Combine 框架问世之前，Apple 平台的开发，包括系统级别的 API 和大多数的第三方框架，也都是按照指令式编程的方式来编写的。想要在 Combine 的响应式框架中使用这些指令式的 API，我们需要一些手段来把这些命令转换为合适的 Publisher。

Future

在上一章中，我们介绍过 Just：它在初始化时接受一个具体值，在被订阅时，Just 将这个预设值发布出来。对 Just 来说，订阅和值的发布其实是同步过程，它往往被用来将某个值“包装”成一个 Publisher 来提供给各类 Publisher 的合并操作。

如果我们希望订阅操作和值的发布是异步行为，不在同一时间发生的话，可以使用 Future。Future 提供了一种方式，可以让我们创建一个接受未来的事件的 Publisher。

比如我们有一个使用 URLSession 的传统的网络请求 API：

```
func loadPage(  
    url: URL,  
    handler: @escaping (Data?, URLResponse?, Error?) → Void)  
{  
    URLSession.shared.dataTask(with: url) {  
        data, response, error in
```

```
    handler(data, response, error)
}.resume()
}
```

这个 API 在被调用时会创建请求，访问网络，下载数据。这些都是“未来”的行为和数据，我们可以使用 Future 来将这一行为包装成一个 Publisher：

```
let future = check("Future") {
    Future<(Data, URLResponse), Error> { promise in
        loadPage(url: URL(string: "https://example.com")!) {
            data, response, error in
            if let data = data, let response = response {
                promise(.success((data, response)))
            } else {
                promise(.failure(error!))
            }
        }
    }
}

// 输出:
// ----- Future -----
// receive subscription: (Future)
// request unlimited
// receive value: ((1270 bytes, <NSHTTPURLResponse: ...>))
// receive finished
```

在 iOS 13 / macOS 10.15 SDK 中，现在 Foundation 中 URLSession 也有 Publisher 的请求方式了，所以我们并不需要像上面这样再自己用 Future 进行包装，而只用使用对应的 URLSession 的返回 Publisher 的请求方法就可以了。我们会在下

一节中看到基于 Combine 的网络请求的内容。而对于其他很多并没有提供直接产生 Publisher API 来说，像例子中这样用 Future 来完成从异步指令式向响应式的桥接，是简单有效，也被普遍认可的方案。

但是要注意，Future 只能为我们提供一次性 Publisher：对于提供的 promise，你只有两种选择：发送一个值并让 Publisher 正常结束，或者发送一个错误。因此，Future 只适用于那些必然会产生事件结果，且至多只会产生一个结果的场景。比如刚才看到的网络请求：它要么成功并返回数据及响应，要么直接失败并给出 URLError。一个 dataTask 的网络请求不会永远不发送任何事件，也不会产生多次的响应，用 Future 进行包装恰得其所。如果你的异步 API 有可能不发送任何一个值，而是可能发布两个或更多的值的话，你会需要一个更加一般性的 Publisher 类型来把指令式程序转换为响应式程序，这个类型就是 Subject。

使用 Subject

相对于单次事件的网络请求，可以发布多次事件的操作在日常开发中更加常见：比如把每次由于 TextField 导致的键盘出现看作一次事件的话，一般情况下你并不能控制用户调出多少次键盘；再比如设定了某个计时器，希望间隔一秒反复调用某个方法，直到计时器被停止之前，对这个方法的调用也是一个可重复的多次发生的事件。

想要把这种情景下的异步事件转换为可订阅的事件流，Subject 会是最容易的方案：

```
var observer: NSObjectProtocol?  
  
let subject = PassthroughSubject<(), Never>()  
observer = NotificationCenter.default.addObserver(  
    forName: UIDevice.keyboardDidShowNotification,  
    object: nil,  
    queue: .main)  
{ _ in subject.send() }
```

这样一来，我们就可以使用 Publisher 的丰富的操作来对 subject 结果进行变形了。

```
let subject = PassthroughSubject<Date, Never>()

Timer.scheduledTimer(withTimeInterval: 1, repeats: true) { _ in
    subject.send(Date())
}

let timer = check("Timer") {
    subject
}

// 输出:
// ----- Timer -----
// receive subscription: (PassthroughSubject)
// request unlimited
// receive value: (<Date>)
// receive value: (<Date + 1s>)
// receive value: (<Date + 2s>)
// ...
```

当然，我们马上就会看到，对于上面例子中的 URLSession, Notification 和 Timer, Foundation 框架中已经为我们提供了简便的直接获取 Publisher 的方法，并不需要我们自己再去封装一次。但是，对于 Cocoa 中其他茫茫多的闭包回调和 delegate 等，使用 Future 或者 Subject 将它们封装为响应式的事件流，是很常见的做法。

Foundation 中的 Publisher

为了方便使用，Foundation 中为几个常见的异步任务直接提供了获取 Publisher 的方式。本节中将对这些方式进行一些简单的介绍。

URLSession Publisher

为了能看到一个实际的例子，我们使用 [httpbin](https://httpbin.org) 的 API 来进行一些实践。

httpbin.org 提供了一套简单的 HTTP 请求和响应的测试 API，比如我们如果访问 /get，并带上一个自定义的参数，那么返回的 JSON 结果中 args key 下也会带上这些参数。我们可以把 /get 请求看作一个简单的回声 (echo) 服务。比如，下面的请求所对应的响应，在结果 JSON 中包含了请求时的 “foo=bar”

```
> curl -X GET "https://httpbin.org/get?foo=bar" -H "accept: application/json"
```

```
# 结果:
{
  "args": {
    "foo": "bar"
  },
  "headers": {
    "Accept": "application/json",
    "Host": "httpbin.org",
    "User-Agent": "curl/7.64.1"
  },
  "url": "https://httpbin.org/get?foo=bar"
}
```

在 Swift 中，我们可以创建一个满足 Decodable 的 struct 来表示这个返回的 JSON。这里，我们只关心 args 下的回声值：

```
struct Response: Decodable {
  struct Args: Decodable {
    let foo: String
  }
  let args: Args?
}
```

URLSession 的 `dataTaskPublisher(for:)` 方法可以直接返回一个 Publisher，它会在网络请求成功时发布一个值为 (Response, Data) 的事件，请求过程失败时，发送类型为 URLError 的错误。以上面的请求为例：

```
let subscription = check("URL Session") {
    URLSession.shared
        .dataTaskPublisher(
            for: URL(string: "https://httpbin.org/get?foo=bar")!)
        .map { data, _ in data }
        .decode(type: Response.self, decoder: JSONDecoder())
        .compactMap { $0.args?.foo }
}

// 输出:
// -----
// receive subscription: (CompactMap)
// request unlimited
//
// receive value: (bar)
// receive finished
```

在 `dataTaskPublisher` 发送一个新的事件值时，我们将其中的 Data 通过 map 的方式提取出来，并交给 decode 这个 Operator 进行处理。decode 要求上游 Publisher 的 Output 类型是 Data，它会使用参数中接受的 decoder (本例中是 `JSONDecoder()`) 来对上游数据进行解析，生成对应类型 (本例中是 Response) 的实例，并作为新的 Publisher 事件发布出去。

Timer Publisher

Foundation 中的 Timer 类型也提供了一个方法，来创建一个按照一定间隔发送事件的 Publisher：

```
extension Timer {  
    public static func publish(  
        every interval: TimeInterval,  
        tolerance: TimeInterval? = nil,  
        on runLoop: RunLoop,  
        in mode: RunLoop.Mode,  
        options: RunLoop.SchedulerOptions? = nil  
    ) -> Timer.TimerPublisher  
}
```

interval, runLoop 和 mode 参数分别定义了 Timer 事件的间隔秒数，计时所使用 RunLoop 和相应的模式等。比如我们想要在 RunLoop.main 上创建一个每隔一秒发送一次事件的 Timer：

```
let temp = check("Timer") {  
    Timer.publish(every: 1, on: .main, in: .default)  
}
```

在 Playground 中运行这段代码，你会发现这个 Publisher 并没有像你预想那样，源源不断地以一秒为间隔发布新值：

```
// 输出:  
// ----- Timer -----  
// receive subscription: (Timer)  
// request unlimited
```

如果我们检查这个返回值的类型，会发现 Timer.TimerPublisher 是一个满足 ConnectablePublisher 的类型。ConnectablePublisher 不同于普通的 Publisher，你需要明确地调用 connect() 方法，它才会开始发送事件：

```
let timer = Timer.publish(every: 1, on: .main, in: .default)
check("Timer Connected") {
    timer
}
timer.connect()

// 输出:
// ----- Timer Connected -----
// receive subscription: (Timer)
// request unlimited
// receive value: (<Date>)
// receive value: (<Date + 1s>)
// receive value: (<Date + 2s>)
// ...
```

一个显而易见的问题是，既然我们需要调用 connect() 才能让事件开始发生，那当我们不再关心这个事件流的时候，是不是应该本着资源使用的“谁创建，谁释放”的原则，让这个事件流停止发送呢？答案是肯定的：connect() 会返回一个 Cancellable 值，我们需要在合适的时候调用 cancel() 来停止事件流并释放资源。同样地，对于订阅来说，大多数情况下我们也需要及时取消，以保证内存不发生泄漏。我们在本章稍后部分，会更加深入地讨论 Combine 中的一些内存管理的问题。

对于普通的 Publisher，当 Failure 是 Never 时，就可以使用 makeConnectable() 将它包装为一个 ConnectablePublisher。这会使得该 Publisher 在等到连接（调用 connect()）后才开始执行和发布事件。在某些

情况下，如果我们希望延迟及控制 Publisher 的开始时间，可以使用这个方法。

对 ConnectablePublisher 的对象施加 autoconnect() 的话，可以让这个 ConnectablePublisher “恢复” 为被订阅时自动连接。

Notification Publisher

和 Timer 类似，Foundation 中的 NotificationCenter 也提供了创建 Publisher 的辅助 API：

```
extension NotificationCenter {
    public func publisher(
        for name: Notification.Name,
        object: AnyObject? = nil
    ) -> NotificationCenter.Publisher
}
```

NotificationCenter.Publisher 的 Output 值是 Notification，这和传统的通知体系没有任何区别，使用 NotificationCenter 的 Publisher 的方法也和其他的 Publisher 完全一样，你可以自由地与其他 Publisher 进行组合，这正是 Combine 对于异步操作的统一所带来的便利。

@Published

我们在前面的章节中已经看到过 @State 和 @ObservedObject 这样的 Property Wrapper，它们对属性值的 getter 和 setter 进行包装，来实现一些常见功能。类似地，Combine 中存在 @Published 封装，用来把一个 class 的属性值转变为 Publisher。它同时提供了值的存储和对外的 Publisher (通过投影符号 \$ 获取)。在被订阅时，当前值也会被发送给订阅者，它的底层其实就是一个 CurrentValueSubject：

```
class Wrapper {  
    @Published var text: String = "hoho"  
}  
  
var wrapper = Wrapper()  
check("Published") {  
    wrapper.$text  
}  
  
wrapper.text = "123"  
wrapper.text = "abc"  
  
// 输出:  
// ----- Published -----  
// receive subscription: (CurrentValueSubject)  
// request unlimited  
// receive value: (hoho)  
// receive value: (123)  
// receive value: (abc)
```

在 ObservableObject 的场景中，@Published 被用来自动生成对应的 Publisher 并处理 objectWillChange 的调用。我们在 SwiftUI 数据状态一章中已经涉及到相关内容了。对于在 ObservableObject 类型下的 @Published 的行为，Swift 编译器和 SwiftUI 进行了特殊的处理。

订阅和绑定

本章前半部分我们将“Publisher 从何而来”这个问题梳理清楚了，现在可以谈一谈“到何处去”的问题了。

通过 sink 订阅 Publisher 事件

在前面的所有例子中，我们都使用了预先定义的 check 函数来检查 Publisher 的输出。这个函数的实现如下：

```
public func check<P: Publisher>(  
    _ title: String,  
    publisher: () -> P  
) -> AnyCancellable  
{  
    print("----- \\\(title) -----")  
    defer { print("") }  
    return publisher()  
        .print()  
        .sink(  
            receiveCompletion: { _ in },  
            receiveValue: { _ in }  
        )  
}
```

在函数中，通过调用 publisher()，我们将输入的闭包转换为实际的 Publisher，然后附上一个 print() 操作，来把所有的事件输出到控制台。最后，sink 负责订阅这个 Publisher，并返回一个 AnyCancellable 值。

具体来说，sink 的函数签名如下：

```
func sink(  
    receiveCompletion:  
        @escaping ((Subscribers.Completion<Self.Failure>) -> Void),  
    receiveValue:
```

```
@escaping ((Self.Output) → Void)
) → AnyCancellable
```

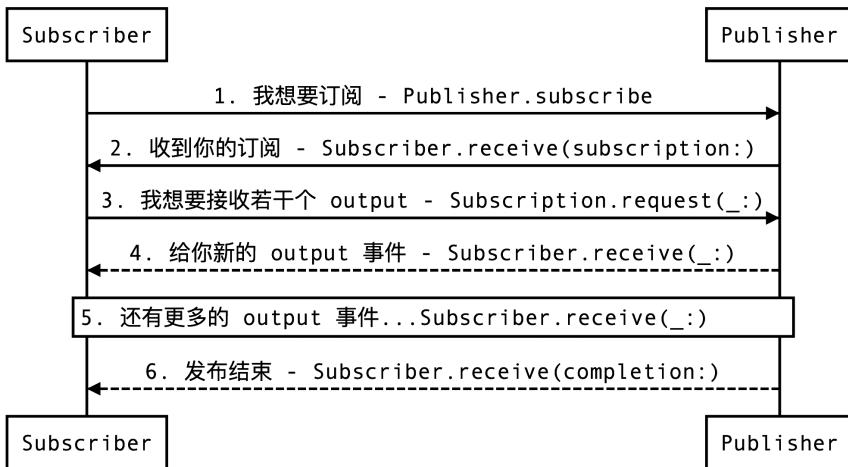
`sink` 接受两个参数，它们都是“闭包回调”：`receiveCompletion` 在 `Self` (也就是调用 `sink` 的 `Publisher`) 发布结束事件时被调用，`receiveValue` 则在 `Self` 发布值时被调用。在 `check` 的例子里，因为 `print()` 承接了将事件输出到控制台的工作，所以 `sink` 什么都没有做。如果 `Combine` 中没有 `print()` 操作的话，`check` 方法的返回就需要变为：

```
return publisher().sink(
    receiveCompletion: { complete in
        switch complete {
            case .failure(let error):
                Swift.print("receive error: \(error)")
            case .finished:
                Swift.print("receive finished")
        }
    },
    receiveValue: { value in
        Swift.print("receive value: \(value)")
    }
)
```

细心的读者应该发现了，`check` 里对 `Publisher` 的 `print()` 操作除了打印发布的值和结束事件之外，毫无例外地每次都打印了额外的两项内容：

```
// receive subscription: ({Publisher-Type})
// request unlimited
```

如果你还记得的话，一个完整的“发布-订阅”流程如下所示：



当我们使用 `sink` 完成订阅时，一个特殊的 `Subscriber` 类型 `Subscribers.Sink` 会被创建，并参与到上面的流程中。它将订阅对象 `Publisher`，并自动声明想要接收无限多个新值。在此之后，响应式世界中的 `Publisher` 将把新的值和结束事件作为参数，调用 `sink` 传入的两个闭包，从而将响应式的事件流转化为普通的指令操作。

`Subscriber` 可以指定想要接收的新值的个数，这不仅在订阅初期可以通过 `Subscription.request` 来告知 `Publisher`，也可以通过 `Subscriber.receive` 返回特定的 `Subscribers.Demand` 值来指定接下来能够处理的值的个数。通过这些机制，`Combine` 将可以实现背压 (Backpressure)。`.unlimited` 表示不设上限，但当上游 `Publisher` 的值生产速度大于下游的消费速度时，下游的缓冲区就会发生溢出。指定合适的背压策略，通过控制上限，可以让系统下游不发生崩溃的同时，有机会对部分溢出事件做额外处理 (比如丢弃或者告知上游不要再接受新的事件)。

在客户端开发中，需要处理背压的场景非常有限，但是在服务端开发处理大规模数据时，这会是无法绕过机制。相关话题超出了本书内容的涵盖范围，但有兴趣的读者不妨自行查找相关资料进行理解。

通过 assign 绑定 Publisher 值

除了 Subscribers.Sink 以外，Combine 里还有另一个内建的 Subscriber：Subscribers.Assign，它可以用来自将 Publisher 的输出值通过 key path 绑定到一个对象的属性上去。在 SwiftUI 中，这种值通常会是 ObservableObject 中的属性值，它进一步会被用来驱动 View 的更新。

假设我们有下面这样一个 class，它用来表示一个时钟：

```
class Clock {  
    var timeString: String = "---:---:---"  
    didSet { print("\(timeString)") }  
}  
}
```

我们通过 Timer.publish 创建一个每秒发布一次当前 Date 值的计时器 Publisher：

```
let clock = Clock()  
  
let formatter = DateFormatter()  
formatter.timeStyle = .medium  
  
let timer = Timer.publish(every: 1, on: .main, in: .default)  
var token = timer  
.map { formatter.string(from: $0) }  
.assign(to: \.timeString, on: clock)
```

```
timer.connect()
```

在 Playground 中运行上面的代码，可以看到 `timeString` 的 `didSet` 将被反复调用，并输出当前的时间字符串。

我们使用了预先定义的 `formatter` 通过 `map` 将 `Date` 变形为我们需要的字符串表示形式，然后用 `assign` 将这个字符串绑定给了 `clock` 的 `timeString`。注意 `assign` 所接受的第一个参数的类型为 `ReferenceWritableKeyPath`，也就是说，只有 `class` 上用 `var` 声明的属性可以通过 `assign` 来直接赋值。

`assign` 的另一个“限制”是，上游 `Publisher` 的 `Failure` 的类型必须是 `Never`。如果上游 `Publisher` 可能会发生错误，我们则必须先对它进行处理，比如使用 `replaceError` 或者 `catch` 来把错误在绑定之前就“消化”掉。

关于上例中，一个有意思的地方是，我们用 `token` 暂时存储了 `assign` 的返回值。这是一个类型为 `AnyCancellable` 的对象。如果我们把 `var token =` 从上面的代码中去掉，`clock` 的 `timeString` 将不再被设置，这段代码也即停止工作。这是因为返回值没有被 `Playground` 持有，导致了这个 `AnyCancellable` 被立即释放，进而取消了 `assign` 的订阅。我们会在本节最后关于 `Combine` 内存管理的部分，详细解释这个问题。

Publisher 的引用共享

让我们来看一个更具体一些的用例，设想我们有一个涉及到网络请求的界面，它需要同时显示网络请求是否成功，以及请求所得到的结果。我们用 `isSuccess` 来代表网络请求成功返回，即 HTTP 状态码为 200 的情况；用 `text` 来代表 `response` 中的返回数据：

```
class LoadingUI {  
    var isSuccess: Bool = false  
}
```

```
    var text: String = ""  
}  
}
```

对于请求本身，就使用 URLSession Publisher 的例子，对 httpbin.org 进行访问：

```
struct Response: Decodable {  
    struct Foo: Decodable {  
        let foo: String  
    }  
    let args: Foo?  
}  
  
let dataTaskPublisher = URLSession.shared  
.dataTaskPublisher(  
    for: URL(string: "https://httpbin.org/get?foo=bar")!)
```

为了驱动 UI，我们需要完成两项任务：检查 response 中的 HTTP statusCode，以及将 data 解码为 Response 对象，我们使用 Operator 创建两个执行不同逻辑的 Publisher：

```
let isSuccess = dataTaskPublisher  
.map { data, response → Bool in  
    guard let httpRes = response as? HTTPURLResponse else {  
        return false  
    }  
    return httpRes.statusCode == 200  
}  
.replaceError(with: false)  
  
let latestText = dataTaskPublisher
```

```
.map { data, _ in data }

.decode(type: Response.self, decoder: JSONDecoder())

.compactMap { $0.args?.foo }

.replaceError(with: "")
```

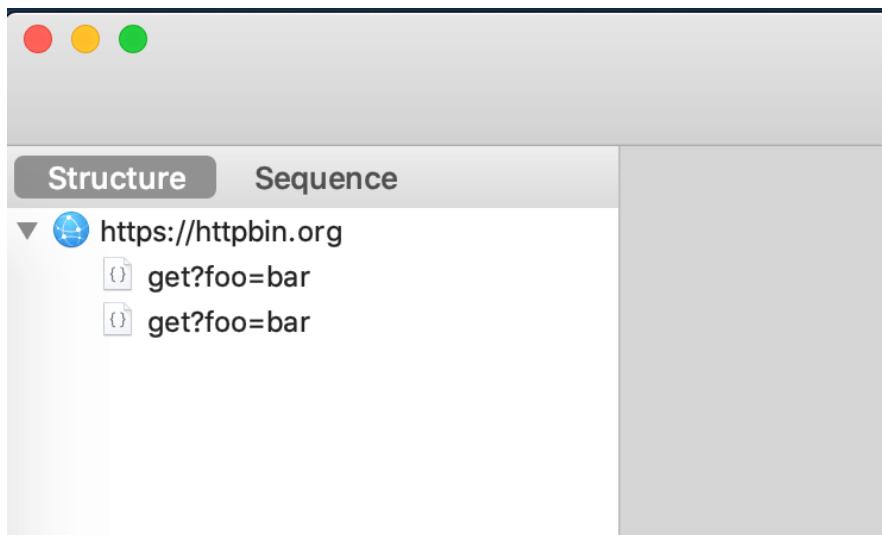
接下来，我们就可以将这两个 Publisher 的输出值绑定到 LoadingUI 的对应属性上了：

```
let ui = LoadingUI()

var token1 = isSuccess.assign(to: \.isSuccess, on: ui)

var token2 = latestText.assign(to: \.text, on: ui)
```

一切正常运行，`isSuccess` 和 `text` 都可以被网络请求的 Publisher 正常驱动。但是，如果我们检查实际发生的请求，会看到发生了两次对目标 URL 的访问：



在绝大多数情况下，这都是严重的 bug：比如有可能一个请求失败了，但另一个成功了，这会造成 `isSuccess` 和 `text` 表示的是不同响应的状态；如果在一个副作用更大

的请求里，比如删除资源或者注册用户的时候，这带来的问题就更加严重。本例的 `dataTaskPublisher` 是 `struct`，它遵循值语义 (`value semantics`)。在变形的时候，负责网络请求的 `Publisher` 将被复制一份。这样一来，最后使用 `assign` 操作 `isSuccess` 和 `latestText` 时，最终订阅的是两个不同的 `Publisher`，因此网络请求也发生了两次。

想要改变这个行为，可以将值语义的 `dataTaskPublisher` 转变为引用语义 (`reference semantics`)。我们只要在创建 `dataTaskPublisher` 后加上 `share()` 即可。通过 `share()` 操作，原来的 `Publisher` 将被包装在 `class` 内，对它的进一步变形也会适用引用语义：

```
let dataTaskPublisher = URLSession.shared
    .dataTaskPublisher(
        for: URL(string: "https://httpbin.org/get?foo=bar")!)
    .share()
```

对于多个 `Subscriber` 对应一个 `Publisher` 的情况，如果我们不想让订阅行为反复发生 (比如上例中订阅时会发生网络请求)，而是想要共享这个 `Publisher` 的话，使用 `share()` 将它转变为引用类型的 `class`。

Cancellable, AnyCancellable 和内存管理

在上面 `Timer` 的案例中，对计时器的 `Publisher` 执行 `connect()` 后得到的结果是一个满足 `Cancellable` 协议的对象；用 `sink` 或 `assign` 来订阅某个 `Publisher` 时，我们必须持有返回值才能让这个订阅正常工作，该返回值的类型为 `AnyCancellable`。

上面的操作有一个共同的特点，那就是它们都要求随着时间流动，计时器或者订阅要能继续响应和工作。这必然需要某种“资源”，并持有它们，以保持自己在内存中依然存在。对于 `Cancellable`，我们需要在合适的时候主动调用 `cancel()` 方法来完结：比如停止 `Timer` 的继续计时。如果我们在没有调用 `cancel()` 的情况下就将

`connect` 的返回值忽略或者释放掉，那么我们就将永远无法停止这个 `Timer`，它会一直计时，并造成内存泄漏。

和 `Cancellable` 这个抽象的协议不同，`AnyCancellable` 是一个 `class`，这也赋予了它对自身的生命周期进行管理的能力。对于一般的 `Cancellable`，例如 `connect` 的返回值，我们需要显式地调用 `cancel()` 来停止活动，但 `AnyCancellable` 则在自己的 `deinit` 中帮我们做了这件事。换句话说，当 `sink` 或 `assign` 返回的 `AnyCancellable` 被释放时，它对应的订阅操作也将停止。在实际里，我们一般会把这个 `AnyCancellable` 设置为所在实例（比如 `UIViewController`）的存储属性。这样，当该实例 `deinit` 时，`AnyCancellable` 的 `deinit` 也会被触发，并自动释放资源。如果你对 RxSwift 有了解的话，它的行为和 DisposeBag 很类似，为了操作简便，我们也完全可以用类似的方法在 Combine 中管理订阅并释放资源。

针对上面 Combine 中常见的内存资源相关的操作，可以总结几条常见的规则和实践：

1. 对于需要 `connect` 的 `Publisher`，在 `connect` 后需要保存返回的 `Cancellable`，并在合适的时候调用 `cancel()` 以结束事件的持续发布。
2. 对于 `sink` 或 `assign` 的返回值，一般将其存储在实例的变量中，等待属性持有者被释放时一同自动取消。不过，你也完全可以在不需要时提前释放这个变量或者明确地调用 `cancel()` 以取消绑定。
3. 对于 1 的情况，也完全可以将 `Cancellable` 作为参数传递给 `AnyCancellable` 的初始化方法，将它包装成为一个可以自动取消的对象。这样一来，1 将被转换为 2 的情况。

对于第三点，其实有一个现成的例子，那就是 `sink` 方法的处理方式。定义在 `Publisher` 上的 `sink` 方法，在底层对应的其实是 `Subscribers.Sink`。这个类型只满足 `Cancellable`，而非一个 `AnyCancellable`。在 Combine 的早期 beta 版本，`sink` 返回的也确实是 `Subscribers.Sink`，并且需要使用者手动调用 `cancel()` 来确保资源

释放。但是，为了使用上的方便以及 API 的统一，Apple 最终选择了将它包装成一个 AnyCancellable。

总结

本章中，我们探索了一些 Combine 框架的边界问题：如何使用 Subject 将指令式的操作转换到响应式的世界中，如何依靠 Foundation 里各类 extension 中提供的响应式支持开启事件流，如何通过订阅和绑定的方式让事件流驱动逻辑，以及如何取消订阅并管理内存资源。

响应式编程的理念并不算是新鲜概念，它的发展也已经经过了一段时间。在这几章中，我们有机会通过 Combine 提供的 API 一瞥响应式编程的门径，这让我们在使用 SwiftUI 时，可以利用 Combine 来确定 app 架构。我们在后面的章节中，会使用到这几章里所提到的内容，并看到如何在一个实际的 SwiftUI 项目中，使用 Combine 来构建出坚实可靠的逻辑层，并以此驱动声明式 UI。

练习

1. zip 和 combineLatest 的订阅时机

zip 和 combineLatest 将两个或多个 Publisher 事件值合并为多元组，我们在本章里看到过几个例子：

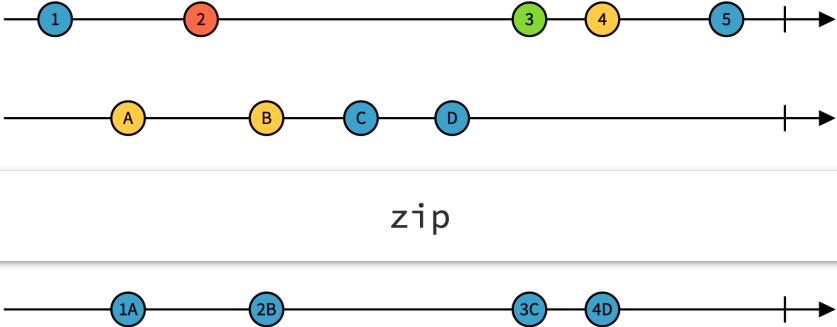


Figure 7.7: zip 操作

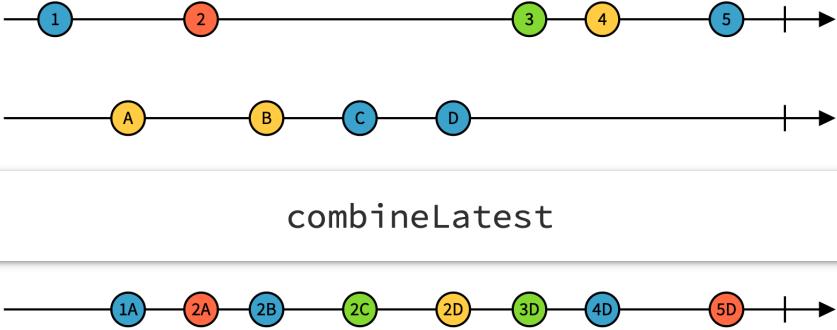


Figure 7.8: combineLatest 操作

这两个例子中，都是先进行订阅，然后两个 PassthroughSubject 才开始发送事件。如果订阅开始的时间晚一些，你能画出对应的事件流示意图吗？比如订阅开始发生于 2 和 B 之间，zip 和 combineLatest 的输出分别会是什么？可以在 Playground 里

确认你的想法吗？如果把 PassthroughSubject 换成 CurrentValueSubject 呢？输出会有变化吗？

2. 学习和使用 debounce

网络请求、I/O 读写或是复杂渲染等都是很耗费资源的操作，对于这类操作，我们希望尽可能少发生。Combine 中有两个 Operator 可以帮助“限流”，它们是 debounce 和 throttle。

debounce 又叫做“防抖”：Publisher 在接收到第一个值后，并不是立即将它发布出去，而是会开启一个内部计时器，当一定时间内没有新的事件来到，再将这个值进行发布。如果在计时期间有新的事件，则重置计时器并重复上述等待过程。比如下面的代码：

```
let searchText = PassthroughSubject<String, Never>()
```

```
check("Debounce") {  
    searchText  
    .debounce(for: .seconds(1), scheduler: RunLoop.main)  
}
```

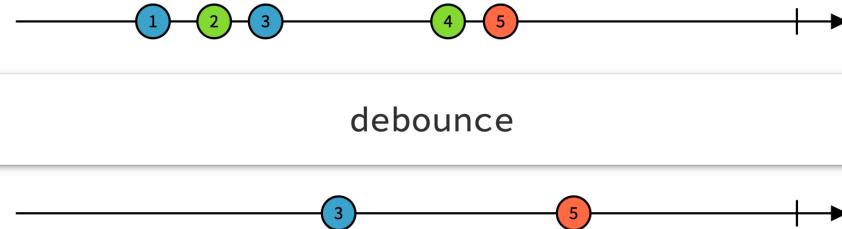
```
delay(0) { searchText.send("S") }  
delay(0.1) { searchText.send("Sw") }  
delay(0.2) { searchText.send("Spi") }  
delay(1.3) { searchText.send("Swift") }  
delay(1.4) { searchText.send("Swift") }
```

```
// 输出:  
// ----- Debounce -----  
// receive subscription: (Debounce)
```

```
// request unlimited  
// receive value: (Swi)  
// receive value: (Swift)
```

我们模拟了一种用户使用键盘输入的情况：先使用比较快的速度顺序输入了“Swi”，进行较长停顿之后，再输入剩余的“ft”。这在日常用户习惯中是非常常见的操作。假设我们希望用这个输入值作为 API 请求时的搜索参数，当用户键入内容时，实时地给出搜索结果。如果不加干预直接使用 `searchText` 的话，每次键盘输入都导致一次值的改变，总共将发出五次请求，这是相当耗费资源的操作，对于用户体验来说也不太理想。

我们为 `searchText` 加上为期 1 秒的 `debounce` 后，将新值事件的次数从五次降低到了两次，不论是对服务器负载，还是用户体验来说，这都是一种更合理的请求策略。



对本章中使用 `URLSession Publisher` 访问 “[httpbin](#)” 的例子，请你也用类似的方式创建一个 `PassthroughSubject` 来控制输入参数，并设定 `debounce` 进行防抖处理，处理下面的输入：

输入时间 (s) - 输入值
0.1 - I
0.2 - Love
0.5 - SwiftUI
1.6 - And
2.0 - Combine

希望的响应结果是通过 `URLSession` 进行网络请求返回的 `Response` 对象内的数据，且响应两次，分别是“`I Love SwiftUI`”和“`I Love SwiftUI And Combine`”。也即，两次请求的 URL 分别为：

```
https://httpbin.org/get?foo=I%20Love%20SwiftUI  
https://httpbin.org/get?foo=I%20Love%20SwiftUI%20And%20Combine
```

提示：可以灵活使用 `scan` 操作来暂存结果，使用 `map` 来对参数字符串进行组合及变形，利用 `URLComponents` 和 `queryItems` 得到编码后的 URL 和请求。最后可以使用 `flatMap` 来把通过输入的 `PassthroughSubject` 和变形操作得到的值传递给 `URLSession` 的 `dataTaskPublisher`。注意在合适的位置插入 `debounce` 限流。

我们有很多手段可以达到相同目的，所以本题答案不止一种。你也可以无视上面的提示，自行探索，甚至给出多种解答。

你可以将下面的代码复制到 Playground 中作为开始。

```
struct Response: Decodable {  
    struct Foo: Decodable {  
        let foo: String  
    }  
    let args: Foo?  
}  
  
let searchText = PassthroughSubject<String, Never>()  
  
check("Debounce") {  
    searchText  
    // ... 在这里添加防抖，变形和网络等操作
```

```
}

delay(0.1) { searchText.send("I") }
delay(0.2) { searchText.send("Love") }
delay(0.5) { searchText.send("SwiftUI") }
delay(1.6) { searchText.send("And") }
delay(2.0) { searchText.send("Combine") }

// 希望的输出:
// ----- Debounce -----
// receive subscription: xxx
// request unlimited
// receive value: (I Love SwiftUI)
// receive value: (I Love SwiftUI And Combine)
```

3. 学习和使用 throttle

如果你能完成上面的练习，相信你对 debounce 已经有深入理解了。Combine 中有关限流的另一个 Operator 是 throttle。它在收到一个事件后开始计时，并忽略计时周期内的后续输入。

请你使用上面的“Swift”输入的例子，[结合文档](#)，来确认 throttle 的行为。并完成以下问答：

1. 你能够仿照 debounce 中的图示画出 throttle 的事件图吗？
2. throttle 中的最后一个参数 latest 会如何影响输出？
3. 当 latest 为 true 时，输出的结果和 debounce 的结果是否相同？是不是可以说 throttle 在这种情况下和 debounce 是等价的？

SwiftUI 架构

8

你现在应该已经具备一些对 Combine 的理解了，本章中我们将回到之前的示例 app：PokeMaster。在真实世界的 SwiftUI一章中，我们已经完成了这个示例 app 的纯 UI 部分，包括宝可梦列表，弹出面板，以及设置表单。从本章开始，我们会逐渐添加代码，来处理用户的行为以及用户行为造成的 app 状态的改变。在此过程中，你可以看到如何在一个相对真实的 SwiftUI 环境中使用和处理这些逻辑。

你可以继续在之前章节的项目上跟随本章内容进行修改和新增，如果你不确定你的实现是否正确，也可以使用本章提供的 Starter 项目。你可以在“SwiftUI-Architecture/PokeMaster-Starter”文件夹中找到它。

UI 调整

作为第一步，我们来对现有的 UI 进行一些“善后”工作。现在，PokemonList 和 SettingView 被包装在了各自的 NavigationView 中，它们仍然是独立存在的。在示例 app 里，我们希望通过屏幕下方的 Tab，在两个导航栈之间进行切换。在 SwiftUI 里，这种 Tab 切换使用 TabView 进行定义。

新建一个 SwiftUI View 文件“MainTab.swift”，编辑其中内容，为 TabView 定义两个页面，PokemonRootView 和 SettingRootView：

```
struct MainTab: View {
    var body: some View {
        TabView {
            // 1
            PokemonRootView().tabItem {
                Image(systemName: "list.bullet.below.rectangle")
                Text("列表")
            }
        }
    }
}
```

```
SettingRootView().tabItem {
    Image(systemName: "gear")
    Text("设置")
}
}

// 3

.edgesIgnoringSafeArea(.top)
}
```

1. `tabItem` 定义了 `TabView` 上对应 `Tab` 里应该显示的图片和文字。在 `tabItem` 里，只有 `Image` 和 `Text` 是被接受的，其他类型的 `View` 将被忽视。
2. 希望你还记得 iOS 13 开始系统为我们提供的 SF Symbol 图标。在 `Tab` 上显示的图标是它的一个绝佳使用场景。
3. `TabView` 默认会尊重 `safe area` 的顶部，这会导致 `TabView` 里的宝可梦列表在滚动时无法达到“刘海屏”上部的状态栏，这不是我们需要的。使用 `.edgesIgnoringSafeArea(.top)` 忽略掉 `safe area`，让界面占满屏幕。

接下来，把“`SceneDelegate.swift`”里的 `ContentView` 替换成 `MainTab`，运行 app 我们就可以在宝可梦列表和设置表单两个主要界面之间切换了：



有一个很明显的问题，弹出面板被 TabView 遮挡住了。这是因为弹出面板现在是作为 PokemonList 的 overlay 存在的，我们可以通过将这个 overlay 移到 TabView 的层级，来让它在最上层弹出。在下一章我们实际处理列表 cell 点击和面板弹出时再来处理这个问题。本章中，我们简单地将它移出界面。在 PokemonList 的 body 里，注释掉 overlay 的部分：

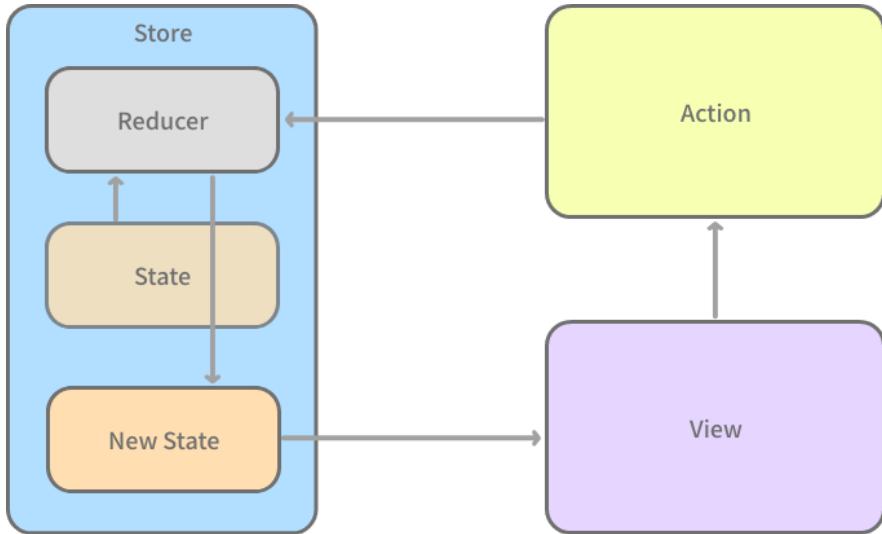
```
struct PokemonList: View {  
    var body: some View {  
        ScrollView {
```

```
        }
        // .overlay(
        //   VStack {
        //     Spacer()
        //     PokemonInfoPanel(model: .sample(id: 1))
        //   }.edgesIgnoringSafeArea(.bottom)
        // )
      }
    }
```

Swift UI 的架构方式

在本书写作时，SwiftUI 还在非常早期的阶段，Swift 技术的狂热者们（包括本书作者在内）都在探索和寻找一种有效的架构方式。Apple 在语言和内置框架层面为开发者提供了 `@State`, `@ObservedObject`（或 `@StateObject`）和 `@EnvironmentObject` 三种不同作用范围的数据订阅方式，并在 SwiftUI 的官方示例项目中进行了一些说明。随着 SwiftUI 版本的更新，相信会有更多的状态管理的方式会被加入进来，但是至今为止，Apple 并没有实际上规定在复杂 app 中这些数据应该以怎样的方式有序流动，官方示例在这些数据的使用场景上也过于简单，缺乏很强的说服力。

在这一节中，我们会介绍一种类似于 Redux，但是针对 SwiftUI 的特点进行了一些改变的数据管理方式。它遵循我们在计算器数据状态和绑定里提到过的这张图：



在这里我想要重复一下这套数据流动方式的特点：

1. 将 app 当作一个状态机，状态决定用户界面。
2. 这些状态都保存在一个 **Store** 对象中。
3. **View** 不能直接操作 **State**，而只能通过发送 **Action** 的方式，间接改变存储在 **Store** 中的 **State**。
4. **Reducer** 接受原有的 **State** 和发送过来的 **Action**，生成新的 **State**。
5. 用新的 **State** 替换 **Store** 中原有的状态，并用新状态来驱动更新界面。

传统 Redux 有两点比较大的限制，在 SwiftUI 中会显得有些水土不服，可能需要一些改进。

首先，“只能通过发送 **Action** 的方式，间接改变存储在 **Store** 中的 **State**”这个要求太过严格。SwiftUI 有着方便和现成的 **Binding** 行为，来完成状态和界面的双向绑定。

使用这个特性可以大幅简化程序的编写，同时保持数据流的清晰稳定。因此，我们希望为状态改变设置一个例外：除了通过 Action 外，也可以通过 Binding 来改变状态。

其次，我们希望 Reducer 具有纯函数特性，但是在实际开发中，我们会遇到非常多带有副作用 (side effect) 的情况：比如在改变状态的同时，需要向磁盘写入文件，或者需要进行网络请求。在上图中，我们没有阐释这类副作用应该如何处理。有一些架构实现选择不区分状态和副作用，让它们混在一起进行，有一些架构选择在 Reducer 前添加一层中间件 (middleware)，来把 Action 进行预处理。我们在 PokeMaster app 的架构中，选择在 Reducer 处理当前 State 和 Action 后，除了返回新的 State 以外，再额外返回一个 Command 值，并让 Command 来执行所需的副作用。

理论阐述到此为止，我们通过实际的例子来看看这套基于 Redux，为 SwiftUI 设计的“Redux for SwiftUI”的运作方式吧。

通过 Binding 改变状态

本节中，我们的目标是创建基本的 Store 和 State 类型，然后用它来驱动设置页面的 Toggle 开关和排序选择，这可以让我们看到如何使用 SwiftUI 提供的 Binding 来修改状态。新建“Store.swift”和“AppState.swift”文件：

```
// AppState.swift

struct AppState {
    // 1
    var settings = Settings()
}

extension AppState {
    // 2
    struct Settings {
}
```

```
enum Sorting: CaseIterable {
    case id, name, color, favorite
}

var showEnglishName = true
var sorting = Sorting.id
var showFavoriteOnly = false
}

}

// Store.swift
```

```
import Combine
// 3

class Store: ObservableObject {
    @Published var appState = AppState()
}
```

1. 在这里，我们选择按照页面的方式把状态组织到各自的名称下。注意，这里的 `Settings` 指的是接下来要定义的 `AppState.Settings`，而不是我们已经定义在 `SettingView` 中的全局的 `Settings`。Swift 在局部 (`AppState`) 中存在同名类型的时候，会优先使用局部的类型。
2. 在 `AppState` 中定义 `Settings`，并把原来的全局 `Settings` 中的选项相关部分的模型移动过来。
3. 将 `Store` 声明为 `ObservableObject`，这样我们就可以在 `View` 里通过 `@ObservedObject` 或者 `@EnvironmentObject` 来订阅它了。

在 SettingView 中，我们添加上 @EnvironmentObject 的 Store 实例，并对选项 section 中所使用的 Binding 进行更新：

```
struct SettingView: View {  
  
    @EnvironmentObject var store: Store  
    var settingsBinding: Binding<AppState.Settings> {  
        $store.appState.settings  
    }  
  
    // ...  
  
    var optionSection: some View {  
        Section(header: Text("选项")) {  
            Toggle(isOn: settingsBinding.showEnglishName) {  
                Text("显示英文名")  
            }  
            Picker(  
                selection: settingsBinding.sorting,  
                label: Text("排序方式"))  
            {  
                ForEach(AppState.Settings.Sorting.allCases, id: \.self) {  
                    Text($0.text)  
                }  
            }  
            Toggle(isOn: settingsBinding.showFavoriteOnly) {  
                Text("只显示收藏")  
            }  
        }  
    }  
}
```

```
    }
}

}
```

为了让代码简短一些，避免重复，我们将 \$store.appState.settings 提取为计算属性 settingsBinding。在 SettingView.swift 里，还需要进行一些改变：

```
// extension Settings.Sorting {
extension AppState.Settings.Sorting {
    var text: String {
        switch self {
            case .id: return "ID"
            case .name: return "名字"
            case .color: return "颜色"
            case .favorite: return "最爱"
        }
    }
}
```

最后，将 Store 设置为 environment object，将它“注入”到 View 的环境中。在 SceneDelegate.swift 中，初始化 MainTab 后，添加上 environmentObject 调用：

```
let contentView = MainTab().environmentObject(Store())
```

运行 app，切换到设置页面，一切应该没有变化。不过在底层，选项部分的三个控件现在是由 AppState 中的模型驱动了。

在最终的 app 中，这些状态会影响宝可梦列表的排序和显示内容，我们会在之后实际实现列表时再看到它们。

因为 SettingView 现在需要我们传入一个环境对象 Store，而在定义预览的部分我们并没有传入一个可用的 Store，所以现在预览会崩溃。我们可以在 SettingView_Previews 中生成 SettingView 后也加上类似的 environmentObject 调用，来让预览功能重新工作：

```
SettingView().environmentObject(store())
```

除此之外，我们可以在创建 Store 后，对它进行配置后再传入。这样可以帮助我们确认在不同模型的情况下 UI 的状态：

```
static var previews: some View {
    let store = Store()
    store.appState.settings.sorting = .color
    return SettingView().environmentObject(store)
}
```

通过 Action 改变状态

整理现有代码

接下来我们来看看设置页面里注册和登录的部分，我们会同样的方式将数据绑定到登录注册切换、用户邮箱、密码及密码验证这些操作上。在点击登录或者注册按钮时，通过 Action 来对实际的流程进行模拟，并改变用户的登录状态。

在上面的小节中，我们已经把一部分状态从原来的 Settings 里移动到了 AppState.Settings 中。对于用户输入，我们也做类似的操作。在 AppState.swift 里，向 Settings 添加如下枚举和变量：

```
extension AppState {
```

```
struct Settings {

    enum AccountBehavior: CaseIterable {
        case register, login
    }

    var accountBehavior = AccountBehavior.login
    var email = ""
    var password = ""
    var verifyPassword = ""

}

}
```

同样地，由于原来的 `Settings.AccountBehavior` 现在被定义到了 `AppState` 中，所以在 `SettingView.swift` 里，需要作出对应的改变：

```
// extension Settings.AccountBehavior {

extension AppState.Settings.AccountBehavior {

    var text: String {
        switch self {
            case .register: return "注册"
            case .login: return "登录"
        }
    }
}
```

现在，我们可以把之前定义的 class `Settings: ObservableObject` 整个去掉，并且在 `SettingView` 里用 `store.appState.settings` 来替代原来的 `@ObservedObject` 了。除此之外，我也还需要把所有的 `$settings` 都替换为 `settingsBinding`：

```
// Settings 已经被全部移动到 `AppState` 中
```

```
/*
class Settings: ObservableObject {
    // ...
}

*/
struct SettingView: View {
    // ...

    // @ObservedObject var settings = Settings()
    var settings: AppState.Settings {
        store.appState.settings
    }

    // ...

    var accountSection: some View {
        Section(header: Text("账户")) {
            Picker(
                selection: settingsBinding.accountBehavior,
                label: Text(""))
            {
                ForEach(
                    AppState.Settings.AccountBehavior.allCases,
                    id: \.self)
                {
                    Text($0.text)
                }
            }
        }
    }
}
```

```
.pickerStyle(SegmentedPickerStyle())
TextField("电子邮箱", text: settingsBinding.email)
SecureField("密码", text: settingsBinding.password)
if settings.accountBehavior == .register {
    SecureField("确认密码", text: settingsBinding.verifyPassword)
}
Button(settings.accountBehavior.text) {
    print("登录/注册")
}
}

// ...
}
```

添加用户模型

对于未登录用户，我们希望显示注册和登录的表单。但对于已经登录的用户，作为替代，需要在同样位置显示用户邮箱以及一个注销按钮。这需要我们以某种形式持有用户信息，为此，新建一个 User 类型：

```
// 1
struct User: Codable {
    var email: String
    // 2
    var favoritePokemonIDs: Set<Int>
    // 3
    func isFavoritePokemon(id: Int) -> Bool {
        favoritePokemonIDs.contains(id)
    }
}
```

```
}
```

```
}
```

1. 这是一个 Codable 类型，我们稍后会通过一个虚拟的请求获取数据，并将 User 实例序列化为 JSON 保存在磁盘上，这样我们就不需要每次都登录了。 Codable 不是本章的重点，但它可以让这件事情非常方便。
2. 我们同时还保存了一个 favoritePokemonIDs 集合，用户使用宝可梦列表中的“最爱”按钮，可以将对应的宝可梦添加到这个集合中。
3. 列表中 cell 里的“最爱”按钮会依据用户状态显示不同颜色，`isFavoritePokemon` 帮助获取最爱的状态。

在 AppState.Settings 里，我们可以添加 `loginUser` 状态，它是一个可选值类型 `User?`，当用户没有登录时，其值为 `nil`：

```
extension AppState {  
    struct Settings {  
        // ...  
        var loginUser: User?  
    }  
}
```

在 SettingView 里，现在我们可以通过 `loginUser` 的有无，来选择是要显示注册/登录表单，还是显示用户信息和注销按钮了：

```
struct SettingView: View {  
    // ...  
    var body: some View {  
        Form {  
            Section(header: Text("账户")) {
```

```
// 1
if settings.loginUser == nil {
    Picker(
        /* ... */
    )
    Button(settings.accountBehavior.text) {
        // 2
        print("登录/注册")
    }
} else {
    // 3
    Text(settings.loginUser!.email)
    Button("注销") {
        print("注销")
    }
}
// ...
}
```

1. `loginUser` 不存在，当前用户没有登录，显示之前创建的含有用户邮箱，密码的表单。
2. 现在我们在按下按钮时只进行了简单的打印。在下一节中，我们会在这里发送一个 Action 去影响 app 的状态变化。
3. 在 `loginUser` 存在时，显示用户 `email` 和注销按钮。

发送并处理 Action

因为 `settingsBinding` 的存在，`AppState.Settings` 中的 `accountBehavior`, `email`, `password` 这些属性已经和 UI 联动了，现在我们来处理按下登录按钮时的行为。按照我们的原则，UI 不能直接改变 `AppState`，而需要通过发送 Action 并被 Reducer 处理。我们先来定义 Action 的部分。

新建 Swift 文件，命名为“`AppAction.swift`”，并添加以下内容：

```
enum AppAction {
    case login(email: String, password: String)
}
```

在“`Store.swift`”里，我们需要添加两个内容。首先是 Reducer 角色：它负责处理某个 `AppAction`，并返回新的 `AppState`。在这里，我们用一个静态的纯函数来完成操作：

```
class Store: ObservableObject {
    // ...
    static func reduce(
        state: AppState,
        action: AppAction
    ) -> AppState
    {
        var appState = state
        // ...
        switch action {
            case .login(let email, let password):
```

```
// 2

if password == "password" {
    let user = User(email: email, favoritePokemonIDs: [])
    // 3
    appState.settings.loginUser = user
}

}

return appState
}

}
```

1. 由于我们选用了 enum 作为 AppAction 的类型，这可以让我们对 action 使用 switch 语句，编译器会帮助我们保证所有的 AppAction 都得到了处理。
2. 这里做了一些“虚假”的检查，只有密码正确才允许登录。之后，我们会把这部分替换成异步的更接近于真实情况的登录逻辑。
3. 登录成功，设置 appState 中的 loginUser，并返回这个新的 appState。

其次，是可以让 View 调用的用于表示发送了某个 Action 的方法。在这个方法中，我们将当前的 AppState 和收到的 Action 交给 reduce，然后把返回的 AppState 设置为新的状态。

```
class Store: ObservableObject {

@Published var appState = AppState()

func dispatch(_ action: AppAction) {
    #if DEBUG
    print("[ACTION]: \(action)")
    #endif
    appState.reduce(action)
}

}
```

```
#endif

let result = Store.reduce(state: appState, action: action)
appState = result

}

// ...

}
```

在 `dispatch` 方法中，我们打印了接收到的 Action，这可以帮助我们在调试的时候监视具体收到了那些 Action。

最后，在 `SettingView` 中按下登录按钮时，把刚刚定义的 `.login` Action 发送给 store：

```
Section(header: Text("账户")) {

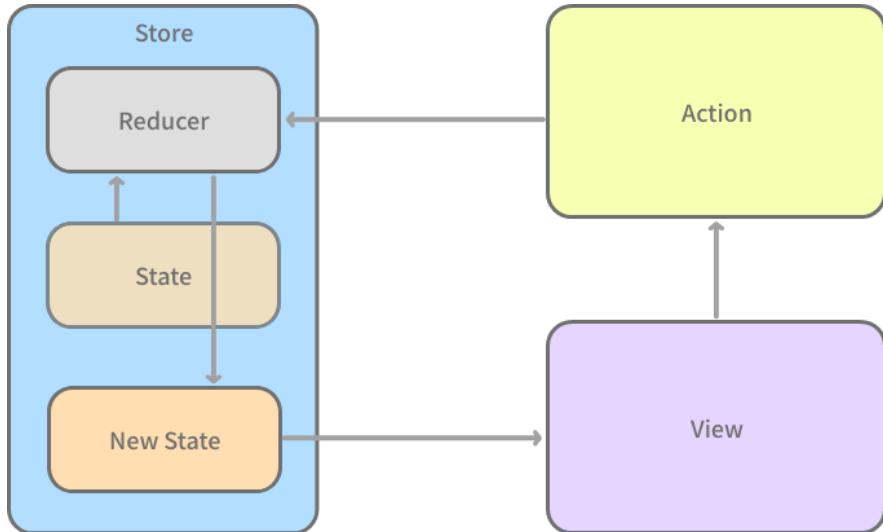
    // ...

    Button(settings.accountBehavior.text) {

        self.store.dispatch(
            .login(
                email: self.settings.email,
                password: self.settings.password
            )
        )
    }
}
```

尝试运行 app，输入任意的用户邮箱和“password”作为密码，并按下登录按钮。`action` 通过 `dispatch` 传达给 `store`，`store` 依据当前 `state` 和收到的 `action` 产生新的 `state` (它含有当前登录用户 `loginUser` 的状态)。由于新的 `appState` 是一个由 `@Published` 所修饰的变量，它的更新将触发所有 (使用 `@ObservedObject` 或者 `@EnvironmentObject`) 对 `Store` 进行观察的 `View` 进行更新。

这样一来，我们完整地完成了一个状态更新的事件流：



为什么选择单向数据流

对于这个事件流周期一个常见的疑问是，为什么我们需要这么麻烦，使用 Action 去间接地改变状态。难道不能在按钮点击的时候像下面这样把修改模型的逻辑放进来么？这样岂不是逻辑更加直接和明确？比如：

```
Button(settings.accountBehavior.text) {
  if self.settings.password = "password" {
    self.store.appState.settings.loginUser
      = User(
        email: self.settings.email,
        favoritePokemonIDs: []
      )
  }
}
```

诚然，这样的做法比走一圈 Action 要直接得多，但是这在三个维度上存在风险。

1. 首先，将类似密码验证和登录这样的具体逻辑放到 View 中，这让 Model 和 View 形成了高度的耦合。随着我们把越来越多的逻辑放在 View 中，最终也必然导致 View 的功能和职责越趋复杂。除了定义样式布局之外，它还需要承担修改 app 状态和组织逻辑的工作，最终出现类似 Massive View Controller 那样的庞大化问题。
2. 更重要的是，我们对于 AppState 的修改将分散在 app 各处：对于同一个属性的修改可能会在不同的地方进行；某些属性的修改可能对另外的一些属性产生影响；在状态逐渐复杂时，对现有逻辑的修改往往需要在 app 各个 View 代码之间跳转和来回确认。很快 app 状态的复杂程度和维护难度都将超过人类极限，这是绝大部分 bug 产生的来源，会导致项目开发难以持续。
3. 最后，将逻辑代码放在 View 中，导致这部分逻辑难以测试。想要对它进行测试，我们需要写一段代码来模拟按钮的点击，然后判断写在 View 的事件里的逻辑是否正确，这往往不容易。但是如果我们把单纯的状态变化逻辑统一放到 Reducer 中，那单元测试就将轻而易举。通过构建需要的 AppState，我们可以以任意 app 状态作为测试的起始。利用完全不依赖于 UI 视图层行为的 app 状态，通过调用没有副作用的 reducer 方法，我们就可以用测试覆盖所有的使用情况。

通过 Action 间接改变状态，在短时间和状态很简单 app 中，似乎有点“得不偿失”。但是在长期和复杂的情况下，它的优势将非常明显。我们在书中的后续章节会不断在这个架构的基础上进行补充和扩展，来进一步说明它的优势。

Command 和异步操作

异步操作的 Action

在上一节的用户登录的示例中，检查用户密码和设置 `loginUser` 属性这些操作都是以同步方式立即完成的，但这显然和真实情况相去甚远。实际上，我们可能需要将用户的邮箱和密码通过网络请求发送给服务器，并由服务器完成用户验证，并把结果返回给 app，这些都涉及到异步操作。

对于一个异步操作，一般来说我们比较关注**两个时间点**。首先是**异步操作开始**的时候，我们可能希望在此时显示像是“正在加载”的界面，让用户知道正在进行一项耗时操作。另一个时间点是**操作完成时**，这时候我们可以使用异步操作的结果（比如网络请求返回的数据）来更新界面。因此，一个异步操作一般会对应两个 State：一个代表操作开始，app 进入等待状态；另一个代表操作结束，可以按照需要更新 UI。Reducer 负责返回新的 State，对于像网络请求这种耗时的异步操作，我们不可能阻塞线程去等待请求完成再返回新状态，因此，我们需要一种另外的方式来处理异步操作，让它在一次请求中拥有两次更新状态的机会。

在前面，我们已经多次重申过，Reducer 的唯一职责应该是计算新的 State，而发送请求和接收响应，显然和返回新的 State 没什么关系，它们属于设置状态这一操作的“副作用”。在我们的架构中我们使用 Command 来代表“在设置状态的同时需要触发一些其他操作”这个语境。Reducer 在返回新的 State 的同时，还返回一个代表需要进行何种副作用的 Command 值（对应上一段中的第一个时间点）。Store 在接收到这个 Command 后，开始进行额外操作，并在操作完成后发送一个新的 Action。这个 Action 中带有异步操作所获取到的数据。它将再次触发 Reducer 并返回新的 State，继而完成异步操作结束时的 UI 更新（对应上一段中的第二个时间点）。

接收和处理 Command

光说不练假把式，我们实际操作看看如何把上面的用户登录转换为异步操作吧。

定义异步操作

首先我们定义一个类型，来模拟用户登录的网络请求。这个类型使用 Combine 框架中的 Future 模拟网络延迟。新建一个“LoginRequest.swift”文件：

```
import Foundation
import Combine

struct LoginRequest {
    let email: String
    let password: String

    // 1
    var publisher: AnyPublisher<User, AppError> {
        Future { promise in
            // 2
            DispatchQueue.global()
                .asyncAfter(deadline: .now() + 1.5)
            {
                if self.password == "password" {
                    let user = User(
                        email: self.email,
                        favoritePokemonIDs: []
                    )
                    promise(.success(user))
                } else {

```

```
        promise(.failure(.passwordWrong))
    }
}
}
// 3
.receive(on: DispatchQueue.main)
// 4
.eraseToAnyPublisher()
}
}
```

1. 使用一个 Publisher 来发布登录操作的在未来的可能结果。当登录成功时，可以得到一个 User 值，否则给出 AppError。我们马上会看到 AppError 的定义。
2. 把新建的 Future Publisher 发送到后台队列，并延时 1.5 秒执行。这用来模拟网络请求的延时状况。
3. 因为我们会想用这个 Publisher 的值更新 UI，所以我们指定后续的接收者应该在主队列接收事件。
4. 我们不关心变形后的 Publisher 的具体类型，所以将它的类型抹掉。

在上面我们定义了 AppError 类型来代表 app 中可能发生的错误，现在我们只有一种可能的错误，那就是密码不正确。新建“AppError.swift”文件，添加下面的内容：

```
import Foundation

// 1

enum AppError: Error, Identifiable {
// 2
var id: String { localizedDescription }
```

```
case passwordWrong
}

// 3

extension AppError: LocalizedError {
    var localizedDescription: String {
        switch self {
            case .passwordWrong: return "密码错误"
        }
    }
}
```

1. 除了 Error 外，我们还顺便让 AppError 遵守了 Identifiable 协议。对于 LoginRequest 本身来说，Identifiable 并不是必须的。但我们在接下来的错误处理一节中会需要用到 Identifiable。
2. 在这里，为了简化，我们使用了 localizedDescription 作为 id。实际中更推荐为每个错误定义自定义 error code，并使用这个唯一不变的数字作为 id。
3. localizedDescription 是 LocalizedError 协议的一部分。通过为错误添加本地化描述，我们可以显示对用户友好的错误信息。

定义 Command

在 Reducer 负责返回新的 State 的同时，我们还希望它同时返回一个 Command 来表示“需要执行的副作用”。新建“AppCommand.swift”文件，我们用一个协议来表示它：

```
import Foundation
import Combine
```

```
protocol AppCommand {
    func execute(in store: Store)
}
```

这个协议定义了一个唯一的方法 `execute(in:)`，它是开始执行副作用的入口。参数 `Store` 则提供了一个执行后续操作的上下文，让我们可以在副作用执行完毕时，继续发送新的 Action 来更改 app 状态。在 `AppCommand` 定义下方，我们创建一个遵守该协议的具体类型 `LoginAppCommand`，它使用 `LoginRequest` 发送一个登录请求，并处理结果：

```
struct LoginAppCommand: AppCommand {
    let email: String
    let password: String

    func execute(in store: Store) {
        LoginRequest(
            email: email,
            password: password
        ).publisher
        // 1
        .sink(
            receiveCompletion: { complete in
                if case .failure(let error) = complete {
                    // 2
                }
            },
            receiveValue: { user in
                // 3
            }
        )
    }
}
```

```
    }
)
}
}
```

1. `LoginRequest.publisher` 是我们刚才定义好的模拟登录操作的 Publisher，我们使用 `sink` 进行订阅。
2. 当错误发生时，`receiveCompletion` 将会被调用，闭包的传入参数是一个描述错误的 `.failure` 成员。使用 `if case` 我们可以在过滤出这个 `case` 的同时，将关联的 `error` 值提取出来。在这里，我们需要通过向 `store` 发送 Action 来显示错误。
3. 登录成功时我们可以收到 Publisher 发出的 `User` 值。类似上面，我们也需要通过 Action 来改变 State。

避免提前取消

编译项目的话，Xcode 会给你一个警告：在 `execute(in:)` 里，我们并没有使用到通过 `sink` 订阅的 `LoginRequest publisher` 所返回的值。这个返回值是一个 `AnyCancellable`，在它被释放时，`cancel()` 会被自动调用，导致订阅取消。上面的代码里发生的正是这一情况：因为我们没有存储这个值，它在创建后就立即被释放掉，导致订阅取消。如果我们不想要这个异步操作在完成之前就被取消掉，就需要想办法持有 `sink` 的返回值，直到异步操作完成。为了达到这一点，可以添加一个 `SubscriptionToken` 来持有 `AnyCancellable`：

```
class SubscriptionToken {
    var cancellable: AnyCancellable?
    func unseal() { cancellable = nil }
}
```

```
extension AnyCancellable {
    func seal(in token: SubscriptionToken) {
        token.cancellable = self
    }
}
```

AnyCancellable extension 上的 seal 会把当前的 AnyCancellable 值 “封印” 到 SubscriptionToken 中去。对上面 execute(in:) 里的代码进行一些改造：

```
func execute(in store: Store) {
    // 1
    let token = SubscriptionToken()
    LoginRequest(
        email: email,
        password: password
    ).publisher
    .sink(
        receiveCompletion: { complete in
            if case .failure(let error) = complete {
                }
            // 3
            token.unseal()
        },
        receiveValue: { user in
            }
    )
    // 2
    .seal(in: token)
}
```

1. 创建一个 `SubscriptionToken` 值备用，它需要存活到订阅的异步事件结束。
2. 在 `sink` 订阅后，把返回的 `AnyCancellable` 存放到 `token` 里。
3. 调用 `token` 的 `unseal` 方法将 `AnyCancellable` 释放。在这里，`unseal` 中里将 `cancellable` 置为 `nil` 的操作其实并不是必须的，因为一旦 `token` 离开作用域被释放后，它其中的 `cancellable` 也会被释放，从而导致订阅资源的释放。这里的关键是利用闭包，让 `token` 本身存活到事件结束，以保证订阅本身不被取消。

对于内部含有异步操作，并使用 `Combine` 来处理的 `AppCommand`，我们都可以通过类似的方式来维持订阅，这可以让我们以最小限度使用资源的同时，保持一个相对干净的写法。`Combine` 框架内部为 `AnyCancellable` 准备了 `store(in:)` 方法，用来将自身存储到一个 `Array` 或者 `Set` 中。我们在后面的章节会看到它们的例子，而这里的 `seal(in:)` 正是借鉴了 `Combine` 的处理方式。

设定和执行 Command

我们暂时把上面的 2 和 3 放一边，先来看看如何改造 `Store`，让它能够处理 `AppCommand`。在 `Store.swift` 中，将 `reduce(state:action:)` 的返回值从 `AppState` 改为 `(AppState, AppCommand?)`，然后更新这个方法的内容：

```
static func reduce(  
    state: AppState,  
    action: AppAction  
) -> (AppState, AppCommand?)  
{  
    var appState = state  
    var appCommand: AppCommand?  
  
    switch action {
```

```
case .login(let email, let password):
    // 1
    guard !appState.settings.loginRequesting else {
        break
    }
    appState.settings.loginRequesting = true
    // 2
    appCommand = LoginAppCommand(
        email: email, password: password
    )
}

return (appState, appCommand)
}
```

1. 我们在本节一开始就强调过，对于一个异步操作来说，有两个时间点比较关键，其中一个就是操作开始的时候。在 AppState.Settings 中，我们添加一个 Bool 变量 loginRequesting 来表示登录请求是否正在进行中。为了避免重复请求，在收到 .login Action 时，先检查是否已经在登录过程中。如果可以继续登录，则把这个值置为 true。
2. 除了改变 loginRequesting 这个 State 值，我们还想要实际发送登录请求。这可以通过把 appCommand 设置为 LoginAppCommand 来触发。

因为 reduce 现在返回一个多元组，所以 dispatch(_:) 方法也需要一些更新。除了设置新的 state 以外，当 AppCommand 存在时，我们需要执行它。修改 dispatch(_:) 的内容：

```
func dispatch(_ action: AppAction) {
    #if DEBUG
```

```
print("[ACTION]: \(action)")

#endif

let result = Store.reduce(state: appState, action: action)
// 1

appState = result.0
// 2

if let command = result.1 {
    #if DEBUG
    print("[COMMAND]: \(command)")
    #endif
    command.execute(in: self)
}

}
```

1. `result` 的类型不再是 `AppState`，而是 `(AppState, AppCommand?)`。对 `appState` 的设置要用到的是新返回的多元组中的第一个值。
2. 检查如果 `AppCommand` 存在的话，就调用 `execute(in:)` 来执行它。和 `ACTION` 一样，我们也在调试期间把待执行的 `COMMAND` 打印出来，方便我们观测发生了什么。

异步 Command 结果

异步操作的第二个关键时间点是操作结束，希望使用得到的结果更新 UI 时。在我们的例子中，这对应着 `LoginAppCommand` 里 `receiveCompletion` 和 `receiveValue` 两个回调。和 `View` 中不能直接更改 `State`，而是使用 `Action` 的规则一样，在 `Command` 里我们也会向 `Store` 发送一个 `Action` 来修改状态。在 `AppAction` 里，新加一个 `enum` 成员：

```
enum AppAction {
```

```
// ...
case accountBehaviorDone(result: Result<User, AppError>
}
```

现在，编译器会在 Store 的 reduce 方法中给出一个错误，这将帮助我们定位到合适的位置并添加对这个 Action 的处理。在这里，我们检查 .accountBehaviorDone 的关联值，如果用户正确登录，我们设置 loginUser：

```
static func reduce(
    state: AppState,
    action: AppAction
) → (AppState, AppCommand?)
{
    // ...
    case .accountBehaviorDone(let result):
        // 1
        appState.settings.loginRequesting = false
        switch result {
            case .success(let user):
                // 2
                appState.settings.loginUser = user
            case .failure(let error):
                // 3
                print("Error: \(error)")
        }
    }

    return (appState, appCommand)
}
```

1. 在 .login Action 里，我们把 loginRequesting 设为了 true，.accountBehaviorDone 代表这个异步请求完成，所以不论结果如何，我们都将这个值重置回 false。
2. 如果正确登录，LoginAppCommand 将会给回有效的 User 值，我们通过将它设定给 loginUser，可以让 View 正确显示已登录用户的信息。
3. 对于错误的情况，现在只是将错误描述打印出来。我们会在下一节实现弹出提示对话框作为简单的错误处理。

现在，我们可以把 .accountBehaviorDone 添加到 LoginAppCommand 里了：

```
struct LoginAppCommand: AppCommand {  
    // ...  
    func execute(in store: Store) {  
        // ...  
        .sink(  
            receiveCompletion: { complete in  
                if case .failure(let error) = complete {  
                    store.dispatch(  
                        .accountBehaviorDone(result: .failure(error))  
                    )  
                }  
                token.unseal()  
            },  
            receiveValue: { user in  
                store.dispatch(  
                    .accountBehaviorDone(result: .success(user))  
                )  
            }  
        )  
    }  
}
```

```
.seal(in: token)  
}  
}
```

更新 View

现在，运行 app 并尝试使用任意用户名和 “password” 作为密码登录，在等待在一定时间后，登录表单将显示为“已登录”状态。为了在耗时异步操作时改善用户体验，我们在上面引入了 `loginRequesting` 这个新状态。在 `SettingView` 中，我们可以用它来显示一些提示信息，告诉用户正在执行异步的登录操作：

```
struct SettingView: View {  
    // ...  
  
    var accountSection: some View {  
        Section(header: Text("账户")) {  
            if settings.loginUser == nil {  
                Picker // ...  
                TextField("电子邮箱") // ...  
                SecureField("密码") // ...  
  
                // ...  
  
                if settings.loginRequesting {  
                    // 1  
                    Text("登录中 ... ")  
                } else {  
                    // 2  
                    Button(settings.accountBehavior.text) {  
                        self.store.dispatch(  
                            .login(  
                                .withEmail("...")  
                                .withPassword("..."))  
                        )  
                    }  
                }  
            }  
        }  
    }  
}
```

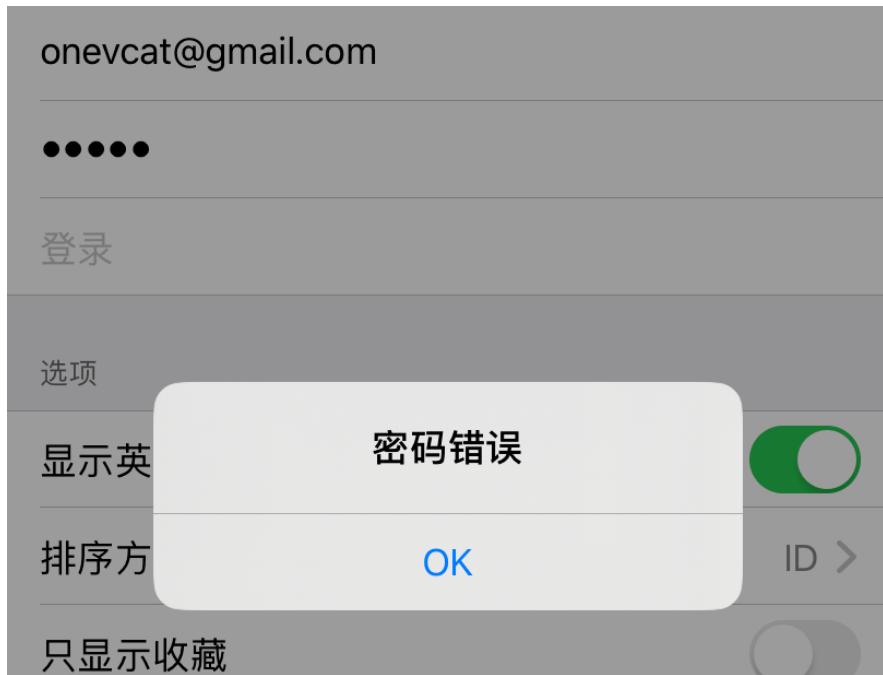
```
        email: self.settings.email,  
        password: self.settings.password  
    )  
)  
}  
}  
}  
}  
} else {  
    // ...  
}  
}  
}  
}
```

1. 当 loginRequesting 为 true 时，显示一个“登录中...”的文本信息，让用户知道确实发出了登录请求。
2. 原来的 Button 只在 loginUser 不存在以及 loginRequesting 为 false 时才显示。



错误处理

正常的异步请求流程已经完成，接下来就是开发者们“最喜爱”的错误处理环节了。现在，当密码错误时，我们只是在控制台将它打印出来。在实际使用中，用户可能等期望看到一个弹出对话框，来告知用户密码错误：



弹框是 UI 的一部分，UI 由 State 驱动。因此，我们考虑在 AppState.Settings 里添加上对应的状态 loginError：

```
extension AppState {  
    struct Settings {  
        // ...  
    }  
}
```

```
    var loginError: AppError?  
}  
}  
}
```

在 Store 的 `reduce(state:action:)` 中，处理 `.accountBehaviorDone` 时，除了在成功时设置 `loginUser` 以外，在失败时，我们将 `error` 设置给 `loginError`:

```
// ...  
  
case .accountBehaviorDone(let result):  
    appState.settings.loginRequesting = false  
    switch result {  
  
        case .success(let user):  
            appState.settings.loginUser = user  
  
        case .failure(let error):  
            appState.settings.loginError = error  
    }  
// ...
```

在 `SettingView` 中，使用 `.alert` 来弹出一个 `Alert` 对话框。`alert(item:content:)` 接受一个 `Identifiable?` 的绑定，当这个绑定值不为 `nil` 时，显示一个弹框。用户在点击弹框按钮 `dismiss` 弹框时，绑定值将被置回 `nil`。这正好满足我们使用 `loginError` 来驱动弹窗的需求。我们在 `SettingView` 的 `body` 最后添加相关代码：

```
struct SettingView: View {  
    var body: some View {  
        Form {  
            accountSection  
            optionSection
```

```
        actionSection  
    }  
    .alert(item: settingsBinding.loginError) { error in  
        Alert(title: Text(error.localizedDescription))  
    }  
}  
}
```

alert 还有一种接受 Binding<Bool> 的变形：alert(isPresented:content:)。当 isPresented 绑定的底层值为 true 时显示弹窗。相比起来，接受 item 的版本可以让我们传递一个更详细的上下文，是相对更强大的版本。

在 UIKit 中，弹窗是由 UIAlertController 负责的，除了示例中展示的弹窗方式以外，还有一种形式，是从屏幕下方弹出选项卡。在 SwiftUI 里，actionSheet 对应的就是选项卡形式的 UI。和 alert 类似，它也接受 isPresented: Binding<Bool> 和 item: Binding<Identifiable?> 两种配置形式。

纯副作用

关于登录，我们还有最后一个问题需要处理：现在用户登录和 loginUser 的状态只在内存中暂存。当我们重启 app 时，之前已登录的用户又将回到未登录的状态。想要在 app session 之间维持登录状态，我们需要把 loginUser 进行持久化，也就是写到硬盘上，你可以将它保存到 UserDefaults、Keychain 或者某个自定义文件中。

对于 Reducer 来说，这又是一个和设置 State 无关的副作用。因此最“标准”的做法是，我们定义一个 WriteUserAppCommand 的新 AppCommand 类型，在处理 .accountBehaviorDone 中用户正常登录时，除了设置 loginUser 以外，同时返回一

个 WriteUserAppCommand 实例作为需要执行的副作用。在这个副作用的 execute(in:) 中，完成对 loginUser 的持久化工作。下面是一种可能的实现方式：

```
struct WriteUserAppCommand: AppCommand {
    let user: User
    func execute(in store: Store) {
        try? FileHelper.writeJSON(
            user,
            to: .documentDirectory,
            fileName: "user.json")
    }
}
```

登录时的请求所对应的 LoginAppCommand，在异步行为结束后再次通过 Action 改变了 State。而 WriteUserAppCommand 与它不同，它不会再去改变状态。我们将这类不再改变状态的副作用称为“纯副作用”。最正规的做法自然是给每个副作用都定义自己的 AppCommand，不过，这类“纯副作用”在 app 中会很常见。对每个纯副作用都去定义一个类型，将会导致我们的 app 中存在很多 AppCommand，它们往往不会很复杂，但是却让人烦躁。

在这个架构中，我们规定一个特例，那就是对于“纯副作用”，我们可以考虑通过属性的 didSet 来执行这个副作用，而跳过严格的 Command 流程。像是将 loginUser 写入磁盘或者从磁盘读出这样的任务，就非常符合这个情景。在 AppState.Settings 中，将 loginUser 修改为：

```
var loginUser: User? =
// 1
try? FileHelper.loadJSON(
    from: .documentDirectory,
    fileName: "user.json")
```

```
{  
    didSet {  
        if let value = loginUser {  
            // 2  
            try? FileHelper.writeJSON(  
                value,  
                to: .documentDirectory,  
                fileName: "user.json")  
        } else {  
            // 3  
            try? FileHelper.delete(  
                from: .documentDirectory,  
                fileName: "user.json")  
        }  
    }  
}
```

1. `loginUser` 在初始化时通过 `loadJSON` 从磁盘上的 “`user.json`” 文件中进行读取。这让我们的用户信息可以在 app 重启后保持。
2. 当设置 `loginUser` 时，如果值存在，那么将它序列化并写到文件中。
3. 如果我们将 `loginUser` 设为 `nil` 时，作为纯副作用，我们将 `user.json` 删除。

对代码质量敏锐的读者肯定已经发现了一些问题，那就是在上面的代码中 `.documentDirectory` 和 “`user.json`” 这两个决定持久化文件保存位置的值重复出现了好几次，这无疑增加了出错的风险和未来重构的难度。另外，“设置一个变量值，并且把它的内容持久化” 在未来可能是一个会经常出现的常见操作，针对每个这样的属性都写一遍类似逻辑显然是很愚蠢的，我们需要考虑一种更方便的方式。

在介绍 SwiftUI 数据状态和绑定的时候，我们提到过 `@propertyWrapper`。它可以用 来包装某个属性，并在 `getter` 和 `setter` 被调用时赋予其额外的逻辑。在 SwiftUI 中，类似 `@State`, `@ObservedObject` 实际上都是 `@propertyWrapper` 的具体应用。在这里，从文件中读取数据和将值保存到文件中的操作，恰好是 `@propertyWrapper` 的一个绝佳应用场景。

新建一个“`FileStorage.swift`”文件，创建 `@propertyWrapper` 的 `FileStorage` 类型。它是一个泛型类型，对于满足 `Codable` 的待存储值，通过设定 `wrappedValue` 的方式触发磁盘读写和对应的序列化/反序列化操作：

```
@propertyWrapper
struct FileStorage<T: Codable> {
    var value: T?

    let directory: FileManager.SearchPathDirectory
    let fileName: String

    init(directory: FileManager.SearchPathDirectory, fileName: String)
    {
        value = try? FileHelper.loadJSON(
            from: directory,
            fileName: fileName)
        self.directory = directory
        self.fileName = fileName
    }

    var wrappedValue: T? {
        set {
            value = newValue
        }
    }
}
```

```
if let value = newValue {  
    try? FileHelper.writeJSON(  
        value,  
        to: directory,  
        fileName: fileName)  
} else {  
    try? FileHelper.delete(  
        from: directory,  
        fileName: fileName)  
}  
}  
  
get { value }  
}  
}
```

如果你已经忘记了 `@propertyWrapper` 的工作方式，我强烈建议你回顾前面章节的相关内容。它可以帮助我们简化很多重复劳动，让代码在更简洁的同时，具有更强的表达能力。有了 `FileStorage`，我们现在可以把 `AppState.Settings` 里的 `loginUser` 换个写法了：

```
@FileStorage(directory: .documentDirectory, fileName: "user.json")  
var loginUser: User?
```

它和上面依赖于 `didSet` 的代码达到的效果是相同的，但是现在看起来要舒服多了！

总结

在本章中，我们介绍了一种类似 `Redux` 的 app 架构在 `SwiftUI` 情景中的使用。`SwiftUI` 和 `Combine` 框架提供了基本的数据组织和订阅的方式，那就是在 `View` 中使

用 @State, @ObservedObject 和 @EnvironmentObject 对状态的变更进行监听。但是具体如何使用这些工具，以及应该分别在什么层级上使用它们，Apple 并没有给出更多的强制规定。在架构选择上，我们有一定灵活性。

Redux 在前端领域已经被证明是能够有效降低开发复杂度，有利于维护大规模的项目。本章的架构在 Redux 的思想上，针对 SwiftUI 的一些特点进行了修改。这只是笔者的一家之言，也还有很多改进的余地。在 SwiftUI 中，究竟应该使用怎样的架构，一直都是 app 开发中被热议的话题；究竟什么样的架构适合一般的 SwiftUI app，大家也还在激烈讨论中。可以预见的是，广大开发者在未来一段时间会更加深入地探索这两个“究竟”。

在后面的章节中，我们会看到更多的关于 State 的使用，包括结合 Combine 来处理多个状态。另外，我们会实际接入 PokeAPI，处理真实的网络请求，并最终完成 PokeMaster app 的列表构建。

练习

1. 登录时使用 UIActivityIndicatorView

当前实现中，在 loginRequesting 为 true 时，我们使用了一个 Text 来表示正在请求：

```
// SettingView.swift

var accountSection: some View {
    // ...
    if settings.loginRequesting {
        Text("登录中 ... ")
    } else {
        // ...
    }
}
```

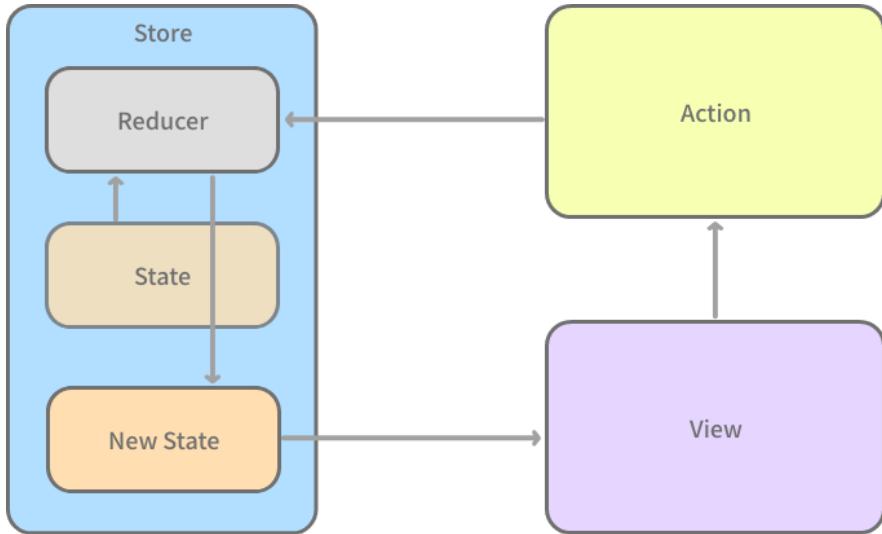
在 iOS 中，用户可能更倾向于看到一个 loading indicator，如下图：



请尝试包装一个 UIActivityIndicator， 并将“登录中...”的 Text 替换成这样的动画的加载指示器。

2. 画出带有 Command 的流程图

Redux 事件流的流程图如下所示，我们已经在本章一开始的部分看到过它了：



不过这张图并不完整，它没有包括处理副作用的 AppCommand 类型。请尝试在这张图上进行补充，在合适的地方添加 Command 角色和相关的箭头，并说明它是如何进一步再次影响 State 的。

3. 实现用户注销功能

我们完成了用户登录的功能，现在你可以用任意用户名或者邮箱，配合“password”作为密码登入到 app 中。这个模拟的用户登录将返回新用户，该返回值会被保存在 loginUser 中，与此同时它也会被写入到磁盘进行存储。在下次启动 app 时，我们从磁盘中拿出这个值，让用户能直接处在已登录状态。

现在的问题是，除非删除 app，否则用户无法注销。在用户已经登录时，SettingView 中有一个注销按钮，但是现在点击注销只是进行了简单的打印：

```

Text(settings.loginUser!.email)
Button("注销") {
    print("注销")
}

```

```
}
```

请按照本章中介绍的架构模式，完成用户注销的功能。注意记住，在我们的架构中，View 是不能直接对 State 进行改动的。

4. 使用 @propertyWrapper 处理设置选项

使用 `@FileStorage` 能把 `loginUser` 持久化的相关代码大幅简化。理论上说，对于任何需要在 `getter` 或者 `setter` 中有重复处理的代码，我们都可以通过 `@propertyWrapper` 来定义特殊“标记”对其简化。

`SettingView` 里像是“显示英文名”、“排序方式”之类的选项，现在是只存储在内存中，不会在 app 重启后被保留的。在 iOS 开发中，对于这类用户设定，如果想要在重启 app 后保存，我们一般使用 `UserDefaults` 来存储它们。我们需要为要存储的值设定一个它在 `UserDefaults` 中的 key，比如：

```
// 读取
let showEnglishName =
    UserDefaults.standard.bool(forKey: "showEnglishName")

// 写入
UserDefaults.standard.set(false, forKey: "showEnglishName")
```

对于本章架构来说，写入 `UserDefaults` 的操作是一个纯副作用，我们完全可以在 State 设定的时候通过 `didSet` 来触发这个副作用。请尝试模仿 `@FileStorage` 的方法，创建一个针对 `UserDefaults` 进行读写的 `property wrapper`，并使用这个新标记来处理 `AppState.Settings` 中的 `showEnglishName` 和 `sorting` 属性。

SwiftUI 中的 Combine

9

SwiftUI 确实激动人心，它将为 Apple 平台的 app 开发方式带来巨大的革新和转变。在图形层，当前 iOS 上的 SwiftUI 基本都是基于 UIKit 的封装，来实现绘制的。而在数据层，新的 Combine 框架则在背后扮演了重要的角色。

在前面的章节中，我们已经看到了一些 SwiftUI 中处理状态和组织数据的方式。它们大都依赖于特定的属性标记，例如 `@State` 或 `@ObservedObject`。无一例外，它们都是通过 Combine 中所提供的特性，来“监视”数据变化，并将这些状态同步到 UI 中去。可以说，这些“专门”为 SwiftUI 设计的异步处理事件的特性，潜在地驱动了整个 SwiftUI 世界的运作。在上一章里，我们利用了这些特定设计了一套 SwiftUI 的 app 架构方式。

在本章中，我们会看看在 SwiftUI app 里用更直接的方式来使用 Combine 的案例。其中包括把已有的状态进行组合的方法，以及如何真正地进行网络请求，并把请求的结果按照特定方式合并，最后用它来驱动 UI 的一些实例。

如果你一直跟随书中的案例，你可以直接用上一章结束时的项目作为开始。你也可以在随书附带的“9.Combine-in-SwiftUI”文件夹的“PokeMaster-Starter”下找到本章的起始代码。

复合状态驱动 UI

在上一章里，我们已经完成了 Setting 界面的部分内容，包括使用绑定直接更改状态、通过 Action 间接更改状态、执行副作用并在副作用结束时再次发送 Action 等。这些规范共同构成了我们的 Redux for Swift 架构的基础，它们的最终目标非常明确，那就是在双向上保证单一数据源的可靠性：即所有影响 UI 的数据都应当来自于 Store，且 UI 的状态应当与 Store 中的状态同步。

不过，有一个重要的话题我们还没有涉及。对于通过 Action 改变的状态，如果我们想要执行网络请求这样的副作用，可以通过同时返回合适的 AppCommand 完成。但

是对于通过绑定来更新的状态，由于不会经过 Store 的 reduce 方法来返回 Command，我们缺少一种有效的手段来在它们改变时执行副作用。

在我们的项目中，一个很典型的例子是用户注册的处理。在 UI 上，我们指明了用户注册时必须使用电子邮箱，但是事实上我们并没有对这个假设进行任何检查和限制：用户可以填写一个普通用户名，而非邮箱地址，来进行注册；也可能虽然填写了一个有效的邮箱地址，但是这个地址已经注册过所以无法再次使用。我们当然可以“无脑”将这些信息在用户点击注册按钮时发送给服务器，然后根据服务器的返回进行错误判断和 UI 提示。但是，如果我们想要提供更好的用户体验，可以在用户还在输入时，就对内容做以下两点检查：

1. 用户的输入是否为有效邮箱地址
2. 这个邮箱地址是否已经在服务器注册过

对于第一点，我们可以在客户端本地进行检查（比如使用正则表达式）。但是对于后一条，只能通过触发向服务器的网络请求来获取相关信息。这就涉及到要如何通过 UI 绑定触发副作用，而这一点现在还是我们的盲区。

最容易想到的方式，当然就是为这类绑定属性添加 didSet，并在它被调用时触发副作用。不过这种做法会有不少局限：我们很难精确控制这个副作用触发的时间和行为，比如对于键盘输入所触发的网络请求，我们会希望加上**防抖 (debounce)** 和**过滤重复元素 (removeDuplicates)**。另外，其他状态的变化也有可能影响当前处理的状态，比如在用户选择登录时我们并不需要对邮箱是否重复进行检查。这类复合状态有可能变得非常复杂，特别是在状态的更新还结合了异步操作的时候，我们有没有一种更好的工具来处理它们呢？答案已经呼之欲出了，那就是 Combine。

创建 Publisher

Publisher 在 Combine 中是一切的源头，为相关属性创建 Publisher 是我们要做的第一步。对于属性值来说，最简单的方式是在属性声明前面加上 @Published 标记。

但是 @Published 需要在内部生成并持有存储，因此我们只能针对定义在 class 里的变量添加 @Published。当前，AppState.Settings 是一个 struct，我们的第一步是要将 email, accountBehavior 提取到 class 中。

在 AppState.Settings 里，添加一个新的内层 class AccountChecker：

```
extension AppState {  
    struct Settings {  
  
        // ...  
  
        // 1  
        // var accountBehavior = AccountBehavior.login  
        // var email = ""  
        // var password = ""  
        // var verifyPassword = ""  
  
        // 2  
        class AccountChecker {  
            @Published var accountBehavior = AccountBehavior.login  
            @Published var email = ""  
            @Published var password = ""  
            @Published var verifyPassword = ""  
        }  
  
        // 3  
        var checker = AccountChecker()  
  
        // ...
```

```
}
```

```
}
```

1. 我们在本章正文中只会使用 accountBehavior 和 email，不过在练习里你会用到 password 和 verifyPassword，所以我们把这些注册和登录相关的属性都进行移动。
2. 使用一个 class 来持有这些内容，这样一来，我们就可以将变量声明为 @Published，并使用前缀美元符号 \$ 来获取 Publisher 了。
3. 在 struct Settings 中持有一个 class 类型的 checker，这会导致 Settings 的值语义被破坏。不过因为我们的架构中只会有一份有效的状态，我们并不在意值语义和引用语义的区别。

因为我们破坏了 Settings 的结构，在继续之前，我们需要修改 SettingView 里用到这些属性的地方：比如将 settingsBinding.accountBehavior 改为 settingsBinding.checker.accountBehavior 等。在继续之前，请确保你可以成功编译和运行项目。

状态合并

接下来，将 accountBehavior 和 email 两个状态“合并”，来实现上面我们提到的逻辑。也即：

1. 在本地检测用户输入的是不是有效的邮箱地址。
2. 使用用户的输入访问 API，判断是不是重复的邮箱。我们需要减少请求量，因此对用户输入做防抖和去重。
3. 如果用户在登录界面，而非注册界面时，不需要检查邮箱是否重复。
4. 将这些状态组合起来，变成一个新的状态，用它去驱动 UI。

因为 4 中这个新的状态是和用户输入以及网络请求相关的异步状态，它会随着用户操作不断改变，显然它应该是一个 Publisher。在 AccountChecker 里添加上：

```
class AccountChecker {

    // ...

    // 1
    var isEmailValid: AnyPublisher<Bool, Never> {
        // 2
        let remoteVerify = $email
            .debounce(
                for: .milliseconds(500),
                scheduler: DispatchQueue.main
            )
            .removeDuplicates()
            .flatMap { email → AnyPublisher<Bool, Never> in
                let validEmail = email.isValidEmailAddress
                let canSkip = self.accountBehavior == .login

                switch (validEmail, canSkip) {
                    // 3
                    case (false, _):
                        return Just(false).eraseToAnyPublisher()
                    // 4
                    case (true, false):
                        return EmailCheckingRequest(email: email)
                            .publisher
                            .eraseToAnyPublisher()
                }
            }
    }
}
```

```
        case (true, true):
            return Just(true).eraseToAnyPublisher()
        }
    }

    // 5 ...
    return ...
}

}
```

1. `isValidEmail` 是一个验证用户输入的 Publisher。我们稍后会订阅它，并用它来更新 UI。
2. `remoteVerify` 是构成整个 `isValidEmail` 的一部分，它负责调用 Server API 来验证有效性。首先，针对 `$email` 使用 `debounce` 和 `removeDuplicates` 来控制用户输入，它将为我们过滤掉输入抖动和重复输入，这样我们将能尽量减少 API 调用。
3. 如果 `validEmail` 为 `false`，那说明输入的邮箱地址在本地就被验证为无效，就不需要再进一步去发送检查了。
4. 如果本地检查通过，而且我们处于注册页面时，发送 `EmailCheckingRequest` 请求。
5. 把 `remoteVerify` 和其他状态组合起来，返回最终代表 `email` 是否有效的 Publisher。我们会在稍后继续在这里添加内容。

和前面章节的“网络请求”一样，`EmailCheckingRequest` 也是一个延时模拟：

```
struct EmailCheckingRequest {
    let email: String
```

```
var publisher: AnyPublisher<Bool, Never> {
    Future<Bool, Never> { promise in
        DispatchQueue.global().asyncAfter(deadline: .now() + 0.5) {
            if self.email.lowercased() == "onevcat@gmail.com" {
                promise(.success(false))
            } else {
                promise(.success(true))
            }
        }
    }
    .receive(on: DispatchQueue.main)
    .eraseToAnyPublisher()
}
}
```

当 promise(.success(true)) 被调用时，说明检查通过。remoteVerify 再配合本地的检查，返回最终 email 的可用状态：

```
var isEmailValid: AnyPublisher<Bool, Never> {
    let remoteVerify = $email
    // ...

    let emailLocalValid =
        $email.map { $0.isValidEmailAddress }
    let canSkipRemoteVerify =
        $accountBehavior.map { $0 == .login }

    return Publishers.CombineLatest3(
        emailLocalValid, canSkipRemoteVerify, remoteVerify
    )
}
```

```
)  
.map { $0 && ($1 || $2) }  
.eraseToAnyPublisher()  
}  

```

订阅状态

我们在 AccountChecker 中已经添加了 isEmailValid 状态，接下来我们要做的是订阅这个状态，并且通过合适的方式，也就是发送 Action 来更新它。

UI 都应该是 State 驱动的，为了表示用户输入的邮箱是否有效，在 AppState.Settings 里添加一个状态：

```
extension AppState {  
    struct Settings {  
        // ...  
        var isEmailValid: Bool = false  
    }  
}
```

在 UI 中，根据 Settings.isEmailValid 的状态，我们将用户输入邮箱的 Text 颜色做一些改变，这样能让我们直接看到界面上的改变。在 SettingView 的 accountSection 里，为邮箱的 TextField 添加字体颜色：

```
var accountSection: some View {  
    Section(header: Text("账户")) {  
        // ...  
        TextField("电子邮箱", text: settingsBinding.checker.email)  
            .foregroundColor(settings.isEmailValid ? .green : .red)  
    }  
}
```

```
}
```

现在，`Settings.isValid` 在用户输入时没有发生任何改变，所以用户输入会一直是红色。我们更倾向于在 app 开始时就把所有内容设定好，并用声明式和响应式的方法让 app 维护自己的状态，因此，订阅的一个比较好的时机是在 `Store` 的初始化方法中：

```
class Store: ObservableObject {
    init() {
        setupObservers()
    }

    func setupObservers() {
        appState.settings.checker.isValid.sink {
            isValid in
            // ...
        }
    }
}
```

和通过 UI 事件改变状态一样，想要变更 `Settings.isValid`，并以此影响 UI 状态，我们只能通过发送 Action 来进行。在 `AppAction` 里添加一个 case：

```
enum AppAction {
    // ...
    case emailValid(valid: Bool)
}
```

最后，对 AccountChecker.isEmailValid 这个 Publisher 进行订阅，在结果改变时，发送 Action 去修改 Settings.isEmailValid。更新 setupObservers 的内容，并在 reduce 里检查 .emailValid 并变更状态：

```
func setupObservers() {
    appState.settings.checker.isEmailValid.sink {
        isValid in
        self.dispatch(.emailValid(valid: isValid))
    }
}

static func reduce(
    state: AppState,
    action: AppAction
) → (AppState, AppCommand?)
{
    // ...
    switch action {
        // ...
        case .emailValid(let valid):
            appState.settings.isEmailValid = valid
    }
    return (appState, appCommand)
}
```

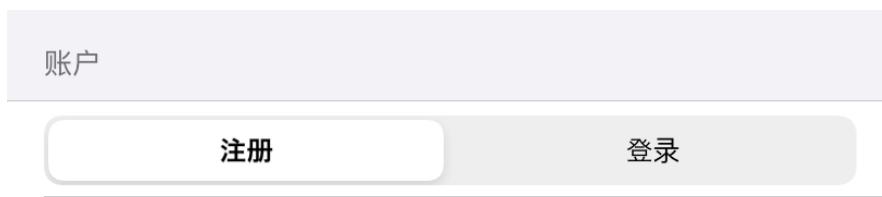
编译并运行，你会发现用户邮箱并不会随着输入而发生变化，对 AccountChecker.isEmailValid 的订阅也没有被调用。可能你已经注意到了编译器其实给了我们警告，对 isEmailValid.sink 的结果，我们并没有把它暂存起来。这个返回的 AnyCancellable 会被立即 deinit，因此 cancel() 也被立即调用，导致订阅失效。

我们可以简单地在 Store 里声明一个成员变量来持有返回值，来避免这个情况发生。不过对于每个订阅都声明一个变量写起来过于麻烦了。Store 的声明周期贯穿于整个 app，而我们也希望这个订阅一直有效。对于这类与“宿主”生命周期绑定，我们没有通过调用 cancel 主动取消的订阅的需求，可以集中一下。可以通过一个 AnyCancellable 的数组来持有这些订阅：

```
var disposeBag = [AnyCancellable]()
```

在 Store 里，把订阅所产生的 AnyCancellable 加到 disposeBag 里，这样订阅就可以正常工作了。未来如果引入更多的订阅，我们也可以用同样的方式来确保订阅一直有效，而不必再引入新的变量来持有它。这个操作非常常见，而 Combine 也为此专门在 AnyCancellable 添加了一个方法，来将订阅产生的 AnyCancellable 加到一个集合中：

```
appState.settings.checker.isEmailValid.sink { isValid in  
    self.dispatch(.emailValid(valid: isValid))  
}.store(in: &disposeBag)
```



当验证正确的时候，邮箱显示的文字将由红转绿。可以特别关注一下本地验证和网络验证的时间差，你也可以考虑在 isEmailValid Publisher 的合适的地方添加 .print()，来检视这个复杂状态下具体的事件发送情况。

实际的网络请求

迄今为止，我们在介绍状态和 app 架构时，都把注意力集中在了 `SettingView` 上。虽然 app 运行起来看上去已经“有模有样”，但是我们涉及到网络请求时，全部都用了模拟的方式，而没有真正从网络 API 获取一丁点儿数据。列表页面中的内容，全部来源于 `app bundle` 内 JSON 里的预先存储的内容。在这一节里，我们会对如何实际用 `Combine` 进行网络请求，并让将它加入到我们已有架构中的方式，进行说明和检视，

使用 `Combine` 的网络请求

在 `Combine` 和 `Foundation` 相关章节里，我们已经介绍了 `URLSession.dataTaskPublisher` 的基本用法。对于一般性的 API，通用的（同时也是比较粗糙的）单个请求的处理方式如下：

```
URLSession.shared  
.dataTaskPublisher(for: entryPoint)  
.map { $0.data }  
.decode(type: Response.self, decoder: decoder)
```

它从 `entryPoint` 获取和下载数据，通过 `.map` 将得到的数据取出，并通过 `.decode` 转换为满足 `Decodable` 协议的 `Response` 的实例。如果 `dataTask` 的网络请求发生错误，或者 `.decode` 无法将数据解码为 `Response`，那么这个 `Publisher` 也将以 `failure` 结束。

现在的项目里，首屏中显示 `List` 的 `View` 是 `PokemonList`，它通过对 `PokemonViewModel.all` 进行 `ForEach` 操作，来生成每行 `cell`。如果你已经忘了这部分内容的话，你可以在 `PokemonList.swift` 找到相关代码，我们摘录如下：

```
struct PokemonList: View {  
    var body: some View {
```

```
ScrollView {  
    // ...  
    ForEach(PokemonViewModel.all) { pokemon in  
        // ...  
    }  
}  
}  
}
```

PokemonViewModel.all 从本地数据读取，对应的 PokemonViewModel 类型持有 Pokemon 和 PokemonSpecies 这两个由 JSON 转换的 data model，并把它们的属性转换为 View 中需要显示的样式。

```
struct PokemonViewModel: Identifiable, Codable {  
    // ...  
    let pokemon: Pokemon  
    let species: PokemonSpecies  
  
    init(pokemon: Pokemon, species: PokemonSpecies) {  
        self.pokemon = pokemon  
        self.species = species  
    }  
}
```

我们在本节中的任务，就是通过 API 请求来获取 Pokemon 和 PokemonSpecies 这两个模型类型对应的数据，并生成全部的 PokemonViewModel，用它们来替换掉现在的本地数据以及更新 UI。

使用 flatMap 和 zip 合并请求

对于大多数自建后端来说，可以在前后端进行协调，让后端 API 的返回值恰好满足前端的需要。



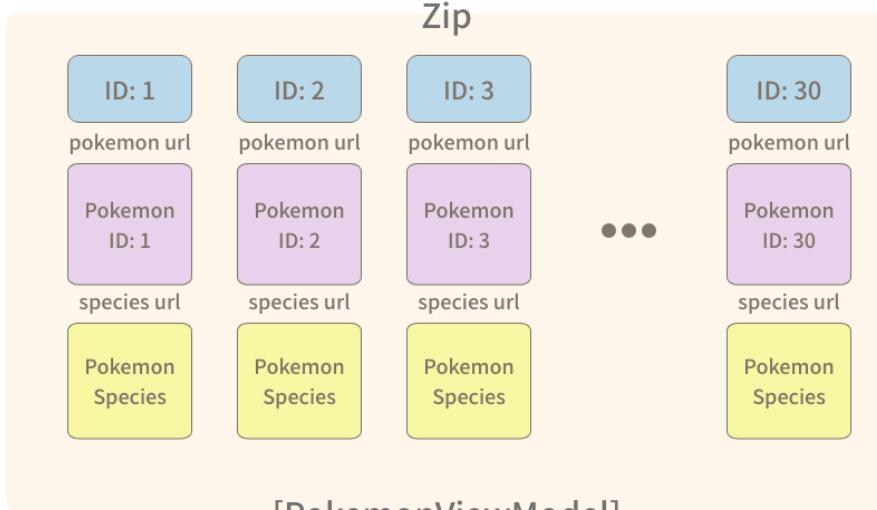
比如，对于这样一个 cell，理想状况应该是 API 返回中包括构建它所需要的所有信息，也就是通过宝可梦的 ID 进行请求，返回中包括宝可梦的中英文名字，主题颜色等信息。理想情况是美好的，但现实总是残酷的：Pokemon 对应的 API 是 “<https://pokeapi.co/api/v2/pokemon/#{id}>” ，这个 API 的返回中只有身高体重（我们在点击 cell 详情，弹出详细信息面板时会用到它们），而不包含名字颜色等信息。这些内容被定义在 species 中，它是另一个 API 的返回值。麻烦之处在于，我们无法直接通过宝可梦的 ID 获取对应 species 的 API 地址，这个地址是被包含在 “/api/v2/pokemon/#{id}” 的结果中的，也就是说，我们需要先对它进行解析，才能得到下一个 API 的 URL：

```
// /api/v2/pokemon/1
{
  "species": {
    "name": "bulbasaur",
    "url": "https://pokeapi.co/api/v2/pokemon-species/1/"
```

```
},
"height": 7,
"weight": 69,
//...
"abilities": [
{
  "ability": {
    "name": "chlorophyll",
    "url": "https://pokeapi.co/api/v2/ability/34/"
  },
  // ...
},
// ...
],
}
```

详细面板中要显示的宝可梦技能 abilities 的情况也类似，在后面我们会把它作为练习留给读者。

因此，想要完整获取到显示一个 cell 所需要的全部信息，我们将需要两次请求：首先根据 ID 去请求 “/api/v2/pokemon/#{id}”，然后根据返回的结果中的 species.url，再对这个 URL 进行一次请求。最后把这两个响应的值组合起来，创建一个 PokemonViewModel。为了显示整个列表，我们需要所有宝可梦的数据：最简单的方式是通过使用 zip 把对所有 ID 的请求结果统合起来，并在这些请求完成后统一地去更新 UI。



实际动手试试看吧。

新建一个“LoadPokemonRequest.swift”文件，在这里我们的最终目标是创建一个“异步版本”的PokemonViewModel.all。不过首先，我们需要从宝可梦的id产生Pokemon的方法：

```
struct LoadPokemonRequest {
    let id: Int

    func pokemonPublisher(_ id: Int) -> AnyPublisher<Pokemon, Error> {
        URLSession.shared
            .dataTaskPublisher(
                for: URL(string: "https://pokeapi.co/api/v2/pokemon/\" + id)!)!
            .map { $0.data }
            .decode(type: Pokemon.self, decoder: appDecoder)
            .eraseToAnyPublisher()
    }
}
```

```
    }
}
```

这就是一个最标准的基于 URLSession 和 dataTaskPublisher 的请求。接下来，是通过 Pokemon 来获取 PokemonSpecies 的请求。为了后续使用方便，我们将 Pokemon 和 PokemonSpecies 一并作为最终值输出：

```
struct LoadPokemonRequest {
    // ...
}

func speciesPublisher(
    _ pokemon: Pokemon
) -> AnyPublisher<(Pokemon, PokemonSpecies), Error>
{
    URLSession.shared
        .dataTaskPublisher(for: pokemon.species.url)
        .map { $0.data }
        .decode(type: PokemonSpecies.self, decoder: appDecoder)
        .map { (pokemon, $0) }
        .eraseToAnyPublisher()
}
}
```

speciesPublisher 依赖于 pokemonPublisher，而它所要求的输入和输出，也决定了基本上我们只有一种方式能使用它，那就是 flatMap。在 LoadPokemonRequest 添加一个计算属性，它将 pokemonPublisher 和 speciesPublisher 组合起来，提供获取 PokemonViewModel 的 Publisher：

```
struct LoadPokemonRequest {
```

```
// ...

var publisher: AnyPublisher<PokemonViewModel, AppError> {
    pokemonPublisher(id)
        .flatMap { self.speciesPublisher($0) }
        .map { PokemonViewModel(pokemon: $0, species: $1) }
        .mapError { AppError.networkingFailed($0) }
        .receive(on: DispatchQueue.main)
        .eraseToAnyPublisher()
    }
}
```

为了统一错误类型，我们在 `AppError` 中添加了一个 `.networkingFailed` 成员，它将网络请求相关的错误“聚合”起来，这样在 UI 中我们处理起来也能简单一些：

```
enum AppError: Error, Identifiable {
    // ...

    case networkingFailed(Error)

}

extension AppError: LocalizedError {
    var localizedDescription: String {
        switch self {
            // ...
            case .networkingFailed(let error):
                return error.localizedDescription
        }
    }
}
```

```
}
```

最后，我们把所有可用的宝可梦 ID (在示例 app 里，我们只显示 ID 为 1~30 的宝可梦) 转换成对应的 Publisher，并用 zip 将它们关联起来。这样一来，在全部数据下载完成后，这个 zip 后的最外层 Publisher 会给出最终结果。

在本书写作时，Combine 里只提供了 Zip，Zip3 和 Zip4 三种方式，最多可以对四个 Publisher 做 zip 操作，这显然没有办法满足我们的要求。我们可以通过自定义一个符合 Publisher 协议的 ZipAll 类型，来把一个 Publisher 序列进行合并。但是本书中我们没有提及如何自定义 Publisher 的内容，相关的话题超出了本书的范围。在示例 app 里，我们用一种相对不那么优雅的方式，来把 zip 逐渐合并，来“模拟”出 zipAll 的行为：

```
extension Array where Element: Publisher {
    var zipAll: AnyPublisher<[Element.Output], Element.Failure> {
        let initial = Just([Element.Output]())
            .setFailureType(to: Element.Failure.self)
            .eraseToAnyPublisher()
        return reduce(initial) { result, publisher in
            result.zip(publisher) { $0 + [$1] }.eraseToAnyPublisher()
        }
    }
}
```

Combine 本身也还是新出现的框架，随着发展，可能今后像 zipAll 这样的方法会直接出现在 Combine 中。不过现在，有了这个简便方法，我们就能最终写出加载全部 PokemonViewModel 的 LoadPokemonRequest.all 了：

```
struct LoadPokemonRequest {
    static var all:
```

```
AnyPublisher<[PokemonViewModel], AppError>

{
    (1 ... 30).map {
        LoadPokemonRequest(id: $0).publisher
    }.zipAll
}
}
```

使用请求结果

最后，剩下的工作就是使用 `LoadPokemonRequest.all` 来获取实际数据，并将它显示出来。和 app 的其他部分一样，我们遵循 UI 事件触发 Action，Action 返回 Command，并在 Command 中执行请求的方式，来处理 `PokemonList` 的加载。

我们先来定义两个 Action，它们分别用来表示开始加载和加载结束的事件：

```
// AppAction.swift

enum AppAction {
    // ...
    case loadPokemons
    case loadPokemonsDone(
        result: Result<[PokemonViewModel], AppError>
    )
}
```

以及执行“加载所有宝可梦”这个副作用的 AppCommand：

```
// AppCommand.swift

struct LoadPokemonsCommand: AppCommand {
    func execute(in store: Store) {
```

```
let token = SubscriptionToken()
LoadPokemonRequest.all
    .sink(
        receiveCompletion: { complete in
            if case .failure(let error) = complete {
                store.dispatch(
                    .loadPokemonsDone(result: .failure(error)))
            }
        }
    )
    token.unseal()
},
receiveValue: { value in
    store.dispatch(
        .loadPokemonsDone(result: .success(value)))
}
)
.seal(in: token)
}
}
```

LoadPokemonsCommand 简单地对 LoadPokemonRequest.all 进行了订阅。不论结果如何，都通过 .loadPokemonsDone 将它传递出去。

为了在 Store 的 reducer 中使用 Action 和 AppCommand，我们还需要在 AppState 里加上持有加载结果的状态。这里创建了一个新的内嵌 struct 来持有列表页面的状态。在 “AppState.swift” 中：

```
struct AppState {
// ...
```

```
var pokemonList = PokemonList()

}

extension AppState {
    struct PokemonList {
        var pokemons: [Int: PokemonViewModel]?
        var loadingPokemons = false

        var allPokemonsByID: [PokemonViewModel] {
            guard let pokemons = pokemons?.values else {
                return []
            }
            return pokemons.sorted { $0.id < $1.id }
        }
    }
}
```

为了今后使用方便，我们希望将宝可梦的信息存储在一个以宝可梦的 ID 为 key 的字典里。同时，我们提供了一个将 `pokemons` 中的值按照 id 排序的属性。

有了上面这些，就可以在 `reducer` 里实际去写逻辑了：

```
static func reduce(
    state: AppState,
    action: AppAction
) -> (AppState, AppCommand?)

{
    // ...
    switch action {
        // ...
    }
}
```

```
// 1
case .loadPokemons:
    if appState.pokemonList.loadingPokemons {
        break
    }
    appState.pokemonList.loadingPokemons = true
    appCommand = LoadPokemonsCommand()

// 2
case .loadPokemonsDone(let result):
    switch result {
        case .success(let models):
            appState.pokemonList.pokemons =
                // 3
                Dictionary(
                    uniqueKeysWithValues: models.map { ($0.id, $0) }
                )
        case .failure(let error):
            // 4
            print(error)
    }
}

}

}
```

1. 当接收到 AppAction.loadPokemons 时，判断是否已经在加载过程中，如果没有开始，则返回 LoadPokemonsCommand。

- 当接收到 `AppAction.loadPokemonsDone` 时，重置 `PokemonList.loadingPokemons` 这个加载状态，如果有事件值，则暂存到 `PokemonList.pokemons` 中。
- `models` 的类型是 `[PokemonViewModel]`。`Dictionary` 的 `init(uniqueKeysWithValues:)` 将一个键值对序列转换为字典，其中键值对的首个元素会被作为 `key`。
- 我们没有进行错误处理，只是将它简单地打印出来。在本章的练习里，你会看到一个关于处理加载错误的题目。

我们已经将所有的状态和逻辑都准备好了，最后只需要在 UI 上使用这些状态来进行显示，并在合适的时候发送 `AppAction.loadPokemons` 就可以了。由于宝可梦列表是 app 的第一个页面，所以页面显示时，就开始请求数据并加载列表。在 “`PokemonRootView.swift`” 里，将它的内容更新为：

```
struct PokemonRootView: View {
    @EnvironmentObject var store: Store

    var body: some View {
        NavigationView {
            if store.appState.pokemonList.pokemons == nil {
                // 1
                Text("Loading ... ").onAppear {
                    self.store.dispatch(.loadPokemons)
                }
            } else {
                // 2
                PokemonList()
                    .navigationBarTitle("宝可梦列表")
            }
        }
    }
}
```

```
    }
}
}
```

1. 当 `appState` 中没有宝可梦数据时，显示一个“Loading...”的 `Text`，并在它出现在屏幕时发送 `.loadPokemons` 开始加载。
2. 当 `.loadPokemonsDone` 成功设置 `pokemonList.pokemons` 后，`PokemonRootView.body` 被重新求值，此时 `PokemonList` 被显示出来。

最后，在“`PokemonList.swift`”中，引入 `@EnvironmentObject`，并使用刚才添加的 `allPokemonsByID` 作为数据源：

```
struct PokemonList: View {
    @EnvironmentObject var store: Store

    var body: some View {
        ScrollView {
            // ...
            ForEach(store.appState.pokemonList.allPokemonsByID) {
                pokemon in
            }
        }
    }
}
```

至此，我们应该已经把宝可梦列表的数据源完全替换为实际的网络数据了。尝试一下再次运行 app，你应该可以看到一段时间的“Loading...”信息，并在全部数据完成加载后看到和原来一样的宝可梦列表。在 `LoadPokemonRequest.all` 里，我们指定

了加到 ID 在 1 至 30 之间的宝可梦。你也可以修改 1...30 这个值，让它加载更多的条目（比如修改为 1...50），以确认数据加载的代码是否真的工作。

你也可以通过观察控制台的输出来确认下载是否正确完成，比如，想定的 Action 是否真的被接收到。需要注意的是，PokeAPI 对每个 IP 有一定访问限制，如果你在短时间内进行了大量请求的话，可能会遭遇失败。这种情况下你可以尝试稍后再试。

示例 app 中列表的数据是从网络 API 请求的，但是现在每个 cell 显示的图片还是从 xcassets 文件夹中获取的。当前本地只有 ID 1 到 30 的图片数据，所以当你尝试加载更多条目时，将无法显示图片。你可以使用类似 API 请求的方法获取图片数据并刷新 UI 进行显示，不过我们会在下一章看到一些其他更简单的异步图片获取方式。

缓存

`LoadPokemonRequest.all` 是一个很“重”的请求，它是若干个请求的复合体。每次打开 app 都进行请求，会对性能造成很大影响。而且对于 Pokemon 的信息，每次请求的结果都不会改变，所以将它缓存起来，避免无意义的重复请求会是一个好办法。

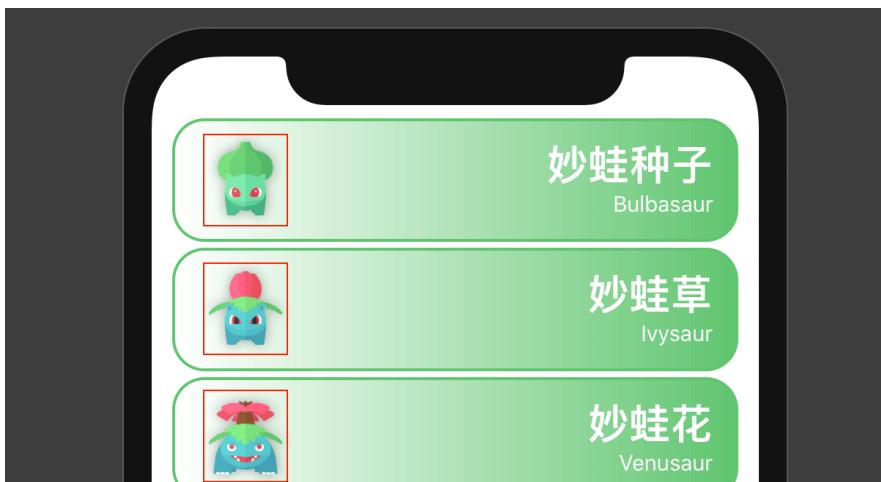
在上一章里，我们已经引入了 `@FileStorage` 来处理纯副作用，它会将所修饰的属性存放在本地文件，并从本地文件读取。我们已经看到过如何将它用在 `loginUser` 上，来存放用户登录数据的案例。这里，用同样的方法就能非常简单地完成宝可梦列表数据的缓存。只需要为 `AppState.PokemonList` 上的 `pokemons` 添加对应标注即可：

```
extension AppState {  
    struct PokemonList {  
        @FileStorage(  
            directory: .cachesDirectory,  
            fileName: "pokemons.json"  
    }  
}
```

```
)  
var pokemons: [Int: PokemonViewModel]?  
  
// ...  
}  
}
```

异步图片和图片缓存

现在，宝可梦列表的数据部分已经是通过网络进行请求了。接下来我们会看看如何从网络获取对应的图片，这些图片会被显示在 cell 的最左侧 (红框部分)：



到目前为止，我们都使用了 `Image` 从本地加载图片：

```
struct PokemonInfoRow: View {  
    var body: some View {  
        // ...  
        Image("Pokemon-\(model.id)")
```

```
.resizable()
.frame(width: 50, height: 50)
.aspectRatio(contentMode: .fit)
.shadow(radius: 4)

}

}
```

实际上，在 model 的 iconImageURL 属性中，已经包含了网络上对应图片的 URL 地址，为了能对应任意的宝可梦，我们会希望从网络去加载这张图片。另外，为了避免每次都重复下载，我们也会希望实现一些缓存的功能。

使用我们本章所介绍的技术，是可以完成发送 Action，触发副作用，下载数据，保存数据等一系列操作的。不过，这些任务几乎是每个 app 都会面临的，每次做类似操作也太繁琐。在本节里我们会使用一个叫做 Kingfisher 的框架来解决这个问题。

Kingfisher 是一个图片下载和缓存的框架，而且最近它也支持了 SwiftUI。我们只需要把 URL 传递进去，它就能帮助我们处理一系列异步图片加载的任务：包括尝试从缓存中寻找图片，如果没有找到则从 URL 下载图片，并将它缓存起来供日后使用。这可以让我们的代码变得非常简单。

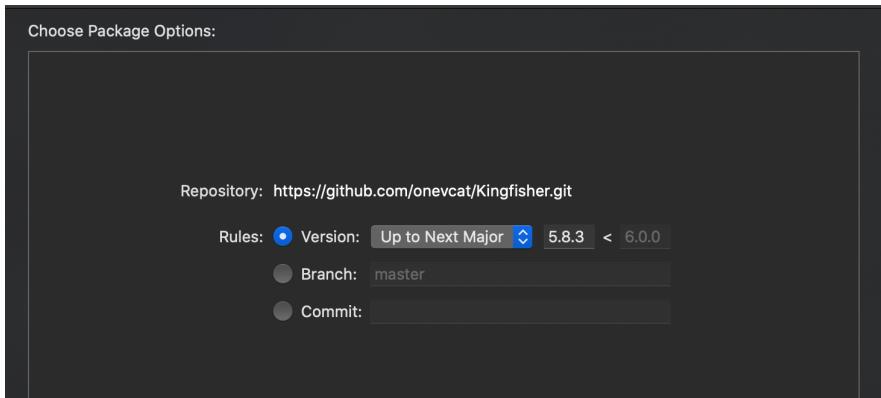
利益相关：笔者就是 Kingfisher 框架的创建者和主要维护者。

使用 Swift Package Manager 添加 Kingfisher

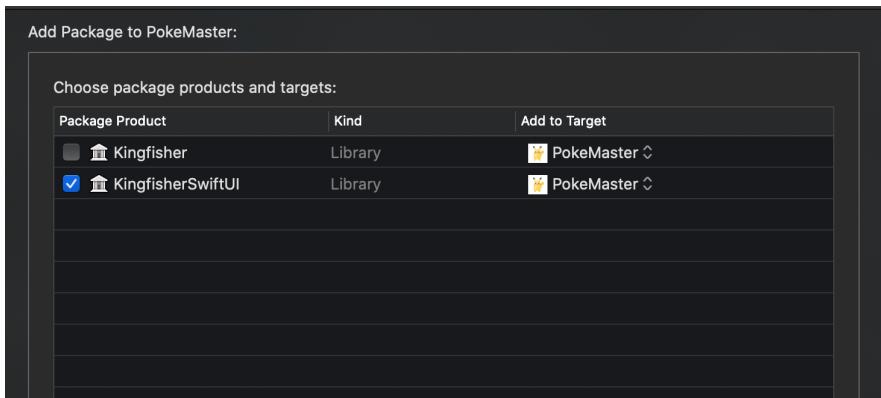
CocoaPods 和 Carthage 都是 iOS 开发中很常用的包管理工具。不过从 Xcode 11 开始，Apple 官方的 Swift Package Manager 被正式添加到了 Xcode 中。本节里，我们使用这种方式来添加 Kingfisher。

打开 Xcode 的 File - Swift Packages - Add Package Dependency 菜单，将 Kingfisher 的 git 地址 “<https://github.com/onevcat/Kingfisher.git>” 填入。Xcode 将拉取仓库并解析可用版本。在版本页面，选择 Up to Next Major 并保持最低版本

为当前最新版本(本书写作时, 5.8.3 是最新版本, 读者可能会看到一个更新的起始版本):



最后, 我们需要把 KingfisherSwiftUI 添加到 PokeMaster app 中, 就可以使用这个框架了:



使用 KFImage 加载图片

在“PokemonInfoRow.swift”里, 导入新加入的框架:

```
import KingfisherSwiftUI

struct PokemonInfoRow: View {
    // ...
}
```

然后，把 body 中的 Image 部分替换为 KFImage，并传入 PokemonViewModel 里定义好的 iconImageURL 就行了：

```
struct PokemonInfoRow: View {
    var body: some View {
        // ...
        // Image("Pokemon-\(model.id)")
        KFImage(model.iconImageURL)
            .resizable()
            .frame(width: 50, height: 50)
            .aspectRatio(contentMode: .fit)
            .shadow(radius: 4)
    }
}
```

KFImage 提供了和 SwiftUI.Image 同样的接口，所以原来作用在 Image 上的 modifier 一般来说可以不加修改地继续作用在 KFImage 上。就这么简单，我们完成了从 URL 获取图片的功能，而且顺带实现了图片缓存和快速读取。关闭 app 并再次打开，你可以看到整个 app 从数据到图片都不再有加载时间。

总结

在上一章提出的架构基础上，本章中我们看到了如何将 Combine 框架进一步结合到项目中。我们完成了两个任务：对用户邮箱进行验证，以及进行实际的网络请求。这

两个任务非常相似，它们的核心都是开发出能直接驱动 UI 的 Publisher，并用它来发送的数据来驱动 UI。

Combine 为我们处理复杂的异步逻辑提供了一种实际可用的简化手段。通过将不同的 Operator 进行组合，我们可以灵活地完成多种不同任务。这是一种自下而上开发方式，通过最基本的元素，甚至是自定义的 Operator，描述和组合出完全契合当前需求的行为，往往有助于让程序更容易理解。而 SwiftUI 中自动完成的状态订阅和 View 刷新，恰好与这种开发方式相得益彰。

最后，我们稍微涉及了使用 Swift Package Manager 来集成第三方框架，并使用 Kingfisher 来在 SwiftUI 中下载和显示图片的例子。随着 SwiftUI 的发展和成熟，相信这类第三方的快速解决方案也会越来越多，帮助我们快速完成某些特定任务。

阅读到这里，相信你已经具备使用 SwiftUI 创建界面，架构 app 以及实现大部分基本功能的能力了。在下一章里，我们将进一步深入，去看一些 SwiftUI 特有的界面和交互处理技巧，这些技巧将提升用户体验，帮助我们创建出更符合习惯的 app。

在此之前，我强烈建议你完成本章的练习，它们会为你实际理解和使用 Redux for SwiftUI 架构和 Combine 框架带来很大帮助。这些练习几乎都是本章所介绍的内容的延伸，如果一开始你觉得有些困难的话，请回头再阅读相关部分的内容，也许问题就能迎刃而解。

练习

1. 密码检查

在 AccountChecker 中，除了 email 和 accountBehavior，我们还加入了 password 和 verifyPassword，不过我们还完全没有使用过它们。现在请你添加一些逻辑，实现以下两点。

1-1

实现一个新的 Publisher，检查 password 和 verifyPassword 不为空，而且两者的值相等。

1-2

结合已有的 isEmailValid 和 1-1 中新建的 Publisher 来驱动“注册”按钮的显示状态：当邮件地址无效或已被占用，或者密码不符合要求的时候，将注册按钮设置为不可点按的无效状态。

提示：可以使用 combineLatest 来把两个简单的 Publisher 关联起来。类似 isEmailValid 的订阅和使用方式，我们也可以在创建 Store 时就订阅和持续按照新值来更新 UI 状态。

2. 完成注册功能

SettingView 中，虽然我们可以在注册和登录之间切换，但是实际上在按下按钮时我们只发出了 AppAction.login。请参照登录的部分，实现整套的注册功能，包括：

- 创建新的代表注册的 AppAction.register。
- 创建 RegisterAppCommand，并在注册请求完成时，发送合适的 Action 来改变 UI (可以重复利用已有的 AppAction.accountBehaviorDone 表示注册完成)。
- 用一定延时来模拟注册请求的 RegisterRequest。
- 在 reducer 中处理注册请求 Action 的逻辑。
- 更新“注册”按钮的逻辑，让它在被按下时发送相应的 Action 开始整个流程。

3. 清除缓存

现在，由 `@FileStorage` 修饰的 `PokemonList.pokemons` 会在被设置时自动写入到磁盘。如果想要重新触发下载，我们需要删除 app 重新安装，这有些不太合理。在 `SettingView` 中，我们准备了一个“清空缓存”按钮，请实现清空已加载数据和重置 app 状态的特性。

4. 宝可梦列表加载的错误处理

网络请求必然可能伴随错误，reducer 里对 `.loadPokemonsDone` 的处理中，遇到错误时，我们只是简单地将错误打印出来：

```
case .loadPokemonsDone(let result):
    switch result {
        case .success(let models):
            // ...
        case .failure(let error):
            print(error)
    }
```

这相当于没有进行任何处理：当错误发生时，界面将直接卡住。在这个练习里，你的任务是遇到错误时，显示一个如图这样的重试按钮。当点击重试时，重新去请求宝可梦列表数据。



提示：你可能需要在 AppState.PokemonList 里添加一个状态来持有错误，然后在 reducer 中维护该状态。重试按钮可以通过组合 Image 和 Text 得到，图中的圆圈箭头是名为“arrow.clockwise”的 SF Symbol。

5. 重构 @State，获取 Pokemon Ability

5-1

现在我们的项目中还剩两个 @State，它们是“PokemonList.swift”中的 expandingIndex 和 searchText。前者控制哪个 cell 应该处于展开状态，后者控制搜索框中的字符。我们反复强调过，示例 app 中所采用的架构规定了，和界面相关得状态都应该存放在 AppState 里。请尝试将 expandingIndex 和 searchText 都放到 AppState.PokemonList 中，并保持所有相关功能不变。

5-2

在显示 cell 时，对应的数据类型为 PokemonViewModel。而对于每个宝可梦详细面板 PokemonInfoPanel，除了 PokemonViewModel 中的信息外，我们还需要 [AbilityViewModel]，它包含了对应的 Pokemon 所持有的技能。

请创建一组合适的 Action，Command 和 State，在 PokemonList 中点击列表中某一行时，开始加载对应宝可梦的技能。宝可梦的技能信息被存储在 Pokemon Model

类型的 abilities 属性中。例如，你可以通过 `pokemon.abilities.first?.ability.url` 来获取第一个技能对应的 URL。

作为参考，你可以使用下面这些命名和定义。我们在下一章中也会涉及到部分内容，使用同样的名字有助于在本书的参考实现和你自己的实现之间进行互换和同步。

“AppState.swift”：

```
extension AppState {
    struct PokemonList {
        // 按 ID 缓存所有 AbilityViewModel
        var abilities: [Int: AbilityViewModel]?

        // 返回某个 `Pokemon` 的所有技能的 AbilityViewModel
        func abilityViewModels(
            for pokemon: Pokemon
        ) → [AbilityViewModel]?
    }
}
```

“AppAction.swift”：

```
enum AppAction {
    // ...
    // 切换 cell 展开状态
    case toggleListSelection(index: Int?)

    // 技能开始加载
    case loadAbilities(pokemon: Pokemon)
```

```
// 技能加载结束
case loadAbilitiesDone(
    result: Result<[AbilityViewModel], AppError>
)
```

“AppCommand.swift”：

```
struct LoadAbilitiesCommand: AppCommand {
    // ...
}
```

我们需要达到的效果是，在“PokemonList.swift”中，ForEach 里代表 cell 的 PokemonInfoRow 具有如下形式：

```
PokemonInfoRow(
    model: pokemon,
    expanded: // ... 从 State 中获取
)
.onTapGesture {
    withAnimation(
        .spring(
            response: 0.55,
            dampingFraction: 0.425,
            blendDuration: 0
        )
    )
{
    self.store.dispatch(
        .toggleListSelection(index: pokemon.id)
    )
}
```

```
)  
    self.store.dispatch(  
        .loadAbilities(pokemon: pokemon.pokemon)  
    )  
}  
}
```

并且，在 PokemonInfoPanel 中加载技能时，不再使用 AbilityViewModel.sample 方法来获取固定值，而是使用新添加的 abilityViewModels(for:) 来读取真正的技能数据。

最后，作为选做，如果你还有余力，可以尝试添加新的状态，让点击展开状态下的 cell 里中间的“显示面板”按钮时，显示 PokemonInfoPanel。我们在下一章会直接用一个预先设计好的包括这些内容的项目作为起始。因此你不用担心，即使无法完成这些内容，也不会对继续阅读本书造成影响。但如果你预先自行实现过这部分内容，将会对深入理解有很大帮助。

手势处理和导航

10

我们已经实现了 PokeMaster 这个示例 app 中的大部分内容了。一开始，通过利用本地的数据，我们构建了基本的 UI 元素，包括展示宝可梦的列表界面和一个用户注册/登录及设定选项的设置界面。然后，通过对 Combine 基本的学习，我们提出了一种适合处理 SwiftUI 数据流动的架构，并在此基础上为 PokeMaster 的 UI 框架注入了基本的行为：包括如何同步用户设置的数据和 UI 状态，如何检查用户输入的有效性，以及如何通过网络请求获取数据并加以展示。

本章中，我们会对 PokeMaster 示例中一些 UI 方面的细节进行讨论，比如如何处理用户手势，确定导航层级的方式等等。这些话题虽然相对零散琐碎，但是对提升 app 的用户体验会很有帮助。针对每个话题，我们会展示一到两个例子和对应的实现，它们作为入口和起始，能够帮助你大致了解相关部分的处理方式。

你可以在随书附带的“10.SwiftUI-UX/PokeMaster-Starter”中找到本章内容的初始代码。它在上一章结束的基础上，添加了显示详细信息面板和加载技能列表部分的实现。在本章中，我们的第一个任务是，让详细信息面板的出现和消失更加自然。

手势处理

面板显示

初始项目里，现在点击宝可梦列表 cell 展开后的“详细面板”按钮，会直接在屏幕下方显示出面板。点击这个按钮所发出的 Action 是 .togglePanelPresenting，你可以在“PokemonInfoRow.swift”里找到这部分代码：

```
Button(action: {  
    let target =  
        !self.store.appState.pokemonList  
            .selectionState.panelPresented  
    self.store.dispatch(  
        .togglePanelPresenting(presenting: target)  
    )  
})
```

```
)  
}) {  
    Image(systemName: "chart.bar")  
        .modifier(ToolButtonModifier())  
}  
}
```

在 `.togglePanelPresenting` 被发出后，`SelectionState` 里的 `panelPresented` 将被设置，最终触发 `MainTab` 里的 `body` 更新：

```
struct MainTab: View {  
    // ...  
  
    var body: some View {  
        TabView {  
            // ...  
        }  
        .edgesIgnoringSafeArea(.top)  
        // 1  
        .overlay(panel)  
    }  
}  
  
var panel: some View {  
    // 2  
    Group {  
        if pokemonList.selectionState.panelPresented {  
            if selectedPanelIndex != nil &&  
                pokemonList.pokemons != nil  
            {  
                PokemonInfoPanelOverlay(  
            }  
        }  
    }  
}
```

```
        model: pokemonList.pokemons![selectedPanelIndex!]!)
    } else {
        EmptyView()
    }
}
} else {
    EmptyView()
}
}
}
```

其中 PokemonInfoPanelOverlay 是一个 VStack，它用 Spacer 来“填充”上半部分：

```
struct PokemonInfoPanelOverlay: View {
    let model: PokemonViewModel
    var body: some View {
        VStack {
            Spacer()
            PokemonInfoPanel(model: model)
        }
        .edgesIgnoringSafeArea(.bottom)
    }
}
```

1. 以 overlay 的方式，在最外层的 TabView 上方显示面板。如果你还记得的话，在之前的章节，我们进行了一些界面上的调整，在那边，由于面板被 TabView 挡住了，所以我们暂时把原本添加在 PokemonList 上的 overlay 注释掉了。如果没有 @EnvironmentObject 的 store 来存储状态，我们将很难把 List 内部导致的面板显示状态变化应用到外层 TabView 来，这可能需要额外的好几层 Binding。但是在我们现有架构中，直接在最外层使用状态是一个自然而然且毫无痛苦的选择。

2. panel 返回一个 some View，虽然没有指定具体的 View 类型，但是编译器要求各个返回路径的类型一致。但我们的例子中，PokemonInfoPanelOverlay 和 EmptyView 的类型显然不一致。使用 Group，在内层利用 @ViewBuilder 支持 if...else 语句的特性，可以把不同类型的 View 包装到 Group View 里。另一种方式是使用 AnyView 把它们的具体类型抹消掉。

现在面板的显示是直接出现，想要让它消失，也只能再次点击按钮来切换。这在界面操作来说非常生硬。在本节中，我们专注于两件事：为面板出现添加动画，让它从屏幕下方弹出（类似于 Action Sheet 的样式）；以及使用手势来取消面板显示。

自定义通用容器

虽然我们这次只需要处理 PokemonInfoPanel，但是考虑到通用性，弹出动画和弹出面板的容器不应该局限于某个类型，也不应该和具体的某个状态耦合。为此，我们可以考虑创建一个通用的容器。

在“OverlaySheet.swift”中新建一个满足 View 的 OverlaySheet 类型，它具有一个用来表示内容 View 的泛型类型参数 Content，和一个表示是否正在被展示的 Binding<Bool>：

```
struct OverlaySheet<Content: View>: View {  
  
    private let isPresented: Binding<Bool>  
    private let makeContent: () → Content  
  
    init(  
        isPresented: Binding<Bool>,  
        @ViewBuilder content: @escaping () → Content  
    )  
}
```

```
    self.isPresented = isPresented
    self.makeContent = content
}

var body: some View {
    VStack {
        Spacer()
        makeContent()
    }
    .offset(y: isPresented.wrappedValue ?
        0 : UIScreen.main.bounds.height)
    .edgesIgnoringSafeArea(.bottom)
}
}
```

当 `isPresented` 为 `false` 时，我们为 `VStack` 设置了 `offset`，让它处于屏幕外部。以达到隐藏的目的。为了让 `OverlaySheet` 使用起来更方便，我们可以针对它为 `View` 添加一个 `extension`:

```
extension View {
    func overlaySheet<Content: View>(
        isPresented: Binding<Bool>,
        @ViewBuilder content: @escaping () -> Content
    ) -> some View
{
    overlay(
        OverlaySheet(isPresented: isPresented, content: content)
    )
}
```

```
}
```

这样一来，我们就可以把 TabView 中和 .overlay(panel) 相关的部分替换为：

```
struct MainTab: View {
    // ...
    var body: some View {
        TabView {
            // ...
        }
        .edgesIgnoringSafeArea(.top)
        // .overlay(panel)
        .overlaySheet(
            isPresented: pokemonListBinding.selectionState.panelPresented)
    }
    if self.selectedPanelIndex != nil &&
        self.pokemonList.pokemons != nil
    {
        PokemonInfoPanel(
            model: self.pokemonList.pokemons![self.selectedPanelIndex!]!
        )
    }
}
}
```

现在，overlaySheet 可以接受任意的 View，并通过 isPresented 来控制是否显示它。从行为上看起来和原来没有区别，但是我们在 overlaySheet 中摆脱了对 Store 里的状态的依赖，这让我们今后可以用同样的行为方式显示和处理任意 View。

动画显示

接下来的工作就可以都在 OverlaySheet 结构体中完成了。首先是显示时的弹出动画，这十分简单。对于不同的 isPresented，我们已经设置了不同的 offset。这里就只需要为 OverlaySheet 的 body 加上 .animation 就行了：

```
var body: some View {
    VStack {
        Spacer()
        makeContent()
    }
    .offset(y: isPresented.wrappedValue ?
        0 : UIScreen.main.bounds.height)
    // 添加动画
    .animation(.interpolatingSpring(stiffness: 70, damping: 12))
    .edgesIgnoringSafeArea(.bottom)
}
```

现在，整个 VStack 中可以动画的部分的值，都将以这个插值弹簧的动画方式进行。运行 app 的话，应该可以发现点击详情按钮时，面板可以从下向上弹出了。

不过如果你继续尝试，可能会发现一个问题。当详情面板在已经弹出的状态时，我们点击宝可梦列表里别的宝可梦，详情面板也将切换为显示为新的宝可梦。这个行为本身符合我们的状态逻辑。是有意为之：因为点击 cell 是发送的 `.toggleListSelection` 并不去改变 `panelPresented`，所以面板会一直存在，只是 `selectedPanelIndex` 的更新将同时更新面板的内容。但是，你会注意到，在切换时也会发生动画，这导致 UI 上的文本短暂出现布局问题，比如：



究其原因，这是由于我们添加的动画不仅作用于 `VStack`，也会作用于它的全部子 `View`。对于图示的这部分文本，我们所希望的是，即使起始状态和终结状态间的文本宽度和高度（行数）不匹配，也直接显示最终状态，而不是逐渐改变宽度和高度。为了达到这一点，在 `PokemonInfoPanel` 里，我们可以显式地指明不需要动画：

```
struct PokemonInfoPanel: View {  
    var body: some View {  
        VStack(spacing: 20) {  
            topIndicator  
  
            // Header(model: model)  
            // pokemonDescription  
            Group {  
                Header(model: model)  
                pokemonDescription  
            }.animation(nil)  
    }  
}
```

```
        Divider()  
        // ...  
    }  
}  
}
```

使用 Group 把 Header 及 pokemonDescription 组合起来，然后用 .animation(nil) 去掉它们的动画。再次运行 app，现在应该看到，在通过点击 cell 来直接切换面板显示内容的时候，文本部分不再有布局问题了。

用手势关闭面板

因为详细信息面板是从下方弹出的，所以下划手势将它关闭会更符合用户直觉。在这一节里，我们会为 OverlaySheet 添加手势识别和下划关闭的功能。

SwiftUI 中已经有一系列预先定义好的手势，比如处理点击的 TapGesture，处理长按的 LongPressGesture，或者拖拽的 DragGesture 等，它们都遵守 Gesture 协议。同时，作为 View modifier，SwiftUI 也预定义了像是 onTapGesture 这样把添加手势和处理手势合并在一起的简便方法，比如：

```
Text("点击按钮").onTapGesture {  
    print("按钮被点击")  
}
```

不过对于更一般的情况，我们需要明确地定义手势，并使用 gesture(_:including:) 方法来将它添加到 View 上。我们的计划是使用一个 DragGesture 来处理用户在面板上的拖拽行为。

首先，在 OverlaySheet 上添加一个状态，用它来记录手势的移动：

```
struct OverlaySheet<Content: View>: View {
```

```
// ...
@GestureState private var translation = CGPoint.zero

// ...
}

}
```

我们会在稍后解释 `@GestureState`，在此之前，你可以将它的行为简单地认为和普通的 `@State` 等效。

接下来，在 `OverlaySheet` 创建一个 `DragGesture` 的属性：

```
struct OverlaySheet<Content: View>: View {
    // ...

    var panelDraggingGesture: some Gesture {
        // 1
        DragGesture()
            // 2
            .updating($translation) { current, state, _ in
                state.y = current.translation.height
            }
            // 3
            .onEnded { state in
                if state.translation.height > 250 {
                    self.isPresented.wrappedValue = false
                }
            }
    }
}
```

1. 使用 `DragGesture` 来监听用户的拖拽手势。除了使用这个不带任何参数的初始化方法外，`DragGesture` 的 `init` 还支持设定最小侦测距离和要使用的座标系等参数。对于我们的需求，直接使用默认值的初始化方法就可以了。
2. 在手势触发后，每次状态变更时 `updating` 都会被调用。`updating` 的尾随闭包中第二个参数是一个标记为 `inout` 的待设定值。对这个值进行设置，`SwiftUI` 将可以通过 `$translation` 对 `@GestureState` 状态进行更新。这里我们在每次手势状态更新后都记录了手势当前所处位置相对于起始位置的位移高度。
3. 在手势结束时，判断相对于原始位置，手势是否下划超过了 250 point。如果下划距离足够，则将 `isPresented` 的包装值 (`wrappedValue`) 设为 `false`，这会关闭当前的弹出界面。

最后，我们把这个 `panelDraggingGesture` 添加到 `VStack` 上，并更新 `body` 部分，将 `offset` 的计算和 `translation` 值关联起来，让面板跟手滑动：

```
var body: some View {  
    VStack {  
        Spacer()  
        makeContent()  
    }  
    .offset(  
        y: (isPresented.wrappedValue ?  
            0 : UIScreen.main.bounds.height)  
            + max(0, translation.y)  
    )  
    .animation(.interpolatingSpring(stiffness: 70, damping: 12))  
    .edgesIgnoringSafeArea(.bottom)  
    .gesture(panelDraggingGesture)  
}
```

运行 app 试试看！现在弹出面板应该可以跟随手势向下移动（我们在计算 offset 时设定了 `max(0, translation.y)`，也就是忽略了手势向上的情况）。被标记为 `@GestureState` 的变量，除了具有和普通 `@State` 类似的行为外，还会在 `panelDraggingGesture` 手势结束后被自动置回初始值 0。所以下划距离不足以让面板关闭时，手势结束后面板将回到原地（你也许注意到了，我们设定的弹簧动画依然有效）。当下划距离足够，面板将被正常关闭，通过 `onEnded`, `isPresented` 这个 `Binding` 将追溯到表示面板显示状态的 `panelPresented` 变量，并将它设为 `false`。

小结

通过实现这个手势驱动的泛型 `OverlaySheet` 类型，我们看到了如何创建一个通用的，由 `Binding` 控制行为的 `View` 容器类型。`isPresented` 和 `makeContent` 都是由外部“注入”进来的行为，而 `OverlaySheet` 本身则只和 `View` 的自身行为有关，并不关心具体到底要显示什么内容。这种由外部注入控制的行为，在 SwiftUI 中颇为常见。我们会在下一节看到更多这方面的例子。

本节中使用了 `@GestureState` 持有手势导致的 `translation` 变化，关于这一点，进行三点补充说明。

- 首先，除了 `@GestureState` 外，你也可以使用普通的 `@State` 来暂存划动距离。除了 `updating(_:body:)` 以外，你也可以通过 `onChanged` 来设定新的手势状态。打个比方，假如 `translation` 被声明为 `@State` 的话，`onChanged` 里同步划动手势的部分会被写为：

```
@State private var translation = CGPoint.zero
DragGesture().onChanged { state in
    self.translation = CGPoint(x: 0, y: state.translation.height)
}
```

普通的 `@State` 和 `@GestureState` 最大的不同在于，当手势结束时，`@GestureState` 的值会被隐式地置为初始值。当这个特性正是你所需要的时

候，它可以简化你的代码，但是如果你的状态值需要在手势结束后依然保持不变，则应该使用 `onChanged` 的版本。

2. SwiftUI 提供了三种对手势进行组合的方式，分别是代表手势需要顺次发生的 `SequenceGesture`、需要同时发生的 `SimultaneousGesture` 和只能有一个发生的 `ExclusiveGesture`。我们不会详细展开讨论细节，但是如果你对手势有组合的要求的话（比如想要区分直接拖拽和长按以后的拖拽），可以详细对这几个类型进行研究。
3. 在本书前面涉及架构的部分，我们尽可能地避免了对 `@State` 的使用，而是通过访问 `@EnvironmentObject` 获取状态。但是在 `OverlaySheet` 里我们使用了本地的 `@GestureState` 保存状态，这是不是和介绍的架构有冲突？答案是否定的。本书中的架构并不是无脑鼓吹 `@EnvironmentObject`，对于只和 `View` 相关，且是用户操作造成的 UI 暂态，使用 `private` 的 `@State` 把状态限制在 `View` 的内部，对维护 app 状态（或者称为模型状态）的简洁，无疑是更明智的选择。

导航层级

在一个传统 iOS app 里，最常见的页面导航方式有三种：从右侧飞入的 `navigation` 方式，从下侧出现的 `modal` 方式，以及点击屏幕底部切换画面的 `tab` 方式。在 SwiftUI 中，这三种导航方式都是存在的，我们在计算器示例 app 中已经看到过 `modal` 方式的例子，本节里会对 `navigation` 和 `tab` 的切换做一些补充。

NavLink

我们来尝试实现点击 `cell` 中最右侧“更多信息”按钮时的功能吧。当点击这个按钮后，我们希望将一个展示宝可梦信息的网页推入到屏幕。SwiftUI 里没有直接让我们能显示网页的 `View`，所以我们需要求助于 iOS 开发中已有的技术，比如 `SFSafariViewController`。

UIViewControllerRepresentable

SFSafariViewController 是一个 UIViewController 的子类，我们已经看到过如何使用 UIViewRepresentable 来将一个 UIView 类型包装成 SwiftUI.View。对于 UIViewController 来说，SwiftUI 提供了一个相似的协议来帮助我们把任意的 UIViewController 桥接为 SwiftUI.View，那就是 UIViewControllerRepresentable。我们先来创建一个接受 url 并展示它的 SafariView：

```
import SwiftUI
import SafariServices

struct SafariView: UIViewControllerRepresentable {
    let url: URL

    func makeUIViewController(
        context: UIViewControllerRepresentableContext<SafariView>
    ) -> SFSafariViewController {
        let controller = SFSafariViewController(url: url)
        return controller
    }

    func updateUIViewController(
        _ uiViewController: SFSafariViewController,
        context: UIViewControllerRepresentableContext<SafariView>
    ) {
    }
}
```

和 UIViewRepresentable 的用法非常类似，我们至少需要实现两个方法：makeUIViewController(context:) 和 updateUIViewController(_:context:)。前者在 View 被创建时调用，后者在 body 刷新时调用。

有了 SafariView，我们就可以在 PokemonInfoRow 里添加展示逻辑了。想要以推入的方式显示新 View，需要使用 NavigationLink。在 “PokemonInfoRow.swift” 中，把展开状态下的最后一个按钮部分替换为：

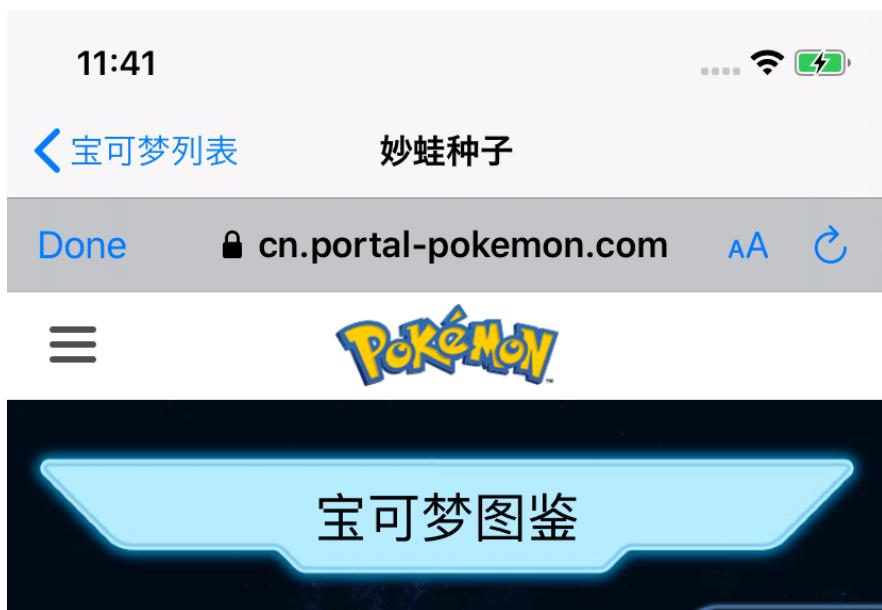
```
struct PokemonInfoRow: View {
    // ...
    var body: some View {
        // ...
        // Button(action: {}) {
        //     Image(systemName: "info.circle")
        //     .modifier(ToolButtonModifier())
        // }
        NavigationLink(
            destination:
                SafariView(url: model.detailPageURL)
            .navigationBarTitle(
                Text(model.name),
                displayMode: .inline
            ),
            label: {
                Image(systemName: "info.circle")
                .modifier(ToolButtonModifier())
            }
        )
    }
}
```

```
)  
// ...  
}  
}  
}
```

注意，`NavLink` 只在当前 View 处于 `NavigationView` 中才有效。在我们的例子中，`NavigationView` 被定义在 `PokemonRootView` 里。

用 Context 作为 Delegate

现在这个按钮应该已经可以将我们导航到新的网页页面了，而且我们也可以使用导航栏的返回按钮回到列表页面。



但是，SFSafariViewController 还提供了一个默认的“完成”按钮，现在点击 Done 时，并不会关闭这个页面。我们需要一种方法获取到该按钮的点击事件，并主动将网页弹出导航栈。在 SFSafariViewController 中，完成按钮的点击是通过 SFSafariViewControllerDelegate 中的 safariViewControllerDidFinish(_) 方法传递的。想要在 SafariView 获取它，需要我们为 SFSafariViewController 设置 delegate。这可以通过在 SafariView 里传递一个 context 并使用其中的 coordinator 对象来做到。这也是在 SwiftUI 中处理 UIKit 相关的 delegate 的一般性的方式。

更新 SafariView：

```
struct SafariView: UIViewControllerRepresentable {
    let url: URL
    // 1
    let onFinish: () -> Void

    // 2
    class Coordinator: NSObject, SFSafariViewControllerDelegate {
        let parent: SafariView

        init(_ parent: SafariView) {
            self.parent = parent
        }

        func safariViewControllerDidFinish(
            _ controller: SFSafariViewController)
        {
            parent.onFinish()
        }
    }
}
```

```
// 3

func makeCoordinator() → Coordinator {
    Coordinator(self)
}

func makeUIViewController(
    context: UIViewControllerRepresentableContext<SafariView>
) → SFSafariViewController
{
    let controller = SFSafariViewController(url: url)
    // 4
    controller.delegate = context.coordinator
    return controller
}

// ...
}
```

1. 为了尽量保持 SafariView 的独立，我们考虑把按钮点击事件传递出去，而不是在内部处理和 app 相关的逻辑。SafariView 的使用者在创建这个 View 时来指定点击完成时要进行的操作。
2. 内嵌的 Coordinator 类型满足 SFSafariViewControllerDelegate，它在按钮点击时调用 `onFinished` 方法。
3. `makeCoordinator` 是 `UIViewControllerRepresentable` 协议定义的一个方法。你可以在这里创建并返回一个任意类型的对象。这个对象在之后相关调用中都会被设置在 `context.coordinator` 属性里。使用这种方法，SwiftUI 让你可以

在 View 的不同时间段传递任意数据。本例中，我们所传递的是被作为 delegate 使用的 Coordinator。

4. 最后，将生成的 coordinator 设为 SFSafariViewController 的 delegate。

控制 Navigation 导航

最后，我们需要一种方法来把通过 NavigationLink 进入屏幕的页面推出 (pop)。除了像上面那样最简单的 init(destination:label:) 以外，NavigationLink 还有另外的接受一个 Binding<Bool> 的初始化方法。当 NavigationLink 按下时，被绑定的布尔值会被置为 true；相反地，当我们把它置为 false 时，就可以将正在显示的页面从导航栈中推出。

最简单的方式，就是在 PokemonInfoRow 里添加 @State 并设置它：

```
struct PokemonInfoRow: View {
    // ...
    @State var isSFViewActive = false
    // ...

    NavigationLink(
        destination:
            SafariView(url: model.detailPageURL) {
                // 1
                self.isSFViewActive = false
            }
        .navigationBarTitle(
            Text(model.name),
            displayMode: .inline
        ),
    )
}
```

```
// 2

    isActive: $isSFViewActive,
    label: {
        Image(systemName: "info.circle")
            .modifier(ToolButtonModifier())
    }
)

// ...
}
```

1. 在 SafariView 的 `onFinished` 调用后，把属于当前 `PokemonInfoRow` 的 `isSFViewActive` 设为 `false`。
2. 将 `$isSFViewActive Binding` 作为参数传入 `isActive`。点击 `NavigationLink` 时 `SwiftUI` 会将它置为 `true`。

不过，上面这样用 `@State` 的方式，不太符合示例 app 中的架构模式：

`isSFViewActive` 控制的不仅仅是自身 View 中临时状态，而是影响了整个 app UI 结构的状态。简单地使用这样的 `@State`，破坏了 `model` 和 `view` 的对应关系。我们会想办法将这个状态放到 `AppState` 里，用 `AppAction` 来修改状态并驱动 UI。

我们对此应该已经驾轻就熟了。首先在 `AppState.PokemonList` 里加上状态：

```
extension AppState {
    struct PokemonList {
        // ...
        var isSFViewActive = false
    }
}
```

在 AppAction 里添加 .closeSFView：

```
enum AppAction {  
    // ...  
    case closeSafariView  
}
```

同时，Store 中的 reduce 方法里也需要对应更新：

```
static func reduce(  
    state: AppState,  
    action: AppAction  
) -> (AppState, AppCommand?)  
{  
    // ...  
    case .closeSafariView:  
        appState.pokemonList.isSFViewActive = false  
  

```

最后，需要维护 “PokemonInfoRow.swift” 中相关部分：

```
struct PokemonInfoRow: View {  
    // ...  
    // 1  
    // @State var isSFViewActive = false  
  
    NavigationLink(  
        destination:
```

```
SafariView(url: model.detailPageURL) {
    // 2
    self.store.dispatch(.closeSafariView)
}

.navigationBarTitle(
    Text(model.name),
    displayMode: .inline
),
// 3
isActive: expanded ?
$store.appState.pokemonList.isSFViewActive :
.constant(false),
label: {
    Image(systemName: "info.circle")
    .modifier(ToolButtonModifier())
}
)
}
```

1. 我们不再需要 cell 中的这个状态，所以可以将它删掉。
2. 通过发送 .closeSafariView 来修改 isSFViewActive 状态。
3. 注意，这里我们先判断了 expanded，然后再选择是使用 isSFViewActive Binding 还是使用一个常数的 Binding.constant(false) 来把一个固定值传入 NavigationLink。

关于第 3 点我想需要一些额外说明。这不仅是使用 AppState 时需要注意的，在一般处理像是 cell 这样的会重复多次出现的 View 中的状态时，我们需要牢记，如果使用的状态不属于 cell 本身，那么它们是可能会在多个 cell 之间共享的。为了说清楚这个问题，我们来看一个最简单的例子。请考虑下面的一段代码，它显示了一个 List：

```
struct ContentView : View {
    @State var show: Bool = false
    var body: some View {
        NavigationView {
            List(0 ..< 10) { i in
                NavigationLink(
                    destination: Text("Detail \(i)"),
                    isActive: self.$show)
                {
                    Text("Cell \(i)")
                }
            }
        }
    }
}
```

它在 `NavigationView` 里展示了一个列表，其中 `cell` 显示了自己的序号 ("Cell \((i)")，并且希望导航到一个展示序号的详细页面 ("Detail \((i)"). 第一眼看上去可能察觉不到问题所在。不过，当你点击某个 `cell` 时，根据你使用的 Xcode 版本不同，可能会出现各种奇怪的现象。比如在本书写作时的 Xcode 中，它将会顺次把从 0 到 9 的 `Detail Text` 都推入一遍，最后再回到初始的列表状态。也许在你使用的版本中会有不同的情况，但是应该都无法像我们所想象的那样工作。

究其原因，是因为 `show` 是定义在 `ContentView` 中的变量，它被所有的 `cell` 共享。当你点击某个 `cell` 时，`show` 的值被设置，这导致 `body` 部分被重新求值。而此时，`show` 不仅会作用在你点击的 `cell` 上，也会作用在所有其他 `cell` 上，也就是同时有多个 `NavigationLink` 的 `isActive` 参数接收到 `true`，这会带来未定义的行为。

同样的情况也会发生在 List 的 cell 上使用 .sheet 以 modal 方式弹出的界面中。.sheet 最常见的用法是 sheet(isPresented:content:), 它需要 Binding 来决定展示与否，在实际 app 中更容易让人犯错。

因此，面对这类 cell 中需要某个状态的时候，我们需要引入这个状态和当前 cell 的某种关联。在示例 app 里，我们使用了 cell 的 expanded 来判定是不是需要真的把 Binding 传递进去。另一种常见的解决方式是不要在列表 cell 上定义导航行为（包括 NavigationLink 或 sheet），而是将它们放到列表外层，这样就不会出现多个 cell 共用一个状态的情况了。在本章的练习中，我们会看到一个具体的例子。

TabBar

整个 PokeMaster app 的顶层是 MainTab，现在它定义了一个包含两个页面的 Tab 导航：

```
TabView {  
    PokemonRootView().tabItem {  
        Image(systemName: "list.bullet.below.rectangle")  
        Text("列表")  
    }  
    SettingRootView().tabItem {  
        Image(systemName: "gear")  
        Text("设置")  
    }  
}
```

用户可以通过点击屏幕下方的 tab 按钮切换页面，但是我们现在还没有办法使用代码来进行切换。为了达到这个目的，我们需要对 TabView 和其中的 tabItem 做一些改造。

首先，在 AppState 里追加一组 State 定义，它包括 tab 的描述和一个持有当前选中 tab index 的变量：

```
extension AppState {  
    struct MainTab {  
        enum Index: Hashable {  
            case list, settings  
        }  
  
        var selection: Index = .list  
    }  
}
```

在 AppState 里，添加这个 MainTab 成员作为控制 Tab 导航的状态：

```
struct AppState {  
    // ...  
    var mainTab = MainTab()  
}
```

在 MainTab 中创建 TabView 的时候，我们将代表选中项目的 index 的 Binding 传入进去，用户点击 Tab 时将更新这个 index 值。同时，我们也可以通过代码更改它，来控制 UI 上显示的 tab。我们已经在不止一个地方看到过这种模式了，大部分涉及到导航和不同 View 之间关系的 SwiftUI 类型，都有提供了类似设置方式。具体来说，在 MainTab 里，进行以下修改：

```
struct MainTab: View {  
    // ...  
    var body: some View {  
        // 1
```

```
// TabView {
    TabView(selection: $store.appState.mainTab.selection) {
        PokemonRootView().tabItem {
            Image(systemName: "list.bullet.below.rectangle")
            Text("列表")
        }
        // 2
        .tag(AppState.MainTab.Index.list)
        SettingRootView().tabItem {
            Image(systemName: "gear")
            Text("设置")
        }
        .tag(AppState.MainTab.Index.settings)
    }
    // ...
}
```

1. 将刚才新加的 `selection` 作为 Binding 传入 `TabView`。
2. 给每个 `tabItem` 加上一个类型符合要求的 `tag`。SwiftUI 将使用这个 `tag` 值来辨别所选中的 Tab。

现在，设置 `AppState.MainTab` 里的 `selection` 变量就可以控制 tab 切换了。我们在这里不去实际使用它，不过我们会在练习中涉及到这部分内容。

支持 URL Scheme

本章已经有很多内容了，作为收尾，我们来看看如何让我们的 app 支持 URL Scheme。在 iOS app 中，通过支持 URL Scheme，我们可以定义让 app 接受某种形

式的 URL，当用户在其他 app 或者网页中打开这类 URL 时，iOS 将唤起 app，并让 app 有机会根据收到的 URL 导航到合适的页面。

这一节里，我们来实现让 app 支持一个形如 “pokemaster://showPanel?id={id}” 的 URL。其他 app 在打开这个 URL 时，PokeMaster 应该展示对应 id 的宝可梦的详细面板。在不同 app 中，这类特性实现起来的难度，和 app 所采用的架构息息相关。有时候这个特性的实现会非常困难，有时候它会随着 app 结构的变动变得难以维护，有时候甚至不大幅修改架构就难以实现。不过对于我们在示例 app 里所采用的完全由数据驱动的 UI 来说，支持 URL Scheme 特性可以说是小菜一碟：我们要做的仅仅是在 app 启动时将 AppState 设置为我们所需要的值，然后 SwiftUI 就将为我们“自动”把剩余的事情全部搞定。

为了让代码简单一些，我们首先对 SceneDelegate 进行一些重构。在 “SceneDelegate.swift” 中，创建一个新的 showMainTab(scene:with:) 方法，把原来在 scene(_:willConnectTo:options:) 中的 window 设置部分的代码提取出来：

```
class SceneDelegate: UIResponder, UIWindowSceneDelegate {
    private func showMainTab(
        scene: UIScene,
        with store: Store)
    {
        if let windowScene = scene as? UIWindowScene {
            let window = UIWindow(windowScene: windowScene)
            window.rootViewController =
                UIHostingController(
                    rootView:
                        MainTab().environmentObject(store))
        }
        self.window = window
        window.makeKeyAndVisible()
    }
}
```

```
    }

}

// ...

}
```

这样，我们只需要构建合适的 Store，将它传递给 showMainTab，就可以显示符合需要的界面了。在 SceneDelegate 里，添加 createStore 方法。它接受一组可能的 URL Scheme，并返回对应的所需要的 Store 示例：

```
private func createStore(
    _ URLContexts: Set<UIOpenURLContext>
) → Store {

    let store = Store()

    // 1
    guard let url = URLContexts.first?.url,
        let components =
            URLComponents(url: url, resolvingAgainstBaseURL: false)
    else {
        return store
    }

    // 2
    switch (components.scheme, components.host) {
    case ("pokemaster", "showPanel"):

        guard let idQuery = (components.queryItems?.first {
            $0.name = "id"
        })
```

```
        }),

let idString = idQuery.value,
let id = Int(idString),
id ≥ 1 && id ≤ 30

else {
    break
}

// 3

store.appState.pokemonList.selectionState =
.init(expandingIndex: id, panelIndex: id, panelPresented: true)

default:
    break
}

return store
}
```

1. 从 URLContexts 中提取打开 app 所使用的 URL。如果没有期望的 URL，则直接使用默认状态。
2. 对 scheme 和 host 进行匹配，我们的目标 URL 形如 “pokemaster://showPanel?id={id}”。相比于写一堆 if 语句，switch 会更清晰。在今后需要把匹配支持扩展到更多的 URL 时，这个写法会非常有优势。
3. 为 AppState 赋值，然后返回这个设置好状态的 Store。

为了让 app 在启动时和收到 URL Scheme 调用被打开时，能按照 URL Scheme 的方式初始化，我们还需要在 SceneDelegate 中更新原有的 scene(_:willConnectTo:options:) 方法，并添加一个 openURLContexts 方法。它们的内容非常相似：

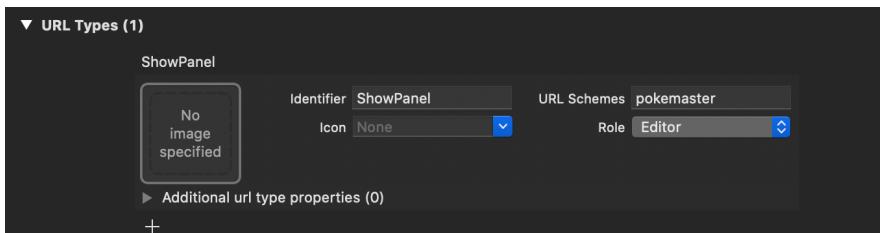
```

func scene(
    _ scene: UIScene,
    willConnectTo session: UISceneSession,
    options connectionOptions: UIScene.ConnectionOptions
) {
    let store = createStore(connectionOptions.urlContexts)
    showMainTab(scene: scene, with: store)
}

func scene(
    _ scene: UIScene,
    openURLContexts URLContexts: Set<UIOpenURLContext>
) {
    let store = createStore(URLContexts)
    showMainTab(scene: scene, with: store)
}

```

最后，我们需要在 Info.plist 中添加需要监听的 URL Scheme。在 Xcode 项目面板中，选中 PokeMaster target，在 Info 栏的 URL Types 里添加设置一个条目，把“URL Schemes”设为“pokemaster”。



编译并运行 app，然后切换到 Mobile Safari，并在地址栏输入“pokemaster://showPanel?id=25”进行访问。PokeMaster app 应该会从后台被唤起，并且显示对应宝可梦的详情面板。

相对于指令式编程中一步步指导程序如何达到需要的界面的方式，声明式 UI 只需要对 app 处在的状态进行描述即可。而在我们的架构中，在完全不涉及 app 界面的情况下，构建一个完整的 Store 是完全可行，并且非常简单的。这两点的组合，让我们能很方便地支持导航至任意界面和任意状态的 URL Scheme，这种便利性是传统 iOS app 所难以企及的。

总结

本章中我们以拖拽手势为例，研究了在 SwiftUI 中处理手势的一般方式。同时，我们构建了一个和 app 状态脱钩的，接受外部输入来做具体构建的 OverlaySheet 类型，并把手势处理的代码封装到了这个自定义类型中。OverlaySheet 通过一个 Binding 来接受显示控制，同时反过来，也将手势导致的状态变化结果同步回去。这在 SwiftUI 的导航结构中是非常常见的做法，我们在 NavigationLink，sheet 和 TabView 中，都看到了类似的模式。

在示例 app 架构下，对于状态的配置，我们大体上有两种选择：要么将状态放到 AppState 中并通过 Store 注入到 View，要么使用 @State 或者 @ObservedObject 把状态和 View 放在一起。选择哪个取决于你是否需要从外界对状态进行设置，或者说取决于你在多大程度上依赖于通过状态驱动 UI。对于多个 View 可能会共用或者交互的状态，显而易见，选择 AppState 会更好。对于单个 View 中，由用户行为产生的暂态状态，比如示例 app 中手势拖动的距离，则选择直接放在 View 自身中。如果你无法在一开始就能确定状态的位置，建议可以直接先放到 AppState 中，它会带来更多的灵活性和泛用性。

练习

1. 更改 SafariView 的展示方式

在示例 app 中，我们实现了使用 `NavigationLink` 来把 `SafariView` 推入到界面中。请你尝试修改为使用 sheet 方法来把这个网页以 modal 的方式打开。

在实现时，请注意以正确的方式放置 `.sheet` 调用的方法。比如使用下面这样的 View 组织层级时，应当慎重考虑：

```
List {  
    ForEach {  
        cell.sheet  
    }  
}
```

我们需要特别注意，控制 `sheet` 是否展示的 `isPresented Binding` 必须只和当前的 `cell` 相关。这一点在实践中很容易被忘记，或者可能难以实现。在一般的情况下，可以考虑将 `sheet` 提取到外面，让整个 `List` 使用同一个状态，而非多个 `cell` 共享的状态，来控制是否展示：

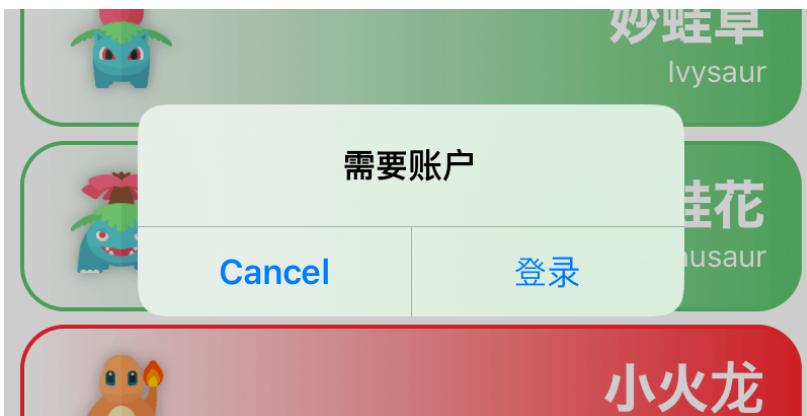
```
List {  
    ForEach {  
        cell  
    }  
}.sheet
```

当然，你也可能会有其他实现方法，但请确保你充分你能够理解和解释这些实现背后的运作方式。

2. 切换 TabView 展示的页面

在 cell 展开时，从左到右依次是“收藏”、“面板”和“网页”按钮。对于“收藏”按钮，我们现在还没有提供任何实现。请遵照下面的计划实现这个按钮的功能：

- 收藏功能是提供给登录用户的：如果用户在点击收藏按钮时还没有登录的话，使用 alert 弹出对话框提示用户需要登录，对话框应该有两个按钮，分别对应取消和登录。



- 当点击对话框的“登录”按钮时，切换到设置页面。

3. 更复杂 URL Scheme 支持

请实现一个新的 URL 支持：形如

“pokemaster://userRegister?email=admin@example.com”的 URL 应该可以打开 PokeMaster app，并在还没有用户登入的情况下，导航到设置页面的注册栏。为了方便用户快速注册，将 URL 里的 email 参数自动填入到注册表单的邮箱输入框中，这样用户只需要输入密码即可迅速完成登入。

用户体验和布局 进阶

11

PokeMaster app 现在已经是一个具有完整功能的 SwiftUI app 了。麻雀虽小，五脏俱全，在这个示例 app 里，我们涉及到了一个一般性的 iOS app 所需要的大部分内容：

- 如何构建内容展示的列表
- 如何构建用户交互的表单
- 如何进行网络请求并把内容展示出来
- 如何响应用户的手势
- 如何在不同页面之间进行导航
- 以及，如何通过一定的架构将所有上面的内容整合起来

这一章里，我们会涉及到一些稍微进阶的内容，包括自定义绘制的方法，和一些 SwiftUI 布局方面的话题。在学习和实际练习里，可能你已经遇到过这样的情况：有时候觉得某个布局难以实现，或者需要不断通过试错，来确定 View 的某种布局写法是否确实有效，甚至或者觉得某些“奇怪”的行为只是由于 SwiftUI 还在初期，所以是一个暂时性的 bug。根据笔者自身的经验，有时候确实是因为 SwiftUI 还不完善，但在更多情况下，这意味着你还没有真正理解 SwiftUI 的布局和工作方式。

本章将会试图带领你去触及 SwiftUI 一些表层之下的更深入的话题，包括自定义的 Path 绘制和动画，SwiftUI 布局的原理，View 对齐方式和基础等。不过由于 SwiftUI 的文档并不丰富，本书写作时，很多内容的注解在 Apple 开发者网站上也还是一片空白。所以这些内容更多的是基于尝试后的猜测和经验总结。如果你想获取更精确和深入的理解，还是需要自己动手加以实践。

自定义绘制和动画

SwiftUI 提供了很多常见的标准 UI 控件，比如 Button、Text、Image、Toggle 等等。如果我们想要更复杂和更自由的 UI，可能就需要进行一些自定义绘制。

Path 和 Shape

Shape 协议是自定义绘制中最基本的部分，它只要求一个方法，即给定一个 CGRect 的绘制范围，返回某个 Path。

```
public protocol Shape : Animatable, View {  
    func path(in rect: CGRect) -> Path  
}
```

SwiftUI 提供的部分形状，比如 Circle 或者 Rectangle，都是 Shape 协议的具体实现。如果你对传统 iOS 开发中的 Core Graphics 有所了解的话，可能对“给定 CGRect，请开始你的绘制”这种模式感到熟悉。Shape 的 path(in:) 和 UIView 的 drawRect: 方法如出一辙。而具体的绘制也很类似，Core Graphics 为我们提供的最基本的在上下文中绘制线段和圆弧的 API，在 SwiftUI 的 Path 中也能找到等价的方法。比如下面的代码就定义绘制了一个底部为圆弧的三角形箭头：



```
struct TriangleArrow: Shape {
    func path(in rect: CGRect) -> Path {
        Path { path in
            // 1
            path.move(to: .zero)
            // 2
            path.addArc(
                center: CGPoint(x: -rect.width / 5, y: rect.height / 2),
                radius: rect.width / 2,
                startAngle: .degrees(-45),
                endAngle: .degrees(45),
                clockwise: false
            )
            // 3
            path.addLine(to: CGPoint(x: 0, y: rect.height))
            path.addLine(to: CGPoint(x: rect.width, y: rect.height / 2))
            // 4
            path.closeSubpath()
        }
    }
}

// 5
TriangleArrow()
    .fill(Color.green)
    .frame(width: 80, height: 80)
```

上面对于 TriangleArrow 的绘制只是为了说明基本的 Path API 中添加线段和圆弧的方式，请不要纠结于具体的数字或者圆弧角度。

1. 为了完成满足 Shape 所需的 path(in:) 方法，直接创建一个 Path 结构体是最灵活和普遍的做法。第一步我们将绘制起点设定在 rect 的零点(左上)。
2. 添加一段圆弧。
3. 为 path 添加线段。
4. 最后一段线段不需要手动添加，可以直接使用 closeSubpath 让绘制回到原点，从而得到闭合曲线。
5. 由于 Shape 是一个遵守 View 的协议，所以我们可以直接按照其他 View 同样的方式来使用它。使用 .fill 进行单色(比如例子中的 Color.green)或者渐变(比如在之前章节介绍过的 LinearGradient)填充。最后，.frame 会给 Shape 一个参考的尺寸，我们会在下一节中涉及到更多关于 frame 的话题。

Geometry Reader

有时候，我们会想要在 View 里进行一些更精细的尺寸及布局计算，这需要获取一些布局的数值信息：比如当前 View 可以使用的 height 或者 width 是多少，需不需要考虑 iPhone X 系列的安全区域(safe area)等。SwiftUI 中，我们可以通过 GeometryReader 来读取 parent View 提供的这些信息。和 SwiftUI 里大部分类型一样，GeometryReader 本身也是一个 View，它的初始化方法需要传入一个闭包，这个闭包也是一个 ViewBuilder，并被用来构建被包装的 View。和其他常见 ViewBuilder 不同，这个闭包将提供一个 GeometryProxy 结构体：

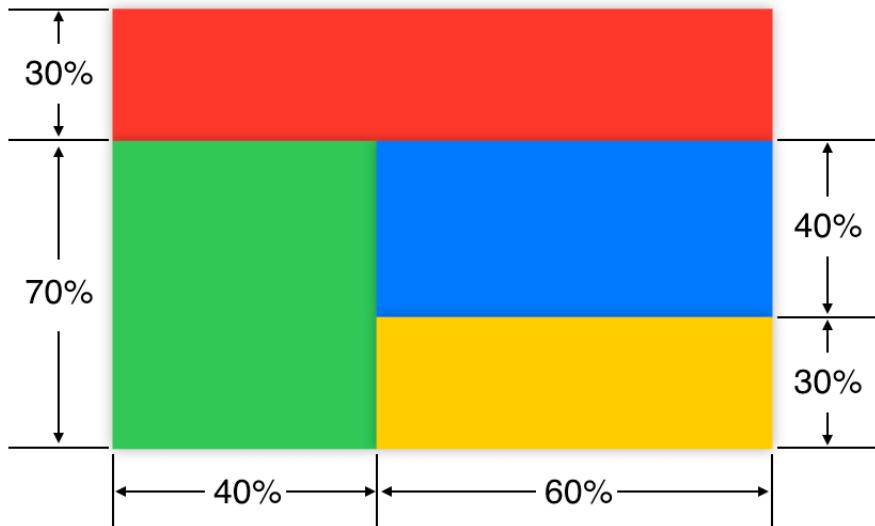
```
struct GeometryReader<Content>
: View where Content : View
{
    init(@ViewBuilder content:
```

```
@escaping (GeometryProxy) → Content)
// ...
}
```

GeometryProxy 中包括了 SwiftUI 中父 View 层级向当前 View 提议的布局信息，它会为 Content View 提供一个上下文(关于 SwiftUI 布局的过程和更多详细信息，我们会在下一小节进行更多介绍)。GeometryProxy 的定义如下：

```
public struct GeometryProxy {
    public var size: CGSize { get }
    public subscript<T>(anchor: Anchor<T>) → T { get }
    public var safeAreaInsets: EdgeInsets { get }
    public func frame(
        in coordinateSpace: CoordinateSpace
    ) → CGRect
}
```

在本章中，我们只会涉及 size，它表示 SwiftUI 布局系统所能提供的尺寸。这让我们可以按照尺寸自适应缩放构建 UI。比如我们想要按照下面的尺寸百分比进行布局：



可以使用下面的代码：

```
struct FlowRectangle: View {
    var body: some View {
        GeometryReader { proxy in
            VStack(spacing: 0) {
                Rectangle()
                    .fill(Color.red)
                    .frame(height: 0.3 * proxy.size.height)
                HStack(spacing: 0) {
                    Rectangle()
                        .fill(Color.green)
                        .frame(width: 0.4 * proxy.size.width)
                    }
```

```
VStack(spacing: 0) {
    Rectangle()
        .fill(Color.blue)
        .frame(height: 0.4 * proxy.size.height)
    Rectangle()
        .fill(Color.yellow)
        .frame(height: 0.3 * proxy.size.height)
}
.frame(width: 0.6 * proxy.size.width)
}
}
}
}
```

FlowRectangle 自身并不知道自己会被放置在多大的“画布”上，使用 GeometryReader 包装后，让内层 View (本例中为最外的 VStack 和它的所有子 View) 可以根据外层尺寸自适应调整大小。在使用 FlowRectangle 时，我们一般会将它限制在某个 frame 里，让内部 View 读取 GeometryProxy 的内容来确定最终尺寸。

六边形绘制实例

在了解了 Shape 和 GeometryReader 后，我们可以看一个实际的例子。在 PokeMaster app 的详细信息面板，其实在最初的设计稿中还有一个表示宝可梦能力的雷达图，即下图红框部分：



妙蛙草

Ivysaur

种子宝可梦

身高 1.0m

体重 13.0kg

草

毒

There is a bud on this Pokémon's back. To support its weight, Ivysaur's legs and trunk grow thick and strong. If it starts spending more time lying in the sunlight, it's a sign that the bud will bloom into a large flower soon.

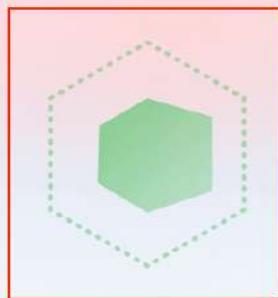
技能

叶绿素

晴朗天气时，速度会提高。

茂盛

HP减少的时候，草属性的招式威力会提高。



SwiftUI里并没有这种六边形的内建形状，所以我们需要自己对它进行绘制。

为了简明，下面的代码只展示了布局和绘制 API 的使用，不会具体解释绘制逻辑和线段位置的计算（它们涉及到一些基本的三角函数）。如果你对完整的代码感兴趣的话，可以查阅随书“source/PokeMaster-Finished”里“RadarView.swift”的源码。

```
struct RadarView: View {
    // ...
    var body: some View {
        // 1
        GeometryReader { proxy in
            ZStack {
                // 2
                Hexagon(
                    values: Array(repeating: self.max, count: 6),
                    max: self.max
                )
                .stroke(
                    style: StrokeStyle(
                        lineWidth: 2,
                        dash: [6, 3]))
                .foregroundColor(self.color.opacity(0.5))
            }
            // 3
            Hexagon(
                values: self.values,
                max: self.max,
                progress: self.progress
            )
            .fill(self.color)
        }
        // 4
        .frame(
            width: min(proxy.size.width, proxy.size.height),
            height: min(proxy.size.width, proxy.size.height)
        )
    }
}
```

```
    }
}

}
```

1. 当我们希望对 Content 进行限制时，使用 GeometryReader 来读取可用的尺寸。proxy 的内容会在 “// 4” 中使用。
2. 为了实现需要的效果，我们使用 ZStack 将两个六边形 (Hexagon) 堆叠起来，首先是外层固定的正六边形虚线，它的六个定点值和所接受的最大值相同。对于一个 Shape，我们使用 .stroke 来获取它的边缘路径。
3. 内层六边形通过 .fill 进行颜色填充，表示具体的数值。
4. 我们始终希望外层是一个正六边形，因此需要一个正方形的 .frame。通过比较 proxy.size 的 width 和 height，对 ZStack 的尺寸进行限制。

Hexagon 满足 Shape 协议，其中核心的 path(in:) 代码如下：

```
struct Hexagon: Shape {

    let values: [Int]
    let max: Int

    func path(in rect: CGRect) -> Path {
        Path { path in
            let points = self.points(in: rect)
            path.move(to: points.first!)
            for p in points.dropFirst() {
                path.addLine(to: p)
            }
            path.closePath()
        }
    }
}
```

```
    }
}

// 三角函数计算，将 values 转换为 rect 中的座标点
func points(in rect: CGRect) -> [CGPoint] {
    // ...
}
```

在完成这些绘制后，我们就可以在 Preview 或者是实际的 PokemonInfoPanel 中使用 RadarView 了。比如：

```
struct PokemonInfoPanel: View {
    // ...
    var body: some View {
        // ...

        HStack(spacing: 20) {
            AbilityList(
                model: model,
                abilityModels: abilities
            )
            RadarView(
                values: model.pokemon.stats.map { $0.baseStat },
                color: model.color,
                max: 120
            )
            .frame(width: 100, height: 100)
        }
    }
}
```

```
    }
}
```

注意，在使用时，我们可以通过 frame 为 RadarView 指定尺寸。否则，AbilityList 和 RadarView 将会等分屏幕宽度。关于这种行为的原因，我们会在下一节里详细谈及。

Animatable Data

SwiftUI 中 Shape 非常强大，对 Shape 按照路径制作动画也很简单。如果你有注意，会发现在 Shape 协议定义已经规定了 Shape 是满足 Animatable 的：

```
protocol Shape : Animatable, View {
    // ...
}
```

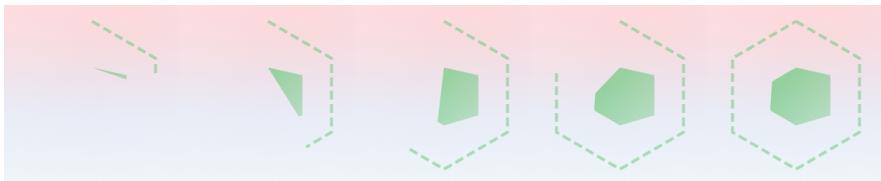
Animatable 本身的定义很简单：

```
protocol Animatable {
    associatedtype AnimatableData : VectorArithmetic
    var animatableData: Self.AnimatableData { get set }
}
```

基本在事实上，它只要求你定义一个可以读取和设置的 animatableData，而这个值需要是一个可以支持加减的矢量值 VectorArithmetic。在 SwiftUI 中，像是 CGFloat, Double 等都是满足 VectorArithmetic 要求的。除此之外，如果提供一对满足 VectorArithmetic 的类型，将它们组合起来生成的 AnimatablePair 也满足 VectorArithmetic，这让我们可以同时为多个值定义动画。

SwiftUI 为包括 CGPoint、CGSize 和 Angle 在类的很多基本类型实现了 Animatable。这也是为什么我们能通过改变某个 View 上或者它的 modifier 上的对应值来进⾏动画的原因。不过对于一般的 Shape 来说，它所默认提供的

`animatableData` 没有做什么特殊操作。这其实很合理，像是 `CGPoint`、`CGSize` 和 `Angle` 这些类型，我们给定初始值和最终值后，总是可以通过插值的方式在这两个值之间进行过渡，也就是动画。但是为一般性的 `Shape` 定义这类过渡是不可能的。所以想要实现 `Shape` 的动画，我们需要自己定义合适的 `animatableData`。



作为示例，想要实现的是上图这样的动画：当雷达图出现时，我们希望它从顶端的原点开始，内外两个六边形都按照顺时针方向通过动画扇形展开并显示出来。

首先，在 `Hexagon`，我们需要一个变量来控制当前的绘制进度。一个 `CGFloat` 的 `progress` 值就能很好地完成任务：

```
struct Hexagon: Shape {  
    var progress: CGFloat  
  
    // ...  
}
```

`CGFloat` 已经满足 `VectorArithmetic` 了，在我们实现 `animatableData` 的 `getter` 和 `setter` 时，直接将它和 `progress` 关联起来就可以了：

```
struct Hexagon: Shape {  
    // ...  
    var animatableData: CGFloat {  
        set { progress = newValue }  
        get { progress }  
    }
```

```
    }
}
```

当我们通过 `.animate` 或者 `withAnimation` 操作 Hexagon (或者它的 Container View) 时，SwiftUI 将自动按照动画要求的曲线为 `animatableData` 从 0 到 1 进行插值。这会调用它的 `setter`，并通过 `setter` 更新 `progress` 的值，然后触发 Shape 的 `path(in:)` 进行绘制。所以，我们最后只需要把 `progress` 应用到 `path(in:)` 里，让每次插值重绘时的形状满足要求就可以了。Path 提供一个 `trimmedPath(from:to:)` 方法，它可以按照输入值截取一段路径：

```
struct Hexagon: Shape {
    // ...
    func path(in rect: CGRect) -> Path {
        Path { path in
            // ...
        }
        .trimmedPath(from: 0, to: progress)
    }
}
```

比如 `progress` 值为 0.4 时，将返回上面图中左起第二张所显示的路径图形。

最后，只需要我们设置合适的 `progress` 值，就可以让 Hexagon 以动画方式显示出来了。相关代码可能类似这样：

```
@State var progress: CGFloat = 0

VStack {
    Hexagon(
        values: [165, 129, 148, 176, 152, 140],
```

```
    max: 200,  
    progress: progress  
)  
.animation(.linear(duration: 3))  
.frame(width: 100, height: 100)  
  
Button(action: { self.progress = 1.0 })  
{  
    Text("动画")  
}  
}
```

布局和对齐

看到本节的标题，可能你会赶到有一些奇怪。毕竟我们在本书一开始就已经谈及过 SwiftUI 布局方面的事情，并且贯穿本书我们也实现了不少常见布局的 UI。不过，在练习的时候，你可能会发现有时候 SwiftUI 并没有按照你的想象放置 View。通过不断尝试和修改，有时候你能够“碰巧”获得一个可行的写法；也有时候不论如何努力，都无法达到需求。不过如果我们能够对 SwiftUI 的布局方式有更深入了解的话，在遇到这种情况时，就可以少一些胡乱猜测，多一些努力方向。

布局规则

SwiftUI 布局流程

SwiftUI 遵循的布局规则，可以总结为“协商解决，层层上报”：父层级的 View 根据某种规则，向子 View “提议”一个可行的尺寸；子 View 以这个尺寸为参考，按照自己的需求进行布局：或占满所有可能的尺寸（比如 Rectangle 和 Circle），或按照自己要显示的内容确定新的尺寸（比如 Text），或把这个任务再委托给自己的子 View 继续

进行布局 (比如各类 Stack View)。在子 View 确定自己的尺寸后，它将这个需要的尺寸汇报回父 View，父 View 最后把这个确定好尺寸的子 View 放置在座标系合适的位置上。

让我们回归本源，来看一个最简单的情况，结合这个实例说明布局流程：

```
struct ContentView: View {
    var body: some View {
        HStack {
            Image(systemName: "person.circle")
            Text("User:")
            Text("onevcat | Wei Wang")
        }
        .lineLimit(1)
    }
}

// SceneDelegate.swift
window.rootViewController =
    UIHostingController(rootView: ContentView())
```

结果如下，蓝框范围是 HStack 的尺寸：

 User: onevcat | Wei Wang

在这里，在 HStack 的父 View 是 ContentView，而 ContentView 直接就是 UIHostingController 的 rootView。SwiftUI 系统经历了以下步骤来进行布局：

1. rootView 使用整个屏幕尺寸作为“提议”，向 HStack 请求尺寸。
2. HStack 在接收到这个尺寸后，会向它的子 View 们进行“提议”。
3. 第一步，扣除掉默认的 HStack spacing 后，把剩余宽度三等分 (因为 HStack 中存在三个子 View)，并以其中一份向第一个子 View 的 Image 进行提议。
4. Image 会按照它要显示的内容决定自身宽度，并把这个宽度汇报给 HStack。
5. HStack 从 3 中向子 View 提案的总宽度中，扣除掉 4 里 Image 汇报的宽度，然后将剩余的宽度平分为两部分，把其中一份作为提案宽度提供给 Text("User:")。
6. Text 也根据自身内容来决定宽度。不过和 Image 不太一样，Text 并不“盲目”遵守自身内容的尺寸，而是会更多地尊重提案的尺寸，通过换行(在没有设定 .lineLimit(1) 的情况下) 或是把部分内容省略为“...”来修改内容，去尽量满足提案。注意，父 View 的提案对于子 View 来说只是一种建议。比如这个 Text 如果无论如何，需要使用的宽度都比提案要多，那么它也会将这个实际需要的尺寸返回。

- 对于最后一个 Text，采取的步骤和方法与 6 类似。在这里，由于我们没有过多约束，分配给两个 Text 的宽度也完全足够，因此它们都会按照内容宽度进行汇报。在三个子 View 都决定好各自尺寸后，HStack 会按照设定的对齐和排列方式把子 View 们水平顺序放置。
- 不要忘记，HStack 是 rootView 的子 View，因此，它也有义务将它的尺寸向上汇报。由于 HStack 知道了三个子 View 和使用的 spacing 的数值（在例子中我们使用了默认的 spacing 值），HStack 的尺寸也得以确定。最后它把这个尺寸（也就是图中的蓝框部分）上报。
- 最后，rootView 把 HStack 以默认方式放到自己的座标系中，也即在水平和竖直方向上都居中。

布局优先级

在上例中，布局系统在处理 HStack 的三个子 View 时，会按照顺序处理 Image，“User:” Text 和最后的用户名 Text。如果我们对它的 frame 进行一些限制，就可以看到 Text 在处理布局上的一些细节。对上面的代码做一些修改，将宽度限制到 200，为了清楚，我们还可以为它们加上一些背景颜色：

```
HStack {  
    Image(systemName: "person.circle")  
        .background(Color.yellow)  
    Text("User:")  
        .background(Color.red)  
    Text("onevcat | Wei Wang")  
        .background(Color.green)  
}  
.lineLimit(1)  
.frame(width: 200)
```



绿色部分的 Text 无法从父 View 中获得足够的宽度提案，因此它只能在用 “...” 截断字符串的前提下，尽可能使用所有的提案宽度。你可能已经注意到，在黄色 Image 左侧和绿色 Text 右侧都有一小段空隙，但这些空隙并不足以再支撑绿色 Text 再多显示一个字符。两侧都有空隙，而并非只有最右侧有空隙的原因，是 .frame 默认采取的是 .center 对齐。注意。这里的对齐方式和 HStack 无关，而是 .frame 所导致的效果。如果你为最后的 .frame(width: 200) 添加上对齐方式 (例如 .frame(width: 200, alignment: .leading))，就能看到实际效果：Image 将会靠到最左侧进行对齐。我们会在本章后面的部分深入探讨 frame 和各种对齐方式的本质。

有些情况下，在有限空间中布局多个 View 时，我们会希望某些更重要的部分优先显示。比如上例中，相对于 “User:” 这个描述，可能实际的用户名更加重要。通过 .layoutPriority，我们可以控制计算布局的优先级，让父 View 优先对某个子 View 进行考虑和提案。默认情况下，布局优先级都是 0，我们可以传递一个更大的值 (比如 1)，来让绿色的 Text 可以首先决定自己的尺寸，这样它就能显示全部内容：

```
HStack {  
    Image(systemName: "person.circle")  
        .background(Color.yellow)  
    Text("User:")  
        .background(Color.red)
```

```
Text("onevcat | Wei Wang")
    .layoutPriority(1)
    .background(Color.green)
}
.lineLimit(1)
.frame(width: 200)
```



强制固定尺寸

除去那些刻意而为的自定义绘制，SwiftUI 中默认情况下 View 所显示的内容的尺寸一般不会超出 View 自身的边界：比如 Text 会通过换行和截取省略来尽可能让内容满足边界。有些罕见情况下我们可能希望无论如何，不管 View 的可用边界如何设定，都要完整显示内容。这时候，可以使用 `fixedSize`。这个 modifier 将提示布局系统忽略掉外界条件，让被修饰的 View 使用它在无约束下原本应有的**理想尺寸**。

继续用上面的 View 为例，我们如果在 `frame` 之前添加 `fixedSize`，那么原本被缩略的“User:”也将被显示出来。

```
HStack {
    Image(systemName: "person.circle")
        .background(Color.yellow)
```

```
Text("User:")
    .background(Color.red)

Text("onevcat | Wei Wang")
    .layoutPriority(1)
    .background(Color.green)

}

.lineLimit(1)
.fixedSize()
.frame(width: 200)
```



不过，需要特别注意的是，代表 HStack 尺寸的蓝框，现在比它的实际内容要窄。这很可能不是我们真正所需要的，它会让这个 HStack 在参与和其他 View 相关的布局时变得很奇怪：SwiftUI 仍然会按照蓝框部分的尺寸来决定 HStack 的位置，有时这导致显示内容的重叠。

Frame

继续上面的例子，如果我们把 fixedSize 从 frame 之前拿到 frame 之后的话，会发现布局和不加 fixedSize 的时候完全一致。这至少给我们提供了一个很重要的暗示：这些 modifier 的调用顺序不同，可能会产生不同的结果。

究其原因，这是由于大部分 View modifier 所做的事情，并不是“改变 View 上的某个属性”，而是“用一个带有相关属性的新 View 来包装原有的 View”。frame 也不例外：它并不是将所作用的 View 的尺寸进行更改，而是新创建一个 View，并强制地用其指定的尺寸，对内容（其实也就是它的子 View）进行提案。这也是为什么将 fixedSize 写在 frame 之后会变得没有效果的原因：因为 frame 这个 View 的理想尺寸就是宽度 200，它已经是按照原本的理想尺寸进行布局了，再用 fixedSize 包装也不会带来任何改变。

除了直接指定宽高的 frame(width:height:alignment:) 以外，我们也在之前章节多次看到过另一方法：

```
func frame(  
    minWidth: CGFloat? = nil,  
    idealWidth: CGFloat? = nil,  
    maxWidth: CGFloat? = nil,  
    minHeight: CGFloat? = nil,  
    idealHeight: CGFloat? = nil,  
    maxHeight: CGFloat? = nil,  
    alignment: Alignment = .center  
) → some View
```

和固定宽高的版本不同，这个方法为尺寸定义了一套约束：如果从父 View 中获得的提案尺寸小于 minXXX 或者大于 maxXXX，这个 frame 将会把这个提案尺寸截取到相应的最小值或者最大值，然后进行提案。frame 方法的两种版本里，所有的参数都有默认值 nil，如果你使用这个默认值，那么 frame 将不在这个方向上改变原有的尺寸提案，而是将它直接传递给子 View。

frame 方法的最后一个参数表示所使用的对齐方式。不过，很多时候单纯地改变这个对齐方式不会有任何效果：

```
HStack {  
    // ...  
}  
.frame(alignment: .leading)  
.background(Color.purple)
```

因为这个 alignment 指定的是 frame View 中的内容在其内部的对齐方式，如果不指定宽度或者高度，那么 frame 的尺寸将完全由它的内容决定。换言之，内容都已经占据了 frame 的全部空间，不论采用哪种方式，内容在 frame 里都是“贴边的”。对齐也就没有任何意义了。想要体现和实验 frame 里的对齐方式，可以为 frame 添加一个多余内容所需空间的尺寸参数：

```
HStack {  
    // ...  
}  
.frame(width: 300, alignment: .leading)  
.background(Color.purple)
```

结果为：



Alignment Guide

SwiftUI 中有不少 API 涉及到对齐，这也是最让人困惑的地方之一。你时不时总会遇到疑问：为什么写在这里的对齐不起作用？为什么这里的对齐方式表现非常奇怪？在文档中某个有关对齐的 API 并没有任何解释，它是不是并非准备给一般开发者使用的？

由于 SwiftUI 还处于早期阶段，所以对于上面这些问题，很多时候都可能被误认为是 bug 或者框架还不完善的结果，但大部分时候这并不是事实。上一节里，我们看到了 frame 中的对齐设定。在这一小节，我们会专注来看看各类 Stack View 的对齐，以及它所对应的 Alignment Guide 的相关概念。

Stack View 的对齐

HStack, VStack 和 ZStack 的初始化方法都可以接受名为 alignment 的参数，不过它们的类型却略有不同：

- HStack 接受 VerticalAlignment，典型值为 .top、.center、.bottom、lastTextBaseline 等。
- VStack 接受 HorizontalAlignment，典型值为 .leading、.center 和 .trailing。
- ZStack 在两个方向上都有对齐的需求，它接受 Alignment。Alignment 其实就是对 VerticalAlignment 和 HorizontalAlignment 组合的封装。

三种具体的对齐类型中都定义了 .center，它也是默认情况下各类 Stack 所定义的对齐方式。不过由于重名，在一些既可以接受 VerticalAlignment 又可以接受 HorizontalAlignment 的地方，简单地使用 .center 会导致歧义，这种情况下，我们可能会需要在前面加上合适的类型名称。

让我们仔细看看 `VerticalAlignment` 和 `HorizontalAlignment` 的这些值到底都是什么。以 `VerticalAlignment.top` 为例，它其实是定义在 `VerticalAlignment` extension 里的一个静态变量：

```
extension VerticalAlignment {  
    static let top: VerticalAlignment  
    // ...  
}
```

`VerticalAlignment` 本身也提供了一个初始化方法，它接受一个 `AlignmentID` 的类型作为参数：

```
struct VerticalAlignment {  
    init(_ id: AlignmentID.Type)  
}
```

也就是说，如果我们能定义一个自己的 `AlignmentID`，我们就可以创建 `VerticalAlignment` 的实例，并把它用在 `HStack` 的对齐上了。

那么 `AlignmentID` 又是什么呢？它是一个只有单个方法的 `protocol`：

```
protocol AlignmentID {  
    static func defaultValue(  
        in context: ViewDimensions  
    ) -> CGFloat  
}
```

这个方法需要返回一个 `CGFloat`，该数字代表了使用对齐方式时 `View` 的偏移量。我们当然可以简单地返回一个数字，不过，更常见的做法是从 `context` 里获取需要的值。`ViewDimensions` 的定义如下：

```
struct ViewDimensions {  
    // 1  
    var width: CGFloat { get }  
    var height: CGFloat { get }  
  
    // 2  
    subscript(  
        guide: HorizontalAlignment) -> CGFloat { get }  
    subscript(  
        guide: VerticalAlignment) -> CGFloat { get }  
  
    // 3  
    subscript(  
        explicit guide: HorizontalAlignment) -> CGFloat? { get }  
    subscript(  
        explicit guide: VerticalAlignment) -> CGFloat? { get }  
}
```

1. 当前处理的 View 的宽和高，这是很直接的数据。
2. 通过 `HorizontalAlignment` 或者 `VerticalAlignment` 以下标的方式从 `ViewDimensions` 中获取数据。默认情况下，它会返回对应 `alignment` 的 `defaultValue` 方法的返回值。
3. 通过下标获取定义在 View 上的**显式**对齐值。关于对齐的“显式”和“隐式”的区别，我们稍后再说。

举个实际的例子，如果我们想要自己手动实现一个 `VerticalAlignment.center` (在下面我们把它叫做 `myCenter`)，可以这样进行定义：

```
extension VerticalAlignment {
```

```
struct MyCenter: AlignmentID {
    static func defaultValue(
        in context: ViewDimensions
    ) → CGFloat {
        context.height / 2
    }
}

static let myCenter = VerticalAlignment(MyCenter.self)
}
```

在 defaultValue(in:) 里，我们指定对齐位置为 height 值的一半，这恰好就是竖直方向上各个 View 的水平中线所在位置。将 HStack 的对齐方式替换为 .myCenter，我们能得到的布局和默认的 .center 完全一样：

```
HStack(alignment: .myCenter) {
    Image(systemName: "person.circle")
        .background(Color.yellow)
    Text("User:")
        .background(Color.red)
    Text("onevcat | Wei Wang")
        .background(Color.green)
}
.lineLimit(1)
.background(Color.purple)
```

隐式对齐和显式对齐

通过 alignmentGuide，我们可以进一步调整 View 在容器（比如各类 Stack）中的对齐方式，这提供给我们更多的灵活性。alignmentGuide modifier 有两个重载方法：

```
func alignmentGuide(  
    _ g: HorizontalAlignment,  
    computeValue: @escaping (ViewDimensions) → CGFloat  
) → some View  
  
func alignmentGuide(  
    _ g: VerticalAlignment,  
    computeValue: @escaping (ViewDimensions) → CGFloat  
) → some View
```

和之前定义 Alignment 时类似，这个方法涉及到一个 CGFloat 值。alignmentGuide 所做的事情，是负责修改 g (HorizontalAlignment 或者 VerticalAlignment) 的对齐方式，把原来的 defaultValue(in:) 所提供的默认值，用 computeValue 返回的 CGFloat 值进行替代。对于容器里的 View，如果我们不明确指定 alignmentGuide，它们都将继承使用容器的对齐方式。举例来说，如果我们在 HStack 里不指定任何对齐：

```
HStack {  
    Image(systemName: "person.circle")  
    Text("User:")  
    .font(.footnote)  
    Text("onevcat | Wei Wang")  
}
```

实际上，HStack 默认使用 VerticalAlignment.center，且 Image 和两个 Text 都使用 .center 的默认值作为**隐式**对齐。如果将所有对齐**显式**写出来，这段代码相当于：

```
HStack(alignment: .center) {  
    Image(systemName: "person.circle")  
    .alignmentGuide(VerticalAlignment.center) { d in
```

```
d[VerticalAlignment.center]
}

Text("User:")
    .font(.footnote)
    .alignmentGuide(VerticalAlignment.center) { d in
        d[VerticalAlignment.center]
    }

Text("onevcat | Wei Wang")
    .alignmentGuide(VerticalAlignment.center) { d in
        d[VerticalAlignment.center]
    }
}
```

每个 alignmentGuide 都返回了对应 VerticalAlignment.center 的默认值。对于想要使用这个默认对齐的 View，我们可以省略掉 alignmentGuide。如果我们想要对某个部分进行微调，可以在 computeValue 中进行计算。比如让 “User:” 变成 “上标” 形式（注意，我们顺便调整了一下文本和图片的顺序）：

```
// 1
HStack(alignment: .center) {
    Text("User:")
        .font(.footnote)
    // 2
    .alignmentGuide(VerticalAlignment.center) { d in
        d[.bottom]
    }
    // 3
    Image(systemName: "person.circle")
    Text("onevcat | Wei Wang")
```

}

1. 对整个 HStack，我们指定了 VerticalAlignment.center 为对齐方式
2. 对于 “User:”，返回的是 d[.bottom]，也即使用 Text 的底边作为对齐线。注意，只有当 alignmentGuide 的第一个参数 VerticalAlignment.center 和外层容器 HStack 的 alignment 参数一致时，它才会被考虑。因为 alignmentGuide API 的作用就是修改传入的 alignment 的数值。
3. Image 和最后一个 Text 没有显式指定 alignmentGuide，它们将使用默认的 d[VerticalAlignment.center]，也即 height / 2 水平中线作为对齐线。

所以，上面代码的效果是，让 Text("User:") 的底边与 Image 和实际用户名 Text 的中线对齐，如图：



当然，除了直接使用 d[.bottom]，你也可以进行计算并返回任意的对齐数值，比如 d[.bottom] + 5 或者 d.height * 0.7 等，从而达到设计上的任何要求。

需要特别指出的是，alignmentGuide 中指定的 alignment 必须要和 HStack 这类容器所指定的 alignment 一致，它才会在布局时被考虑。不过，对于那些和当前对齐不相关的 alignmentGuide，如果有需要，我们可以通过 ViewDimensions 的 explicit 下标方法读取。和普通的下标方法不同，这个方法返回的是可选值

CGFloat?。如果当前 View 上没有显式定义相关的对齐，那么会得到 nil。这在设定自定义对齐中，需要考虑其他方向的对齐时会很有用：

```
HStack(alignment: .center) {
    Text("User:")
        .font(.footnote)
        .alignmentGuide(.leading) { _ in
            10
        }
        .alignmentGuide(VerticalAlignment.center) { d in
            d[.bottom] + (d[explicit: .leading] ?? 0)
        }
    Image(systemName: "person.circle")
    Text("onevcat | Wei Wang")
}
```

上例中，如果显式定义了 .leading，则在计算 center 这个 alignmentGuide 实际作用时，就可以通过 explicit 的下标方法读取它，并将它加到对齐中去。在这里的 HStack 里，可能看不太出这么做的意义。不过在 ZStack 中，同时会涉及到水平和竖直两种情况，如果两个方向上的对齐方式具有相关性，那么 d[explicit:] 就非常有用了。

自定义 Alignment 和跨 View 对齐

上例中我们已经通过创建满足 AlignmentID 的 MyCenter，自定义了一个 VerticalAlignment 值。我们可以通过类似的方法，定义出任意多个对齐。不过，如果只是像上例那样的微调的话，还不需要“大动干戈”定义新的对齐。新建对齐的主要目的，还是为了跨越 View 的层级来进行对齐。

例如，我们想要实现下图所示的布局，用来让用户选择想要使用的用户名称：

User: onevcat | Wei Wang
zaq | Hao Zang
② tyyqa | Lixiao Yang

这个布局以上例为基础，最外层是一个 HStack，它包含三个元素：上标的“User”Text，表示当前选中的用户的 Image 图标，以及一个由 VStack 组成的用户列表。在点击某行时，我们希望 Image 移动到和被选中行对齐的位置。这就需要我们拥有跨 View 对齐的手段。

首先，我们可以添加一个自定义对齐：

```
extension VerticalAlignment {  
    struct SelectAlignment: AlignmentID {  
        static func defaultValue(  
            in context: ViewDimensions  
        ) -> CGFloat {  
            context[VerticalAlignment.center]  
        }  
    }  
  
    static let select =  
        VerticalAlignment(SelectAlignment.self)  
}
```

接下来，将它指定为外层 HStack 的 alignment：

```
@State var selectedIndex = 0

let names = [
    "onevcat | Wei Wang",
    "zaq | Hao Zang",
    "tyyqa | Lixiao Yang"
]

var body: some View {
    HStack(alignment: .select) {
        Text("User:")
            .font(.footnote)
            .foregroundColor(.green)
        Image(systemName: "person.circle")
            .foregroundColor(.green)
    }
    VStack(alignment: .leading) {
        ForEach(0 ..< names.count) { index in
            Text(self.names[index])
                .foregroundColor(
                    self.selectedIndex == index ? .green : .primary
                )
                .onTapGesture {
                    self.selectedIndex = index
                }
        }
    }
}
```

因为 `SelectAlignment` 默认返回的对齐值是 `context[VerticalAlignment.center]`，上面的更改中，所有的子 `View` 都隐式地使用了这个默认值，它和简单的 `.center` 对齐没有区别，`HStack` 的三部分都中央对齐：

onevcat | Wei Wang
User:  zaq | Hao Zang
tyyqa | Lixiao Yang

接下来，为三部分设定各自的对齐行为，为了清晰一些，下面只写出了需要添加 `alignmentGuide` 的部分：

```
HStack(alignment: .select) {  
    Text("User:")  
    // ...  
    // 1  
    .alignmentGuide(.select) { d in  
        d[.bottom] + CGFloat(self.selectedIndex) * 20.3  
    }  
    Image(systemName: "person.circle")  
    // ...  
    // 2  
    .alignmentGuide(.select) { d in  
        d[VerticalAlignment.center]  
    }  
    VStack(alignment: .leading) {
```

```
ForEach(0 ..< names.count) { index in
    Text(self.names[index])
    // ...
    // 3
    .alignmentGuide(
        self.selectedIndex == index ? .select : .center
    ) { d in
        if self.selectedIndex == index {
            return d[VerticalAlignment.center]
        } else {
            return 0
        }
    }
}
}
}
}
// 4
.animation(.linear(duration: 0.2))
```

1. 对于上标 Text，以底部为基准，再加上选中的行到整个 HStack 上端的总高度。这会将“User:”上标文本固定在 HStack 最上方且超出自身一半高度的位置上。
2. 明确指定 Image 的中心部位应该和其他部分对齐。
3. VStack 中的这个显式 alignmentGuide 将会覆盖其他隐式行为。VStack 有自己的布局规则，那就是顺次将每个子 View 竖直放置。在这个基础上，我们把被选中行的中线位置设定成了对齐位置。这样，SwiftUI 将尝试在满足 VStack 的竖直叠放特性的同时，去满足把选中行和 HStack 中其他部分的对齐。而对

于没有被选中的 Text，我们随意为它设置一个不是 .select 的对齐。由于顶层容器 HStack 指定了 .select，所以任意非 .select 的对齐方式都将被忽略。

4. 最后，为了让选择切换更自然一些，可以为整个效果加上动画。

经过这些修改，这个用户选择列表就可以完整工作了。相关代码可以在源码文件夹的“11.Layout”中找到。

总结

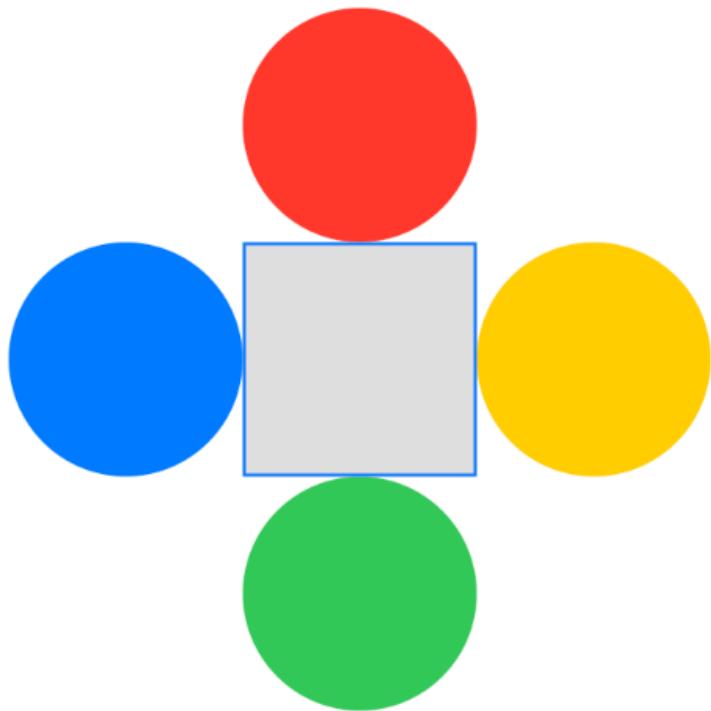
本章里，我们研究了关于 SwiftUI 布局的一些进阶话题。就算不知道这些知识，你可能也可以通过不断尝试和修改，最终找到满足要求的布局写法。但另一方面，我们需要看到，就算是像 frame 或者 alignment 这样的每天都在使用的简单方法背后，所蕴藏的规则和使用方式，也远不止看起来那样容易。大致了解 SwiftUI 布局背后的原理，以及掌握常用的布局方法，有助于你迅速确定 View 之间的关系，达到事半功倍的效果。

虽然乍看起来有这样那样的问题，但 SwiftUI 的布局系统是经过精心设计的。它使用了声明式的方法让开发者通过描述语句来布局。和传统的 Auto Layout 不同，SwiftUI 中的描述性布局不会出现冲突或者缺失，也不存在由于运行时的布局而导致错误的可能性。它提供的是一种安全的布局方式：只要能够用 SwiftUI 的语句进行描述并通过编译，布局系统就会按照一定的规律生成合理的满足描述的布局。问题在于，你是否足够熟悉这套规律和机制，解释布局语句所带来的结果，并让代码朝向你希望的方向进行改变。

练习

1. 灵活使用 GeometryReader

请尝试用 GeometryReader 和 Circle 画出以下图形：



其中灰色的矩形蓝框部分的尺寸由 `frame` 给出，且四周的圆形直径和矩形短边长度一致。你可以从下面的代码片段开始：

```
var body: some View {  
    GeometryReader { proxy in  
        // ...  
        // 使用 Circle 绘制  
    }  
}
```

```
.frame(width: 100, height: 100)  
.background(Color.gray.opacity(0.3))  
}
```

提示，Circle 也是 Shape 的一种，因此可以使用 Circle().path(in: rect) 的方式将一个 Circle 重绘到另外的 rect 中。当然，你也可以使用像是 offset 这样的 modifier 来进行移动。注意我们要求对于任意矩形都能正确绘制四周贴边的圆形，你可以自行更改 frame 的数值，来确认你的实现对任意矩形都是通用的。

2. 研究子 View 对提案尺寸的敏感度

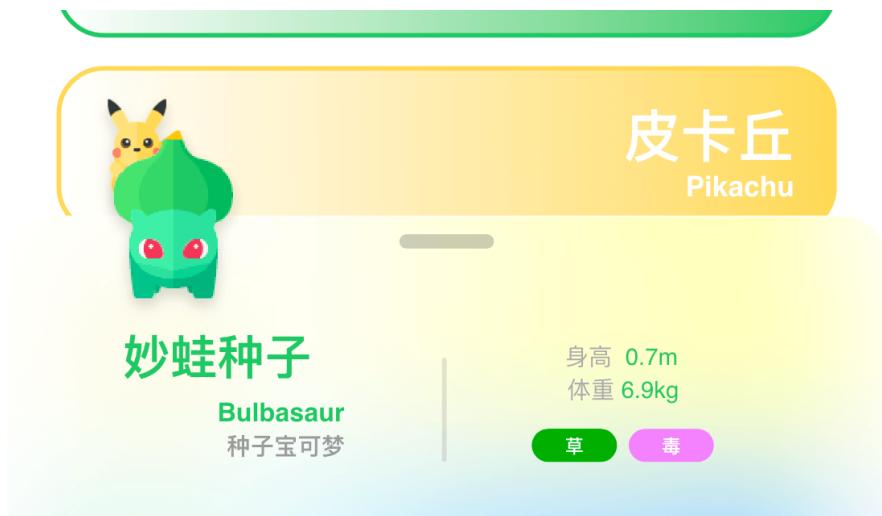
在本章中布局例子中：

```
HStack {  
    Image(systemName: "person.circle")  
    Text("User:")  
    Text("onevcat | Wei Wang")  
}  
.lineLimit(1)  
.frame(width: 200)
```

将 .frame 的 width 设定为一个很小的值时 (比如比 Image 的宽度还小)，会发生什么。如果保持 width 足够大，而是将 height 设定为一个很小的值时，又会发生什么？请总结一下 Image, Text, Rectangle 和 HStack 各自对于 frame 宽高提案是如何响应的，它们是严格遵守提案尺寸呢，还是更多地去满足内容的尺寸？

3. ZStack 的复杂布局

尝试修改最终的 PokeMaster app，让详细信息面板的 ZStack 的对齐方式更加“灵活炫酷”一些：



主要来说，希望实现：

1. 详细面板的图标实现“骑墙”的对齐效果，图标的中部和面板背景上边缘齐平。
2. 宝可梦的图标和它的中文名字（“妙蛙种子”）首端 (.leading) 对齐。
3. 宝可梦的英文名字（“Bulbasaur”）的 .leading 与它的中文名字（“妙蛙种子”）的 .center 对齐。
4. 宝可梦的英文名字（“Bulbasaur”）和它的种族名字（“种子宝可梦”）尾端 (.trailing) 对齐。

信息面板的内容和背景是以 ZStack 的关系组织起来的，与 HStack 和 VStack 不同，ZStack 可以同时接受水平和竖直两个方向上的对齐。这也是使用多个 alignmentGuide 进行不同方向对齐的先决条件。

你可以使用源码文件夹中“PokeMaster-Finished”里的项目作为本题练习的起始。