

objc.io | objc 中国

SwiftUI 编程思想

针对 iOS 17 更新

Chris Eidhof, Florian Kugler 著
王巍 译

1 介绍 4

2 视图树 7

视图构建器 9

渲染树 17

身份标识 19

3 状态和绑定 26

State 28

Observable 宏 36

ObservableObject 协议 46

Binding 57

视图更新和性能 63

属性包装器及其用途 65

4 布局 67

叶子视图 71

视图修饰器 77

容器视图 88

对齐 98

5 环境 108

从环境中读取 110

自定义环境键 112

自定义组件样式 116

EnvironmentObject 122

6 动画 125

控制动画 129

动画协议 138

过渡 143

7 进阶布局 156

Layout 协议 157

基于首选项的布局 162

可变数量视图 170

坐标空间 171

锚点 173

几何匹配效果 175

介绍

1

当 SwiftUI 问世时，它带来了一种和 UIKit 截然不同的开发方式。我们之前撰写了本书的第一版，用来帮助你建立对 SwiftUI 工作方式的心智模型。几年过去了，我们有机会举办了一些研讨会，来向许多规模或大或小的开发团队讲授这些材料。在这个过程中，我们根据研讨会的反馈不断改进和完善了解释 SwiftUI 基本原理的方法。这本全新的《SwiftUI 编程思想》就是这趟旅程的结果：我们从头重新编写了整本书，让它的内容和我们在研讨会中教授 SwiftUI 的方式保持一致。

虽然 Apple 的 SwiftUI API 文档在这几年中有很大的进步，但我们仍然认为需要更多的概念性的文档来解释 SwiftUI 的工作原理。和第一版一样，这仍然是本书所关注的重点。我们希望能够帮助你建立对 SwiftUI 概念的牢固理解，这样你就可以自行学习那些不断扩展的平台特定的 API 了。

我们认为，高效使用 SwiftUI 的关键是要了解我们编写的代码是如何转化为视图树的。我们在第一章中就会详细介绍这方面的内容，之后我们会继续讨论这些视图树在状态、布局、动画以及更多其他方面的处理方式。

为了进行更好的解释，在这一版书中，我们增加了更多的视觉要素，这部分要归功于我们使用了全新的书籍生成工具。我们把原来基于 LaTeX 的工作流程转变为了基于纯 Swift 和 TextKit 的工具，这让我们能够直接将 SwiftUI 的视图嵌入到书中。除了简化我们的工具链，这还让我们能够生成许多图表、插图和视图预览，希望这些内容能帮助解释原先抽象的概念。

在我们更新本版书籍的同时，WWDC 23 也已经举行，苹果宣布了一系列新的 SwiftUI API。贯穿本书，我们为许多新 API 都添加了解释。我们也特意在涉及到这些新 API 时说明了它只适用于 iOS 17 (以及同时发布的其他平台，比如 macOS 14)。

从我们举办的许多研讨会中多次观察到的结果表明，学习 SwiftUI 的最佳方法就是亲自编写代码。这本书无法取代这一点，但它可以成为你学习路上的好伙伴。我们鼓励你经常将从本书中学到的知识付诸实践。想让知识更加牢固，最好的方法就是通过自己动手实验，并观察事情的工作方式。

我们想感谢在本书写作过程中帮助过我们的所有人。感谢 Natalye 进行校对，感谢 Ole 进行技术审查，感谢 Marcin 帮助我们处理 TextKit 的相关部分。我们还要

感谢 Robb、Ole 和 Juul 帮助我们改进研讨会，进而改进了这本书。我们也要感谢之前阅读过我们旧版书籍和参加过我们研讨会的热心读者们，感谢你们提供的所有反馈。最后，非常感谢 SwiftUI 的创作者们。

来自 Florian 和 Chris

视图树

2

视图树 (view tree) 和渲染树 (render tree) 也许是使用 SwiftUI 时，所需要了解的最基本和最重要的概念。想要实现所需要的布局，我们必须理解视图树的构建方式。想要理解状态在 SwiftUI 中的工作方式，我们必须理解视图的生命周期，并理清它与视图树之间的关系。此外，理解生命周期对于编写高效的 SwiftUI 代码同样重要，理想的代码应该只在需要的时候才加载数据和更新视图。最后，想要理解动画和过渡效果，也需要对视图树有深入的认识。

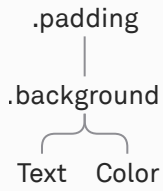

来考虑下面这个视图：

Code	View Tree	Preview
<pre>Text("Hello") .padding() .background(Color.blue)</pre>	<pre>graph TD A[.background] --- B[.padding] A --- C[Color] B --- D[Text]</pre>	

针对右边的代码，我们可以看到它对应的视图树。`.background` 修饰器 (modifier) 位于视图树的根部位置。它的主要子视图是一个带有 `padding` 填充的文本，`.background` 就被添加到了这个视图上，文字则被绘制到背景的上方。这个背景还有一个次要的蓝色子视图 `Color`，它被绘制在主要子视图的背后。每当我们在文本上使用像是 `padding` 或者 `background` 这样的视图修饰器时，它都会被包装到另外的层级中去。当我们处理上面例子中这样的视图修饰器链的代码时，我们必须从下向上看，才能得到正确的视图树表示；在这个例子中，最后一个视图修饰器是 `background`，它变成了视图树中最上层的视图。

请注意，`background` 视图修饰器本身并不绘制任何东西。即便 `background` 修饰器是视图树中最顶层的视图，实际的蓝色背景仍然被绘制在文本的后方。

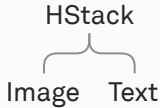

下面是一个稍有不同的例子，padding 和 background 互换了一下位置：

Code	View Tree	Preview
<pre>Text("Hello") .background(Color.blue) .padding()</pre>	 <pre>graph TD A[.padding] --> B[.background] B --> C[Text] B --> D[Color]</pre>	

背景现在是文本的直接父级，而填充 (padding) 是背景的父级。在[布局篇](#)中，我们会详细介绍为什么最终视图的布局产生了变化。现在一言以蔽之，布局之所以发生改变，是因为我们构建了一个不同的视图树。

视图构建器

SwiftUI 使用一种特殊的**视图构建器** (view builder) 语法来构建视图的列表。视图构建器是建立在 Swift 的结果构建器 (result builder) 特性之上的，这个特性是专门为了构建视图才被添加到语言中的。例如，想要构建一个在文本旁边显示图片的视图时，我们可以这样做：

Code	View Tree	Preview
<pre>HStack { Image(systemName: "hand.wave") Text("Hello") }</pre>	 <pre>graph TD A[HStack] --> B[Image] A --> C[Text]</pre>	

HStack 的初始化方法接受一个闭包作为参数，这个闭包被标记为了 @ViewBuilder。该特性允许我们在闭包里书写多个代表视图的表达式。从本质来说，传递给 HStack 的闭包将构建出一个视图的**列表**，在此例中，这些视图将会

成为 HStack 的子视图。

查看 ViewBuilder 结构体的声明，你会发现下面这个处理两个视图的方法：

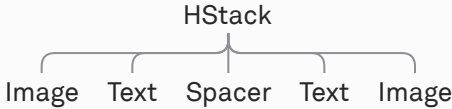
```
extension ViewBuilder {  
    public static func buildBlock<C0, C1>(_ c0: C0, _ c1: C1) ->  
        TupleView<(C0, C1)> where C0 : View, C1 : View  
}
```

因为我们上面例子中的 HStack 中有两个视图的表达式，所以视图构建器中具有两个参数的 buildBlock 方法被调用了。正如我们从返回类型中看到的那样，该方法构建出一个 TupleView，它包装了 Image 和 Text 这两个视图。我们可以把视图构建器看作是一种构建多元组视图 (tuple view) 的机制，而多元组视图则代表了视图的列表。

如果我们在视图构建器的闭包中只写了一个视图表达式，它将被包裹在多元组视图中，而只是被简单地按原样传递。不过，为了简化思考，我们也可以把这个例外看作是只有单个视图的列表。

SwiftUI 在许多地方都使用了视图构建器。所有像 Stack 和 Grid 这样的容器视图，以及像 background 和 overlay 这样的修饰器中，都采用了视图构建器闭包来创建它们的子视图。此外，所有视图的 body 属性都被隐式标注了 @ViewBuilder，ViewModifier 协议的 body(content:) 方法也是如此。我们也可以使用 @ViewBuilder 标注来把我们自己创建的属性和方法标记为视图构建器，我们很快会看到这样的例子。

为了更好地理解 SwiftUI 是如何使用视图列表的，以及它们是如何进行组合的，让我们把上面的例子扩展一下：

Code	View Tree
<pre>HStack(spacing: 20) { Image(systemName: "hand.wave") Text("Hello") Spacer() Text("And Goodbye!") Image(systemName: "hand.wave") }</pre>	 <pre>graph TD HStack --> Image1[Image] HStack --> Text1[Text] HStack --> Spacer[Spacer] HStack --> Text2[Text] HStack --> Image2[Image]</pre>

现在，这个 stack 有五个子视图，它们被一个含有五个元素的多元组视图所表示。为了读起来方便一些，我们可能会想把那些变得较大的 stack 分解成单独的组件，在实践中这种做法也很常见。下面是一种可行方案：

```
struct Greeting: View {
  @ViewBuilder var hello: some View {
    Image(systemName: "hand.wave")
    Text("Hello")
  }

  @ViewBuilder var bye: some View {
    Text("And Goodbye!")
    Image(systemName: "hand.wave")
  }

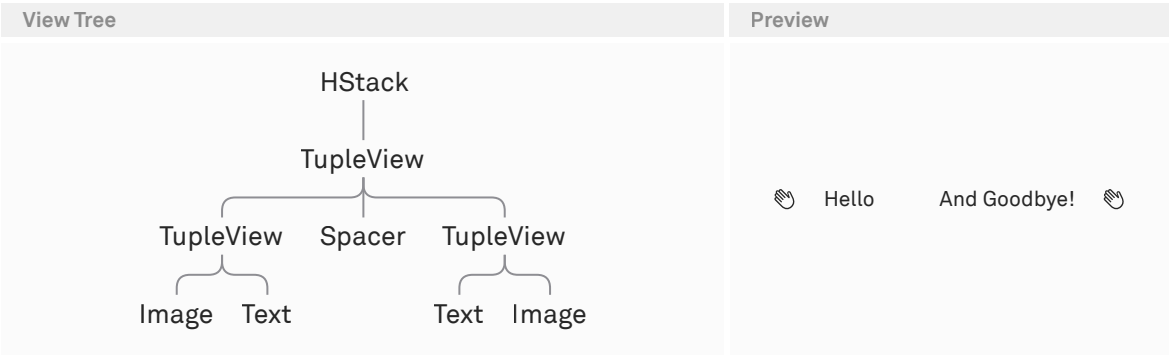
  var body: some View {
    HStack(spacing: 20) {
      hello
      Spacer()
      bye
    }
  }
}
```

通过将属性也标记为 @ViewBuilder，我们就可以像在 body 或者在上面 HStack 的闭包中所做的那样，在这个属性的主体中使用视图构建器的语法。

观察 HStack 的视图构建器闭包的类型，现在它是一个包含三个元素的
TupleView：多元组 hello、代表间隔的 Spacer、以及另一个多元组 bye：

```
TupleView<(  
  TupleView<(Image, Text)>,  
  Spacer,  
  TupleView<(Text, Image)>  
)>
```

然而，对于 HStack 来说，这和我们之前在 HStack 的视图构建器闭包中直接写五个视图的方式相比，并没有任何区别。HStack 依然有五个子视图，我们可以看到，HStack 中设定的 spacing 被应用到了每个子视图之间。



除了上面的这个图表外，在本书其他部分，我们会把 TupleView 从视图树图表中移除出去，这可以让图表具有更好的可读性。我们可以将父视图和它的子视图之间的连线看作是一个多元组视图。

视图列表拥有一个特殊的性质：当像 HStack 这样的容器类视图迭代视图列表时，嵌套的列表将会以递归的方式进行展开。这样一来，一棵多元组视图的树最终将被展平为一个一维的视图列表。这个特性甚至在我们打算把 hello 和 bye 的

视图构建器属性重构为独立的视图时，也依然适用：

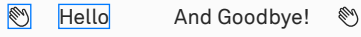
```
struct Hello: View {  
    var body: some View {  
        Image(systemName: "hand.wave")  
        Text("Hello")  
    }  
}
```

```
struct Bye: View {  
    var body: some View {  
        Text("And Goodbye!")  
        Image(systemName: "hand.wave")  
    }  
}
```


```
struct Greeting: View {  
    var body: some View {  
        HStack(spacing: 20) {  
            Hello()  
            Spacer()  
            Bye()  
        }  
    }  
}
```

因为 Hello 和 Bye 视图的 body 本身也是包含两个元素的视图列表，它们在 HStack 中依然会像之前那样被展开。

我们还可以在这些视图列表上使用视图修饰器，但它的行为可能会让人有些震惊。比如，我们可以为 Hello 视图加上一个边框：

Code	Preview
<pre>HStack(spacing: 20) { Hello() .border(.blue) Spacer() Bye() }</pre>	

这会为视图列表中的每个元素添加边框，也就是说，图片和文本都拥有各自的边框。我们经常会在使用 Group 时遇到这个行为，它所抽象的其实是一个和布局无关的视图构建器：

Code	Preview
<pre>struct Greeting: View { var body: some View { HStack { Group { Image(systemName: "hand.wave") Text("Hello") } .border(.blue) } } }</pre>	

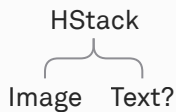
这里，因为 Group 所得到的结果是拥有两个元素的多元组视图，所以边框将会被应用到每个视图上去。如果我们想在每个视图上都应用同样的修饰器，我们可以利用这个技巧。不过，我们发现如果过度使用这个技巧，很快事情就会变得十分混乱：因为在其他所有情况下，我们通常所期望的修饰器行为和这里的表现都是大相径庭的。

不过，存在一个例外情况，我们也不确定它是不是 SwiftUI 有意为之：当把

Group (以及它的修饰器) 当作 ScrollView 的根视图或者唯一子视图时, Group 的行为就会和 VStack 很像, 修饰器也不再会被应用到 Group 中每个单独的视图里去。这条规则还有另外一个例外, 那就是将 Group 放到 overlay 或者 background 里, 这时候它表现得会像是一个 ZStack。

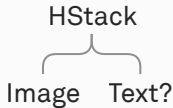
动态内容

使用视图构建器创建的视图列表也可以是动态的。下面就是一个按照条件包含视图的例子：

Code	View Tree
<pre>HStack { Image(systemName: "hand.wave") if showText { Text("Hello") } }</pre>	 <pre>graph TD HStack --> Image HStack --> Text?</pre>

在示意图中, 我们可以看到 HStack 依然拥有两个子视图: 一个 Image, 以及一个可选值的 Text。从这个视图树中, SwiftUI 将会知道 HStack 里的第一个子视图总会是一个 Image, 而第二个子视图则是一个有可能存在的 Text。

除了 if 语句, 我们也可以使用其他语句来创建条件视图, 比如 if let, switch 或者 if/else:

Code	View Tree
<pre>HStack { Image(systemName: "hand.wave") if let g = greeting { Text(g) } }</pre>	 <pre>graph TD HStack --> Image HStack --> Text["Text?"]</pre>

本章 (以及全书) 中的视图树示意图是通过视图的类型自动生成的。在 SwiftUI 大多数地方使用的是不透明的返回类型 `some View`，这个不透明类型实际上对视图的确切结构进行了编码，把复杂的嵌套类型隐藏了起来。视图的类型会准确地指出视图树中哪部分是静态的，哪部分是动态的，这让 SwiftUI 可以完全了解哪些视图可能会被动态地插入和删除。

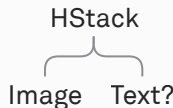
渲染树

SwiftUI 使用视图树来构建一个持久化 (persistent) 的渲染树。视图树本身是暂态的：我们偏向于把视图树看作是蓝图，它们会被一次又一次地构建，然后又被抛弃。而另一方面，持久化渲染树上的节点，则拥有更长的寿命：它们会在视图渲染的整个期间都存在，并按照当前的状态进行更新。

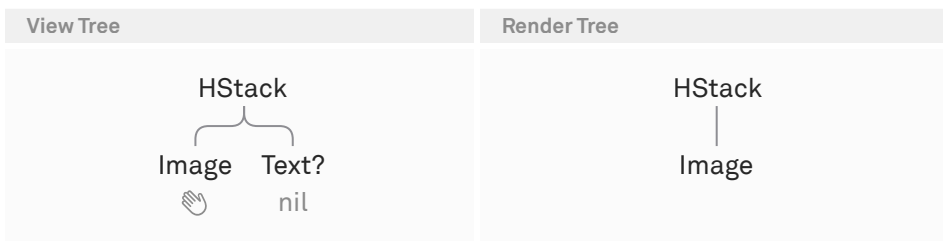
为了区分两者，我们会在谈论视图树时把它的元素称作视图，而在说到渲染树时，把元素叫做节点。这样一来，我们就可以把将视图转换为节点的过程称为“渲染”。注意，我们从来不会去直接操作渲染树，它是一个 SwiftUI 内部的概念。

渲染树其实并不是实际存在的，但是它是一个能帮助理解 SwiftUI 工作方式的很有用的模型。在现实中，SwiftUI 里有被称为属性图 (attribute graph) 的东西，它除了包含被渲染的视图以外，还负责对状态和依赖进行追踪。Apple 把渲染树里的节点称为属性。

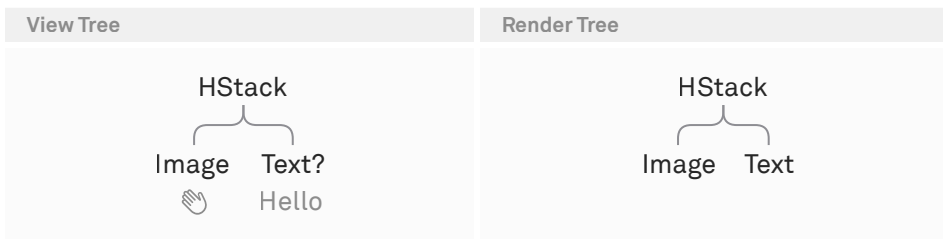
当我们第一次显示一个 SwiftUI 视图时，所构建的渲染树大多和视图树是一一对应的。比如之前的例子：

Code	View Tree
<pre>HStack { Image(systemName: "hand.wave") if let g = greeting { Text(g) } }</pre>	

当 greeting 的值是 nil 时，上面视图的渲染树中 HStack 只有一个子视图，就是图片节点。



当 `greeting` 变成一个非 `nil` 的值时，视图将会被重新构建，渲染树也将会根据新的视图树进行更新：SwiftUI 知道 `HStack` 会一直存在，所以它不需要去修改渲染树中的这部分内容。同样，它也知道图片会一直是第一个子视图。这些视图都是完全静态的。



一旦视图的更新机制检测到带有条件的视图，它就知道这个条件有可能会发生变化。当条件从 `nil` 变成非 `nil` 值时，SwiftUI 会把一个 `Text` 节点插入到渲染树中。类似地，当条件从非 `nil` 值变成 `nil` 时，SwiftUI 把 `Text` 节点从渲染树中移除出去。当一个节点被从渲染树中移除时，它所关联的状态也会全部消失。我们会在[状态一章](#)中对此进行详细讨论。

在这个例子中，还有一个场景会导致渲染树的更新：如果在更新前后 `greeting` 的值都是非 `nil` 的，那么渲染树在更新前后也都会持有相同的 `Text` 节点。不过，如果 `greeting` 的字符串值发生了变化，那么 `Text` 节点的字符串值也会被更新。

生命周期

我们之前提到过，视图树本身是暂态的，所以它的生命周期概念没什么意义。不过，渲染树中的节点拥有生命周期：节点从第一次被渲染开始存在，直到它们不再需要被显示为止，就是节点的整个生命周期。

然而，渲染树中节点的寿命和它们在屏幕上的可见性并不是一回事儿。如果我们在一个 `ScrollView` 中渲染一个很大的 `VStack`，那么不管 `VStack` 的子视图内容是不是正被显示在屏幕上，渲染树中都将包含这个 `VStack` 的所有子视图。`VStack` 会选择更“急切”地渲染内容，这点和 `LazyVStack` 这种懒加载的容器是相反的。但即使实在懒加载的 `stack` 中，渲染树中的节点在离开屏幕后也还是会被保留，以维持它们的状态 (我们会在[状态章节](#)中详细深入这些内容)。总的来说，渲染树中的节点有其生命周期，但它并不受我们的控制。

出于实际考虑，`SwiftUI` 提供了三个钩子方法 (hook) 来处理生命周期相关的事件：

1. `onAppear` 在每次视图出现在屏幕上时执行。即使渲染树中的对应节点从未消失，这个钩子方法也可以对一个视图进行多次调用。比如，在 `LazyVStack` 或 `List` 中的视图重复地滚动离开屏幕然后再回来时，每次 `onAppear` 都会被调用。在 `TabView` 中切换 `tab` 时也是如此：不仅仅是第一次显示某个 `tab`，而是每次切换到某个 `tab` 时，而它的 `onAppear` 都会被调用。
2. `onDisappear` 在视图从屏幕上消失时执行。它和 `onAppear` 是对应关系，使用同样的规则 (就算背后的节点没有消失，这个方法也可能被多次调用)。
3. `task` 是前面两者与异步操作的结合。这个修饰器在 `onAppear` 会被调用的时间点创建一个新的异步任务 (task)，并在 `onDisappear` 将被调用的时候取消这个任务。

身份标识

由于 `SwiftUI` 中的视图树并不是由那些本身就具有身份 (identity) 的引用类型 (或者说对象) 构成的，`SwiftUI` 将会根据视图在视图树中的位置，赋予这些视图身份标识。这种身份被称为**隐式身份** (implicit identity)。为了说明这一点，我们来看一

个和上面示例类似，但稍微修改过的版本：

Code	View Tree
<pre>HStack { Image(systemName: "hand.wave") if let g = greeting { Text(g) } else { Text("Hello") } }</pre>	<pre>graph TD HStack --> Image HStack --> ConditionalContent ConditionalContent --> Text1[Text] ConditionalContent --> Text2[Text]</pre>

现在视图树包含的不是一个可选值的 `Text` 了，它是一个拥有两个子视图的 `ConditionalContent` 视图：非 `nil` 的情况对应着一个 `Text`，而 `nil` 的情况则对应着另一个 `Text`。视图树中的每个视图都按照它们在树中的位置拥有一个唯一的身份标识。为了说明这个概念，我们来想象构建一个“路径”字符串，并用它来标识每个视图：

Code	View Tree
<pre>HStack { Image(systemName: "hand.wave") // 0 if let g = greeting { Text(g) // 1.ifBranch } else { Text("Hello") // 1.elseBranch } }</pre>	<pre>graph TD HStack -- 0 --> Image HStack -- 1 --> ConditionalContent ConditionalContent -- 1.ifBranch --> Text1[Text] ConditionalContent -- 1.elseBranch --> Text2[Text]</pre>

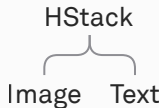
因为 `Image` 是 `HStack` 的第一个子视图，我们设定它的路径是 "0"。而 `ConditionalContent` 是 `HStack` 的第二个子视图，`if let` 语句所指定的非 `nil` 分支

中的 `Text` 是这个 `ConditionalContent` 的第一个子视图，我们把它的路径设置为 `"1.ifBranch"`。

我们并不是说这些路径字符串是 SwiftUI 在内部实现隐式身份标识的方式；它们只是用来向读者展示隐性身份含义的一种模型。

现在来考虑一下 `if let` 语句中的两个分支中的两个 `Text` 视图。它们具有不同的身份，因此 SwiftUI 会认为它们是两个不同的视图。当条件发生变化时，旧的 `Text` 将从渲染树中移除，新的 `Text` 被插入。这在状态、动画和过渡方面会产生各种后果，我们将在后面讨论。

让我们来看一个相同示例，但写法稍微不同：

Code	View Tree
<pre>HStack { Image(systemName: "hand.wave") Text(greeting ?? "Hello") }</pre>	 <pre>graph TD HStack --> Image HStack --> Text</pre>

一个显而易见的结果是，视图树现在更简单了：`HStack` 拥有两个静态的子视图：`Image` 和 `Text`。现在 `nil` 和非 `nil` 的 `greeting` 值的唯一区别，就只有显示在 `Text` 视图上的字符串了。`Text` 视图本身的隐式身份（也就是 `HStack` 中的第二个子视图）将不再被 `greeting` 值的变化所影响，这个视图将始终存在。

除了隐式身份外，视图也可以具有**显式身份**。这主要用在 `ForEach` 中的视图中，`ForEach` 里的每个项目都会被分配一个显式的身份标识。一般我们会使用想要显示的数据的某个唯一标识符来作为身份标识（比如让项目的类型实现 `Identifiable` 协议，或者是提供一个指向项目唯一标识的键路径）。但是，我们也可以使用 `id` 修饰器来手动分配显式标识符。

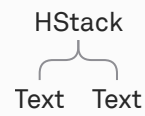
id 的参数可以是任何 Hashable 的值。在下面的示例中，我们使用一个布尔值来将 greeting 与 nil 进行比较。如果它是 nil，则显式身份标识为 true。否则，它为 false。这意味着当身份发生更改时，SwiftUI 会认为 Text 视图是一个不同的视图。同样，这将从渲染树中删除先前的 Text 节点，然后再插入一个新的节点。

Code	View Tree
<pre>HStack { Image(systemName: "hand.wave") Text(greeting ?? "Hello") .id(greeting == nil) }</pre>	<pre>graph TD HStack -- 0 --> Image HStack -- 1 --> id id -- true --> Text</pre>

值得注意的是，像上面这样的显式身份标识并不会覆盖视图原先的隐式身份标识。显式身份只是隐式身份的一种“附加说明”。换句话说，SwiftUI 并不会因为在多个视图上使用了相同的显式身份标识符而把它们弄混。正如所见，视图的路径是一种为视图赋予隐式身份的具体方式，而对于显式身份，我们则可以把它想成“附加到这个路径”上的额外内容。

现在我们对 SwiftUI 的视图身份有更深入的理解了，让我们来看看关于这个话题的两个最常见的问题吧。

首先，来看看下面这个例子：

Code	View Tree
<pre>HStack { let v = Text("Hello") v v }</pre>	 <pre>graph TD HStack --- Text1[Text] HStack --- Text2[Text]</pre>

这里，我们再用一个本地变量中构建了一个 `Text` 视图，并在 `HStack` 里使用了两次。从这两个 `Text` 视图的身份标识来看，这意味着什么呢？

很显然，正如视图树的图中所示，在视图树中这两个 `Text` 视图是处于不同位置的。因为，它们拥有不同的隐式身份标识，并且被 SwiftUI 识别为不同的视图。我们也可以从“蓝图”的概念的角度来思考：我们创建了一个具有字符串 "Hello" 的 `Text` 视图的蓝图，然后我们用了两次这个蓝图。

这里是另一个与视图身份相关的例子——这是一个简单的视图扩展，用来根据条件选择是否要应用某个视图修饰器：

```
// ⚠️ 反面模式  
extension View {  
  @ViewBuilder  
  func applyIf<V: View>(_ condition: Bool, transform: (Self) -> V) -> some View {  
    if condition {  
      transform(self)  
    } else {  
      self  
    }  
  }  
}
```

`applyIf` 方法可以这样使用：

Code	View Tree
<pre>HStack { Image(systemName: "hand.wave") Text("Hello") .applyIf(highlighted) { \$0.background(.yellow) } }</pre>	<pre>graph TD HStack --> Image HStack --> ConditionalContent ConditionalContent --> background[".background"] ConditionalContent --> Text2[Text] background --> Text1[Text] background --> Color[Color]</pre>

观察这个视图树的结果，我们可以看到 `applyIf` 修饰器引入含有两个子视图的 `ConditionalContent`：它包含一个未被修改的 `Text`，以及一个包含背景的 `Text`。这意味着，当条件 (`highlighted`) 改变时，屏幕上的 `Text` 的身份标识也会跟着发生改变。

我们强烈建议不要使用这种模式，因为看似无害的 `applyIf` 修饰器在视图树中引入了一个分支，它有可能会在下游导致一系列无法预测的后果。相反，下面的方法会更加安全：

Code	View Tree
<pre>HStack { Image(systemName: "hand.wave") Text("Hello") .background(highlighted ? .red : .clear) }</pre>	<pre>graph TD HStack --> Image HStack --> background[".background"] background --> Text[Text] background --> Color[Color]</pre>

大多数视图修饰器都能够接受可选值，所以我们可以使用条件运算符模式来选择指定一个值或者是在不想指定值时传递 `nil`。比如，`frame` 修饰器的 `width` 和 `height` 都是可选值，`foregroundColor` 的颜色是可选值，`padding` 的长度也是可选值。出于同样的原因，像是 `bold` 或者 `disabled` 这样的视图修饰器也接受一个

布尔值参数，而只是想当然的话，可能会认为这些参数是不必要的。

状态和绑定

3

在前一章中，我们看到了将视图树当作蓝图进行构建，并将它们转换为持久化的渲染树的方式。为了构建动态的应用程序，我们需要基于当前的状态来创建不同的视图树，并依靠 SwiftUI 来更新对应的渲染树。这是 SwiftUI 的最大优势之一：它会自动观察状态，并始终让我们的视图和模型保持同步。

一般来说，视图更新的周期可以总结如下：

1. 视图树被构建出来。
2. 渲染树中的节点被创建、删除或更新，以匹配当前的视图树。
3. 某个事件引起了状态的改变。
4. 重复这个过程。

原则上，我们不需要担心视图树需要被重新创建的时机，不需要担心被状态改变所影响的到底是哪些部分，也不需要担心要如何更新屏幕上的内容才能正确对应当前的视图状态：因为 SwiftUI 将负责处理所有这些事情。与之相对，我们的任务是根据特定的状态来描述应该显示在屏幕上的到底是什么内容。

这里叠个甲，我们应该补充一下，有些时候确实需要考虑视图树的哪些部分正在被重新渲染，以及被渲染的原因是什么。如果遇到性能问题，原因很可能是需要进行更新的视图实在是太多了。我们将在本章末尾进一步讨论这个问题。

SwiftUI 提供了好几种不同的状态包装类型，取决于状态是一个值还是一个对象，以及它是视图私有的状态还是是由外部传入的状态，应该选用不同的包装。不过，通常我们不需要直接处理这些包装器的类型，因为 SwiftUI 通过像是 `@State`、`@StateObject` 和 `@ObservedObject` 这样的属性包装器将它们全部暴露出来了。

在 iOS 17 中，SwiftUI 与对象交互的方式已经完全改变。SwiftUI 不再依赖 Combine 框架进行观察，而是采用基于宏的解决方案，这也使得 `@StateObject` 和 `@ObservedObject` 属性包装器变得不再必要。`@State` 属性包装器现在可以

用于值和对象，而在 iOS 17 之前，我们通常只将其用于值。

由于 `@State` 和所有版本的 SwiftUI 都相关，我们将首先深入了解这个属性包装器，然后再对 iOS 17 之前和之后有关对象观察方面的情况进行区分。

State

`@State` 属性包装器是将状态引入到 SwiftUI app 中最简单的方式。它用于私有的视图状态值。比如，这里有一个简单的计数器视图：

```
struct Counter: View {
    @State private var value = 0
    var body: some View {
        Button("Increment: \(value)") {
            value += 1
        }
    }
}
```

当计数器被首次渲染时，状态属性 `value` 所具有的初始值是 0。在 `body` 属性被执行期间，SwiftUI 会注意到这个状态属性被访问了，它会在 `value` 状态属性和渲染树中计数器视图的节点之间添加一个依赖关系。这样一来，每当 `value` 的值发生变化（比如按钮被点击了），SwiftUI 就会重新执行计数器视图的 `body`。

注意，如果我们没有把 `value` 放到按钮的文本中的话，SwiftUI 将会足够聪明地指出，就算这个状态属性发生了变化，它也不需要重新渲染计数器的 `body` 属性。

我们可能会好奇，每次计数器视图被初始化，`Counter` 实例被创建时，我们都把 `value` 属性赋值成了 0，那它到底是怎么才能发生变化的。为了让这个行为更通透一些，并且消除掉一些神奇的魔法，我们可以试着不使用 `@State` 属性包装器来书写相同的代码：

```

struct Counter: View {
    private var _value = State(initialValue: 0)
    private var value: Int {
        get { _value.wrappedValue }
        nonmutating set { _value.wrappedValue = newValue }
    }

    var body: some View {
        Button("Increment: \(value)") {
            value += 1
        }
    }
}

```

我们现在没有依赖 `@State`，而是自己创建了一个 `State` 的值，并把它分配给 `_value` 属性。`State(initialValue:)` 这个初始化方法明确地显示了 0 只是这个状态属性的初始值。它在渲染树中只对应了计数器视图节点被首次创建时使用的值。一旦节点存在，状态属性的这个初始值就会被忽略，SwiftUI 将会负责在重新渲染的过程中保留当前的状态值。

除了 `_value` 属性，我们还添加了一个计算属性 `value`，它可以让状态的使用更容易些：我们可以只使用 `value`，让计算属性把调用透明地转发给 `_value.wrappedValue`，这样我们就不需要每次都调用 `_value.wrappedValue` 来进行读写了。当我们在视图的 `body` 中使用 `State` 属性的 `wrappedValue` 时，我们实际上处理的是一个存在于渲染树中的持久化状态值的引用。

`@State` 属性包装器为我们完成了所有这些事情：它创建了一个 (用来存储实际的 `State` 值的) 带下划线版本的属性，以及将 `getter` 和 `setter` 转发给 `wrappedValue` 的计算属性。

让我们来研究两个场景：首先是上面的视图被首次渲染的情况，然后是按钮被点击后第二次渲染的情况。首先，计数器视图第一次出现在屏幕上时，发生了这些事情：

为了使本章中的状态示意图更易读，在图表中我们用这种颜色来把与上一步骤中的示意图相比发生了变化的部分进行高亮标记。

步骤 1 当 Counter 结构体被第一次构建时，渲染树上还不存在与之对应的节点。在下面的示意图中，视图结构体位于左侧。它的上半部分代表的是状态属性，其中包含了两个内部值：initialValue 是在属性初始化时我们分配给它的值，wrappedValue 则是我们在视图的 body 中进行交互所用的值。我们可以将 wrappedValue 视为指向此状态属性实际值的指针，目前它尚未指向任何内容。视图结构体的下半部分表示视图的 body，此刻它尚未被执行，所以仍然为空。



计数器视图

∅

渲染树中的计数器节点

步骤 2 当 SwiftUI 在渲染树中创建这个计数器视图的节点时 (也就是右侧)，它会为视图的状态属性申请内存。系统使用状态属性的初始值 initialValue，也就是 0，在渲染树节点中为 value 属性初始化内存。状态属性的 wrappedValue 现在将指向渲染节点中的内存。

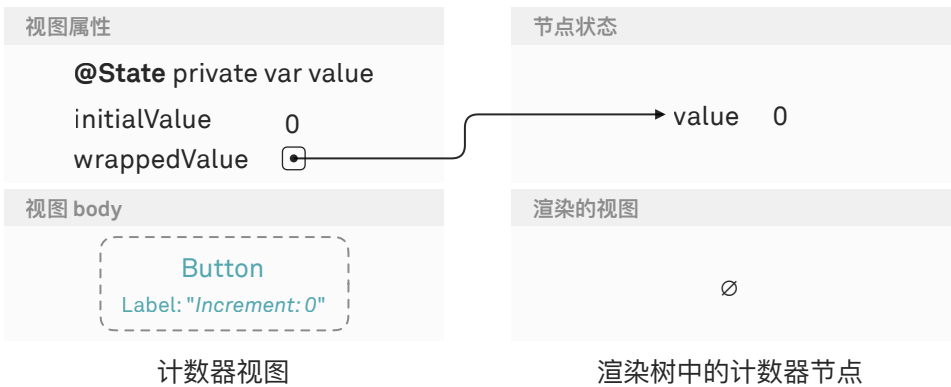


计数器视图

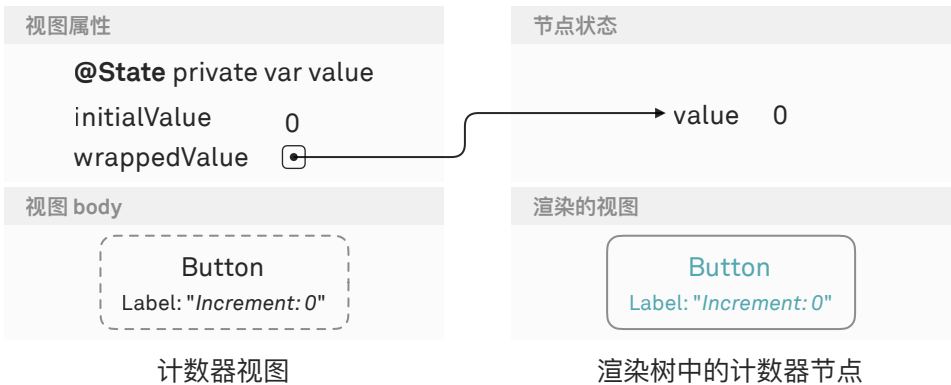
渲染树中的计数器节点

步骤 3 视图的 body 被执行，它创建了 Button 视图。由于状态属性的

wrappedValue 现在指向的是渲染节点中的内存，所以在构建按钮的标题时，使用了渲染节点中存储的值。



步骤 4 将计数器视图的 body 作为蓝图，在渲染树中创建按钮视图的节点。

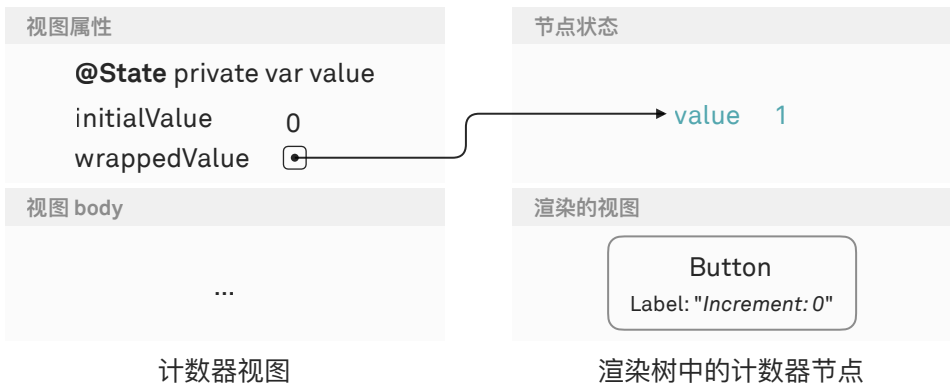


因为我们在 Counter 的 body 中要渲染按钮的文本，在那里我们使用了 value，所以 SwiftUI 在渲染树中的内存和计数器视图的 body 之间建立了依赖关系。

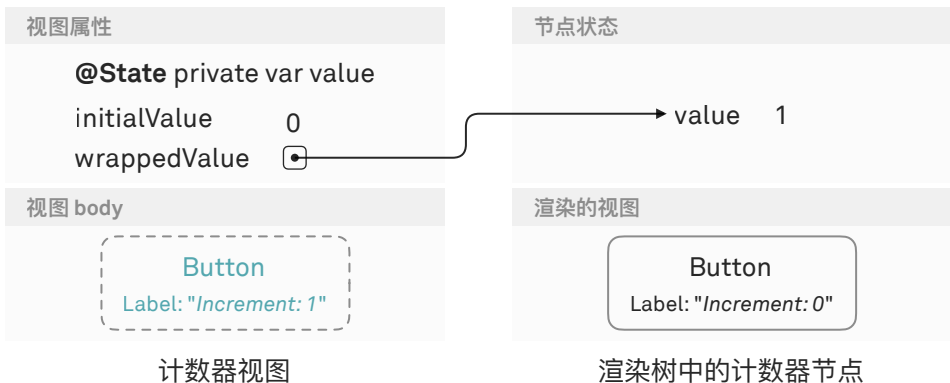
现在，我们来看看按钮被点击时候的情况：

步骤 1 由于 value 属性实际上是 `_value.wrappedValue`，它是指向渲染树中的内

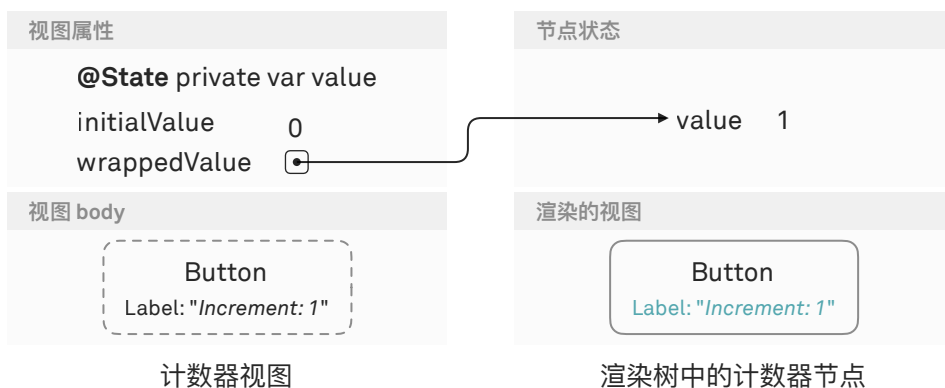
存的指针，所以该内存的内容将被递增。



步骤 2 由于计数器的 `body` 依赖状态内存，所以它会被重新执行，这将构建一个新的按钮视图。现在，为了得到按钮的文本，我们对 `value` 属性进行访问，即使状态属性的初始值依然是 0，但此时它所指向的内存将返回 1。



步骤 3 使用这个新构建的计数器视图的视图树作为蓝图，渲染树中的按钮文本将会变为新的值。



在本节一开始，我们提到了 `@State` 应该被用于**私有 (private)** 的视图状态，这就是我们使用 `private` 关键字标记所有状态属性的原因。虽然这种标记并不是必须的，但我们认为这是一个好习惯。而且通过对 `@State` 属性包装器底层原理的瞥视，我们现在可以更好地对它进行说明：在上面的代码中，我们看到 `State` 的初始化器只接受一个状态的初始值。让我们考虑一下，如果我们通过视图的初始化器将它公开出来，会发生什么情况：

```
struct Counter: View {
    @State private var value: Int

    init(value: Int = 0) {
        _value = State(initialValue: value)
    }

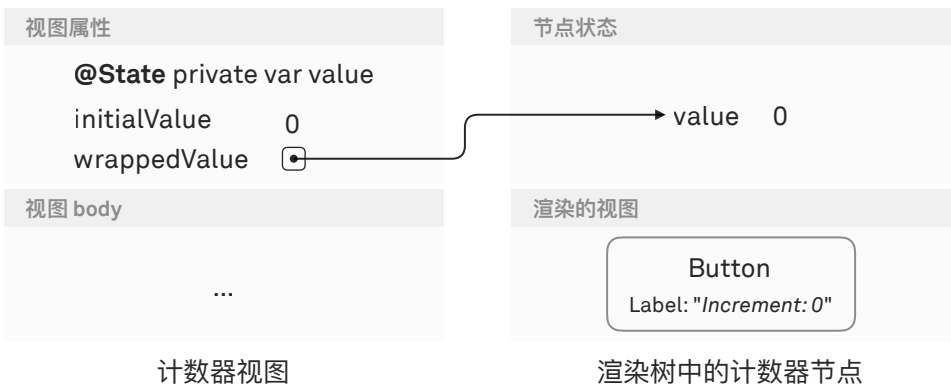
    var body: some View {
        Button("Increment: \(value)") {
            value += 1
        }
    }
}
```

现在我们可以从外部传入值，但因为在视图的初始化方法中我们无法访问状态的当前值，这个传入值只会改变状态属性的**初始值**。一旦该视图的节点在渲染树中

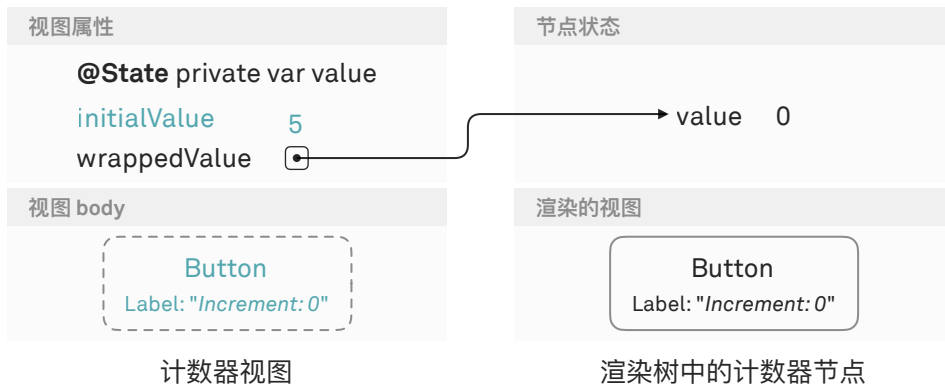
被创建，传入不同的初始值将没有任何效果，或者至少不是我们所期望的效果。这里发生的只是初始值 (而不是实际值) 的改变，只有当节点被移除并重新插入到渲染树中时，这个初始值才会产生影响。

比如，当我们一开始以值为 0 去渲染计数器视图，然后像上面所提到的那样，将这个从外部传入的值更新为 5 时，会发生以下情况：

步骤 1 我们从最初渲染的计数器视图开始。它的初始值是 0，wrappedValue 指向渲染节点中的状态内存。



步骤 2 现在我们从外部向计数器视图传入 5，这会构建一个新的计数器视图，但是这里只有初始值发生了改变。wrappedValue 依然指向原来的状态内存，它的值还是 0。所以这时按钮的文本依然显示的是 0。



如果我们确实想要一个能从外界传入值的初始化方法，至少把初始化方法的参数标签名修改一下，会更加清晰，比如：

```
init(initialValue: Int = 0) {  
    _value = State(initialValue: initialValue)  
}
```

将参数命名为 `initialValue` 可以更清晰地传达这一行为，但是根据我们的经验，从外部初始化状态属性其实并没有什么实用价值。因此，我们还是建议将所有的 `@State` 都声明为私有属性。

我们可能会想，为什么我们不能用下面的方式来写初始化方法呢？编译器似乎对此并无意见：

```
struct Counter: View {  
    @State private var value = 0  
  
    init(value: Int = 0) {  
        self.value = value  
    }  
    ...  
}
```

我们之前就知道，`self.value = value` 语句实际上会被转变为 `self._value.wrappedValue = value`。从编译器的角度来看，这个语句没什么问题，但是它还是没法做到我们所期望的事情。不过，从它无法工作的原因中，我们可以得到一个宝贵的教训。

在上一章里，我们讨论过 SwiftUI 基于视图结构的身份标识的概念。简而言之，视图会通过它在视图树中的位置来获取身份标识。在上面的初始化方法中进行赋值，无法成功的原因是双重的：视图的状态是和它的身份标识所关联的，而在初始化方法运行的时候，视图还没有身份标识。为了更好地理解 SwiftUI 为什么在这个时间点上还无法得知视图的身份标识，我们可以进行一次思维实验。考虑以下代码片段：

```
struct ContentView: View {  
    let counter = Counter()  
    VStack {  
        counter  
        counter  
    }  
}
```

在这个例子中，`Counter` 的初始化方法只在我们创建结构体和将它分配给本地变量 `counter` 的时候运行一次。不过，在这个时间点上，我们还没有把视图放到视图层级中去，所以在初始化方法执行时，视图还不具有任何身份标识，因此我们就无法访问视图的状态。稍后，当计数器的 `body` 被执行时，视图将拥有身份标识，因此状态属性也可以引用到渲染树中的正确状态值。

虽然原本 `@State` 属性包装器几乎是限定用于值类型的，但随着 iOS 17 引入的 `Observable` 宏，这种情况现在发生了变化，我们将在下一节中看到这个话题。然而，在旧版本平台上，`@State` 仍然只应该被应用于值类型，除非我们确切知道为什么需要违反这个规则。我们将在[本章的后续部分](#)中讨论在旧版 SwiftUI 中观察对象的方式。

Observable 宏

Observable 宏是 SwiftUI 中用于观察对象状态的机制。它是 Observation 框架的一部分，于 WWDC23 推出，是我们此前一直使用的整个基于 Combine 框架的对象观察模型的替代品。

Observable 宏做的事情有两件：

- 它会为所附加的类型添加遵守 Observable 标记协议的声明。这是一个空协议，用于在编译时标记一个类，它在运行时没有实际的作用。
- 它修改对象的属性，以追踪读取和写入访问。

不管对象存储在哪里，也不需要任何特殊的属性包装器，只需访问 Observable 对象的属性，对它的观察就自动建立了。因此，我们只需要使用 @State 属性包装器，并与 Observable 对象结合起来，就可以更改对象的生命周期管理方式了。存在两种使用方式：

- 想要将 Observable 对象的生命周期与视图的渲染节点的生命周期 (换句话说，它是视图私有的对象) 关联起来，我们使用 @State 属性包装器来声明属性。
- 想要使用具有独立于视图的渲染节点的生命周期的对象 (换句话说，我们从外部传递的对象)，我们只需将它存储在普通属性中。

相较于过去 SwiftUI 处理对象观察的方式，在 iOS 17 中，对象的生命周期和观察的概念被分开了。此前，我们使用 @State 来让 SwiftUI 对状态值在视图更新之间的生命周期进行管理；使用 @StateObject 来获得相同的生命周期管理行为，并附加上对于对象的观察。在 Observable 中，观察的部分不再包含于属性包装器中，我们只剩下使用 @State 来表示 SwiftUI 对状态对象的生命周期进行管理的需求了。

想要使用模型对象而不是一个简单的值来编写上例中的计数器，可以这样做：

```
@Observable final class Model {  
    var value = 0  
}
```

```
struct Counter: View {
```

```

@State private var model = Model()
var body: some View {
    Button("Increment: \(model.value)") {
        model.value += 1
    }
}
}

```

从值的归属权的角度来看，将 @State 与可观察对象一起使用，与上面章节中使用值类型的 @State 示例的工作方式是相同的。SwiftUI 在渲染树节点中为对象分配内存，并在节点存在的同时保持对象的存活。@State 属性包装器的包装值将指向该对象。

首先让我们来看一个示例，这里我们使用一个没有属性包装器的 @Observable 对象：

```

@Observable final class Model {
    var value = 0
    static let shared = Model()
}

struct Counter: View {
    var model: Model
    var body: some View {
        Button("Increment: \(model.value)") {
            model.value += 1
        }
    }
}

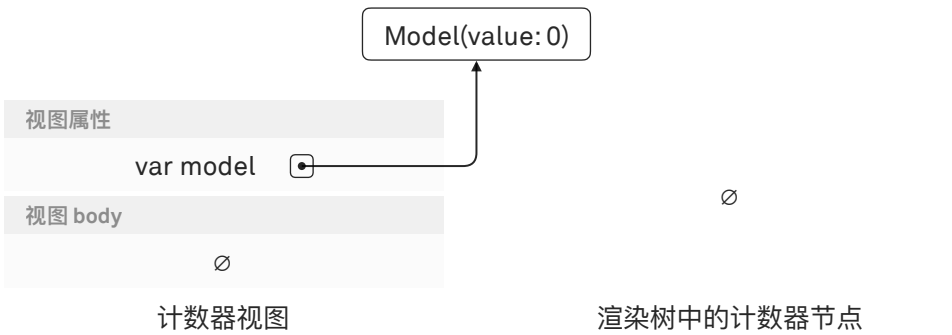
struct ContentView: View {
    var body: some View {
        Counter(model: Model.shared)
    }
}

```

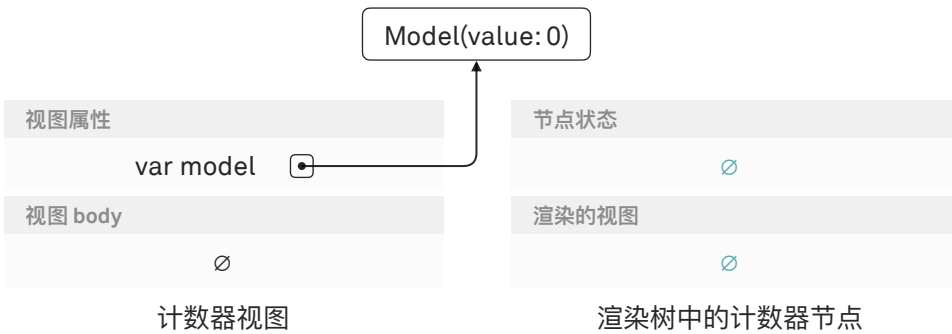
现在，计数器视图中的 model 属性不再声明为 @State，我们可以像在上面的

ContentView 中那样从外部传递它。当首次创建计数器时，会发生下面这些事情：

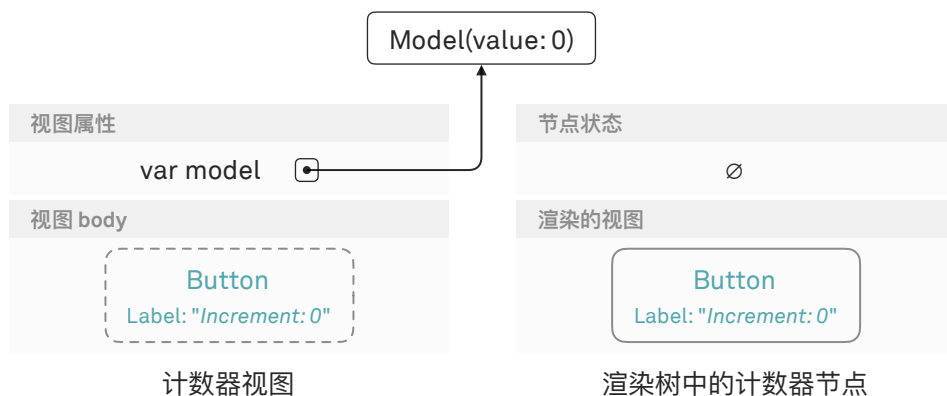
步骤 1 model 属性指向我们传递给计数器视图的 model 实例。



步骤 2 渲染节点在创建时不包含任何状态，因为计数器视图中只存在普通的属性。



步骤3 执行计数器视图的 body，并构建按钮视图。在 body 内，按钮的标题使用的是存储在计数器视图结构体的 model 属性中的外部 model 对象。之后，更新渲染节点以反映计数器视图的新视图树。



在没有 `@State` 的情况下，SwiftUI 对于这个对象没有任何生命周期的责任，当视图的 `body` 被执行并且访问对象的属性时，观察会自动发生。因此，计数器视图的渲染节点自身不需要维护任何状态。

当然，在这个特定的示例中，我们可以在计数器视图中直接编写以下内容，而不是从外部传入模型：

```
struct Counter: View {  
    var model = Model.shared  
    ...  
}
```

因为我们始终传递的是相同的 `Model.shared` 实例，所以这里做的事情是完全相同的。不过，根据其他状态，我们也可以把不同的模型对象传递到视图中。

新的基于宏的对象观察模型不仅引入了更方便的语法，还改变了视图和 `Observable` 对象之间的依赖关系的形成方式：使用旧的属性包装器时，SwiftUI 会无脑订阅视图中所有被声明为 `@StateObject` 或 `@ObservedObject` 的属性中的 `objectWillChange publisher`。而使用新的 `Observable` 宏时，不论 `Observable` 对象来自何处，我们只要在视图 `body` 中访问到的它的属性，这个属性就会成为视图的依赖。

这套新模型比起之前要简单得多。例如，在视图 `body` 中访问一个 (可观察的) 全

局单例，将会自动形成访问属性与视图之间的依赖关系，而无需我们使用 `@ObservedObject` 将此单例传递给视图。同样，`Observable` 对象可以嵌套在可选值、数组或其他集合中。由于这是在视图 `body` 中在属性层级上进行的追踪和依赖，所以观察和视图更新等都会按照预期工作。

新的模型同时也十分高效。如果你在视图 `body` 中仅使用了对象的某一个属性，那么对其他属性的更改并不会导致该视图重新绘制。同样，如果你没有使用模型对象 (例如，当它仅在代码的一个分支中存在时)，则根本不会观察这个对象。这可以减少很多不必要的视图更新，从而提高性能，而在以前，我们必须手动拆分模型对象，才能获得更精细的视图更新行为。

State 和 Observable

就观察对象而言，我们主要看到有两类常见的错误：使用 `@State` 来处理从外部传入的对象，以及不使用 `@State` 来处理视图内部私有的对象。换句话说，前一个问题是，在对象的生命周期已经由视图外部进行管理时，依然使用了 `@State`；而后一个问题是，在对象的生命周期没有在视图外部进行任何管理时，却仍然不使用 `@State`。

下面是第一个问题的例子，这里我们使用了 `@State` 属性来持有一个从外部传入的对象：

```
struct Counter: View {
    @State var model: Model

    init(model: Model) {
        self.model = model
    }

    var body: some View {
        Button("\(model.value)") { model.value += 1 }
    }
}
```

正如在[关于 @State 的部分](#)所讨论的那样，问题在于在初始化方法运行时，视图

还没有标识符。因此，这个初始化方法所做的只是更改 `@State` 属性的初始值，但它不会影响视图已经在屏幕上时正在使用的状态。

就算我们只是从外部传入一个值，然后在初始化方法中用这个值来构建模型对象，情况也是一样的：

```
struct Counter: View {
  @State var model: CounterViewModel

  init(value: Int) {
    self.model = CounterViewModel(value: value)
  }

  var body: some View {
    Button("\(model.value)") { model.value += 1 }
  }
}
```

和上面一样，如果计数器视图已经在屏幕上时，就算我们将一个不同的值传递给初始化方法，这也不会改变按钮的标题。只有 `@State` 属性的初始值将被更改为新的 `CounterViewModel` 实例，而这个值仅只会在视图下次被插入渲染树时使用。

作为反例，让我们看看如果把 `@State` 标签从这个视图中移除时，会发生什么：

```
struct Counter: View {
  var model: CounterViewModel

  init(value: Int) {
    self.model = CounterViewModel(value: value)
  }
  ...
}
```

现在，计数器视图本身不再维护模型对象的生命周期了。当我们传入一个新值时，新的视图模型将被构建，并且视图的 `body` 将使用这个新对象来渲染自身。

然而，我们会遇到另一个问题：只有在视图层级没有被改变，计数器视图没有被重新创建时，这些代码才能正确工作。一旦它被重新创建，那么视图就会失去它的状态，因为在视图的初始化方法中，计数器的 view model 会被再次重新创建。

想要解决这个问题，我们可以从外部传入视图模型，或从某个全局的 model 对象中获取这个模型，或者也可以采取类似的其他方法。但无论如何，在视图中，如果模型属性没有使用 @State 声明，那么视图模型的生命周期必须要在视图之外的某个地方进行管理。

Observable 宏的工作方式

这个新的观察模型行为一开始似乎非常奇妙：我们在视图 body 中访问对象属性时，观察就这么自动发生了。为了实现这一点，SwiftUI 利用了新的 Swift 宏，来把整套机制正常工作所需的代码隐藏了起来。不过，Observable 宏只是为了让我们的模型对象保持干净，而我们依然可以在不使用宏的情况下获得相同的功能。让我们逐步查看生成的代码：

```
@Observable final class Model {
    var value = 0 {
        get {
            access(keyPath: \.value)
            return _value
        }
        set {
            withMutation(keyPath: \.value) {
                _value = newValue
            }
        }
    }
}

@ObservationIgnored private var _value = 0
...
}
```

宏把 value 属性从存储属性转变为计算属性，并添加了一个额外的私有存储属性 _value。这个私有属性被用作计算属性背后的存储。在计算属性 value 中，

getter 先 (通过调用带有键路径的 `access` 方法) 记录了对该属性的访问, 然后返回 `_value`。setter 在 `withMutation` 闭包中对私有 `_value` 属性进行更改, 而这个方法参数中也含有属性的键路径。

`access` 和 `withMutation` 是两个宏生成的方法:

```
@Observable final class Model {  
    ...  
    @ObservationIgnored private let _$observationRegistrar = ObservationRegistrar()  
  
    internal nonisolated func access<Member>(keyPath: KeyPath<Model, Member>) {  
        _$observationRegistrar.access(self, keyPath: keyPath)  
    }  
  
    internal nonisolated func withMutation<Member, T>(  
        keyPath: KeyPath<Model, Member>,  
        _ mutation: () throws -> T  
    ) rethrows -> T {  
        try _$observationRegistrar.withMutation(of: self, keyPath: keyPath, mutation)  
    }  
}
```

这两个方法都将调用转发到对象的观察注册器 (observation registrar) 上的对应的方法。这个注册器负责跟踪对特定属性感兴趣的观察者, 并在属性更改时通知这些观察者。

那么, 对象的属性与 SwiftUI 视图之间的连接又是如何形成的呢? 存在一个全局函数 `withObservationTracking(_ apply:onChange:)`, 它接受两个闭包。第一个闭包 `apply` 会立即执行, 并且观察系统会跟踪在 `apply` 期间访问了哪些属性。第二个闭包 `onChange` 则是负责观察的闭包: 在 `apply` 中访问过的任何可观察属性发生更改时, `onChange` 闭包会被调用。

`withObservationTracking` 将 `onChange` 观察闭包存储在全局变量中, 然后执行 `apply`。在 `apply` 完成后, 被访问的对象会在被访问的属性和观察闭包之间添加一个依赖关系。我们可以想象, 在执行视图的 `body` 时, SwiftUI 做了类似这样的

操作：

```
withObservationTracking {  
    view.body  
} onChange: {  
    view.needsUpdate()  
}
```

通过这样做，我们在视图的 body 中对任何可观察属性的访问都会 (直接或间接地) 通过对象的观察注册器，并形成该特定属性和当前正在执行的视图 body 之间的依赖关系。

ObservableObject 协议

在 iOS 17 之前，我们需要使用 ObservableObject 协议，并组合使用 @StateObject 或者 @ObservedObject 属性包装器，来观察对象的状态变化。如果我们需要针对旧版平台，我们仍然需要了解这些机制。因此，在本节中，我们将详细描述 @StateObject 和 @ObservedObject 的工作原理，以及何时应该使用哪个包装器。

StateObject

@StateObject 属性包装器的工作方式与 @State 大致相同：我们指定一个初始值 (在这里是一个对象)，当渲染树中的节点被创建时，这个值会被用作状态的起点。接下来，SwiftUI 将在渲染树中节点的整个生命周期内持有这个对象，以应对多次渲染。

除了 @State 的功能外，@StateObject 还通过 ObservableObject 协议来观察对象上的更改，这是 SwiftUI 与模型层之间的少数几个接口之一。下面是一个新版本的计数器视图，它使用了模型对象，而不是一个简单值：

```
final class Model: ObservableObject {
    @Published var value = 0
}

struct Counter: View {
    @StateObject private var model = Model()
    var body: some View {
        Button("Increment: \(model.value)") {
            model.value += 1
        }
    }
}
```

让我们来探索一下这些代码是如何让计数器工作起来的。首先，我们来看看

@StateObject 属性包装器背后做了些什么。我们可以在不使用属性包装器的情况下编写相同的代码，如下所示：

```
struct Counter: View {
    private var _model = StateObject(wrappedValue: Model())
    private var model: Model { _model.wrappedValue }

    var body: some View {
        Button("Increment: \(model.value)") { model.value += 1 }
    }
}
```

和 @State 类似，@StateObject 属性包装器也会创建一个下划线版本的属性，也就是本例中的 _model，它包含有一个 StateObject 类型的值。此外，它还会生成一个计算属性，将 getter 转换为 StateObject 的 wrappedValue。

StateObject 的 wrappedValue 必须是一个遵守 ObservableObject 协议的值。要遵守这个协议，要做的事情只有一件，那就是它的实现必须拥有一个名为 objectWillChange 的 Combine Publisher 变量，且该 Publisher 的失败类型需要为 Never (也就是说这个 Publisher 不会失败)。通常来说，这个 Publisher 的输出值类型会是 Void，也就是说该 Publisher 所发布的事件不带有额外信息。不过，这并不是实现该协议所需要的正式要求。对于 objectWillChange 的唯一要求是，这个 Publisher 必须在模型中的值每次将要变化之前发出一个事件。

在 SwiftUI 的早期测试版中，这个要求曾经是 objectDidChange，而非 objectWillChange。不过，objectWillChange 可以更好地对更新进行合并。

在上面的示例中，我们依赖了从 ObservableObject 协议中所默认实现的 objectWillChange publisher，并使用了 @Published 属性包装器来在 value 属性更改之前发送一个事件。除了使用 @Published 外，我们也可以编写以下代码：

```
final class Model: ObservableObject {
    var value = 0 {
```

```

    willSet { objectWillChange.send() }
  }
}

```

`@Published` 是上述代码的语法糖。然而，我们只能在使用 `objectWillChange` publisher 的默认实现时，才能使用这个属性包装。如果我们由于某种原因实现了我们自己的 publisher，那么 `@Published` 将会失去作用。

当上面的视图被构建时，会发生以下步骤：

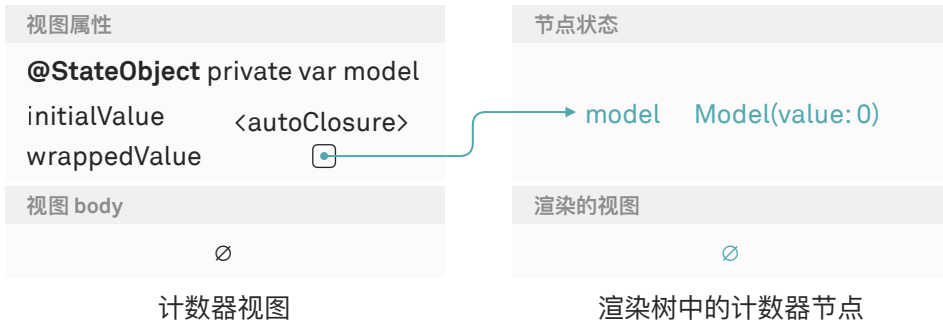
步骤 1 计数器视图将被创建，但是 `state` 对象中的值还不存在，这是因为初始值是被存放在一个 autoclosure 中的。



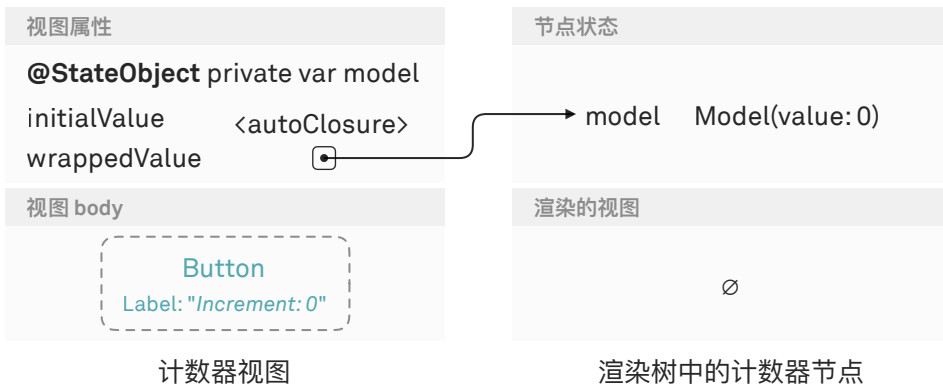
计数器视图

渲染树中的计数器节点

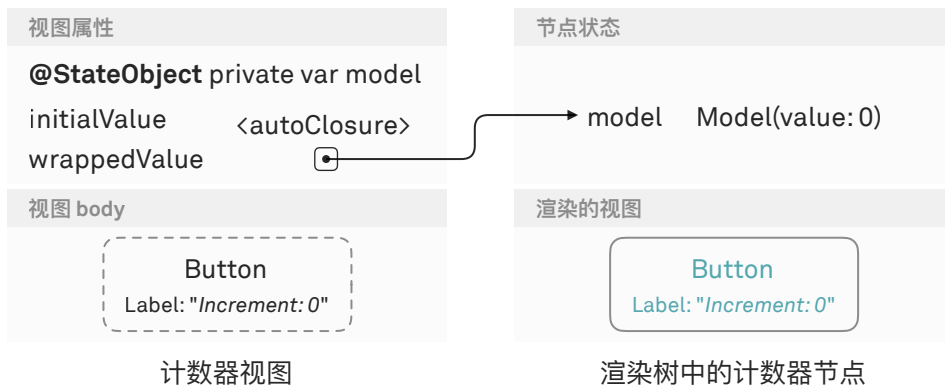
步骤 2 在渲染树中创建计数器节点，`autoclosure` 中返回的值被用来初始化 `model` 这个状态对象的内存。`StateObject` 结构体中的 `wrappedValue` 指向储存在渲染树中的 `model` 对象。



步骤 3 计数器视图的 `body` 被执行，它创建了按钮视图，渲染树中的 `model` 对象的当前值被用在按钮的文本标签上。



步骤 4 按钮视图被当作蓝图，用来更新渲染树中的节点。



和 `@State` 一样，`@StateObject` 也只应该用于私有的视图状态；我们不应该尝试从外部传入对象或在视图的初始化方法中创建对象并将其分配给 state object 属性。这些方法行不通的原因，和我们在讨论 `@State` 的时候完全相同：在视图的初始化方法运行的时候，视图还不具有身份标识。作为经验法则，就像在计数器示例中所做的那样，如果我们不能在声明属性的同时就为该属性分配初始值，那么我们就应该使用 `@StateObject`：

```
struct Counter: View {  
    @StateObject private var model = Model()  
    ...  
}
```

在上面这个简单的 model 例子中，`@StateObject` 的实现和使用 `Observable` 的 `@State` 的实现几乎是相同的。但是，`Observable` 的实现是在属性层级上追踪变更，而 `@StateObject` 则是在对象层级上进行追踪。另外，`@StateObject` 的初始化方法接受一个 autoclosure，而 `@State` 属性则是在每次视图初始化时计算其初始值。

Observed Object

`@ObservedObject` 属性包装器要比 `@StateObject` 简单得多：它没有初始值的概念，也不需要多次渲染之间维持被观察对象的存在。它所做的事情，就只有

订阅这个对象的 `objectWillChange` publisher，并且在这个 publisher 发出事件时重新渲染视图。这些特性决定了，当我们 (把 iOS 17 之前的版本作为目标平台) 想要明确地从外部将对象传递到视图内部时，`@ObservedObject` 是唯一正确的工具。这和一个普通属性中的 `Observable` 对象是等价的。

下面的例子中我们用 `@ObservedObject` 重写了计数器的例子：

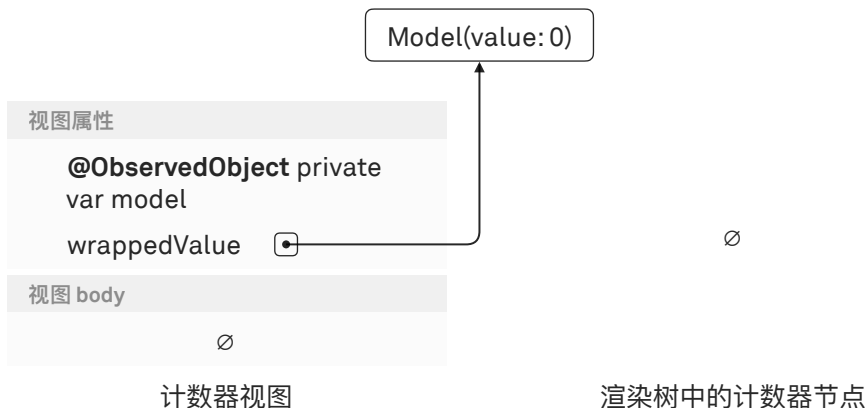
```
final class Model: ObservableObject {
    @Published var value = 0
    static let shared = Model()
}

struct Counter: View {
    @ObservedObject var model: Model
    var body: some View {
        Button("Increment: \(model.value)") {
            model.value += 1
        }
    }
}

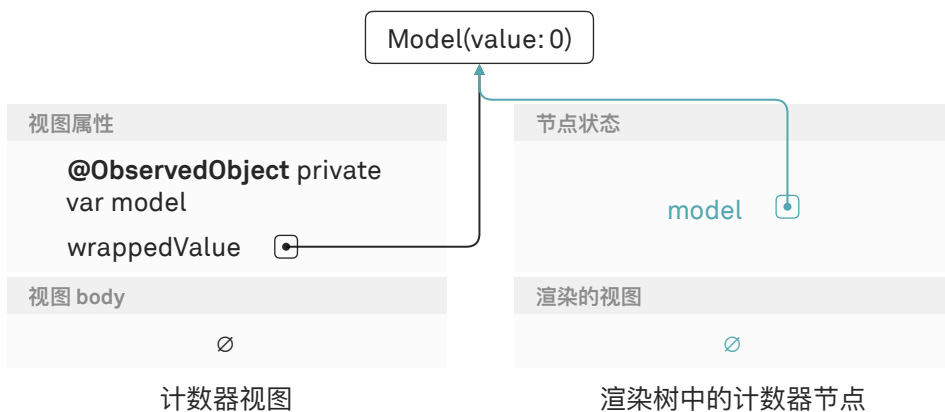
struct ContentView: View {
    var body: some View {
        Counter(model: Model.shared)
    }
}
```

在计数器视图中，我们使用 `@ObservedObject` 属性替换了 `@StateObject` 属性。现在，就像是 `ContentView` 里做的那样。我们可以从外部传入 `model` 对象了。当计数器被首次创建时，发生的事情是：

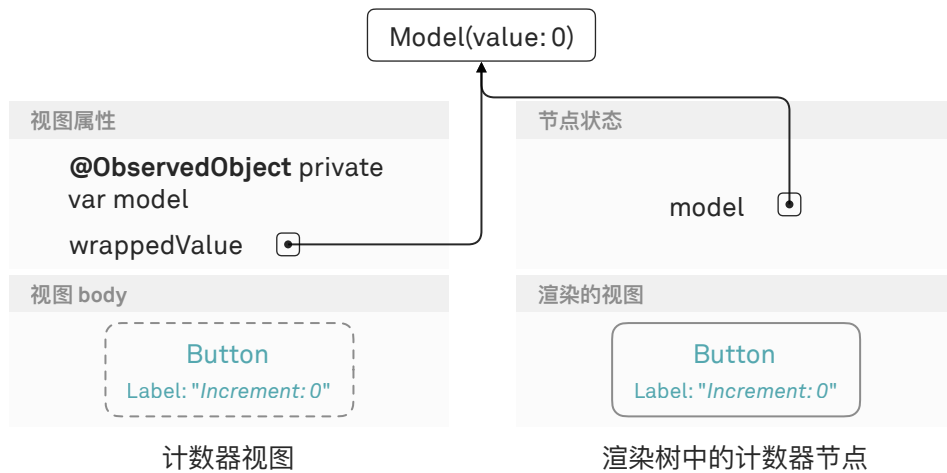
步骤 1 `ObservedObject` 的 `wrappedValue` 指向我们传入计数器视图的 `model` 实例。



步骤 2 创建渲染节点。和 `@StateObject` 不同的是，这个节点引用的 `model` 就是我们传入计数器视图的那个外部 `model` 对象。



步骤 3 计数器视图的 `body` 被执行，按钮视图在此时被构建。在 `body` 中，按钮的标题使用的是指向外部 `model` 对象的 `ObservedObject` 的引用。接下来，渲染节点得到更新，以反映新的计数器视图树中的内容。



当然，在这个特殊的例子中，我们可以直接在计数器视图中这样写：

```
struct Counter: View {
    @ObservedObject var model = Model.shared
    ...
}
```

上面代码做的事情是完全一样的，因为之前我们从外部传入的一直都是相同的 `Model.shared` 实例。但是，我们可以扩展一下这个例子，让它能够支持基于一些其他状态，来传入不同的 `model` 对象。下面是一个简单的示例，它定义了针对不同房间的人数计数器：

```
final class CounterModel: ObservableObject {
    @Published var value = 0
}

final class Model {
    static let shared = Model()
    var counters: [String: CounterModel] = [:]

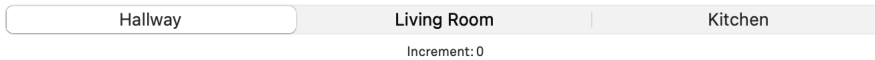
    func counterModel(for room: String) -> CounterModel {
        if let m = counters[room] { return m }
    }
}
```

```

    let m = CounterModel()
    counters[room] = m
    return m
  }
}

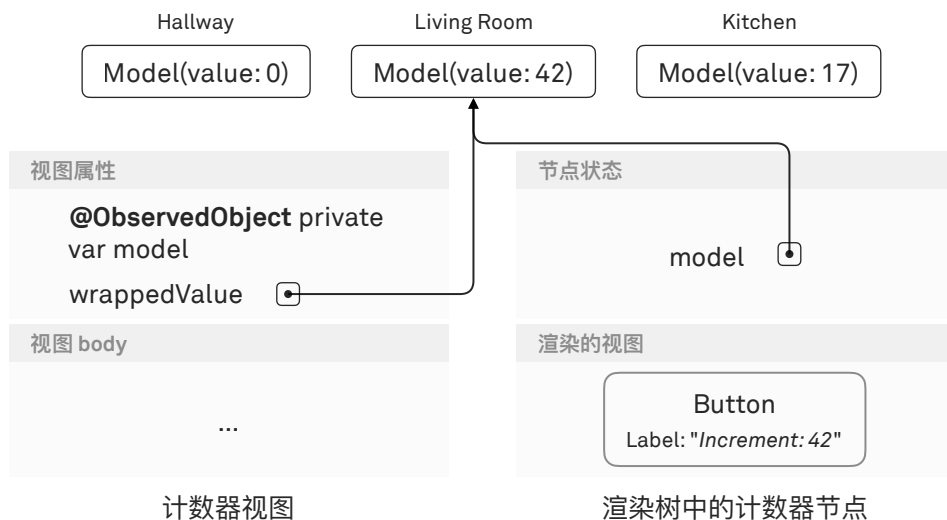
struct ContentView: View {
    @State private var selectedRoom = "Hallway"
    var body: some View {
        VStack {
            Picker("Room", selection: $selectedRoom) {
                ForEach(["Hallway", "Living Room", "Kitchen"], id: \.self) { value in
                    Text(value).tag(value)
                }
            }
            .pickerStyle(.segmented)
            Counter(model: Model.shared.counterModel(for: selectedRoom))
        }
    }
}

```

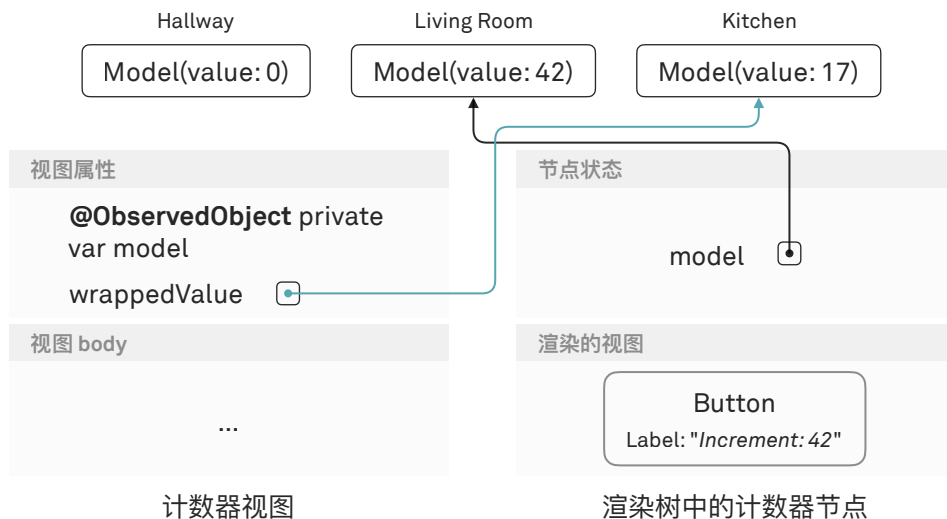


我们可以使用 `Picker` 来切换房间，从共享的 `model` 中取得正确的 `CounterModel`，然后将它传递到计数器视图中。共享 `model` 负责管理不同的 `CounterModel` 实例，并在视图更新时保持它们的存在。`@ObservedObject` 的唯一任务是订阅当前实例 (并取消对先前实例的订阅)，以便在更改发生时收到通知。请注意，当我们切换房间时，计数器视图在渲染树中的节点不会被重新创建，它只是观察了一个不同的对象。

举个例子，让我们想象一下目前计数器正在观察“Living Room” (客厅) 对象。当 `Counter` 被渲染时，视图树和渲染树都包含指向该对象的指针。要注意，只有渲染树才负责观察对象，视图树依然只扮演蓝图的角色：

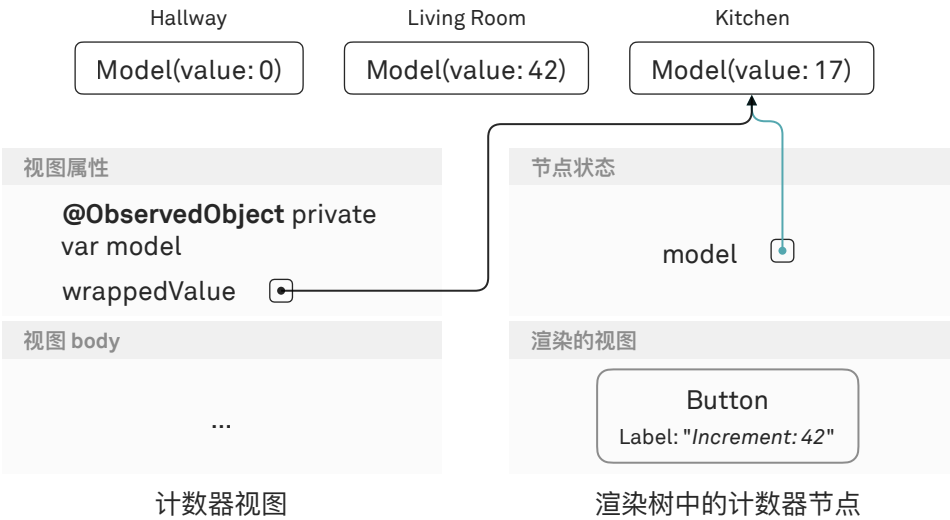


当用户选择“Kitchen” (厨房) 时，计数器视图的 `ObservedObject` 现在指向了另一个 `model` 对象。

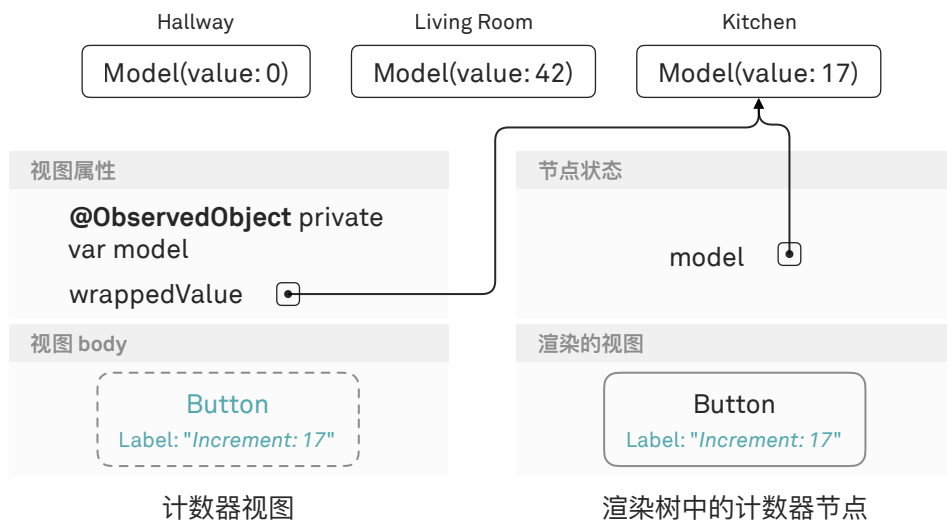


因此，渲染树也许要相应地进行更新。首先，渲染树将它所观察的 `model` 引用从

“Living Room” 更改为现在的 “Kitchen”。



最后，计数器视图的 body 被重新执行并构建出一棵新的视图树，渲染树也随之更新。



有意思的是，当 SwiftUI 刚推出时，并不存在 `@StateObject` 这个属性包装器，我们只能使用 `@ObservedObject`。尽管我们仍然能用它编写出相同的程序，但是通过引入 `@StateObject`，我们可以用更少的代码替代掉部分 `@ObservedObject` 的使用场景。

Binding

为了保持 app 状态的一致性，每个状态都必须拥有单一的**真实来源** (source of truth)。否则，表示同一件事的两个值可能会发生冲突。当编写组件时，通常我们会需要对一个值进行读写操作，但是我们并不需要知道该值实际的真实来源是什么。绑定 (Binding) 正是为此而存在的特性，我们在很多的 SwiftUI 内置的组件中都可以看到它。比如，Toggle 接受一个 Bool 的绑定值，TextField 接受一个 String 的绑定值，Stepper 接受一个 Strideable 的绑定值。

让我们将之前基于 `@State` 的计数器视图转换为一个接受整数类型绑定的组件：

```
struct Counter: View {
    @Binding var value: Int
    var body: some View {
```

```

    Button("Increment: \(value)") { value += 1 }
}
}

```

我们所要做的，只是将 value 属性上的 @State 标记更改为 @Binding，并删除之前在状态属性上使用的 private 关键字。切换到绑定的目的，就是使这些状态可以从外部传入。

为了探索绑定实际上的工作方式，我们退后一步，思考一下如果不使用 @Binding，我们要如何实现外部状态。计数器视图需要读取当前值，并且需要能更改当前值。我们可以通过使用一个普通的 value 属性，并为它添加另一个 setValue 属性来实现对值的修改：

```

struct Counter: View {
    var value: Int
    var setValue: (Int) -> ()
    var body: some View {
        Button("Increment: \(value)") { setValue(value + 1) }
    }
}

```

我们可以按照如下方式使用这个版本的 Counter 视图：

```

struct ContentView: View {
    @State private var value = 0
    var body: some View {
        Counter(value: value, setValue: { value = $0 })
    }
}

```

我们可以使用 Binding 类型，来将 value 和 setValue 这对属性进行合并：

```

struct Counter: View {
    var value: Binding<Int>
    var body: some View {
        Button("Increment: \value.wrappedValue") {

```

```

        value.wrappedValue += 1
    }
}
}

```

Binding 充当了 wrappedValue 的 getter 和 setter 的包装层。为了避免在代码中出现 wrappedValue，我们可以再次使用之前看到的模式，使用下划线的 Binding 属性和非下划线的计算属性，将 getter 和 setter 转发给 Binding 的 wrappedValue：

```

struct Counter: View {
    var _value: Binding<Int>
    var value: Int {
        get { _value.wrappedValue }
        set { _value.wrappedValue = newValue }
    }
    init(value: Binding<Int>) {
        self._value = value
    }
    var body: some View {
        Button("Increment: \(value)") { value += 1 }
    }
}

```

初始化方法唯一的任务就是暴露一个不带有下划线 _value 标签的参数，让 API 好看一些。使用 Counter 的方式如下：

```

struct ContentView: View {
    @State private var value = 0
    var body: some View {
        Counter(value: Binding(get: { value }, set: { value = $0 }))
    }
}

```

这里，我们使用 Binding(get:set:) 初始化方法手动创建了一个 binding，来展示 binding 作为 getter 和 setter 的包装的本质。

现在我们已经手写了这些代码，其实它也是可以简化的。对于计数器视图中 `_value` 和 `value` 属性的相关代码，它们正是 `@Binding` 属性包装器为我们生成的代码。在 `ContentView` 中，我们可以在变量名前使用一个美元符 `$value` 来替代掉手动创建 `binding` 的部分，这会我们自动创建一个 `binding`：

```
struct Counter: View {
    @Binding var value: Int
    var body: some View {
        Button("Increment: \(value)") { value += 1 }
    }
}
```

```
struct ContentView: View {
    @State private var value = 0
    var body: some View {
        Counter(value: $value)
    }
}
```

为了揭开 `$value` 语法的神秘面纱，我们来试着再次重写 `ContentView`，这次不使用 `@State` 和 `$value`：

```
struct ContentView: View {
    private var _value = State(initialValue: 0)
    private var value: Int {
        get { _value.wrappedValue }
        set { _value.wrappedValue = newValue }
    }
    var body: some View {
        Counter(value: _value.projectedValue)
    }
}
```

这里，我们可以看到 `$value` 其实就是 `_value.projectedValue` 的简写。美元符语法并不是 SwiftUI 特有的，它是 Swift 属性包装器 的一个特性，美元符就是访问属性包装器的 `projectedValue` 值的一个语法糖。

美元符语法不仅可以用在 @State 属性上，我们也可以在 @StateObject 属性或 @ObservedObject 属性上使用它。例如，我们可以将值存储在 ContentView 中的一个模型对象中：

```
final class Model: ObservableObject {  
    @Published var value = 0  
}
```

```
struct ContentView: View {  
    @StateObject var model = Model()  
    var body: some View {  
        Counter(value: $model.value)  
    }  
}
```

我们甚至可以创建一个计算属性，然后为它创建绑定。例如，基于 value 我们可以创建一个计算属性，将它的值限制在 0 到 10 之间：

```
final class Model: ObservableObject {  
    @Published var value = 0  
    var clampedValue: Int {  
        get { min(max(0, value), 10) }  
        set { value = newValue }  
    }  
}  
  
struct ContentView: View {  
    @StateObject var model = Model()  
    var body: some View {  
        Counter(value: $model.clampedValue)  
    }  
}
```

只要在 \$ 后的引用具有 getter 和 setter，我们就可以为它创建一个绑定。

Observable 和 Bindable

SwiftUI 的基于宏的对象观察模型在绑定方面提出了一个挑战：正如我们上面所看到的，用于构建绑定的 \$ 语法依赖于属性包装器的 projectedValue 值。由于在使用 @Observable 宏时，通常不再需要使用属性包装器，因此我们无法像使用 @ObservedObject 属性包装器时那样以相同的方式创建绑定。

为了解决这个问题，SwiftUI 引入了 Bindable 包装器，你可以将它作为属性包装器使用，也可以直接以内联的方式使用。例如，作为属性包装器使用时：

```
struct Counter: View {
    @Bindable var model: Model

    var body: some View {
        Stepper("\(model.value)", value: $model.value)
    }
}

struct ContentView: View {
    var model = Model.shared

    var body: some View {
        Counter(model: model)
    }
}
```

Counter 的基于成员的初始化方法接受一个普通的 Observable 对象，并在内部使用 Bindable(wrappedValue:) 初始化方法将该对象包装在一个 Bindable 值中。我们还可以通过使用更简洁的 Bindable(_) 初始化方法来以内联的方式创建这个 bindable 值，而不需要参数标签（这两个初始化方法都执行相同的操作）：

```
struct ContentView: View {
    var model = Model.shared
    var body: some View {
        Stepper("\(model.value)", value: Bindable(model).value)
    }
}
```

与 @StateObject 和 @ObservedObject 类似，@Bindable 的 projectedValue

允许我们通过动态成员查找的语法访问属性。例如，当我们重新编写所有 Bindable 语法糖时，上面的 Counter 视图如下所示：

```
struct Counter: View {
    var model: Model { _model.wrappedValue }
    var _model: Bindable<Model>

    init(model: Model) {
        _model = Bindable(wrappedValue: model)
    }

    var body: some View {
        Stepper("\(model.value)", value: _model.projectedValue[dynamicMember:
        \.value])
    }
}
```

在计数器视图的 body 中，当我们访问模型的 value 属性时，我们可以看到动态成员查找语法：model.value 被翻译为 _model.projectedValue[dynamicMember: \.value]。

视图更新和性能

SwiftUI 所带来的一个主要优势，是它可以自动地让 UI 和我们的状态保持同步。为了避免在任何状态更新时重新渲染整个视图树，SwiftUI 只在视图和它们所依赖的特定状态值 (或对象) 之间建立依赖关系。

当视图的 body 运行时，SwiftUI 知道在 body 执行期间访问了哪些状态属性，它会在两者间创建一份依赖关系的记录。当特定的状态值发生改变时，SwiftUI 知道具体哪个视图依赖了这个值，并且只会去重新执行该视图的 body。然后，通过比较状态变化前后的视图的值，它可以进而确定这个视图中的哪些子视图的 body 需要被重新执行。

正如我们所看到的，SwiftUI 会尽最大努力确保在某些状态发生变化时，只重新渲染那些视图树中绝对必须重绘的部分。然而，同样重要的是，我们也需要以这样

的方式编写代码，来避免抵消这些努力。

比如，要是我们将所有的状态都放在一个庞大的可观察对象中 (并使用“传统”的 `@StateObject` 或 `ObservedObject` 属性包装器)，那么在任何更改发生时，就算这个特定更改可能只会影响观察该对象的视图之中的一小部分，SwiftUI 都必须重新渲染所有视图。当性能成为问题时，将状态分解为较小的单元可以帮助我们实现更细粒度的视图更新。`@Observable` 宏在很大程度上缓解了一个问题，因为它将观察从对象层级转移到了对象的各个属性上。

另外也需要着重注意，只应该向子视图传递它实际所需要的数据。例如，假设我们有一个包含许多属性的大型结构体，某个视图只需要从这个结构体中取一个值，但我们还是把整个结构体层层向下传递，那么每次当这个结构体中有任何东西发生变化，这个视图都会被重新渲染。这种情况其实可以通过在视图上明确定义我们所需要的那一个属性来避免；同时，这样做也会给我们带来额外的好处：在预览中显示这个视图也会变得更加容易。因此，将较大的视图拆分为依赖于不同子状态的多个子视图是有意义的。

当我们遇到性能问题时，我们有好几种方法可以找出哪些视图的 `body` 正在被执行。第一种选择，是在视图的 `body` 中插入一条 `print` 语句：

```
var body: some View {  
    let _ = print("Executing <MyView> body")  
    ...  
}
```

匿名变量的赋值语句 `let _ = ...` 是必要的，因为视图构建器只接受类型为 `View` 的表达式。我们不能简单地调用 `print("...")`，因为 `print` 函数的返回值是 `Void`，这是视图构建器无法处理的情况。使用 `let _ = ...` 可以回避这个问题。

如果想更直观地看到重新渲染的视图，Peter Steinberger 提出过一个巧妙的技巧，可以对视图使用[随机背景颜色](#)。

如果你想寻找视图 `body` 被重新执行的原因，可以像这样，在视图的 `body` 里使

用 `Self._printChanges()` API:

```
var body: some View {  
    let _ = Self._printChanges()  
    ...  
}
```

同样地，我们需要使用匿名变量赋值的技巧，来绕过函数的返回值是 `Void` 这一限制。这条打印语句在终端中记录渲染发生的原因：

1. 如果视图是因为状态变化而发生的渲染，相关的状态属性的名字将会以下划线版本被记录下来。
2. 如果是视图值本身发生了变化，比如视图中某个属性的值改变了，会打印 `"@self"`。
3. 如果视图的身份标识发生了变化，会打印 `"@identity"`。在实践中，这一般意味着视图刚刚被插入到渲染树中。

最后，在 Instruments 工具中，也有一个 SwiftUI 的分析模板。我们可以使用它来诊断哪些视图 body 的执行频率要高于我们的预期。

属性包装器及其用途

我们经常遇到本章中所讨论到的几个属性包装器在 SwiftUI 代码中被过度使用的问题。第一条规则是，尽可能在视图中使用不带任何属性包装器的普通属性。如果我们只想要把一个值传递到视图中，这个视图不需要对该值进行写入，那么普通属性就足够了。

当视图需要对一个值 (而不是对象) 同时进行读写访问，并且它应该把这个值作为本地的私有视图状态 (也就是不能从外界传递进来) 时，应该使用 `@State`。而与之相对，如果这个视图需要读写一个不属于它自己的值 (也不关心这个值具体是储存在哪里) 时，使用 `@Binding`。

如果视图需要以对象的形式拥有状态，而且这个状态应该是本地私有的视图状态

(也就是不能从外界传入) 时，在 iOS 17 之前，我们使用 `@StateObject`；在 iOS 17 之后，使用 `@State` 和一个 `@Observable` 对象。不过，如果我们需要从外部传入的是一个对象，则在 iOS 17 之前使用 `@ObservedObject`，在 iOS 17 之后使用普通的属性。

按照我们的经验，如果属性无法直接在它被声明的同时就进行初始化，那么就不应该使用 `@State` 或者 `@StateObject`。这种时候，我们也许应该使用 `@Binding`，`@ObservedObject` 或者是一个普通属性的 `@Observable`。

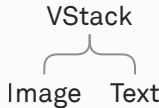
布局

4

SwiftUI 的布局算法非常直接：父视图向它的子视图提供一个建议尺寸 (proposed size)，子视图基于这个建议决定自己的大小，并将结果汇报给父视图。然后，父视图在自己的坐标系中放置这个子视图。从本质上讲，布局过程的目标是为每个视图提供位置和大小。

但是，当我们要努力去搞明白为什么视图树会按照某个方式进行渲染的时候，布局算法的行为理解起来可能就不是那么简单了。

首先需要记住的是，SwiftUI 的布局算法是沿着视图树从上向下进行处理的。因此，了解我们的视图构建器代码所产生的视图树，将会是重中之重 (有关视图树的细节，可以参看之前的相关章节)。让我们通过一个例子，来更好地理解布局算法是怎样应用于真实视图树的：

Code	View Tree
<pre>VStack { Image(systemName: "globe") Text("Hello, World!") }</pre>	

在这个例子里，VStack 是根视图，所以它会接收到整个屏幕的安全区域 (safe area) 作为建议尺寸。VStack 在决定自己的尺寸时，它会以递归的方式先向它的子视图提出尺寸建议。Image 按照地球 (globe) 符号的尺寸汇报自己的尺寸，Text 则基于建议尺寸和它需要渲染的字符串来报告自己的尺寸 (在稍后，我们会深入研究 Image 和 Text 到底是如何确定它们尺寸的细节)。然后，VStack 将两个子视图垂直排列在一起，并在两个视图之间插入默认的间距。VStack 在计算自己的大小时，会把子视图的框架 (frame) 合并，然后把尺寸报告回去。

正式一些的话，我们可以把布局算法描述如下：

1. 父视图向它的子视图提供一个建议尺寸。
2. 子视图基于这个建议尺寸决定自己的尺寸，如果这个子视图还有自己的子视

图，从步骤 1 开始递归。

3. 子视图将它的尺寸报告给父视图。
4. 父视图负责将子视图摆放在合适的位置。

需要注意的是，步骤 3 中子视图报告的大小是子视图的最终尺寸，父视图不能单方面改变这个大小。父视图可以返回到步骤 2，并提出另一个建议尺寸，但最终子视图会根据自己的需要，参考建议尺寸来确定自己的大小。

尽管在这个过程中涉及的 API 不属于公开的 View 协议的一部分，但我们可以设想每个视图都有一个类似这样的方法：

```
extension View {  
    func sizeThatFits(in: ProposedViewSize) -> CGSize {  
        ...  
    }  
}
```

ProposedViewSize 类型在 iOS 16 和 macOS 13 中被公开了。它被用在 Layout 协议中，我们会在[进阶布局](#)一章中进行讨论：

```
struct ProposedViewSize {  
    var width, height: CGFloat?  
}
```

ProposedViewSize 和 CGSize 之间的区别在于，在 ProposedViewSize 中 width 和 height 这两个属性都是可选值。在建议尺寸中使用 nil 意味着视图可以在这个方向上使用它的**理想尺寸** (ideal size)。每种视图的理想尺寸各不相同，我们会在下面详细讨论。

从 API 的角度，我们可以将布局算法表达如下：

1. 父视图调用它的子视图上的 sizeThatFits 方法，并传入建议尺寸。

- 2. 子视图根据建议尺寸，来确定自己的大小。如果这个子视图还拥有自己的子视图，也许会再调用它们上的 `sizeThatFits`。有一些视图则会完全忽视掉建议尺寸 (比如，`Image` 默认情况下就是如此)，也有一些视图会直接把建议尺寸作为它们的实际尺寸返回 (比如 `Rectangle`)。当一个视图返回的尺寸就是建议尺寸时，我们就说它**接受**了这个建议尺寸。
- 3. 子视图将它自己的尺寸通过 `sizeThatFits` 的返回值，汇报给父视图。
- 4. 父视图基于它自己的对齐方式和子视图的对齐参考线 (alignment guide)，来放置子视图 (在本章稍后，我们会详细介绍对齐的工作方式)。

下面是一个更详细的例子，包括了所有的步骤以及涉及到的视图的具体尺寸：

Code	View Tree	Preview
<pre>Text("Favorite") .padding(10) .background(Color.teal)</pre>		

为了简化这个例子，我们会假设最外层窗口的安全区域尺寸是 320×480。

- 1. 系统将 320×480 作为建议尺寸，提交给 `background`。
- 2. `background` 将同样的 320×480 作为建议提交给它的主要子视图 `padding`。
- 3. `padding` 从每个边减去 10，然后将新的建议尺寸 300×460 提供给 `Text`。
- 4. `Text` 返回它的文本尺寸 51×17。
- 5. `padding` 为每个边加上 10，把自己的尺寸汇报为 71×37。

6. background 把带有内边距的文本的尺寸 (71×37) 作为建议提交给次要子视图 Color。
7. Color 接受这个提议，返回 71×37 作为自己的尺寸。
8. background 把主要子视图的 71×37 作为自己的尺寸返回给系统。

有了对 SwiftUI 的布局系统遍历视图树方式的理解，我们下面来看看 SwiftUI 内置的部分视图里所特有的布局行为。

一般来说，在尝试理解视图是如何布局时，我们最喜欢的方式是给视图添加边框。此外，我们还可以通过几何读取器 (geometry reader，它是 SwiftUI 中的一种特殊视图) 来为视图添加一个覆盖层，以此渲染出视图的尺寸。不过按我们的经验来说，边框就已经几乎就足够帮我们调试所有的布局问题了。

虽然一般性的布局算法用一两句话就能概括，但是要理解所有内置视图特定的布局行为绝对算不上简单的任务。

但就算如此，长期来看我们还是应该尽量避免对 SwiftUI 的工作方式进行太多猜测。

在这部分内容中，我们会介绍许多 SwiftUI 布局相关的视图和视图修饰器，并解释它们确定自己大小的方式是什么，它们会向自己的子视图提供的建议尺寸是什么，以及决定它们的理想尺寸的因素是什么等问题。我们将从叶子视图 (也就是那些没有子视图的视图) 开始，然后研究视图修饰器，最后讨论容器视图。

叶子视图

Text

默认情况下，Text 视图会去适应任意的建议尺寸，设法让自己适配 (不超过) 这个尺寸。Text 使用多种策略，按照以下顺序来实现这一目标：将文本分成多行 (英文内容按单词换行)，单词内换行 (使用连词符号)，截断文本，最后裁剪文本。Text 始终都会返回它所需要渲染的内容的大小，这个尺寸在宽度上一定小于或等于建议的宽度，在高度上除非提议的是 0×0，否则至少是一行的高度。换句话

说，Text 的宽度可以从零到完整渲染内容所需的宽度之间的任意值。

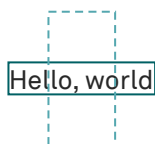
以下是 Text("Hello, World!") 在不同建议尺寸下的渲染示例。虚线矩形代表建议尺寸，实线矩形代表报告的尺寸：



在 Text 上，我们可以使用一系列修饰器来改变它的行为：

- `.lineLimit(_ number:)` 允许我们指定渲染时的最大行数，不论在建议尺寸的垂直方向上空间足够与否，文本都不能超过这个行数。指定为 `nil` 代表没有行数限制。
- `.lineLimit(_ limit:reservesSpace:)` 允许我们指定渲染时的最大行数，并且让我们有机会选择是否要在建议尺寸足够的情况下，让额外的空格去将行数补充到这个最大行数。
- `.truncationMode(_ mode:)` 允许我们指定文本截断应该发生的位置。
- `.minimumScaleFactor(_ factor:)` 允许我们指定 Text 在需要缩小字体以适应建议尺寸时，所能够使用的最小比例。

如果我们将 `.fixedSize()` 应用于 Text，它将使用理想尺寸，因为 `fixedSize` 向 Text 所建议的尺寸是 `nil×nil`。Text 的理想尺寸是在不换行和不截断的情况下呈现内容所需的尺寸。下面是 `Text("Hello, World!").fixedSize()` 在不同建议尺寸下的渲染情况：



建议尺寸: 25×50
报告尺寸: 55×12



建议尺寸: 50×50
报告尺寸: 55×12



建议尺寸: 100×50
报告尺寸: 55×12

正如所见，实线矩形 (报告尺寸) 在虚线矩形 (建议尺寸) 之外进行了渲染，`fixedSize` 修饰器会确保子视图 (这里的 `Text`) 始终以其理想尺寸进行渲染，而不考虑建议尺寸。这是 SwiftUI 中父视图不能期望子视图一定会遵守建议尺寸的一个很好的例子。最终而言，子视图才能决定自己的尺寸。

形状

大部分内置的形状类型 (`Rectangle`, `RoundedRectangle`, `Capsule`, 以及 `Ellipse`) 接受从零到无限的任意大小的建议尺寸，并且会填充所有可用空间。`Circle` 是一个特例：它会按照建议尺寸的短边作为直径进行适配，然后将圆形的实际尺寸进行汇报。如果我们使用 `nil` 来对形状进行尺寸建议 (比如，我们把形状包装到 `fixedSize` 里)，它将使用 10×10 这个默认尺寸。

除了内置的形状，我们也会实现自定义形状。这是 SwiftUI 中目前为数不多的可以根据建议尺寸来编写自己的逻辑去计算视图尺寸的地方之一。作为示例，我们来构建一个书签形状：

Code	Preview
<pre> struct Bookmark: Shape { func path(in rect: CGRect) -> Path { Path { p in p.move(to: rect.topLeading) p.addLines([rect.bottomLeading, rect.init(x: 0.5, y: 0.8), rect.bottomTrailing, rect.topTrailing, rect.topLeading]]) p.closeSubpath() } } } </pre>	

为了让编写形状简单一些，我们使用了下面这个 CGRect 的扩展，它接受一个 UnitPoint 并返回在矩形中相应位置的点：

```

extension CGRect {
  subscript(_ point: UnitPoint) -> CGPoint {
    CGPoint(x: minX + width*point.x, y: minY + height*point.y)
  }
}

```

假设这个形状应该始终以 2×3 的宽高比进行渲染，我们当然可以在使用这个形状的地方添加上 .aspectRatio 修饰器，但是这个 API 的其他使用者很可能会意外地以不同的宽高比来渲染我们的书签形状。为了确保书签形状始终拥有 2×3 的宽高比，我们可以实现 sizeThatFits。对于形状来说，这个方法的默认实现将返回提议的尺寸。我们可以覆盖这个方法，使用硬编码的方式把宽高比设为 2×3：

```

func sizeThatFits(_ proposal: ProposedViewSize) -> CGSize {
  var result = proposal.replacingUnspecifiedDimensions()
  let ratio: CGFloat = 2/3
  let newWidth = ratio * result.height

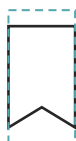
```

```

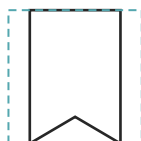
if newWidth <= result.width {
    result.width = newWidth
} else {
    result.height = result.width / ratio
}
return result
}

```

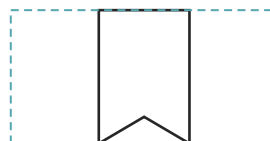
因为建议尺寸的里的属性可能是 nil，因此我们首先要调用 `replacingUnspecifiedDimensions` 来获取一个 `CGSize`。然后，我们计算形状的宽度和高度，使其宽高比为 2x3，并适配到建议尺寸中去。



建议尺寸：25×50
报告尺寸：25×37



建议尺寸：50×50
报告尺寸：33×50



建议尺寸：100×50
报告尺寸：33×50

我们也可以使用 `sizeThatFits` 方法来更好地理解容器视图。如果我们在一个形状中添加一些日志，来记录下建议尺寸和报告尺寸，然后在 `HStack` 中使用这个形状，我们就能在控制台中看到 `HStack` 向其子视图建议的尺寸内容。要注意，`sizeThatFits` 是 `Shape` 协议的一部分，而不是 `View` 协议的一部分。

颜色

当直接将颜色当作视图使用时，比如 `Color.red`，从视图布局的视角来看，它的行为和 `Rectangle().fill(...)` 是一样的。

但是有一个特例：如果我们将一个颜色放在与非安全区域有接触的背景中，颜色

将神奇地“渗入”到非安全区域中去。虽然这个行为并不会影响布局，但我们还是想在这里提一下，因为我们大概迟早都会遇到这个问题。如果我们想要避免这种情况，可以使用 `.background` 上的 `ignoresSafeAreaEdges` 参数，或者使用 `Rectangle().fill(...)` 来取代 `Color`。

图片

默认情况下，`Image` 视图会报告一个固定值：也就是它所持有的图片的尺寸。一旦我们在 `Image` 上调用 `.resizable()`，这个视图就会完全灵活可变：`Image` 将会接受任意的建议尺寸，并将它报告回去，同时将图像压缩或者拉伸到这个尺寸。在实践中，实际上任何一个 `resizable` 的图片都会和 `.aspectRatio(contentMode:)` 或 `.scaleToFit()` 修饰器一同使用，以避免图片产生变形。我们会在后面一节探讨这些修饰器的行为。

`resizable` 修饰器拥有两个参数。第一个参数允许我们指定 `capInsets`，它用来指示 (边缘上那些) 不应该被调整尺寸的部分。第二个参数则让我们可以指定调整尺寸的模式：图片是应该拉伸还是平铺。

分隔线

在水平堆栈 (比如 `HStack`) 外使用 `Divider` 时，它接受任意的建议宽度，并将分隔线自身的默认高度进行汇报。在水平堆栈内部使用时，分隔线会接受建议的高度，并报告分隔线的宽度。如果建议值是 `nil`，那么它将根据所处环境，在可变轴上使用默认尺寸 10。

Spacer

在水平堆栈和垂直堆栈之外，`Spacer` 接受从最小长度到无限大的所有建议尺寸。然而，当 `Spacer` 放置在水平或垂直堆栈中时，其行为会发生变化：在垂直堆栈中，`Spacer` 接受从其最小长度到无限大的任何高度，但报告的宽度为零。在水平堆栈中，它的行为相同 (只是轴互换)。除非使用 `Spacer` 初始化方法中的 `minLength` 参数指定一个长度，否则 `Spacer` 的最小长度会是默认填充 (`padding`) 的长度，

Spacer 的一个常见场景是用于对齐视图。例如，我们经常看到下面这样的代码：

```
HStack {  
    Spacer()  
    Text("Some right-aligned text")  
}
```

我们推荐改用一個帶有對齊的可變 frame (我们会在下一节详细讨论可变 frame)：

```
Text("Some right-aligned text")  
    .frame(maxWidth: .infinity, alignment: .trailing)
```

我们倾向于使用这个方案，而不是去用 HStack 和 Spacer，这是因为 HStack 的方案中存在一个不太明显的边缘情况：Spacer 是有最小长度的 (等于默认的间距)，这样一来，因为 Spacer 会占用 HStack 所收到的部分建议尺寸宽度，文本可能会在必要之前就被更早地换行或截断。

因为 Mac app 默认情况下就可以调整窗口大小，所以通过将上面两个视图 (基于 HStack 的方案以及使用 frame 的方案) 放到 Mac app 的内容视图的 VStack 中，我们就可以通过调整窗口大小来快速看到它们行为上的差异。

视图修饰器

视图修饰器总是把已经存在的视图包装到另一层中去：修饰器会变为它所作用的视图的父视图。虽然 SwiftUI 中有 .modifier API 来让一个值遵守 ViewModifier 协议，但是 SwiftUI 自己内置的修饰器全都是以 View 的扩展的形式暴露出来的 (对于我们自己的视图修饰器来说，这也是一种很好的做法)。在本节中，我们会介绍一些影响布局的视图修饰器。

Padding

.padding 修饰器使用它收到的内边距值来修改建议尺寸，它会从建议尺寸的对应边上减去这个边距值。修改后的尺寸将被提供给 .padding 的子视图 (也就是这个修饰器所作用的视图)。当子视图将其尺寸报告回来时，padding 再将边距值加回到对应的边上，然后把扩展后的尺寸作为自己的尺寸进行报告。

.padding 有两个有用的变体：

- .padding(_ inset:) 让我们能使用 EdgeInsets 值，在单次调用中就给不同的边指定不同的边距值。
- .padding(_ edges:_ length:) 让我们能够为一组边 (比如分别代表左右两边和上下两边的 .horizontal 或者 vertical, 使用起来特别方便) 指定一个边距值。

当我们没有指定具体值，而是使用不带任何参数的 .padding() 时，则会使用当前平台的默认边距值。

固定框架

固定框架 (fixed frame) 修饰器 .frame(width:height:alignment) 所拥有的布局行为非常简单：它把所指定的尺寸原封不动地提供给子视图，同时不管子视图所报告的尺寸是多少，总是把这个指定的尺寸作为自己的大小进行汇报。换句话说，固定框架是一个尺寸完全固定的不可见的视图，我们可以用它来将指定的尺寸提供给另一个视图。

如果我们只指定宽度或者高度，而把另一个参数设定为 nil 或者完全省略掉它，那么固定框架将不会对该维度产生影响。向固定框架提供的建议尺寸在该维度上将被直接转发给子视图，固定框架在这个维度上也会将子视图所报告的尺寸作为自己的尺寸。

随着对动态类型 (dynamic type) 和各种屏幕尺寸的支持，在产品代码中使用固定框架修饰器实际上是很罕见的。除非绝对必要 (比如某些设计元素本身不能也不应该支持缩放)，否则最好避免在固定框架中去硬编码一些魔术数字 (magic number)。

灵活框架

灵活框架 (flexible frame) 的 API 支持非常多的参数：我们不仅可以为框架的宽和高提供最小、最大和理想值，还可以为它指定对齐方式。灵活框架的行为并不是非常直观，但是它们是很有用的工具，学习它们的行为十分重要。我们现在先忽略掉理想的宽度和高度参数，只考虑最大值和最小值这两个定义边界的参数。

最大值和最小值的边界会被灵活框架使用两次：一次在它提供尺寸给子视图时，另一次在决定框架自己要报告的尺寸时。

其中“传入途中”发生的夹断 (clamp)，也就是计算需要提供给框架的子视图的尺寸，是相对比较简单。灵活框架会把所获得的建议尺寸，按照指定的最小值和最大值进行限制：如果传入的尺寸没有落在最小和最大值之间，则强制使用最小和最大值作为建议尺寸并提供给框架的子视图。

当子视图的尺寸被计算出来后，灵活框架将基于它所收到的建议尺寸，再次应用设定边界值，来确定自己的尺寸。但是，这时候对于缺少的边界值 (比如，假设我们只指定了 `minWidth`，但是没有指定 `maxWidth`)，将使用子视图所报告的尺寸来替代。再之后，这个被替换过的建议尺寸将被设定的边界再进行一次夹断，结果被作为框架的尺寸进行报告。

在实践中，这意味着如果我们只指定了 `minWidth`，那么灵活框架的宽度至少会是 `minWidth`，而最多会是它的子视图的宽度。如果我们只指定了 `maxWidth`，那么灵活框架的宽度至少是它的子视图的宽度，而最多则为 `maxWidth`。高度也同理。

这一行为促成了 SwiftUI 中的两种常见书写模式，一开始它们可能很难理解，但是多用几次，就能轻车熟路。

第一种模式，可以这样来使用灵活框架：

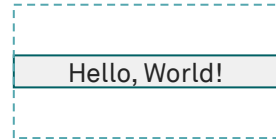
```
Text("Hello, World!")  
    .frame(maxWidth: .infinity)  
    .background(.quaternary)
```



建议尺寸: 25×50
报告尺寸: 25×48



建议尺寸: 50×50
报告尺寸: 50×24



建议尺寸: 100×50
报告尺寸: 100×12

`.frame(maxWidth: .infinity)` 模式确保了灵活框架的宽度至少和被建议的宽度相同，如果子视图的宽度要比框架接受的建议宽度还宽的话，则使用子视图的宽度。通常，这个模式被用来创建占据整个可用宽度的视图。

我们可以把这个模式提取成一个 `View` 上的扩展：

```
extension View {  
    func proposedWidthOrGreater() -> some View {  
        frame(maxWidth: .infinity)  
    }  
}
```

另一种常见模式如下：

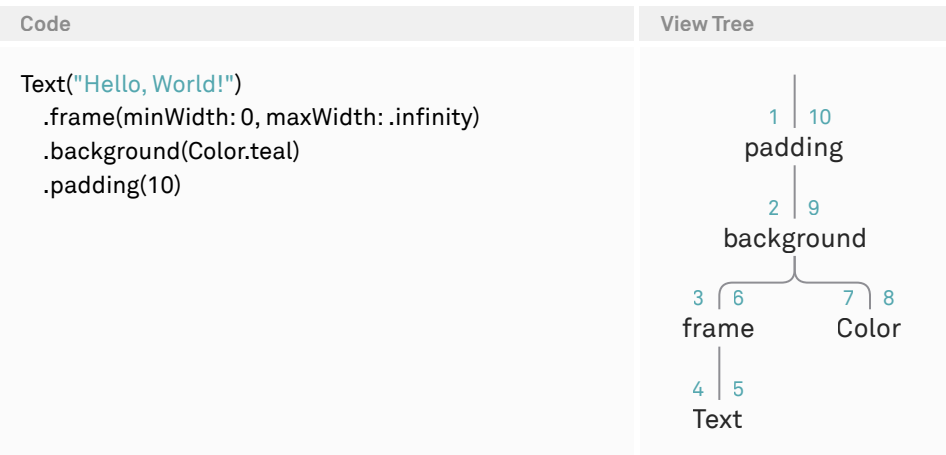
```
Text("Hello, World!")  
    .frame(minWidth: 0, maxWidth: .infinity)  
    .background(Color.teal)
```

这可以确保框架无视它的子视图的尺寸，框架报告的宽度将总是它接受到的建议宽度。同样，我们可以为这个模式编写一个辅助函数：

```
extension View {  
    func acceptProposedWidth() -> some View {  
        frame(minWidth: 0, maxWidth: .infinity)  
    }  
}
```


这两种模式同样可以作用在高度上。

使用上面的例子，让我们来看看当下面这样的视图渲染到 320×480 的屏幕上时，会发生什么：



1. 系统向 padding 提出 320×480 作为建议尺寸。
2. padding 向 background 建议 300×460。
3. background 将同样的 300×460 建议给它的主要子视图 (也就是 frame)。
4. frame 把同样的 300×460 提供给它的子视图 Text。
5. Text 汇报它的尺寸为 76×17。
6. frame 的宽度变为 $\max(0, \min(.infinity, 300)) = 300$ 。注意，这里的 0 和 .infinity 值都是灵活尺寸的参数里指定的值。
7. background 把灵活框架的尺寸 (300×17) 提供给次要子视图 (Color)。
8. 颜色视图接受这个建议，并将它作为自己的尺寸汇报。
9. background 将它的主要子视图的尺寸 (300×17) 进行汇报。

10. padding 在每条边上加上 10，最终它的尺寸为 320×37。

灵活框架是 SwiftUI 中唯一可以明确指定理想尺寸的 API。所谓理想尺寸，就是如果某个维度上收到的建议尺寸为 nil 时将采用的尺寸。如果指定了 `idealWidth` 或 `idealHeight` 参数，在灵活框架收到包含 nil 的建议尺寸时 (通常来源于 `fixedSize` 修饰器)，它会把指定的理想尺寸提供给子视图，并且这个尺寸也会作为框架自身的尺寸进行报告，而再不考虑其子视图的尺寸。

宽高比

`aspectRatio` 修饰器在处理完全灵活的视图时非常有用。例如，我们可以使用以下代码绘制一个宽高比为 4/3 的矩形：

`Color.secondary`

`.aspectRatio(4/3, contentMode: .fit)`



建议尺寸： 25×50
报告尺寸： 25×18



建议尺寸： 50×50
报告尺寸： 50×37



建议尺寸： 100×50
报告尺寸： 66×50

这个 `aspectRatio` 修饰器将会计算出一个能够适配进建议尺寸的宽高比为 4/3 的矩形，然后将它作为建议尺寸提供给子视图。对于父视图，它总是把子视图的尺寸汇报上去，而不去理会建议尺寸或者设定的比例。

为了让这个例子具体一些，我们来假设一个 200x200 的建议尺寸。`aspectRatio` 通过计算，得到在这个建议尺寸下满足 4/3 宽高比的最大矩形，此处是 200×150。它把这个尺寸建议给 `Color`，后者会直接接受建议。最后，`aspectRatio` 把 200×150 作为它自己的尺寸进行汇报。

让我们来看看，如果把 `contentMode` 从 `.fit` 替换为 `.fill` 时，会发生什么：

```
Color.secondary
```

```
.aspectRatio(4/3, contentMode: .fill)
```

我们还是从 200x200 的建议尺寸开始。基于这个尺寸，`.aspectRatio` 将会计算一个 4/3 宽高比，能够覆盖住整个建议尺寸的最小矩形。本例中这个矩形的尺寸约为 266.6x200。它把这个尺寸建议给 `Color`，后者会直接接受建议。最后，`aspectRatio` 把 266.6x200 这个子视图的尺寸作为它自己的尺寸进行汇报。

`.aspectRatio` 最常见的使用方式是配合图像。我们上面讨论过，当我们在图像上调用 `.resizable` 让它具有灵活性时，图像会尝试适配到任意尺寸并发生变形。但是大多数情况下，我们会希望图像在被缩放以适应或填充可用空间的同时，还能保持其自身的宽高比。在不知道底层图像的宽高比的时候，我们可以编写这样的代码：

```
Image("header")
```

```
.resizable()
```

```
.aspectRatio(contentMode: .fit)
```

`.scaleToFit` 和 `.scaleToFill` 修饰器分别是 `.aspectRatio(contentMode: .fit)` 和 `.aspectRatio(contentMode: .fill)` 的简写形式。

因为我们没有指定具体的宽高比，`.aspectRatio` 将会 (通过提供 `nilxnil` 尺寸) 探测子视图的理想尺寸，并依据这个尺寸计算宽高比。假设建议尺寸为 200x200 且图像的理想尺寸为 100x30，这个例子的布局步骤将如下所示。请注意，`.resizable` 修饰器并不会为视图树添加额外的层级，它做的是对 `Image` 进行修改：


Code	View Tree
<pre>Image("header") .resizable() .aspectRatio(contentMode: .fit)</pre>	<pre> 1 6 aspectRatio 2, 4 3, 5 Image </pre>

1. 200×200 的建议尺寸被提供给 aspectRatio。
2. aspectRatio 向图像建议 nil×nil。
3. 图像报告自己的理想尺寸 100×30。
4. aspectRatio 将一个宽高比为 100/30 的矩形适配到 200×200 中，得到 200×60，并将这个尺寸提供给图片。
5. 图片将自己的尺寸报告为 200×60。
6. aspectRatio 将子视图的尺寸 200×60 作为自己的尺寸进行报告。


要注意，aspectRatio 视图最终不一定会具有我们指定的宽高比。虽然修饰器会向其子视图提议一个具有指定宽高比的尺寸，但和其他情况一样，子视图才是根据建议决定自己尺寸的角色。如果子视图无法灵活地将自身适应到指定的宽高比，那么 aspectRatio 修饰器自身也不会满足这个宽高比，因为它的尺寸取决于其子视图的尺寸。

覆盖层和背景

覆盖层 (overlay) 和背景 (background) 是 SwiftUI 中最有用的几个修饰器中的成员了。在布局方面，两者的工作方式完全相同。唯一的区别在于 overlay 会把次要视图绘制在主要视图的前面，而 background 则是把次要视图绘制在主要视图的后方。比如，要是我们想在一些文本后方绘制背景的话，可以这样做：

Code	View Tree	Preview
<pre>Text("Hello") .background(Color.teal)</pre>	<pre>.background ├── Text └── Color</pre>	

这种工作方式十分优雅：background 首先确定它的主要子视图 (上例中的 Text) 大小，然后将这个主要子视图的尺寸作为建议尺寸提供给次要子视图。这样，青绿色背景的大小就和 Text 视图大小一致了。我们也可以修改一下 Text，让它包含一些边距：

Code	View Tree	Preview
<pre>Text("Hello") .padding(10) .background(Color.teal)</pre>	<pre>.background ├── .padding │ └── Text └── Color</pre>	

现在，我们可以看到青绿色背景的大小等于在文本大小基础上加上每边扩展 10 point。

背景和覆盖层都不会影响其主要子视图的布局，两者所报告的尺寸始终都是主要子视图的尺寸。举个例子，我们可以创建一个高亮 (highlight) 修饰器，在视图后方绘制一个略大于视图本身的橙色背景：

```
extension View {
    func highlight(enabled: Bool = true) -> some View {
        background {
            if enabled {
                Color.orange
                .padding(-3)
            }
        }
    }
}
```

```

    }
}
}

```

我们切换 `enabled` 时，这个背景不会导致其他周围的视图发生移动：
`background` 修饰器始终保持为其主要子视图的大小，而不依赖于背景中的视图 (也就是次要子视图)。

在 `overlay` 或者 `background` 中，当次要视图是包含有多个视图的一系列视图时，这些视图会被放到一个隐式的 `ZStack` 中。

固定尺寸

`fixedSize()` 修饰器不考虑它自己收到的建议尺寸，转而会向它的子视图建议 `nil` 的宽度和高度。这样，子视图将使用自己的理想尺寸。它具有一个重载方法 `fixedSize(horizontal:vertical:)`，可以允许我们使子视图只在某一个维度上变为其理想尺寸 (例如，将宽度建议为 `nil`，但不改变高度)。


`fixedSize` 最常见的用例之一，是配合 `Text` 视图来显示文本。如上所述，`Text` 会尝试各种办法，来确保把自己渲染在建议尺寸之中 (比如截断或者换行)。想要规避这种行为，可以向 `Text` 提供 `nil×nil` 的建议尺寸，这样文本就会变为理想尺寸。注意，根据视图树的不同，这么做可能导致文本的绘制超出边界。例如：




在上面的示例中，我们可以看到，文本绘制在了 `frame` 修饰器 (它用来向 `Text` 提供一个确定的建议尺寸) 的边界范围之外。这是因为，在默认情况下 `SwiftUI` 的视图是不会被裁剪的。如果我们需要进行裁剪，可以使用 `clipped` 修饰器来启用

它。

我们刚才已经看到，`overlay` 和 `background` 具有一个有用的特性：它们都会把主要子视图的尺寸提供给次要子视图。不过，有时候可能我们并不希望次要视图的尺寸依赖于主要视图。比如，创建一个自定义的 `badge` 修饰器，来在视图的右上角绘制一个角标：

Code	Preview
<pre>extension View { func badge<Badge: View>(@ViewBuilder contents: () -> Badge) -> some View { self.overlay(alignment: .topTrailing) { contents() .padding(3) .background { RoundedRectangle(cornerRadius: 5).fill(.teal) } } } }</pre>	

上面的修饰器在角标的理想尺寸小于主视图尺寸时效果良好。比如，我们有一个比较大的按钮，然后尝试先是一个只有文本“1”的标记时，一切都很正常。但是，如果我们把一个比较大的角标用到一个很小的主视图 (比方说，一个 `Text("Hi")`) 上时，角标可能就无法达到所需要的尺寸了。

Code	Preview
<pre>Text("Hi").badge { Text("2023").font(.caption) }</pre>	

要解决这个问题，我们可以在 `overlay` 里插入一个 `fixedSize`。它会让角标保持自己的理想尺寸，而不再使用调用者的尺寸：

```
extension View {  
    func badge<Badge: View>(@ViewBuilder contents: () -> Badge) -> some View {  
        self.overlay(alignment: .topTrailing) {  
            contents()  
                .padding(3)  
                .background(RoundedRectangle(cornerRadius: 5))  
                .fixedSize()  
        }  
    }  
}
```

在本章稍后我们谈到对齐参考线的时候，我们会对这个角标的对齐方式进行一些改进。

容器视图

HStack 和 VStack

根据我们的经验，在使用 `HStack` 和 `VStack` 时，虽然它们的工作方式一开始很直观，可一旦遇到麻烦，它们就不再那么和蔼可亲了。因为堆栈的布局算法相当复杂，所以想要具体弄清发生了什么，会是很麻烦的事情。

水平和竖直堆栈在布局子视图时采用的方法相同，只是对应的轴不一样。所以，我们把下面的讨论限制在 `HStack` 上。

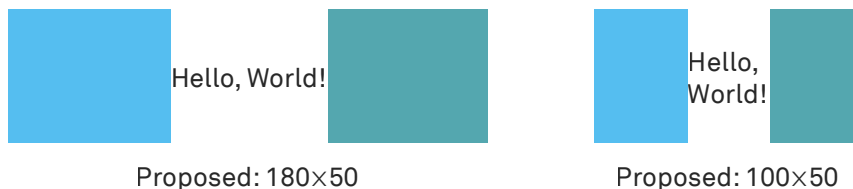
下面这个堆栈：

```
HStack(spacing: 0) {  
    Color.cyan  
    Text("Hello, World!")  
    Color.teal
```



```
}
```

如果我们提供一个足够大的宽度，那么渲染会按照预期进行：“Hello,” 文本显示在中间，它的左侧显示蓝色，右侧显示绿色。但是随着宽度逐渐变小，会发生有趣的事情：虽然还有足够的空间在同一行内完整显示文本和两边的颜色，但是文本还是选择了换行。



导致这一行为的原因，和 HStack 向子视图分配可用宽度的算法有关：

- 首先，HStack 会确定它的子视图们的**灵活性**。两个颜色视图是无限灵活的，不管向它们提供什么样的尺寸，它们都会欣然接受。但是 Text 的宽度会有上限，Text 的宽度可能介于 0 到它的理想宽度之间，但是绝对不可能超过理想宽度。
- HStack 根据子视图的灵活性从低到高进行排序。它会跟踪所有的剩余子视图和可用的剩余宽度。
- 只要还有剩余的子视图，HStack 就会把剩余的宽度除以子视图的数量，然后把结果作为建议宽度提供给这个子视图。

为了简单起见，我们假设文本的理想宽度是 100。当我们向 HStack 提供 180x180 时，因为剩余三个子视图，它首先把宽度除以 3，然后把 180/3，也就是 60，提供给灵活性最低的视图，也就是 Text。接下来 Text 根据需要进行换行或者裁断。我们假设文本的尺寸结果为 50x40 (插入了一个换行)。于是两个矩形颜色分别会得到 130/2 和 65/1 的宽度。由于这个算法，就算文本原本其实有足够的空间显示在一行，它也还是会进行换行。

有几种方法可以修改这种行为。首先，我们可以对 Text 使用 `fixedSize()` 修饰

器。这样一来，Text 将忽视掉建议尺寸并始终使用理想尺寸。这个方法确实能生效，但是一旦 HStack 得到的建议尺寸宽度比文本的理想宽度还要小时，文本仍然会按照全宽进行渲染。也就是说，这种情况下文本的宽度要大于建议宽度，文本将被渲染到边界之外。



另一种替代选择，是使用 `.layoutPriority` 修饰器给 Text 设定一个**布局优先级**。这会让 HStack 首先把全部剩余宽度提供给 Text，然后再将除去 Text 之外的剩余宽度提供给两个颜色。这样，就算没有足够的空间让文本以理想尺寸渲染自己，文本也会正确进行换行。

为了更好地理解堆栈的布局算法，这里提供一个非正式的描述 (依然以 HStack 为例)：

1. 通过向每个子视图提供 $0 \times \text{proposedHeight}$ 和 `.infinity` $\times \text{proposedHeight}$ 作为建议尺寸，来确定每个子视图的灵活性。这个向子视图提议多个尺寸的过程也被称为**探测 (probing)**。灵活性被定义为两个结果宽度的差值。
2. 将所有子视图按照它们的布局优先级进行分组。
3. 从收到的建议宽度中减去所有子视图的最小宽度，以及它们之间 (由 HStack 所定义) 的间距，把这个结果称为 `remainingWidth` (如果建议宽度为 `nil` 的话，这个值也是 `nil`)。
4. 当等待布局的分组还有剩余时：
 1. 取出布局优先级最高的分组。
 2. 将这个分组中所有子视图的最小宽度加回到 `remainingWidth` 中去。

3. 当这个分组中还有剩余的视图时：

1. 选取灵活性最小的视图。
2. 向这个视图提供建议宽度 $(\text{remainingWidth} / \text{numberOfRemainingViews}) \times \text{proposedHeight}$ 。
3. 从 `remainingWidth` 里减去视图所报告的宽度。

尽管这个算法没有写在正式的文档里，但自从我们在 2020 年本书的第一版中首次描述以来，它并没有发生过变化，因此我们认为它是稳定的。通过将实现了 `sizeThatFits` 方法的自定义形状放入到堆栈中，并记录建议尺寸和报告尺寸，至少可以对布局行为进行部分验证。

ZStack

乍一看，`ZStack` 似乎做的事情和 `overlay` 或者 `background` 是一样的，但是实际上它们的行为不尽相同。如上所述，`overlay` 和 `background` 使用主要子视图的尺寸，并把次要子视图的尺寸丢弃掉。当使用 `ZStack` 时，它的所有子视图的 `frame` 将组合起来，进行并集 (`union`) 操作，并依此计算 `ZStack` 自身的尺寸。

考虑下面的视图：

```
ZStack {  
    Color.teal  
    Text("Hello, world")  
}
```

当我们把这个视图作为一个新的 iOS app 的根视图时，颜色部分会拉伸到整个安全区域的完整尺寸。这是因为 `ZStack` 接受到的建议尺寸是整个安全区域，然后它会将同样的值再提供给它每个子视图。

如果我们不使用 `overlay`，而是用 `ZStack` 来实现 `overlay` 和 `background` 那一节中有关角标的例子，那这个角标将会影响到它所附加的视图的布局，因为此时 `ZStack` 的尺寸将会包括这个角标。虽然使用了角标的单个视图看上去外观是一样的，但是与其他视图之间的布局行为却已经发生了改变。



上面的两个视图都设定为了在 HStack 中进行底边对齐。两者也都拥有顶边和尾端对齐的角标。左侧的视图使用了 overlay 来实现角标，所以它在 HStack 中的布局并不会受到角标视图的影响。而右侧的视图在实现角标时使用的是 ZStack，因此这个视图的底边变成了角标的底边。

滚动视图

对于滚动视图 (scroll view)，下面两者是不同的东西，必须对它们区分对待：

- 实际的滚动视图本身的布局行为。
- 滚动视图中的内容 (也就是那些能在滚动视图的可见区域内滚动的部分) 的布局行为。

滚动视图本身，在滚动轴的方向上会接受父视图所提供的建议尺寸；在另一个轴上，则使用它的内容的尺寸。比如，如果我们把一个垂直的滚动视图作为根视图的话，它的高度就是会整个安全区域，宽度则取决于滚动内容。

在滚动方向上，滚动视图本质上具有无限的空间，它的内容在这个轴上也可以无限增长。因此，滚动视图在它的滚动轴上将使用 nil 作为建议尺寸，在另外的轴上则将滚动视图自身所收到的尺寸不加修改地提供给内容视图。考虑下面的示例：

```
ScrollView {  
  Image("logo")  
    .resizable()  
    .aspectRatio(.fit)  
  Text("This is a longer text")  
}
```

默认情况下，当我们为滚动视图的内容设定多个视图时 (就像我们上面做的这

样), 不管滚动方向如何, 这些子视图都会被放在一个隐式的 VStack 中。为了简化这个例子, 我们假设滚动视图接受到的建议尺寸是 320×480。它接下来会把 320×nil 提供给它的两个子视图, 然后将它们分别放到滚动容器中。滚动视图本身的高度将始终是收到的建议高度 (480)。

当滚动视图中存在文本时, 有时候文本会指定换行, 这样文本的宽度就有可能比收到的建议宽度要窄一些。如果滚动视图中没有其他的子视图将被建议的宽度接受为自己的宽度时, 就可能会导致滚动视图的宽度要比建议宽度小一些的情况。想要修复这个问题, 我们可以向 Text 添加 .frame(maxWidth: .infinity)。对于 Text 之外那些不一定要接受建议宽度的其他视图, 也可以同样处理。

当我们把一个形状放入滚动视图中时, 结果可能会让人大吃一惊。SwiftUI 中内置的形状的理想尺寸都是 10×10。这是由于 ProposedViewSize 上 .replacingUnspecifiedDimensions(by:) 的默认参数是 10×10。因为滚动视图会在滚动轴上提议 nil, 所以形状在这个轴上就将使用理想尺寸的 10。如果我们想要改变这个行为, 可以使用 .frame(height:) 或 .frame(idealHeight:) 来明确指定一个高度, 或者也可以用 .aspectRatio 基于另一个轴来确定高度。

GeometryReader

在[进阶布局](#)一章中, 我们将使用几何读取器 (GeometryReader) 来访问建议尺寸的大小。几何读取器本身也算一个视图, 它会始终接受被建议的尺寸, 并将这个尺寸通过一个 GeometryProxy 报告给它的视图构建闭包, 通过这个值我们就可以获取到几何读取器的尺寸了。GeometryProxy 还可以让我们访问当前的安全区域的缩进值 (inset) 以及视图在特定坐标空间中的 frame 值, 它还可以用来解析锚点 (anchor)。举个例子, 下面是一个用来显示建议尺寸的几何读取器:

```
GeometryReader { proxy in
    Text(verbatim: "\(proxy.size)")
}
```

不过, 在阅读 Stack Overflow 或者浏览社交媒体时, GeometryReader 是那些经常引起麻烦的视图之一。这是因为几何读取器的功能不仅仅只是“阅读”, 它总是会把自己的大小变成建议尺寸。比如, 我们想要把 GeometryReader 放到某个

Text 视图外围来测量它的宽度时，将会影响到 Text 周围的布局。

一旦我们对 SwiftUI 的布局系统有了深入的理解，我们就可以在很多地方避免使用几何读取器了。这么说，显然就意味着在某些情况下我们确实需要用到这个类型。如果我们不想让几何读取器影响布局，有下面两种方法可以做到：

- 当我们把一个完全灵活的视图放到 GeometryReader 里，它就不会影响布局。比如对于滚动视图来说，它本身就会变成建议尺寸的大小，我们可以把它放到 GeometryReader 里来获取建议尺寸。
- 当把几何读取器放到 background 或者 overlay 修饰器中，它不会影响主视图的尺寸。在 background 或 overlay 中，我们可以使用 GeometryProxy 来读取和视图几何特性相关的不同的值。因为主要子视图的尺寸会被当作建议尺寸给到次要子视图 (也就是几何读取器)，所以这种做法在测量视图的大小时非常有效。我们会在[进阶布局](#)一章中看到更多例子。

和其他 SwiftUI 的容器视图相比，几何读取器是非常特殊的存在，它不提供对齐参数，在默认情况下它会将子视图放置在顶边和前边 (top-leading) 位置 —— 而除了 ScrollView 的其他所有容器视图默认会使用居中对齐。

List

在 SwiftUI 中，List 等效于 UITableView 或 NSTableView。在写作本书时，List 的功能还很有限，支持的配置也不多。List 视图本身会使用被建议的尺寸，和 ScrollView 类似，它也会把自己的宽度和 nil 的高度提议给它的子视图。

在竖直方向上，列表内容的尺寸计算是很复杂的，这主要是因为列表中的项目是按需布局的。和 UITableView 中非固定高度的项目情况类似，List 基于列表中已经完成布局的项目来估算出其内容的整体高度。

LazyHStack 和 LazyVStack

除了主要轴和次要轴不同之外，LazyHStack 和 LazyVStack 拥有相同的行为模式。懒加载的堆栈和那些非懒加载的普通堆栈在布局行为上来说是一样的，它们最后的尺寸都是所有子视图框架的并集。但是，懒加载的堆栈不会尝试在主要轴上为子视图进行可用空间的分配。举例来说，LazyHStack 只会把

`nil×proposedHeight` 提供给它的子视图，也就是说，子视图会使用它们自己的理想宽度。

要计算 `LazyVStack` 的高度是相当困难的一件事，事实上，如果 `LazyVStack` 被内嵌在一个滚动视图中时，它会只在需要时才按需创建子视图 (对于宽度，`LazyVStack` 将直接接受被建议的宽度)。例如，考虑以下视图：

```
ScrollView {
  LazyVStack {
    ForEach(0..<100) { i in
      Text("View \(i)")
        .padding(.vertical)
        .onAppear {
          print("View \(i) appeared")
        }
    }
  }
}
```

随着不断滚动，`print` 会打印出越来越多的信息，渲染树中的视图也被按需创建。因此，就像 `List` 一样，`LazyVStack` 也许要基于已经布局的子视图来估算自己的高度，并在新的子视图出现在屏幕上时更新自己的尺寸。

LazyVGrid 和 LazyHGrid

在这节中，我们来研究网格 (grid) 的布局。我们重点关注 `LazyVGrid`，因为 `LazyVGrid` 和 `LazyHGrid` 在计算它们的行 (row) 或列 (column) 的尺寸时，使用的是相同的底层算法。

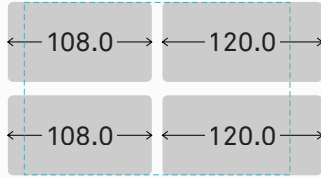
`LazyVGrid` 布局的第一步是根据网格收到的建议尺寸的宽度，来决定网格中列的宽度。网格支持三种列的类型：固定列、灵活列 (flexible column) 和自适应列 (adaptive column)。固定宽度的列无条件地使用指定的宽度，灵活列 (在它们的宽度上界和下界之间) 灵活可变，而自适应列实际上是包含了多个子列的容器。

网格首先从建议宽度中减去所有固定列的宽度。对于剩余的列，算法将把剩余宽度除以剩余的列数，然后按照顺序建议给每一列。灵活列将会使用它的 (最小和/

或最大宽度) 来限制这个建议宽度。自适应列比较特殊：网格会通过将建议的列宽度除以为这个自适应列指定的最小宽度，从而尽可能多地容纳自适应列中的子列。之后，这些子列会被拉伸到指定的最大宽度，以填充剩余空间。

现在，列的宽度已经都被计算出来了，网格会把所有列的宽度加起来，再算上列之间的间距，把结果作为它自己的宽度。对于高度，网格会向它的子视图建议 `columnWidth×nil` 的尺寸来计算行的高度，然后把所有的行高和行间距加起来，作为自己的高度。

和 `HStack` 及 `VStack` 相反，网格会执行两次列布局算法：第一次在布局阶段中进行，而之后在渲染阶段又执行一次。在布局阶段，网格使用它收到的建议宽度作为起始宽度进行计算。而在渲染阶段中，它将使用布局阶段的结果作为起始宽度，并再一次将该宽度分配给各个列。这可能会带来让人非常惊讶的结果。举个例子，考虑下面的网格：

Code	Preview
<pre>LazyVGrid(columns: [GridItem(.flexible(minimum: 60)), GridItem(.flexible(minimum: 120))], spacing: 10, content: { ... }) .frame(width: 200) .border(.teal, /* ... */) </pre>	

虽然两列的最小宽度加起来也可以轻松适配到 200 的可用宽度中去，但是网格不仅渲染出了边界，它甚至都没能在 200 宽度的中心进行渲染。让我们一步步看看网格的排序算法，来理解到底发生了什么。

我们从剩余宽度 200 开始，首先减去 10 的间隔，得到 190。对于第一列，我们将 190 的宽度除以剩余的列数 2，得到 95。因为第一列的最小宽度设定为了 60，所以 95 并不会被限制，于是剩下的宽度保持在 95。第二列使用这个 95 的

剩余宽度，但它的最小宽度需要是 120，所以它使用 120 作为宽度。

但是，这并不是我们看到的网格渲染的结果：渲染出的网格第一列宽度是 108，第二列宽度 120，而我们计算得到的应该是 95 和 120。这就是第二遍布局过程发挥作用的地方。

在第一遍布局中，计算得到的网格整体宽度是 $95 + 10 + 120 = 225$ 。具有 200 固定框架的 frame 将把网格放置在正中，所以它会向左边偏移 $(225 - 200) / 2 = 12.5$ 的距离。当网格需要渲染自己的时候，它会再次执行列布局算法，但这次的初始剩余宽度会从 225 开始。

第二遍布局从 225 开始，先减去 10 的间隔，剩下 215。于是第一列变成了 215 除以剩余列数 2，四舍五入后得到了 108。剩余的宽度是 215 减去 108，结果是 107。107 比第二列设定的最小宽度 120 还小，所以该列将使用 120 作为宽度。这与我们上面看到的示例完全吻合。

除了令人感到惊讶的宽度外，我们现在也能解释为什么网格的渲染偏离了固定 frame 中心的情况了：因为网格外的 frame 是按照网格的原始宽度 225 进行计算的，但是网格现在会将将自己的整体宽度渲染为 $108 + 10 + 120 = 238$ ，所以网格呈现出偏离中心 7 point 的现象。

想要了解更多关于网格布局的例子，可以参考这篇[博客文章](#)。

Grid

Grid 是在 iOS 16/macOS 13 引入的，它的工作方式类似于 HStack 和 VStack，但是在两个维度上都进行堆叠。在布局时，Grid 的子视图将按照灵活性排序。取决于建议尺寸，既可以按照两个维度上的灵活性计算，也可按照一个维度的灵活性计算。之后，Grid 将可用空间在两个维度上分配给子视图。

截至撰写本书时，Grid 的布局行为尚未有文档说明。在 iOS 17 之前，这个类型存在[一些 bug](#)。在 iOS 17 及更高版本中，其行为和旧版本有些不同，但仍存在一些 bug。因此，在这里我们就不详细展开了。

ViewThatFits

当我们想要根据不同的建议尺寸显示不同的视图时，可以使用 `ViewThatFits`。它接受多个子视图，并显示第一个能适配建议尺寸的子视图。`ViewThatFits` 的工作方式是向每个子视图建议 `nil` 尺寸，来获取子视图的理想尺寸，然后显示 (按照出现在代码中的) 第一个理想尺寸能适配到建议尺寸之中的子视图。如果所有子视图都无法适配，那么它选取最后的一个子视图。

渲染修饰器

SwiftUI 有一系列会影响视图渲染方式，但不影响布局本身的视图修饰器，比如 `offset`, `rotationEffect`, `scaleEffect` 等。我们可以将这些修饰器想象成类似于执行了 `CGContext.translate`，它们修改视图绘制的位置。然而，从布局系统的角度来看，视图依然位于它最初的位置。

对齐

SwiftUI 使用对齐 (alignment) 来确定某个视图相对于其他视图的位置。默认情况下，几乎所有的视图都会把子视图居中对齐。比如，如果我们创建一个仅包含文本或矩形的 iOS app，这个视图将在安全区域内按照水平和垂直两个方向居中对齐。

为了更好地理解对齐的工作原理，我们来看看下面这个例子：

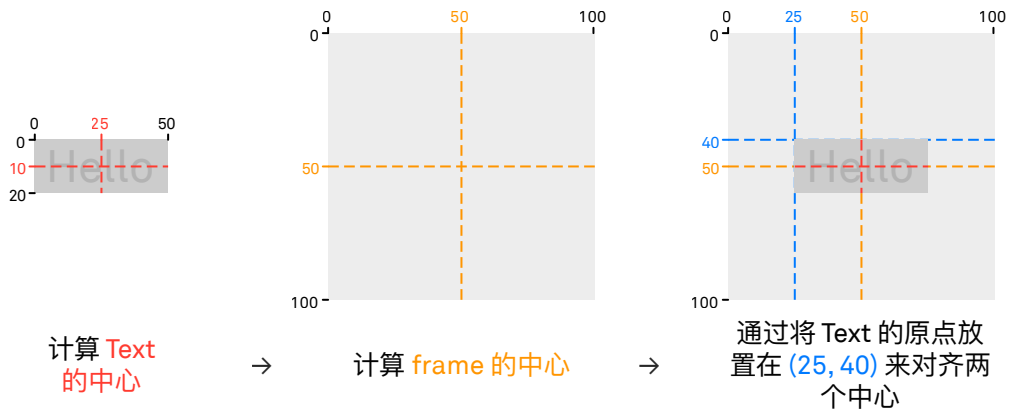
```
Text("Hello")
```

```
.frame(width: 100, height: 100)
```

`frame` 修饰器拥有一个默认值为 `.center` 的 `alignment` 参数。我们在前面的部分讨论过，固定的 `frame` 总会使用我们所指定的尺寸，在本例中，是 `100×100`。在这 `100×100` 的区域中，文本将被居中对齐。为了简单起见，我们假设 `Text` 报告的尺寸是 `50×20`。当放置这个子视图时，`frame` 修饰器将执行以下步骤：

1. 它向子视图询问水平中心。由于子视图的宽度是 `50`，所以它将自己的水平中心报告为 `25` (在子视图的本地视图坐标系中)。这个值被叫做子视图的**水平中心对齐参考线**。

2. 它向子视图询问垂直中心。这个子视图的高度为 20，所以它将自己的垂直中心报告为 **10**。这是子视图的**垂直中心对齐参考线**。
3. `.frame` 计算自己的水平和垂直中心，结果为 **(50, 50)**。
4. 现在，`frame` 可以将子视图中心对齐了，通过计算两个中心点的差值 **(50-25, 50-10)** 并将子视图的原点放在 `frame` 坐标空间的 **(25, 40)** 就可以做到这一点。



想要理解 SwiftUI 的对齐系统，最重要的一点是理解对齐总是父视图和子视图之间“协商”决定的结果。父视图不会独自决定子视图的放置位置，而是会向子视图协商相关的对齐参考线，然后根据子视图自己的大小或者其他子视图来决定它的位置。

下面是同样的例子，只不过采用了 `.bottomTrailing` 对齐：

```
Text("Hello")
```

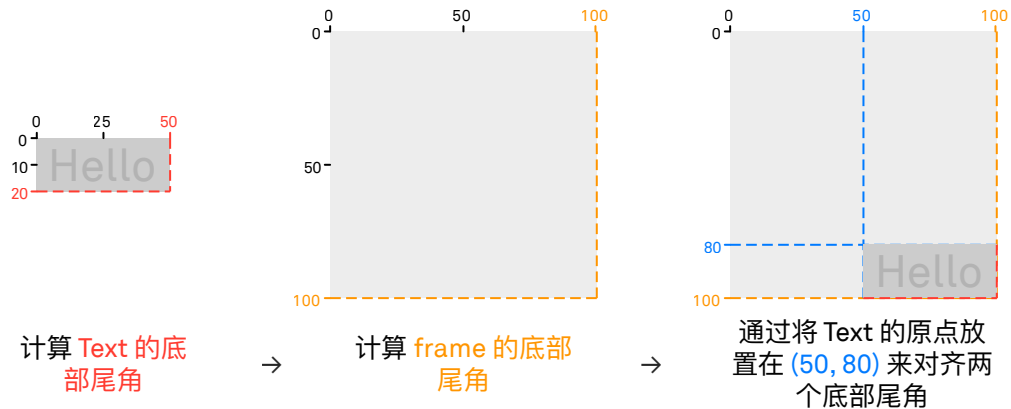
```
.frame(width: 100, height: 100, alignment: .bottomTrailing)
```

算法和上面的 `center` 对齐完全相同：

1. 它向子视图询问子视图的尾端对齐参考线。子视图回应 **50**，这个数值是子

视图在自己的视图坐标系中的尾边的位置。

2. 它向子视图询问其底部的对齐参考线。子视图回应`20`。
3. `frame` 计算得到它自己的尾边 (`100`) 和底边 (`100`) 的对齐参考线。
4. `frame` 将子视图的原点放置在 (`50, 80`)。同样地，这也是通过求解 (`100-50, 100-20`) 的差所得到的。

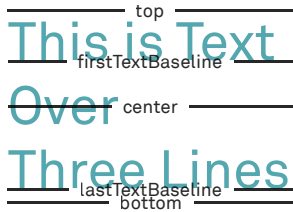


对于像 `frame` 这样的视图修饰器，对齐可以在两个方向上生效。`frame` 的对齐参数类型是 `Alignment`，它是一个组合了 `HorizontalAlignment` (水平对齐) 和 `VerticalAlignment` (垂直对齐) 的组合结构体。其他能够在两个方向上进行对齐的类型包括 `.overlay`，`.background` 和 `ZStack`。相比之下，`VStack` 只拥有水平对齐，而 `HStack` 只拥有垂直对齐。注意，`Alignment` 类型并不是对齐参考线，它只是决定了在执行对齐时，要使用哪一个对齐参考线。

`HorizontalAlignment` 和 `VerticalAlignment` 结构体内包含了内置的对齐参考线的静态常量：它们分别是针对水平方向的 `.leading`，`.center` 和 `.trailing`，以及垂直方向的 `.top`，`.center`，`.bottom`，`.firstTextBaseline` 和 `.lastTextBaseline`。组合结构体 `Alignment` 将这些常量组合起来，形成诸如 `.topLeading` 或 `.bottomTrailing` 这样的常量。

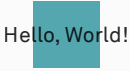
`HorizontalAlignment` 和 `VerticalAlignment` 类型都有一个方法，来计算在给定

视图方向上的默认值。因此，每个视图都自动定义了所有的内置对齐参考线。例如，下面是三行的 Text 视图在垂直方向上最重要的几条对齐参考线：



当计算不包含任何文字的 Text 视图的 firstTextBaseline 和 lastTextBaseline 时，这个参考线将使用视图的高度 (此时这两个参考线等效于 .bottom)。

接下来，让我们看一个包含多个子视图的简单 ZStack 的示例。ZStack (以及除了 frame 之外的所有其他容器视图) 必须向所有子视图咨询它们的对齐参考线，并相对于彼此进行对齐。与上面的 frame 示例相比，这个过程要复杂得多，因为 ZStack 首先必须 (基于其子视图的尺寸以及它们的对齐参考线) 计算自己的大小，然后才能使用对齐方式来放置子视图：

Code	Preview
<pre>ZStack { Rectangle() .fill(.teal) .frame(width: 50, height: 50) Text("Hello, World!") }</pre>	

因为 ZStack 里对齐的默认值是 .center，所以它将两个子视图按照以下步骤进行中心对齐：

1. 确定 ZStack 自己的尺寸：

1. 向第一个子视图 (带有蓝色矩形的 frame 视图) 询问它的尺寸和中心对齐参考线。它回应尺寸为 50×50 ，对齐点为 $(25, 25)$ 。
 2. 向第二个子视图 (Text，我们假设它的尺寸是 100×20) 询问尺寸和中心对齐参考线。它回应尺寸为 100×20 ，对齐点为 $(50, 10)$ 。
 3. 计算 Text 相对于矩形的原点位置，通过将两个对齐点相减，可以得到结果： $(25, 25) - (50, 10) = (-25, 15)$ 。
 4. 确定每个子视图的框架：框架指的是子视图的原点和尺寸的组合。
 5. 计算两个子视图框架的并集，这个并集的原点为 $(-25, 0)$ ，尺寸为 100×50 。该尺寸将被作为 ZStack 自己的尺寸。
2. 放置 ZStack 的子视图：

1. 基于步骤 1 中计算出的尺寸，ZStack 算出自己的中心为 $(50, 25)$ 。
2. 计算矩形子视图的原点，用 ZStack 的中心点减去矩形的对齐点： $(50, 25) - (25, 25) = (25, 0)$ 。
3. 计算 Text 的原点，用 ZStack 的中心点减去文本的对齐点： $(50, 25) - (50, 10) = (0, 15)$ 。

更通用地描述，每个容器视图在对齐它的子视图时，采用如下方式：

1. 确定自身尺寸：
 1. 确定它的子视图的尺寸。这具体取决于特定视图的类型，我们在本章的第一部分已经讨论过这个问题了。
 2. 基于容器的对齐方式，向子视图们询问它们的对齐参考线。
 3. 使用任意一个特定的子视图作为参考，计算所有子视图的框架。
 4. 计算这些子视图框架的并集，将所得到的结果尺寸作为容器的尺寸。
2. 放置子视图：
 1. 依据容器视图自己的尺寸计算出它自己的对齐参考线。
 2. 通过从容器视图的对齐参考线中减去子视图的对齐参考线，计算出每个子视图的原点。

依据我们要处理的容器视图的不同，第一个步骤可能会被部分或者完全省略。比如说，这个容器被包裹在同时给定了宽和高的固定 frame 中，这时它的尺寸就已知了。或者容器位于 overlay 中，它的尺寸永远基于主要子视图，所以这时对齐就不再决定 overlay 的尺寸了。

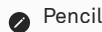
修改对齐参考线

使用内置的对齐参考线，对于多个视图，我们只能为每个视图都使用同样的对齐。比如说，我们可以将一个视图的顶边和另一个视图的顶边对齐。或者我们可以将一个视图的底部尾角和另一个视图的底部尾角对齐。然而我们不能将一个视图的中心对齐到另一个视图的顶部尾角等。不过好消息是，我们可以覆盖内置(隐式)的对齐参考线，甚至也可以创建自己的对齐方式。

SwiftUI 允许我们通过为特定的某个对齐方式提供**显式**的对齐参考线，来改变视图原先的隐式对齐参考线。例如，我们可以覆盖一个 Image 的 firstTextBaseline 的计算方法：

```
let image = Image(systemName: "pencil.circle.fill")
    .alignmentGuide(.firstTextBaseline, computeValue: { dimension in
        dimension.height/2
    })
```

提供显式对齐参考线本身并不会做任何事。只有当这个对齐参考线在实际被使用来放置子视图时，它才会生效。比如：

Code	Preview
<pre>HStack(alignment: .firstTextBaseline) { image Text("Pencil") }</pre>	

然而，一旦我们将 HStack 的对齐方式改为 .center，上面自定义的对齐参考线就

不再会影响对齐了。

传递给 `.alignmentGuide` API 的 `computeValue` 闭包，拥有一个类型为 `ViewDimensions` 的参数。它很类似于 `CGSize`，拥有 `width` 和 `height`，但它同时还允许我们获取底层视图的对齐参考线。比如，我们也可以按照下面的方式来编写之前的 `image` 示例：

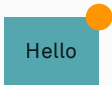
```
let image = Image(systemName: "pencil.circle.fill")
    .alignmentGuide(.firstTextBaseline, computeValue: {
        $0[VerticalAlignment.center]
    })
```

在我们想要使用两种不同的对齐参考线来对齐视图时，修改对齐参考线的方式会非常有用。举例来说，如果我们想要把一个角标视图的中心覆盖到另一个视图的顶部尾角时，可以使用下面的代码：

```
extension View {
    func badge<B: View>(@ViewBuilder _ badge: () -> B) -> some View {
        overlay(alignment: .topTrailing) {
            badge()
                .alignmentGuide(.top) { $0.height/2 }
                .alignmentGuide(.trailing) { $0.width/2 }
        }
    }
}
```

它将把 `.topTrailing` 的对齐参考线修改到角标的视觉中心。注意，`overlay` 上指定的对齐方式 (`.topTrailing`) 和在角标上显式设定的对齐参考线 (`.top` 和 `.trailing`) 是相匹配的。

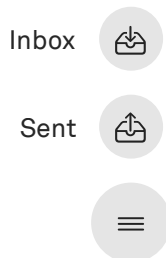
上面的修饰器可以这样使用：

Code	Preview
<pre>Text("Hello") .padding() .background(.teal) .badge { Circle() .fill(Color.orange) .frame(width: 20, height: 20) }</pre>	

自定义对齐标识

到目前为止，我们使用了内置的隐式对齐方式，并学习了如何使用 `.alignmentGuide` API 来为内置的对齐方式指定显式的对齐参考线。其实我们还可以更进一步，去设置完全自定义的对齐方式。

让我们来考虑这样一个布局：



最下方稍大一点的菜单按钮需要和上面几个单独的带文字的菜单按钮的圆形部分在竖直方向上居中对齐。下面是这个菜单大致的结构：

```
VStack {
  HStack {
    Text("Inbox")
    CircleButton(symbol: "tray.and.arrow.down")
  }
  CircleButton(symbol: "tray.and.arrow.up")
  CircleButton(symbol: "list")
}
```

```

        .frame(width: 30, height: 30)
    }
    HStack {
        Text("Sent")
        CircleButton(symbol: "tray.and.arrow.up")
            .frame(width: 30, height: 30)
    }
    CircleButton(symbol: "line.3.horizontal")
        .frame(width: 40, height: 40)
}

```

这里的问题是 VStack 的水平对齐方式：内置的对齐方式都无法符合我们的需求。我们可以尝试在 VStack 上指定 `.trailing` 对齐，然后在两个小菜单的 CircleButton 上为 `.trailing` 指定显式的对齐参考线。但是这也行不通：当 VStack 放置它的子视图时，它确实会向每个子视图询问 `.trailing` 对齐参考线。但因为 HStack 也有它自己的 `.trailing` 对齐参考线，所以在 CircleButton 上定义的显式对齐参考线将永远不会被使用。

这时候就要用到自定义对齐了，它能够跨越容器视图使用。实现自定义对齐的第一步，是需要创建一个遵守 AlignmentID 协议的类型：

```

struct MenuAlignment: AlignmentID {
    static func defaultValue(in context: ViewDimensions) -> CGFloat {
        context.width/2
    }
}

```

AlignmentID 协议的唯一要求，是一个静态的 `defaultValue(in:)` 方法。在这个方法里，我们需要为特定的对齐选择一个默认值。之后，除非我们指定一个显式的值，否则都将使用这个默认值来进行对齐。对于我们想要达成的目的，可以选择将视图的水平中心用作默认值。

现在我们可以为 HorizontalAlignment 结构体中为我们的自定义对齐添加一个静态常量，就像 SwiftUI 在定义内置对齐方式时做的一样：

```

extension HorizontalAlignment {

```

```
static let menu = HorizontalAlignment(MenuAlignment.self)

}
```

现在，.menu 的对齐方式将由我们的 MenuAlignment 结构体进行标识。我们可以将它用在菜单的 VStack 里，并为 CircleButton 的 .menu 对齐方式指定显式的对齐参考线：

```
VStack(alignment: .menu) {
  HStack {
    Text("Inbox")
    CircleButton(symbol: "tray.and.arrow.down")
      .frame(width: 30, height: 30)
      .alignmentGuide(.menu) { $0.width/2 }
  }

  HStack {
    Text("Sent")
    CircleButton(symbol: "tray.and.arrow.up")
      .frame(width: 30, height: 30)
      .alignmentGuide(.menu) { $0.width/2 }
  }
  CircleButton(symbol: "line.3.horizontal")
    .frame(width: 40, height: 40)
}
```

我们没有为大的菜单按钮本身指定 .menu 的显式值，因为 .menu 的默认值已经是中心对齐了，这和我们的目标是相同的。

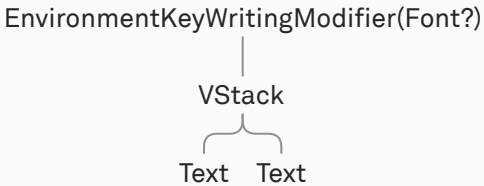
现在，当 VStack 向它的子视图询问 .menu 对齐参考线时，HStack 将会先去咨询它的子视图，看看它们是否能提供一个显式的对齐参考线。如果子视图中没有明确定义，再回滚到默认值。这意味着现在当被问到 .menu 对齐参考线时，HStack 会返回我们定义在 CircleButton 上的显式对齐参考线。也就是说，就算我们在这些显式对齐参考线上使用的是默认值，现在这些显式的对齐参考线也会被优先使用，来取代 HStack 上的隐式对齐参考线。

环境

5

环境 (environment) 是让 SwiftUI 代码能如此紧凑的一个关键机制。本质上，它是一种内置的依赖注入 (dependency injection) 技术。

下面这个例子中，当我们在 VStack 上设置字体时，它会传播到两个 Text 视图中去，并改变它们的外观：

Code	View Tree
<pre>VStack { Text("Item 1") Text("Item 2") } .font(.title)</pre>	 <pre>graph TD A[EnvironmentKeyWritingModifier(Font?)] --> B[VStack] B --> C[Text] B --> D[Text]</pre>

在视图树中，我们可以看到 .font 修饰器被转换成了一个带有 Font? 值的 EnvironmentKeyWritingModifier。

除了 .font(.title) 外，我们也可以用这样的方式来设定字体：

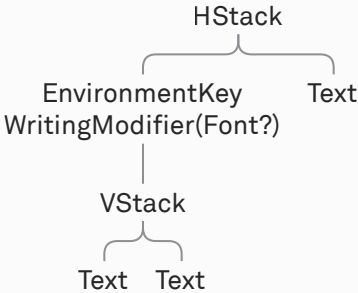
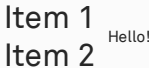
```
VStack {  
  ...  
}  
  
.environment(\.font, .title)
```

用这种方法我们可以得到完全一样的视图树；.font 修饰器仅仅只是提供了一种更容易被发现和更方便的语法，来帮助我们z将字体写入到环境中。

.environment(_:value:) API 的第一个参数是 EnvironmentValues 中的一个键路径 (key path)，我们不妨将整个环境想象成某种存储了所有环境值的字典。我们在为自定义的值设定键路径时，会更深入地了解它的工作原理。

环境是 SwiftUI 中用来将值沿着视图树向下传递的机制。在我们的例子中，这意味着在 EnvironmentKeyWritingModifier 之下的所有视图 (也就是 VStack 和两

个 Text 视图) 都可以从我们指定的环境中读取到字体值。从 EnvironmentKeyWritingModifier 往上的视图，以及那些视图树中不在这个分支的视图，则看不到我们在这里设置的自定义值。下面是一个简单的例子，可以展示这种传播的行为：

Code	View Tree	Preview
<pre>HStack { VStack { Text("Item 1") Text("Item 2") } .font(.title) Text("Hello!") }</pre>	 <pre>graph TD HStack --> EnvKey[EnvironmentKeyWritingModifier(Font?)] HStack --> Text1[Text] EnvKey --> VStack[VStack] VStack --> Text2[Text] VStack --> Text3[Text]</pre>	

与 EnvironmentKeyWritingModifier 并列层级的 Text 仍然显示了默认字体，而在 VStack 内部的两个 Text 则使用了大号的标题字体。

我们可以将环境想象为一个带有键和值的字典，它从视图树的根部一直向下传递到叶子节点。在视图树的任意位置上，我们都可以通过写入环境的方式来改变某个键的对应值 (就像我们使用 .font 修饰器时所做的那样)。之后，位于这个环境写入修饰器下方的子树将接受到修改后的环境。

SwiftUI 中的许多修饰器都使用环境值来对整棵子树产生影响，而不仅仅只作用于单一视图。例如，我们可以设定背景样式、文本大小写、字体以及一系列其他东西 ([EnvironmentValues](#) 的文档中包含了一个完整的列表)。环境值也会被用来进行“全局”设定，比如当前的区域设置或者时区等，这些设定对 app 中的每个视图来说都是可用的。

从环境中读取

我们可以使用 `@Environment` 属性包装来从环境中读取一个值。这让我们可以访问环境中特定的值，也让我们可以观察这个值的变更。这意味着，在 SwiftUI 中，我们不需要考虑像是“自动更新到当前区域设置”这种事情，因为当区域设置发生变化的时候，它通过环境进行传播，所有读取了当前环境的视图都会被重新渲染。类似地，我们可以读取像是当前动态类型 (dynamic type) 尺寸这样的设置：

```
struct ContentView: View {
    @Environment(\.dynamicTypeSize) private var dynamicTypeSize
    var body: some View {
        HStack {
            Text("Hello")
            if dynamicTypeSize < .large {
                Image(systemName: "hand.wave")
            }
        }
    }
}
```

当我们在模拟器中使用默认设置运行上面的视图时，我们将看不到图像。但是我们使用调试工具覆盖动态类型的设置时，我们会看到 `ContentView` 被自动重新渲染，图像也按照设置相应地被插入或者移除。

要注意，每当动态类型尺寸变化时，`ContentView` 都会被重新渲染。大多是情况下这不是什么问题，但是有时我们只对环境值 (也就是本例中的动态类型) 中某一个非常特定的部分感兴趣，环境属性包装 `@Environment` 恰好支持键路径取值，这正是我们想要的。打个比方，如果我们只对动态类型尺寸是否是 `accessibility` 对应的尺寸感兴趣时，可以这么做：

```
@Environment(\.dynamicTypeSize.isAccessibilitySize) private var isAccessibilitySize
```

这个属性将只会在 `isAccessibilitySize` 改变时触发一次重新渲染，而不会在每次 `dynamicTypeSize` 变更时都重新渲染。对于动态类型来说，因为它几乎在 app 中不太改变，所以这个变更不太可能带来实质的性能提升。但是当我们将一个很大的值放到环境中的话，我们可以通过书写更具体的键路径，来避免不必要的重新

渲染。


和 `@State` 属性类似，我们没有任何理由将 `@Environment` 属性暴露给外界。因此，我们通常会环境值标记为私有，并使用一个代码检查规则 (linting rule) 来验证它们的私有访问权限。和 `State` 属性相似的另一个点是，我们只能在视图的 `body` 中从环境里读取值；在视图的初始化方法中，视图还不具有身份标识，因此我们不能在那里对环境进行读取。如果我们尝试在视图的初始化方法中读取环境属性，我们会得到一个运行时的警告。

自定义环境键

在[对齐的章节](#)中，我们看到过如何构建一个角标视图。在本节里，我们将扩展这个角标，让它的样式可以定制。现在，我们忽略掉对齐的问题，而专注于角标内容的绘制。下面是一个简单的实现：

```
struct Badge: ViewModifier {
    func body(content: Content) -> some View {
        content
            .font(.caption)
            .foregroundColor(.white)
            .padding(.horizontal, 5)
            .padding(.vertical, 2)
            .background {
                Capsule(style: .continuous)
                    .fill(.blue)
            }
    }
}
```


我们可以这样使用角标：

Code	Preview
<pre>Text(3000, format: .number) .modifier(Badge())</pre>	

现在，角标的样式是硬编码成蓝色胶囊背景和白色文本的。为了让角标的颜色可以定制，我们首先将蓝色切换成 tint 颜色：

```
content
...
.background {
  Capsule(style: .continuous)
    .fill(.tint)
}
```

现在，角标将使用当前的 tint 颜色进行渲染了，如果我们没有指定颜色，就回滚到系统默认的 tint。比如，我们可以将整棵子树的 tint 颜色变为红色：

Code	Preview
<pre>VStack { Text(3000, format: .number) .modifier(Badge()) Text("Test") .modifier(Badge()) } .tint(.red)</pre>	

我们也可以不依赖 tint 颜色，而是创建我们自己的环境键。要实现这种方式，需要两个必须步骤和一个可选步骤：

→ 我们必须实现一个自定义的 EnvironmentKey，用来它作为角标颜色在环境中的键，并且将 Color 类型关联到这个键上。

- 我们必须在 `EnvironmentValues` 上添加扩展，并提供一个属性，让我们能在环境中获取和设置该值。
- (可选) 我们可以在 `View` 上提供一个辅助方法，用来轻松地整棵子树设置角标颜色。这可以让我们将自定义的键和扩展隐藏起来，同时为用户提供一个易于发现的 API。

第一步，我们先来创建一个自定义的环境键。刚开始时可能你会觉得有些复杂，但是我们很快就会习惯这个模式，它使得环境值能以类型安全的方式工作：环境键并不是一个值，而是使用类型来定义的。我们将创建一个空的 `enum` 类型 (或者也可以使用 `struct` 类型)，并让它满足 `EnvironmentKey` 协议：

```
enum BadgeColorKey: EnvironmentKey {  
    static var defaultValue: Color = .blue  
}
```

`EnvironmentKey` 协议要求我们实现静态的 `defaultValue` 属性。因为我们使用 `Color` 作为它的类型，编译器将知道这个键所对应的值始终会是 `Color` 类型。我们同时提供了 `.blue` 作为默认值。除非我们在视图树上游的某个地方在环境中为这个键设置了明确的值，否则当我们从环境中读取角标颜色时，都会取到这个默认值。

第二步，我们需要向 `EnvironmentValues` 结构体添加一个计算属性。我们在环境中进行读写时需要一个键路径，本例中，我们使用 `badgeColor` 这个名字：

```
extension EnvironmentValues {  
    var badgeColor: Color {  
        get { self[BadgeColorKey.self] }  
        set { self[BadgeColorKey.self] = newValue }  
    }  
}
```

最后，我们添加一个辅助方法来设置颜色：

```
extension View {  
    func badgeColor(_ color: Color) -> some View {
```

```

        environment(\.badgeColor, color)
    }
}

```

Apple 在关于 EnvironmentValues 的[文档](#)中，也提供了包含这些步骤的模板。


想要读取角标颜色，我们可以在 Badge 视图修饰器里使用 @Environment 属性包装器：

```

struct Badge: ViewModifier {
    @Environment(\.badgeColor) private var badgeColor
    func body(content: Content) -> some View {
        content
        .font(.caption)
        .foregroundColor(.white)
        .padding(.horizontal, 5)
        .padding(.vertical, 2)
        .background {
            Capsule(style: .continuous)
                .fill(badgeColor)
        }
    }
}

```

现在，我们可以使用环境来改变角标的样式了：

Code	Preview
<pre> VStack { Text(3000, format: .number) .modifier(Badge()) Text("Test") .modifier(Badge()) } .badgeColor(.orange) </pre>	

当我们没有指定 `badgeColor` 时，将会使用默认值。一般来说，我们可以认为除非我们明确地进行了覆盖，否则环境中的默认值总是会被使用。

对于一些组件，我们可能会想要比这更多的样式定义。比如 SwiftUI 中的 `buttonStyle` 修饰器，和 `ButtonStyle` 协议组合使用时，可以让我们完全改变按钮渲染的方式。想要在我们自己的组件上实现类似效果时，我也同样可以使用环境。

自定义组件样式

当我们想要创建一个类似于 SwiftUI 的按钮那样，可以用类似 `.buttonStyle` 的 API 通过环境进行样式定制的自定义组件时，大致的做法和普通的环境值是一样的。不过，为了让类型能正确，我们需要一些额外的工作。首先，我们希望角标能有如下 API：

```
Text("Test")
    .badge {
        Text(3000, format: .number)
    }
```

这个角标应该以默认样式渲染，但是我们想要之后在视图层级中可以用这样的 API 来覆盖样式设置：

```
someView
    .badgeStyle(.custom)
```

所有包含在 `someView` 中的角标都应该接受到这个新的 `custom` 角标样式。想要实现这个功能，我们需要完成以下步骤：

1. 创建一个 `BadgeStyle` 协议来为角标样式定义接口。
2. 为角标样式创建一个环境键。
3. 在 `badge` 修饰器中使用这个自定义角标样式。

角标样式的协议和 `ViewModifier` 协议十分相似：它要求一个 `body` 方法，来将已经存在的视图包装进角标里。我们只希望通过这个样式能给某个标签添加上角标的外观要素，而不需要它考虑角标相对于视图的位置，这些布局操作将在 `badge` 修饰器中完成。注意，`makeBody` 方法接受的参数是一个 `AnyView`，这是因为我们在这里需要一个具体的视图类型来让代码能够编译。

```
protocol BadgeStyle {
    associatedtype Body: View
    @ViewBuilder func makeBody(_ label: AnyView) -> Body
}
```

现在我们可以创建一个默认的角标样式：

```
struct DefaultBadgeStyle: BadgeStyle {
    var color: Color = .red
    func makeBody(_ label: AnyView) -> some View {
        label
            .font(.caption)
            .foregroundColor(.white)
            .padding(.horizontal, 5)
            .padding(.vertical, 2)
            .background {
                Capsule(style: .continuous)
                    .fill(color)
            }
    }
}
```

但是，当我们想要创建一个环境键时，不能简单地使用 `DefaultBadgeStyle` 来作为键所对应的值的类型。那样做的话我们就不能使用类型不同的自定义角标样式了。相反，我们会用一个**存在体** (existential) 来把具体的类型隐藏起来。本质上来说，它会把具体类型包装到一个盒子中 (这和 `AnyView` 将一个具体的视图类型包装到盒子中的做法类似)。下面是环境键和对应的属性：

```
enum BadgeStyleKey: EnvironmentKey {
    static var defaultValue: any BadgeStyle = DefaultBadgeStyle()
}
```

```

extension EnvironmentValues {
    var badgeStyle: any BadgeStyle {
        get { self[BadgeStyleKey.self] }
        set { self[BadgeStyleKey.self] = newValue }
    }
}

```

为了使用角标样式，我们需要创建一个自定义的视图修饰器来从环境中读取样式，对标签进行转换，然后根据指定的对齐确定角标的位置。因为我们不希望角标的尺寸和它所附着的视图的尺寸有关联，所以角标样式还需要确保文本使用的是理想尺寸。下面示例中带有对齐参考线的布局相关代码，和我们在[布局章节](#)中的对齐部分里讨论过的是一致的：

```

struct OverlayBadge<BadgeLabel: View>: ViewModifier {
    var alignment: Alignment = .topTrailing
    var label: BadgeLabel
    @Environment(\.badgeStyle) private var badgeStyle

    func body(content: Content) -> some View {
        content
        .overlay(alignment: alignment) {
            AnyView(badgeStyle.makeBody(AnyView(label)))
            .fixedSize()
            .alignmentGuide(alignment.horizontal) { $0[HorizontalAlignment.center] }
            .alignmentGuide(alignment.vertical) { $0[VerticalAlignment.center] }
        }
    }
}

```

这个 OverlayBadge 修饰器不需要公开。我们可以在 View 协议上编写一个简短的辅助方法，来让本节一开始时提出的语法成真：


```

extension View {
    func badge<V: View>(alignment: Alignment = .topTrailing,
        @ViewBuilder _ content: () -> V) -> some View {
        modifier(OverlayBadge(alignment: alignment, label: content()))
    }
}

```

```
}  
  
}
```

现在，我们可以这样使用 `badge` 修饰器了，它会自动使用环境中当前的角标样式：

Code	Preview
<pre>Text("Hello") .badge { Text(100, format: .number) }</pre>	

作为演示，我们来创建一个自定义带光泽的角标样式，并实际在环境中看看这个样式的使用方式：

```
struct FancyBadgeStyle: BadgeStyle {  
  var background: some View {  
    ZStack {  
      ContainerRelativeShape()  
        .fill(Color.red)  
        .overlay {  
          ContainerRelativeShape()  
            .fill(LinearGradient(colors: [.white, .clear],  
                                startPoint: .top, endPoint: .center))  
        }  
      ContainerRelativeShape()  
        .strokeBorder(Color.white, lineWidth: 2)  
        .shadow(radius: 2)  
    }  
  }  
  
  func makeBody(_ label: AnyView) -> some View {  
    label  
      .foregroundColor(.white)  
      .font(.caption)  
      .padding(.horizontal, 7)  
      .padding(.vertical, 4)  
      .background(background)  
  }  
}
```

```

        .containerShape(Capsule(style: .continuous))
    }
}

```

为了能更容易地在环境中设置角标样式，我们再一次向 View 添加一个扩展：

```

extension View {
    func badgeStyle(_ style: any BadgeStyle) -> some View {
        environment(\.badgeStyle, style)
    }
}

```


现在，我们可以在视图层级的任何地方指定 `.badgeStyle(FancyBadgeStyle())` 来使用我们的这个 fancy 角标样式，整棵子树在遇到角标时都将使用这个样式进行渲染。为了让语法更优雅一些，我们可以向 `BadgeStyle` 添加下面这样的扩展：

```

extension BadgeStyle where Self == FancyBadgeStyle {
    static var fancy: FancyBadgeStyle {
        FancyBadgeStyle()
    }
}

```

现在我们可以把代码写成 `.badgeStyle(.fancy)` 了：

Code	Preview
<pre> HStack { Text("Inbox") Text("Spam") .badge { Text(3000, format: .number) } } .badgeStyle(.fancy) </pre>	

很遗憾，我们在定义 `BadgeStyle` 时不得不使用 `AnyView`。因为关联类型 `Body` 不能依赖于 `Label`，它需要是静态的，所以我们不能将它声明为泛型类型。

[Kasper Lahti](#) 展示过，我们至少可以通过创建额外的 `Label` 结构体来隐藏这个事实，但是在撰写本书时，我们无法在不使用 `AnyView` 的情况下实现这个特性。在同一篇博客文章中，Kasper 还展示了如何在自定义角标样式中对 `@Environment` 这样的属性进行访问的方式。

EnvironmentObject

我们不仅可以用环境来传递值，也可以用它来传递对象。不过这在 iOS 17/ macOS 14 中的工作方式发生了变化，因此我们必须区分我们想要针对的平台。

在 iOS 17 中，环境对象都应该使用新的 `@Observable` 宏，并且对应的属性应该使用我们在本章中一直使用的 `@Environment` 属性包装器进行声明。唯一的区别是，我们可以使用环境对象的类型作为键，而无需声明一个单独的键类型来遵守 `EnvironmentKey` 协议。

下面是一个使用环境对象的简单例子：

```
@Observable final class UserModel {  
    ...  
}  
  
struct Nested: View {  
    @Environment var userModel: UserModel?  
  
    var body: some View {  
        Text(userModel?.name ?? "default name")  
        ...  
    }  
}
```

如果在视图层次结构中更高级别的某个视图已经在环境中设置了一个 `UserModel`，我们就可以在这里访问到它，而无需将对象通过视图树的所有层进行传递。如果对象尚未在环境中设置，`userModel` 属性将为 `nil`。设置对象很简单：

```
struct ContentView: View {  
    var body: some View {  
        Nested()  
        .padding()  
        .environment(UserModel())  
    }  
}
```

```

    }
}

```

我们还可以将环境对象的属性声明为非可选类型。在这种情况下，我们必须保证环境中存在该对象。否则，当我们尝试访问环境属性时，应用程序将崩溃。

使用对象的类型作为环境键非常方便，但并不是绝对必要的。我们还可以像在本章前面所做的那样定义一个环境键，并使用此显式键将对象传递到视图树中。

如果我们想要针对 iOS 17 之前的平台，我们必须使用两个不同的API：

`@EnvironmentObject` 属性包装器用于从环境中读取对象，而 `environmentObject` 修饰器用于将对象设置在环境中。这些API也依赖于对象的类型作为键。但是，对象必须遵循 `ObservableObject` 协议（就像状态对象或观察对象一样）。

当使用 `@EnvironmentObject` 时，没有办法将属性声明为可选类型。如果在环境中没有设置过指定类型的对象，当我们访问 `@EnvironmentObject` 属性时代码将会崩溃。一个有用的技巧是将所有的环境对象设置器至少捆绑到一个单独的辅助方法中：

```

extension View {
    func injectDependencies() -> some View {
        environmentObject(UserModel())
        .environmentObject(Database())
    }
}

```

我们不仅可以在 app 的根视图上使用这个辅助方法，也可以将所有依赖项注入到预览中，同时仍然能够在本地对依赖项进行覆盖：

```

struct Nested_Previews: PreviewProvider {
    static var previews: some View {
        Nested()
        .environmentObject(UserModel.mock())
        .injectDependencies()
    }
}

```

```
}
```

环境对象对子类也能生效：即使 `UserModel.mock()` 返回的是 `UserModel` 的子类，`Nested` 视图仍然能够正确读取它。

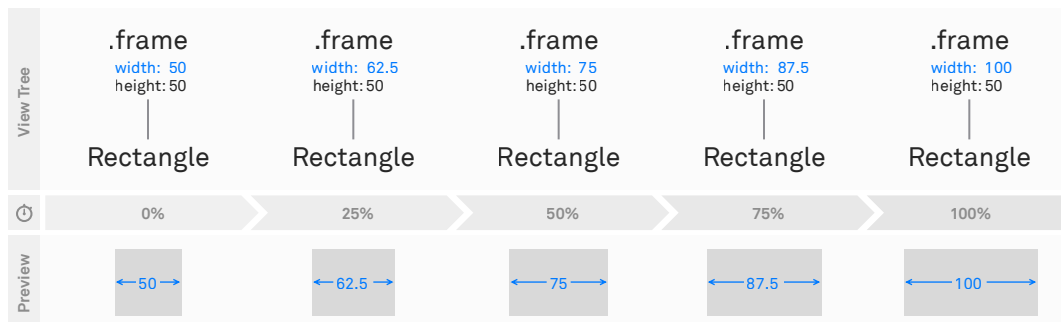
动画

6

在 SwiftUI 中，修改状态是唯一能触发视图更新的手段。默认情况下，旧视图树与新视图树之间的更改不会进行动画处理，但是有多种方法可以对 SwiftUI 进行指示，让它针对由状态改变所带来的视图上的部分或全部更改进行动画处理。以下是一个示例：

```
struct ContentView: View {
    @State private var flag = false
    var body: some View {
        Rectangle()
            .frame(width: flag ? 100 : 50, height: 50)
            .onTapGesture {
                withAnimation(.linear) { flag.toggle() }
            }
    }
}
```

当用户首次点击矩形时，flag 状态会从 false 变为 true，body 将会被重新执行。新的视图树依然拥有相同的结构：也即一个包含矩形的 frame (为了简洁起见，在这里我们忽略了点击手势，在这里它其实也是视图树的一部分)。



不过，frame 的宽度从 50 变为了 100。由于状态变化的代码被包围在了一个 `withAnimation { ... }` 调用中，SwiftUI 将会对比新的视图树和当前的渲染树，并寻找那些可以被动画的变更。因为 frame 的宽度是一个可动画的属性 (其实几乎所有的视图修饰器的参数都是支持动画的)，`withAnimation` 调用中指定的时间曲

线 `.linear` 会被用来为动画生成从 0 到 1 的进度值。这些进度值将被用作关键点，来把矩形从 50 的旧宽度插值渐变到 100 这个新宽度。

虽然你可以将动画进度视为从 0 开始到 1 结束，但在动画过程中这个进度值可能会小于 0 或大于 1。例如，当使用 iOS 17 的 `.bouncy` 时间曲线时，动画会出现过冲 (overshoot) 现象。

动画始终从渲染树的当前状态开始，向着新的视图树所描述的新状态移动，而这个新视图树正是通过对状态更改进行响应所得到的。渲染树的当前状态也可能是当前正在运行的动画所造成的瞬时状态。这一特点让 SwiftUI 的动画默认就是可以添加和取消的。

尽管本章大部分内容都适用于所有最近版本的 iOS，不过 iOS 17 中也引入了许多大大小小对 SwiftUI 动画系统的更新。主要的增加内容包括：

- 现在，动画具有完成时的回调，它会在动画完成后触发。
- `animation` 修饰器增加了一个新的重载，允许将动画限定到特定的修饰器。
- 我们现在可以使用新的 `CustomAnimation` 协议创建完全自定义的动画曲线。
- 基于阶段的动画允许我们指定一系列自动运行的动画，一个接一个地运行，始终返回到它们的起始值。
- 基于关键帧的动画提供了一种新的抽象形式，可以让我们基于任意值进行插值来创建完全自定义的动画。
- 引入了新的 API，围绕 `Transition` 协议定义过渡效果。

属性动画和过渡

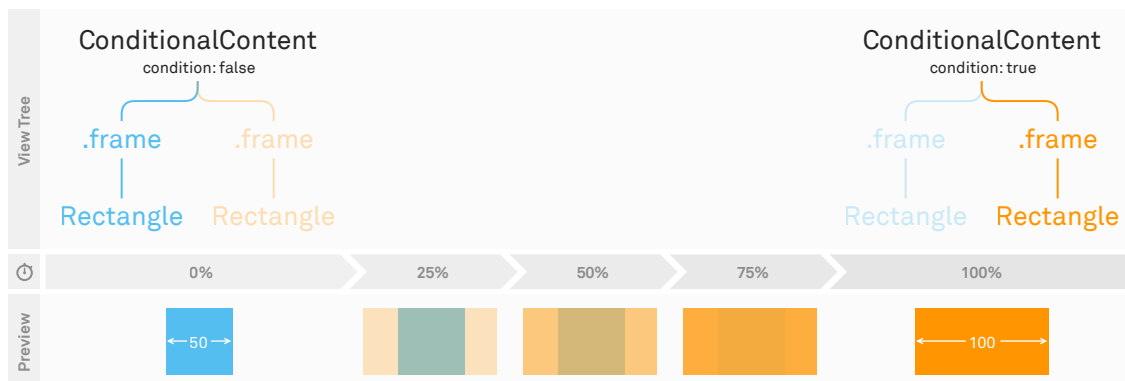
对那些在视图树中已经存在，且在状态变化前后属性也相应改变的视图，属性动画会为它们进行插值。换句话说，视图的属性发生了变化，但是在屏幕上的那个视图本身并没有改变。在上面的例子中，这意味着最重要的是只修改了 `frame` 上

的 width 参数，而并非存在两个不同的视图：frame 视图本身在状态改变前后都是同一个。

考虑下面这种替代的写法 (为了简洁起见，我们在示意图里省去了点击手势)：

Code	View Tree
<pre>var body: some View { let rect = Rectangle().onTapGesture { withAnimation { flag.toggle() } } if flag { rect .frame(width: 200, height: 100) } else { rect .frame(width: 100, height: 100) } }</pre>	<pre>graph TD CC[ConditionalContent] -- blue --> F1[.frame] --> R1[Rectangle] CC -- orange --> F2[.frame] --> R2[Rectangle]</pre>

当 flag 是 false 时，渲染树中只有蓝色这边的子树被显示出来。当 flag 转换为 true 时，蓝色子树被从渲染树中移除，作为替代橘色的子树被显示。也就是说，在状态改变之前的那个 frame，和状态改变后的那个 frame，**并不是**同一个东西了。这两个 frame 在视图树中的位置不同，它们的身份标识也不相同。由于在状态变更后，渲染树上已经不存在原来的那个 frame 了，因此 SwiftUI 不会为 frame 的宽度属性添加动画。



当尝试这个例子的时候，你会发现窄的矩形进行了一个淡出，而宽的矩形进行了淡入。我们在这里看到的就是所谓的**过渡 (transition)** 效果。和属性动画不同，过渡是那些用来将视图从渲染树中移除或者将视图插入到渲染树中的动画。我们会在[本章稍后](#)更详细地研究过渡效果。

控制动画

在 SwiftUI 中，有多种方式可以指定动画发生时时机：

1. 当某个特定的值改变时，触发隐式动画。
2. 当某个特性事件发生时，触发显示动画。

要指定隐式动画，可以在视图树的任意位置使用 `.animation(_:value:)` 视图修饰器。这个方法有两个参数：第一个参数指定动画所使用的时间曲线，第二个参数指定触发动画所需要修改的值。这个 `value` 参数用来限定动画的范围，让动画只在特定的值被更改时才发生。这对于控制动画实际生效的时机非常重要。

在 SwiftUI 的第一个版本中，还有一个现在已经被废弃的不带 `value` 参数的 `.animation(_:)` 视图修饰器。这个修饰器在应用动画时过于广泛了，它经常导致意外的行为，比如在设备方向改变的时候也会错误地进行动画。

当我们使用隐式动画来重写第一个例子中原本的显式动画时，它看起来会是这样的：

Code	View Tree
<pre>struct ContentView: View { @State private var flag = false var body: some View { Rectangle() .frame(width: flag ? 100 : 50, height: 50) .animation(.default, value: flag) .onTapGesture { flag.toggle() } } }</pre>	<pre>graph TD A[.onTapGesture] --> B[.animation(Bool)] B --> C[.frame] C --> D[Rectangle]</pre>

使用 `.animation` 视图修饰器的隐式动画会使视图子树中的所有内容都进行动画处理，因此将修饰器放在正确的位置就非常重要。通常，为了避免不必要的副作用，我们最好尽可能在离所需要的视图附近的位置添加动画，特别是当我们之后更改代码时更容易出现问题。当我们查看上面的动画的视图树时，我们可以看到 `frame` 和矩形形成了动画修饰器的子树。

很遗憾，在编写本书时，这条规则还存在一些例外，会让 `.animation(_:value:)` 不仅仅只针对它的子树进行动画。比如，在上面例子中，就算我们把 `.frame` 和 `.animation` 两行位置对调，`frame` 依旧会进行动画。据我们了解，这是预期内的行为。Ole Begemann 在他的[博客文章](#)中很好地描述了这些例外情况。

在 iOS 17 中，新的 `.animation(_:body:)` 修饰器让我们可以将隐式动画限定在特定的修饰器中，就像下面代码这样：

```
Text("Hello World")
    .opacity(flag ? 1 : 0)
    .animation(.default) {
```

```
$0.rotationEffect(flag ? .zero : .degrees(90))  
}
```

在 `flag` 值更改时，只对旋转效果进行动画，而不对透明度进行动画。在我们的测试中，这与以下代码的效果相同：

```
Text("Hello World")  
  .opacity(flag ? 1 : 0)  
  .animation(nil)  
  .rotationEffect(flag ? .zero : .degrees(90))  
  
  .animation(.default)
```

这个新的 `.animation` 变体带来了两个潜在的陷阱：

1. 与 `.animation(_:value:)` 修饰器相比，具有闭包的新动画修饰器不接受值作为参数来控制动画何时起效。这意味着闭包体内的可动画的修饰器，不论它的参数是在哪里被修改的，都会在参数被修改时进行动画处理。
2. 前面已经提到过，有一些修饰器，比如说 `.frame`、`.offset` 或 `.foregroundColor`，在与动画一起使用时可能会出现意外的“错位”行为。这些修饰器最终会在叶子视图上生效，而不是在我们插入它们的视图树中的位置生效。因此，尽管在我们用到它们的视图树中的点上并没有设置动画，但这些修饰器可能仍然会发生动画。而如果我们在 `.animation(_:body:)` 的闭包体内指定这种“错位”修饰器，则反而会由于在叶子视图实际上并没有动画存在，所以结果是不会发生动画。

除了为特定的视图子树和一个特定的值的变化定义动画外，我们也可以将一个动画限定在某个特定的状态改变上，我们把它称为**显式动画**。实际上，在本章的第一个实例中，我们使用的 `withAnimation` 修饰器就创建了一个显式动画：

```
struct ContentView: View {  
  @State private var flag = false  
  var body: some View {  
    Rectangle()  
    .frame(width: flag ? 200 : 100, height: 100)  
    .onTapGesture {
```

```

        withAnimation { flag.toggle() }
    }
}
}

```

只要我们有一个像是 `onTapGesture` 的 `perform` 这样的闭包，我们就可以使用 `withAnimation` 来把需要进行动画的属性改变包装进去。

针对绑定的动画是另一类的显式动画。我们可以在一个绑定值上调用 `.animation`，从而把绑定值的 `setter` 包装到一个显式动画里。举例说明，我们可以这样来重写上面的代码：

```

struct ToggleRectangle: View {
    @Binding var flag: Bool
    var body: some View {
        Rectangle()
        .frame(width: flag ? 100 : 50, height: 50)
        .onTapGesture { flag.toggle() }
    }
}

struct ContentView: View {
    @State private var flag = false
    var body: some View {
        ToggleRectangle(flag: $flag.animation(.default))
    }
}

```

这段代码和之前的例子得到的结果完全相同，不过当然了，原来的版本要直接得多。然而，将动画运用在一个绑定值上有时会是很好的选择，它可以让我们在不修改事件闭包代码（或者根本不需要一个事件闭包）的情况下，来添加显式动画。

乍一看，隐式和显式动画似乎完成的事情是相同的。但是需要注意，相比起显式动画来说，隐式动画可能会造成潜在的不同效果。例如，考虑这样一种情况，模型层的变更（也许是来自服务器推送的新数据）会引起某个特定值的变化。当使用的是限定在这个值的隐式动画时，不论变化的源头到底是服务器推送还是用户交

互，对渲染树的更改都将会导致动画，但是视图中动画的范围是被明确定义的。而另一方面，使用显式动画时，我们则可以轻松区分变更的来源究竟是模型层还是用户交互，但是我们无法直接将动画限定在视图树中的某个特定部分。

这里并没有对或者错。我们究竟应该选择显式动画还是隐式动画，取决于我们想要实现的行为。重要的是，我们要意识到定义动画的不同方法确实可能会导致不同的行为，而且这种不同并不一定会在测试中立即显现出来。

时间曲线

SwiftUI 内置了所有常见的动画时间曲线，比如线性 (linear)、渐入 (ease-in)、渐出 (ease-out) 和弹簧 (spring) 等时间曲线。所有动画 API 都拥有一个用来指定时间曲线的参数，在省略该参数的时候，调用会回退到使用默认的时间曲线。

- `.speed(_:)` 让我们可以通过一个系数来减慢或加快动画的速度。
- `.delay(_:)` 让我们可以通过固定的时间来推迟动画的开始。
- `.repeatCount(_:autoreverses:)` 让我们可以将动画循环播放指定的次数。

如果内置的时间曲线不能满足我们的需求，我们可以使用在 iOS 17 中引入的 `CustomAnimation` 协议来实现完全自定义的时间曲线。而在 iOS 17 之前，创建具有自定义时间曲线的动画的唯一方法，则是通过使用我们将在[下面](#)看到的 `Animatable` 协议，在线性时间曲线的基础上实现自定义动画。

事物

在幕后，不论隐式动画还是显式动画都用了同样的构造方法，那就是通过**事物** (transaction) 来创建。每次由状态变更触发的视图更新都会被包装到一个事物中去，它携带了有关需要进行的动画的信息。默认情况下，事物所描述的动画信息是 `nil`。

`withAnimation` API 会隐式地创建一个事务，并对该事物的 `animation` 属性进行设置。这个隐式事物的范围就是传递给 `withAnimation` 的闭包的函数体。事实

上，我们可以通过使用 withTransaction API 来达成一样的效果：

```
struct ContentView: View {
    @State private var flag = false
    var body: some View {
        Rectangle()
            .frame(width: flag ? 100 : 50, height: 50)
            .onTapGesture {
                var t = Transaction(animation: .default)
                withTransaction(t) { flag.toggle() }
            }
    }
}
```

类似地，我们也可以用 .transaction 视图修饰器重写隐式动画：

```
struct ContentView: View {
    @State private var flag = false
    var body: some View {
        Rectangle()
            .frame(width: flag ? 100 : 50, height: 50)
            .transaction { t in
                t.animation = .default
            }
            .onTapGesture { flag.toggle() }
    }
}
```

不过还请注意，这段代码和不带 value 的那个已经被废弃的 .animation(_:) API 行为是一致的。想要用 .transaction 来模拟 .animation(animation:value:) API 行为的话，我们还需要记住 flag 的旧值，并在把动画设置到事物之前将它与新值进行比较。和隐式动画类似，事物也会沿着沿着视图树向下传递 (在本例中，传递到 frame 和 Rectangle 中)。

在幕后，隐式动画和显式动画都是为当前的事物设置 animation 属性，隐式动画的优先级要比显式动画更高。这是因为当视图树被执行时，设置隐式动画的修饰

器运行会更晚一些，它会把在状态变更开始时就由显式动画所设定的值覆盖掉。

说到动画的优先级，Transaction 里有一个非常有用的属性，叫做 `disablesAnimations`。当我们把这个属性设置为 `true` 时，隐式动画在事物期间就不会产生任何效果。`disablesAnimations` 属性不会把所有的动画全部禁用，它只起到防止隐式动画覆盖事物上已有动画的作用。

如果我们把上面例子进行改写，使用显式事物，并通过把 `transaction.disablesAnimations` 设置为 `true` 来禁用隐式动画，我们可以看到当点击时，绿色矩形会缓慢移动。

换句话说，我们可以认为隐式动画的实现类似于这样：

```
extension View {  
    func animation(_ animation: Animation?) -> some View {  
        transaction { t in  
            guard !t.disablesAnimations else { return }  
            t.animation = animation  
        }  
    }  
}
```

`.transaction` API 也可以用于移除当前的动画。在之前，当 `.animation(_)` API 还没有被废弃时，我们可以通过 `.animation(nil)` 来把指定子树的动画全部禁用掉。不过现在这会导致一个方法废弃的警告。取而代之，我们可以用 `.transaction { $0.animation = nil }` 来达到同样的效果。

在 iOS 17 中，与环境值一样，事务也可以通过自定义键来进行扩展了。类似于 `EnvironmentKey` 协议，我们可以用 `TransactionKey` 协议来扩展事物。这允许我们将状态附加到事务上，并在稍后处理事务时读取它。例如，我们可以将更改的来源添加到事务中（比如记录这个更改源自服务器还是客户端），并根据这个值选择隐式动画。

完成时的回调

动画完成回调也是事务的一部分，回调也从 iOS 17 开始可用。我们可以在使用

显式的 `withAnimation` API 时直接设置完成处理回调，或者在 `.transaction` 修饰器的闭包内使用 `.addAnimationCompletion` 将回调添加到事务中。通常情况下，它们会按预期工作，但让我们看一些并不完全直观的情况。

当手动向事务添加完成回调时，我们需要确保每次触发动画时都执行修改事务的闭包。否则，完成处理程序只会触发一次。考虑以下示例：

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

    var body: some View {
        VStack {
            Button("Animate!") { flag.toggle() }
            Circle()
                .fill(flag ? .green : .red)
                .frame(width: 50, height: 50)
                .animation(.default, value: flag)
                .transaction {
                    $0.addAnimationCompletion { print("Done!") }
                }
        }
    }
}
```

我们每次点击按钮时都会切换 `flag` 状态属性，所以视图的 `body` 每次也会重新执行，并且圆圈的填充颜色会从红色动画到绿色再回到红色。然而，SwiftUI 似乎对 `.transaction` 闭包执行了依赖追踪，由于此闭包没有依赖于任何状态，所以它并不会在每次 `body` 执行的时候也重新执行。这样一来，完成回调只会在第一次被触发。

要解决这个问题，我们可以通过使用 `.transaction(value:_ transform:)` 修饰器来强制让 `transform` 闭包在指定的值每次更改时都被调用：

```
var body: some View {
    VStack {
        Button("Animate!") { flag.toggle() }
        Circle()
    }
}
```



```
...  
    .transaction(value: flag) {  
        $0.addAnimationCompletion { print("Done!") }  
    }  
}  
}
```

完成回调可以对调用时机进行指定，选择是在动画在逻辑上完成时进行调用 (即在人类观察者的眼中“感觉”它完成)，还是在动画曲线真正完成并且动画被移除时进行调用 (这可能会发生得晚得多，比如弹簧动画的情况)。在使用新的 `CustomAnimation` 协议实现自定义动画曲线时，我们可以通过在 `AnimationContext` 上设置 `isLogicallyComplete` 来指定我们的动画应该在何处被视为在逻辑上完成。

动画协议

SwiftUI 属性动画系统的核心是 Animatable 协议。这个协议可以被视图和视图修饰器所实现，它们通过该协议把可以进行动画的属性暴露给 SwiftUI。

这个协议唯一的要求是一个 animatableData 属性，它的默认实现什么都没有做。animatableData 的类型必须遵守 VectorArithmetic 协议，这个协议本质上要求满足协议的值可以进行矢量计算：换言之，也就是这个值可以乘上一个浮点数，而且两个该类型的值可以相加（显然，标量值也满足这个需求）。有了这个保证，SwiftUI 就可以在两个值之间进行插值，从而实现从一个值到另一个值的过渡。

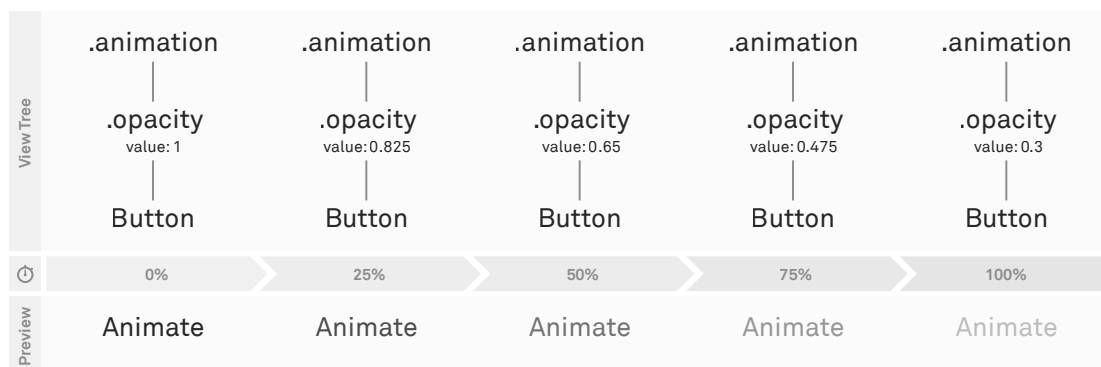
当一个事物拥有关联动画时，SwiftUI 会检查每个视图，并查找所有发生了变化的 animatableData 属性。然后它使用当前动画的时间曲线对更改进行插值。每个被插值的值都会通过 animatableData 属性进行设定，然后重新执行视图 body 或者视图修饰器的 body，以此来实现动画效果。

通常，这一切都在后台以我们看不见的方式发生，我们一般也不需要自己去遵守这个协议。但是，有些情况下我们为了达成想要的效果，可能会需要实现自定义的可动画视图或者视图修饰器。为了更好地理解 Animatable 协议在 SwiftUI 中用于实现属性动画的方式，我们先来看一个包含了透明度动画的简单例子：

```
struct ContentView: View {
    @State private var flag = true
    var body: some View {
        Button("Animate") { flag.toggle() }
            .opacity(flag ? 1 : 0.3)
            .animation(.linear(duration: 1), value: flag)
    }
}
```

当我们点击按钮时，flag 状态属性将会改变，一个包含线性动画的新事物会被创建。SwiftUI 在视图树中向下进行遍历，在树中寻找遵守 Animatable 协议且与原先状态相比拥有不同 animatableData 值的那些视图或视图修饰器。在我们的例

子中，唯一改变的是 `opacity` 这个不透明度的值。`opacity` 视图修饰器默认就是可以动画的，它把不透明度的值作为 `animatableData` 暴露出来。SwiftUI 现在会开始在一秒的持续时间内，使用线性时间曲线将这个值从 1 插值到 0.3。在每秒 60 帧的情况下，这意味着视图修饰器的 `animatableData` 会被设置 60 次，它的值会以线性方式从 1 小幅递减到 0.3。



若想更深入了解幕后发生的事情，我们也可以自己实现一个支持动画的 `opacity` 修饰器。

```
struct MyOpacity: ViewModifier, Animatable {
    var animatableData: Double

    init(_ opacity: Double) {
        animatableData = opacity
    }

    func body(content: Content) -> some View {
        let _ = print(animatableData)
        content.opacity(animatableData)
    }
}
```

当然，在实现我们自己的修饰器时，我们还是使用了 SwiftUI 的 `opacity` 修饰器，但是这让我们有机会将 `animatableData` 属性插值后的值记录到控制台里。

让我们把上面的例子改写成使用我们自己的 opacity 修饰器的形式：

```
Button("Animate Opacity") { flag.toggle() }  
    .modifier(MyOpacity(flag ? 1 : 0.3))  
  
    .animation(.linear(duration: 1), value: flag)
```

当我们运行 app 时，会看到控制台里出现了很多值：

```
1.0  
0.988333511352539  
0.9766663551330566  
0.9649998664855957  
...  
0.33500013351440433  
0.3233336448669433  
0.3116664886474609  
  
0.3
```

对于 animatableData 属性的每一个值，视图修饰器的 body 都会被重新执行，这让我们有机会基于当前的动画状态更新视图。

有一种常见情况，我们会需要使用到 Animatable 协议，那就是想要实现视觉上需要回到开始的状态的动画。在 iOS 17 之前，这类动画面临的问题是，如果视图树不改变的话，我们就没有办法告诉 SwiftUI 应该如何使用动画效果。举个例子，如果我们想要在点击按钮时让其抖动，而按钮的最终位置还是和它开始抖动时一样的话，视图树中是不会有需要动画的状态改变的。在 iOS 17 中，我们可以使用阶段动画或者关键帧动画来实现这个效果，我们会在本章最后对它们进行讨论。

让我们来看看怎么用一个自定义的可动画视图修饰器 (也就是不依赖新的 iOS 17 API) 来实现这个抖动的动画，我们设计的抖动动画先从当前位置向左移动，然后向右移动，最后回到初始位置：

```
struct Shake: ViewModifier, Animatable {  
    var numberOfShakes: Double  
    var animatableData: Double {
```

```

    get { numberOfShakes }
    set { numberOfShakes = newValue }
}

func body(content: Content) -> some View {
    content
    .offset(x: -sin(numberOfShakes * 2 * .pi) * 30)
}
}

```

我们可以这样来使用这个修饰器：

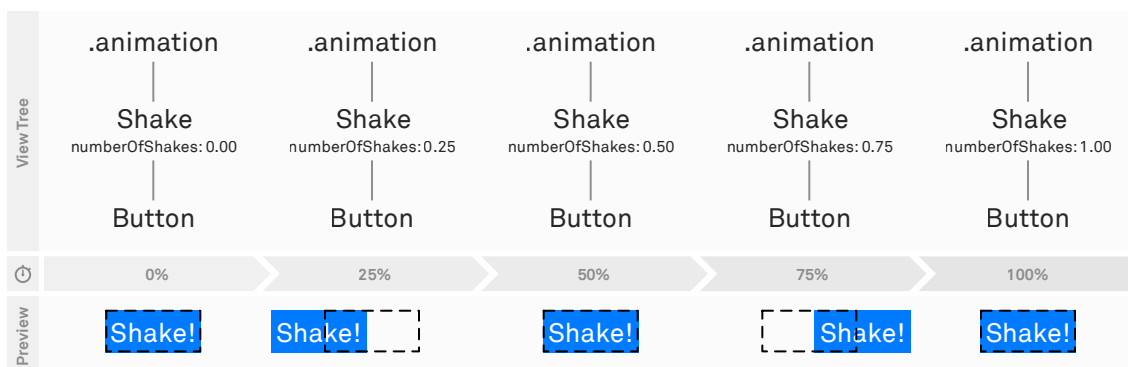
```

struct ContentView: View {
    @State private var shakes = 0

    var body: some View {
        Button("Shake!") { shakes += 1 }
        .modifier(Shake(numberOfShakes: Double(shakes)))
        .animation(.default, value: shakes)
    }
}

```

当用户点击这个抖动按钮时，shakes 状态数据被加一。新的视图树的结构和渲染树是相同的，只有属性发生了改变。在渲染树中，Shake 的 numberOfShakes 值为 0，而在新的视图树中，它的值是 1。



因为 Shake 上的 `animatableData` 属性只是简单地转发和承接了 `numberOfShakes` 属性，所以 Shake 上的 `animatableData` 也从 0 变成了 1。SwiftUI 将会把这个变更通过 `.default` 时间曲线从 0 到 1 进行插值。每次 `animatableData` 改变时，视图修饰器的 `body` 都会被重新执行，于是视图在屏幕上的按钮将被更新。

在 Shake 修饰器的 `body` 方法中，我们可以使用 `numberOfShakes` 值 (或者是 `animatableData` 值，它们是同样的东西) 来设定偏移量，以实现实际的抖动动画。我们把 `numberOfShakes` 值喂给一个正弦函数，再进行了一些简单的额外计算，来确保它首先向左移动 30 point，然后向右移动 60 point，最后回到初始位置。

我们可以使用 macOS 里 Apple 自带的 Grapher 应用程序来直观地重现这个视图修饰器中的数学计算。

如果我们需要在自定义的可动画视图或者视图修饰器中处理多个值时，可以使用 `AnimatablePair` 类型来将多个可动画的值组合成一个多元组。

由于 `animatableData` 属性在 `Animatable` 协议的扩展里有一个类型为 `EmptyAnimatableData` 的默认实现 (它意味着没有内容可以进行动画), 所以我们很容易会忘记实际去添加 `animatableData` 的实现。而且, 就算我们添加的 `animatableData` 属性没有满足 `VectorArithmetic`, 编译器也会看到原来的默认实现, 而不会向我们提出任何警告或错误, 这种时候, 我们的动画是不起作用的。

虽然这类“回到原点”的动画是实现自定义动画的典型示例, 但是通过这种方法我们还能达到不少其他效果。基本上, 基于 SwiftUI 提供给我们的这些插值, 我们可以完全自由地构建我们想要的任意动画。例如, 我们可以构建自定义时间曲线的动画, 或者在单一的 SwiftUI 动画中创建多段相连甚至交错的动画效果。

过渡

除了对已经在屏幕上的视图的动画属性进行动画, 我们还可以为视图的插入和移除设定动画。SwiftUI 将这些动画称为**过渡** (transition)。在 iOS 17 中, 框架提供了一套新的基于协议的 API, 但是功能上来说和以前并没有区别。我们会从解释 iOS 17 之前的 API 开始, 然后我们会对新旧 API 之间的区别进行讨论。

当我们使用动画来改变那些会导致视图插入或者移除的状态时, SwiftUI 会添加一个默认的 `.opacity` 过渡效果。让我们来看看这个最小示例, 它依据 `flag` 状态来插入或者移除一个矩形:

```
struct ContentView: View {
    @State private var flag = true

    var body: some View {
        VStack {
            Button("Toggle") {
                withAnimation { flag.toggle() }
            }
            if flag {
                Rectangle()
            }
        }
    }
}
```

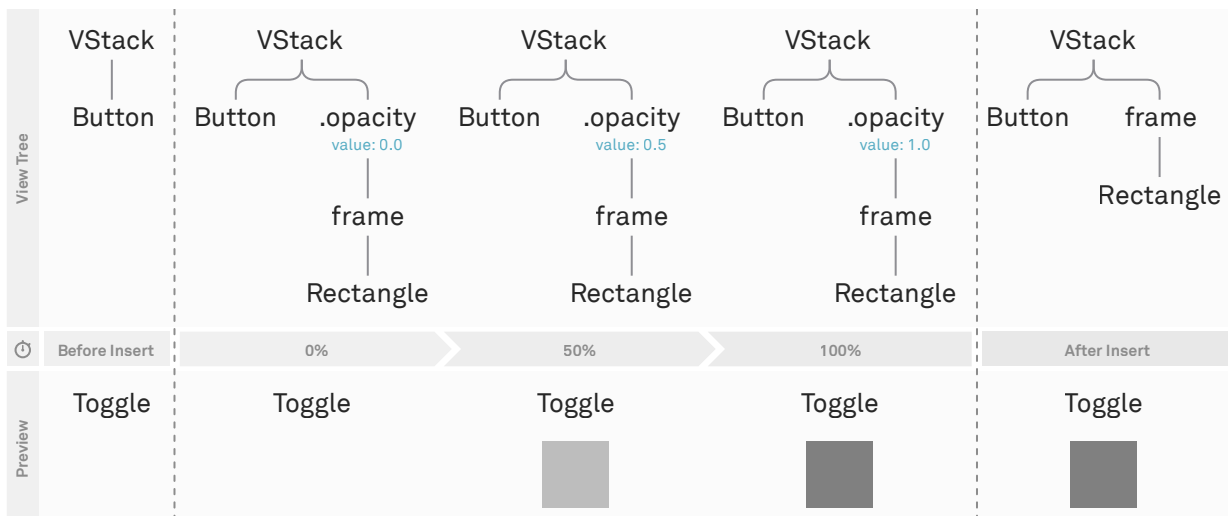
```
        .frame(width: 100, height: 100)
    }
}
}
```

在按钮的 `action` 中，我们使用了显式动画的方式来向事物添加了一个动画。点击时矩形的显示和消失会进行淡入淡出动画，这正是默认的 `.opacity` 过渡的表现。

要明确地指定过渡的动画效果，可以在矩形上添加一个 `.transition` 修饰器：

```
if flag {
    Rectangle()
        .frame(width: 100, height: 100)
        .transition(.opacity)
}
```

过渡有两种状态，分别是**活动** (active) 状态和**常时** (identity) 状态。当一个视图被插入时，过渡效果会从活动状态变为常时状态。当移除时，它从常时状态变为活动状态。对于 `.opacity` 过渡效果，活动状态时 `.opacity(0)` 修饰器会被添加到视图上，而在常时状态时则是 `.opacity(1)`。



我们可以在 SwiftUI 的 `AnyTransition.modifier(active:identity:)` API 里用过渡的这两个状态来构建自定义的过渡效果。该方法的两个参数都接受定义对应状态的视图修饰器。举个例子，如果我们想重新实现 SwiftUI 默认的 `.opacity` 过渡，我们可以这样写：

```

struct MyOpacity: ViewModifier {
    var value: Double
    func body(content: Content) -> some View {
        content.opacity(value)
    }
}

extension AnyTransition {
    static let myOpacity = AnyTransition.modifier(
        active: MyOpacity(value: 0),
        identity: MyOpacity(value: 1)
    )
}

```

在我们的 `.myOpacity` 所附着的视图被插入之前，`MyOpacity(value: 0)` 视图修饰器会被用来将视图置于过渡的初始活动状态。接着修饰器将会变为常时状态，在

插入动画完成后，`MyOpacity(value: 1)` 修饰器让视图变得完全不透明。对于移除操作，整个过程相反：视图从常时状态修饰器的“休息”位置开始，然后使用活动状态的修饰器进行动画，过渡到最终状态并消失。

注意，过渡效果的触发不一定非要通过使用 `if` 或 `switch` 语句来添加或移除一个视图。如果视图的身份标识改变了，过渡效果就会被触发。因为在 SwiftUI 的视角，这相当于 (由旧的身份标识定义的) 旧视图已经被移除了，同时 (带有新身份标识的) 新视图被加入。因此，使用 `.id(_)` 修饰器可以用来有效地触发过渡效果。

SwiftUI 提供了一系列内置的过渡效果，比如 `.slide`，`.move(edge:)` 和 `.scale(scale:anchor:)` 以及其他更多效果，我们还可以使用 `.combined(with:)` 过渡把它们组合起来同时运行。想为插入和移除指定不同的过渡效果，也可以使用 `.asymmetric(insertion:removal:)`：

```
if flag {
    Rectangle()
        .frame(width: 100, height: 100)
        .transition(
            .asymmetric(insertion: .slide, removal: .scale)
            .combined(with: .opacity)
        )
}
```

这个过渡使用 `.slide` 效果进行插入，使用 `.scale` 效果进行移除，并且为两者都组合上了 `.opacity` 过渡。

如果内置的过渡效果不够的话，可以使用我们上面提到过的 `.modifier(active:identity:)` 来实现完全自定义的过渡。比如，我们可以用它来构建一个模糊过渡。第一步，我们需要一个自定义的修饰器：

```
struct Blur: ViewModifier {
    var radius: CGFloat
```

```

func body(content: Content) -> some View {
    content
        .blur(radius: radius)
}

```

注意这里我们不需要把视图修饰器标记为 `Animatable`，因为我们依赖的内建 `.blur` 修饰器已经满足了 `Animatable`。我们也可以向 `AnyTransition` 添加一个扩展方法，来提供一个便于使用的 API：

```

extension AnyTransition {
    static func blur(radius: CGFloat) -> Self {
        .modifier(active: Blur(radius: 5), identity: Blur(radius: 0))
    }
}

```

在之前例子的情况下，我们可以这样来使用这个过渡：

```

if flag {
    Rectangle()
        .frame(width: 100, height: 100)
        .transition(.blur(radius: 5))
}

```

要牢记，过渡始终是和动画一起使用的。仅仅将一个 `.transition` 修饰器放到视图上，并不会导致视图用动画的方式进行改变。为了让过渡发生，我们总是需要在 `.transition` 修饰器旁边使用一个显式动画或者隐式动画。如果我们使用隐式动画，这个动画不应该放在被移除或者插入的视图树的子树上。举个例子，下面的代码是无法工作的：

```

if flag {
    Rectangle()
        .frame(width: 100, height: 100)
        .transition(.blur(radius: 5))
        .animation(.default, value: flag)
}

```

```
}
```

为了正确过渡，我们必须将动画放在一个稳定的地方，让它不被插入或者移除影响，比如：

```
VStack {  
  if flag {  
    Rectangle()  
      .frame(width: 100, height: 100)  
      .transition(.blur(radius: 5))  
  }  
}  
  
.animation(.default, value: flag)
```

这里，我们使用一个 `VStack` 来把 `if` 条件包进去，这样我们就可以从外面添加动画了。我们也可以用 `HStack` 或者 `ZStack`，在这里具体用什么容器不关键，我们只需要有个东西能把 `if` 条件包装起来，并在视图树的这个不变的部分添加动画就行。当然了，我们也可以用一個显式动画来替代。

在 iOS 17 中，过渡相关的 API 进行了修改，在保持功能相同的前提下，框架现在提供了一套更加清晰的接口。内置的过渡效果现在遵守 `Transition` 协议的类型。为了使用方便，这些过渡效果也可以通过 `Transition` 协议上的静态属性来访问，因此我们可以继续编写 `.transition(.opacity)` 或 `.transition(.scale(2))` 这样的代码。

我们现在可以通过创建一个遵守 `Transition` 协议的自定义类型，来创建自定义过渡了。活动状态和常时状态的概念已被明确的 `TransitionPhase` 类型所取代，该类型区分了 `willAppear`、`identity` 和 `didDisappear` 阶段。自定义的模糊过渡现在如下所示：

```
struct BlurTransition: Transition {  
  var radius: CGFloat  
  
  func body(content: Content, phase: TransitionPhase) -> some View {  
    content  
      .blur(radius: phase.isIdentity ? 0 : radius)  
  }  
}
```

```
}  
}
```

为了让这个过渡可以作为 `Transition` 的静态方法使用，我们可以编写：

```
extension Transition where Self == BlurTransition {  
    static func blur(radius: CGFloat) -> Self {  
        BlurTransition(radius: radius)  
    }  
}
```

在本节前面的部分，我们看到过组合多个过渡和为插入及移除指定不同过渡的方式。使用新的 API，我们可以把上面那个“使用 `.slide` 插入，使用 `.scale` 移除，并与 `.opacity` 组合起来使用”的例子进行重写：

```
if flag {  
    Rectangle()  
    .frame(width: 100, height: 100)  
    .transition(  
        AsymmetricTransition(insertion: .slide, removal: .scale)  
        .combined(with: .opacity)  
    )  
}
```

在自定义过渡里，除了创建不对称的过渡，我们还可以使用 `TransitionPhase` 来明确区分插入、常时和移除。

总的来说，我们认为新的 API 是一种进步，它与 SwiftUI 的其他部分更加契合了。然而，旧的过渡 API 也还没有被弃用，因此现有的代码在一段时间内还将继续正常工作，也不会产生警告。

基于阶段的动画

iOS 17 新引入的阶段动画器 (phase animator) 是实现“回到原点”动画最简单的方式。这些阶段是离散的数值，阶段动画器会依次遍历这些数值：一旦一个阶段完

成，系统就会继续执行下一个阶段。阶段动画器的持续时间可以是无限的 (当完成最后一个阶段时，它会回到初始阶段并继续)，或者也可以在每次触发值的变化时才循环遍历一次所有阶段。

例如，我们可以使用阶段动画器重新实现之前的抖动动画：

```
struct Sample: View {
    @State private var shakes = 0
    var body: some View {
        Button("Shake") {
            shakes += 1
        }
        .phaseAnimator([0, -20, 20], trigger: shakes) { content, offset in
            content.offset(x: offset)
        }
    }
}
```

初始阶段是数组中的第一个元素 (0)，按钮按照 `offset(x: 0)` 进行渲染。当 `shakes` 属性发生更改时，阶段动画器将其内部状态更改为下一个阶段 (-20)。内部状态变化会被包装在一个显式动画中。当该动画完成时，动画器再次使用动画将其状态更改到第三个阶段 (20)。最后，当该动画完成时，动画器将其状态再次更改为初始状态。因为我们提供了触发值，所以动画器会在此处停止。如果我们省略触发值的话，动画器将会进行循环。

注意，阶段动画器本身不会进行插值任何值。虽然上例中我们在阶段里使用了 `CGFloat`，但我们也可以使用具有三个情况的枚举。在上面的示例中，是 `offset` 修饰器为视图的位置设置了动画。在撰写本文时，对于 `Phase` 值唯一的要求是它必须符合 `Equatable` 协议。

每个阶段的变化都发生在单独的动画中，也就是说，我们无法指定一个总的时间曲线去让阶段动画遍历所有阶段，但我们可以使用 `animation` 参数为每个阶段指定一个时间曲线。单独的动画是与[上面](#)的自定义抖动动画的关键区别：在之前的

例子中，我们使用了单一动画来驱动整个抖动效果。

基于关键帧的动画

关键帧动画是一种描述动画的方法，动画在关键帧所指定的离散值之间进行过渡。与阶段动画不同，关键帧动画是使用完全独立的子系统构建的：它们不是建立在常规动画之上的。

在撰写本文时，关键帧动画似乎仍然存在一些问题。例如，有些情况下动画第一次可以正确运行，但第二次不正确。在未来的 SwiftUI 版本中关键帧动画肯定会持续得到改进。

关键帧动画会随时间对单一值进行动画。这个值可以是一个双精度浮点数，也可以是一个包含多个属性的结构体——SwiftUI 对值的类型没有进行任何限制。关键帧动画由一个或多个轨道 (track) 构成，每个属性有一个轨道。例如，想象一个包含旋转值和偏移量的结构体。我们可以使用各自的轨道独立地对每个属性进行动画。将单一轨道视为包含多个项的时间线很有帮助：每个项描述了移动到下一个值的方式。这个描述包括目标值，持续时间，以及如何进行插值。

当我们使用基于关键帧的动画重新实现我们的抖动动画时，我们只需要对单个的浮点数 (偏移量) 进行动画。因此，我们不需要指定多个轨道：

```
struct ShakeSample: View {
    @State private var trigger = 0

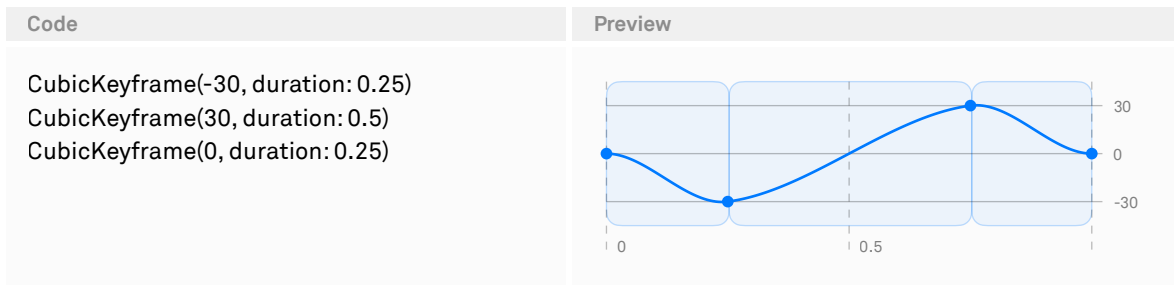
    var body: some View {
        Button("Shake") {
            trigger += 1
        }
        .keyframeAnimator(initialValue: 0, trigger: trigger) { content, offset in
            content.offset(x: offset)
        } keyframes: { value in
            CubicKeyframe(-30, duration: 0.25)
            CubicKeyframe(30, duration: 0.5)
        }
    }
}
```

```

        CubicKeyframe(0, duration: 0.25)
    }
}
}

```

每当触发值改变时，动画就会运行。偏移量将从 0 开始，第一个关键帧在开始的 0.25 秒内向 -30 进行动画。第二个关键帧在接下来的半秒内向 30 动画，最后一个关键帧向 0 动画。由于我们选择了三次插值 (cubic interpolation) 的方式，动画将 (从初始速度 0 开始) 平稳向上，并在关键帧之间平滑过渡。下面是显示随时间绘制的值的图表 (点表示关键帧)：

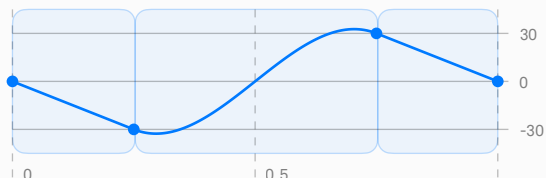


每个关键帧会通过检查它们周围的其他关键帧来计算曲线。例如，在上面的图中，我们看到了平滑的曲线穿过了所有的 CubicKeyframe 关键帧；而在下面，我们看到中间的 CubicKeyframe 关键帧考虑了第一个 LinearKeyframe (线性关键帧) 的结束速度和最后一个 LinearKeyframe 的初始速度，以此来创建一个平滑的过渡：

Code

```
LinearKeyframe(-30, duration: 0.25)
CubicKeyframe(30, duration: 0.5)
LinearKeyframe(0, duration: 0.25)
```

Preview



注意，虽然 `keyframeAnimator` 对值没有任何约束，但各个关键帧 (`LinearKeyframe`、`CubicKeyframe` 和 `MoveKeyframe`) 要求目标值需要遵守 `Animatable` 协议。

多个轨道

上面，我们只对单个 `CGFloat` 值进行了动画。对单个值动画是一种特殊情况；更一般地，关键帧动画允许我们提供多个同时运行的轨道。例如，我们可以扩展抖动动画，通过旋转效果来包含轻微的摆动。首先，我们定义一个结构体，该结构体具有两个属性，来持有我们想要进行动画的值：

```
struct ShakeData {
    var offset: CGFloat = 0
    var rotation: Angle = .zero
}
```

除了关键帧的定义之外，代码本身并没有发生太大变化。我们现在除了需要使用不同类型的初始值作为开始外，还需要在 `content` 闭包中添加旋转效果：

```
struct ShakeSample2: View {
    @State private var trigger = 0

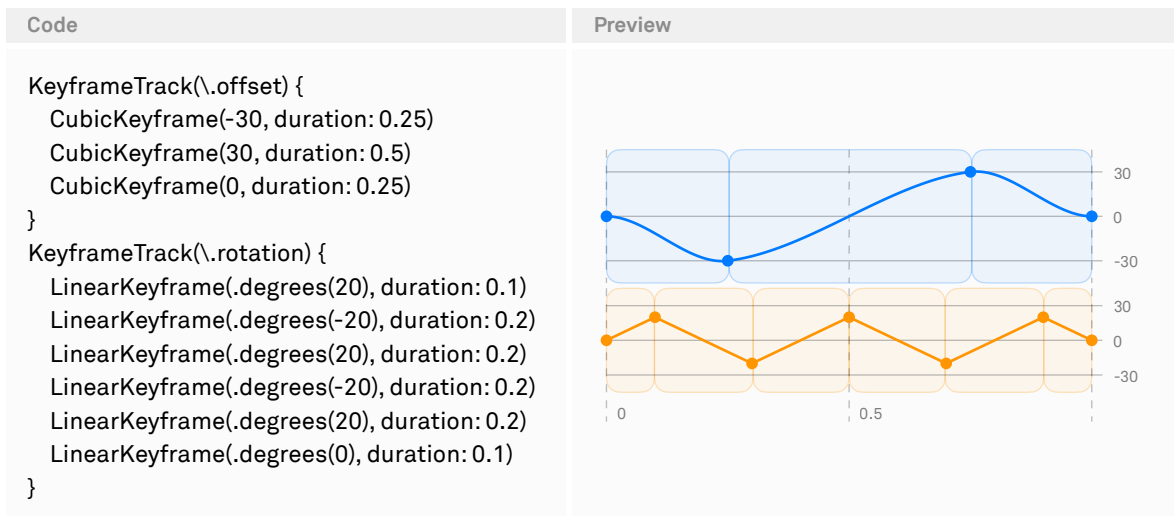
    var body: some View {
        Button("Shake") {
            trigger += 1
        }
        .keyframeAnimator(initialValue: ShakeData(), trigger: trigger) { content, data in
```

```

        content
            .offset(x: data.offset)
            .rotationEffect(data.rotation)
    } keyframes: { value in /* ... */ }
}
}

```

不过，关键帧的定义要略微复杂一些。现在，我们不再只是对一个值动画，而是使用 `KeyframeTrack` 构建了两个独立的轨道。SwiftUI 仍然会对一个单一的 `ShakeData` 值进行动画，但它使用第一个轨道来设定偏移量动画，用第二个轨道来设定旋转值动画。虽然单个轨道内的关键帧是按顺序运行的，但轨道本身是并行的。我们可以在下面的图表中看到这一点，在这个图表中：偏移量和旋转值被单独动画，但是最后它们被合并成一个单一的值。



一般来说，不同轨道的持续时间不必相同。如果其他轨道仍在运行时，某个轨道就已经结束了，那么这个结束的轨道将把最终值报告给动画的其余部分。请注意，在 `keyframeAnimator` 方法中，我们也会收到当前值。最初的时候这个值是 `initialValue`，但如果我们再次点击的话，它会是前一个动画的最终值。如果在前一个动画结束之前就触发了动画，那么闭包的 `initialValue` 将是当前（被插值）的

值。

到目前为止，我们使用的都是 `keyframeAnimator` 修饰器，不过也存在一个 `KeyframeAnimator` 视图。与修改视图不同，这个类型让我们构建一个原本就应该进行动画的视图。关键帧修饰器和关键帧动画视图都有类似的变体版本，可以控制动画是在每次触发值更改时运行一次，还是一直无限运行。

当内置的关键帧动画不能满足需求时，我们还可以使用 `KeyframeTimeline` 结构体。它接受一个初始值和关键帧的描述，但这些信息并不直接用来驱动视图的动画，而是让我们可以查询在特定时间点的值。如果我们想要针对 SwiftUI 视图以外的东西进行动画时，这就会非常有用。例如，我们可以使用它在快照测试 (snapshot test) 中对动画的不同阶段进行渲染，在 `TimelineView` 中进行动画，或者是记录动画的值 (这也是我们用来生成上面图表的方式)。

不过，关键帧动画也有一些限制：

- 由于关键帧动画的特性，它们只能作为隐式动画使用 (并不存在 `withKeyframeAnimation`)。
- 单独的关键帧只能对 `Animatable` 的值进行动画。例如，我们不能直接使用关键帧来动画一个 `Color` 值，因为它不遵守 `Animatable` (但我们可以动画一个 `Color.Resolved` 值)。
- 最后，在撰写本文时，基于触发器的动画仍然存在一些 bug。

进阶布局

7

在使用 SwiftUI 原始的基础布局时，我们无从得知视图被建议的尺寸，也无法获取同一层级视图或者子视图的尺寸信息。因此，如果想要依赖于这类信息来进行布局，我们就必须用到本章中所介绍的更低层级的技术。

在 iOS 16 和 macOS 13 中，我们可以使用 Layout 协议来创建自定义的布局容器。使用这个协议分为两步：

- 首先，使用 `sizeThatFits` 方法确定容器的尺寸。在该方法内部，我们通过子视图的代理，来访问这些子视图。视图代理允许我们通过向子视图提供不同尺寸的建议来测量子视图。
- 第二步中，将子视图放置在容器的尺寸内。同样，我们可以通过子视图的代理访问这些子视图。

目前，Layout 协议只允许我们测量和放置子视图，而不允许我们对子视图进行筛选或是添加额外视图（比如，在子视图间插入分隔线等）。比如，我们无法轻松地用 Layout 协议构建出自己的 `ViewThatFits` 类型。虽然有一些技巧可以进行实现，但了解这些限制是很有益的。

和很多新 API 类似，Layout 协议在 2022 年之前发布的平台上是不可用的。当我们需要支持这些平台时，我们仍然可以使用像是几何读取器 (GeometryReader)、首选项 (Preferences) 和 overlay 来构建出许多自定义布局。我们将在本章后面讨论这些技术。

Layout 协议

使用 Layout 协议，我们可以创建自定义的容器视图，让它们根据我们所编写的算法来布局子视图。例如，我们可以使用该协议构建类似砖墙布局 (masonry)、流式布局 (flow) 或环形布局 (circular) 等自定义布局。我们还可以使用该协议来包装和调整已有的布局。

在本节里，我们会构建一个流式布局。它和 `HStack` 不同：`HStack` 总是将它的子视图放在同一个单行里，而流式布局则在下一个视图不再适配当前行的时候进行换行，它的行为和文本视图中的文本在下一个单词无法适配时会进行换行类似。

使用 Layout 协议，我们将按照以下步骤实现流式布局：

1. 通过向每个子视图建议 nil×nil 尺寸，来询问它们的**理想尺寸**。
2. 根据容器自身的尺寸，计算所有子视图的位置。
3. 基于步骤 2 的结果，将每个子视图放在计算出的位置上。

步骤 2 中的基于行的布局算法的参数包含尺寸列表 (其中是步骤 1 得到的各个理想尺寸)、间距和容器宽度。它返回一个 frame 的数组 (其中每个元素对于着一个输入尺寸)。它的工作方式如下：

1. 我们使用一个初始位置为 (0, 0) 的 CGPoint 来追踪当前位置。其中 x 分量是当前行内的水平位置，每添加一个视图都会改变。y 分量是当前行的顶部，它仅在开始新的一行时才会改变。
2. 我们遍历所有的子视图。
 1. 如果子视图无法适应当前行，我们通过将当前位置的 x 分量重置为零并将 y 分量增加当前行的高度加上间距来开始新的一行。
 2. 我们将当前位置用作此子视图矩形的原点。
 3. 我们将当前位置的 x 分量增加子视图的宽度并添加间距。

我们可以独立于 Layout 协议编写这个算法：

```
func layout(
    sizes: [CGSize],
    spacing: CGSize = .init(width: 10, height: 10),
    containerWidth: CGFloat
) -> [CGRect] {
    var result: [CGRect] = []
    var currentPosition: CGPoint = .zero

    func startNewline() {
        if currentPosition.x == 0 { return }
        currentPosition.x = 0
        currentPosition.y = result.union().maxY + spacing.height
    }
}
```

```

for size in sizes {
    if currentPosition.x + size.width > containerWidth {
        startNewline()
    }
    result.append(CGRect(origin: currentPosition, size: size))
    currentPosition.x += size.width + spacing.width
}
return result
}

```

使用这个算法，我们的实际布局的实现就变得非常简短了。sizeThatFits 方法使用所有理想尺寸和建议的容器宽度来调用 layout，并返回所有 frame 的并集的大小。这一行为和 ZStack 中会将所有子视图的并集大小作为其自身的大小进行报告是一样的。union 方法是我们编写的一个自定义扩展。

placeSubviews 方法首先计算了所有的 frame，然后将每个子视图放置在其对应 frame 的原点处：

```

struct FlowLayout: Layout {
    func sizeThatFits(proposal: ProposedViewSize, subviews: Subviews, cache: inout ())
        -> CGSize
    {
        let containerWidth = proposal.replacingUnspecifiedDimensions().width
        let sizes = subviews.map { $0.sizeThatFits(.unspecified) }
        return layout(sizes: sizes, containerWidth: containerWidth).union().size
    }

    func placeSubviews(in bounds: CGRect, proposal: ProposedViewSize,
        subviews: Subviews, cache: inout ())
    {
        let subviewSizes = subviews.map { $0.sizeThatFits(.unspecified) }
        let frames = layout(sizes: subviewSizes, containerWidth: bounds.width)
        for (f, subview) in zip(frames, subviews) {
            let offset = CGPoint(x: f.origin.x + bounds.minX, y: f.origin.y + bounds.minY)
            subview.place(at: offset, proposal: .unspecified)
        }
    }
}

```

}

这里有一个有趣的效果，该布局并不总是会完全变为被建议的宽度，类似于 Text 被分成多行时的情况。假设我们设定了10 point 的间距，以及三个理想宽度为100 的子视图，当我们建议宽度为250时，考虑以下情况：我们的布局无法将所有三个项目都放在同一行上，它会将第三个项目放到第二行里。因此，它所报告的宽度为210 (最宽行的宽度)。

Code	Preview
<pre>FlowLayout { ForEach(Array(0...10), id: \.self) { idx in Rectangle() .fill(Color.random) .frame(width: .random(in: 10...35)) } } .border(.blue) .frame(width: 125) .border(.black)</pre>	

有很多方式可以扩展这个布局。下面是我们能想到的一些点子：

- 增加同一行内对齐方式的参数。目前，我们在单行中的所有子视图都是顶部对齐的，但我们其实可以支持任意的垂直对齐方式。为此，我们可以通过 (使用代理来) 询问每个子视图，来获取它们的对齐值。
- 支持水平对齐 (目前我们的行都是前部对齐的)。
- 让外部可以对间距进行控制。
- 增加对特殊视图 (例如标题行) 的支持，这些视图始终有自己的行。
- 在一个 (或两个) 轴上建议的尺寸为 nil 时，提供更好的默认行为。

这些功能中的大多数添加起来并不难，但它们所需要的代码量超出了本书的合理

范围。本章中，我们就不去进行这些扩展了；相反，我们会一起来看看 SwiftUI 里当前现成可用的一些 API，并研究它们如何帮助我们扩展流式布局。让我们从 Layout 协议上的 API 开始：

- 我们可以添加一个自定义的 Cache 类型，用它来在不同的方法之间共享计算结果。例如，在 `sizeThatFits` 中，我们可以使用缓存来存储计算出的子视图的 frame，并在 `placeSubviews` 中重用它们。为此，我们可以选择重写 `makeCache` 方法，该方法会被用来初始化或者构建缓存。然而，正如文档中所述，通常我们并不需要构建自己的缓存。在继续朝着缓存的路线深入之前，我们应当先对是否确实存在性能上的问题进行验证。
- 我们可以使用 `explicitAlignment(...)` API 来提供显式的对齐参考线。与 `.alignmentGuide` API 不同，我们还可以通过查询子视图 (通过它们的代理) 来计算它们的对齐参考线。例如，我们可以通过重写 `explicitAlignment` API 的默认实现，将 `firstFlowBaseline` 和 `lastFlowBaseline` 对齐方式的参考线暴露给外界。

`LayoutSubview` 类型也给我们提供了一系列有用的 API，让我们可以查询子视图：

- 框架并没有限制我们只能调用一次 `sizeThatFits`；实际上，我们可以多次调用它。这种探测正是 `stack` 视图用来确定子视图的灵活性的方式。
- 除了 `sizeThatFits`，我们还可以向子视图提供一个尺寸，并获取它的 `ViewDimensions`。`ViewDimensions` 类型类似于 `CGSize`，除了具有宽度和高度，它还可以被用来查询子视图的对齐参考线。例如，我们可以使用这个结果在水平方向上对齐我们的子视图，同时尊重它们的自定义对齐参考线。
- 我们可以询问子视图所希望使用的间距值，并在没有显式指定间距的情况下使用这些值。
- 我们可以询问视图的优先级。这对于流式布局来说并没有太多意义，但在实现 (例如) 自定义的 `stack` 布局时会很有用。
- 子视图还可以添加自定义的布局值。这些值可以由容器通过视图代理的下标来读取。例如，我们可以使用这个特性来让子视图声明一个类型为布尔值的

heading 布局值，并在容器视图中读取它。

和 AnyView 相似，布局中也存在 AnyLayout 类型，用它可以抹杀掉某个布局的具体类型。此外，它还可以被用来动态地在不同的布局之间进行切换。当这种切换是以动画的一部分完成时，子视图的 frame 变化也将以动画方式呈现。

限制

Layout 协议是在 iOS 16 和 macOS 13 中添加的，因此当针对旧平台进行开发时，我们无法使用它。然而，在不使用 Layout 协议的情况下，我们依然可以使用首选项和几何读取器来构建几乎任意的自定义布局，我们将在下一节中探讨这些内容。

另一个限制是我们无法直接与子视图进行通信：例如，我们无法对子视图使用任何修饰器、读取它们的 preference 或执行其他需要 View 协议的任务。如前所述，所有子视图都要被放置，我们无法从子视图列表中移除或添加子视图。例如，我们无法插入像是分隔线这样的装饰性视图，也无法使用筛选器来跳过某些子视图。

基于首选项的布局

正如我们在前一节中看到的，通过 Layout 协议我们能够提议不同的尺寸并将视图放置在容器内。使用首选项、几何读取器和 ZStack，几乎可以复制所有这些行为，甚至还能实现更多内容。

几何读取器


使用几何读取器 (GeometryReader)，我们可以测量被建议的尺寸。几何读取器无条件接受被建议尺寸，并通过一个几何代理 (GeometryProxy) 将该尺寸报告给它的视图构建器闭包。通过几何代理，我们可以访问几何读取器的大小 (它的类型是 CGSize，表示读取器被建议的尺寸，其中 nil 被默认值 10 点替代)。我们还可以读取安全区域的内移量 (inset)，在特定坐标空间中读取 frame，以及解析锚

点等。

我们经常会看到人们建议尽量不要使用几何读取器。我们认为这种情绪的主要原因是几何读取器被误解了。例如，考虑一个包含三个 Text 视图的 HStack：当我们将其其中一个 Text 视图包装在几何读取器内时，我们的布局将完全不同。这是因为被包装的这个 Text 视图现在会接受被建议的尺寸，而不是基于其内容来确定尺寸。

当使用几何读取器测量视图的尺寸时，我们建议将其放置在 overlay 或 background 中。正如我们在[布局章节](#)中讨论的那样，overlay 的次要子视图所得到的建议尺寸就是 overlay 的主要子视图的尺寸。通过将几何读取器放置在 overlay 内，我们就可以测量主要子视图的尺寸，同时不会影响布局。


一旦获得了尺寸，我们就可以做很多事情了。例如，假设我们想要在背景中显示一个圆角矩形，并根据几何读取器的尺寸调整圆角半径：

Code	Preview
<pre>myView .padding() .background { GeometryReader { proxy in let radius: CGFloat = proxy.size.width > 50 ? 10 : 5 RoundedRectangle(cornerRadius: radius) .fill(.blue) } } }</pre>	

然而，当我们想要获取某个视图的尺寸，并在它的祖先视图中使用这个尺寸时，就比较困难了：因为几何读取器只是另一个视图，我们无法在视图构建器闭包中设置状态属性或以其他方式修改状态。

在处理单个视图的测量时，我们可以使用 onAppear 结合 onChange(of:) 来绕过这个限制。比如，假设我们希望一个视图使用其理想尺寸，但我们希望将其显示

在宽度为 100 的 frame 内，并在其绘制超出该框架时进行检测。我们可以创建一个 Text，通过使用 `fixedSize` 修饰器确保它使用理想尺寸。然后，我们添加一个宽度为 100 的 frame，以确保布局尺寸宽度始终恰好为 100。我们可以使用带有几何读取器的 `overlay` 来测量文本的尺寸。由于几何读取器需要某种视图作为它的内容，我们这里使用了一个透明的颜色。当渲染树中的节点创建 (并调用 `onAppear`) 或者文本的尺寸越过 100 点阈值 (并调用 `onChange`) 时，我们将为状态属性设置一个新值：

Code	Preview
<pre>struct ContentView: View { var text: String @State private var overflows: Bool = false var body: some View { Text(text) .fixedSize() .overlay { GeometryReader { proxy in let tooWide = proxy.size.width > 100 Color.clear .onAppear { overflows = tooWide } .onChange(of: tooWide) { _ in overflows = tooWide } } } .frame(width: 100) .border(overflows ? Color.red : Color.clear) } }</pre>	

注意，我们需要同时使用 `onAppear` 和 `onChange(of:)`：前者仅在节点首次创建时触发，后者则在值实际发生更改时触发 (但在视图首次渲染时不会触发)。如果

我们经常需要使用这种模式，我们可以创建一个辅助函数来组合这两个修饰器：

```
extension View {  
    func onAppearOrChange<Value: Equatable>(of value: Value,  
        perform: @escaping (Value) -> ()) -> some View  
    {  
        self  
            .onAppear { perform(value) }  
            .onChange(of: value, perform: perform)  
    }  
}
```

如果我们的目标平台是 iOS 17/macOS 14，就可以使用带有 initial 参数的 onChange 新变体，它指定了值在初始更改时是否应该调用 perform 闭包：

```
Color.clear  
    .onChange(of: tooWide, initial: true) {  
        overflows = tooWide  
    }
```

在许多情况下，这是测量单个视图的尺寸并将这个测量传播到祖先视图的最简单的解决方案。然而，当我们需要测量多个相关视图时，使用 onAppear 和 onChange(of:) 将面临可扩展性的问题。相反，我们可以使用**首选项**。

首选项

在 SwiftUI 中，首选项是和环境所对应的概念。环境值负责将值沿着视图树向下传播，而**首选项则将值沿着树向上传播**。例如，在流式布局里，我们可以测量每个子视图的大小，并使用首选项将其传播到容器视图中。此外，首选项还允许我们定义这些值应当如何合并。例如，在流式布局的例子中，我们希望将每个单独的测量组合成一个尺寸数组或字典。

与环境类似，首选项也使用了一些稍微复杂的语法来确保类型安全。

PreferenceKey 协议需要一个默认值 (以对应首选项没有被设置的情况)。同时它还需要一个 reduce 方法，该方法用来组合两个值。例如，如果我们想测量多个

视图的最大宽度，我们可以设定一个 `CGFloat` 值，并通过取最大值来实现 `reduce`。如果我们对某个单个的值感兴趣，而这个值可能出现在子树的任何地方，那我们可以使用可选值来建模，并获取第一个非 `nil` 的值。

在我们的例子中，我们想要测量所有子视图的尺寸。我们可以将它建模为一个 `CGSize` 的数组。这样一来，我们的 `PreferenceKey` 看起来会是下面这样：

```
struct SizeKey: PreferenceKey {
    static var defaultValue: [CGSize] = []

    static func reduce(value: inout [CGSize], nextValue: () -> [CGSize]) {
        value.append(contentsOf: nextValue())
    }
}
```

要测量单个视图，我们可以借鉴前面一节中的知识，在 `overlay` 中使用几何读取器进行测量。在那里，我们使用首选项将此值传播到视图树的上层：

```
extension View {
    func measureSize() -> some View {
        overlay {
            GeometryReader { proxy in
                Color.clear
                    .preference(key: SizeKey.self, value: [proxy.size])
            }
        }
    }
}
```

要测量多个视图的尺寸，我们使用 `ForEach` 生成多个子视图，并在每个子视图上调用 `measureSize`。此外，我们还在测量尺寸的视图树上游的某个位置添加了 `onPreferenceChange(_:perform:)` 修饰器，并打印出首选项值：

```
ZStack(alignment: .topLeading) {
    ForEach(0..<5) { ix in
        Text("Item \(ix)" + String(repeating: "\n", count: ix/2))
            .padding()
    }
}
```

```

        .measureSize()
    }
}

.onPreferenceChange(SizeKey.self) { print($0) }

```

当在 macOS 上运行时，print 语句将打印出一个包含所有尺寸的数组：

```
[(70.5, 48.0), (68.5, 48.0), (70.0, 64.0), (70.5, 64.0), (70.5, 80.0)]
```

每个 .preference 修饰器的值都向上传递到视图树中，PreferenceKey 的 reduce 方法则被用来将来自多个子视图的值组合起来。一旦遇到带有相同键的 .onPreferenceChange(_ key:perform:) 修饰器，框架就会使用 .onPreferenceChange 内子树中的累加值来调用 perform 闭包。当影响首选项的状态更改发生时，此过程将从空白状态重新开始。

实现流式布局的下一个步骤，是使用这些尺寸来实际布局子视图。

将所有内容整合

一旦有了尺寸，我们就可以重用本章稍早前的 layout 方法了。那个方法接受的正是一系列输入尺寸，并为每个输入尺寸返回对应的 frame。另外，它还要求容器的宽度：

```

func layout(
    sizes: [CGSize],
    spacing: CGFloat = 10,
    containerWidth: CGFloat

) -> [CGRect]

```

第一步，我们需要测量容器的宽度。在不使用 Layout 协议的情况下，我们无法轻松地测量被建议尺寸，也就无法根据它来计算出我们自己的尺寸，因此我们需要一些变通方法来实现这一点。首先，我们会创建一个外部的 ZStack，它负责将一个零高度的几何读取器 (它不会影响垂直布局的尺寸) 和我们的容器视图 ZStackFlowLayout 组合在一起。在几何读取器内部，我们测量建议的尺寸并将

其设置为状态属性。在容器视图上，我们将 `minWidth` 设置为 0，将 `maxWidth` 设置为 `.infinity`。这是一种让视图“恰好成为被建议宽度”的奇特方式 (有关更多的详细信息，请参见[布局](#)章节)。换句话说，这个外部的 `ZStack`、和它里面的 `GeometryReader` 以及 `ZStackFlowLayout` 都将使用被建议宽度。因此，外部视图的高度将成为 `ZStackFlowLayout` 的高度：

```
struct ZStackFlowLayoutExample: View {
    @State private var containerWidth: CGFloat?
    var body: some View {
        ZStack {
            GeometryReader { proxy in
                let width: CGFloat = proxy.size.width
                Color.clear
                    .onAppearOrChange(of: width) { containerWidth = $0 }
            }
            .frame(height: 0)
            ZStackFlowLayout(containerWidth: containerWidth ?? 0)
                .frame(minWidth: 0, maxWidth: .infinity)
        }
    }
}
```

现在有了这个包装视图之后，我们就可以实现 `ZStackFlowLayout` 视图了。我们定义一个状态属性来保存子视图们的尺寸，并将容器的宽度作为参数传入。此外，`subview(for:)` 方法会为流式布局中的每个索引生成带有背景的背景文本：

```
struct ZStackFlowLayout: View {
    @State private var sizes: [CGSize]? = nil
    var containerWidth: CGFloat
    let subviewCount = 5

    func subview(for index: Int) -> some View {
        Text("Item \(index)" + String(repeating: "\n", count: index/2))
            .padding()
            .background {
                RoundedRectangle(cornerRadius: 5)
                    .fill(Color(hue: .init(index)/10, saturation: 0.8, brightness: 0.8))
            }
    }
}
```



```

    }

    var body: some View {
        ...
    }
}

```

在 `body` 属性中，我们首先根据尺寸来计算出所有的偏移量。然后，我们创建一个具有 `topLeading` 对齐方式的 `ZStack`，并使用对齐参考线来移动每个子视图。因为 `ZStack` 的布局大小将是所有子视图 `frame` 的并集，所以这将直接向布局系统报告正确的尺寸。这在将流式布局与其他视图结合在一起使用时 (例如将其放入 `ScrollView` 中) 非常重要。当 `sizes` 参数为 `nil` 时，我们将每个视图都放置在 `.zero`。我们使用 `fixedSize` 来渲染每个子视图，以确保子视图的大小不依赖于我们测量的任何内容，从而防止无限的布局循环。然后，我们测量子视图的尺寸并设置对齐参考线。注意，要将子视图移动到例如 `100×50` 的位置，我们必须将先头对齐参考线设置为 `-100`，将顶部对齐参考线设置为 `-50`。然后，我们在最顶层的视图上添加一个 `onPreferenceChange`，来读取所有首选项值：

```

var body: some View {
    let offsets = sizes.map {
        layout(sizes: $0, containerWidth: containerWidth).map { $0.origin }
    } ?? Array(repeating: .zero, count: subviewCount)

    ZStack(alignment: .topLeading) {
        ForEach(0..

```

我们可能会认为，不去把这些尺寸存储起来，而是直接在 `onPreferenceChange`

中计算布局，会更有效率。虽然确实可行，但它会引入一个问题：一旦我们这样做，视图就不再会响应 `containerSize` 的变化。按照我们现在的方式编写 `body`，视图将在容器大小改变时，或我们测量到不同的尺寸时被无效化。尽管我们的子视图是静态的，但当环境中的某些内容，例如动态类型 (dynamic type) 等，发生变化时，视图的尺寸仍然可能会改变。

基于 `ZStack` 的算法来构建流式布局，比我们在使用 `Layout` 协议时展示的第一个版本要复杂得多。此外，它始终会使用被建议宽度，就算其中只有一个项目也依然如此。但是，它也向我们展示了使用自 `SwiftUI` 第一个版本以来就可用的工具来构建复杂布局的可能性。

其中一些细微之处可能不是立刻显而易见的，在我们自己编写这些代码时也很容易被忽略。例如，当上面的视图被首次渲染时，`onPreferenceChange` 在布局阶段的某个时间发生，`@State` 属性会被无效化。甚至在渲染当前帧之前，视图的 `body` 就会被重新执行。这可以防止闪烁，通常非常有用。然而，我们还必须确保在布局中不会引入意外的循环。当我们在 `ForEach` 中去掉 `fixedSize` 时，实际上就可能会陷入循环中，在此循环里，一个子视图会被测量，报告出不同的尺寸，导致布局不断地被无效化。当出现此问题时，通常在控制台中会得到 “Bound preference x tried to update multiple times per frame.” 的警告信息。

同样，当我们尝试在 `onPreferenceChange` 中计算位置时，我们的代码看起来第一次运行时是正确的，但它无法支持不断变化的容器宽度。一个简单的验证方法是将代码作为 `macOS` 应用程序的一部分运行：调整窗口大小应该会导致布局中的条目“流动”起来才是正确行为。

可变数量视图

我们的 `ZStack` 和基于首选项的流式布局还有最后一个缺点，那就是我们只能支持单一类型的子视图。当然，我们可以用更复杂的方式 (例如，通过循环遍历一个 `Identifiable` 项目数组) 来替换 `ForEach`。然而，使用 `Layout` 时，我们是传递任意的视图列表的。例如，这里我们添加了六个子视图——前五个是从 `ForEach` 中添加的，最后一个作为视图构建器的一部分添加的。这在基于首选项的布局中不太容易实现：

```

FlowLayout() {
    ForEach(0..<5) { ix in
        Text("Item \($ix)")
    }
    RoundedRectangle(cornerRadius: 10)
        .frame(width: 100, height: 100)
}

```

在基于首选项的布局中，我们可以使用一个名为 [VariadicView](#) 的带下划线的半公开 API 来实现这种语法。这为我们提供了一个用来访问底层视图的视图代理的集合，我们可以用 `ForEach` 来使用该集合。这使我们能够构建出一套与基于 `Layout` 协议的版本非常相似的 API。虽然完整的代码超出了本书的范围，但已经完成的版本可以在[这里](#)找到。

`Preference` 不仅仅局限于布局中的使用，我们还可以将它们用于其他用途。例如，如果我们要实现自己的 `navigation` 视图的替代方案，我们可以使用首选项来构建 `navigationTitle` API。通常情况下，当我们要构建一个可重用组件来包装其他视图时，这些视图就可以使用首选项的方式与组件进行通信。

坐标空间

在处理几何读取器时，我们可以在不同的坐标空间中测量视图的几何属性。框架为我们提供了两种内置的坐标空间：全局坐标空间和本地坐标空间。此外，我们也可以定义自己的坐标空间。考虑以下视图，我们在 `stack` 上明确定义了一个坐标空间：

```

struct ContentView: View {
    var body: some View {
        VStack {
            Text("Hello")
            Text("Second")
            .overlay { GeometryReader { proxy in
                let _ = print([
                    proxy.frame(in: .global).pretty,
                    proxy.frame(in: .local).pretty,

```

```

        proxy.frame(in: .named("Stack").pretty)
    ].joined(separator: "\n"))
    Color.clear
  }}
}
.coordinateSpace(name: "Stack")
}
}

```

当我们在 iOS 模拟器上运行上面的代码时，会打印三个不同的值：

- 全局坐标空间中的原点是 (167, 438)。这是距离屏幕顶部和前侧边缘的距离。
- 本地坐标空间中的原点是 (0, 0)。本地始终是指相对于视图的本地坐标空间，在这种情况下，视图是几何读取器，它与第二个文本标签的坐标空间相同 (而不是容器的坐标空间)。
- “Stack” 坐标空间内的原点是 (0, 20)，因为带几何读取器的视图是 stack 里的第二个子视图。

虽然原点不同，但上面的视图所打印的尺寸都是一样的。不过，如果考虑我们在使用一个 `scaleEffect` 后的情况：

```

Text("Hello")
.overlay { GeometryReader { proxy in
    let _ = print([
        proxy.frame(in: .global).size.pretty,
        proxy.frame(in: .local).size.pretty,
    ].joined(separator: "\n"))
    Color.clear
  }}

.scaleEffect(0.5)

```


正如我们在布局章节中讨论过的那样，`scaleEffect` 只会改变视图的绘制方式，但不会改变视图的布局。然而，向几何代理询问相对于特定坐标空间的视图 frame 时，将会考虑到该坐标空间内的 `effect` 修饰器。在上面的示例中，以全局坐标空

间测量到的尺寸，将只有本地坐标系中测量到的尺寸的一半 (更准确地说：宽度和高度都是其本地对应尺寸的一半)。

锚点

锚点 (Anchors) 是建立在几何读取器、首选项和坐标空间之上的概念。实质上，锚点是一个在**全局坐标空间**中测量得到的几何值 (CGRect、CGSize 或 CGPoint) 的包装。几何代理对锚点有特殊的支持，它可以自动将几何值转换到**本地坐标空间**中。

比如，考虑构建一个新用户引导流程，在这个流程中，我们希望通过在当前项目周围放置一个椭圆来突出显示界面的某些部分：

Code	Preview
<pre>VStack { Button("Sign Up") {} Button("Log In") {} .overlay { Ellipse() .strokeBorder(Color.red, lineWidth: 1) .padding(-5) } } .border(Color.green)</pre>	

很多时候，椭圆可能会被登录按钮同层级中的其他视图或者是位于视图树更高位置的视图所遮挡。在这种情况下，即使椭圆的位置可能是由较深层的视图所确定的，我们还是希望将它绘制在靠近顶部的视图层级上。

为了实现这一点，我们可以使用首选项来传递登录按钮的 frame，然后将椭圆作为覆盖层绘制在整个视图树的顶部。作为第一步，我们创建一个用于传递锚点的键：

```

struct HighlightKey: PreferenceKey {
    typealias Value = Anchor<CGRect>?
    static var defaultValue: Value

    static func reduce(value: inout Value, nextValue: () -> Value) {
        value = value ?? nextValue()
    }
}

```

接下来，我们把绘制椭圆的操作替换为设置锚点首选项。这将在视图树中把 (在全局坐标空间中测量到的) 第二个按钮的位置和大小进行传递：

```

VStack {
    Button("Sign Up") {}
    Button("Log In") {}
    .anchorPreference(key: HighlightKey.self, value: .bounds, transform: { $0 })
}

```

现在，我们来考虑如何在 VStack 的外部添加绘制代码。在本章早些时候，我们使用过 onPreferenceChange 来获取首选项的值，但在这个例子中，我们可以使用一个更简单的 API，来让我们可以在 overlay 内部访问到首选项的值。这样可以避免使用单独的状态属性。在 overlay 里，我们使用几何读取器及其代理来解析锚点，将 Anchor<CGRect> 转换为本地坐标空间内的 CGRect：

```

VStack {
    ...
}
.border(Color.green)
.overlayPreferenceValue(HighlightKey.self) { value in
    if let anchor = value {
        GeometryReader { proxy in
            let rect = proxy[anchor]
            Ellipse()
                .strokeBorder(Color.red, lineWidth: 1)
                .padding(-5)
                .frame(width: rect.width, height: rect.height)
                .offset(x: rect.origin.x, y: rect.origin.y)
        }
    }
}

```

```
    }  
  }  
}
```

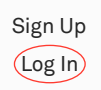
注意，锚点的存在只是为了编写方便；我们也可以通过设定自定义的坐标空间 (并在坐标空间内测量所有内容)，或者通过在全局坐标空间内测量所有内容，来编写相同的代码。然而，这两种替代方法都会更加繁琐，而且可能更容易出错。

以我们的经验来说，当需要在不同的坐标空间之间转换坐标或尺寸时，锚点是最为有用的。这通常发生在我们需要处理位于视图层次结构不同深度的多个视图的时候。在本章前面提到的流式布局示例中，所有子视图都共享相同的父视图，因此在那里我们不需要在不同的坐标空间之间执行任何转换，锚点也就没有什么用武之地了。

几何匹配效果

几何匹配效果，也就是 `matchedGeometryEffect` API，用于使一个或多个“目标”视图的位置、尺寸或 `frame` (也就是位置和尺寸的组合) 与另一个“源”视图相同。它的实现包括了测量源视图的尺寸和位置，然后将相同的尺寸建议给目标视图，并为它们设置偏移。属于同一个匹配几何组的源视图和目标视图，可以位于视图树中的任何位置，它们使用命名空间和一个次要标识符 (可以是任何可散列的值) 来进行标识。

举例来说，我们可以使用几何匹配效果来重新实现上面的高亮椭圆，而不需要手动通过首选项来传递锚点：

Code	Preview
<pre>struct ContentView: View { @Namespace var namespace let groupId = "highlight" var body: some View { VStack { Button("Sign Up") {} Button("Log In") {} .matchedGeometryEffect(id: groupId, in: namespace) } .overlay { Ellipse() .strokeBorder(Color.red, lineWidth: 1) .padding(-5) .matchedGeometryEffect(id: groupId, in: namespace, isSource: false) } } }</pre>	

因为我们为 Stack 中的按钮和带有边距的描边矩形指定了同样的命名空间和共用的标识符，所以这两个视图现在属于相同的几何组了。在本例中，我们使用了字符串作为标识符，但我们也可以使用任何其他哈希值，比如整数或 UUID。

在一个几何组中，只允许存在一个源视图。isSource 的参数默认为 true，因此我们需要在 overlay 中明确地为椭圆指定 false，因为匹配是从按钮到椭圆来进行的。

实际上，我们可以使用本章中讨论的技术自己实现 `matchedGeometryEffect` 的 API。我们需要在顶层的某个地方明确使用一个视图修饰器，来从首选项中读取源视图的几何信息，并通过环境将它们传递到目标视图中。作为框架来说，SwiftUI 是有能力自动处理这些操作的。除此之外，创建一个几何匹配效果的 API 并没有什么神奇之处。

当椭圆和按钮的 `frame` 相匹配时，我们可以将其视为向椭圆插入了一个 `frame` 和一个偏移量 (这不一定是真实的实现方式，但作为思维模型来说很合适)：

```
VStack {  
    ...  
}  
.overlay {  
    Ellipse()  
        .strokeBorder(Color.red, lineWidth: 1)  
        .padding(-10)  
        .frame(width: <matchedWidth>, height: <matchedHeight>)  
        .offset(x: <offsetX>, y: <offsetY>)  
}
```

也就是说，源视图的大小会被作为建议尺寸提供给目标视图。正如我们之前讨论过的，如何处理这个建议尺寸完全取决于 `frame` 的子视图的行为。例如，如果目标视图是固定尺寸的视图 (例如，不可调整大小的图像或另一个固定的 `frame`)，那么这个几何匹配效果将不会影响目标视图的大小。

如果我们想要将一个视图的尺寸与另一个视图进行匹配，最重要的是两个视图应当具有相同的灵活度；否则，匹配就可能无法正常工作。当然，我们还可以有意限制目标视图的灵活度 (例如，通过使用带有最大宽度的可变 `frame`) 来规定目标视图的行为。

几何匹配效果和过渡

几何匹配效果在视图插入之前和插入后都会被应用，这和过渡的活动状态的情况类似：后者也会在插入或移除的视图的时候被应用。这使我们能够用非常少的代码创建从一个视图到另一个视图的平滑过渡。

当我们将一个新视图作为几何组的一部分插入到视图层次结构中时，它的初始几何属性按照 `isSource` 参数设置为 `false` 来进行计算，也就是说，视图的几何属性将与状态变更之前的视图树里的源视图进行匹配。当我们从视图树中移除一个视图时，它的最终几何属性将与状态变更之后的视图树中的源视图匹配。

这个特性让我们能够非常轻松地在视图之间写出过渡效果。下面是一个从30×30的圆形到全屏圆形的过渡示例：

```
struct ContentView: View {
    @State var hero = false
    @Namespace var namespace

    var body: some View {
        let circle = Circle().fill(Color.green)
        ZStack {
            if hero {
                circle
                    .matchedGeometryEffect(id: "image", in: namespace)
            } else {
                circle
                    .matchedGeometryEffect(id: "image", in: namespace)
                    .frame(width: 30, height: 30)
            }
        }
        .onTapGesture {
            withAnimation(.default) { hero.toggle() }
        }
    }
}
```

在上面的例子中，当我们切换 `hero` 时，可以看到圆形在 30×30 大小和全屏大小

之间平滑地进行动画。

这个动画有两个部分：几何匹配效果负责位置和尺寸，而默认的不透明度过渡则在两个视图之间进行混合。换句话说，在过渡期间，**两个视图**都在屏幕上。当我们将 `hero` 从 `false` 更改为 `true` 时，第一个视图会以不透明度 0 以及与第二个视图相匹配的 `frame` 被插入 (这个时候是在状态更改之前，第二个视图担任了几何组中的源视图)。在该过渡期间，第一个视图一边进行淡入，一边从初始的匹配位置向其最终位置进行动画。与此同时，第二个 (已被删除的) 视图则逐渐消失，并从状态更改前的原始位置过渡到状态更改后的新位置。在这个阶段，它会去匹配新插入的源视图。

请注意，修饰器的顺序至关重要，且容易出错。比如，如果我们在 `else` 分支中交换 `matchedGeometryEffect` 和 `frame` 的位置，那么较小的圆圈将始终保持其 30x30 的大小：

```
if hero {  
  ...  
} else {  
  circle  
    .frame(width: 30, height: 30)  
    .matchedGeometryEffect(id: "image", in: namespace)  
}
```

我们上面提到过，`.matchedGeometryEffect` 修饰器本质上是 `.frame` 和 `.offset` 修饰器的组合。由于具有 30 点宽度和高度的最内层 `frame` 会直接确定圆形的大小，所以几何匹配效果对于视图，至少在尺寸方面，就没有影响了。

除了从视图树中完全移除和插入视图之外，几何匹配效果还可以创造性地用于对保留在视图层次结构中的视图进行动画处理。我们可以通过更改命名空间或标识符，从几何组中插入或移除视图。