objc.io | objc 中国

SwiftUI 编程思想

Chris Eidhof, Florian Kugler 著
王巍 译

英文版本 1.0 (2020 年 3 月)，中文版本 1.0 (2020 年 4 月)

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

介绍 **5**

1 概览 6

View 的创建 **8**
View 布局 **14**
View 更新 **17**
重点 **18**

2 View 更新 19

更新 View 树 **20**
状态属性标签 **30**
重点 **37**
练习 **37**

3 环境 41

环境是如何工作的 **42**
使用环境 **45**
依赖注入 **50**
Preferences **51**
重点 **55**
练习 **55**

4 布局 57

基本 View **59**
布局修饰器 **63**
Stack View **70**
组织布局代码 **76**
重点 **79**

练习 **80**

5 自定义布局 **82**

几何读取器 **83**
锚点 **88**
自定义布局 **92**
重点 **96**
练习 **96**

6 动画 **99**

隐式动画 **100**
动画是如何工作的 **102**
显式动画 **106**
自定义动画 **107**
重点 **111**
练习 **112**

总结 **116**

习题答案 **118**

第 2 章：View 更新 **119**
第 3 章：环境 **122**
第 4 章：布局 **124**
第 5 章：自定义布局 **126**
第 6 章：动画 **130**

介绍

SwiftUI 与 UIKit，AppKit 和类似的面向对象的 UI 框架有着天壤之别。

在 SwiftUI 中，view 是值，而非对象。相较于在面向对象的框架中的处理方式，view 的创建和更新是以完全不同的声明式方式完成的。尽管这消除了一整类错误 (view 和 app 的状态不同步)，但这也意味着你必须转变思考方式，你需要重新思考如何用 SwiftUI 代码来表达你的想法。本书的主要目标，就是帮助你发展出以这种新方法来进来思考的直觉。

SwiftUI 也带有适合其声明式特性的布局系统。布局系统的核心非常简单，但一开始它的行为可能会表现得有些复杂。为了解决这个问题，我们将说明基本的 view 和 view 容器的布局行为，以及要如何组合使用它们。我们还将展示创建复杂的自定义布局所需要的先进技术。

最后，本书也涵盖了动画方面的话题。像 SwiftUI 中的所有 view 更新一样，动画也是由状态变更所触发的。我们会考察包括从隐式动画到自定义动画的几个示例，来展示如何使用这个新的动画系统。

这本书没有什么

由于 SwiftUI 还是一个很年轻的框架，因此本书并不是完整的 (特定平台) SwiftUI API 文档。例如，我们不会详细介绍如何在 iOS 上使用 NavigationView，如何在 macOS 上使用 SplitView，或者如何在 watchOS 上使用轮播等。这是因为，这些特定平台的 API 很可能随着开发在未来几年中发生变化。相反，本书会着重于 SwiftUI 背后的概念，我们认为对这些概念的理解是必不可少的，它们将让你为接下来十年的 SwiftUI 开发做好准备。

致谢

感谢 Javier Nigro，Matt Gallagher 和 Ole Begemann 对我们的书提供的宝贵反馈。感谢 Natalye Childress 进行的出版编辑。Chris 在这里也想感谢 Erni 和 Martina 为他提供了一个舒适的写作环境。

概览

1

在本章中，我们将概述 SwiftUI 的工作原理，以及它与 UIKit 等框架的不同之处。SwiftUI 在概念上与以前的 Apple 平台上开发 app 的方式完全不同，它需要你重新考虑如何将你脑中的构想转换为实际可工作的代码。

我们将介绍一个简单的 SwiftUI app，来探索如何创建，布局和更新 view。希望这可以让你首先了解使用 SwiftUI 所需的新思维模型。在随后的章节中，我们将深入探讨本章中所描述的各个方面。

作为示例 app，我们会构建一个简单的计数器。这个 app 有一个增加计数的按钮，在按钮下方有一个标签 (label)。当计数按钮被点击过时，这个标签会显示点击的次数；在没有被点击的情况下，它会显示一个占位符：

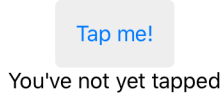


Figure 1.1: app 启动状态时...

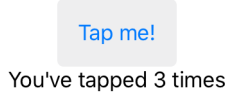


Figure 1.2: ...点击几次后

我们强烈建议你跟着我们的例子，自己来运行和修改代码。毕竟有句话叫做：

> 学习一种新的编程语言的唯一方法，就是用它编写程序。— Dennis Ritchie

我们认为这条建议不仅适用于编程语言，也适用于像是 SwiftUI 这样的复杂框架。实际上，这也是作为作者的我们在学习 SwiftUI 时所经历过的事情。

View 的创建

要在 SwiftUI 中创建 view，你需要创建一棵包含 view 的值的树，来描述应该在屏幕上显示的内容。要更改屏幕上的内容，你可以修改 state 值，这样新的 view 值的树会被重新计算。然后，SwiftUI 会更新屏幕，以反映这些新的 view 值。例如，当用户点击计数按钮时，我们应该增加状态值，并让 SwiftUI 重新渲染 view 树。

> 注意：在撰写本文时，Xcode 的 SwiftUI 内置预览，Playground 以及模拟器并不总是能正确工作。当你看到意外的行为时，请确保在真实设备上再次进行检查，因为在某些情况下，即使模拟器的行为也可能和真实设备有所不同。

这是整个计数器 app 的 SwiftUI 代码：

```
import SwiftUI

struct ContentView: View {
  @State var counter = 0
  var body: some View {
    VStack {
      Button(action: { self.counter += 1 }, label: {
        Text("Tap me!")
          .padding()
          .background(Color(.tertiarySystemFill))
          .cornerRadius(5)
      })

      if counter > 0 {
        Text("You've tapped \(counter) times")
      } else {
        Text("You've not yet tapped")
      }
    }
  }
}
```

ContentView 包含一个带有两个嵌套 view 的垂直堆栈 (stack)：其中包括一个按钮，它在点击时会去增加 counter 属性；另一个是用来显示点击次数或者占位符文本的文本标签。

请注意，按钮在 action 闭包中并没有直接去对点击次数的 Text view 进行更改。闭包没有对 Text view 的引用进行捕获。而且就算是引用了，程序对 SwiftUI 中的 view 在屏幕上呈现之后所进行的常规属性修改，也不会改变它在屏幕上的呈现方式。相反，我们必须修改状态 (在本例中为 counter 属性)，这会导致 SwiftUI 调用 view 的 body，并使用 counter 的新值来生成 view 的新描述。

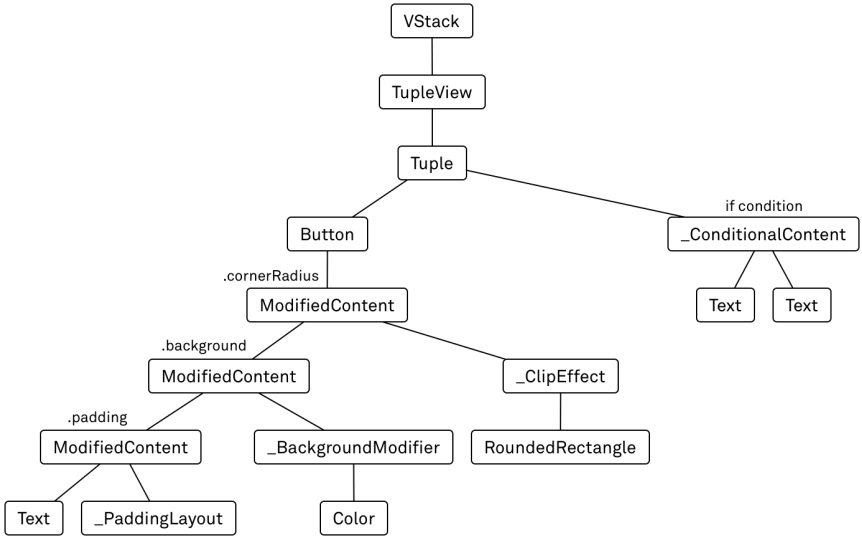
注意 view 的 body 属性的类型，它是 some View，这个类型并没有包含关于正在创建的 view 的树的更多信息。它只是说，无论 body 返回的确切类型到底是什么，都可以保证该类型肯定满足 View 协议。在我们的例子中，body 的真实类型如下：

```
VStack<
  TupleView<
    (
      Button<
        ModifiedContent<
          ModifiedContent<
            ModifiedContent<
              Text,
              _PaddingLayout
            >,
            _BackgroundModifier<Color>
          >,
          _ClipEffect<RoundedRectangle>
        >
      >,
      _ConditionalContent<Text, Text>
    )
  >
>
```

这是一个带有许多泛型参数的庞大类型，它在瞬间就为我们解释了为什么我们会需要创建一个 some View (它是一个非透明类型) 来把这些复杂的 view 类型进行抽象。不过，如果是为了学习的话，探究这个类型的细节还是有益的。

> 为了检查 body 的实际类型，我们创建了下面这个辅助函数，它使用了 Swift 的 Mirror API：
>
> ```
> extension View {
>   func debug() -> Self {
>     print(Mirror(reflecting: self).subjectType)
>     return self
>   }
> }
> ```
>
> 这个函数在 body 被执行时可以打印出 view 的类型：
>
> ```
> var body: some View {
>   VStack { /*... */ }.debug()
> }
> ```

用一个树形图来对这个类型进行可视化的话，我们得到：



首先要注意的是，在 body 属性中构造的 view 的类型，包含了整个 view 树的结构：它不仅包含了当前屏幕上显示的部分，还包含了 app 生命周期中可能会在屏幕上显示的所有 view。if 语句会被编码为一个 _ConditionalContent 类型的值，其中包含两个分支的类型。你可能想知道这是怎么做到的，if 难道不是一个语言层级的概念吗？它难道不应该是在运行时才被求值的吗？

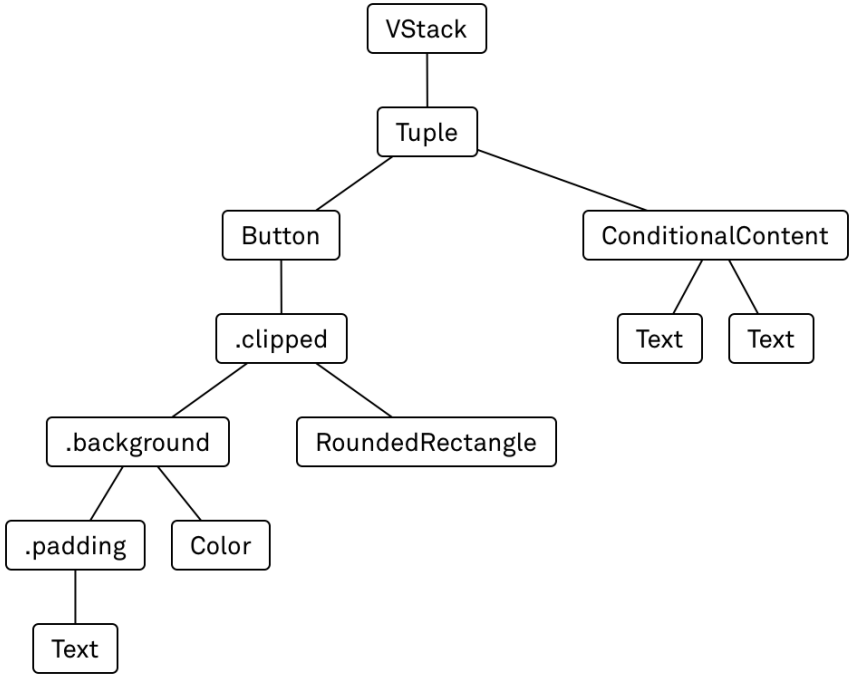
为了实现这一点，SwiftUI 利用了称为函数构建器 (function builder) 的 Swift 特性。举个例子，VStack 之后的尾随闭包并不是一个普通的 Swift 函数；它是一个 ViewBuilder (它是由 Swift 的函数构建器特性实现的)。在 view 的构建闭包中，你只能使用 Swift 的一个非常有限的子集来编写程序：例如，你不能使用循环、guard 或 if let。但是，你可以编写简单的 if 语句来构造出依赖于 app 当前状态的 view 树 — 就像上面示例中的 counter 变量一样 (有关 view 构建器的更多详细信息，请参见下面的部分)。

View 的树不仅只包含当前可见的部分，它包含的是整个结构，这是有优点的：SwiftUI 能够更有效地找出 view 更新后发生了什么变化，我们将在本章稍后介绍 view 的更新。

关于这个类型第二个需要特别强调的是 ModifiedContent 值的深层嵌套。我们在按钮上使用的 padding、background 和 cornerRadius API 并不是简单地去更改按钮的属性。实际上，这些方法 (我们通常称其为 “修饰器”) 的调用都会在 view 树中创建新的一层。在按钮上调用 .padding() 会将按钮包装为 ModifiedContent 类型的值，这个值中包含有关应该如何设置 padding 填充的信息。在该值上再调用 .background，又会把现有值包装起来，创建另一个 ModifiedContent 值，这一次将添加有关背景色的信息。注意，.cornerRadius 是通过把 view 用圆角矩形进行裁剪来实现的，这也会反映在类型中。

因为所有这些修改操作都会在 view 树中创建新层，因此它们的顺序通常很重要。调用 .padding().background(...) 与调用 .background(...).padding() 是不一样的。在前一种情况下，背景将延伸到填充部分的外边缘；而在后一种情况下，背景只会出现在填充范围的内侧。

在本书的其余部分，为了简洁，我们将简化图表，省略诸如 ModifiedContent 的内容。例如，对于前一张图，下面是其简化版本：



View 构建器

如上所述，SwiftUI 非常依赖于 view 构建器来构建 view 树。View 构建器看起来类似于常规的 Swift 闭包表达式，但是它仅支持非常有限的语法。虽然你可以编写返回 View 的任何形式的**表达式**，但是你可以使用的**语句**很少。以下示例中包含了 view 构建器中几乎所有可能的语句：

```
VStack {
  Text("Hello")
  if true {
    Image(systemName: "circle")
  }
  if false {
    Image(systemName: "square")
  } else {
    Divider()
```

```
  }
  Button(action: {}, label: {
    Text("Hi")
  })
}
```

上面的视图类型是：

```
VStack<
  TupleView<(
    Text,
    Optional<Image>,
    _ConditionalContent<Image, Divider>,
    Button<Text>
  )>
>
```

构建器中的每个语句都会被转换为不同的类型：

- → 具有单个语句 (例如，按钮的标签) 的 view 构建器的求值就是该语句的类型 (在本例中的 Text)。
- → 构建器中只有 if 而没有 else 的语句，将成为一个可选值。例如，if true { Image(...) } 的类型为 Optional<Image>。
- → if/else 语句会变成 _ConditionalContent。例如，上面的 if/else 被转换为 _ConditionalContent<Image, Divider>。请注意，在 view 构建器的内部，为 if/else 语句的每个分支指定不同的类型是完全可以的 (不过，在 view 构建器外部，if 语句的不同分支只能返回同样的类型)。
- → 多个语句被转换为带有一个元组 (tuple) 的 TupleView，该元组的每个元素都对应一个语句。例如，我们传递给 VStack 的 view 构建器包含有四个语句，得到的类型是：

```
TupleView<(
  Text,
  Optional<Image>,
  _ConditionalContent<Image, Divider>,
  Button<Text>
)>
```

目前，我们还无法编写循环或 switch 语句，不能声明变量，也不能使用 if let 之类的语法。这些语句大多数会在将来被支持，但是在本书写作的时候，它们还没有被实现。

与 UIKit 相比

当我们谈论 UIKit 中的 view 或 view controller 时，我们指的是 UIView 或 UIViewController **类的实例**。UIKit 中的 view 创建，意味着建立 view controller 和 view **对象**的树，稍后我们可以对其进行修改以更新屏幕上的内容。

SwiftUI 中的 view 创建指的是完全不同的过程，因为 SwiftUI 中没有 view 类的实例。当我们谈论 view 时，我们所说的是符合 View **协议**的**值**。这些值描述了应该在屏幕上显示的内容，但和 UIKit 的 view 不同，这些值与你在屏幕上看到的内容并没有一对一的关系：SwiftUI 中的 view 值是暂时的，它们可以随时被重新创建。

和 UIKit 相比，另一个很大的不同是，UIKit 中计数器 app 的 view 创建仅仅是必要代码的一部分；为了修改计数，你还必须为按钮实现事件处理的程序，从而触发对文本标签的更新。在 UIKit 中，view 的创建和 view 的更新是两条不同的代码路径。

在上面的 SwiftUI 示例中，这两个代码路径合二为一了：我们不必编写多余的代码来更新屏幕上的文本标签。每当状态改变时，view 树都会被重建，然后 SwiftUI 负责确保屏幕内容反映出 view 树中的描述。

View 布局

SwiftUI 的布局系统明显不同于 UIKit 中基于约束或基于 frame 的系统。在本节中，我们将带你了解基础知识，并在本书后面的view 布局一章中对该话题进行扩展。

SwiftUI 从最外层的 view 开始布局过程。在我们的例子中，是包含着单个 VStack 的 ContentView。因为 ContentView 是 view 层级的根节点，布局系统会为它提供整个屏幕边界以进行布局。接下来，ContentView 把相同的尺寸提供给 VStack 让它自行布局。VStack 根据其子 view 的数量，将可用空间进行划分，并将它提供给每个子 view (这里我们过度简化了堆栈是如何在其子 view 之间划分可用空间的过程，我们稍后会在布局章节再讨论这个话题)。在我们的示例中，VStack 将询问 (包裹在多个修饰器中的) 按钮和其下方的条件文本标签，并决定如何划分空间。

堆栈的第一个子元素 (按钮) 包裹在三个修饰器中：第一个 (cornerRadius) 负责剪切圆角，第二个 (background) 定义背景色，第三个 (padding) 添加间隙填充。前两个修饰器不会修改 view 所提议的尺寸。但是，padding 修饰器将占用其父 view 提供的空间，减去 padding，然后为按钮提供略微减小的空间。接下来，这个按钮为其中的标签提供这个空间，标签则根据文本内容实际需要的大小进行响应。该按钮将选用文本标签的大小，padding 修饰器则选用按钮的尺寸再加上填充的大小，其他两个修饰器直接使用其子元素的大小，最终的尺寸被向上传给 VStack。

VStack 与它的第二个子 view (即条件文本标签) 经过相同的处理后，它就可以确定自己的大小，并报告给其父对象了。最初，整个屏幕的边界都被 ContentView 提供给了 VStack，但是由于它所需要的空间要小得多，所以默认情况下布局算法会将其居中显示在屏幕上。

一开始的时候，SwiftUI 中的 view 布局感觉有点像 UIKit：设置 frame 和使用 stack view。但是，在 SwiftUI 里，因为我们只是在描述屏幕上应该显示的内容，所以我们永远不会去直接设置一个 view 的 frame 属性。举例来说，将下面的内容添加到 VStack 上时，看起来好像我们正在设置 frame，但事实并非如此：

```
VStack {
  // ...
}.frame(width: 200, height: 200)
```

当调用 .frame 时，我们所做的其实是将这个 VStack 包装到了另一个修饰器中 (这个修饰器自身也满足 View 协议)。现在，view 的 body 的类型会变成：

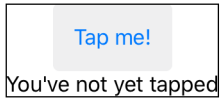
```
ModifiedContent<VStack<...>, _FrameLayout>
```

这次，整个屏幕空间都将提供给 frame 修饰器，该修饰器又将 (200, 200) 的空间提供给 VStack。堆栈最终的大小仍然会和之前的大小相同，在默认情况下位于 (200, 200) 的 frame 修饰器的中心。需要牢记的是，像是 .frame 和 .offset 这类 API 并不会修改 view 的属性，它们所做的是将 view 包装在修饰器中。当你尝试将这些调用与其他像是背景或边框等组合在一起时，会出现很多的不同。

假设我们要为垂直堆栈中指定的 (200, 200) frame 添加边框。首先，我们可以尝试如下操作：

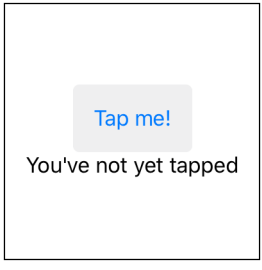
```
VStack {
  // ...
}
```

```
.border(Color.black)
.frame(width: 200, height: 200)
```



可能你会有些惊讶，边框只出现在垂直堆栈的最小区域周围，而不是显示在我们指定的 200 x 200 的区域中。原因是 .border 调用在垂直堆栈周围添加了 overlay 的修饰器，该修饰符使用其子元素的大小。如果要在整个 (200, 200) 区域周围绘制边框，我们必须反过来调用：

```
VStack {
  // ...
}
.frame(width: 200, height: 200)
.border(Color.black)
```



这次，frame 修饰器被包装到了 overlay 修饰器中，因此这个 overlay 叠层 (也就是边框) 现在使用它的子元素大小，而这个子元素正是我们指定的固定的 200 x 200 尺寸的 frame 修饰器。这表明，尽管看起来似乎是一个理论问题，但除非你只想写一些很简短例子，否则修饰器的顺序是非常重要的。

> 在 view 的周围放上边框，以此来让 view 的 frame 可视化，是一种非常有用的调试方法。

在 SwiftUI 中，你永远不会强迫 view 直接使用一个特定的大小。你只能将其包装在 frame 修饰器中，它的可用空间将被提供给子元素。正如我们将在view 布局章节中看到的那样，view 可以定义自己的理想大小 (类似于 UIKit 的 sizeThatFits 方法)，你可以强制让 view 变成它们的理想大小。

如果想要实现父 view 的布局依赖子 view 的尺寸 (比如，如果你想要重新实现自己的 VStack)，那事情会复杂一些，你需要用到 GeometryReader 和 Preference，我们在本书后面将介绍这些内容。

View 更新

现在 view 已经创建好，并且完成了布局，SwiftUI 在屏幕上显示它们，并等待影响 view 树的状态更改。在我们的示例中，点击按钮会修改 @State counter 属性，这会触发这种更新 view 的状态更改。

需要触发 view 更新的属性会被用 @State、@ObservedObject 或者 @EnvironmentObject 属性标签进行标记 (我们会在下一章介绍其他标签)。到目前为止，只需要知道，对带有这些标签的属性进行修改将导致 view 树的重新计算就足够了。

当我们的示例中的 counter 属性被更改时，SwiftUI 将再次访问 content view 的 body 属性，并取得对应新状态的 view 树。注意，view 树的类型 (上面讨论的隐藏在 some View 后面的复杂类型) 不会改变。实际上，因为类型在编译时已经固定了，它也不能更改。因此，唯一可以改变的是 view 的属性 (例如显示点击次数的标签文本) 以及采用 if 语句的哪个分支。将 view 类型中的 view 树结构进行静态编码，具有重要的性能优势，我们将在下一章中详细讨论。

请务必牢记，更改状态属性是在 SwiftUI 中触发 view 更新的**唯一**方法。我们无法执行像是在事件处理闭包中修改 view 树等这样的 UIKit 中的常见操作。这种新的处理方式消除了 view 和 app 状态不同步这一整个类别的常见错误，但它要求我们以不同的方式思考：我们必须对 app 状态进行明确建模，并向 SwiftUI 描述每个给定状态所对应的屏幕内容。

重点

- → SwiftUI 的 view 是值，而非对象：它们是不可变的，用来暂时描述屏幕上应该显示什么。
- → 我们在一个 view 上几乎所有的方法调用 (像是 frame 或 background) 都会将 view 包装在一个修饰器中。因此，这些调用的顺序很重要，这一点和 UIView 里的属性不同。
- → 布局是自上而下的：父 view 向子 view 提供它们的可用空间，子 view 基于这个空间来决定自己的尺寸。
- → 我们不能直接更新屏幕上的内容。相反，我们必须修改状态属性 (比如 @State 或 @ObservedObject)，然后让 SwiftUI 去找出 view 树的变化方式。

View 更新

2

在第一章中，我们研究了如何在 SwiftUI 里构造 view 树，以及如何根据状态变化对其进行更新。在本章中，我们将详细介绍 view 更新的过程，并说明编写简洁高效的 view 更新代码所需的知识。

更新 View 树

在大多数面向对象的 GUI 程序 (例如 UIKit app 和浏览器中的 DOM (文档对象模型，Document Object Model) app) 中，有两条与 view 相关的代码路径：一条路径处理 view 的初始构造，另一条路径负责在事件发生时更新 view。由于这些代码路径是分离开的，而且涉及手动更新，所以很容易出现错误：我们可能会响应事件来更新 view，但却忘了更新 model，反之亦有可能。无论哪种情况，view 都会与 model 不同步，app 可能会表现出不确定的行为、卡死甚至崩溃。通过遵守规则和测试可能可以避免这些错误，但这并不容易。

在 AppKit 和 UIKit 编程中，有许多技术可以尝试解决此问题。AppKit 里使用 Cocoa Binding 技术，它是一个可以使 model 和 view 保持同步的双向层。在 UIKit 里，人们使用像是响应式编程这样的技术来让这两个代码路径 (在大部分情况下) 得到统一。

SwiftUI 的设计完全避免了此类问题。首先，只有 view 的 body 属性这一个代码路径可以构造初始的 view，而且这条路径也会用于所有的后续更新。其次，SwiftUI 让使用者无法绕过正常的 view 的更新周期，也无法直接修改 view 树。在 SwiftUI 中，想要更新屏幕上的内容，触发对 body 属性的重新求值是唯一的方法。

当框架需要渲染一个更新后的 view 树时，最简单的实现是丢弃掉所有内容，并从头开始重新绘制屏幕。但这是很低效的做法，因为重新创建那些底层的 view 对象 (比如 UITableView) 可能会是很昂贵的操作。更糟糕的是，重新创建这些 view 对象可能意味着 view 状态的丢失，比如像是滚动位置、或是当前选中的内容等，很难被轻易地保存并应用到新的 view 里去。

为了解决这个问题，SwiftUI 需要知道哪些底层 view 对象需要更改、添加或删除。换句话说：SwiftUI 需要将先前的 view 树的值 (body 求值的结果) 与当前 view 树的值 (状态改变后重新对 body 求值的结果) 进行比较。SwiftUI 使用了很多技巧来优化此过程，即使在大型的 view 树中进行的更改也可以高效地执行。

理想情况下，在使用 SwiftUI 时，我们原本并不需要去了解这个过程。但是，这些实现细节总是会以这样或者那样的形式出些问题。而且，对 SwiftUI 如何执行 view 更新有一个基本的了

解，可以帮助我们判断到底是不是由于我们的代码，导致了 SwiftUI 无法高效地完成工作。为了探索这个过程，我们将从上一章中使用的示例开始，将它稍微简化：

```
struct ContentView: View {
  @State var counter = 0
  var body: some View {
    VStack {
      Button("Tap Me") { self.counter += 1 }
      if counter > 0 {
        Text("You've tapped \(counter) times")
      }
    }
  }
}
```

当我们的 view 首次被渲染时，ContentView 的 body 属性被求值，它得到的是一个满足 View 协议 (body 的类型是 some View) 的值。不过，当我们查看它的实际类型时，我们可以看到不论树中的 view 当前是否可见，它们全都被编码为了相应的类型：

```
VStack<TupleView<(Button<Text>, Text?)>>
```

尽管 if 条件里的文本标签还没有显示在屏幕上 (因为 counter 还是零)，但它从一开始就以 Text? 的形式出现在 view 树的类型中。当 counter 属性更改时，body 会被再次执行，从而产生具有完全相同类型的新 view 树值。Text? 值将始终存在：如果标签不应该显示在屏幕上，它为 nil，否则它应该包含一个 Text view。SwiftUI 中 view 树里的元素类型在每次更新时都是相同的，并且由于 view 树的结构是由类型系统进行编码的，因此编译器可以对这种不变性作出保证。

为什么 view 树需要每次都拥有相同的结构呢？难道每次都把程序中所有可能的 view 树的整个结构都编码到这个值里不是一种浪费吗？

每次 app 状态发生变化并重新计算 view 树时，SwiftUI 必须找出前一棵树与新树之间发生了什么变化，以便有效地更新显示，这样才能避免从头开始重新构建和渲染所有内容。如果能保证旧树和新树具有相同的结构，那么这个任务将更容易，更高效。

树的 diff 算法 (也就是比较两棵树结构之间不同之处的算法) 的算法复杂度是 O(n^3)。也就是说，如果对于 10 个元素的树，我们需要进行 1,000 次操作来进行比较的话，对于 100 个元素的树，我们需要多达 1,000,000 (一百万) 次操作。为了能控制复杂度，像 React 这样的框架使用了具有 O(n) 复杂的的启发式 diff 算法，它在 diff 的精度和效率之间进行了权衡：这种算法可能会导致实际被重建的部分要比严格意义上真正需要重建的部分更大，开发者们可能会需要提供一些提示，来表明树的哪些部分在更新时是稳定的，从而应对这种影响。

SwiftUI 针对此问题采用了不同的方法：由于 view 树的结构在更新时始终相同，因此它不需要执行完整的树 diff。因为结构是相同的，SwiftUI 可以简单地同时遍历旧树和新树。例如，即使我们的counter 属性仍然是零，SwiftUI 还是能通过树中的 Text? 元素知道，在不同状态下，可能会存在一个文本标签。当我们将计数器从零增加到一时，它可以比较树中的项目，并且会注意到在以前是 nil 的地方现在有一个 Text view。当 counter 从 1 变为 2 时，两棵树中都会有 Text view，SwiftUI 可以比较两者的文本属性以确定是否必须重新渲染内容。

类似地，SwiftUI 知道顶层总会是一个 VStack，且其中是一个包含两个元素的 TupleView。它不需要担心在根级别出现完全不同的 view，而只需要比较新旧两个 VStack 上可能会被改变的属性值 (比如 stack 的对齐方式和间距等) 就行了。

现在，你可能会再次好奇：即便 view 树的比较要比执行完整树比较要快得多，但每次重新创建和比较整个 view 树值难道不是依然很浪费吗?

事实上，SwiftUI 在这一点上也很聪明。为了研究 view 的 body 会在什么时候被运行，我们可以使用上一章中的 debug 辅助方法，或者简单地插入一条 print 语句。在我们的示例中，每次 counter 变量改变时，都会执行 ContentView 的 body，因为 counter 是使用 @State 标签声明的 (我们会在下面具体探讨 @State 和其他触发 view 更新的标签)。如果我们将示例中的 content view 分为两个 view，将 counter 的值从 ContentView 传递到新的 LabelView，每次更新时整个 view 树依然会被重新计算：

```
struct LabelView: View {
  let number: Int
  var body: some View {
    print("LabelView")
    return Group {
      if number > 0 {
        Text("You've tapped \(number) times")
```

```
        }
      }
    }
}

struct ContentView: View {
  @State var counter = 0
  var body: some View {
    print("ContentView")
    return VStack {
      Button("Tap me!") { self.counter += 1 }
      LabelView(number: counter)
    }
  }
}

// 每次更新的控制台输出：
//  ContentView
//  LabelView
```

SwiftUI 会追踪哪些 view 使用了哪些状态变量：因为我们访问了 counter，并将其作为参数传递给了 LabelView (在下面的属性包装的部分中，我们将详细讨论依赖追踪的工作原理)，所以它知道 content view 的 body 在构造过程中使用了 @State 变量 counter。因此，当 counter 改变时，它将重新执行 content view 的 body。

如果我们将这个 view 按照别的方式分割，比如让 counter 只在 subview 里使用的话，情况会有所改变：

```
struct LabelView: View {
  @State var counter = 0
  var body: some View {
    print("LabelView")
    return VStack {
      Button("Tap me!") { self.counter += 1 }
      if counter > 0 {
        Text("You've tapped \(counter) times")
      }
    }
```

```
  }
}

struct ContentView: View {
  var body: some View {
    print("ContentView")
    return LabelView()
  }
}

// 开始时的控制台输出：
//  ContentView
//  LabelView

// 每次更新的控制台输出：
//  LabelView
```

SwiftUI 只会重新去执行那些**使用了** @State 属性的 view 的 body (对于其他属性包装，例如 @ObservedObject 和 @Environment，也是一样的)。因此，当 counter 变化时，只有 label view 的 body 被执行了。理论上说，这样做是会把整个 label view 的子树都失效重绘，不过 SwiftUI 对这个过程也做了优化：当它知道子树没有变更的情况下，SwiftUI 会避免去重新执行子树的 body。

我们可以在 label view 里定义一个 binding (绑定)，然后在 content view 里声明 @State 属性来达到类型的效果。当 counter 改变时，LabelView 的 body 会被重新执行，而 ContentView 的 body 并不会：

```
struct LabelView: View {
  @Binding var number: Int
  var body: some View {
    print("LabelView")
    return Group {
      if number > 0 {
        Text("You've tapped \(number) times")
      }
    }
  }
}
```

```
struct ContentView: View {
  @State var counter = 0
  var body: some View {
    print("ContentView")
    return VStack {
      Button("Tap me!") { self.counter += 1 }
      LabelView(number: $counter)
    }
  }
}

// 开始时的控制台输出：
//  ContentView
//  LabelView

// 每次更新的控制台输出：
//  LabelView
```

本质上来说，binding 是它所捕获变量的 setter 和 getter。SwiftUI 的属性包装 (比如 @State，@ObservedObject 等) 都有对应的 binding，你可以在属性名前加上 $ 前缀来访问它。(在属性包装的术语中，binding 被叫做一个**投射值** (projected value)。) 如上例所示，SwiftUI 会追踪哪些 view 使用了哪些 state 变量：它知道 ContentView 在渲染 body 时并没有用到 counter，但 LabelView (通过 binding 间接地) 用到了它。因此，对 counter 属性的更改只会触发对 LabelView body 的重新求值。

> SwiftUI 的绑定和 Cocoa 绑定是用于相似目的的不同技术。SwiftUI 和 Cocoa 两者都提供双向绑定，但是它们的实现有很大不同。

动态 view 树

如果 SwiftUI 在根 view 的类型中就将 view 树的结构进行了编码，那么我们要怎么才能构建那些非静态的 view 树呢？显然，我们需要一些手段来动态地替换 view 树的一部分，或者插入一些我们在编译期间不知道的 view。

SwiftUI 提供了三种不同的机制来构建一棵树的动态部分：

1. View builder 中的 if/else 条件
2. ForEach
3. AnyView

这些机制都有各自的特定属性和能力，我们来逐一介绍。

对于动态地改变运行时屏幕上的内容操作来说，view builder 中的 if/else 条件是限制最多的选择。if/else 的分支被完整地编码到 view 的类型中 (其类型是 _ConditionalContent)：在编译时就非常明确，屏幕上的 view 会在 if 分支或者 else 分支中则其一。换句话说，if/else 允许我们在运行时根据条件隐藏或是显示 view，但我们必须在编译时就决定好 view 的类型。同样，不包含 else 的 if 会被编码为仅在条件为 true 时显示的可选 view。

在 ForEach 中，view 的数量是可以改变的，但它们都需要拥有相同的类型。ForEach 最常见的是和 List (类似 UIKit 中的 table view) 一起使用。列表 List 中的项目数量通常是基于 model 数据的，这通常不能在编译期间知道：

```
struct ContentView: View {
  var body: some View {
    ForEach(1...3, id: \.self) { x in
      Text("Item \(x)")
    }
  }
}

// 类型： ForEach<ClosedRange<Int>, Int, Text>
```

ForEach 有三个参数，它们分别直接对应了类型中的三个泛型参数。我们会介绍最长的那个初始化方法，不过它也还有其他两个便捷初始化方法。

ForEach 的第一个参数是所要显示数据的集合。第一个参数的类型也是 ForEach 的第一个泛型参数的类型，在我们的例子中是 ClosedRange<Int>，但它可以是任意的 RandomAccessCollection。

第二个参数是键路径 (keypath)，它指定应该使用哪个属性来标识元素 (集合的元素要么必须遵守 Identifiable 协议，要么我们需要为它指定标识符的键路径)。我们通过指定 \.self 作为标识键路径，将元素本身用作标识符。因此，ForEach 的第二个泛型参数 (标识符的类型) 为 Int。

ForEach 的第三个参数负责从集合中的元素构造 view。这个 view 的类型就是 ForEach 的第三个泛型参数 (在我们的例子中，我们渲染了一个 Text)。

由于 ForEach 要求每个元素都是可标识的，因此它可以在运行时 (通过计算 diff) 找出自上次 view 更新以来所添加或删除的视图。尽管对于动态子树来说，工作量要比 if/else 条件求值更大一些，但它仍然使 SwiftUI 能够聪明地去更新屏幕上的内容。同样，元素具有唯一标识这一特性，也能在动画，甚至是某一个元素上的属性发生变更时提供帮助。

最后，AnyView 是一个可以用任意 view 来初始化的 view，它可以对输入的 view 进行包装并擦除它的类型。这意味着 AnyView 可以包含完整的任意 view 树，而无需在编译时将它们的类型静态固定。虽然这给了我们很多自由，但 AnyView 应该是我们迫不得已的最后手段。这是因为使用 AnyView 会删除有关 view 树的基本静态类型信息，而这些信息将可以帮助 SwiftUI 更高效地执行更新。我们将在下一节中更详细地介绍这一点。

高效的 View 树

SwiftUI 在每次更新时要高效地对 view 树的值进行比较，这需要依赖 view 树结构的静态信息。View builder 帮助我们构造这些树并捕获类型系统中的静态结构。但是，有时我们并不处在 view builder 中。比如下面这个示例，它无法编译：

```
struct LabelView: View {
  @Binding var counter: Int
  var body: some View {
    if counter > 0 {
      Text("You've tapped \(counter) times")
    }
  }
}
```

View 的 body 属性并不使用 view builder 的语法，这里的 if 条件是一个普通的 Swift if 条件语句。Swift 并不知道上面的 view 的类型是什么：如果条件是 true，那么它会渲染一个文本，但

在条件是 false 的时候应该渲染什么呢？我们真正想要的是，仅当条件为 true 时显示 Text。我们可以把这个 view 包装到 Group 中 (它接受一个 view builder 的尾随闭包)：

```
struct LabelView: View {
  @Binding var counter: Int
  var body: some View {
    Group {
      if counter > 0 {
        Text("You've tapped \(counter) times")
      }
    }
  }
}
```

现在，view builder 中的 if 条件被转换为了可选的 Text view，以满足编译器的要求。当我们具有不同类型的 if/else 条件时，也会发生类似的问题。在普通的 Swift 代码中，这是不允许的：

```
var body: some View {
  if counter > 0 {
    return Text("You've tapped \(counter) times")
  } else {
    return Image(systemName: "lightbulb")
  }
}
// error: Function declares an opaque return type, but the return statements
// in its body do not have matching underlying types
```

虽然 Text 和 Image 都遵守 View 协议，但是我们无法从 body 的不同分支返回具有不同具体类型的值。我们可以返回遵守 View 的任何内容，但是我们必须从所有分支都返回相同类型的内容。和之前一样，我们可以通过将整个 body 包装在一个 Group 中来解决此问题。由于 Group 的初始化方法参数是 view builder 闭包，因此 if/else 条件被编码为 view 树的一部分。结果类型为：

```
Group<_ConditionalContent<Text, Image>>
```

对于这个问题来说，使用 Group 的 view builder 来将按条件返回的不同类型进行封装，是一种不错的解决方案，因为它保留了 view 树所有有关可能的信息：SwiftUI 现在知道，这里要么是一个文本，要么是一张图片。

> 除了使用 Group 外，你也可以为 body 计算属性加上 @ViewBuilder 标签。不过，在本书写作时 (Xcode 11.3)，这种方法还并不稳定。

对于这个问题，下面这种方案就不太好：

```
var body: some View {
  if counter > 0 {
    return AnyView(Text("You've tapped \(counter) times"))
  } else {
    return AnyView(Image(systemName: "lightbulb"))
  }
}
```

通过将返回的 view 包装在 AnyView 中，我们满足了编译器从所有分支返回相同具体类型的要求。但是，我们也从类型系统中抹除了此位置可以显示哪种 view 的所有信息。SwiftUI 现在必须做更多的工作来确定当 counter 属性改变时如何更新屏幕上的 view。正如其名，AnyView 可以是任何一种 view，在 SwiftUI 决定什么东西发生了变更的时候，它必须去比较 AnyView 中实际的值，而不能像原本那样只是去检查 _ConditionalContent 值上的条件。

另一个可能的性能问题与我们使用状态属性触发 view 更新的方式和位置有关。在上一节中，我们看到 SwiftUI 知道在哪里使用了 @State 属性 (或其他状态属性包装器，我们将在下面进行详细讨论)：一旦我们分解了 LabelView 并将计数器作为绑定传递，在 counter 改变时，SwiftUI 就只会执行 LabelView 的 body，而不会去重新构建整个 view 树。一般而言：SwiftUI 会跟踪哪些 view 使用了哪些状态属性，并在 view 更新时，SwiftUI 仅去执行实际上可能被更改了的 view 的 body。

在构建 view 时，我们应该利用这一点，因为这关系到我们在哪里放置状态属性，以及如何使用它们。想要最好地利用 SwiftUI 的智能 view 树的更新特性，我们应该尽可能地将状态属性放在本地。相反，在根 view 上用一个状态属性表示所有的 model 状态，并以简单的参数形式将所有数据向下传递到 view 树中，会是最糟糕的选择，因为这将导致很多不必要的 view 被重新构建。

状态属性标签

能让 SwiftUI 更新 view 的唯一方法是更改信息源，也就是在我们的示例中那些用 @State 声明的状态属性。SwiftUI 用来触发 view 更新的所有属性包装器均遵守 DynamicProperty 协议。在文档中查看这个协议，会发现以下类型实现了它：

- → Binding
- → Environment
- → EnvironmentObject
- → FetchRequest
- → GestureState
- → ObservedObject
- → State

下面，我们将更详细地介绍最常见的包装器：@State，@Binding 和 @ObservedObject。我们将在环境 (Environment) 一章中讨论基于 @Environment 的包装器。

属性包装

SwiftUI 发布时，为了能写出简洁且易读的 SwiftUI 程序，Swift 中添加了两个新特性：函数构建器 (function builder) 和属性包装器 (property wrapper)。具体来说，SwiftUI 使用 view builder (在前一章讨论过)，以及像是 @State，@Environment 和 @Binding 等一组内置的属性包装器。

属性包装器本身是一个庞大的话题，我们在这里将展示它们是如何使 SwiftUI 更具可读性的。考虑本章前面使用 @State 属性包装器的示例：

```
struct ContentView: View {
  @State var counter = 0
  var body: some View {
    return VStack {
```

```
        Button("Tap me!") { self.counter += 1 }
        LabelView(number: $counter)
      }
    }
}
```

因为 body 不是一个 mutating 的函数或者属性，如果我们移除 @State 前缀，那么 body 中的 counter 就不再能变更了。为了更好地理解属性包装器做了什么，这次我们不使用属性包装器语法，来重写上面的示例：

```
struct ContentView: View {
  var counter = State(initialValue: 0)
  var body: some View {
    return VStack {
      Button("Tap me!") { self.counter.wrappedValue += 1 }
      LabelView(number: counter.projectedValue)
    }
  }
}
```

首先，我们现在必须用 State 的显式初始化方法创建我们的 counter，State 是定义在 SwiftUI 中的结构体，它被标记了 @propertyWrapper。其次，我们仍然无法在 body 内修改 counter 变量本身 (因为 body 是不可变的)。但是，State 实际上定义了一个被标记为 nonmutating set 的 wrappedValue 属性，这意味着我们可以在不可变方法或属性 (例如 body) 的内部修改它。最后，我们没有传递 $counter 到我们的 LabelView，而是传递了 counter.projectedValue，它的类型是 Binding<Int>。

当我们调用上面的代码片段时，可以发现它们是彼此等效的。不过，使用属性包装的例子视觉噪音较小：我们可以像使用 Int 一样直接对 counter 变量进行初始化和修改 (实际上，被修改的是 wrappedValue，但是我们不必把它写出来)。同样地，我们可以使用 $counter 来访问它的绑定，而不需要自己去写。

State 类型还可以启用依赖追踪。当 view 的 body 访问 State 变量的 wrappedValue 时，这个 view 会与该 State 变量建立依赖。这意味着 SwiftUI 知道 wrappedValue 更改时要去更新哪些 view。依赖追踪并不是属性包装本身的一部分，但是属性包装的简化语法可以把这些内容隐藏起来，使其对程序员不可见。

在本节的例子中我们使用了 State，不过 SwiftUI 中的其他属性包装也都具有相同的语法优势，它们也都提供依赖追踪的功能。

有关属性包装的更多信息，这个 Swift 进化提案 涉及了该主题，并包含完整规范。另外，WWDC 2019 Session 415 展示了如何在设计你自己的 API 时使用属性包装。

State 和 Binding

在所有属性包装中，@State 是在 SwiftUI 中进行尝试时使用起来最简单的一个：我们只需在属性前面写上 @State，然后照常使用即可。每次更改属性时，都会触发 view 的更新。@State 非常适合用来表示局部 view 的状态，例如那种我们必须在一个小组件中去跟踪的状态。

作为示例，我们来构建一个简单的圆形旋钮 (Knob 类型)，它类似于音频程序中使用的那种旋钮。首先，这是一个简单的旋钮组件，其中包含当前值的常规属性 (我们省略了 KnobShape 的定义，你可以在随书代码中找到它)。当值为 0 时，旋钮显示为一个带有指向顶部的小指针的圆。当值接近 1 时，我们将旋钮旋转整整一圈：

```
struct Knob: View {
  // 0 到 1 之间的值
  var value: Double
  var body: some View {
    KnobShape()
      .fill(Color.gray)
      .rotationEffect(Angle(degrees: value * 330))
  }
}
```

在另一个 view 中，我们可以使用这个旋钮。比如说，我们可以设置一些由滑块控制的局部状态。当滑块改变时，旋钮会被自动重新渲染：

```
struct ContentView: View {
  @State var volume: Double = 0.5
  var body: some View {
    VStack {
      Knob(value: volume)
        .frame(width: 100, height: 100)
      Slider(value: $volume, in: (0...1))
```

```
    }
  }
}
```

与 (直接使用 volume 的) 旋钮不同，滑块是通过 $volume 来配置的，它的类型是 Binding<Double>。当滑块第一次绘制自身时，它需要一个 (由绑定提供的) 初始值。用户与滑块互动后，还需要写回更改后的值。滑块本身并不真正在乎 Double 的来源、存储的位置或是代表的含义：它只需要一种读写 Double 的方式，而这正是 Binding<Double> 所提供的。

我们来向旋钮添加一个简单的交互。当轻触旋钮时，我们想在 0 和 1 之间进行值的切换。这意味着我们需要一种将新值传递回 content view 的方法。虽然我们可以在此处使用绑定，但我们将首先尝试使用简单的回调函数重新创建绑定的功能：

```
struct Knob: View {
  var value: Double // 0 到 1 之间的值
  var valueChanged: (Double) -> ()

  var body: some View {
    KnobShape()
      .fill(Color.gray)
      .rotationEffect(Angle(degrees: value * 330))
      .onTapGesture {
        self.valueChanged(self.value < 0.5 ? 1 : 0)
      }
  }
}
```

当我们配置这个旋钮时，现在我们还需要传入一个回调函数，在其中将 self.volume 设置为新的值。这个状态变更会触发再次渲染，滑块和旋钮都将展示这个新值：

```
Knob(value: volume, valueChanged: { newValue in
  self.volume = newValue
})
```

虽然这种做法是可行的 (而且可以很好地帮助我们理解绑定的背后到底发生了什么)，但直接使用绑定显然会更加优雅：

```
struct Knob: View {
  @Binding var value: Double // 0 到 1 之间的值

  var body: some View {
    KnobShape()
      .fill(Color.gray)
      .rotationEffect(Angle(degrees: value * 330))
      .onTapGesture {
        self.value = self.value < 0.5 ? 1 : 0
      }
  }
}
```

通过绑定，创建旋钮变得更简单了：

```
Knob(value: $volume)
```

在编写自定义控件时，我们通常先使用 @State 变量，以便更快地进行原型开发。一旦我们需要在控件外部存储并观察该状态，我们只需要将 @State 改为 @Binding 就可以了，这个操作不需要对代码进行任何其他改动。

注意，我们不仅仅可以创建整个状态值的绑定，也可以创建到一个状态值上的某个属性的绑定。如果我们的状态结构体中包含有多个属性，我们可以通过 Knob(value: $state.volume) 来初始化旋钮。如果我们有一个 volume 的数组，我们也可以写这样的代码 Knob(value: $volumes[0])。

和 @State 类似，@GestureState 也是一个属性包装器，它被专门用来进行手势识别。@GestureState 属性在初始化时接受一个初始值，当手势发生时，这个值会被更新。一旦手势结束，手势状态中的属性值将被自动重置为初始值。

ObservedObject

在几乎所有的实际 app 中，我们会使用到我们自己的模型对象。为了让我们的模型对象可以被 SwiftUI 观察到，它的类型需要遵守被定义在 Combine 框架中的 ObservableObject 协议

举例来说，考虑构建一个简单的时钟 app。我们需要找到一种方法来告诉 SwiftUI 每秒对我们的 view 树重新渲染。为此，我们可以创建一个 CurrentTime 类，它会创建和开始一个计时器。我们也将暴露出一个 now 属性，它包含有当前的日期和时间：

```
final class CurrentTime: ObservableObject {
  @Published var now: Date = Date()
  let interval: TimeInterval = 1
  private var timer: Timer? = nil

  // ...
}
```

ObservableObject 协议的唯一要求是实现 objectWillChange，它是一个 publisher，会在对象变更时发送事件。通过在 now 属性前面添加 @Published，框架会为我们创建一个 objectWillChange 的实现，并在每次 now 改变的时候发送事件。

要开始计时器，我们可以向 CurrentTime 中添加一个 start 方法 (为了简明，我们没有写出 stop 方法和 deinit)：

```
// ...
func start() {
  guard timer == nil else { return }
  now = Date()
  timer = Timer.scheduledTimer(withTimeInterval: interval, repeats: true) {
    [weak self] _ in
    self?.now = Date()
  }
}
// ...
```

要使用这个新类，我们需要创建一个属性，并将它标记为 @ObservedObject：

```
struct TimerView: View {
  @ObservedObject var date = CurrentTime()

  var body: some View {
    Text("\(date.now)")
  }
```

```
}
```

现在，我们还需要一种调用 start 的方法。在 CurrentTime 的初始化方法里简单地调用 start 看起来很诱人，但一般来说，在一个 ObservableObject 被观察之前，就去开始昂贵或长期运行的工作，是一个糟糕的选择。在像这样的简单 app 中，它可以正常工作：当 view 树被创建时，我们的 CurrentTime 也会被创建，并且它会自动开始更新。不过，如果考虑下面这样的例子：

```
struct ContentView: View {
  var body: some View {
    NavigationView {
      NavigationLink(destination: TimerView(), label: {
        Text("Go to timer")
      })
    }
  }
}
```

上面的 app 显示了一个 root view，它上面有一个显示 “Go to timer” 的按钮，当用户点击按钮时，它会导航到 TimerView。然而，在初始化构建的时候，包括 TimerView 的整个 view 树就已经被构建了。这意味着我们的 CurrentTimer 值也被构建了，如果我们在初始化方法中调用了 start 的话，即使我们完全没有显示 view，计时器也会开始更新。

最简单的解决方法是在 TimerView 出现的时候再调用 start：

```
struct TimerView: View {
  @ObservedObject var date = CurrentTime()

  var body: some View {
    Text("\(date.now)")
      .onAppear { self.date.start() }
      .onDisappear { self.date.stop() }
  }
}
```

在创建其他可观察对象时，请记住它们会在视图构造时被创建。例如，当 app 中的某个界面必须从网络加载大量图像时，我们通常希望在需要图像时才去开始这些请求，而非在 view 树构建时就开始请求。

> 如果我们忘记了添加 @ObservedObject 前缀，我们的大部分代码仍将继续工作：我们可以访问对象的属性，也能调用它的方法。但是，SwiftUI 将不会去订阅 objectWillChange 的通知，所以 view 也不会随着状态改变而重新渲染。

其他的动态 View 属性

到目前为止，我们研究了 @State、@GestureState、@Binding 和 @ObservedObject。其实还有更多类似属性。首先，@Environment 和 @EnvironmentObject 也很常见，这些属性包装用于观察环境中的值或对象。在下一章中我们会详细谈到 SwiftUI 中的环境这一概念，并在那里详细讨论这两个属性包装。

还有一个属性包装是专门用来观察 Core Data 中 fetch request 的结果的，它是 @FetchRequest。我们可以通过一个 NSFetchRequest 来初始化它，然后在数据发生变化时，它将会自动去更新 view。你可以把它想象成 SwiftUI 版本的 NSFetchedResultsController。

重点

- → 在 SwiftUI 中，我们不能直接去操作 view 树。相反，我们通过改变状态，来触发 view 树的重新求值。
- → 最常用的状态属性包装是 @State、@Binding 以及 @ObservedObject。它们被用来响应状态变更，以触发 view 的更新。
- → 我们需要在 view 的 body 中去访问状态属性，并以此指定和当前状态相匹配的 view 树。
- → View 树中的很多部分，甚至是那些并不在当前屏幕上显示的部分，是会提前被创建的。因此，我们必须避免在 view 创建时就去做那些昂贵或者不必要的工作。

练习

为了练习本章中的概念，你需要构建一个简单的 app，用来显示照片的元数据列表。当你点击列表里的条目时，app 会显示对应的照片。这个练习中涉及到的挑战有：

→ 你必须从网络加载数据，并通过 SwiftUI 的方式来让数据被加载时 view 树可以得到更新。

→ 对于 (列表 view 和照片 view) 两个画面，你需要实现两个状态：一个用于数据正在加载时，一个用于数据被展示时。

→ 由于整个 view 树都是被提前构建的，因此你需要在 view 显示在屏幕上时再开始加载数据，而不是在它被构建时立即加载。

你可以在书末的答案附录中找到解答，你也可以从 GitHub 下载到完整的项目代码。

步骤 1：加载元数据

你可以从 https://picsum.photos/v2/list 加载这些数据。所以第一步是创建一个新的 Photo 结构体，来表示这个 API 的响应。你需要的信息包括 ID、作者、宽度、高度、URL 以及下载链接。这个结构体需要遵守 Codable 协议，这样在稍后就可以使用 JSONDecoder 了。下面是响应 JSON 的一个示例：

```
[
  {
    "id": "0",
    "author": "Alejandro Escamilla",
    "width": 5616,
    "height": 3744,
    "url": "https://unsplash.com/photos/yC-Yzbqy7PY",
    "download_url": "https://picsum.photos/id/0/5616/3744"
  },
  ...
]
```

步骤 2：创建 ObservableObject

创建一个新的 Remote class，让它遵守 ObservableObject 协议。这个类型存储某个 URL，并拥有一个可以将从网络获取到的数据转变为期望类型的函数。比如说，这个期望类型可以是一个 Photo 的数组 (这意味着 Remote 需要支持泛型)。Remote 类还需要一个用来存储已加载数据的属性，它的初始值为 nil。最后，添加一个 load 方法来通过 URLSession 从网络加载数据。

步骤 3：显示列表

创建一个 view，用来将作者名字显示在一个 List 中。对于每个 Photo 值，将 author 显示为列表中条目的文本。你还需要让 Photo 结构体满足 Identifiable，这样才能在 ForEach 中使用它们。使用 Remote 对象从网络加载照片数组，然后在数据加载时显示占位文本。

步骤 4：显示图片

将列表 view 包装到 NavigationView 中，然后将作者名字包装到 NavigationLink 里。创建一个新的 PhotoView，使用照片的 download_url 来进行初始化，它将作为 navigation link 的目标 view。在 PhotoView 中显示照片 (或者当照片正在加载时显示占位符)。通过将 resizable() 和 aspectRatio 组合起来，你可以让图片填满屏幕 (如果你需要一些关于这个的说明，在view 布局的章节中有对 resizable 和 aspectRatio 工作方式的解释)。

一个常见错误是在 view 被构建时就开始进行图片加载。请确保只有当 view 在屏幕上时再进行加载！

> 请注意，从网络异步加载内容时，SwiftUI 预览无法正常工作。如果要使用预览，最简单的选择是暂时插入一些静态内容去替代动态加载的内容。例如，您可以将一个特定高宽比的 Rectangle 放置到 PhotoView 中，来代替图像的 view。
>
> 或者，你可以通过实现一种更优雅的方法，来把资源加载的部分抽象为一个协议，并通过注入的方式控制 view 是要从网络加载还是从静态文件加载，来让它们在预览中更加流畅地工作。

附加练习

你可以将这个练习通过多种方式进行扩展。比如：

- → 对 UIActivityIndicatorView 进行包装，把静止的占位符替换成原生的 activity indicator。
- → 确保 PhotoView 中的图片占位和图片本身的高宽比一致。

→ 在列表中显示图片的缩略图。在列表中的 cell 被重新创建时，避免多次加载相同的缩略图图片。

→ 将触发资源加载的逻辑从 onAppear 中移动到 ObservableObject 里。你需要找到一种方法，来在 SwiftUI 订阅 objectWillChange publisher 时获取通知。

→ 在 Remote 包含错误而非已加载数据的时候，显示合适的错误。

环境

3

环境 (environment) 是帮助我们理解 SwiftUI 工作方式的一块重要拼图。简而言之，环境是 SwiftUI 用于将值沿 view 树向下传递的机制。也就是说，值从父 view 传递到其包含的子 view 树，是依靠环境完成的。SwiftUI 中广泛使用了环境，不过我们也可以将其用于我们自己的目的。

在本章的第一部分中，我们将探索环境的工作方式，以及 SwiftUI 是如何使用它的。在第二部分中，我们将从环境中进行读取，来自定义 view 的绘制。我们还将使用环境来存储自定义的值，SwiftUI 也使用同样的方式来通过环境定义预置的 view。最后，我们将研究环境对象，这些对象可以让我们通过一种基于环境所构建的特殊机制来进行依赖注入。

环境是如何工作的

在 SwiftUI 中，view 上有很多看起来和当前 view 并不相关的方法：比如说 font、foregroundColor 和 lineSpacing 这些方法在所有的 view 类型上都可用。打个比方，我们可以为 VStack 设置字体，或者为 Color 设置行间距，但它们有什么意义呢？

为了探索这里发生的事情，我们再来看看所构建的 view 的类型。让我们从一个包含文本标签的简单 VStack 开始：

```
var body: some View {
  VStack {
    Text("Hello World!")
  }.debug()
}

// VStack<Text>
```

我们使用了第一章中的 debug 辅助函数来将 view 的具体类型打印出来。在这里，它是 VStack<Text>。现在，让我们在 stack 上调用 font 方法，来看看类型的变化：

```
var body: some View {
  VStack {
    Text("Hello World!")
  }
  .font(Font.headline)
  .debug()
}
```

```
/*
ModifiedContent<
  VStack<Text>,
  _EnvironmentKeyWritingModifier<Optional<Font>>
>
*/
```

这个类型告诉了我们，.font 调用将会把 VStack 包装到另一个叫做 ModifiedContent 的 view 中。这个 view 包含有两个泛型参数：第一个参数是内容本身的类型，第二个是将被应用到这个内容上的修饰器。在本例中，第二个参数是私有的 _EnvironmentKeyWritingModifier，正如其名，它负责将一个值写入到环境中。对于 .font 调用来说，一个可选的 Font 值会被写入到环境。因为环境会依据 view 树向下传递，所以 stack 中的文本标签可以从环境中读取这个字体。

虽然为一个 VStack 设置字体没有直接的意义，但是这个字体设定并不会丢失；它会通过环境被保留下来，树上的所有对此感兴趣的子 view 都可以使用它。为了验证字体值对于这个文本标签是否真的有效，我们可以打印出我们所关心的环境值。(通常，我们会使用 @Environment 属性包装来从环境中读取一个特定的值，但是出于调试目的的话，我们可以使用 transformEnvironment)：

```
var body: some View {
  VStack {
    Text("Hello, world!")
      .transformEnvironment(\.font) { dump($0) }
  }
  .font(Font.headline)
}

// ...
// - style: SwiftUI.Font.TextStyle.headline
```

我们可以在 EnvironmentValues 类型上查找所有公开可用的环境属性。然而，除了这些公开属性之外，SwiftUI 还在环境中存储了更多的属性。要查看这里所有东西，我们可以使用 transformEnvironment 修饰器并将 \.self 作为键路径传递进去。

因为 font 是 EnvironmentValues 上的公开属性，我们也可以直接用它来在环境上设置字体，而不去调用 font 方法：

```
var body: some View {
  VStack {
    Text("Hello World!")
  }
  .environment(\.font, Font.headline)
  .debug()
}

/*
ModifiedContent<
  VStack<Text>,
  _EnvironmentKeyWritingModifier<Optional<Font>>
>
*/
```

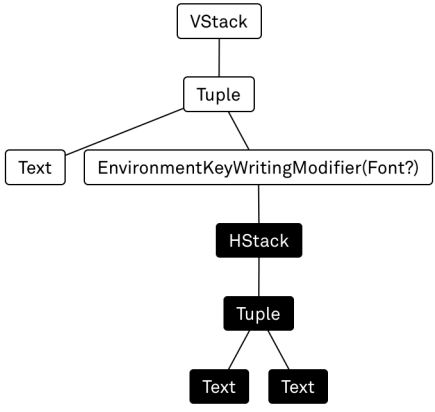
调用 .environment(\.font, ...) 所产生结果的类型和 .font(...) 调用完全一致。我们没有理由去选择用 .environment 来设置字体，但是这可以表明我们在像是 VStack 上调用的 font 方法，其实只是 .environment 函数的一个简单包装而已。

> 当我们直接在 Text 上调用 .font 时，返回类型依然是 Text。换句话说，字体被存储在了 Text 中，而非写入到环境里。然而，在 VStack 上调用的 .font 是一个完全不同的 (重载) 方法。这种模式在 SwiftUI 中随处可见。

当我们想要一次性更改多个 view 时，环境就变得非常有用了。举例来说，在下面的例子中，第一个标签被渲染为默认字体，而后两个嵌套的标签则使用 .largeTitle 字体进行渲染：

```
VStack {
  Text("Text 1")
  HStack {
    Text("Text 2")
    Text("Text 3")
  }.font(.largeTitle)
}
```

环境修饰器只会改变它的直属子 view 树的环境，而绝不会更改同层其他节点或是父 view 的环境。我们也可以通过上面代码的结构的可视化图看到这一点：只有黑色的节点会接收到带有 .largeTitle 字体的被更改后的环境：



使用环境

在本节中，我们会首先使用一个 SwiftUI 提供的已经存在的环境值来调整自定义 view 的外观，然后我们会介绍如何像配置 SwiftUI 的内置 view 那样，使用自定义的环境值来配置自定义 view。

使用既有的环境值

举个例子，我们回顾一下上一章里的旋钮控件：

```
struct Knob: View {
  @Binding var value: Double // 0 到 1 之间的数

  var body: some View {
    KnobShape()
      .fill(Color.gray)
      .rotationEffect(Angle(degrees: value * 330))
      .onTapGesture { ... }
  }
```

```
}
```

为了简明，旋钮实际形状的代码没有被包含在这里，不过你可以参考示例代码。

为了让这个控件能同时适配浅色和深色模式，我们会使用 SwiftUI 的 colorScheme 环境值来从环境里读取当前的配色方案，并对应地设置旋钮颜色：

```
struct Knob: View {
  @Binding var value: Double // 0 到 1 之间的数
  @Environment(\.colorScheme) var colorScheme

  var body: some View {
    KnobShape()
      .fill(colorScheme == .dark ? Color.white : Color.black)
      .rotationEffect(Angle(degrees: value * 330))
      .onTapGesture { ... }
  }
}
```

属性包装 @Environment 接受一个我们想要从环境中读取的值的键路径，我们稍后就可以像一个普通属性一样使用它。当填充旋钮形状时，我们读取这个 colorScheme 属性，并基于它选择不同的颜色。从一个 asset catalog 中使用颜色定义可以做到同样的事情，不过通过使用环境值，我们可以做的事情要比调整颜色多得多；我们也可以调整绘制旋钮的方式，比如，为了能和颜色更加匹配，我们可能会想要调整形状线条的宽度等。

当环境中的配色方案发生改变时，旋钮 view 会自动被重绘。从这一点来说，@Environment 属性和 @State 属性表现得一致：当值改变时，view 的更新会被触发。不过要注意，环境并不是一个全局的值的字典，一棵 view 的子树所拥有的环境，可以和另一棵子树不同。该环境中的值严格地在 view 树中向下传播 (从父级到子树)。

自定义环境值

我们实际用来绘制旋钮时所使用的 KnobShape 有一个参数，可以用来配置旋钮指针的大小。我们想要将这个自定义选项暴露给外面，同时我们也想找到一种方法，来让指针尺寸能够在旋钮所在 view 树的上游的任意位置都能被设定 (就像在上面我们在 VStack 上设置字体那样，字体设置最后被 stack 中的文本标签所获取)。

当然，将所有自定义 view 的属性像这样公开出来是没有意义的。更多时候，我们通过 view 的初始化方法来公开它们。不过，假设如果我们正在写一个音频类的 app 的话，可能在更高的级别上控制旋钮样式的能力会有一些意义。

让我们从最简单的方式开始，把配置项通过初始化方法公开：

```
struct Knob: View {
  @Binding var value: Double // 0 到 1 之间的数
  var pointerSize: CGFloat = 0.1
  @Environment(\.colorScheme) var colorScheme

  var body: some View {
    KnobShape(pointerSize: pointerSize)
      .fill(colorScheme == .dark ? Color.white : Color.black)
      .rotationEffect(Angle(degrees: value * 330))
      .onTapGesture { ... }
  }
}
```

通过为指针尺寸设置一个默认值，我们不需要在构建时去提供值。理想情况下，我们之后也还应该能在 view 层级中设置指针大小，比如像这样：

```
Knob(value: $value)
  .knobPointerSize(0.2)
```

不过，下面这样也应该能工作：

```
Knob(value: $value)
  .frame(width: 100, height: 100)
  .knobPointerSize(0.2)
```

通过在旋钮上调用 .frame，我们所处理的 view 的类型不再是 Knob。frame 修饰器会将旋钮包装到一个 ModifiedContent struct 里，也就是说，我们需要将 knobPointerSize 添加到任意满足 View 的类型中去，而不仅仅只是加到 Knob 上：

```
extension View {
  func knobPointerSize(_ size: CGFloat) -> some View {
    // ...
```

```
  }
}
```

环境允许我们将指针的尺寸值从 knobPointerSize 被调用的地方开始，向下传递到实际的旋钮 view 去。knobPointerSize 的实现应该只去调用 self 的 .environment，并指定某个键路径和尺寸值。要实现这一点，我们首先需要定义一个新的类型，让它遵守 EnvironmentKey 协议：

```
fileprivate struct PointerSizeKey: EnvironmentKey {
  static let defaultValue: CGFloat = 0.1
}
```

EnvironmentKey 协议的唯一要求是一个静态的 defaultValue 属性。当我们尝试从一个环境中读取指针尺寸值，但这个环境并不包含该值时，该默认值将会被返回。(在这里编译器通过对 defaultValue 属性的声明进行推断，隐式地将协议的关联值类型 Value 指定成了 CGFloat)

因为 .environment API 通过从 EnvironmentValues 的键路径来获取对应类型的值，所我们还要为 EnvironmentValues 添加一个属性，这样我们才能将它用作键路径：

```
extension EnvironmentValues {
  var knobPointerSize: CGFloat {
    get { self[PointerSizeKey.self] }
    set { self[PointerSizeKey.self] = newValue }
  }
}
```

属性的 getter 使用了 EnvironmentValues 上已经存在的下标方法来通过自定义的 PointerSizeKey 获取值，setter 通过同样的下标来设置新的值。有了这个属性，我们就可以回到 knobPointerSize 的实现了。注意，environment 是 self 上的一个方法，它返回的是带有更改后的环境的 self：

```
extension View {
  func knobPointerSize(_ size: CGFloat) -> some View {
    environment(\.knobPointerSize, size)
  }
}
```

唯一剩下要做的是在旋钮 view 本身中，实际去从环境里使用这个值。我们将 pointerSize 属性改为可选值，并使用 @Environment 属性包装去从环境中读取这个值 (如果我们没有在 view 树上的某个地方调用 knobPointerSize 的话，会读取到默认值)。接下来，我们就可以选择是直接使用初始化方法提供的 pointerSize，或是选择传入 nil，让旋钮去从环境中去读取指针尺寸：

```
struct Knob: View {
  @Binding var value: Double // 0 到 1 之间的数
  var pointerSize: CGFloat? = nil
  @Environment(\.knobPointerSize) var envPointerSize
  @Environment(\.colorScheme) var colorScheme

  var body: some View {
    KnobShape(pointerSize: pointerSize ?? envPointerSize)
      .fill(colorScheme == .dark ? Color.white : Color.black)
      .rotationEffect(Angle(degrees: value * 330))
      .onTapGesture { ... }
  }
}
```

现在，我们可以像这样自定义旋钮的外观了：

```
Knob(value: $value)
  .frame(width: 100, height: 100)
  .knobPointerSize(0.2)
```

或者甚至这样一次性定义多个旋钮：

```
HStack {
  VStack {
    Text("Volume")
    Knob(value: $volume)
      .frame(width: 100, height: 100)
  }
  VStack {
    Text("Balance")
    Knob(value: $balance)
      .frame(width: 100, height: 100)
  }
```

```
}
.knobPointerSize(0.2)
```

环境是用于将数据向下传递到 view 树的强大工具。但是，这也会使读取配置值和设定它的地方脱钩。结果就是，我们会非常容易忘记设定某个环境值，或者是将它该设置在了错误的 view 子树上。我们建议始终先将自定义选项作为简单的 view 参数进行公开，然后在当注意到耦合更少的 API 会比较有用时，再改变为使用环境值来进行实现，这种改变也会很容易。

依赖注入

我们可以把环境看作是一种依赖注入；设置环境值等同于注入依赖，而读取环境值则等同于接收依赖。

不过，环境中通常使用的都是值类型：一个通过 @Environment 属性依赖某个环境值的 view，只会在一个新的环境值被设置到相应的 key 时才会失效并重绘。如果我们在环境中存储的是一个对象，并通过 @Environment 观察它，view 并不会由于对象中的一个属性变化而重绘，重绘只在将 key 设置为整个不同的对象时才会发生。然而，当我们在使用对象作为依赖时，完整的对象替换往往不是我们期望的行为。

SwiftUI 的环境系统为对象的注入提供了特殊的支持，那就是使用 environmentObject(_:) 修饰器。这个方法接受一个 ObservableObject，并将它沿着 view 树传递下去。它不需要指定 EnvironmentKey，因为这个对象的类型会自动被用作 key。要在 view 中观察一个环境对象，我们使用 @EnvironmentObject 属性包装。这会使 view 在被观察对象的 objectWillChange publisher 被触发时失效并重绘。

举例来说，我们可以使用这种方式来将一个数据库连接传递到代码各处。作为开始，我们创建一个从环境中使用 DatabaseConnection 的 view：

```
struct MyView: View {
  @EnvironmentObject var connection: DatabaseConnection
  var body: some View {
    VStack {
      if connection.isConnected {
        Text("Connected")
      }
```

```
    }
  }
}
```

为了注入这个数据库连接，我们需要在 MyView 的某个父级 view 上为环境提供一个 DatabaseConnection 的实例。如果我们忘了这件事，我们的代码将会崩溃。比如，我们可以像这样提供它：

```
struct ContentView: View {
  var body: some View {
    NavigationView {
      MyView()
    }.environmentObject(DatabaseConnection())
  }
}
```

如果我们将 environmentObject 调用从 content view 里拿出来，我们现在就能很容易地用一个以测试为目的的子类替换掉 DatabaseConnection 的实现了。

虽然 @EnvironmentObject 模式很有用，我们还是建议在可能的时候用带有 key 的 @Environment 并传递值类型，因为这相对来说更安全：EnvironmentKey 要求我们提供默认值。当使用 @EnvironmentObject 的时候，我们非常容易忘记注入对象从而引起严重崩溃。

Preferences

环境允许我们将值从一个父 view 隐式地传递给它的子 view，而 preference 系统则允许我们将值隐式地从子 view 传递给它们的父 view。

作为 SwiftUI 中对 preference 使用的例子，我们来看看 SwiftUI 的 NavigationView。它很像 UIKit 中的 UINavigationViewController，我们可以通过修饰器定义单个 view 的 navigationBarTitle、navigationBarItems 和其他一些属性。比如说，下面的代码创建了一个 navigation view，它具有单个 root view 和一个导航标题：

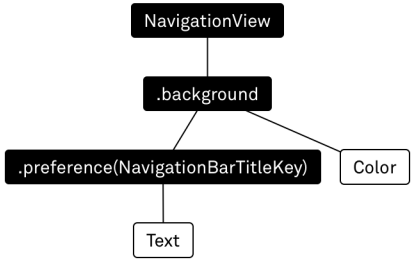
```
NavigationView {
  Text("Hello")
    .navigationBarTitle("Root View")
```

}

在本章前面的部分，我们看到过像是 .font 和 .foregroundColor 这样的修饰器，它们会改变各自的 view 子树的环境。不过，.navigationBarTitle 要做的事情恰好相反：Text 并不关心标题，不过它的父 view 对此关心，然而，NavigationView 有可能不是它的直接父 view。比如说，我们可能写这样的代码：

```
NavigationView {
  Text("Hello")
    .navigationBarTitle("Root View")
    .background(Color.gray)
}
```

当我们查看这棵 view 树 (被简化过) 的类型图时，我们可以看到，background 是 navigation view 的子 view，而 preference 又是 background 修饰器的子 view。.preference 的每一个祖先 (在图中的黑色节点) 都可以读取 .preference：



navigationBarTitle 定义了 preference，这个值被沿着树向上传递，通过 .background 修饰器，一直到达 navigation view，并最终被读取。

和环境一样，一个 preference 也由一个 (由类型表示的) key，一个对应值的关联类型 (在 navigation title 的例子中，这是一个包含 Text view 的私有类型) 和一个默认值 (本例中，也许是 nil) 组成。和 EnvironmentKey 不同，PreferenceKey 还需要一种合并多个值的方式，以对应多个 view 子树中都定义了同一个 preference 的情况。

作为示例，我们会重新创建 NavigationView 的一小部分。第一步，我们创建一个新的 PreferenceKey。对于关联类型，我们选择 String?。我们还需要一个 reduce 方法，来对应多

个子树都定义了 navigation title 的情况；在我们的实现中，我们简单地选择第一个遇到的非 nil 值：

```
struct MyNavigationTitleKey: PreferenceKey {
  static var defaultValue: String? = nil
  static func reduce(value: inout String?, nextValue: () -> String?) {
    value = value ?? nextValue()
  }
}
```

第二步，我们需要一种方式来在任意 view 上定义 navigation title。我们可以通过在 View 上的一个方法来设置 preference：

```
extension View {
  func myNavigationTitle(_ title: String) -> some View {
    preference(key: MyNavigationTitleKey.self, value: title)
  }
}
```

最后，我们需要在我们的 MyNavigationView 中读取 preference。要使用这个值，我们需要将它存储在 @State 变量中：

```
struct MyNavigationView<Content>: View where Content: View {
  let content: Content
  @State private var title: String? = nil
  var body: some View {
    VStack {
      Text(title ?? "")
        .font(Font.largeTitle)
      content.onPreferenceChange(MyNavigationTitleKey.self) { title in
        self.title = title
      }
    }
  }
}
```

当 view 首次被渲染时，title 是 nil。当子 view (content) 被渲染时，它将对应 MyNavigationTitleKey 的值沿树向上传递，onPreferenceChange 闭包会被调用。这会更改 title 属性，然后，因为 title 是一个 @State 属性，MyNavigationView 的 body 将被再次执行。

我们可以将这个新的 MyNavigationView 和 myNavigationTitle 修饰器组合起来使用，让它看起来 (除了 view builder 语法以外) 就像是我们正在使用的是一个普通的 navigation view 那样。

```
MyNavigationView(content:
  Text("Hello")
    .myNavigationTitle("Root View")
    .background(Color.gray)
)
```

我们在上面提到，多个 view 子树可以设置同一个 preference key 的值。在 navigation title 的例子中我们挑选了 preference key 里对应的第一个非 nil 值，我们也可以从所有的子树中收集 preference 值。举例来说，相比于在实现 navigation view 时，这个特性在实现 tab view 的时候就很有用：前者只需要显示导航栈最上层的标题，而 tab view 需要显示它所有子 view 的标题。Tab 项目标题的 preference key 的可能实现如下：

```
struct TabItemKey: PreferenceKey {
  static let defaultValue: [String] = []
  static func reduce(value: inout [String], nextValue: () -> [String]) {
    value.append(contentsOf: nextValue())
  }
}
```

这个 preference key 使用了 [String] 作为关联类型，这样一来，我们就可以在 reduce 方法中收集所有的 tab 标题了。Tab view 本身可以像上面的 navigation view 例子中那样读取和使用这个 preference：

```
struct MyTabView: View {
  @State var titles: [String] = []
  var body: some View {
    VStack {
      // ...
      HStack {
        ForEach(Array(titles.enumerated()), id: \.offset) { item in
```

```
            Text(item.element)
          }
        }
      }.onPreferenceChange(TabItemKey.self) { self.titles = $0 }
    }
}
```

在我们至今为止的 SwiftUI 经验中，我们几乎只会在处理布局系统中使用 preferences (我们会在后面的自定义布局一章中进行展示)。不过，在一些其他情况 (比如 navigation view) 下，这项技术也相当有用。

重点

- → 环境在 SwiftUI 中无处不在，它让我们可以写出简洁的代码。比如，对一个 VStack 设置字体，所有的子 view 都会继承这个字体。
- → @EnvironmentObject 是一种内建的实现依赖注入的方式 (即，通过对象而非简单的值，来使用环境)。但是我们需要小心，不要忘了设定这些依赖。
- → 和环境相对的是 preference 系统：它用来隐式地将值沿着树向上传递。

练习

可配置的旋钮颜色

在这个练习中，你需要实践如何像 SwiftUI 的内建 view 那样，通过环境来让一个自定义的 view 可配置。基于上面旋钮的例子 (你可以从 GitHub 上下载完整代码)，为它添加配置颜色的方法。

由于旋钮已经包含了根据环境配色风格 (浅色或者深色模式) 来使用不同颜色的逻辑，这个基于环境的颜色配置会是一种用来覆盖默认的行为的可选配置。

步骤 1：创建 Environment Key

以 ColorKey 为名字，创建一个新的 environment key，它包含一个可选的颜色值。然后为这个 key 在 EnvironmentValues 的扩展中添加一个属性。之后，在 View 上创建一个 knobColor 渐变方法，用来设置环境中的旋钮颜色。

步骤 2：创建 @Environment 属性

在 Knob view 中为旋钮颜色创建一个 @Environment 属性，如果它不是 nil，则使用它；否则，使用基于配色方案的填充色。

步骤 3：通过滑条控制颜色

创建一个滑条控件 (Slider) 来控制颜色 (比如，你可以用这个滑条来控制色调 (hue) 值)，并创建一个开关 (Toggle) 来切换旋钮是使用自定义颜色还是使用默认颜色。

布局

4

View 布局过程的任务是为 view 树中的每个 view 分配位置和大小。在 SwiftUI 中，布局算法原则上很简单：对于层级中的每个 view，SwiftUI 都会提供一个建议尺寸 (proposed size，可用空间)。View 将自己布局在这个可用空间内，并报告自己的实际尺寸。然后，系统 (默认情况下) 将 view 置于可用空间的中心。尽管没有为此提供的公开的 API，请想象每个 View 都实现了以下方法：

```
struct ProposedSize {
  var width, height: CGFloat?
}
extension View {
  func layout(in: ProposedSize) -> CGSize {
    // ...
    for child in children {
      child.layout(in: ...)
    }
    // ...
  }
}
```

为了解释各个 view 的布局行为，我们假定上述 layout 方法真的存在。

在 SwiftUI 中，让布局变得复杂的原因是，在通过建议尺寸来决定实际尺寸的时候，每个 view (或者是 view 修饰器) 的表现会有所不同。比如，一个形状 (Shape) view 总会尝试将自己完整填入建议尺寸中；而 HStack 则占用它的子 view 所需要的尺寸 (最多占用到建议尺寸)；一个 Text view 则需要足够渲染文本的尺寸，除非文本的大小超过了建议尺寸，在这种情况下，文本将会被缩短省略。

通常来说，建议尺寸的两个方向上都是非 nil 的值。在某个方向上如果值是 nil，则表示 view 可以在这个方向上使用它的理想尺寸 (ideal size)。关于理想尺寸，我们会在本章后面的部分再研究。

虽然布局过程会为每一个 view 设定一个尺寸和位置 (也就是说，一个矩形区域)，但是 view 并不总是需要在这个边界中绘制自身。这在处理动画的时候会非常有用：我们会想要保持 view 的布局位置，这样其他的 view 就可以保持在原地，但是这个 view 可以按照一定的偏移和旋转来进行绘制。

在本章的剩余部分，我们会探索 SwiftUI 的布局过程。我们会从一些像是 Text 和 Image 这样的基础 view 开始，然后再研究几个布局修饰器，像是 .frame 和 .offset。最后，我们会谈谈 stack view。

基本 View

让我们仔细研究一些常用的 view 的布局行为，包括 Shape、Image 和 Text。

由于 SwiftUI 中 view 的确切的布局行为并没有写在文档里，所以我们只能通过实验来确定它。一种简单的方法是把任意的 view 包装到一个 frame 里。frame 的尺寸由两个滑条来控制，这可以让实验变得容易一些：

```
struct MeasureBehavior<Content: View>: View {
 @State private var width: CGFloat = 100
 @State private var height: CGFloat = 100
 var content: Content

 var body: some View {
   VStack {
     content
       .border(Color.gray)
       .frame(width: width, height: height)
       .border(Color.black)
     Slider(value: $width, in: 0...500)
     Slider(value: $height, in: 0...200)
   }
 }
}
```

我们绘制了两条边线，来显示实际的尺寸：灰色边线表示的是 view 的边框范围，而黑色则是 frame 的边框。有时候 view 选择的尺寸比建议尺寸要小，这时候灰色边框就会比黑色边框小。当 view 要大于建议尺寸时 (绘制在建议尺寸的边界之外)，灰色边框就会比黑色边框要大。

如果我们想要看到 view 的实际尺寸，最简单的方式就是在它周围加上边框。

Path

Path 类型代表了一组 2D 的绘制指令 (和 Cocoa 中的 CGPath 类似)。当 Path 上的 layout 方法被调用时，它总会将建议的尺寸作为实际尺寸返回。如果所建议的某个方向的值为 nil，那么它返回默认值 10。举个例子，下面这个 Path 会在左上角绘制一个三角形：

```
Path { p in
  p.move(to: CGPoint(x: 50, y: 0))
  p.addLines([
    CGPoint(x: 100, y: 75),
    CGPoint(x: 0, y: 75),
    CGPoint(x: 50, y: 0)
  ])
}
```

上面的例子中，这条路径的边界矩形原点为 0，宽度 100，高度 75。虽然这条路径是绘制在这个矩形内的，但是 layout 方法会忽略掉这个边界矩形，它依然返回被建议的尺寸。

Shape

通常，我们会想让 Path 适配或者填充被建议的尺寸。我们可以通过使用 Shape 协议来达到这个目的。下面是 Shape 的完整定义：

```
protocol Shape : Animatable, View {
  func path(in rect: CGRect) -> Path
}
```

Shape 和 Path 拥有最容易被预测的布局行为。和 Path 一样，Shape 的 layout 方法也总会将建议的尺寸作为实际尺寸返回。类似于 Path，Shape 也会在建议尺寸的某个维度为 nil 时选择使用默认值 10。在布局过程中，Shape 会接收到 path(in:) 的调用，其中的 rect 参数所包含的尺寸正是建议尺寸。这样就可以绘制一个依赖与建议尺寸的 Path 了。

像是 Rectangle、Circle、Ellipse 和 Capsule 这些内建的形状，会将它们自身绘制在建议尺寸中。那些没有高宽比约束的形状，像是 Rectangle，会选择填满整个可用的空间，而像是 Circle

这样的形状则会适应可用空间。当我们创建自定义形状时，最好遵守同样的行为，把可用空间纳入考虑。比如说，下面的代码同样是绘制三角形，但它定义了一个填满建议尺寸的 Shape：

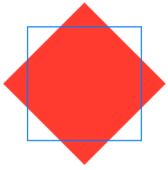
```
struct Triangle: Shape {
  func path(in rect: CGRect) -> Path {
    return Path { p in
      p.move(to: CGPoint(x: rect.midX, y: rect.minY))
      p.addLines([
        CGPoint(x: rect.maxX, y: rect.maxY),
        CGPoint(x: rect.minX, y: rect.maxY),
        CGPoint(x: rect.midX, y: rect.minY)
      ])
    }
  }
}
```

Shape 可以拥有修饰器。比如，我们可以创建一个旋转过的 Rectangle：

```
Rectangle()
  .rotation(.degrees(45))
  .fill(Color.red)
```

rotation 修饰器返回的是一个 RotatedShape<Rectangle>。在 RotatedShape 的 layout 方法中，它会将建议尺寸不加修改地传递给子 view，而且它也会返回同样的尺寸。换句话说，上面的旋转后的矩形其实在它的边界之外进行了绘制。要实际看到这一点，我们可以为矩形加上边线：

```
Rectangle()
  .rotation(.degrees(45))
  .fill(Color.red)
  .border(Color.blue)
  .frame(width: 100, height: 100)
```



蓝色的边界显示了一直以来布局系统所考虑的尺寸，但矩形本身在这个边界外部进行了绘制。形状上的 offset 修饰器也表现出同样的行为：它不会去更改布局，不过它 (也就是 OffsetShape) 会将形状绘制到一个不同的位置上去。

Image

默认情况下，Image view 的尺寸是固定的，也就是图片以 point 为单位的大小。这意味着图片 view 的 layout 方法会忽略掉布局系统所建议的尺寸，总是返回图片的尺寸。

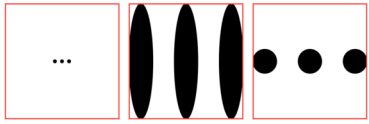
要让一个图片 view 尺寸可变，或者说，想让它能接受建议尺寸，并将图片适配显示在这个空间里，我们可以在它上面调用 .resizable。默认情况下，这会拉伸图片，让它填满整个建议尺寸的空间 (我们也可以设置成以瓷砖平铺的方式，或者只拉伸图片的某个部分来进行填充)。因为大部分的图片应该是需要以固定的高宽比展示的，所以 .aspectRatio 经常被直接搭配在 .resizable 后面组合使用。

.aspectRatio 修饰器会去获取建议尺寸，并且基于给定的高宽比，创建一个能最大限度填满建议尺寸的新的尺寸值。接下来，它会将这个尺寸建议给大小可变的图像 (resizable 的图像会填满整个建议尺寸)，并将该尺寸返回给上层 view。我们可以选择适配或者填充这个建议尺寸，我们也可以决定是要指定一个高宽比，还是将高宽比留给子 view 去做决定。

下面是相同图片在三种不同的显示方式下的样子。第一张图片没有改变；第二张的图片大小可变，它填满了建议尺寸；第三张图片多加了高宽比的设置，因为我们只定义了 content mode，所以这个 aspectRatio 修饰器从它的子 view (也就是图片) 中去获取高宽比的数据：

```
let image = Image(systemName: "ellipsis")
HStack {
  image
  image.resizable()
```

```
    image.resizable().aspectRatio(contentMode: .fit)
}
```



(对于上面的图像，我们为 HStack 中的每个 view 都添加了 100×100 的 frame，并为它加上了红色边界。)

Text

Text view 的 layout 方法总会尝试将它的内容适配到建议尺寸中去，作为结果，它将返回被渲染的文本的边界框。如果底层的字符串不包含换行符，它会 (当水平方向上空间足够时) 尝试将所有文本渲染到同一行。如果水平方向的空间不够了，Text 会检查竖直方向上是否有空间。如果有足够的竖直空间，那么它会将文本分割为多行 (换行) 然后尝试将整个文本适配到建议尺寸中去。如果竖直方向上也没有空间了，文本将被截断并省略。

我们可以用这些 view 修饰器来自定义大部分的行为：

- → fixedSize 会用 nil 作为建议尺寸，此时 text view 使用它的理想尺寸。更重要的是，这会阻止换行。这确实意味着文本可能会被绘制在向 fixedSize 所建议的尺寸之外。
- → lineLimit 指定最大行数。如果文本包含有换行符且行数限制小于文本中的行数，那么文本将会在最后一行的尾部被截断。
- → minimumScaleFactor 允许 Text (在文本无法适配尺寸时) 以更小的字号进行渲染。
- → truncationMode 决定了截断的行为，比如是在文本开头，中间，还是在尾部进行截断。

布局修饰器

View 的布局会受到修饰器的影响：它们是定义在 View 协议上的扩展方法，能生成一些可以改变布局行为的包装 view。

Frame

.frame 修饰器有两个版本：一个指定固定尺寸的边界，另一个指定可变边界。

对于固定尺寸的 .frame 调用，我们需要指定宽度值或者高度值 (或者两个)，以及一个对齐方式。默认情况下，宽度和高度参数都是 nil，对齐方式是 .center。

比方说，当我们指定了一个 width 值时，frame 的 layout 方法会将这个固定的宽度向它的子 view 进行建议 (height 同理)。layout 方法也会将这个固定的尺寸作为它的尺寸返回。如果只有一个方向上是固定的，那么返回的尺寸将在另一个方向上使用它的子 view layout 方法所返回的值。比如，将一个只固定了宽度的 frame 使用在 text view 上：

```
Text("Hello, world")
  .frame(width: 100)
```

这意味着不管 frame 所接收到的建议尺寸是多少，text 的 layout 方法总会收到宽度为 100 的建议尺寸。因为只有 width 被指定了，.frame 修饰器的 layout 方法将返回一个宽度为 100，高度为 Text 高度的尺寸。用代码表达的话，会是类似这样：

```
func layout(in proposedSize: ProposedSize) -> CGSize {
  let proposedChildSize = ProposedSize(width: 100,
    height: proposedSize.height)
  let childSize = child.layout(proposedSize: proposedChildSize)
  return CGSize(width: 100, height: childSize.height)
}
```

当子 view 的尺寸和 frame 的尺寸不一致时，frame 的 alignment 会被用来确定子 view 的位置。比如说，如果我们把上面代码中的文本缩短，让它窄于 100 point，这样在水平方向上就会有一些空白空间。默认情况下，对齐方式是 .center，text view 将会在 100 point 的宽度内居中显示。我们也可以将 view 做首对齐 (leading) 或者尾对齐 (trailing)，或者甚至使用自定义的对齐准线 (alignment guide)。注意，对齐不仅仅只在子 view 比固定尺寸小的时候有用，在子 view 比固定尺寸大的时候，它也会生效。

可变 frame 的工作方式类似。不过我们不再给定一个固定的尺寸，而是为宽和高给出一组最小值、理想值和最大值 (和固定尺寸的 frame 类似，我们可以将任意参数留空，这样它们会使用

默认的 nil 值)。某个方向上的最小值和最大值将作为建议尺寸和返回尺寸的**钳位** (clamping)。举例来说，当我们对最大宽度进行了配置时，.frame 修饰器的 layout 实现会去检查被建议的宽度，如果这个被建议的宽度超过了设置的最大宽度，那么它只会将最大宽度建议给它的子 view。类似地，如果子 view 返回了一个比最大宽度更大的宽度，那么这个结果也将被钳至最大宽度值。

如果被建议的某个维度为 nil，那么就会使用理想值。举例来说，如果 frame 接收到的建议尺寸的宽度为 nil，同时理想尺寸被设置了，那么 frame 修饰器将把这个理想尺寸建议给它的子 view。同样，frame 修饰器也会在被建议的宽度是 nil 时，忽略掉它的子 view 的宽度，而将理想宽度作为它自身的宽度进行返回。当然，同样的逻辑也适用于高度。想要在尺寸的某个维度上建议 nil，我们可以使用 .fixedSize() 修饰器。

作为示例，我们来试试看把View 更新一章中的 KnobShape 包装到一个 view 里，让它的行为和 Image 更相似：默认情况下，旋钮拥有一个固定尺寸，但是你能通过调用 .resizable() 来创建一个可以缩放的旋钮。注意，这里的 resizable() 不是内建的方法，它是我们自行定义在 Knob 上的：

```
struct Knob: View {
  var body: some View {
    KnobShape().frame(width: 32, height: 32)
  }

  func resizable() -> some View {
    KnobShape()
      .aspectRatio(1, contentMode: .fit)
      .frame(idealWidth: 32, idealHeight: 32)
  }
}
```

Knob().resizable() 会创建一个将自身适配到建议尺寸中的 view (因为它拥有一个可变的 frame)。我们需要调用 aspectRatio 来确保 view 的宽度和高度相同，以避免变形。当我们对这个可缩放的旋钮调用 .fixedSize() 时，它的建议尺寸在高宽两个方向上都是 nil，这样一来，旋钮就将使用理想尺寸的值，也就是我们设定的 32 point。

> 注意，在 stack 上调用 fixedSize() 经常会导致预期外的行为，这是因为 stack view 在布局它的内容时需要进行两轮操作 (我们会在本章末尾讨论 stack view 的工作方式)。

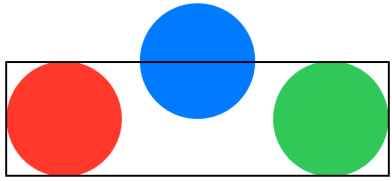
Offset

offset 修饰器将建议尺寸转发给它的子 view，并将子 view 的尺寸作为自身尺寸进行汇报。换句话说，它并不影响布局。不过，它会将子 view 绘制在一个不同的地方 (由 offset 调用所决定水平和竖直方向的偏移量)。

我们发现，offset 在处理动画和交互时会特别有用。比如说，当我们展示一个项目可拖拽的列表时，我们可以用 offset 来把被拖拽的项目绘制到拖拽位置，同时维持它在列表中所占用的空间。

在下面的例子中，我们为中间圆圈的 y 设定了 -30 的偏移量。这会使 view 被绘制在不同的地方，但是布局却不会受到影响：

```
HStack {
   Circle().fill(Color.red)
   Circle().fill(Color.blue).offset(y: -30)
   Circle().fill(Color.green)
}
.frame(width: 200, height: 60)
.border(Color.black)
```



Padding

padding 修饰器用来为 view 添加边缘填充，它是最简单的修饰器之一。这个修饰器的完整版本可以接受一个 EdgeInsets 作为参数。或者，换句话说，我们可以为每条边 (顶边，底边，先头，

末尾) 分别设置填充。同时也存在一些简便的变形方法，比如，我们可以使用不带任何参数的 .padding() 来为每条边添加系统指定的默认填充。

在 padding 修饰器的 layout 方法里，它会将接收到的建议尺寸减去设定的 padding，然后将得到的新尺寸建议给它的子 view。然后，它将子 view 返回的尺寸加上这些 padding 作为自己的尺寸进行返回。相应地，它也会将子 view 放到正确的位置上 (实际上，它将子 view 按照收到的先头和顶边 padding 进行偏移)。

注意，我们同样可以指定负的 padding 值，这将会导致 padding view 向它的子 view 建议一个更大的尺寸。

Overlay 和 Background

overlay 和 background 修饰器在布局系统中同样是非常重要的部分。当我们使用 content.overlay(other) 时，系统会创建一个带有两个子 view 的 overlay 修饰器 view：它们分别是 content 和 other。

当布局一个 overlay 修饰器时，被建议的尺寸会被传给 content。然后 content 报告的尺寸会被作为建议尺寸传递给 other。overlay 修饰器会把 content 的尺寸作为它自己的尺寸进行返回：换句话说，other 所报告的尺寸将被忽略。overlay 的 layout 实现类似这样：

```
extension Overlay {
  func layout(in proposedSize: ProposedSize) -> CGSize {
    let contentSize = content.layout(proposedSize: proposedSize)
    _ = foreground.layout(proposedSize: ProposedSize(
      width: contentSize.width, height: contentSize.height))
    // ...
    return contentSize
  }
}
```

对于 content.background(other)，除了 other 现在被绘制在 content 的后面之外，整个过程大部分是一样的。需要特别注意的是，content.overlay(other) 和 other.background(content) 是**不一样**的：前一种写法，content 的尺寸被用作布局尺寸，而后一种写法中 other 的尺寸被作为最终尺寸。

overlay 和 background 经常被和形状一同使用。具体来说，要在 Text 周围绘制一个胶囊形状的外框，同时不对布局产生影响，可以这么做：

```
Text("Hello").background(
  Capsule()
    .stroke()
    .padding(-5)
)
```

让我们来看看怎么绘制一个和 iOS 内置的秒表 app 里类似的圆形按钮。首次尝试，我们可能会在文本的 .background 里画一个圆圈：

```
Text("Hello, World!")
  .foregroundColor(.white)
  .background(Circle().fill(Color.blue))
```

当我们执行上面的代码时，会看到一个和文本同样高的小圆圈。虽然这个结果不是我们想要的，但是它确实是可以合理解释的：首先，文本的尺寸被确定，接下来圆圈被绘制在文本的尺寸之内。默认情况下，圆圈会将自身适配到可用空间中去 (使用宽度和高度中的最小值作为自己的直径)。

要解决这个问题，我们可以将 text 作为圆圈的 overlay 来进行绘制。这样，我们就可以为圆圈指定宽度和高度：

```
Circle()
  .fill(Color.blue)
  .overlay(Text("Start").foregroundColor(.white))
  .frame(width: 75, height: 75)
```

理想状况下，当文本无法放在圆圈内的时候，按钮应该也会相应变大一些。但不幸的是，要实现这一点，我们需要一些更先进的技术 (我们会在下一章里展示相关内容)。

我们也能将多个叠层组合起来使用。比如，我们可以绘制一个相同的按钮，但是加上一个稍微向内缩进一些的小号圆圈：

```
Circle()
  .fill(Color.blue)
  .overlay(Circle().strokeBorder(Color.white).padding(3))
  .overlay(Text("Start").foregroundColor(.white))
  .frame(width: 75, height: 75)
```

裁切和遮罩

最后介绍的两种修饰器是 clip 和 mask，它们可以和上面的修饰器组合使用，创造一些很有用的效果。两者都不会影响布局，但是它们会影响屏幕上的绘制。当我们使用 .clipped() 时，view 会按照它的边界矩形进行 “裁切”。换句话说，view 绘制在边界矩形外部的部分都将变得不可见。下面这个例子中，我们使用了之前的旋转后的矩形：

```
Rectangle()
  .rotation(.degrees(45))
  .fill(Color.red)
```

默认情况下，矩形会由于旋转，而将部分内容绘制在边界之外。不过，如果我们加上 .clipped 修饰器，对这个旋转后的矩形来说，只有在父 view 的边界框中的部分可见：

```
Rectangle()
```

```
  .rotation(.degrees(45))
  .fill(Color.red)
  .clipped()
  .frame(width: 100, height: 100)
```

还有一个类似的 clipShape 方法，和 clipped 不同，它接受一个形状作为裁剪蒙版，而不是直接使用边界矩形。有一个有趣的事实，圆角效果 (使用 .cornerRadius) 是通过使用一个 RoundedRectangle 调用 clipShape 实现的。

最后，我们可以通过 .mask 来提供遮罩。mask 和 clipped 的不同在于，mask 可以接受任意的 view，并使用这个 view 来作为下层 view 的掩模板 (让下层 view 的内容只在上层 view 的像素区域中进行显示)。

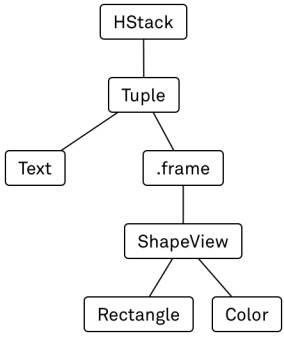
Stack View

在 SwiftUI 中，stack view 是从单个 view 开始构建复杂布局的核心机制。Stack view 有三种不同形式：它们可以将子 view 们按照水平方向 (HStack)，竖直方向 (VStack) 或是垂直于屏幕表面的方向 (ZStack) 来进行布局。在本章中，我们会专注讨论 HStack，不过同样的规则也适用于其他的 stack 类型。

以下面这个水平的 stack 为例：

```
HStack {
  Text("Hello, World")
  Rectangle().fill(Color.red).frame(minWidth: 200)
}
```

它的可视化的 view 树结构会帮助我们进行理解：



将这个 view 包装到 (在本章开头时提到过的) MeasureBehavior view 中，会看到一些有趣的结果。慢慢减少宽度，然后看看发生了什么：当宽度还足够大时，text 会按照适配的方式显示，而矩形将填满剩余的空间。

在我们的例子中，text 的理想尺寸宽度为 93 point。当 HStack 的 layout 方法所接收到的建议尺寸宽度小于 301 (93 + 8 + 200) point 时，布局就不再能够完全适配了。但发生的事情很有趣：一开始，text **并没有**将文本变为多行，红色的矩形也没有缩小。相反，HStack 在绘制自己时，选择了在超出建议尺寸的部分进行绘制 (如果我们使用 MeasureBehavior，我们将会看到灰色边界要比黑色边界更宽)。要理解这个行为，我们需要仔细研究布局算法。

HStack 的 layout 方法将经过两轮工作。在第一轮中，HStack 会指出哪些子 view 拥有固定的宽度，在第二轮中，可变的尺寸将会被分割并提供给那些拥有可变宽度的子 view。

在第一轮中，HStack 的 layout 将获取到的水平方向的空间，减去子 view 之间的间距，然后将剩余的空间均分，并把每一份提供给每个子 view。比如说，我们有两个 view，子 view 间使用的是默认间距 8 point，所以提供给每个子 view 的建议宽度是 (proposedSize.width - 8) / 2。

第一个子 view Text 将会尝试在不换行的前提下将自己适配到建议尺寸中去。如果这个建议尺寸足够大，那么它就会回报文本边框 (也就是布局后的字形的宽度和高度) 作为自己的尺寸。然而，如果没有足够的空间让 text 在一行中渲染，Text 就将在水平方向上压缩自身：根据配置，这种压缩可能是将文本布局为多行，或者是将文本截断。Text 永远不会比它的边框要大。

HStack 的第二个子 view 是包含一个矩形的 frame。矩形会始终将自己适配到提案的尺寸中去。不过，frame 的配置规定了最小宽度为 200。当 HStack 的 layout 方法为 frame 建议的尺寸小

于 200 时，它将忽略掉这个建议的宽度，并将 200 作为建议宽度传递给它的子 view (也就是 Rectangle)。frame 的 layout 方法的返回值也依赖于被建议的宽度：如果被建议的宽度小于 200 的话，frame 的 layout 将返回 200。否则，它将会返回被提议的宽度。

在第一轮工作结束后，HStack 将会知道那些元素的宽度是可变的，哪些是固定的。它也可以知道那些固定元素的实际宽度。一个元素的尺寸可以是完全固定的，也可以是以最大宽度可变，或者以最小宽度可变。在我们的例子中，Text 将会汇报它拥有固定的宽度，而矩形将报告它是可变的 (最小宽度为 200)。注意，虽然同一个 Text 可以对不同的建议尺寸返回不同的宽度，但是对于确定的建议尺寸，它所返回的这个宽度自身是确定不变的。

> 注意，我们还不是完全确定 “可变性” 在底层是如何实现的。可能是布局系统通过建议几个不同的尺寸来观察 view 的表现？也可能是 view 本身会为每个方向报告一个可用的范围？不管如何，对我们的学习目的来说，这关系不大，我们主要还是关心 view 表现出来的行为。

现在 HStack 的第二轮开始了。它会把获取到的固定尺寸元素的宽度从被建议的宽度中减去。同时它也会减去元素间距，最后得到的是可变的空间。接下来它把可变空间平均分配给每个可变的子 view。

在上面的例子中，如果水平方向空间足够的话，HStack 将首先确保 Text 的宽度 (它是固定的) 和元素之间的间距，将它们从被建议的尺寸中减去。因为矩形拥有一个可变的最小宽度，所以它会将剩余的建议宽度全部填满。

Text view 将会渲染 93 point 的宽度，我们已经说过，完整渲染所有内容，需要 301 point。如果 HStack 得到的建议尺寸是 250 point 时，在第一轮中，它会首先减去间距 (8 point)，然后将 121 point (经由 (250-8)/2 算出) 的宽度建议给每个子 view。因为 Text 只需要 93 point，所以当它获取到 121 point 的建议时，它可以很好地将自己适配到其中，并将 93 point 作为固定尺寸汇报。矩形 (或者更准确来说，是 .frame 修饰器) 最少需要 200 point 宽，而这要比所建议的宽度要大。这就是说，Text 会被绘制在 93 point 里，而 Rectangle 会使用 200 point，以及还需要它们之间的 8 point 间距，最后整个 HStack 的宽度为 301。换言之，HStack 会比被建议的宽度要宽，它的内容将会被绘制到边界外部去。

布局优先级

还有一种控制 stack 布局的方式：那就是通过布局优先级。比如，当我们想要显示一个文件的路径时，路径的最后一部分 (文件名) 通常来说是最重要的。要显示一个长路径，我们可以将它分割为基本路径和文件名，然后把它们组合起来构建我们的 view：

```
HStack(spacing: 0) {
  Text(longPath).truncationMode(.middle).lineLimit(1)
  Text("chapter1.md").layoutPriority(1)
}
```

当空间不足时，HStack 会先把可用空间提供给 Text("chapter1.md")。之后剩余的空间再被提供给 Text(longPath)，当它无法将自己适配到剩余空间时，它会自动截断，并在中间部分显示省略符号。

记住 stack 布局的过程的描述：在第一轮，stack 的 layout 方法决定每个子 view 的尺寸是固定的还是可变的。在第二轮，可变的空间被分配给可变子 view。当布局优先级被设置时，第二轮会更加复杂一些。元素们会被按照布局优先级分组：拥有最高布局优先级的组将会首先被提供可变空间，然后是第二高的组，以此类推。

如果最高布局优先级的组包含真正可变的元素 (没有任何约束)，这意味着其他组将不会得到任何空间。不过，布局优先级在大部分时候都用来处理那些有最大宽度的元素。这类元素的最大宽度可能是由 frame 设置的，也可能是 view 的内建特性，比如 Text 就是一例。

Stack 对齐

我们在上面讨论了 stack view 是如何在它们的主轴上排列它们的子 view 的。但是，这些 view 也需要在另一个轴上进行排列：对于 HStack 来说是竖直方向，对 VStack 来说是水平方向，而对 ZStack 类说则是竖直和水平两个方向。在次要轴上的排列需要用到对齐：每个 stack view 都有一种唯一的对齐方式 (对 ZStack 来说，则是两种，屏幕平面上每个轴一种对齐)。对于水平和竖直，默认的对齐都是居中对齐。接下来，我们来详细看看对齐在 HStack 上是如何工作的，当然了，对于 VStack 和 ZStack 来说，只是应用在不同的轴上，但工作的方式是一样的。

在 HStack 中，指定的对齐方式会被用来向每个子 view 进行提问。比如说，当我们指定了 .center 作为对齐方式时，每个子 view 都会被询问 (在它自己的坐标系统中) 它的竖直方向上的中线在哪里。然后，所有子 view 的中线会被放置在一条 (想象中的) 线上。除了 .center，我们还可以选择 .top 或者 .bottom，这些 view 会分别进行顶端对齐和底端对齐。最后，我们还可以选择让 view 们沿着文本基线 (text baseline) 进行对齐，这在处理多个不同字号的文本时会很有用。下图展示了三种 HStack 的对齐方式，它们分别采用顶端对齐，中央对齐和底端对齐：

自定义对齐

我们还可以创建自定义的对齐准线，或者为每个独立的 view 修改它的对齐行为。对齐准线本质上来说很简单：它是一种计算 view 中某个点的方式。举例来说，在竖直对齐中，对齐准线会给出一个竖直轴上的点。每个对齐准线都实现了一个接受 ViewDimensions 参数并返回 CGFloat 的方法。(ViewDimensions 结构体和 CGSize 很类似，不过它有一些额外的辅助方法。)

比方说，.center 这个对齐准线可以用两部分来重新实现。首先，我们需要一个自定义的识别类型，它拥有一个默认的实现，来计算 view 中竖直方向上的对齐方式：

```
enum MyCenterID: AlignmentID {
  static func defaultValue(in context: ViewDimensions) -> CGFloat {
    return context.height / 2
  }
}
```

接下来，我们需要为 VerticalAlignment 添加一个扩展，来使我们的自定义对齐对外可见：

```
extension VerticalAlignment {
  static let myCenter: VerticalAlignment = VerticalAlignment(MyCenterID.self)
}
```

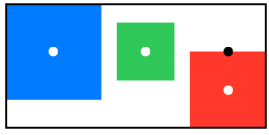
最后，我们就可以使用这个新的布局了：

```
HStack(alignment: .myCenter) {
  Rectangle().fill(Color.blue).frame(width: 50, height: 50)
  Rectangle().fill(Color.green).frame(width: 30, height: 30)
}
```

一旦 HStack 知道了每个子 view 的尺寸，它就可以使用 defaultValue(in:) 方法来计算每个 view 的中心位置了。这个方法会为每个子 view 调用一次，而且它将获得每个子 view 的尺寸作为 ViewDimensions 参数。要注意，子 view 们只能根据它们自己的尺寸来指定位置，它们对同层级的其他 view 或者是父 view 是毫不知情的。

我们也能够覆盖某个指定的子 view 的对齐准线。例如，如果我们想要将所有的子 view 对齐到中线，但是只将其中一个稍微进行一些偏移，我们可以这样做：

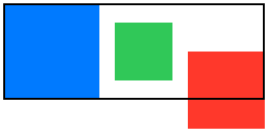
```
HStack(alignment: .myCenter) {
  Rectangle().fill(Color.blue)
    .frame(width: 50, height: 50)
  Rectangle().fill(Color.green)
    .frame(width: 30, height: 30)
  Rectangle().fill(Color.red)
    .frame(width: 40, height: 40)
    .alignmentGuide(.myCenter, computeValue: { dim in
      return dim[.myCenter] - 20
    })
}.border(Color.black)
```



我们将矩形的中心位置都用白点进行了可视化，并且把自定义的对齐值画为了黑点。前两个矩形使用了默认值：在第一个矩形中，ViewDimensions 的高度为 50，所以返回值是 25；第二个矩形 (高度为 30) 返回的值是 15。对于最后一个矩形，我们提供了一个自定义值：我们首先获取了中心位置 20，然后减去 20，返回是 0。这意味着第一个 view 的中线，第二个 view 的中线，以及第三个 view 的顶端，将在一条虚拟的水平线上对齐。注意，如果我们通过这样的方式重载了对齐，alignmentGuide 的第一个参数 (.myCenter) 必须要匹配 HStack 的对齐。

相较于重载对齐准线，我们也可以选择使用 .offset 来移动第三个 view。但是，两者间有很大的差异：offset 并不会更改布局，它只是将 view 绘制在不同的地方，但使用 alignment guide 的方式会改变 stack view 的布局，在计算所有的尺寸和 (使用 alignment guide 的) 水平及垂直对齐后，stack view 会使用一个包围它的所有子 view 的边框，作为它报告的最终尺寸。

下面这个 stack view 看起来和上面一样，但它在第三个 view 上使用的是 offset，而非自定义对齐值。注意，它报告的尺寸要比上面使用自定义对齐值的尺寸要小，第三个 view 将被绘制在边界外面。



组织布局代码

对于 view 布局代码的组织方式，我们可以有很多种选择：我们可以创建新的 View 结构体，可以在 View 上写一些扩展方法，或者可以创建 ViewModifier。要使用哪一种，更多的是有关代码风格的选择。比如，我们可以将我们上面的蓝色圆圈按钮写成一个 View 上的扩展，这样我们就可以在任意 view 上调用它，并在其后方放置一个蓝色圆圈。注意我们使用了 self 作为 overlay 的值 (在之前，这个位置上的值是 Text("Hello"))：

```
extension View {
  func circle(foreground: Color = .white, background: Color = .blue)
    -> some View {
    Circle()
      .fill(background)
      .overlay(Circle().strokeBorder(foreground).padding(3))
      .overlay(self.foregroundColor(foreground))
      .frame(width: 75, height: 75)
  }
}
```

现在，我们可以为任意 view 添加圆圈背景了：

```
Text("Hello")
  .circle(foreground: .white, background: .gray)
```

我们也可以将它写成一个自定义的 View 结构体。这会需要多一些代码，当创建容器 view 的时候，接受一个 view builder 而不是常规的普通 view，是很常见的做法：

```
struct CircleWrapper<Content: View>: View {
  var foreground, background: Color
  var content: Content
  init(foreground: Color = .white, background: Color = .blue,
    @ViewBuilder content: () -> Content) {
    self.foreground = foreground
    self.background = background
    self.content = content()
  }

  var body: some View {
    Circle()
      .fill(background)
      .overlay(Circle().strokeBorder(foreground).padding(3))
      .overlay(content.foregroundColor(foreground))
      .frame(width: 75, height: 75)
  }
}
```

虽然它的渲染方式和其他两种方法中的一样，但是创建 CircleWrapper 的语法会稍有不同：

```
CircleWrapper {
  Text("Hello")
}
```

还有第三种选项：我们可以创建一个 ViewModifier。这通常用在将其他 view 进行包装，或者是改变一个 view 的布局方式的时候：

```
struct CircleModifier: ViewModifier {
  var foreground = Color.white
  var background = Color.blue
  func body(content: Content) -> some View {
```

```
      Circle()
        .fill(background)
        .overlay(Circle().strokeBorder(foreground).padding(3))
        .overlay(content.foregroundColor(foreground))
        .frame(width: 75, height: 75)
    }
}
```

要通过这个修饰器创建一个按钮，我们可以使用 View 上的 modifier 方法。在下面的代码中，结果的类型是 ModifiedContent<Text, CircleModifier>：

```
Text("Hello").modifier(CircleModifier())
```

在动画的章节里，我们会看到关于修饰器的更多的用例。

这三种方式 (通过扩展，通过自定义 view，以及通过修饰器) 所得到的结果是一样的。注意，像是 @State 这样的属性是不能使用在类型扩展里的；它们会需要是自定义 view 或者修饰器的一部分。然而，如果我们选择了创建一个自定义 view 或者修饰器，我们依然可以把它定义到扩展中去。SwiftUI 自己也在这么做，几乎 View 上的每个方法，所返回的都是一个自定义的 View 或者 view 修饰器。要使用哪一种方式，一般都归结到个人偏好和选择。

按钮样式

虽然我们把我们的蓝色圆形按钮称为 “按钮”，但是它其实只是一个后面有个圆圈的文本。这个 “按钮” 没法交互，当我们点击它时，什么都不会发生，用户点击的时候也没有视觉上的效果提示。

要为按钮加上样式，我们可以使用 ButtonStyle 协议。它和 ViewModifier 类似，只不过它并不是接受一个 content 作为参数，而是接受一个包含有按钮文本标签以及按钮点击状态的结构体。我们可以用这些信息来为按钮设定样式：

```
struct CircleStyle: ButtonStyle {
  var foreground = Color.white
  var background = Color.blue

  func makeBody(configuration: ButtonStyleConfiguration) -> some View {
```

```
        Circle()
            .fill(background.opacity(configuration.isPressed ? 0.8 : 1))
            .overlay(Circle().strokeBorder(foreground).padding(3))
            .overlay(configuration.label.foregroundColor(foreground))
            .frame(width: 75, height: 75)
    }
}
```

要按照这个按钮样式创建按钮，我们可以在按钮上调用 buttonStyle：

```
Button("Button", action: {})
  .buttonStyle(CircleStyle())
```

buttonStyle 的另一个优点是，我们可以一次性地为多个按钮添加样式。buttonStyle 修饰器是定义在 View 上的，它会更改环境。比如，我们可以用它同时为 HStack 中的多个按钮设置样式：

```
HStack {
  Button("One", action: {})
  Button("Two", action: {})
  Button("Three", action: {})
}.buttonStyle(CircleStyle())
```

当为按钮设置样式时，自定义一个 ButtonStyle 一般来说都会是最好的选择。

重点

- → SwiftUI 的布局过程操作是从上而下的：父 view 向子 view 建议尺寸，子 view 将它们布局在这个建议尺寸中，然后返回它们实际需要的尺寸。

- → 建议尺寸和实际尺寸的关系根据 view 的不同而有所不同。比如说，形状和图片总是会使用全部的建议尺寸，而 text view 只需要它们的内容所适配占有的尺寸。
- → 像是 .frame 和 .padding 这样的布局修饰器可以用来调整布局。其他的像是 .offset 和 .rotation 等修饰器，则只影响 view 的绘制，而不改变布局。
- → 使用 stack view 将单独的 view 构建为复杂的布局，是很常见的方式。我们可以通过布局优先级和对齐准线来自定义它们的行为。
- → 我们应该将通用的 view 相关的代码组织为扩展、自定义 view、或者是修饰器。

练习

你可以在本书末尾的附录答案中找到解答，你也可以从 GitHub 下载完整的项目代码。

可折叠的 HStack

为 HStack 写一个可折叠的包装 view。下面是它应该提供的接口：

```
struct Collapsible<Element, Content: View>: View {
  var data: [Element]
  var expanded: Bool = false
  var content: (Element) -> Content
  var body: some View { ... }
}
```

当 expanded 为 true 时，这个 view 看起来应该就和一个普通的 HStack 一样，但是如果 expanded 是 false 时，子 view 们需要堆叠起来绘制在前一个的上方。最后一个子 view 将完全可见。两种状态如下截图所示：

当使用动画把 expanded 从 true 改为 false，子 view 们应该在两种状态之间动画切换。注意，我们在 Collapsible view 周围添加了黑色的边框，这告诉我们该 view 的内容不应该被绘制在边框之外。

附加练习

让 Collapsible stack 拥有更多的可配置选项，特别是竖直方向的对齐方式和间距。

角标 View

写一个 view 扩展，来在一个 view 的右上角显示一个角标，同时不影响周围 view 的布局。对应的使用方法应该如下：

```
struct ContentView: View {
  var body: some View {
    Text("Hello")
      .padding(10)
      .background(Color.gray)
      .badge(count: 5)
  }
}
```

如果 count 是 0，那么角标应该被隐藏。或者，也可以让 badge 接受一个可选的 Int，并且在 count 为 nil 是隐藏自身。

附加练习

→ 确保你的角标 view 在 “从右向左” 的布局方向下也可用。

→ 为角标添加可配置选项，来决定它相对于 view 的位置 (尾端还是首端)，尺寸，颜色等。

自定义布局

5

在上一章中，我们研究了 SwiftUI 内建的指定布局的几种方式。在很多情况下，使用 view 布局一章中介绍的 view 和修饰器可以让我们表达出想要的布局，但是有些时候我们会想要更加自定义的布局方式。比如说，如果我们想要依据可用宽度，去显示完全不同的 view，应该要怎么做？或者如果我们想要实现一个类似 flow layout 的方式，让多个项目水平布局在一条线上，然后在新的项目无法再适配进屏幕范围内时新开一行 (就像是渲染文本时的换行那样)，又要怎么做呢？

在这一章中，我们会以上一章的内容作为基础，去构建一些布局，并展示可以帮助你对布局过程进行挂钩 (hook) 和自定义的技术。我们会讨论能让你接收到被建议的布局尺寸的几何读取器 (geometry reader)，能让你沿着 view 树向上传递信息的 preference，以及 anchor。最后，我们会展示如何将这些技术组合起来，去构建出一个自定义的布局。

> 本章中所介绍的很多技术感觉上是对于 SwiftUI 中当前一些限制的临时解决方案。我们相信在未来的版本中，SwiftUI 会为我们提供更好的 API，那时本章中的代码应该会变得更简单，甚至完全不再需要。也就是说，当前，如果不使用这些技术的话，有些布局是不可能实现的。我们建议你在想要使用这样的布局时，需要格外小心谨慎。

几何读取器

我们可以通过使用 GeometryReader 来与布局流程进行挂钩，以改变部分布局行为。几何读取器最重要的功能，是让我们可以接收到一个 view 的被建议的布局尺寸。和其他所有容器 view 一样，GeometryReader 是通过一个 ViewBuilder 来配置的，和其他容器不同的是，几何读取器的 view builder 接收一个参数，它的类型是 GeometryProxy。这个 proxy 有一个属性，可以访问到 view 的被建议的布局尺寸，以及一个用来解析锚点 (anchor) 的下标方法。在 ViewBuilder 中，我们可以使用这些信息来布局子 view。

比如，如果我们想要绘制一个宽度为建议尺寸宽度三分之一的 Rectangle，我们可以这样使用 GeometryReader：

```
GeometryReader { proxy in
  Rectangle()
    .fill(Color.red)
    .frame(width: proxy.size.width/3)
}
```

当使用 GeometryReader 时，有一个重要的注意事项，那就是它会把被建议的尺寸作为实际尺寸进行返回。由于这一尺寸特性，几何读取器经常被用作其他 view 的背景或叠层：它们的尺寸与对象 view 的尺寸将完全一致。我们可以用这个尺寸来在 view 的边界上绘制一些东西，或是测量 view 的大小。

在前一章，我们创建了一个类似于 iOS 内置秒表程序的小型圆形按钮：

```
Circle()
  .fill(Color.blue)
  .overlay(Circle().strokeBorder(Color.white).padding(3))
  .overlay(Text("Start").foregroundColor(.white))
  .frame(width: 75, height: 75)
```

不幸的是，我们只能为按钮设定一个固定大小。要让按钮自动适配文本的话，我们需要求助于一个小技巧：我们可以在 text 的背景里放一个几何读取器，然后用它来绘制圆圈。通过在文本的四周添加一些填充，圆圈会比文本要稍微大一些：

```
Text("Start")
 .foregroundColor(.white)
 .padding(10)
 .background(
    GeometryReader { proxy in
      Circle()
        .fill(Color.blue)
        .frame(width: proxy.size.width,
            height: proxy.size.width)
 })
```

上面这种做法的一个问题是，整个 view 的尺寸将会是文本的尺寸加上填充部分的尺寸；圆圈的高度会被忽略掉。比如，如果我们用 .clipped() 来裁剪一个 view 的话，背景就只会剩下蓝色的矩形了。同样地，如果我们在一个 VStack 中放置多个按钮，圆圈区域将会彼此重叠：

要修复这个问题，我们可以在整个 view 的周围加上一个 frame，让它的宽度和高度与文本的宽度 (加上填充区域) 相同。但不幸的是，在 view 被实际布局之前，我们是不知道文本部分的宽度的。在下一节中，我们来看看如何使用 preference 来处理这个问题。

Preference 和 GeometryReader

通过 preference，我们可以将值沿着 view 树向上传递给它的祖先 view。举个例子，如果我们有一种方法可以测量 view 的尺寸，我们就可以使用 preference 把这个尺寸向上汇报给父 view。Preference 是通过键和值来进行设置的，而 view 可以为一个特定的 preference 键设定一个值。

要解决前面一节中的问题，我们的策略是：使用 GeometryReader 来测量我们按钮中的 Text 尺寸。然后我们使用 preference 将这个值沿树向上传递，并在整个 view 外面加上一个高度和宽度都等于文本宽度的 frame (我们假设文本的宽度始终要比它的高度更大)。

使用 PreferenceKey 协议，我们可以定义我们自己的 preference key。这个协议有一个关联类型 (用来定义值的类型) 以及两个要求实现的内容：一个是 (在没有任何 view 定义该 preference 时所应该使用的) 默认值，另一个是将两个值合并的方法。后一个要求的存在，是因为同一个父 view 可能存在多个子 view，而且每个子 view 都可以定义它们自己的 preference。这样，父 view 就需要一种方式，来将所有子 view 的 preference 合并为一个单一的 preference。

回到我们的按钮例子，我们想要收集 Text view 的宽度。我们可以使用 CGFloat? 作为 preference 值的类型，nil 表示我们没有任何 Text。现在，我们不关心如何合并值，所以我们简单地获取我们所得到的第一个非 nil 值：

```
struct WidthKey: PreferenceKey {
  static let defaultValue: CGFloat? = nil
  static func reduce(value: inout CGFloat?,
    nextValue: () -> CGFloat?) {
    value = value ?? nextValue()
  }
}
```

现在我们已经准备好传递我们的新的 preference 值了。和之前一样，我们为文本添加一个 GeometryReader 的 background，它将接受文本尺寸作为它的建议尺寸。要设置 preference，我们需要在几何读取器闭包中的某个 view 上调用 .preference(...)。因为我们不想进行任何绘制，这里我们使用一个 Color.clear view：

```
Text("Hello, world")
  .background(GeometryReader { proxy in
    Color.clear.preference(key: WidthKey.self, value: proxy.size.width)
  })
```

> 我们也尝试了使用 EmptyView() 而非 Color.clear 来传递 preference，但令人惊讶的是，这种方式无法正确工作。我们不是很清楚这是不是一个 bug，还是说有更深层的原因导致了 EmptyView() 和 Color.clear 的不同。

现在树上的任意父层级 view 都可以通过 .onPreferenceChange 来读取 preference 了。比如说，我们可以将这个 preference 存储到一个 @State 变量中：

```
struct TextWithCircle: View {
  @State private var width: CGFloat? = nil
  var body: some View {
    Text("Hello, world")
      .background(GeometryReader { proxy in
        Color.clear.preference(key: WidthKey.self, value: proxy.size.width)
      })
      .onPreferenceChange(WidthKey.self) {
        self.width = $0
      }
  }
```

```
}
```

要在文本周围显示一个圆圈，我们需要为文本添加一个长宽都等于文本 width 属性的边框。然后，我们就可以将 Circle 作为背景 view 添加上去了：

```
struct TextWithCircle: View {
  @State private var width: CGFloat? = nil
  var body: some View {
    Text("Hello, world")
      .background(GeometryReader { proxy in
        Color.clear.preference(key: WidthKey.self, value: proxy.size.width)
      })
      .onPreferenceChange(WidthKey.self) {
        self.width = $0
      }
      .frame(width: width, height: width)
      .background(Circle().fill(Color.blue))

  }
}
```

下面是一个包含三个 TextWithCircle 的 VStack：

当渲染 TextWithCircle view 时，布局引擎首先为 .frame 修饰器建议一个尺寸。因为 TextWithCircle 的 width 属性初始值为 nil，frame 将会把这个建议尺寸沿 view 层级向下进行传递，直至到达 Text view。在 Text 完成自身布局后，文本背景中的几何读取器将使用和文本完全一样的尺寸进行布局。在里面，我们使用了 .preference 来把文本的尺寸沿树向上传递。接下来，.onPreferenceChange 会被触发，view 的状态发生改变，于是 view 的构建和布局过程再次发生。这一次，self.width 将包含文本的宽度，于是 view 被正确布局了。所有这一切都发生在同一次渲染流程 (也就是在屏幕刷新之间) 中。

在处理 preference 的时候，要特别注意的是，在 .onPreferenceChange 的过程中改变状态是被允许的，但是我们需要注意不要写出无限循环：如果一个状态变更造成了一轮新的布局流程，而这轮布局又导致了 preference 的改变，如此循环往复，最终负责 view 更新的系统将会陷入在循环里并彻底卡死。

这种方式也很容易造成其他失误。比如说，要是我们想要在文本周围填充 10 point 的空白 (这样文本就不会触碰到圆圈的边缘了)，我们需要在某个地方加上 .padding(10) 的调用。如果我们在 .onPreferenceChange 之后，而在 .frame 之前来做这件事的话，我们也会被卡在循环中 (你可以自己试试看结果)。我们需要直接在文本上加上 .padding 修饰器，这样它才会被算在被测量的尺寸之中。

锚点

锚点 (anchor) 可以用来在布局层级的不同部分之间进行点或者矩形的传递。锚点本身是对一个值 (比如，一个点) 的包装，它能够在 view 层级中的其他不同 view 的坐标系统内进行解析并获取到新的坐标。我们可以把锚点想象为 UIKit 中 UIView 的 convert(_:from:) 的一个更加安全的替代方法。

比如说，让我们来考虑一个简单的 tab bar 组件，我们想要用它来显示几个以文本方式表示的 tab。被选中的 tab 需要有一个下划线。当用户点击一个新的 tab 时，我们希望同时对下划线的位置和宽度进行动画。这意味着下划线不会是 tab item 的一部分，而是要绘制在 view 层级的其他某个地方。我们可以使用锚点 (通过 preference) 来将被选中 tab 的位置进行传递，然后在 view 层级的另一个部分去解析这个锚点。

我们将通过 ForEach 把 tab 绘制在一个 HStack 里。被选中的项目将会使用 preference 将它的边界 (使用 Anchor<CGRect> 类型) 进行传递。在 view 层级的上层，我们可以读取这个被选

中项目的锚点，并在该位置绘制一条线。第一步，我们需要一个自定义的 preference key。我们定义的 BoundsKey 与上面的 WidthKey 类似，它也使用第一个非 nil 的值：

```
struct BoundsKey: PreferenceKey {
  static var defaultValue: Anchor<CGRect>? = nil
  static func reduce(value: inout Anchor<CGRect>?,
    nextValue: () -> Anchor<CGRect>?) {
    value = value ?? nextValue()
  }
}
```

下面的代码在 HStack 中绘制项目，并且提供锚点值。我们在我们的按钮上添加 anchorPreference，并使用 .bounds 作为值。anchorPreference 修饰器还接受一个 transform 方法，它可以让我们把锚点变形为其他不同的东西。在我们的例子中，我们将它变形为一个可选值的锚点 (只有当前项目是被选中时，它才会是非 nil 值)：

```
struct ContentView: View {
  let tabs: [Text] = [
    Text("World Clock"),
    Text("Alarm"),
    Text("Bedtime")
  ]

  @State var selectedTabIndex = 0

  var body: some View {
    HStack {
      ForEach(tabs.indices) { tabIndex in
        Button(action: {
          self.selectedTabIndex = tabIndex
        }, label: { self.tabs[tabIndex] })
        .anchorPreference(key: BoundsKey.self, value: .bounds, transform: {
          anchor in
          self.selectedTabIndex == tabIndex ? anchor : nil
        })
      }
    }
  }
```

```
}
```

在前面，我们使用了 .onPreferenceChange 来读取 preference 值，但是 .onPreferenceChange 要求值必须满足 Equatable 协议。但不幸的是，Anchor 并没有满足 Equatable。作为替代，我们需要使用 .overlayPreference 或者 .backgroundPreference。这些修饰器会接收到 preference 值，并且分别为 view 添加叠层或者背景。

要将锚点解析到另一个 view 的坐标系统中，我们需要使用 GeometryReader。在几何读取器的 view builder 闭包里，我们可以使用 GeometryProxy 的一个下标方法来把锚点解析到几何读取器的坐标系统中 (这个坐标系统也就是几何读取器所在的叠层的 view 的坐标系统)。在第一次尝试中，我们也许会试着为 HStack 添加一个宽度为锚点 width 值，基于锚点 minX 作为偏移量的 Rectangle：

```
.overlayPreferenceValue(BoundsKey.self, { anchor in
  GeometryReader { proxy in
    Rectangle()
      .fill(Color.accentColor)
      .frame(width: proxy[anchor!].width, height: 2)
      .offset(x: proxy[anchor!].minX)
  }
})
```


虽然这个矩形的宽度现在可以随着选择而改变了，但是位置还很奇怪：当第一个项目被选中时，它处于竖直和水平中央。几何读取器现在的尺寸是 HStack 的尺寸，但是我们的 Rectangle() 尺寸要小得多，而且默认情况下是居中的。要解决这个问题，我们可以添加一个和几何读取器的建议尺寸同样地 frame 尺寸，并使用 .bottomLeading 作为对齐方式。最后，我们可以添加 .animation(.default) 来为所有变更添加动画效果 (在下一章，我们会详细地研究动画)：

```
// ...
Rectangle()
  .fill(Color.blue)
  .frame(width: proxy[anchor!].width, height: 2)
  .offset(x: proxy[anchor!].minX)
```

```
.frame(
    width: proxy.size.width,
    height: proxy.size.height,
    alignment: .bottomLeading
)
.animation(.default)
```

现在，我们的下划线指示 view 能正确显示宽度和偏移了。

在这个特定例子中，对锚点的强制解包是安全的。总体来说，只有当至少有一个 tab 项目，或者更精确一些，只有在当存在有效的项目选择时，这么做才是安全的。在一个真的 tab bar 里，我们要么需要保证这一点，要么需要在强制解包之前，先检查确认锚点不是 nil 值。

到目前为止，我们已经使用了两种不同的方式，来在 view 层次中向上传递 view 的几何信息。在上面的示例中，我们使用了 .anchorPreference 来传播锚点。在上一节里，我们则是使用了几何读取器来获取 view 的尺寸，并将它直接设为一个 preference。

两种方法都可以达成同样的结果，但是每种都有它特有的取舍。锚点 preference (.anchorPreference) 的长处在于当我们想要把坐标从一个 view 变换到另一个 view 的坐标系统时，可以避免错误百出的手动 frame 的计算。但是，在将锚点值存储到状态数据时，这种做法就有些笨重，因为我们无法使用 onPreferenceChange 来观测锚点 preference，我们也不被允许在 overlayPreferenceValue 或者 backgroundPreferenceValue 里改变状态。

另一方面，如果我们只是用 preference 传递像 CGSize 或者 CGPoint 这样的简单值的话，我们可以在 onPreferenceChange 中观测它并将其存储在状态变量中。不过，SwiftUI 不会帮助我们将这些简单的几何值在不同坐标系统里进行转换。但是如果我们只是想要传递 view 的尺寸的话，我们并不需要在坐标空间之间转换，因此，我们会主要使用简单的 CGSize preference 值。

自定义布局

有了几何读取器，锚点和 preference，我们就拥有了创建更复杂的自定义布局的所有构建模块。作为例子，我们来构建一个自定义布局的容器 view：一个能将自身布局为水平或者竖直的 stack。因为我们想要将 stack 中的项目在水平和竖直布局之间进行动画切换，所以我们不能单纯地把 HStack 和 VStack 进行包装。作为替代，我们使用 ZStack 来自行布局子 view。要计算布局，我们使用几何读取器和 preference 系统来将子 view 们的尺寸收集到一个私有状态变量中。要确定子 view 们的位置，我们可以在水平和竖直的两个轴上对它们的对齐准线 (alignment guide) 进行重载。

第一步，我们要收集子 view 的尺寸，为此我们定义一个 preference key。Preference 的值是一个字典，它的 key 是子 view 的 index，值为它的尺寸。我们用一个空字典作为开始，并在 reduce 方法中将两个字典进行合并。因为在合并的过程中并不会有重复 key 的情况，所以合并中在处理重复 key 时，用哪个值都是可以的，我们这里使用 { $1 } 来始终选取第二个值：

```
struct CollectSizePreference: PreferenceKey {
  static let defaultValue: [Int: CGSize] = [:]
  static func reduce(value: inout Value, nextValue: () -> Value) {
    value.merge(nextValue(), uniquingKeysWith: { $1 })
  }
}
```

接下来，我们用这个 preference key 创建一个 view 修饰器，来将 view 的尺寸和 index 向上传递。我们使用和前面一样的技术，在背景中用一个几何读取器，并使用 Color.clear 来设置 preference：

```
struct CollectSize: ViewModifier {
  var index: Int
  func body(content: Content) -> some View {
    content.background(GeometryReader { proxy in
      Color.clear.preference(key: CollectSizePreference.self,
        value: [self.index: proxy.size])
    })
  }
}
```

现在我们手上有上面定义的 preference key 和 view 修饰器了，我们可以开始构建 stack view 本身了。SwiftUI 内建的 stack 接受一个 view builder，并会指出里面的内容具体是一个单独的 view，是一个 tuple view，还是一个 ForEach。我们没有办法很容易做到这一点，所以我们的 stack view 会接受元素的数组 (类似于 ForEach)，以及一个将元素转换为 view 的函数。我们也会为间距，排列轴和对齐添加参数。我们还要添加一个私有的 @State 变量来把计算出的偏移量存储为 CGPoint：

```
struct Stack<Element, Content: View>: View {
  var elements: [Element]
  var spacing: CGFloat = 8
  var axis: Axis = .horizontal
  var alignment: Alignment = .topLeading
  var content: (Element) -> Content
  @State private var offsets: [CGPoint] = []
}
```

现在我们需要基于子 view 们的尺寸，来计算它们的偏移量。这些尺寸是通过 preference 值进行传递的，我们通过 onPreferenceChange 来更新布局：

```
struct Stack<Element, Content: View>: View {
  // ...
  var body: some View {
    ZStack(alignment: alignment) {
      // ...
    }
    .onPreferenceChange(CollectSizePreference.self,
      perform: self.computeOffsets)
  }
}
```

在 computeOffsets 方法里，我们可以在同一轮里将所有子 view 的水平和竖直偏移量都计算出来。第一个 view 的偏移量为 .zero。对于接下来的 view，我们将最后的偏移量加上间距和这个 view 的宽或高，作为新的偏移量。注意，这里的实现需要 sizes 字典要么是全空 (在第一次布局过程中)，要么对所有元素都有值。如果这个条件不能满足，我们简单地让程序崩溃 (我们也可以更优雅地用一个警告来处理这种情况)：

```
private func computeOffsets(sizes: [Int: CGSize]) {
```

```
    guard !sizes.isEmpty else { return }

    var offsets: [CGPoint] = [.zero]
    for idx in 0..<self.elements.count {
      guard let size = sizes[idx] else { fatalError() }
      var newPoint = offsets.last!
      newPoint.x += size.width + self.spacing
      newPoint.y += size.height + self.spacing
      offsets.append(newPoint)
    }
    self.offsets = offsets
  }
```

要获取指定子 view 的偏移量值，我们实现了一个私有的辅助方法，它会在 offsets 数组里进行查找。在第一轮布局过程中，数字还没有被设置，所以这个辅助方法将返回 .zero：

```
  private func offset(at index: Int) -> CGPoint {
    guard index < offsets.endIndex else { return .zero }
    return offsets[index]
  }
```

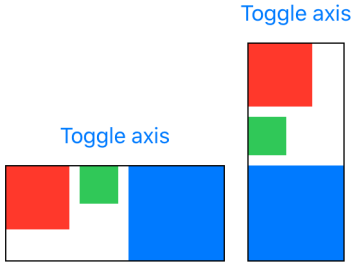
现在，我们可以为我们的 stack view 编写 body 了。我们用给定的对齐创建一个 ZStack，并循环访问 elements 数组的 index。对于每一个元素，我们使用 content 函数创建 view，然后在它上面调用我们的 CollectSize 修饰器来测量它的尺寸，并为每个轴添加对齐准线。对于水平轴，当 view 在水平模式时，我们使用我们所计算的水平偏移量；当 view 在竖直模式时，我们使用默认的对齐。对于竖直轴，对齐准线的计算逻辑正好反过来：

```
  var body: some View {
    ZStack(alignment: alignment) {
      ForEach(elements.indices, content: { idx in
        self.content(self.elements[idx])
          .modifier(CollectSize(index: idx))
          .alignmentGuide(self.alignment.horizontal, computeValue: {
            self.axis == .horizontal
              ? -self.offset(at: idx).x
              : $0[self.alignment.horizontal]
          })
          .alignmentGuide(self.alignment.vertical, computeValue: {
```

```
                self.axis == .vertical
                  ? -self.offset(at: idx).y
                  : $0[self.alignment.vertical]
            })
        })
    }
    .onPreferenceChange(CollectSizePreference.self, perform: self.computeOffsets)
}
```

下面是一个使用自定义 Stack view 的例子，它显示了三个不同尺寸的矩形。我们可以在水平和竖直布局之间流畅地进行动画。Stack 周围的黑色边框也表明，从水平布局切换到竖直布局时，它的尺寸也会发生变化：

```
struct ContentView: View {
    let colors: [(Color, CGFloat)] = [(.red, 50), (.green, 30), (.blue, 75)]
    @State var horizontal: Bool = true
    var body: some View {
        VStack {
            Button(action: {
                withAnimation(.default) {
                    self.horizontal.toggle()
                }
            }) { Text("Toggle axis") }
            Stack(elements: colors, axis: horizontal ? .horizontal : .vertical) { item in
                Rectangle()
                    .fill(item.0)
                    .frame(width: item.1, height: item.1)
            }
            .border(Color.black)
        }
    }
}
```



重点

- → 在任意 view 的背景中使用 GeometryReader 可以挂钩到布局流程中，并读取这个 view 的被建议尺寸。
- → 要将某个尺寸沿着 view 层级向上传递，可以通过 Color.clear.preference(...) 来设定一个 preference 值。
- → 使用 onPreferenceChange 可以观测到这个 preference 值，并基于它修改 view 的状态。
- → 要在一个 view 的坐标系统中读取另一个坐标系中的几何数据，可以通过 anchorPreference API 来设定锚点 preference。
- → 锚点 preference 可以通过 overlayPreferenceValue 或 backgroundPreferenceValue 方法进行读取，我们可以在其中创建一个几何读取器来把锚点解析到目标坐标系统中去。

练习

创建表格

在这个练习里，你需要创建一个 view 来展示表格数据，也就是那些有多行和多列的数据。每一列都应该将它的宽度适配为该列中最宽的那个项目。比如，下面是一个示例的表格：

	Monday	Tuesday	Wednesday
Berlin	Cloudy	Mostly Sunny	Sunny
London	Heavy Rain	Cloudy	Sunny

我们希望使用下面的代码来生成这样的表格：

```
struct ContentView: View {
  var cells = [
    [Text(""), Text("Monday").bold(), Text("Tuesday").bold(),
      Text("Wednesday").bold()],
    [Text("Berlin").bold(), Text("Cloudy"), Text("Mostly\nSunny"), Text("Sunny")],
    [Text("London").bold(), Text("Heavy Rain"), Text("Cloudy"), Text("Sunny")],
  ]
  var body: some View {
    Table(cells: cells)
      .font(Font.system(.body, design: .serif))
  }
}
```

注意每一列的内容需要对齐起始边，每一行的内容需要对齐顶边。

步骤 1：测量单元格

创建一个新的 preference key，用来存储每一列中的最大宽度。为 View 添加一个复制方法，用它读取 view 的尺寸，并将它用 WidthPreference 存储到指定的列中。

步骤 2：布局表格

使用 SwiftUI 内建的 stack 来布局单元格。将测量到的宽度应用到每个单元格上，基于测量到的最大宽度为单元格列设置合适的 frame。

附加练习

一旦表格可以工作后，你可以尝试让单元格变得可以点击。下图是上面表格中数据的第一行第一列的单元格被选中时的样子：

	Monday	Tuesday	Wednesday
Berlin	Cloudy	Mostly Sunny	Sunny
London	Heavy Rain	Cloudy	Sunny

在某个项目被点击选中时，为它添加一个边框相对起来很简单。但是，如果想要在选中项发生变化时，在两个单元格之间为这个表示选中的矩形添加动画，就需要一些额外努力了。我们建议你尝试去实现这两种不同的选中方式。

动画

6

从 iOS 系统一开始，优美的动画就是用户体验的关键要素。比如 scroll view 可以将它们的内容以流畅的动画进行展示，并且在达到尾部的时候也有一个反弹的动画效果。类似地，当你在首页上点击一个 app 图标时，app 将会从图标的位置以动画的方式展现。这些动画除了提供观赏性外，它们还向用户提示了操作的上下文。

SwiftUI 从一开始就内建了动画支持。你可以使用隐式动画，或者显式动画，你也可以完全以手动的方式掌控物件在屏幕上随时间移动的方式。首先，我们会来看一些例子，去了解我们能在基本的隐式动画时做些什么，接下来我们会研究深层的动画工作的方式。最后，我们会展示创建自定义动画所需要的技术。

隐式动画

隐式动画是 view 树的一部分：通过为 view 添加 .animation 修饰器，任何对这个 view 的改变都会自动进行动画。第一个例子，我们来创建一个圆角按钮，在点击时，它会改变自己的颜色和尺寸：

```
struct ContentView: View {
  @State var selected: Bool = false
  var body: some View {
    Button(action: { self.selected.toggle() }) {
      RoundedRectangle(cornerRadius: 10)
        .fill(selected ? Color.red : .green)
        .frame(width: selected ? 100 : 50, height: selected ? 100 : 50)
    }.animation(.default)
  }
}
```

当点击时，按钮的矩形边框会在 (50, 50) 和 (100, 100) 之间以流畅的动画进行切换，颜色也会从红色渐变到绿色。以这种方式所创建的动画感觉上有一点像 Apple 的 “Keynote 演讲” 程序里的神奇移动 (Magic Move)：我们定义初始点和结束点，程序来决定要怎么在两者之间进行动画。

SwiftUI 中的动画是 view 更新系统的一部分，我们在第二章中已经了解过这个更新系统了。和其他 view 更新的机制一样，动画也只会在状态改变时被触发 (在上面的例子中，是由 selected 属性的改变触发的)。当我们为 view 树添加一个像是 .animation(.default) 的修饰器时，SwiftUI 将在 view 更新的时候以动画的方式把旧的 view 树改变为新的 view 树。

由于动画是由状态变更所驱动的，因此某些动画需要我们发挥创造力。例如，如果我们想要构建一个类似 iOS 内建的活动指示器 (activity indicator) 那样的，一个图片一直旋转的 view，要怎么办？我们可以通过一些技巧来解决这个问题。首先，我们需要一些可以更改的状态以触发动画，这里我们使用一个布尔属性，当 view 出现时，我们将它设为 true。其次，我们需要通过在线性动画中加上 repeatForever，来让动画无限次重复。默认情况下，重复动画会在每次重复时反转播放自身，但这并不是我们想要的 (这样会导致指示器旋转一圈之后，反向进行旋转。而我们想要的是一直向同一个方向旋转)，所以我们要将 autoreverses 设置为 false：

```
struct LoadingIndicator: View {
  @State private var animating = false
  var body: some View {
    Image(systemName: "rays")
      .rotationEffect(animating ? Angle.degrees(360) : .zero)
      .animation(Animation
        .linear(duration: 2)
        .repeatForever(autoreverses: false)
      )
      .onAppear { self.animating = true }
  }
}
```

虽然上面的方案因为使用了 onAppear 和一个布尔状态属性，因此显得不太干净，但是我们至少可以把这些实现细节都隐藏起来。我们只需要 LoadingIndicator() 就能够使用这个加载指示器了。

过渡

到目前为止，我们所看到的动画都是将屏幕上的 view 从一个状态动画变为另一个状态。但是有时候，我们会想要以动画的方式插入一个新 view，或者是移除一个已经存在的 view。SwiftUI 为此准备了特定的方式，那就是过渡 (transition)。比如，下面是一个使用滑入滑出动画来将矩形移入和移出屏幕的过渡。当矩形被插入到 view 树中时，它从左侧动画滑入，当它被移出 view 树时，它向右侧动画滑出：

```
struct ContentView: View {
  @State var visible = false
  var body: some View {
```

```
    VStack {
      Button("Toggle") { self.visible.toggle() }
      if visible {
        Rectangle()
          .fill(Color.blue)
          .frame(width: 100, height: 100)
          .transition(.slide)
          .animation(.default)
      }
    }
  }
}
```

注意，过渡效果 (.transition 修饰器) 本身并不进行动画，我们依然需要为它们开启动画。和之前一样，我们使用 .animation(.default) 来达成这一点。我们也可以组合多个过渡效果。比如 AnyTransition.move(edge: .leading).combined(with: .opacity) 会将 view 从前边缘移入或移出，同时也执行一个淡入淡出的效果。要进一步自定义过渡效果，我们可以使用 .asymmetric，它可以让我们为 view 的插入指定一种过渡，同时为 view 的移出指定另一种过渡。

动画是如何工作的

我们继续考察圆角矩形的动画，这次我们只对它的宽度进行动画。我们也会将动画的持续时间改成五秒，并让它按照线性方式进行动画：

```
struct AnimatedButton: View {
  @State var selected: Bool = false
  var body: some View {
    Button(action: { self.selected.toggle() }) {
      RoundedRectangle(cornerRadius: 10)
        .fill(Color.green)
        .frame(width: selected ? 100 : 50, height: 50)
    }.animation(.linear(duration: 5))
  }
}
```

要让圆角矩形的 width 值从 50 动画变更到 100，SwiftUI 需要在这两个值之间进行插值。我们知道，动画需要五秒，且在这段时间内的任意时刻，width 的值和时间会是线性关系：在一开始，这个值是 50，在一秒后，它会是 60，在三秒后是 80，在五秒和之后的时间内，它会是 100。

在动画期间内，SwiftUI 对 width 值的计算分为两部分。首先，SwiftUI 使用我们指定的 Animation 值 (在上面的例子中，这个值是 .linear；我们会在接下来更详细地介绍动画曲线) 来计算动画的**进度**。通常，动画的进度是间于 0 和 1 之间的一个值，0 代表动画的开始，1 代表动画的结束 (不过，有一些动画可以带有 “弹性”，它们的进度值有可能会大于 1 或者小于 0)。

确定了动画的进度，SwiftUI 就可以对被变更的属性在起始值和终止值之间进行插值了。这些变更的属性值包括有本章一开始的例子中的填充色和尺寸，或者只是上面这个简化版本中的矩形宽度。

为了对值进行动画，SwiftUI 使用了 Animatable 协议，该协议仅具有一个要求：一个类型遵守 VectorArithmetic 协议的 animatableData 属性。VectorArithmetic 类型可以进行加法和减法运算，也可以使用 Double 进行乘法。通过这些运算，SwiftUI 可以得出动画的起始值和终止值之间的差值，并将它乘以当前的进度值来得到实际需要的插值。以矩形 width 动画从 50 到 100 为例，SwiftUI 通过 50 + (100 - 50) * progress 就可以计算出当前的宽度。或者，当从绿色动画到红色时，我们可以将它写作 .green + (.red - .green) * progress。一般而言，SwiftUI 将 animatableData 属性的当前值计算为 startValue + (endValue - startValue) * progress。

当为一个 view 的子树设定隐式动画 (比如 .animation(.default)) 时，子树上的所有可动画属性都会被加上动画，我们也可以通过调用 .animation(nil) 来在这棵子树上禁用所有动画 (甚至包括那些显式动画)。理论上，我们可以通过嵌套在不同 view 修饰器间的启用和禁用动画的调用，来为指定的子树打开和关闭动画。

我们强调 “理论上”，这是因为在实际中，在 view 树上某个节点去开启或者禁用动画到底会有什么效果，这一点并不明确。下面的例子是在本章开头的例子中添加一行 .animation(nil)：

```
struct ContentView: View {
  @State var selected: Bool = false
  var body: some View {
    Button(action: { self.selected.toggle() }) {
      RoundedRectangle(cornerRadius: 10)
        .fill(selected ? Color.red : .green)
```

```
            .animation(nil)
            .frame(width: selected ? 100 : 50, height: 50)
        }.animation(.linear(duration: 5))
    }
}
```

也许我们会期待 frame 会被动画，但颜色不进行动画 (因为它紧接了一个 .animation(nil) 的调用)。但实际发生的事情是 frame 和颜色两者都没有动画。frame 修饰器仅仅是将当前的状态所对应的正确的 frame 向下传递，但是它本身并不进行动画。实际的 frame 动画发生在圆角矩形形状那一层中，在那边我们已经禁用了动画。

让我们再看看在这个同样的例子中，但是在 frame 修饰器后面再加一个旋转效果：

```
Button(action: { self.selected.toggle() }) {
    RoundedRectangle(cornerRadius: 10)
        .fill(selected ? Color.red : .green)
        .animation(nil)
        .frame(width: selected ? 100 : 50, height: 50)
        .rotationEffect(Angle.degrees(selected ? 45 : 0))
}.animation(..linear(duration: 5))
```

和之前一样，frame 和颜色依然没有动画。但是旋转会流畅地在 0 度和 45 度之间进行动画。.rotationEffect 不仅仅只是把关于旋转的信息传递给形状，它自身也负责进行动画。

当尝试通过这种方式控制动画时，最后的行为依赖于不透明的实现细节，也很难被预测。因此，作为替代，我们强烈推荐使用显式的动画 (我们会在下面详细说明)。

动画曲线

在上面的例子中，我们指定了 .linear(duration: 5) 作为动画，也就是说动画将以常数速度运行五秒。换句话说，.linear(duration: 5) 这个 Animation 值提供了一条动画曲线，它在五秒钟的过程内将进度平均地从 0 到 1 进行插值。由于在普通速度下，线性动画通常看起来会不太自然，所以我们可以使用其他的内建动画曲线，比如通过指定 .easeInOut(duration: 0.35) 来使用缓入/缓出曲线 (ease-in/ease-out curve)，它将一开始会缓慢移动，然后加快速度，最后再降低速度。

动画曲线可以被看作是一个函数，它接受时间作为输入，返回在这个时间点上对应的动画所达到的进度。下面是内置的动画曲线进行可视化后的样子。横轴代表时间，纵轴代表动画进行的进度：

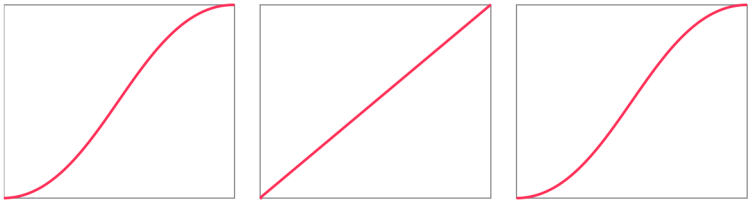


Figure 6.1: .default, .linear 和 .easeInOut

我们可以通过一些修饰器来改变 Animation 的值。比如，我们可以调用 .speed(0.1) 来让动画速度减慢到十分之一，或者我们可以调用 .delay(2) 来让动画延后两秒。另外，repeatCount 是一个很有意思的修饰器，它可以让我们重复一个动画若干次。比如说，这个动画会从绿色开始，然后在红色和绿色之间闪烁，最后变为红色：

```
Rectangle()
  .fill(selected ? Color.red : Color.green)
  .animation(Animation.default.repeatCount(3))
```

当重复动画时，第一次动画将正向播放，第二次则反向播放，以此类推。我们可以为 repeatCount 的 autoreverses 参数传递 false 来改变这个行为。对重复动画的动画曲线进行可视化，看起来是这样的：

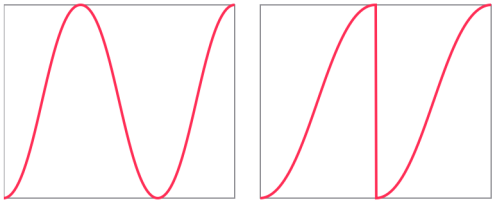


Figure 6.2: .default.repeatCount(3) 和 .default.repeatCount(2, autoreverses: false)

还有不少其他的内建动画曲线，其中最值得注意的是弹簧动画和 (通过控制点进行定义的) 自定义动画曲线。动画的进度是可以超出上限的，比如，对于一些弹簧动画。view 有可能从起始位置开始，然后超过终止位置，最后才停留在终止位置上。我们也可以在有一些动画曲线上看到这种情况：

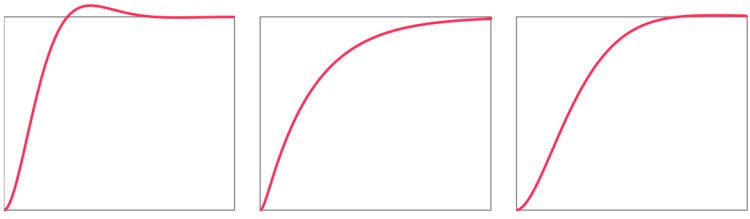


Figure 6.3: .interpolatingSpring, .interactiveSpring 和 .spring()

SwiftUI 在实现动画时一个有趣的特点是，增量动画是默认被支持的。当我们在动画过程中改变了状态时，新的动画会从前一个动画的当前状态开始。

显式动画

至今位置，我们所使用的隐式动画是定义在 view 的层级上的：我们通过在 view 上调用 .animation 来创建一个可以动画的 view。当我们创建一个隐式动画时，每当 view 树被重新计算时，view 树中那些可动画的属性的变更都会以动画呈现。虽然隐式动画十分方便，但是有时候这也会造成意料之外的效果。比如，想象一下下面这个另一版本的加载指示器，它将一个小原点沿着圆圈进行动画：

```
struct LoadingIndicator: View {
  @State var appeared = false
  let animation = Animation
    .linear(duration: 2)
    .repeatForever(autoreverses: false)
  var body: some View {
    Circle()
      .fill(Color.accentColor)
      .frame(width: 5, height: 5)
```

```
            .offset(y: -20)
            .rotationEffect(appeared ? Angle.degrees(360) : .zero)
            .animation(animation)
            .onAppear { self.appeared = true }
    }
}
```

当我们在 iOS 上运行上面的代码时，起初动画看起来是按预想工作的。但是当我们旋转模拟器 (或者设备) 时，这个点将会在一条奇怪的路径上运动：它显然不再在圆上进行动画。设备方向的改变造成了这个问题。因为我们使用了隐式动画，SwiftUI 也将会为点的 frame 改变进行这个永不停止的动画 (这个 frame 改变是由设备转向所造成的)。

解决这个问题的方法是使用显式动画，我们只想为那些 view 树中由于 self.appeared 状态属性变更所造成的变化进行动画。要做到这一点，我们将隐式动画移除，并将状态改变的代码包装到一个 withAnimation (显式动画) 调用中：

```
Circle()
  .fill(Color.accentColor)
  .frame(width: 5, height: 5)
  .offset(y: -20)
  .rotationEffect(appeared ? Angle.degrees(360) : .zero)
  .onAppear {
    withAnimation(self.animation) {
      self.appeared = true
    }
  }
```

现在，圆圈在设备转向时也可以保持正确的动画了。在和其他 SwiftUI 开发者聊起来时，我们注意到很多开发者会选择使用显式动画，因为隐式动画经常会造成像上面那样的意料之外的副作用。

自定义动画

通过创造性地使用内建的 view 修饰器，我们通常是可以构建出我们所想要的动画的。但是有些时候，实现自定义的可动画 view 修饰器是达成某些特定动画的唯一途径。

例如，我们想要实现一个可以摇晃 view 来吸引用户注意的动画。当动画开始时，view 应该从它的初始位置向右移动，然后向左移动到初始位置的左侧，最后在移动回到初始位置。我们无法直接通过 offset 修饰器来实现这个：如果为 offset(x: animating ? 20 : 0) 指定了一个重复动画。那么 view 会在动画最后结束时停留在初始位置右侧 20 point 的地方，因为这是动画被触发后 view 树的新状态。

幸运的是，要实现这个动画，我们并不需要依赖 offset 修饰器：我们可以实现我们自己的自定义 view 修饰器，并使它遵守 AnimatableModifier 协议 (这个协议继承自 Animatable 和 ViewModifier)。我们的修饰器有一个 times 属性，当它增加时，就将晃动 view。我们的目标是当这个属性被增加 1 时，晃动一次 view，当增加 2 时晃动两次，以此类推：

```
struct Shake: AnimatableModifier {
  var times: CGFloat = 0
  let amplitude: CGFloat = 10
  var animatableData: CGFloat {
    get { times }
    set { times = newValue }
  }
  func body(content: Content) -> some View {
    return content.offset(x: sin(times * .pi * 2) * amplitude)
  }
}
```

要满足 AnimatableModifier 的要求，我们需要实现 animatableData 属性，它负责简单地获取和设置 times 属性。它的类型，CGFloat，已经是一个满足 VectorArithmetic 的类型了。在 body 的实现中，我们使用 sin 来为摇晃的动画设置一个平滑的曲线：如果 times 是一个整数，偏移量为 0。在整数之间的数字 (比如说 0 到 1 之间)，偏移量将从 0 动画变更到 10，然后变更到 -10，最后回到 0。

> 如果需要暴露两个属性作为可动画属性，我们可以将它们包装在一个 AnimatablePair 里。想要支持任意数量的属性，我们可以嵌套使用 AnimatablePair。

为了能让我们的新动画更容易被发现，也具有更好的可读性，我们向 View 添加一个辅助方法：

```
extension View {
```

```
  func shake(times: Int) -> some View {
    return modifier(Shake(times: CGFloat(times)))
  }
}
```

要测试这个动画，我们创建一个每次点击后会晃动的按钮。我们可以把总点击次数 (从 0 开始) 保存在一个状态属性中，每次点击按钮时加一。接着我们把 Shake 修饰器加到按钮上，将点击数字乘以 3：

```
struct ContentView: View {
  @State private var taps: Int = 0

  var body: some View {
    Button("Hello") {
      withAnimation(.linear(duration: 0.5)) {
        self.taps += 1
      }
    }
    .shake(times: taps * 3)
  }
}
```

当我们运行代码时，一开始点击数字为 0，所以没有动画。当我们点击按钮时 taps 属性会从 0 变到 1，view 也会被渲染。这次，SwiftUI 会注意到在新的 view 树中，Shake 修饰器的 animatableData 从 0 变到了 3，它会对这个变化进行动画处理。我们使用线性动画曲线，这样摇晃运动的正弦曲线不会失真。

> 要验证 SwiftUI 在动画期间插入不同值的方式，我们可以在 animatableData 的 setter 或者 AnimatableModifier 的 body 中打印 log。

当我们的动画修饰器需要进行动画的值可以被表示为仿射变换 (不论 2D 还是 3D) 时，我们也可以使用 GeometryEffect 来代替 AnimatableModifier。例如，这里是同样的修饰器，只不过使用了 GeometryEffect 来重写：

```
struct ShakeEffect: GeometryEffect {
  var times: CGFloat = 0
```

```
    let amplitude: CGFloat = 10
    var animatableData: CGFloat {
      get { times }
      set { times = newValue }
    }

    func effectValue(size: CGSize) -> ProjectionTransform {
      ProjectionTransform(CGAffineTransform(
        translationX: sin(times * .pi * 2) * amplitude,
        y: 0
      ))
    }
}
```

不幸的是，SwiftUI 中并不存在 AnimatableView 协议 (而且让我们的 View 遵守 Animatable 也不会起作用)。想要编写自定义的 view 动画，我们所知道的唯一途径是通过 AnimatableModifier 协议，或者 GeometryEffect 这个更特殊化的版本。不过，Shape 是**可以**动画的：我们可以对像是 RoundedRectangle 上的圆角半径，或者是我们自定义形状的属性添加动画。

> 在最近的 Xcode 版本中，AnimatableModifier 变成了某些 bug 的原因。关于你可能会遇到的问题的更多细节，请阅读 Advanced SwiftUI Animations - Part 3 文中的 **Dancing with Versions** 部分。

自定义过渡

要自定义 view 的插入和移除动画，我们可以使用 AnyTransition 的 modifier(active:identity:) 方法来创建自定义过渡。这个方法接受两个 view 修饰器：一个用于过渡发生时，另一个用于过渡结束时。当 view 被插入时，active 修饰器在插入开始时被应用到 view 上，然后向着 identity 修饰器所定义的状态进行迁移。在移除过程中，动画反向进行：从 identity 修饰器开始，向 active 修饰器靠拢。作为例子，我们来创建一个模糊过滤。当 view 被插入时，它会以模糊状态出现，并且 opacity 值为零 (全透明)；当移除时，它会淡出并模糊。

第一步，我们创建一个自定义的 ViewModifier。当 active 是 true 时，内容是模糊且透明的，当 active 为 false 时，模糊半径被设为 0，内容非透明：

```
struct Blur: ViewModifier {
  var active: Bool
  func body(content: Content) -> some View {
    return content
      .blur(radius: active ? 50 : 0)
      .opacity(active ? 0 : 1)
  }
}
```

第二步，我们可以向 AnyTransition 添加一个静态属性，这可以帮助我们更容易访问到我们定义的新过渡：

```
extension AnyTransition {
  static var blur: AnyTransition {
    .modifier(active: Blur(active: true),
          identity: Blur(active: false))
  }
}
```

最后，我们可以使用上面的例子，将 .transition(.slide) 替换成 .transition(.blur)。现在，在插入时，view 会从模糊状态淡入；而在它被移除时，会淡出到模糊状态。

重点

- → 动画和 view 更新一样，是由状态变更触发的。
- → 隐式动画 (.animation(...)) 会对 view 子树中所有可动画属性的变更进行动画。
- → 使用显式动画 (withAnimation { ... }) 可以让我们有更多的控制权，并避免不想要的副作用。
- → 使用过渡来完成 view 插入和移除的动画。我们可以组合多个过渡，为插入和移除使用不同的过渡，或者是创建自定义过渡。
- → 要构建自定动画，我们需要实现一个可动画的 view 修饰器 (遵守 AnimatableModifier 协议) 或者实现一个 GeometryEffect，并将可动画的属性作为 animatableData 暴露出来。

练习

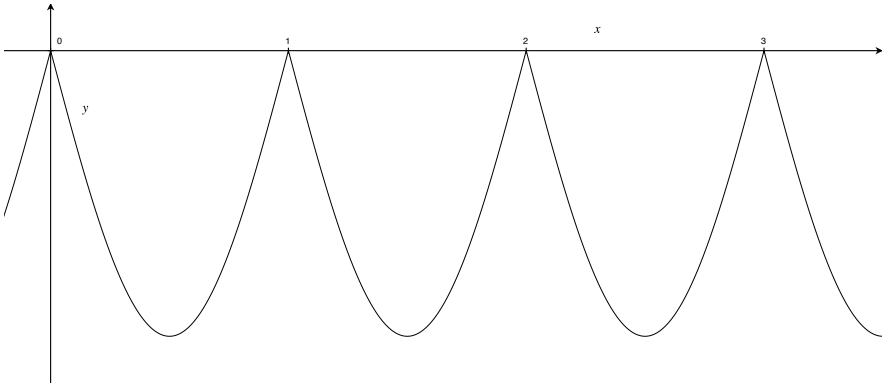
反弹动画

和上面的摇晃动画一样，反弹 (bounce) 动画也被归类到 “起始点和终止点相等” 的一类动画中。对于两者，我们都不能通过重复内建的动画来实现效果。你可以使用上面摇晃动画的代码作为起点，来构建一个反弹动画。在进行一些重命名后，反弹动画修饰器的框架如下：

```
struct Bounce: AnimatableModifier {
  var times: CGFloat = 0
  var amplitude: CGFloat = 10
  var animatableData: CGFloat {
    get { times }
    set { times = newValue }
  }
  func body(content: Content) -> some View {
    // ...
  }
}
```

步骤 1：实现动画曲线

实现 body 方法，来让每次 times 增加时 view 能以反弹的方式进行动画。一次反弹包括 view 从原点向上 “跳起”，然后再落下回到原点的过程。下面是动画曲线粗略看上去的样子：



步骤 2：View 扩展

将反弹次数作为参数，通过 View 的扩展将这个新的反弹动画公开出来。

附加练习

→ 为反弹添加一些 “弹性”，比如，被动画的 view 在落到 “地面” 后应该再以振幅衰减的方式再反弹几次。你可以让这个动画的参数都可配置。

→ 使用反弹修饰器来创建一个自定义的反弹过渡。

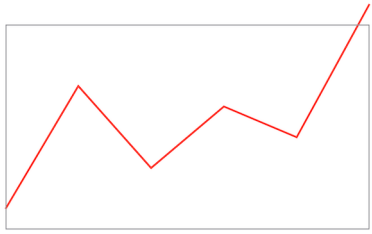
路径动画

在这个练习中，你需要基于一些数据点来绘制折线图。折线图应该被绘制为 Shape，这样它才能将自己适配到任意的建议尺寸中去。图形接受一个 CGFloat 数组作为输入，里面的数字从 0 到 1。范围外的数字将被绘制在边框之外。我们希望 API 的使用方式如下：

```
let sampleData: [CGFloat] = [0.1, 0.7, 0.3, 0.6, 0.45, 1.1]

LineGraph(dataPoints: sampleData)
  .stroke(Color.red, lineWidth: 2)
  .border(Color.gray, width: 1)
```

它应该绘制这样的图形：

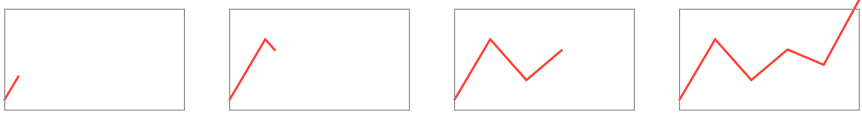


步骤 1：实现 Shape

第一步，我们需要实现一个可以绘制这张图的形状。

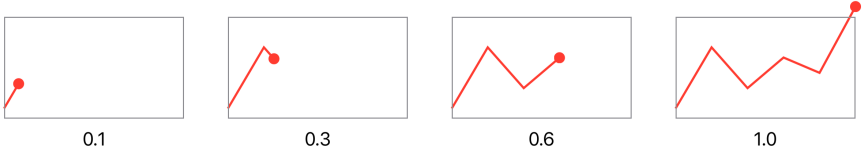
步骤 2：Shape 动画

对这个形状的路径进行动画，让它从一个空路径动画到整条路径。比如，下面是上图中的折线在动画不同阶段的绘制情况：



附加练习：添加引导点

在图中添加一个引导点，来指示路径顶端的移动方式：



要实现这个，你可以创建一个叫做 PositionOnShapeEffect 的 GeometryEffect，它会将 view (也就是这个引导点) 按照特定的偏移量在 Path 上移动。比如，当 offset 偏移量为 1 时，它位于

路径末尾，当 offset 为 0 时，它在路径的起始位置。将它和一个 View 上的自定的扩展组合起来，以确保你不仅能 Path 上，也能在 (可以填满建议宽度的) Shape 上使用它。从调用侧看，这个点的代码应该是下面这样：

```
Circle()
  .fill(Color.red)
  .frame(width: 10, height: 10)
  .position(on: LineGraph(dataPoints: sampleData), at: visible ? 1 : 0)
```

总结

我们希望这本书能让你对 SwiftUI 的开发方式有一个很好的总览性概括。如果你对此还不确信，我们十分推荐你回去把所有的练习都做完。

这本书不是一个完整的参考，即使读完本书，SwiftUI 中也还有很多的概念等待你去探索。比如，有很多平台相关的话题，像是 scroll view 的实现和使用，或者 AppKit 和 UIKit 的集成等，我们都没有涉及。我们期待这些部分能在 SwiftUI 的未来几个版本中有大幅的进化。

同样，对于 Combine 框架，我们也只讨论了最基础的核心部分。因为 SwiftUI 所集成的 model 对象都是构建在 Combine 上的，所以接下来研究 Combine 框架会是一个很好的选择。

我们希望能随着平台的演进，来不断更新本书。

如果你想进一步找一些阅读材料，我们推荐下面这些资源：

- → https://swiftui-lab.com/
- → https://netsplit.com/category/swiftui/
- → https://troz.net/post/2019/swiftui-for-mac-1/
- → https://www.bigmountainstudio.com/swiftui-views-book
- → https://www.hackingwithswift.com/quick-start/swiftui/

Florian 和 Chris

2020 年 3 月

习题答案

习题的答案也能在 GitHub 上找到。

第 2 章：View 更新

步骤 1: 加载元数据

Photo 结构体中的属性定义匹配 JSON API 中的字段的话，它就能自动遵守 Codable 协议了：

```
struct Photo: Codable, Identifiable {
  var id: String
  var author: String
  var width, height: Int
  var url, download_url: URL
}
```

步骤 2：创建 ObservableObject

下面是我们对于 Remote 的实现，它把加载的数据存储为一个可选的 Result 值：

- → 当 value 是 nil 时，表示还没有被加载。
- → 当 value 是 .failure 时，表示加载过程中出现了错误。
- → 当 value 是 .success 时，表示加载完成了。

我们需要把这个存储标记为 @Published，这样 SwiftUI 就会在这个属性改变时触发 objectWillChange。为了简单起见，我们还公开了一个计算属性 value，它将把错误状态和未加载状态合二为一：

```
final class Remote<A>: ObservableObject {
  @Published var result: Result<A, Error>? = nil // nil means not loaded yet
  var value: A? { try? result?.get() }

  let url: URL
  let transform: (Data) -> A?

  init(url: URL, transform: @escaping (Data) -> A?) {
```

```
        self.url = url
        self.transform = transform
    }

    func load() {
        URLSession.shared.dataTask(with: url) { data, _, _ in
            DispatchQueue.main.async {
                if let d = data, let v = self.transform(d) {
                    self.result = .success(v)
                } else {
                    self.result = .failure(LoadingError())
                }
            }
        }.resume()
    }
}
```

注意，在设置值之前，我们需要切换到主线程。SwiftUI 希望所有状态改变都发生在主线程上。

步骤 3：显示列表

首先，我们为 ContentView 添加一个 @ObservedObject 属性。如果你忘记了将属性标记为 @ObservedObject 的话，代码依然能够编译，但是 SwiftUI 将不能观察到任何变更，你的界面也不会更新。

第二步是使用 if 条件来检查数据是否完成了加载。如果数据已经可用了，那么我们将它显示在一个 List 中；否则，我们显示一个 Text。当 Text 首次出现在屏幕上时，我们开始加载数据。这保证了数据只在需要的时候被加载，而不是 view 被创建时立即加载：

```
struct ContentView: View {
    @ObservedObject var items = Remote(
        url: URL(string: "https://picsum.photos/v2/list")!,
        transform: { try? JSONDecoder().decode([Photo].self, from: $0) }
    )

    var body: some View {
        NavigationView {
            if items.value == nil {
```

```
            Text("Loading...")
              .onAppear { self.items.load() }
          } else {
            List {
              ForEach(items.value!) { photo in
                Text(photo.author)
              }
            }
          }
        }
      }
    }
```

步骤 4：显示图片

第一步，我们将每个列表项目中的 Text 包装到 NavigationLink 中：

```
ForEach(items.value!) { photo in
  NavigationLink(destination: PhotoView(photo.download_url), label: {
    Text(photo.author)
  })
}
```

PhotoView 使用下载 URL 来创建一个 Remote 值，当这个 view 初次出现时，它会开始加载图片。当图片加载完成后，Remote 值将会发送 objectWillChange 事件，然后 view 将渲染图片：

```
struct PhotoView: View {
  @ObservedObject var image: Remote<UIImage>

  init(_ url: URL) {
    image = Remote(url: url, transform: { UIImage(data: $0) })
  }

  var body: some View {
    Group {
      if image.value == nil {
        Text("Loading...")
          .onAppear { self.image.load() }
```

```
        } else {
          Image(uiImage: image.value!)
            .resizable()
            .aspectRatio(image.value!.size, contentMode: .fit)
        }
      }
    }
}
```

第 3 章：环境

配置旋钮颜色

步骤 1：创建 Environment Key

我们创建一个名为 ColorKey 的新类型，它满足 EnvironmentKey 协议：

```
fileprivate struct ColorKey: EnvironmentKey {
  static let defaultValue: Color? = nil
}
```

注意我们使用了可选的 Color。如果环境中的颜色值为 nil 的话，旋钮 view 将使用当前的配色方案下的默认颜色。为了为这个 key 提供一个键路径，我们还需要为 EnvironmentValues 添加一个 knobColor 属性：

```
extension EnvironmentValues {
  var knobColor: Color? {
    get { self[ColorKey.self] }
    set { self[ColorKey.self] = newValue }
  }
}
```

步骤 2：创建 @Environment 属性

在 Knob view 中，我们创建一个 fillColor 属性，它将检查环境中是否存在颜色值，如果不存在，它将会使用默认颜色作为替换：

```
struct Knob: View {
  // ...
  @Environment(\.knobColor) var envColor

  private var fillColor: Color {
    envColor ?? (colorScheme == .dark ? Color.white : Color.black)
  }

  var body: some View {
    KnobShape(pointerSize: pointerSize ?? envPointerSize)
      .fill(fillColor)
      // ...
  }
}
```

步骤 3：通过滑条控制颜色

在 ContentView 里，我们创建一个开关，来代表是否要使用默认颜色。我们还需要一个滑条，来控制自定义颜色的色调。然后我们通过在 View 里自定义的 knobColor 方法，来通过环境为旋钮提供颜色值：

```
struct ContentView: View {
  // ...
  @State var useDefaultColor = true
  @State var hue: Double = 0

  var body: some View {
    VStack {
      Knob(value: $value)
        .frame(width: 100, height: 100)
        .knobPointerSize(knobSize)
        .knobColor(useDefaultColor
          ? nil
```

```
            : Color(hue: hue, saturation: 1, brightness: 1)
        )
      // ...
      HStack {
        Text("Color")
        Slider(value: $hue, in: 0...1)
      }
      Toggle(isOn: $useDefaultColor) {
        Text("Default Color")
      }
      // ...
    }
  }
}
```

第 4 章：布局

可折叠的 HStack

为了创建可折叠的 HStack，我们使用一个常规的 HStack，并为每个子 view 配置特定的 frame 修饰器。当 stack 处于展开状态时，我们将子 view 的宽度设为 nil (也就是说，我们不去干涉它的宽度)。当 stack 处于折叠状态时，我们为除了最后一个子 view 外的每个子 view 设定固定的 frame 宽度 (设置为 collapsedWidth)。另外，我们还要将水平对齐显式地设为 .leading。否则，子 view 们将默认地在它们的 frame 中居中对齐：

```
struct Collapsible<Element, Content: View>: View {
  var data: [Element]
  var expanded: Bool = false
  var spacing: CGFloat? = 8
  var alignment: VerticalAlignment = .center
  var collapsedWidth: CGFloat = 10
  var content: (Element) -> Content

  func child(at index: Int) -> some View {
    let showExpanded = expanded || index == self.data.endIndex - 1
    return content(data[index])
```

```
        .frame(width: showExpanded ? nil : collapsedWidth,
            alignment: Alignment(horizontal: .leading, vertical: alignment))
    }

    var body: some View {
      HStack(alignment: alignment, spacing: expanded ? spacing : 0) {
        ForEach(data.indices, content: { self.child(at: $0) })
      }
    }
}
```

角标 View

作为开始，我们先来创建角标 view 本身，而不涉及像是位置这样的话题。角标是由 ZStack 里的一个圆圈和一个位于圆圈上方的文本标签组成 (我们也可以将文本作为圆圈的 overlay 叠层来达到同样的效果)。我们将两个 view 都用 if 条件包装起来，这样当角标数为 0 时，可以隐藏它们：

```
ZStack {
  if count != 0 {
  Circle()
    .fill(Color.red)
  Text("\(count)")
    .foregroundColor(.white)
    .font(.caption)
  }
}
```

要确定角标的位置，我们将 ZStack 包装到一个 offset 和一个 frame 修饰器中。这棵子树随后被放到 overlay 修饰器中，并以 .topTrailing 作为对齐方式进行定位。frame 为角标设置了固定的尺寸。最后，为了让角标的中心和下方 view 的角落重合，我们用它的一半的尺寸进行偏移：

```
extension View {
  func badge(count: Int) -> some View {
    overlay(
      ZStack {
        if count != 0 {
          Circle()
```

```
                .fill(Color.red)
              Text("\(count)")
                .foregroundColor(.white)
                .font(.caption)
            }
          }
          .offset(x: 12, y: -12)
          .frame(width: 24, height: 24)
      , alignment: .topTrailing)
  }
}
```

第 5 章：自定义布局

创建表格

步骤 1：测量单元格

我们创建一个 WidthPreference 的 PreferenceKey，用来存储一个由列的索引映射到它们的最大宽度的字典。因为我们不需要保存同一列中所有单元格的宽度，我们可以在 reduce 方法中从两个字典的对应 key 中获取最大值，来进行合并：

```
struct WidthPreference: PreferenceKey {
  static let defaultValue: [Int:CGFloat] = [:]
  static func reduce(value: inout Value, nextValue: () -> Value) {
    value.merge(nextValue(), uniquingKeysWith: max)
  }
}
```

要测量一个单元格的宽度，我们在 View 上创建一个辅助方法，它接受列的索引序号，并将 view 的尺寸作为 preference 存储到上面的 key 里：

```
extension View {
  func widthPreference(column: Int) -> some View {
    background(GeometryReader { proxy in
      Color.clear.preference(key: WidthPreference.self,
```

```
            value: [column: proxy.size.width])
        })
    }
}
```

步骤 2：布局表格

表格本身是位于 HStack 中的 VStack。对每个单元格，我们使用 widthPreference 辅助方法来测量宽度，并使用这个宽度来设置 frame。在第一轮布局中，columnWidths 字典为空，因此 frame 修饰器接收到 nil 作为宽度。在第二轮中，单元格的宽度被传递下去，每一列的实际宽度将在布局时被使用：

```
struct Table<Cell: View>: View {
    var cells: [[Cell]]
    let padding: CGFloat = 5
    @State private var columnWidths: [Int: CGFloat] = [:]

    func cellFor(row: Int, column: Int) -> some View {
        cells[row][column]
            .widthPreference(column: column)
            .frame(width: columnWidths[column], alignment: .leading)
            .padding(padding)
    }

    var body: some View {
        VStack(alignment: .leading) {
            ForEach(cells.indices) { row in
                HStack(alignment: .top) {
                    ForEach(self.cells[row].indices) { column in
                        self.cellFor(row: row, column: column)
                    }
                }
                .background(row.isMultiple(of: 2) ?
                    Color(.secondarySystemBackground) : Color(.systemBackground)
                )
            }
        }
        .onPreferenceChange(WidthPreference.self) { self.columnWidths = $0 }
```

```
  }
}
```

附加练习

附加练习要求在单元格选中时，添加一个可以在单元格间移动的矩形。这个附加练习十分困难，要解决这个问题，关键在于我们不应该依照单元格的边框 (border) 来绘制矩形，而应该在 stack 之外来单独绘制矩形。

第一步，我们不仅需要传递单元格的宽度，也需要传递单元格的高度。接下来，我们将这些宽度和高度设置给每个单元格。调整后的 cellFor 方法是这样的：

```
func cellFor(row: Int, column: Int) -> some View {
  cells[row][column]
    .sizePreference(row: row, column: column)
    .frame(width: columnWidths[column], height: columnHeights[row],
      alignment: .topLeading)
    .padding(padding)
}
```

sizePreference 为 preference value 设置宽和高。

另外，我们需要创建一个新的 preference，它包含有当前选中单元格的 bounds 锚点，这样，我们就可以知道要在什么地方绘制这个代表选中的矩形了：

```
self.cellFor(row: row, column: column)
  .anchorPreference(key: SelectionPreference.self, value: .bounds, transform: {
    self.isSelected(row: row, column: column) ? $0 : nil
  })
  // ...
```

要知道到底哪个单元格被选中，我们可以为每个单元格添加点击手势，并在被选中时将 (row, column) 值设给状态属性：

```
self.cellFor(row: row, column: column)
  // ...
  .onTapGesture {
```

```
      withAnimation(.default) {
        self.selection = (row: row, column: column)
      }
    }
```

最后，我们在外层 VStack 上添加一个 overlayPreferenceValue，用它来读取选中单元格的 bound 锚点，我们还需要使用 GeometryReader 来将这个锚点进行解析，并绘制矩形：

```
struct Table<Cell: View>: View {
  // ...
  var body: some View {
    VStack(alignment: .leading) { /* ... */ }
    .overlayPreferenceValue(SelectionPreference.self) {
      SelectionRectangle(anchor: $0)
    }
  }
}

struct SelectionRectangle: View {
  let anchor: Anchor<CGRect>?
  var body: some View {
    GeometryReader { proxy in
      ifLet(self.anchor.map { proxy[$0] }) { rect in
        Rectangle()
          .fill(Color.clear)
          .border(Color.blue, width: 2)
          .offset(x: rect.minX, y: rect.minY)
          .frame(width: rect.width, height: rect.height)
      }
    }
  }
}
```

ifLet 辅助方法将使用 if 语句来检查 nil，并将锚点可选值的强制解包抽象出来。你可以在 GitHub 上找到这个解答的完成源码。

第 6 章：动画

反弹动画

步骤 1：实现动画曲线

对于反弹动画，我们可以参照这一章中的摇晃动画，使用正弦曲线作为动画曲线。不过，我们采用 sin 函数的绝对值的相反数，因为我们想要的效果是让 view 能 “弹跳” 起来：

```
struct Bounce: AnimatableModifier {
  var times: CGFloat = 0
  let amplitude: CGFloat = 30
  var animatableData: CGFloat {
    get { times }
    set { times = newValue }
  }
  func body(content: Content) -> some View {
    return content.offset(y: -abs(sin(times * .pi)) * amplitude)
  }
}
```

步骤 2：View 扩展

View 扩展只是一个简单的 modifier 调用：

```
extension View {
  func bounce(times: Int) -> some View {
    return modifier(Bounce(times: CGFloat(times)))
  }
}
```

附加练习

要添加衰减的反弹移动，计算 view 的 y 坐标的方程需要稍微进行一些修改。你可以参照阻尼正弦波或者谐振的相关页面来获取一些启示。

路径动画

步骤 1：实现 Shape

我们的实现只针对那些至少有两个数据点的图形。首先，我们将路径移动到第一个数据点，然后我们为所有其他数据点添加直线。在 SwiftUI 中，y 轴的顶部为 0，但在折线图里，一般 y 轴的底部会被绘制为 0 点，所以我们不直接使用 point，而是通过 (1-point) 将坐标系统进行翻转。接下来，x 和 y坐标将基于形状的宽度和高度进行缩放：

```
struct LineGraph: Shape {
  var dataPoints: [CGFloat]
  func path(in rect: CGRect) -> Path {
    Path { p in
      guard dataPoints.count > 1 else { return }
      let start = dataPoints[0]
      p.move(to: CGPoint(x: 0, y: (1-start) * rect.height))
      for (offset, point) in dataPoints.enumerated() {
        let x = rect.width * CGFloat(offset) / CGFloat(dataPoints.count - 1)
        let y = (1-point) * rect.height
        p.addLine(to: CGPoint(x: x, y: y))
      }
    }
  }
}
```

步骤 2：Shape 动画

要对形状的路径进行动画，我们可以在 Shape 上使用 trim 修饰器。这个修饰器接受 to 和 from 参数，它们都是 0 到 1 之间的 Double 值。trim 会按照这两个参数为我们返回形状的部分内容。to 和 from 都是可以动画的，这就是说，对折线图进行动画可以通过使用 trim 修饰器来实现：

```
struct ContentView: View {
  @State var visible = false
  var body: some View {
    VStack {
      LineGraph(dataPoints: sampleData)
```

```
            .trim(from: 0, to: visible ? 1 : 0)
            .stroke(Color.red, lineWidth: 2)
            .aspectRatio(16/9, contentMode: .fit)
            .border(Color.gray, width: 1)
            .padding()
          Button(action: {
            withAnimation(Animation.easeInOut(duration: 2)) {
              self.on.toggle()
            }
          }) { Text("Animate") }
        }
      }
    }
```

附加练习

我们的解决方案是在 View 上添加一个叫做 position(on:at:) 的方法，它接受一个 Shape 和一个数量值 amount，并通过 GeometryReader 进行实现 (就像形状一样，这会成为它的被建议尺寸)。几何读取器会使用它的尺寸，来将 Shape 转换为 Path，之后它会应用这个自定义的 PositionOnShapeEffect(它是一个 GeometryEffect)。该 effect 将 amount 暴露为 animatable data，这样路径上的这个位置就可以进行动画了。

我们建议你参考 GitHub 上的源码实现，来加强理解。