

objc.io | objc 中国

App 架构

使用 Swift 进行 iOS 架构

Chris Eidhof, Matt Gallagher, Florian Kugler 著
王巍, 茆子君, 李杰 译

英文版本 1.0 (2018 年 5 月), 中文版本 1.0.2 (2018 年 8 月)

© 2018 Kugler und Eidhof GbR
版权所有

ObjC 中国
在中国地区独家翻译和销售授权

获取更多书籍或文章, 请访问 <https://objccn.io>
电子邮件: mail@objccn.io

关于本书 5

1 介绍 13

应用架构 13

Model 和 View 13

App 的本质是反馈回路 15

架构技术 16

App 任务 17

2 App 设计模式概览 19

Model-View-Controller 19

Model-View-ViewModel+协调器 22

Model-View-Controller+ViewState 26

Model 适配器-View 绑定器 (MAVB) 28

Elm 架构 (TEA) 31

网络 34

没有提到的模式 34

3 Model-View-Controller 38

探索实现 39

测试 51

讨论 55

改进 56

总结 68

4 Model-View-ViewModel+协调器 (MVVM-C) 71

探索实现 73

测试 88

讨论 91

较少响应式编程的 MVVM 92

经验和教训 97

5	网络	101
	网络挑战 101	
	Controller 持有网络 102	
	Model 拥有网络 107	
	讨论 112	
6	Model-View-Controller+ViewState	115
	将 View State 作为 Model 的一部分 115	
	探索实现 118	
	测试 131	
	讨论 136	
	经验和教训 138	
7	Model 适配器-View 绑定器	146
	探索实现 147	
	测试 163	
	讨论 168	
	经验和教训 172	
8	Elm 架构	178
	探索实现 179	
	Elm 架构框架 191	
	测试 197	
	讨论 202	
	经验和教训 204	

关于本书

本书所专注的话题是 app 中所使用的架构，也就是那些将较小部分组合在一起形成一个完整 app 时所使用的结构和工具。通常来说，一个 app 会包含非常多种类的部件，像是用户输入、网络服务、文件服务、音频和图像、以及窗口服务等。在 app 开发中，如何对它们进行架构是一个很重要的话题。想要将这些部件组合起来，同时保证它们的状态以及状态的变更稳定可靠，而且能正确地进行传递，并非一件易事。这需要一套有力的规则，来定义组件之间相互协作的方式。

App 设计模式

我们将一组被重复使用的设计规则称为设计模式。本书将会展示如何使用五种最主要的 app 设计模式来完整实现一个 app。我们所挑选的模式中，有的已经经过广泛的验证，有的还处于实验阶段，它们分别是：

- Model-View-Controller (MVC)
- Model-View-ViewModel+Coordinator (MVVM-C)
- Model-View-Controller+ViewState (MVC+VS)
- ModelAdapter-ViewBinder (MAVB)
- Elm 架构 (The Elm Architecture, TEA)

译者注：相较于“模型-视图-控制器”，对于三者连接在一起所组成的架构名称，我们更倾向于保留被广泛接受的原文，也即 Model-View-Controller 的说法。但是，像是协调器 (coordinator)，适配器 (adapter)，绑定器 (binder) 等日常不太常用的词语，在上下文清晰的环境下，我们会选择使用中文译名。

抽象的框图往往被用来在最高层级描述设计模式的使用方式，但是它对这些模式在 iOS app 中应该如何具体使用并没有太多帮助。我们之后将会详细研究每种架构的典型框图，并探讨它们在实际中的使用方式。

在本书中，我们还会看到一系列不同的模式，在书写程序时，并没有哪个单独的模式能在所有情景下都做到最好。根据你、你的程序或者你的团队所想要达成的目标和期望，以及整个过程中所面临的约束，任何一种模式都有可能成为最佳选择。App 的设计模式不仅仅只是一套技术

上的工具，它同时也是审美和社交的手段，可以让你，以及你的代码的其他读者，与你所设计的程序产生交流。换言之，最好的模式就是对你来说最为清晰的模式。

架构技术和经验教训

展示每一种实现方式之后，我们会花一些时间讨论这种模式在解决问题时的优势，以及使用类似策略在其他任意模式中解决问题的方式。比如说，像是 `reducer`，响应式编程，接口解耦，状态枚举和多模型抽象这些技术，往往是和特定的模式绑定的。但是在本书中，在我们研究这些技术在它们各自的模式中的使用方式之后，我们还会继续讨论它们的核心思想，看看这些思想是如何跨越不同的模式来解决问题的。

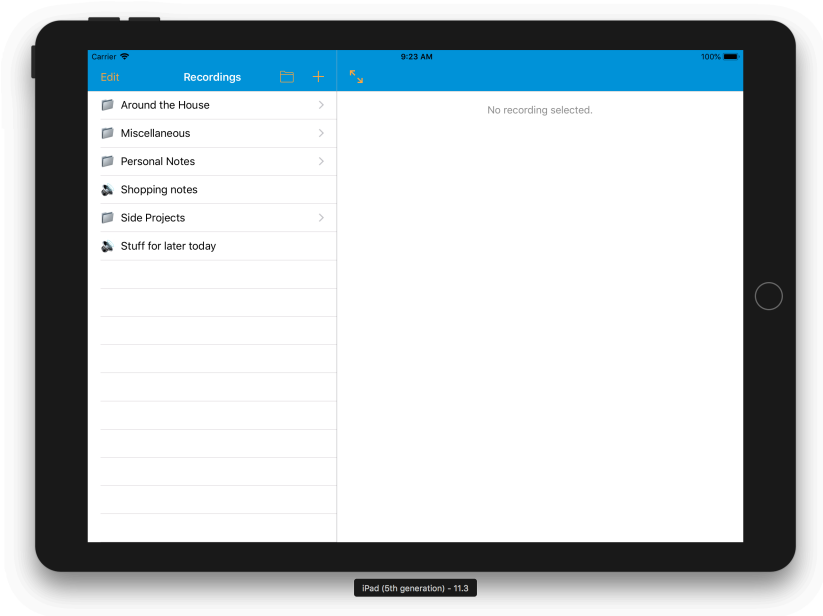
本书将会为你展示，对于每个问题，app 的架构包含了多种解决方案。当被正确实现后，所有的解决方案最终都将为用户呈现出同样的结果。这也就是说，app 架构的选择所关涉到的是我们程序员自己的幸福。我们想要在背后解决的潜在问题是什么？我们需要个别单独考虑的问题又有哪些？我们在何处需要自由度？又在何处需要稳定性？我们在哪里需要抽象？我们又在哪里需要简化？通过一次次地思考这些问题，我们对使用的架构进行选择。

关于录音 App

在本书中，我们会展示同一个录音 app 的五种不同实现。所有实现的完整源码都公开在 [GitHub](#) 上，您可以随时参阅。这个录音 app 正如其名，它可以记录和播放语音笔记。我们可以在 app 里以文件夹的方式组织记录的语音。这个 app 主要由用于组织文件的可以导航的 `FolderViewController`，用于播放文件的 `PlayViewController`，用于记录新文件的以 modal 方式显示的 `RecordViewController`，以及一个用来命名文件和文件夹的文本对话框组成。所有的内容都以标准的 `UIKit master-detail` 界面方式进行展示，其中 `FolderViewController` 位于主栏，`PlayViewController` 位于副栏进行显示。

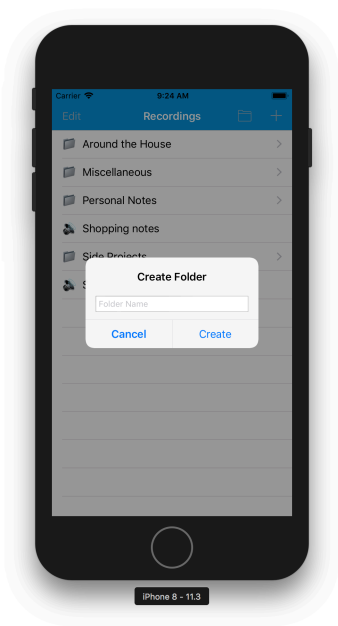
在选择本书的示例 app 时，我们列了一个判断标准，录音 app 满足了表里的所有项目。对于展示架构模式来说，它足够复杂；而放到一书中时，它又足够简单。app 中存在导航关系，其中的一些视图是实时更新的，而不像很多 app 中那样只存在静态视图。对模型层，我们实现了持续化的存储，每个变更都会被自动保存到磁盘。我们还为其中的两个版本加上了网络的支持。这个 app 对 iPhone 和 iPad 都做了适配，最后，我们还支持冷启动的状态恢复。

如果我们选择一个规模更小的 app，书里的内容会更容易被理解，但是这样一来，我们就很难有机会来展示各个架构之间的不同了。如果我们选择的 app 规模再大一些，每种架构在可扩展性方面的差距将显而易见，但是其中的细节也将被淹没。我们的录音 app 恰到好处地在两个极端之间做到了平衡。

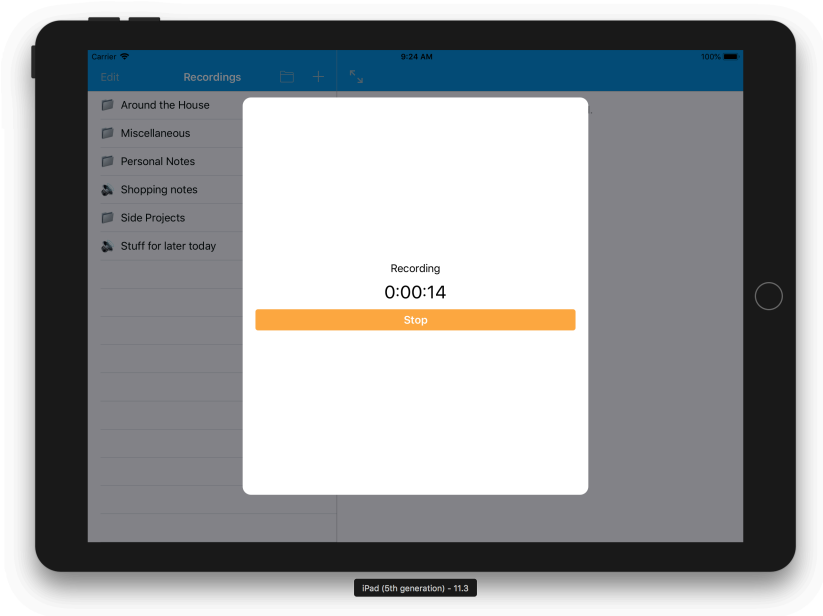


App 开始时显示一个根目录的 `FolderViewController`。这个 controller 存在于一个 `navigation controller` (导航控制器) 中，而这个 `navigation controller` 又是一个 `split view controller` 的 `master controller`。在 iPhone 上，这个 controller 将以全屏模式被展示，而在 iPad 上，它将会被显示在屏幕左侧 (这是 iOS 上 `UISplitViewController` 的标准行为)。

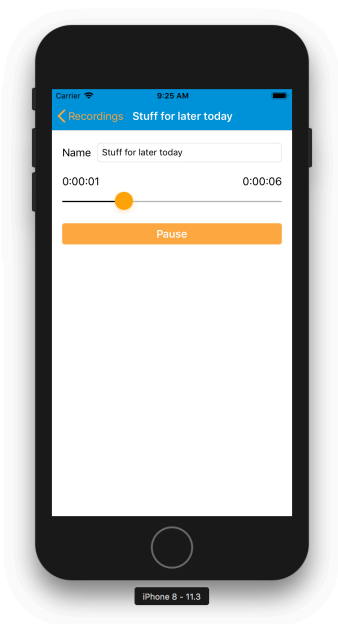
一个文件夹可以包含具体的录音文件以及其他的文件夹。`FolderViewController` 允许我们添加新的文件夹和录音，或者删除已有的项目。



在添加文件夹时，我们通过弹窗来询问文件夹的名字，而添加录音的操作将会直接展示一个 `RecordViewController`。



当在文件夹中选择某个项目时，我们需要根据项目的类型来确定导航操作：如果选择的是文件夹，我们会向导航栈中推入一个嵌套的 `FolderViewController`，如果我们选择的是录音，那么一个 `PlayViewController` 将会被显示在屏幕上。



PlayViewController 里包含一个文本框，用来展示和修改录音的名称。另外，这个 view controller 中还有一个用来指示当前播放进度的滑动条：我们播放录音时，滑动条会跟随移动，当我们去移动它时，录音会从选择的位置开始继续播放。在滑动条上方，我们使用标签来表示当前的播放时间和总时间。最后，在 play view controller 中还有用于控制播放，暂停以及继续播放的按钮。

视频

我们为这本书录制了配套的视频，你可以单独购买它们。这些视频专注探讨了那些更适合以现场编码和讨论的形式进行研究的问题，包括：

- 对五种版本的录音 app 添加新的特性 (一个迷你播放器)。
- 用八种不同的设计模式从头构建一个非常小的 app，这展示了它们之间的共通和不同之处。
- 实现一个小规模的 TEA 框架。

我们希望通过这些视频，你能在实践中对这本书中所涵盖的设计模式的 app 有更深入的理解。

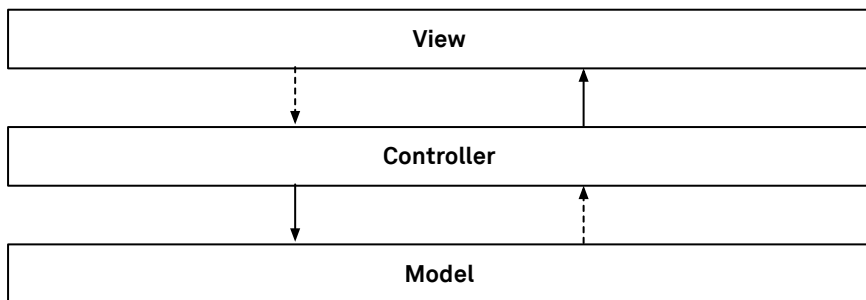
介绍

1

应用架构

App 架构是软件设计的一个分支，它关心如何设计一个 app 的结构。具体来说，它关注于两个方面：如何将 app 分解为不同的接口和概念层次部件，以及这些部件之间和自身的不同操作中所使用的控制流和数据流路径。

我们通常使用简单的框图来解释 app 的架构。比如，Apple 的 MVC 模式可以通过 model、view 和 controller 三层结构来描述。



上面框图中的模块展示了这个模式中不同名字的三个层次。在一个 MVC 项目中，绝大部分的代码都会落到其中某个层上。箭头表示了这些层进行连接的方式。

但是，这种简单的框图几乎无法解释在实践中模式的操作方式。这是因为在实际的 app 架构中，部件的构建有非常多的可能性。事件流在层中穿梭的方式是什么？部件之间是否应该在编译期间或者运行时持有对方？要怎么读取和修改不同部件中的数据？以及状态的变更应该以哪条路径在 app 中穿行？

Model 和 View

在最高的层级上，app 架构其实就是一套分类，app 中不同的部件会被归纳到某个类型中去。在本书中，我们将这些不同的种类叫做**层次**：一个层次指的是，遵循一些基本规则并负责特定功能的接口和其他代码的集合。

Model 层和 View 层是这些分类中最为常见的两个。

Model 层是 app 的内容，它不依赖于（像 UIKit 那样的）任何 app 框架。也就是说，程序员对 model 层有完全的控制。Model 层通常包括 model 对象（在录音 app 中的例子是文件夹和录音

对象)和协调对象(比如我们的 app 例子中的负责在磁盘上存储数据的 Store 类型)。被存储在磁盘上的那部分 model 我们称之为**文档 model** (documentation model)。

View 层是依赖于 app 框架的部分，它使 model 层可见，并允许用户进行交互，从而将 model 层转变为一个 app。当创建 iOS 应用时，view 层几乎总是直接使用 UIKit。不过，我们也会看到在有些架构中，会使用 UIKit 的封装来实现不同的 view 层。另外，对一些其他的像是游戏这样的自定义应用，view 层可以不是 UIKit 或者 AppKit，它可能是 SceneKit 或者 OpenGL 的某种封装。

有时候，我们选择使用结构体或者枚举来表示 model 或者 view 的实例，而不使用类的对象。在实践中，类型之间的区别非常重要，但是当我们在 model 层中谈到对象、结构体和枚举时，我们会将三者统一地称为 **model 对象**。类似地，我们也会把 view 层的实例叫做 **view 对象**，实际上它们也可能是对象、结构体或者枚举。

View 对象通常会构成一个单一的 **view 层级**，在这个层级中，所有的 view 对象通过树结构的方式连接起来。在树的根部是整个屏幕，屏幕中存在若干窗口，接下来在树的分支和叶子上是更多的小 view。类似地，view controller 也通常会形成 **view controller 层级**。不过，model 对象却不需要有明确的层级关系，在程序中它们可以是互不关联的独立 model。

当我们提到 **view** 时，通常指的是像一个按钮或者一个文本 label 这样的单一 view 对象。当我们提到 **model** 时，我们通常指的也是像一个 Recording 实例或者 Folder 实例这样的单个 model 对象。在该话题的大多数文献中，“model”在不同上下文中指的可能是不同的事情。它可以指代一个 model 层，model 层中的具体的若干对象，文档 model，或者是 model 层中不关联的文档。虽然可能会显得啰嗦，我们还是会尝试在本书中尽量明确地区分这些不同含义。

为什么 Model 和 View 的分类会被认为是基础中的基础

当然啦，就算不区分 model 层和 view 层，写出一个 app 也是绝对可能的。比如说，在一个简单的对话框中，通常就没有独立的 model 数据。在用户点击 OK 按钮的时候，我们可以直接从用户界面元素中读取状态。不过通常情况下，model 层的缺失，会让程序的行为缺乏对于清晰规则的依据，这会使得代码难以维护。

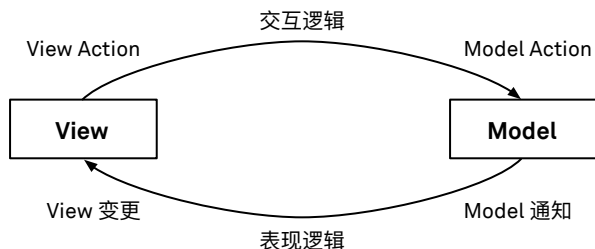
定义一个 model 层的最重要的理由是，它为我们的程序提供一个表述事实的单一来源，这会让逻辑清晰，行为正确。这样一来，我们的程序便不会被应用框架中的实现细节所支配。

应用框架为我们提供了构建 app 所需要的基础设施。在本书中，我们使用 Cocoa - 或者更精确说，根据目标平台，使用 UIKit, AppKit 或者 WatchKit - 来作为应用框架。

如果 model 层能做到和应用框架分离，我们就可以完全在 app 的范围之外使用它。我们可以很容易地在另外的测试套件中运行它，或者用一个完全不同的应用框架重写新的 view 层。这个 model 层将能够用于 Android，macOS 或者 Windows 版本的 app 中。

App 的本质是反馈回路

View 层和 model 层需要交流。所以，两者之间需要存在连接。假设 view 层和 model 层是被清晰地分开，而且不存在无法解耦的联结的话，两者之间的通讯就需要一些形式的翻译：



从根本上说，用户界面是一个同时负责展示和输入功能的反馈设备，所以毫无疑问，这导致的结果就是一个反馈回路。每个 app 设计模式所面临的挑战是如何处理这张图表中箭头所包含的交流，依赖和变换。

在 model 层和 view 层之间不同的路径拥有不同的名字。用户发起的事件会导致 view 的响应，我们把由此引起的代码路径称为 **view action**，像是点击按钮或者选中 table view 中的某一行就属于 view action。当一个 view action 被送到 model 层时，它会被转变为 **model action** (或者说，让 model 对象执行一个 action 或者进行更新的命令)。这种命令也被叫做一个**消息** (特别在当 model 是被 **reducer** 改变时，我们会这么称呼它)。将 view action 转变为 model action 的操作，以及路径上的其他逻辑被叫做**交互逻辑**。

译者注：reducer 这个名字来自于函数式编程的 reduce 操作，它指的是对一系列的值进行操作，把它们按照一定逻辑“减少”(合并)到一个值中。一般来说，架构中的 reducer 所负责的事情是，将一系列未来到来的变化(或者说消息)，与当前的状态进行“合并”，并推导出新的状态。我们会在后面的章节，特别是[TEA 的相关内容](#)中，看到一个实际的经典 reducer 的例子。

一个或者多个 **model** 对象上状态的改变叫做 **model 变更**。Model 的变更通常会触发一个 **model 通知**，比如说从 **model** 层发出一个可观测的通知，它描述 **model** 层中什么内容发生了改变。当 **view** 依赖于 **model** 数据时，通知会触发一个 **view 变更**，来更改 **view** 层中的内容。这些通知可以以多种形式存在：Foundation 中的 Notification，代理，回调，或者是其他机制，都是可以的。将 **model** 通知和数据转变为 **view** 更改的操作，以及路径上的其他逻辑被叫做 **表现逻辑**。

根据 app 模式的不同，有些状态可能是在文档 **model** 之外进行维护的，这样一来，更新这些状态的行为就不会追随文档 **model** 的路径。在很多模式中的**导航状态**就这种行为的一个常见例子，在 **view** 层级中的某个部分 (或者按照 Cocoa Storyboard 中使用的术语，将它称为 **scene**) 可能会被换出或者换入层级中。

在 app 中非文档 **model** 的状态被叫做 **view state**。在 Cocoa 里，大部分 **view** 对象都管理着它们自己的 **view state**，**controller** 对象则管理剩余的 **view state**。在 Cocoa **view state** 的框图中，通常会有加在反馈回路上的捷径，或者单个层自身进行循环。在有一些架构中，**view state** 不属于 **controller** 层，而是属于 **model** 层的部分 (不过，根据定义，**view controller** 并不是**文档 model** 的一部分)。

当所有的状态都在 **model** 层中被维护，而且所有的变更都通过完整的反馈回路路径进行传递时，我们就将它称为**单向数据流**。当任意的 **view** 对象或者中间层对象只能够通过 **model** 发出的通知来进行创建和更新 (换句话说，**view** 或者中间层不能通过捷径来更新自身或者其他的 **view**) 时，这个模式通常就是单向的。

架构技术

Apple 平台的标准 Cocoa 框架提供了一些架构工具。**Notification** 将值从单一源广播给若干个收听者。**键值观察 (KVO)** 可以将某个对象上属性的改变报告给另一个对象。然而，Cocoa 中的架构工具十分有限，我们将会使用到一些额外的框架。

本书中使用到的第三方技术中包含了**响应式编程**。响应式编程也是一种用来交流变更的工具，不过和通知或者 KVO 不同的是，它专注于在源和目标之间进行变形，让逻辑可以在部件之间传输信息的同时得以表达。

我们可以使用像是响应式编程或者 KVO 这样的技术创建属性绑定。绑定接受一个源和一个目标，无论何时，只要源发生了变化，目标也将被更新。这和手动进行观察在语法上有着不同，我们不再需要写观察的逻辑，而只需要指定源和目标，接下来框架将会为我们处理其余部分的工作。

macOS 上的 Cocoa 包含有 Cocoa 绑定技术，它是一种双向绑定，所有的可观察对象同时也是观察者，在一个方向上建立绑定连接，会在反方向也创建一个连接。不论是 (在 [MVVM-C 的章节](#) 中用到的) RxCocoa，还是 (在 [MAVB 章节](#) 中用到的) CwlViews，都不是双向绑定的。所以，在本书中，所有关于绑定的讨论都只涉及到单向绑定。

App 任务

要让程序正常工作，view 必须依赖于 model 数据来生成和存在，我们配置 view，让它可以对 model 进行更改，并且能在 model 更新时也得到更新。

所以，我们需要决定在 app 中如何执行下列任务：

1. **构建** — 谁负责构建 model 和 view，以及将两者连接起来？
2. **更新 model** — 如何处理 view action？
3. **改变 view** — 如何将 model 的数据应用到 view 上去？
4. **view state** — 如何处理导航和其他一些 model state 以外的状态？
5. **测试** — 为了达到一定程度的测试覆盖，要采取怎样的测试策略？

对于上面五个问题的回答，是构成 app 设计模式的基础要件。在本书中，我们会逐一研究这些设计模式。

App 设计模式概 览

2

本书专注于使用下面的设计模式为录音 app 完成五种不同的实现。

前两种模式在 iOS 中有着广泛的应用：

- **标准的 Cocoa Model-View-Controller (MVC)** 是 Apple 在示例项目中所采用的设计模式。它是 Cocoa app 中最为常见的架构，同时也是在 Cocoa 中讨论架构时所采用的基准线。
- **Model-View-ViewModel+协调器 (MVVM-C)** 是 MVC 的变种，它拥有单独的“view-model” (视图模型) 和一个用来管理 view controller 的协调器。MVVM 使用数据绑定 (通常会和响应式编程一起使用) 来建立 view-model 层和 view 层之间的连接。

我们将讨论的另外三种模式在 Cocoa 中并不常见，是还处于实验阶段的架构。这些架构为构建 app 提供了有用的见解，即使我们不选择将整个架构应用到我们的项目中，它们也对改进代码会有所帮助：

- **Model-View-Controller+ViewState (MVC+VS)** 这种模式将所有的 view state 集中到一个地方，而不是让它们散落在 view 和 view controller 中。这和 model 层所遵循的规则相同，
- **Model 适配器-View 绑定器 (Model Adapter-ViewBinder, MAVB)** 是本书的一位作者所使用的实验性质的架构。MAVB 专注于构建声明式的 view，并且抛弃 controller，采用绑定的方式在 model 和 view 之间进行通讯。
- **Elm 架构 (TEA)** 与 MVC 或者 MVVM 这样的常见架构完全背道而驰。它使用虚拟 view 层级来构建 view，并使用 reducer 在 model 和 view 之间进行交互。

在本章中，我们会对每种模式中蕴含的哲学和选择进行概览性的介绍，在后面的章节里，我们会看到这些选择是如何对录音 app 的实现产生影响的。

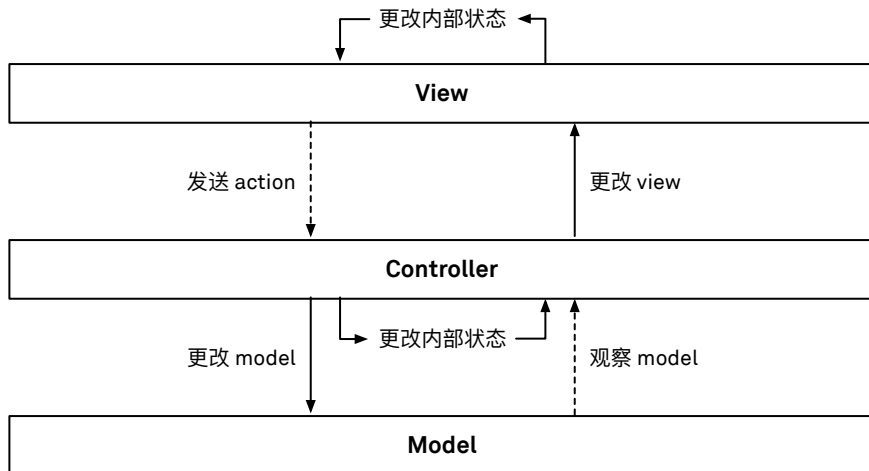
当然，除了这五种模式以外，iOS 的 app 设计模式还有非常多的选择。在讨论每种模式的时候，我们会为你揭示我们选择这种模式进行介绍的原因。在本章的最后，我们也会对没有被我们在本书中介绍的一些其他模式进行简单讨论。

Model-View-Controller

在 Cocoa MVC 中，一小部分 controller 对象负责处理 model 或者 view 层范畴之外的所有任务。

这意味着，controller 层接收所有的 view action，处理所有的交互逻辑，发送所有的 model action，接收所有的 model 通知，对所有用来展示的数据进行准备，最后再将它们应用到 view 的变更上去。如果我们去看一下[介绍一章](#)中的 app 反馈回路的图，会发现在 model 和 view 之间的箭头上，几乎每个标签都是 controller。而且要知道，在这幅图中，构建和导航任务并没有标注出来，它们也会由 controller 来处理。

下面是 MVC 模式的框图，它展示了一个 MVC app 的主要通讯路径：



图中的虚线部分代表运行时的引用，view 层和 model 层都不会直接在代码中引用 controller。实线部分代表编译期间的引用，controller 实例知道自己所连接的 view 和 model 对象的接口。

如果我们将这个图标外部描上边界的话，就得到了一个 MVC 版本的 app 反馈回路。注意在图表中其他的路径并不参与整个反馈回路的路径（也就是 view 层和 controller 层上那些指向自身的箭头）。

1. 构建

App 对象负责创建最顶层的 view controller，这个 view controller 将加载 view，并且知道应该从 model 中获取哪些数据，然后把它们显示出来。Controller 要么显式地创建和持有 model 层，要么通过一个延迟创建的 model 单例来获取 model。在多文档配置中，model 层由更低层的像是 UIDocument 或 NSDocument 所拥有。那些和 view 相关的单个 model 对象，通常会被 controller 所引用并缓存下来。

2. 更改 Model

在 MVC 中，controller 主要通过 target/action 机制和 (由 storyboard 或者代码进行设置的) delegate 来接收 view 事件。Controller 知道自己所连接的 view，但是 view 在编译期间却没有关于 controller 接口的信息。当一个 view 事件到达时，controller 有能力改变自身的内部状态，更改 model，或者直接改变 view 层级。

3. 更改 View

在我们所理解的 MVC 中，当一个更改 model 的 view action 发生时，controller 不应该直接去操作 view 层级。正确的做法是，controller 去订阅 model 通知，并且在当通知到达时再更改 view 层级。这样一来，数据流就可以单向进行：view action 被转变为 model 变更，然后 model 发送通知，这个通知最后被转为 view 变更。

4. View State

View state 可以按需要被 store 在 view 或者 controller 的属性中。相对于影响 model 的 view action，那些只影响 view 或 controller 状态的 action 则不需要通过 model 进行传递。对于 view state 的存储，可以结合使用 storyboard 和 UIStateRestoring 来进行实现，storyboard 负责记录活跃的 controller 层级，而 UIStateRestoring 负责从 controller 和 view 中读取数据。

5. 测试

在 MVC 中，view controller 与 app 的其他部件紧密相连。边界的缺失使得为 controller 编写单元测试和接口测试十分困难，集成测试是余下的为数不多的可行测试手段之一。在集成测试中，我们构建相连接的 view、model 和 controller 层，然后操作 model 或者 view，来测试是否能得到我们想要的结果。

集成测试的书写非常复杂，而且它涵盖的范围太广了。它不仅仅测试逻辑，也测试部件是如何连接的 (虽然在一些情况下和 UI 测试的角度并不相同)。不过，在 MVC 中通过集成测试，通常达到 80% 左右的测试覆盖率是有可能的。

MVC 的重要性

因为 Apple 在所有的实例项目中都使用了这种模式，加上 Cocoa 本身就是针对这种模式设计的，所以 Cocoa MVC 成为了 iOS，macOS，tvOS 和 watchOS 上官方认证的 app 架构模式。

本书在录音 app 中所提供的这个专门的实现里，所使用的是经典的 MVC 方式，它是我们所认为的贯穿 iOS 和 macOS 历史中，最能精确反应共通特点的方式。然后，我们会看到，MVC 的自由度允许我们在使用时引入非常多的变种：很多来源于其他模式的思想，它们都可以被集成到整个 app 中的某个小部分中去。

历史

MVC 这个名字第一次被提出是在 1979 年，Trygve Reenskaug 用它来描述 Smalltalk-76 上已经存在的“template pattern”应用。在他和 Adele Goldberg 讨论了术语方面的问题后，MVC 的名字被确定下来 (之前的名字包括 Model-View-Editor 和 Model-View-Tool-Editor 等)。

在原本的构想中，view 是直接“附着”在 model 层上，并观察所有 model 变化的。Controller 存在的目的仅仅是捕捉用户事件，并把它们转发给 model。这两个特性都是 Smalltalk 运行方式的产物，它们并不是为了现代的 app 框架所设计的，所以今天这种 MVC 的原始构想已经几乎绝迹了。

Cocoa 中的 MVC 实现可以追溯到大约 1997 年的 NeXTSTEP 4 的年代。在那之前，所有现在 controller 所担当的角色，通常都由一个 (像是 NSWindow 那样的) 高层 view 类来扮演。之后，从原始的 Smalltalk 的 MVC 实现中所发展出的理念是**分离展示部分**，也就是 view 层和 model 层应该被完全隔离开，这带来了一个强烈的需求，那就是要引入一个支持对象来辅助两者之间的通讯。NeXTSTEP 中 controller 的概念和 Taligent 稍早的 Model-View-Presenter 中的 presenter (展示器) 很相似。不过，在现在 Model-View-Presenter 这个名字通常被用来指代那些通过协议从 controller 中将 view 抽象出来的类似 MVC 的模式。

Model-View-ViewModel+协调器

MVVM 和 MVC 类似，也是通过基于场景 (scene，view 层级中可能会在导航发生改变时切入或者换出的子树) 进行的架构。

相较于 MVC，MVVM 在每个场景中使用 **view-model** 来描述场景中的表现逻辑和交互逻辑。

View-model 在编译期间不包含对 view 或者 controller 的引用。它暴露出一系列属性，用来描述每个 view 在显示时应有的值。把一系列变换运用到底层的 model 对象后，就能得到这些最终可以直接设置到 view 上的值。实际将这些值设置到 view 上的工作，则由预先建立的绑定来完成，绑定会保证当这些显示值发生变化时，把它设定到对应的 view 上去。响应式编程是用来表达这类声明和变换关系的很好的工具，所以它天生就适合 (虽说不是严格必要) 被用来处理

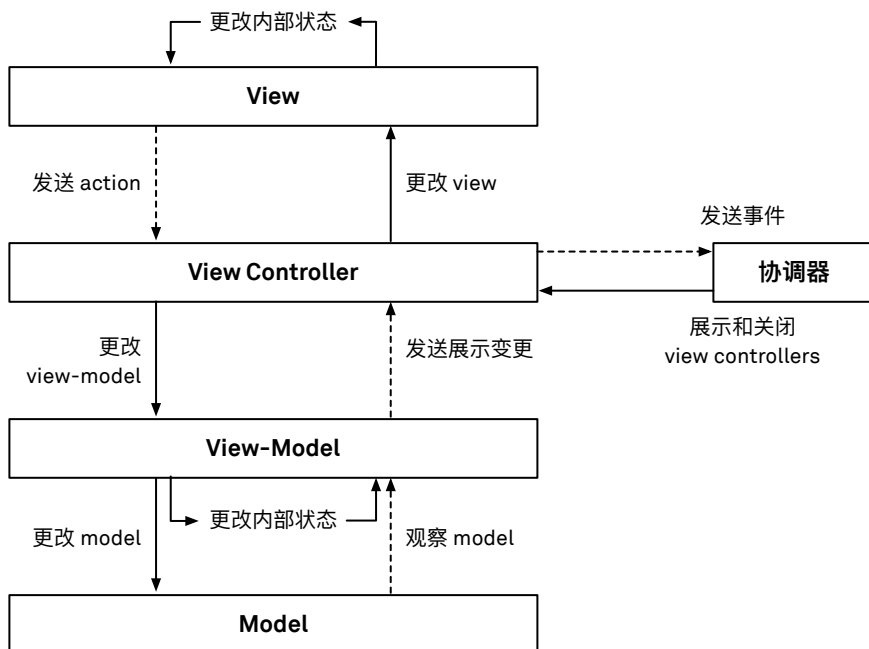
view-model。在很多时候，整个 view-model 都可以用响应式编程绑定的方式，以声明式的方式进行表达。

在理论上，因为 view-model 不包含对 view 层的引用，所以它是独立于 app 框架的，这让对于 view-model 的测试也可以独立于 app 框架。

由于 view-model 是和场景耦合的，我们还需要一个能够在场景间切换时提供逻辑的对象。在 MVVM-C 中，这个对象叫做**协调器** (coordinator)。协调器持有对 model 层的引用，并且了解 view controller 树的结构，这样，它能够为每个场景的 view-model 提供所需要的 model 对象。

和 MVC 不同，MVVM-C 中的 view controller 从来都不会直接引用其他的 view controller (所以，也不会引用其他的 view-model)。View controller 通过 delegate 的机制，将 view action 的信息告诉协调器。协调器据此显示新的 view controller 并设置它们的 model 数据。换句话说，view controller 的层级是由协调器进行管理的，而不是由 view controller 来决定的。

这些特性所形成的架构的总体结构如下图所示：



如果我们忽略掉协调器，那么这张图表就很像 MVC 了，只不过在 view controller 和 model 之间加入了一个阶段。MVVM 将之前在 view controller 中的大部分工作转移到了 view-model 中，但是要注意，view-model 并不会在编译时拥有对 view controller 的引用。

View-model 可以从 view controller 和 view 中独立出来，也可以被单独测试。同样，view controller 也不再拥有内部的 view state，这些状态也被移动到了 view-model 中。在 MVC 中 view controller 的双重角色 (既作为 view 层级的一部分，又负责协调 view 和 model 之间的交互)，减少到了单一角色 (view controller 仅仅只是 view 层级的一部分)。

协调器模式的加入进一步减少了 view controller 所负责的部分：现在它不需要关心如何展示其他的 view controller 了。因此，这实际上是以添加了一层 controller 接口为代价，降低了 view controller 之间的耦合。

译者注：View-model 虽然名字里既有 view 又有 model，但是它所扮演的其实是不折不扣的类似 controller 的角色。在本书中，当上下文不明确时，我们将严格区分 view controller 和 controller。要注意，前者只是后者的一个子集，view controller 仅仅是负责控制 view 的那部分 controller 类型。

1. 构建

对于 model 的创建和 MVC 中的保持不变，通常它是一个顶层 controller 的职责。不过，单独的 model 对象现在属于 view-model，而不属于 view controller。

初始的 view 层级的创建和 MVC 中的一样，通过 storyboard 或者代码来完成。和 MVC 不同的是，view controller 不再直接为每个 view 获取和准备数据，它会把这项工作交给 view-model。View controller 在创建的时候会一并创建 view-model，并且将每个 view 绑定到 view-model 所暴露出的相应属性上去。

2. 更改 Model

在 MVVM 中，view controller 接收 view 事件的方式和 MVC 中一样 (在 view 和 view controller 之间建立连接的方式也相同)。不过，当一个 view 事件到达时，view controller 不会去改变自身的内部状态、view state、或者是 model。相对地，它立即调用 view-model 上的方法，再由 view-model 改变内部状态或者 model。

3. 更改 View

和 MVC 不同，view controller 不监听 model。View-model 将负责观察 model，并将 model 的通知转变为 view controller 可以理解的形式。View controller 订阅 view-model 的变更，这通常通过一个响应式编程框架来完成，但也可以使用任意其他的观察机制。当一个 view-model 事件来到时，由 view controller 去更改 view 层级。

为了实现单向数据流，view-model 总是应该将变更 model 的 view action 发送给 model，并且仅仅在 model 变化实际发生之后再通知相关的观察者。

4. View State

View state 要么存在于 view 自身之中，要么存在于 view-model 里。和 MVC 不同，view controller 中不存在任何 view state。View-model 中的 view state 的变更，会被 controller 观察到，不过 controller 无法区分 model 的通知和 view state 变更的通知。当使用协调器时，view controller 层级将由协调器进行管理。

5. 测试

因为 view-model 和 view 层与 controller 层是解耦合的，所以可以使用接口测试来测试 view-model，而不需要像 MVC 里那样使用集成测试。接口测试要比集成测试简单得多，因为不需要为它们建立完整的组件层次结构。

为了让接口测试尽可能覆盖更多的范围，view controller 应当尽可能简单，但是那些没有被移出 view controller 的部分仍然需要单独进行测试。在我们的实现中，这部分内容包括与协调器的交互，以及初始时负责创建工作的代码。

MVVM 的重要性

MVVM 是 iOS 上最流行的 MVC 的非直接变形的 app 设计模式。换言之，它和 MVC 相比，并没有非常大的不同；两者都是围绕 view controller 场景构建的，而且所使用的机制也大都相同。

最大的区别可能在于 view-model 中对响应式编程的使用了，它被用来描述一系列的转换和依赖关系。通过使用响应式编程来清晰地描述 model 对象与显示值之间的关系，为我们从总体上理解应用中的依赖关系提供了重要的指导。

iOS 中的协调器是一种很有用的模式，因为管理 view controller 层级是一件非常重要的事情。协调器在本质上并没有和 MVVM 绑定，它也能被使用在 MVC 或者其他模式上。

历史

MVVM 由 Ken Cooper 和 Ted Peters 提出，他们当时在微软工作，负责后来变成 Windows Presentation Foundation (WPF) 的项目，这是微软.NET 的 app 框架，并于 2005 年正式发布。

WPF 使用一种基于 XML，被称为 XAML 的描述性语言来描述 view 所绑定的某个 view-model 上的属性。在 Cocoa 中，没有 XAML，我们必须使用像是 RxSwift 这样的框架和一些 (通常存在于 controller 中的) 代码来完成 view-model 和 view 的绑定。

译者注：微软在 MVVM 和响应式编程的领域深耕已久，这也是为什么 RxSwift 或者说 Rx 系列的响应式编程框架如此流行的原因 (整个 ReactiveX 最初就是来源于微软的 Reactive Extensions，然后再将同样的思想移植到不同语言的)。

MVVM 和我们在 MVC 历史中提到的 MVP 模式非常类似。不过，在 Cooper 和 Peters 的论述中，MVVM 中 view 和 view-model 的绑定需要明确的框架支持，但 presenter 是通过传统的手动方式来传递变化。

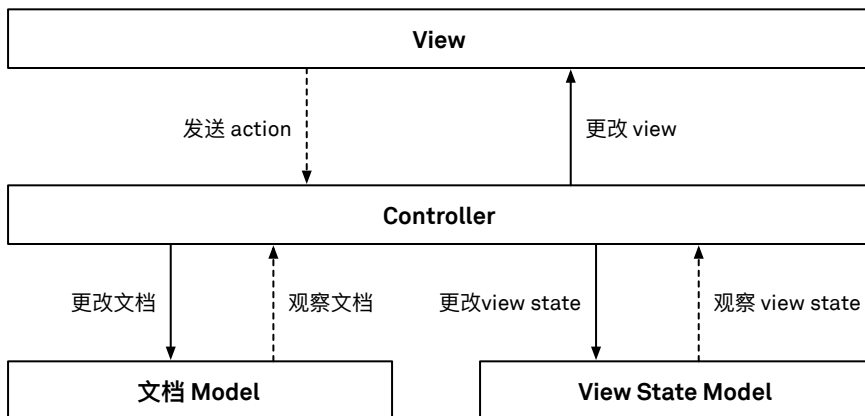
iOS 中的协调器则是最近才 (重新) 流行起来的，Soroush Khanlou 在 2015 年时在他的[网站上](#)描述了这个想法。协调器基于 app controller 这样的更古老的模式，而它们在 Cocoa 和其他平台上已经存在了有数十年之久。

Model-View-Controller+ViewState

MVC+VS 是为标准的 MVC 带来单向数据流方式的一种尝试。在标准的 Cocoa MVC 中，view state 可以由两到三种不同的路径进行操作，MVC+VS 则试图避免这点，让 view state 的处理更加易于管理。在 MVC+VS 中，我们明确地在一个新的 model 对象中，对所有的 view state 进行定义和表达，我们把这个对象叫做 view state model。

在 MVC+VS 中，我们不会忽略任何一次导航变更，列表选择，文本框编辑，开关变更，model 展示或者滚动位置变更 (或者其他任意的 view state 变化)。我们将这些变更发送给 view state model。每个 view controller 负责监听 view state model，这样变更的通讯会非常直接。在表

现或者交互逻辑部分，我们不从 view 中去读取 view state，而是从 view state model 中去获取它们：



结果所得到的图表和 MVC 类似，但 controller 的内部反馈回路的部分 (被用来更新 view state) 有所不同，现在它和 model 的回路类似，形成了一个独立的 view state 回路。

1. 构建

和传统的 MVC 一样，将文档 model 数据应用到 view 上的工作依然是 view controller 的责任，view controller 还会使用和订阅 view state。因为 view state model 和文档 model 都需要观察，所以相比于典型的 MVC 来说，我们需要多得多的通过通知进行观察的函数。

2. 更改 Model

当 view action 发生时，view controller 去变更文档 model (这和 MVC 保持不变) 或者变更 model state。我们不会去直接改变 view 层级，所有的 view 变更都要通过文档 model 和 view state model 的通知来进行。

3. 更改 View

Controller 同时对文档 model 和 view state model 进行观察，并且只在变更发生的时候更新 view 层级。

4. View State

View State 被明确地从 view controller 中提取出来。处理的方法和 model 是一样的：controller 观察 view state model，并且对应地更改 view 层级。

5. 测试

在 MVC+VS 中，我们使用和 MVC 里类似的集成测试，但是测试本身会非常不同。所有的测试都从一个空的根 view controller 开始，然后通过设定文档 model 和 view state model，这个根 view controller 可以构建出整个 view 层级和 view controller 层级。MVC 的集成测试中最困难的部分 (设定所有的部件) 在 MVC+VS 中可以被自动完成。要测试另一个 view state 时，我们可以重新设置全局 view state，所有的 view controller 都会调整自身。

一旦 view 层级被构建，我们可以编写两种测试。第一种测试负责检查 view 层级是不是按照我们的期望被建立起来，第二种测试检查 view action 有没有正确地改变 view state。

MVC+VS 的重要性

MVC+VS 主要是用来对 view state 进行教学的工具。

在一个非标准 MVC 的 app 中，添加一个 view state model，并且在每个 view controller 中 (在已经对 model 进行观察的基础上) 观察这些 view state model，提供了不少优点：任意的状态恢复 (这种恢复不依赖于 storyboard 或者 UIStateRestoration)，完整的用户界面日志，以及为了调试目的，在不同的 view state 间进行跳转的能力。

历史

这种特定的体系是 Matt Gallagher 在 2017 年开发的教学工具，它被用来展示单向数据流和用户界面的时间旅行等概念。这个模式的目标是，在传统的 Cocoa MVC app 上通过最小的改动，实现对 view 的状态在每个 action 发生时都可以进行快照。

Model 适配器-View 绑定器 (MAVB)

MAVB 是一种以绑定为中心的实验模式。在这个模式中，有三个重要的概念：view 绑定器，model 适配器，以及绑定。

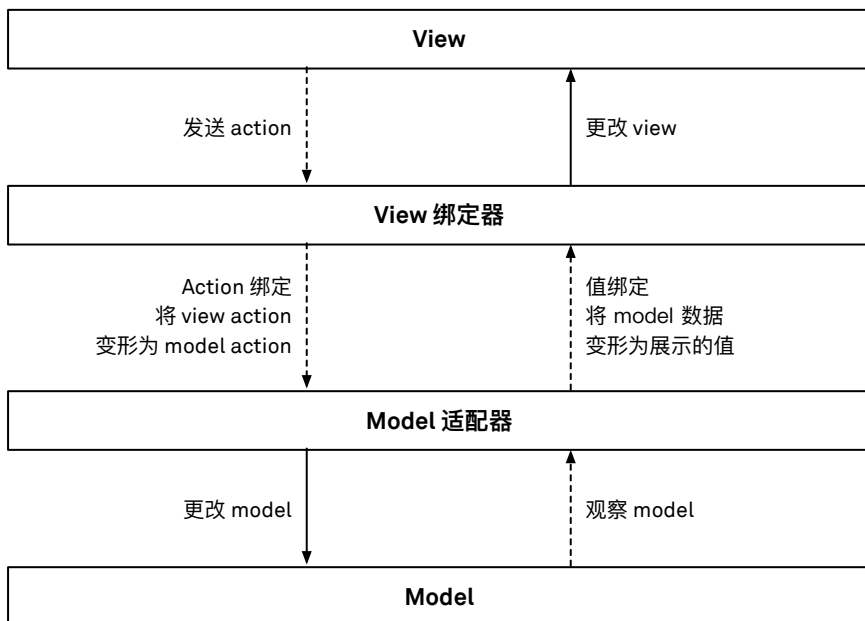
View 绑定器是 view (或者 view controller) 的封装类：它构建 view，并且为它暴露出一个绑定列表。一些绑定为 view 提供数据 (比如，一个标签的文本)，另一些从 view 中发出事件 (比如，按钮点击或者导航变更)。

虽然 view 绑定器可以含有动态绑定，但是 view 绑定器本身是不可变的。这让 MAVB 也成为了一种声明式的模式：你声明 view 绑定器和它们的 action，而不是随着时间去改变 view 绑定器。

Model 适配器是可变状态的封装，它是由所谓的 **reducer** 进行实现的。Model 适配器提供了一个 (用于发送事件的) 输入绑定，以及一个 (用于接收更新的) 输出绑定。

在 MAVB 中，你不会去直接创建 view；相对地，你只会去创建 view 绑定器。同样地，你也从来不会去处理 model 适配器以外的可变状态。在 view 绑定器和 model 适配器之间的 (两个方向上的) 变换，是通过 (使用标准的响应式编程技术) 来对绑定进行变形而完成的。

MAVB 移除了对 controller 层的需求。创建逻辑通过 view 绑定器来表达，变换逻辑通过绑定来表达，而状态变更则通过 model 适配器来表达。结果得到的框图如下：



1. 构建

Model 适配器 (用来封装主 model) 和 view state 适配器 (封装顶层的 view state) 通常是在 main.swift 文件中进行创建的, 这早于任何的 view。

View 绑定器使用普通的函数进行构建, 这些函数接受必要的 model 适配器作为参数。实际的 Cocoa view 则由框架负责进行创建。

2. 更改 Model

当一个 view (或者 view controller) 可以发出 action 时, 对应的 view 绑定允许我们指定一个 action 绑定。在这里, 数据从 view 流向 action 绑定的输出端。典型情况下, 输出端会和一个 model 适配器相连接, view 事件会通过绑定进行变形, 成为 model 适配器可以理解的一条消息。这条消息随后被 model 适配器的 reducer 使用, 并改变状态。

3. 更改 View

当 model 适配器的状态发生改变时, 它会通过输出信号产生通知。在 view 绑定器中, 我们可以将 model 适配器的输出信号进行变形, 并将它绑定到一个 view 属性上去。这样一来, view 属性就会在一个通知被发送时自动进行变更了。

4. View State

View state 被认为是 model 层的一部分。View state action 以及 view state 通知和 model action 以及 model 通知享有同样的路径。

5. 测试

在 MAVB 中, 我们通过测试 view 绑定器来测试代码。由于 view 绑定器是一组绑定的列表, 我们可以验证绑定包含了我们所期望的条目, 而且它们的配置正确无误。我们可以和使用绑定来测试初始构建以及发生变化时的情况。

在 MAVB 中进行的测试, 与在 MVVM 中的测试很相似。不过, 在 MVVM 中, view controller 有可能会包含逻辑, 这导致在 view-model 和 view 之间有可能会存在没有测试到的代码。而 MAVB 中不存在 view controller, 绑定代码是 model 适配器和 view 绑定器之间的唯一的代码, 这样一来, 保证完整的测试覆盖要简单得多。

MAVB 的重要性

在我们所讨论的主要模式之中，MAVB 没有遵循某个直接的先例，它既不是从其他平台移植过来的模式，也不是其他模式的变种。它自成一派，用于试验目的，而且一些奇怪。我们在这儿介绍它的意义在于，它展示了一些很不一样的东西。不过，这并不是说这个模式没有从其他模式中借鉴经验教训：像是绑定、响应式编程、领域专用语言以及 `reducer` 都是已经被熟知的想法了。

历史

MAVB 是 Matt Gallagher 在 Cocoa with Love 网站上首先提出的。这个模式参照了 Cocoa 绑定、函数式响应动画、ComponentKit、XAML、Redux 以及成千上万行的使用 Cocoa view controller 的经验。

本书中的实现使用了 CwlViews 框架来处理 view 构建、绑定器和适配器的实现等工作。

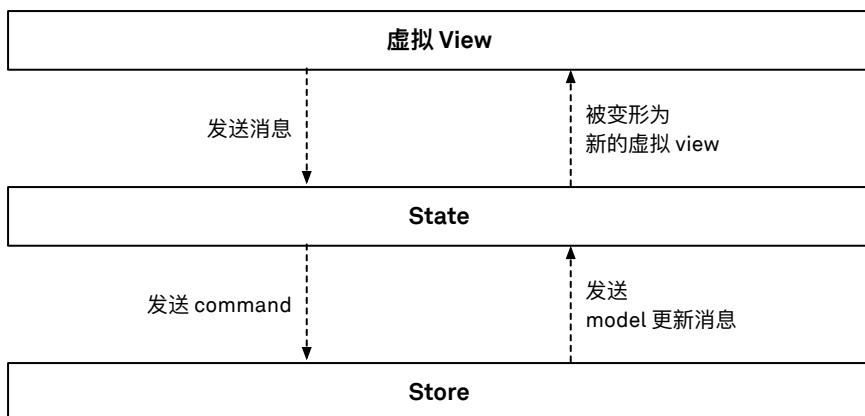
Elm 架构 (TEA)

TEA 和 MVC 有着根本上的不同。在 TEA 中，model 和所有的 view state 被集成为一个单个状态对象，所有 app 中的变化都通过向状态对象发送消息来发生，一个叫做 `reducer` 的状态更新函数负责处理这些消息。

在 TEA 中，每个状态的改变会生成一个新的**虚拟 view 层级**，它由轻量级的结构体组成，描述了 view 层级应该看上去的形式。虚拟 view 层级让我们能够使用**纯函数**的方式来写 view 部分的代码；虚拟 view 层级总是直接从状态进行计算，中间不会有任何副作用。当状态发生改变时，我们使用同样的函数重新计算 view 层级，而不是直接去改变 view 层级。

Driver 类型 (这是 TEA 框架中的一部分，它负责持有对 TEA 中其他层的引用) 将对虚拟 view 层级和 UIView 层级进行比较，并且对它进行必要的更改，让 view 和它们的虚拟版本相符合。这个 TEA 框架中的 driver (驱动) 部件是随着我们 app 的启动而被初始化的，它自身并不知道要对应哪个特定的 app。我们要在它的初始化方法中传入这些信息：包括 app 的初始状态，一个通过消息更新状态的函数，一个根据给定状态渲染虚拟 view 层级的函数，以及一个根据给定状态计算通知订阅的函数 (比如，我们可以订阅某个 model store 更改时所发出的通知)。

从框架的使用者的视角来看，TEA 的关于更改部分的框图是这样的：



如果我们追踪这张图表的上面两层，我们会发现在 view 和 model 之间存在我们在本章开头是就说过的反馈回路；这是一个从 view 到状态，然后再返回 view 的回路 (通过 TEA 框架进行协调)。

下面的回路代表的是 TEA 中处理副作用的方式 (比如将数据写入磁盘中)：当在状态更新方法中处理消息时，我们可以返回一个命令，这些命令会被 driver 所执行。在我们的例子中，最重要的命令是更改 store 中的内容，store 反过来又被 driver 所持有的订阅者监听。这些订阅者可以触发消息来改变状态，状态最终触发 view 的重新渲染作为响应。

这些事件回路的结构让 TEA 成为了遵守单向数据流原则的设计模式的另一个例子。

1. 构建

状态在启动时被构建，并传递给运行时系统 (也就是 driver)。运行时系统拥有状态，store 是一个单例。

初始的 view 层级和之后更新时的 view 层级是通过同样的路径构建的：通过当前的状态，计算出虚拟 view 层级，运行时系统负责更新真实的 view 层级，让它与虚拟 view 层级相匹配。

2. 更改 Model

虚拟 view 拥有与它们所关联的消息，这些消息在一个 view 事件发生时会被发送。Driver 可以接收这些消息，并使用更新方法来改变状态。更新方法可以返回一个命令 (副作用)，比如我们

想在 store 中进行的改动。Driver 会捕获该命令并执行它。TEA 让 view 不可能直接对状态或者 store 进行更改。

3. 更改 View

运行时系统负责这件事。改变 view 的唯一方式是改变状态。所以，初始化创建 view 层级和更新 view 层级之间没有区别。

4. View State

View state 是包含在整体的状态之中的。由于 view 是直接从状态中计算出来的，导航和交互状态也同样会被自动更新。

5. 测试

在大多数架构中，让测试部件彼此相连往往要花费大量努力。在 TEA 中，我们不需要对此进行测试，因为 driver 会自动处理这部分内容。类似地，我们不需要测试当状态变化时 view 会正确随之变化。我们所需要测试的仅仅是对于给定的状态，虚拟 view 层级可以被正确计算。

要测试状态的变更，我们可以创建一个给定的状态，然后使用 update 方法和对应的消息来改变状态。然后通过对比之前和之后的状态，我们就可以验证 update 是否对给定的状态和消息返回了所期望的结果。在 TEA 中，我们还可以测试对应给定状态的订阅是不是正确。和 view 层级一样，update 函数和订阅也都是纯函数。

因为所有的部件 (计算虚拟 view 层级，更新函数和订阅) 都是纯函数，我们可以对它们进行完全隔离的测试。任何框架部件的初始化都是不需要的，我们只用将参数传递进去，然后验证结果就行了。我们 TEA 实现中的大多数测试都非常直截了当。

Elm 架构的重要性

TEA 最早是在 Elm 这门函数式语言中被实现的。所以 TEA 是一种如何用函数式的方法表达 GUI 编程的尝试。TEA 同时也是最为古老的单向数据流架构。

历史

Elm 是 Evan Czaplicki 所设计的函数式编程语言，它最初的目的是为了构建前端 web app。TEA 是归功于 Elm 社区的一个模式，它的出现是语言约束和目标环境相互作用的自然结果。它背后的思想影响了很多其他的基于 web 的框架，其中包括 React、Redux 和 Flux 等。在 Swift 中，还没有 TEA 的权威实现，不过我们可以找到不少研究型的项目。在本书中，我们使用 Swift 按我们自己的理解实现了这个模式。主要的工作由 Chris Eidhof 于 2017 年完成。虽然我们的这个实现还并不是“产品级”的，但是许多想法是可以用在生产代码中的。

在本书中，我们使用**状态 (state)** 这个词，不过在 Elm 里，这个状态被叫做 **model** (模型)。我们选择在提到 TEA 的时候还是叫它**状态**，因为 model 在本书中已经被用来指代其他意涵了。

网络

在书中还有一章关于网络部分的内容。在那里，我们使用 MVC 版本的 app 来展示了两不同的添加网络层的方法。在第一种方法中，**controller 拥有网络**，我们将 model 层用一个网络服务取代。在第二种方法中，**model 拥有网络**，我们在 model 层的上方添加了网络功能。

从研究 controller 拥有的网络开始会简单一些：每个 view controller 在需要的时候执行请求，并将结果缓存在本地。不过，一旦当数据需要在不同的 view controller 之间共享的时候，这种模式就变得难以操作了。此时，一种通常更简单的做法是换成 model 拥有的网络。通过将网络作为 model 层的扩展，我们可以建立一套共享数据和变更通讯的机制。

没有提到的模式

我们在这本书里选择覆盖了五种不同的模式。其中有三种是 MVC 与其变形，另外两种还处于实验阶段，并不适合直接用于产品。为什么我们选择了这些模式，而没有选择另外一些被用在已经发布的 app 中的模式呢？下面，我们将简单介绍一些其他的竞争者，并告诉你们为什么我们没有将它们包含进来。

Model-View-Presenter

Model-View-Presenter (MVP) 是一种在 Android 上很流行的模式，在 iOS 中，也有相应的实现。在总体结构和使用的技术上，它粗略来说是一种位于标准 MVC 和 MVVM 之间的模式。

MVP 使用单独的 presenter 对象，它和 MVVM 中 view-model 所扮演的角色一样。相对 view-model 而言，presenter 去除了响应式编程的部分，而是把要展示的值暴露为接口上的属性。不过，每当这些值需要变更的时候，presenter 会立即将它们推送到下面的 view 中去 (view 将自己作为协议暴露给 presenter)。

从抽象的观点来看，MVP 和 MVC 很像。Cocoa 的 MVC，除了名字以外，**就是一个 MVP** - 它是从上世纪九十年代 Taligent 的原始的 MVP 实现中派生出来的。View，状态和关联的逻辑在两个模式中都是一样的。不同之处在于，现代的 MVP 中有一个分离的 presenter 实体，它使用协议来在 presenter 和 view controller 之间进行界定，Cocoa 的 MVC 让 controller 能够直接引用 view，而 MVP 中的 presenter 只能知道 view 的协议。

有些开发者认为协议的分离对于测试是必要的。当我们在讨论测试时，我们会看到标准的 MVC 在没有任何分离的情况下，也可以被完整测试。所以，我们感觉 MVP 并没有太大不同。如果我们对测试一个完全解耦的展示层有强烈需求的话，我们认为 MVVM 的方式更简单一些：让 view controller 通过观察去从 view-model 中**拉取值**，而不是让 presenter 将值推送到一个协议中去。

VIPER, Riblets, 和其他 “Clean” 架构

VIPER, Riblets 和其他类似的模式尝试将 Robert Martin 的 “Clean Architecture” 从 web app 带到 iOS 开发中，它们主要把 controller 的职责分散到三到四个不同的类中，并用严格的顺序将它们排列起来。在序列中的每个类都不允许直接引用序列中前面的类。

为了强制单方向的引用这一规则，这些模式需要**非常多的**协议，类，以及在不同层中传递数据的方式。由于这个原因，很多使用这些模式的开发者会去使用代码生成器。我们的感觉是，这些代码生成器，以及任何的繁杂到需要生成器的模式，都产生了一些误导。将 “Clean” 架构带到 Cocoa 的尝试通常都宣称它们可以管理 view controller 的 “肥大化” 问题，但是让人啼笑皆非的是，这么做往往让代码库变得更大。

虽然将接口分解是控制代码尺寸的一种有效手段，但是我们认为这应该按需进行，而不是教条式地对每个 view controller 都这么操作。分解接口需要我们对数据以及所涉及到的任务有清楚的认识，只有这样，我们才能达到最优的抽象，并在最大程度上降低代码的复杂度。

基于组件的架构 (React Native)

如果你选择使用 JavaScript 而不是 Swift 编程，或者你的 app 重度依赖于 web API 的交互，JavaScript 会是更好的选择，这时你可能会考虑 React Native。不过，本书是专注于 Swift 和 Cocoa 的，所以我们将探索模式的界限定在了这些领域内。

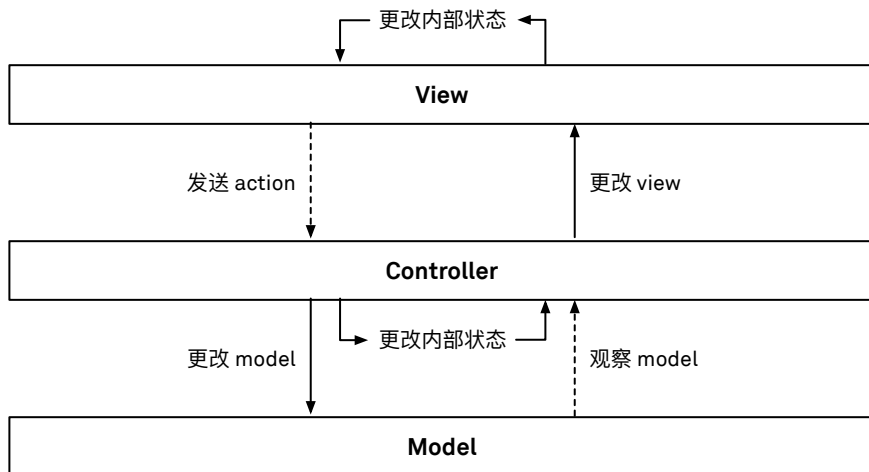
如果你想要找一些类似 React Native，但是是基于 Swift 的东西的话，可以看看我们对 TEA 的探索。MAVB 的实现也从 ComponentKit 中获得了一些启发，而 ComponentKit 本身又从 React 中获取灵感：它使用类 DSL 的语法来进行声明式和可变形的 view 构建，这和 React 中 Component 的 render 方法及其相似。

Model-View- Controller

3

Model-View-Controller (MVC) 模式是本书中讨论的所有其他模式的基准线。它也是 Cocoa 框架的认证模式，并且是我们所讨论的模式中年代最为久远的一个。

MVC 的核心思想是，controller 层负责将 model 层和 view 层撮合到一起工作。Controller 对另外两层进行构建和配置，并对 model 对象和 view 对象之间的双向通讯进行协调。所以，在一个 MVC app 中，controller 层是作为核心来参与形成 app 的反馈回路的：



MVC 基于经典的面向对象原则：对象在内部对它们的行为和状态进行管理，并通过类和协议的接口进行通讯；view 对象通常是自包含且可重用的；model 对象独立于表现形式之外，且避免依赖程序的其他部分。而将其他两部分组合起来成为完整的程序，则是 controller 层的责任。

Apple 将 MVC 描述为三种不同的子模式的集合：

1. 合成模式 - view 被组装成为层级，该层级按组区分，由 controller 对象进行管理。
2. 策略模式 - controller 对象负责协调 view 和 model，并且对可重用的、独立于 app 的 view 在 app 中的行为进行管理。
3. 观察者模式 - 依赖于 model 数据的对象必须订阅和接收更新。

对于 MVC 模式的解释通常相当松散，这些子模式 - 特别是观察者模式 - 很多时候没有被严格遵守。不过，我们将会在我们的实现中包含所有的这三种模式。

MVC 模式有一些众所周知的缺陷，首当其冲的就是 controller 层拥有太多的职责的问题，我们将它戏称为“**view controller 肥大化**”(massive view controller，它的缩写和 MVC 模式相同)。MVC 也面临着难以测试的问题，特别是单元测试和接口测试非常困难，甚至不可能实现。除开这些缺点，MVC 是 iOS app 中使用的最简单的模式，只要你理解它的缺点和如何克服它们，该模式就可以适应任意的程序，而且也能包含健全的测试。

在下一节中，我们会讨论录音 app 的 MVC 模式实现细节。我们在书中只展示了部分源码，完整的代码 (包括这个 app 的其他模式的实现) 都可以在 [GitHub](#) 上找到。

探索实现

创建

Cocoa MVC 对程序的创建过程大体上遵循 Cocoa 框架所提供的默认的启动流程。这一过程的主要目标是保证这三个对象的创建：UIApplication 对象，application delegate，以及主窗口的根 view controller。该过程的配置分布在三个文件中，它们的默认名称分别是 Info.plist，AppDelegate.swift 和 Main.storyboard。

这三个对象都属于 controller 层级，它们提供了一个可以对后续启动流程进行配置的地方，因此，controller 层将负责所有的创建工作。

将 View 连接到初始数据

MVC app 中的 view 不直接引用 model 对象；View 将保持独立可重用。相反地，model 对象被存储在 view controller 中，这让 view controller 变成了一个不可重用的类，不过这正是 view controller 的目的：它将 app 相关的特定知识传达给程序中的其他部件。

保存在 view controller 里的 model 对象会赋予 view controller **身份** (它让 view controller 知道自己在程序中的位置，以及如何与 model 层进行通话)。View controller 将 model 对象的相关属性值提取出来，并进行变形，然后将变形后的值设置到它持有的 view 中去。

我们要通过怎样的方式来为 view controller 设置这个身份对象呢？录音 app 中，在 view controller 上设置一个初始 model 值时，我们有两种不同的方式：

1. 通过判定 controller 在 controller 层级上的位置以及 controller 的类型，直接访问一个全局的 model 对象。

2. 开始时将 model 对象的引用设置为 nil 并让所有东西保持为空白状态，直到另一个 controller 提供了一个非 nil 值。

第三种选择是在 controller 初始化时将 model 对象当作参数传递进来 (也就是依赖注入)，当有可能时，我们应该选择这种做法。但是，storyboard 的构建流程通常不允许我们在 view controller 初始化时向它们传递参数。

FolderViewController 是第一种策略下的一个例子。每个文件夹 view controller 最初的 folder 属性的设定都如下所示：

```
var folder: Folder = Store.shared.rootFolder {  
    // ...  
}
```

也就是说，在初始构建时，每个文件夹 view controller 都假定它表示的是根目录。如果这是一个代表子目录的 view controller，它的父文件夹 view controller 将在 perform(segue:) 时将它的 folder 值设置为其他东西。这种方式可以确保文件夹 view controller 中的 folder 属性不是可选值，所以代码也就不需要包含关于检查文件夹对象是否存在的条件测试部分了，因为这个属性至少会是根文件夹。

在录音 app 中，model 对象 Store.shared 是一个延迟构建的单例。也就是说，第一个文件夹 view controller 在访问 Store.shared.rootFolder 时，可能是这个共享 store 实例被构建的时候。

PlayViewController (最上层 split view 中的 detail view) 使用的是初始为 nil 的 model 对象的可选值引用，它遵循的是设置身份引用时的第二种策略。

```
var recording: Recording? {  
    // ...  
}
```

当 recording 的值是 nil 时，PlayViewController 显示一个空白页面 (“没有选中录音”)。注意这里 nil 是一个预期中的状态，它并非是用来绕开 Swift 严格的初始化检查进行的妥协。从外界对录音属性的设定，可能是通过文件夹 view controller，也可能是通过状态恢复过程来完成的。不管是哪种情况，一旦这个主 model 对象被设置，controller 就通过更新 view 来进行响应。

文件夹 view controller 是 controller 响应主 model 变更的另一个例子。导航标题 (在屏幕顶部导航栏上显示的文件夹名字) 需要在 folder 被设定的时候进行更新：


```

var folder: Folder = Store.shared.rootFolder {
    didSet {
        tableView.reloadData()
        if folder === folder.store?.rootFolder {
            title = .recordings
        } else {
            title = folder.name
        }
    }
}
}

```

对 `tableView.reloadData()` 的调用确保了所有的 `UITableViewCell` 显示也进行了更新。

作为规则，每当我们读取初始的 `model` 数据时，我们必须同时对它的改变进行观察。在文件夹 `view controller` 的 `viewDidLoad` 中，我们将 `view controller` 添加为 `model` 通知的观察者：

```

override func viewDidLoad() {
    super.viewDidLoad()
    // ...
    NotificationCenter.default.addObserver(self,
        selector: #selector(handleChangeNotification(_)),
        name: Store.changedNotification, object: nil)
}

```

我们会在本章稍后提到的 `handleChangeNotification` 方法中讨论如何对这些通知进行处理。

状态恢复

MVC 中的状态恢复需要使用 `storyboard` 系统，它将扮演 `controller` 的角色。要实现这套系统，我们必须在 `AppDelegate` 中实现下面这些方法：

```

func application(_ application: UIApplication,
    shouldSaveApplicationState coder: NSCoder) -> Bool
{
    return true
}

```

```

func application(_ application: UIApplication,

```

```

shouldRestoreApplicationState(coder: NSCoder) -> Bool
{
    return true
}

```

当这两个方法被实现时，storyboard 系统就会接手。对于应当被 storyboard 系统自动保存和恢复的 view controller，我们要配置一个恢复 ID。比如，录音 app 的根 view controller 的恢复 ID 是 splitController，这可以通过 storyboard 编辑器的 Identity 面板进行设定。对于录音 app 中，除了 RecordViewController (它是故意做成不可保存的) 以外，我们为其他每个场景都配置了恢复 ID。

虽然 storyboard 系统可以确保这些 view controller 的存在，但是必须进行额外的工作，才能够保证每个 view controller 中存储的 model 数据与退出 app 前一致。为了存储这些额外的状态，每个 view controller 都必须实现 encodeRestorableState(with:) 和 decodeRestorableState(with:)。在 FolderViewController 中它的实现如下：

```

override func encodeRestorableState(with coder: NSCoder) {
    super.encodeRestorableState(with: coder)
    coder.encode(folder.uuidPath, forKey: .uuidPathKey)
}

```

编码的部分很简单，FolderViewController 将用来标识 Foldermodel 对象的 uuidPath 进行存储。解码部分要稍微复杂一些：

```

override func decodeRestorableState(with coder: NSCoder) {
    super.decodeRestorableState(with: coder)
    if let uuidPath = coder.decodeObject(forKey: .uuidPathKey) as? [UUID],
        let folder = Store.shared.item(atUUIDPath: uuidPath) as? Folder
    {
        self.folder = folder
    } else {
        if let index = navigationController?.viewControllers.index(of: self),
            index != 0
        {
            navigationController?.viewControllers.remove(at: index)
        }
    }
}

```

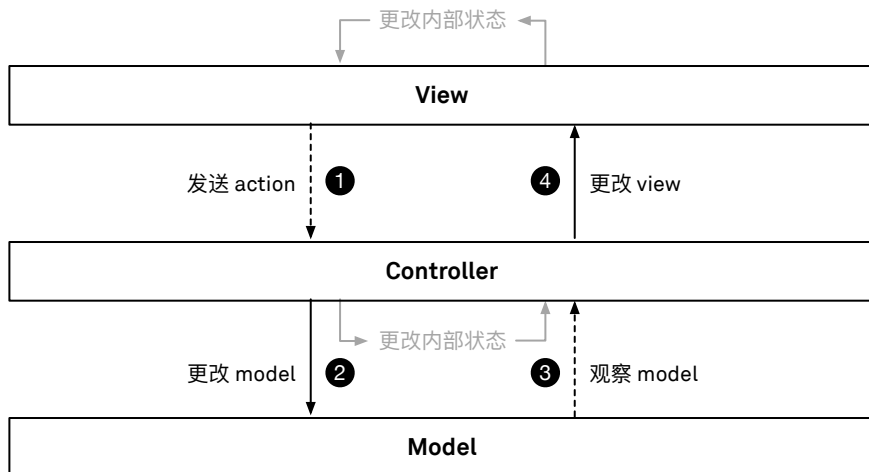
在对 `uuidPath` 进行解码后, `FolderViewController` 必须检查条目是否依然存在于 `store` 之中, 这样它才能将 `folder` 属性设置为该条目。如果该条目已经不在 `store` 之中, 那么 `FolderViewController` 必须尝试将它自身从导航 `controller` 的 `view controller` 列表中移除。

更改 Model

在最广泛的 MVC 的诠释中, 并不包含 `model` 实现方式的细节, 也不包含 `model` 应该如何变更, 或者 `view` 应该如何响应变更等内容。在最早版本的 `macOS` 中, 所遵循的是更早的一套文档 - `view` 模式, 让像是 `NSWindowController` 或者 `NSDocument` 这样的 `controller` 对象直接更改 `model` 来响应 `view action`, 并且直接在相同函数里对 `model` 进行更新的做法是十分普遍的。

在我们的 MVC 实现中, 我们认为对 `model` 改变的行为不应该和对 `view` 层级变更的行为发生在同一函数中。这些行为应该不受 `model state` 的任何影响。在构建阶段结束后, 对于 `view` 层级的变更应该遵循 MVC 中观察者模式的部分, 只发生在观察的回调中。

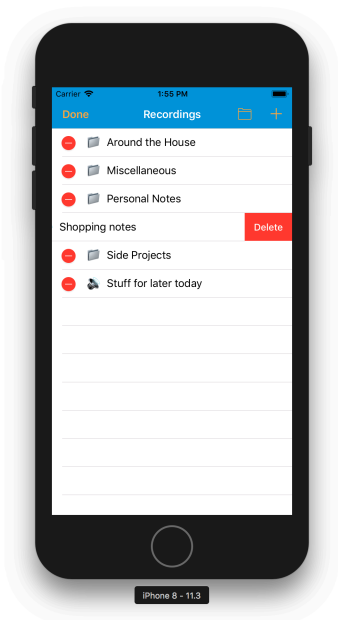
观察者模式是在 MVC 中维持 `model` 和 `view` 分离的关键。这种方式的优点在于, 不论变更究竟是源自哪里 (比如, `view` 事件、后台任务或者网络), 我们都可以确信 UI 是和 `model` 数据同步的。而且, 在遇到变更请求时, `model` 将有机会拒绝或者修改这个请求:



在下面, 我们会看看从一个文件夹中删除某个条目所需要的步骤。

步骤 1: Table View 发送 Action

在我们的示例 app 中, table view 的 data source 被 storyboard 设置为了文件夹 view controller。为了处理删除按钮的点击, table view 将调用它的数据源上的 `tableView(_:commit:forRowAt:)`:



步骤 2: View Controller 改变 Model

`tableView(_:commit:forRowAt:)` 的实现将会 (基于 index path) 查找应该删除的条目, 并要求父文件夹将它移除:

```
override func tableView(_ tableView: UITableView,
    commit editingStyle: UITableViewCellEditingStyle,
    forRowAt indexPath: IndexPath)
{
    folder.remove(folder.contents[indexPath.row])
}
```

注意，我们没有直接将相应的 cell 从 table view 中移除出去。这个操作只在我们观察到 model 变更时发生。

Folder 上的 remove 方法会通过调用 item.deleted() 来通知条目它已经被删除了。接下来它将这个条目从文件夹的内容中移除。然后它告诉 store 应该保存数据，这包括了刚刚所进行的更改的详细信息：

```
func remove(_ item: Item) {
    guard let index = contents.index(where: { $0 === item }) else { return }
    item.deleted()
    contents.remove(at: index)
    store?.save(item, userInfo: [
        Item.changeReasonKey: Item.removed,
        Item.oldValueKey: index,
        Item.parentFolderKey: self
    ])
}
```

如果被移除的项目是一个录音，item.deleted() 将会把相关联的文件从文件系统中删除掉。如果它是一个文件夹，它会递归调用将所有子文件夹和录音都移除。

持久化 model 对象以及发送更改通知发生在 store 的 save 方法中：

```
func save(_ notifying: Item, userInfo: [AnyHashable: Any]) {
    if let url = baseUrl, let data = try? JSONEncoder().encode(rootFolder) {
        try! data.write(to: url.appendingPathComponent(.storeLocation))
        // 错误处理被跳过了
    }
    NotificationCenter.default.post(name: Store.changedNotification,
        object: notifying, userInfo: userInfo)
}
```

步骤 3: View Controller 观察 Model 变更

在构建部分，我们看到文件夹 view controller 已经在 viewDidLoad 为 store 变更的通知设立了观察：

```
override func viewDidLoad() {
    super.viewDidLoad()
```

```
// ...
NotificationCenter.default.addObserver(self,
    selector: #selector(handleChangeNotification(_:)),
    name: Store.changedNotification, object: nil)
}
```

上一步里在 store 上调用的 save 将发送变更通知，作为回应，这个观察将触发并调用 handleChangeNotification。

步骤 4: View Controller 改变 View

当 Store 更改的通知到达时，view controller 的 handleChangeNotification 将被执行，并在 view 层级上作出对应的变更。

对于通知的最简处理方式可能就是不论任意类型的 model 通知到达时，都重新加载 table view 的数据。通常来说，对于 model 通知的正确处理要基于对通知中描述的数据变更进行的正确的理解。因此，我们的实现中对 model 变更的特质，也就是发生变化的索引号，以及发生变化的种类，通过通知的 userInfo 字典进行了发送。

在本例中，对通知的处理涉及到对 tableView.deleteRows(at:with:) 的调用：

```
@objc func handleChangeNotification(_ notification: Notification) {
    // ...
    if let changeReason = userInfo[Item.changeReasonKey] as? String {
        let oldValue = userInfo[Item.newValueKey]
        let newValue = userInfo[Item.oldValueKey]
        switch (changeReason, newValue, oldValue) {
            case let (Item.removed, _, (oldIndex as Int)?):
                tableView.deleteRows(at: [IndexPath(row: oldIndex, section: 0)],
                    with: .right)
                // ...
        }
    } else {
        tableView.reloadData()
    }
}
```

这段代码中有一个值得注意的缺失的部分：我们没有更新 view controller 自身的任何数据。View controller 的 folder 值是一个共享的引用值，它直接使用 model 层中的对象，所以

它已经是更新过的了。在上面的 `tableView.deleteRows(at:with:)` 调用之后, `table view` 将会调用文件夹 `view controller` 上的 `data source` 的实现, 它们将通过访问共享的 `folder` 引用返回最新状态的数据。在 [MVC+VS](#) 一章中, 我们会看到如何使用一层替换的 `model` 层来处理非对象的值类型。

译者注: 如果使用值类型, 它将在赋值时发生复制, 因此 `store` 中对 `folder` 的更改不会影响到 `view controller` 中的 `folder`。这样一来, 我们将会需要额外的逻辑来同步两个 `folder` 的状态。

我们还应该澄清, 这个通知处理还依赖了一个捷径: `model` 存储条目的顺序和显示的顺序是一致的。这并不理想, `model` 不应该知道它的数据是**如何**被显示的。从概念上说, 一个更清晰(但需要更多代码)的实现应该需要 `model` 发送 **set** 的变动信息, 而不是 `array` 的信息, 通知的处理者需要使用它自己的排序, 将变更之前和之后的状态合并, 并确定被删除条目的索引。在 [MAVB](#) 一章中, 我们会把排序从 `model` 层移除, 并将它移动到 `view` 绑定器中。

现在我们在 MVC 中完成了“变更 `model`”事件回路。因为我们只在 `model` 变更的响应中更新 UI, 而不是直接在 `view action` 的响应中这么做, 即使文件夹被以其他方式(比如一个网络事件)移除出 `model`, 或者是 `model` 拒绝这次变更时, UI 也会正确更新。这是一种确保 `view` 层始终与 `model` 层同步的十分健壮的方式。

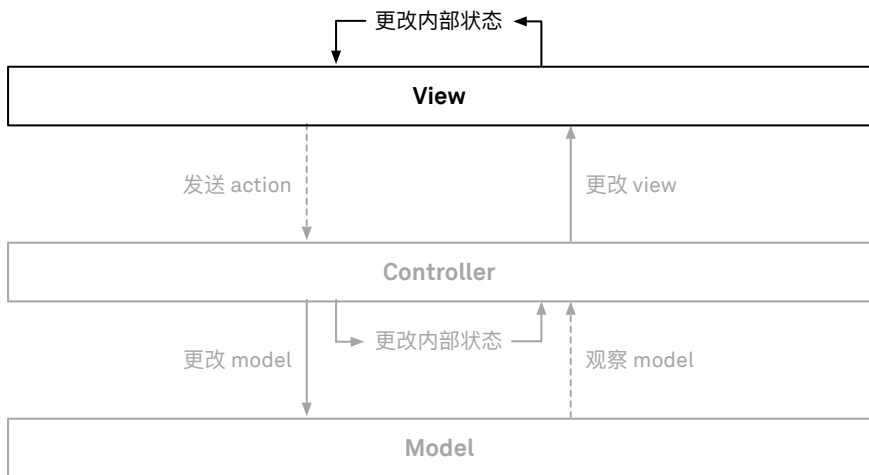
更改 View State

MVC 的 `model` 层起源于典型的基于文档的 app: 任何在保存操作中写入文档的状态都被当作是 `model` 的一部分来考虑。其他的任意状态 - 包括像是导航状态, 临时的搜索和排序值, 异步任务的反馈以及未提交的更改 - 传统意义上是被排除在 MVC 的 `model` 定义之外的。

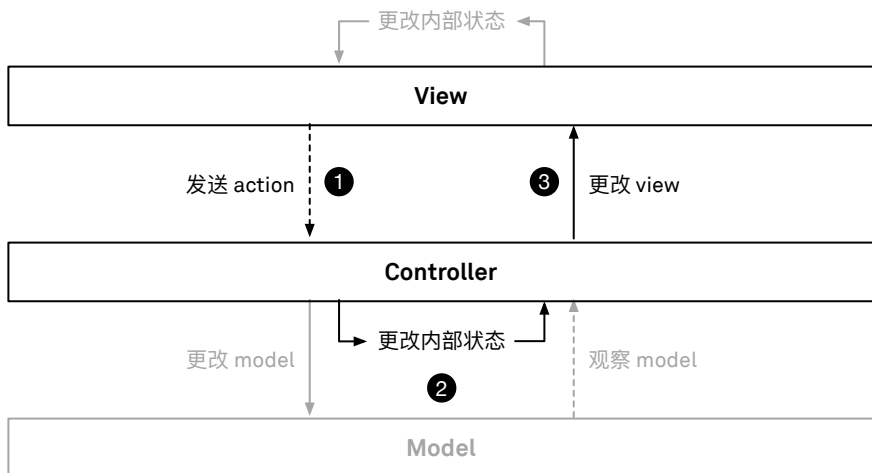
在 MVC 中, 这些被我们统称为 `view state` 的“其他”状态没有被包含在模式的描述中。依照传统的面向对象的原则, 任意的对象都可以拥有内部状态, 这些对象也不需要内部状态的变化传达给程序的其余部分。

基于这种内部处理, `view state` 不需要遵守任何一条程序中的清晰路径。任意 `view` 或者 `controller` 都可以包含状态, 这些状态由 `view action` 进行更新。`view state` 的处理尽可能地在本地进行: 一个 `view` 或者 `view controller` 可以独自响应用户事件, 对自身的 `view state` 进行更新。

大部分的 `UIView` 拥有内部状态, 它们可以响应 `view action` 并进行更新, 而不必将这些改变继续传递下去。比如, 一个 `UISwitch` 就可以响应用户的点击事件, 从 ON 状态切换到 OFF:



如果 view 本身不能更改它的状态，那么事件回路就要变长一步。比如，当一个按钮的标签需要在它被点击时发生变更 (比如我们的播放按钮) 或者用户点击一个列表中的 cell 时，新的 view controller 需要被推送到导航栈上，都属于这样的变更。



View state 依然存在于一个特定的 view controller 和它的 view 中。不过，对比一个改变自身状态的 view，我们现在有机会自定义一个 view 的状态变更 (比如下面第一个例子中播放按钮标

题的变更) 或者让 view state 跨 view 进行变更 (由下面的第二个例子, 推送新的文件夹 view controller, 进行举例说明)。

示例 1: 更新播放按钮

PlayViewController 中的播放按钮会依据播放状态将它的标题在 “Play”, “Pause” 和 “Resume” 之间切换。从用户的视角来看, 当按钮表示为 “Play” 时, 按下它会将标题改变为 “Pause”; 在播放结束前再次按下它会使其变为 “Resume”。让我们来仔细看看在 Cocoa MVC 中要达到这一点所需要的步骤吧。

步骤 1: 按钮向 View Controller 发送 Action

播放按钮通过 storyboard 中的 IBAction 连接到 PlayViewController 的 play 方法。点击这个按钮会调用 play 方法:

```
@IBAction func play() {  
    // ...  
}
```

步骤 2: View Controller 改变内部状态

play 方法的第一行负责更新音频播放器的状态:

```
@IBAction func play() {  
    audioPlayer?.togglePlay()  
    updatePlayButton()  
}
```

步骤 3: View Controller 更新按钮

play 方法的第二行调用了 updatePlayButton, 该方法依据 audioPlayer 的状态直接为播放按钮设置新的标题:

```
func updatePlayButton() {  
    if audioPlayer?.isPlaying == true {  
        playButton?.setTitle(.pause, for: .normal)  
    }
```

```

    } else if audioPlayer?.isPaused == true {
        playButton?.setTitle(.resume, for: .normal)
    } else {
        playButton?.setTitle(.play, for: .normal)
    }
}

```

.pause, .resume 和 .play 是定义在 String 上的本地化后的静态常量。

现在，一切已经搞定了，这个过程中相关的部件数量是最少的：按钮将事件发送给 PlayViewController，接下来，后者为按钮设定新的标题。

示例 2: 推入文件夹 View Controller

文件夹 view controller 中的 table view 负责展示两种对象：录音和子文件夹。当用户点击子文件夹时，我们需要配置新的文件夹 view controller，并将它推到导航栈上。因为我们使用了带有 segue 的 storyboard 来达成这个目的，下面的具体的步骤仅仅是把上面图表中的步骤松散地关联起来。不过，总体的原则是相同的。

步骤 1: 触发 Segue

因为在 storyboard 中，子文件夹的单元格通过一个 push segue 与文件夹 view controller 相连，所以点击子文件夹的单元格将会触发一个 showFolder segue。这会导致 UIKit 为我们创建一个新的文件夹 view controller。

这个步骤是 target/action 模式的变种。在幕后，UIKit 为我们做了更多的工作，但是结果是源 view controller 的 prepare(for:sender:) 方法被调用。

步骤 2 & 3: 配置新的文件夹 View Controller

prepare(for:sender:) 会通知当前的文件夹 view controller 哪个 segue 正在发生中。在我们检查 segue 的 identifier 后，我们对新的文件夹 view controller 进行配置：

```

override func prepare(for segue: UIStoryboardSegue, sender: Any?) {
    guard let identifier = segue.identifier else { return }
    if identifier == .showFolder {
        guard

```

```

        let folderVC = segue.destination as? FolderViewController,
        let selectedFolder = selectedItem as? Folder
    else { fatalError() }
    folderVC.folder = selectedFolder
}
// ...
}

```

首先，我们检查目标 view controller 是否拥有正确的类型，以及我们是否确实选择了一个文件夹。如果两个条件不能同时满足，那么这应该是一个编程上的错误，我们让 app 崩溃。如果所有事情都和预期一样，我们将子文件夹设置给新的文件夹 view controller。

Storyboard 的机制会负责实际将新的 view controller 展示出来的工作。当我们配置好新的 view controller 后，UIKit 会把它推入导航栈 (我们不需要自己去调用 pushViewController)。将 view controller 推入导航栈的操作，将使导航 controller 把所推入的 view controller 的 view 装载到 view 层级上。

和上面的播放按钮例子类似，UI 事件 (选择子文件夹单元格) 尽可能地在本地处理。Segue 机制模糊了这个事件确切的路径，但是不论如何，从我们的代码的角度来看，除了初始的 view controller 以外，没有别的部件受到影响。View state 被相关的 view 及 view controller 隐式表示。

如果想要在 view 层级的不同部分共享 view state，我们需要在 view (或者 view controller) 层级上找到它们共同的祖先，并在那儿管理状态。比如，想要在播放按钮标签和播放器之间共享播放状态的话，我们可以将状态存储到 PlayViewController 中去。当我们在几乎所有部件中需要某个 view state 时 (比如，一个决定我们的 app 是否应该处于深色模式的布尔值)，我们需要将它放在最顶层的 controller 中 (比如，app 代理)。但是，在实践中，将 view state 放到层级的顶层 controller 对象中的做法并不常见，因为这会要求层级的每一层之间存在通讯的管道。所以，大部分人会选择使用单例来作为替代。

测试

自动测试可以由好几种不同形式进行。从最小的粒度到最大的粒度，它们包括：

- 单元测试 (将独立的函数独立出来，并测试它们的行为)。
- 接口测试 (使用接口输入并测试接口输出得到的结果，输入和输出通常都是函数)。
- 集成测试 (在整体上测试程序或者程序的主要部分)。

虽然接口测试和数据驱动的回归测试已经有四十年甚至更长的历史了，而现代的单元测试仅仅是从上世纪 90 年代才真正起步，不过单元测试只花了十多年的时间就成为了 app 中最常见的测试方式。不过即使是现在，许多 app 除了人工测试以外并没有进行常规的测试。

所以已经有超过 20 年历史的 Cocoa MVC 模式在被创建的时候并没有考虑单元测试这件事情就一点都不奇怪了。不管怎样，我们都可以测试 model 层，因为它和程序的其他部分是独立的，但是这并不会对测试面向用户的状态有什么帮助。我们可以使用 Xcode 的 UI 测试，也就是运行整个程序并尝试使用 VoiceOver 或者辅助访问的 API 读取屏幕的自动化脚本，但是这样的测试速度很慢，容易导致时间上的问题，而且难以提取精确的结果。

如果我们真的想要在代码层级对 MVC 的 controller 和 model 层进行测试，唯一可行的选项是写成集成测试。集成测试需要构建一个自包含版本的 app，操作其中的某些部分，然后从其他部分读取数据，确保结果在对象之间按照期望的方式传递。

对于录音 app，这种测试需要一个 Store 对象，所以我们可以创建一个 (只在内存中存在的) 不包含 URL 的 store，并添加一个测试条目 (文件夹和录音)：

```
func constructTestingStore() -> Store {
    let store = Store(url: nil)

    let folder1 = Folder(name: "Child 1", uuid: uuid1)
    let folder2 = Folder(name: "Child 2", uuid: uuid2)
    store.rootFolder.add(folder1)
    folder1.add(folder2)

    let recording1 = Recording(name: "Recording 1", uuid: uuid3)
    let recording2 = Recording(name: "Recording 2", uuid: uuid4)
    store.rootFolder.add(recording1)
    folder1.add(recording2)

    store.placeholder = Bundle(for: FolderViewControllerTests.self)
        .url(forResource: "empty", withExtension: "m4a")!

    return store
}
```

store.placeholder 是 store 中的一个专门用来测试的特性：如果 URL 是 nil，那么这个占位符就会在 Store.fileURL(for:) 被调用时作为被获取的录音的音频文件返回。构建了 store 之后，我们现在需要一个使用该 store 的 view controller 层级：

```

func constructTestingViews(store: Store,
    navDelegate: UINavigationControllerDelegate)
-> (UIStoryboard, AppDelegate, UISplitViewController,
    UINavigationController, FolderViewController)
{
    let storyboard = UIStoryboard(name: "Main", bundle: Bundle.main)
    let navigationController =
        storyboard.instantiateViewController(withIdentifier: "navController")
    as! UINavigationController
    navigationController.delegate = navDelegate

    let rootFolderViewController = navigationController.viewControllers.first
    as! FolderViewController
    rootFolderViewController.folder = store.rootFolder
    rootFolderViewController.loadViewIfNeeded()
    // ...
    window.makeKeyAndVisible()
    return (storyboard, appDelegate, splitViewController,
        navigationController, rootFolderViewController)
}

```

上面展示了主导航 controller 和根 view controller 的创建方式。函数在后面用类似的方式继续构建副导航 controller，play view controller，split view controller，窗口，以及 app delegate，这形成了一个完整的用户界面。一路向上构建到窗口后，我们需要调用 `window.isHidden = false`，否则许多动画和展示行为将不会发生。

注意我们将导航 controller 的代理设置为了 `navDelegate` 参数，这个参数将会是负责运行测试的 `FolderViewControllerTests` 类的实例，根 view controller 的 `folder` 被设置为了测试用的 `store` 的 `rootFolder`。

我们在测试的 `setUp` 方法中调用上面的构造函数，来初始化 `FolderViewControllerTests` 实例的成员。当这一切完成时，我们就可以开始写测试了。

集成测试通常需要对初始环境进行某种程度的配置，才能在它上面执行某个 action 并测量结果。在 app 集成测试中，对结果的测量可能会很简单 (比如访问配置好的环境中的某个对象上的可以读取的属性)，也可能会非常困难 (比如在 Cocoa 框架中进行多次异步交互)。

一个相对简单的测试例子是，在删除列表的行的时候，测试 `commitEditing` 行为：

```

func testCommitEditing() {
    // 验证要调用的行为已被连接
    let dataSource = rootFolderViewController.tableView.dataSource
    as? FolderViewController
    XCTAssertEqual(dataSource, rootFolderViewController)

    // 确认删除之前条目存在
    XCTAssertNotNil(store.item(atUUIDPath: [store.rootFolder.uuid, uuid3]))
    // 执行删除行为
    rootFolderViewController.tableView(rootFolderViewController.tableView,
        commit: .delete, forRowAt: IndexPath(row: 1, section: 0))
    // 确认条目已被删除
    XCTAssertNil(store.item(atUUIDPath: [store.rootFolder.uuid, uuid3]))
}

```

上面的测试验证了根 view controller 被作为 data source 正确配置，并直接对 data source 的 tableView(_:commit:forRowAt:) 进行了调用。最后验证了这个 action 将条目从 model 中被删除。

当涉及到动画或者潜在的依赖于模拟器的变更时，测试就会更加复杂。在录音 app 中，最复杂的测试应该是在文件夹 controller 中选择某个录音条目，并在 split view 的 detail view 一侧确认录音能被正确显示。这个测试需要处理 split view controller 的折叠和非折叠状态的不同，而且它需要等待导航 controller 的推入行为结束：

```

func testSelectedRecording() {
    // 选择一行，这样 `prepare(for:sender:)` 可以读取选择的行
    rootFolderViewController.tableView.selectRow(at: IndexPath(row: 1, section: 0),
        animated: false, scrollPosition: .none)
    // 处理 split view controller 的折叠和非折叠状态
    if self.splitViewController.viewControllers.count == 1 {
        ex = expectation(description: "Wait for segue")
        // 触发转场
        rootFolderViewController.performSegue(withIdentifier: "showPlayer",
            sender: nil)
        // 等待导航 controller 将折叠的 detail view 推入
        waitForExpectations(timeout: 5.0)
        // 移动至 `PlayViewController`
        let collapsedNC = navigationController.viewControllers.last
        as? UINavigationController
    }
}

```

```

    let playVC = collapsedNC?.viewControllers.last as? PlayViewController
    // 测试结果
    XCTAssertEqual(playVC?.recording?.uuid, uuid3)
} else {
    // 处理非折叠状态
}
}
}

```

创建的 expectation (ex) 会在主导航 controller 将新的 view controller 推入导航栈时被 fulfill。主导航 controller 的这个变更发生在 detail view 被折叠到 master view 的时候 (也就是发生在除了 iPhone Plus 横屏模式以外的所有 iPhone 的紧凑显示模式下)：

```

func navigationController(_ navigationController: UINavigationController,
    didShow viewController: UIViewController, animated: Bool) {
    ex?.fulfill()
    ex = nil
}

```

我们为文件夹 view controller 写的测试覆盖了大约 80% 的代码行数，并测试了所有重要行为，虽然这些集成测试可以搞定工作，但 testSelectedRecording 测试已经说明了，想要正确书写集成测试，我们需要大量关于 Cocoa 框架是如何操作的知识，这并非易事。

讨论

MVC 有其优点，它是 iOS 开发中阻力最低的架构模式。Cocoa 中的每个类都是在 MVC 的条件下进行测试的。像 storyboard 这样的特性，是与框架和类高度集成的，它可以和使用 MVC 的程序更顺滑地一同工作。在网络上搜索时，相比于其他任何一种设计模式，你可以找到更多按照 MVC 进行实现的例子。而且，在所有模式中，MVC 通常都是代码量最少，设计开销最小的模式。

作为在本书探索中所有模式里使用得最多的一种模式，我们对 MVC 的缺点的理解也最为深刻。具体来说，MVC 中有两个最常见的问题。

观察者模式失效

第一个问题是，model 和 view 的同步可能失效。当围绕 model 的观察者模式没有被完美执行时，这个问题就会发生。常见的错误是，在构建 view 时读取了 model 的值，而没有对后续的

通知进行订阅。另一个常见错误是在变更 `model` 的同时去更改 `view` 层级，这种做法假设了变更的结果，而没有等待 `model` 进行通知，如果 `model` 拒绝了这个变更的话，就会发生错误。这类错误会使得 `view` 和 `model` 不同步，奇怪的行为也随之而来。

不幸的是，Cocoa 并没有为验证观察者模式是否被正确实现提供任何检查或者内部机制。解决方法只有严格地遵守观察者模式：当读取 `model` 值时，也需要对它进行订阅。其他一些架构（比如 `TEA` 或者 `MAVB`）将初始读取和订阅合并为了一个语句，这样一来就确保了观察的错误不会发生。

肥大的 View Controller

MVC，特别是 Cocoa MVC 中的第二个问题，是这个模式经常会造成很大的 `view controller`。`View controller` 需要负责处理 `view` 层（设置 `view` 属性，展示 `view` 等），但是它同时也负责 `controller` 层的任务（观察 `model` 以及更新 `view`），最后，它还要负责 `model` 层（获取数据，对其变形或者处理）。结合它在架构中的中心角色，这使得我们很容易在不经意间把所有的职责都赋予 `view controller`，从而迅速让程序变得难以管理。

对于一个 `view controller` 可以庞大到什么地步，并没有明确的限制。当一个 Swift 文件超过 2,000 行时，Xcode 在打开和浏览时就会产生明显的卡顿，所以我们可能最好不要让文件尺寸到达这种程度。大部分屏幕大概最多只能显示 50 行代码，所以如果屏数太多的话，想要通过滚动来找到代码在视觉上也会开始变得困难。

不过关于肥大的 `view controller` 最主要的争论不是关于代码的行数，而在于所保存的状态的数量。当整个文件就像 `view controller` 这样是一个单独的类的时候，所有的可变状态都将被文件中的各个部分共享，每个函数需要精诚合作，来共同读取和维护这些状态，避免彼此矛盾。将这些对状态的维护分离到不同的接口中，往往可以带来对于数据依赖的更好的思考，并且可以限制那些为了稳定性而必须合作并遵守特定规则的代码的数量。

改进

观察者模式

对于基础的 MVC 实现，我们选择使用 Foundation 的 `NotificationCenter` 来进行 `model` 通知的广播。作出这个选择的原因是我们希望使用基本的 Cocoa 来进行实现，而不去使用框架或者其他抽象。

不过，这样的实现要求我们在很多地方一起对某个值进行正确的更新。比如文件夹 `viewController` 中的 `folder`，是在父文件夹 `viewController` 的 `prepare(for:sender:)` 方法中被设定的：

guard

```
let folderVC = segue.destination as? FolderViewController,
let selectedFolder = selectedItem as? Folder
else { fatalError() }
folderVC.folder = selectedFolder
```

在之后，这个文件夹 `controller` 将对 `model` 的通知进行观察，以获取该值的变更情况：

```
override func viewDidLoad() {
    super.viewDidLoad()
    // ...
    NotificationCenter.default.addObserver(self,
        selector: #selector(handleChangeNotification(_)),
        name: Store.changedNotification, object: nil)
}
```

在通知处理的部分，如果发出通知的对象是当前文件夹，那么对文件夹进行更新：

```
@objc func handleChangeNotification(_ notification: Notification) {
    // 处理对当前文件夹的变更
    if let item = notification.object as? Folder, item === folder {
        let reason = notification.userInfo?[Item.changeReasonKey] as? String
        if reason == Item.removed, let nc = navigationController {
            nc.setViewControllers(nc.viewControllers.filter { $0 !== self },
                animated: false)
        } else {
            folder = item
        }
    }
    // ...
}
```

最好是有一种方式，能让 `folder` 的初始设定和在 `viewDidLoad` 中建立观察者之间没有时间空隙。如果能把需要对 `folder` 进行设定的地方统一，那就更好了。

使用键值观察 (KVO) 来替代通知是可选项之一，但是，在大部分情况下这通常需要同时观察多个不同的键路径 (比如在观察父文件夹的 `children` 属性的同时还要观察当前的子文件夹)，这让 KVO 和通知的方式相比，实际上并没有更加稳定。由于它需要每个被观察的属性都声明为 `dynamic`，这也让它在 Swift 中远远没有 Objective-C 中那样流行。

对观察者模式最简单的改进方式是，将 `NotificationCenter` 进行封装并为它实现 KVO 所包含的**初始化的概念**。这个概念会在观察被建立的同时发送一个初始值，这允许我们将**设定初始值和观察后续值**的操作合并到一个单一管道中去。

这也让我们得以将 `viewDidLoad` 中的观察的代码和上面 `handleChangeNotification` 中的代码合并为下面的代码：

```
observations += Store.shared.addObserver(at: folder.uuidPath) {  
    [weak self] (folder: Folder?) in  
    guard let strongSelf = self else { return }  
    if let f = folder { // 更改  
        strongSelf.folder = f  
    } else { // 删除  
        strongSelf.navigationController.map {  
            $0.setViewControllers($0.viewControllers.filter { $0 != self },  
                                animated: false)  
        }  
    }  
}
```

这么做并没有减少很多代码，但是 `self.folder` 值现在只在两个地方被设置了 (在观察回调中，以及在父文件夹 `controller` 的 `prepare(for:sender:)` 中)，我们成功减少了代码中对它进行更改的路径数量。另外，我们的用户代码中不再需要进行动态类型转换了，在初始值设定和建立观察之间也不再存在空隙：初始值不再是实际的数据，它只是一个标识，通过观察回调是我们所需要的访问实际数据的唯一方式。

这种类型的 `addObserver` 可以被实现在 `Store` 的扩展上，而不需要对 `Store` 本身进行任何更改，这是这种实现的一个优势：

```
extension Store {  
    func addObserver<T: AnyObject>(at uuidPath: [UUID],  
        callback: @escaping (T?) -> () -> [NSObjectProtocol]  
    ) {  
        guard let item = item(atUUIDPath: uuidPath) as? T else {
```

```

        callback(nil)
        return []
    }
    let o = NotificationCenter.default.addObserver(
        forName: Store.changedNotification,
        object: item, queue: nil) { notification in
        if let item = notification.object as? T, item === item {
            let reason = notification.userInfo?[Item.changeReasonKey] as? String
            if reason == Item.removed {
                return callback(nil)
            } else {
                callback(item)
            }
        }
    }
    callback(item)
    return [o]
}
}

```

Store 依然发送同样的通知，我们只是以另一种方式订阅了这个通知，处理通知的数据 (比如检查 `Item.changeReasonKey` 是否存在)，并且处理一些其他模板代码，这样每个 view controller 需要的工作就能简单一些。

在 [MVC+VS](#) 一章中，我们展示了如何在这个方法的基础上更进一步。MVC+VS 是 MVC 的一个变形，所有的 model 都使用类似的模式进行读取：我们在同一个回调中接收初始值和之后的变更。

肥大 View Controller 的问题

非常大的 view controller 通常进行了它们的主要工作 (观察 model，展示 view，为它们提供数据，以及接收 view action) 之外的无关工作；它们要么应该被打散成多个各自管理一个较小部分的 view 层级的 controller；要么就是因为接口和抽象没能将一段程序的复杂度封装起来，view controller 做了太多打扫垃圾的工作。

在很多情况下，解决这个问题的最佳途径就是主动地将尽可能多的功能移动到 model 层中。比如排序，数据获取和处理等方法，因为不是 app 的存储状态的一部分，所以通常会被放到

controller 中。但是它们依然与 app 的数据和专用逻辑相关，把它们放在 model 中会是更好的选择。

在录音 app 中最大的 view controller 是 FolderViewController，它大约有 130 行代码，这远远没有能够达到让我们看到任何大问题的阶段。为了说明，我们会简单看一些 GitHub 上流行的 iOS 项目的大尺寸 view controller。请注意，我们没有打算以任何角度贬低这些项目，我们只是从它们开源的特性来研究为什么一个 view controller 会增长到上千行代码。另外请注意，我们将要讨论的行数数据是由 cloc 生成的，空格和注释已经被忽略了。

Wikipedia 的 PlacesViewController

本书中评估的代码是 2c1725a 版本的 PlacesViewController.swift。

这个文件一共有 2,326 行。view controller 包含了下列角色：

1. 配置和管理地图 view，该 view 中显示位置结果 (400 行)
2. 通过 location manager 获取用户位置 (100 行)
3. 执行搜索并且收集结果 (400 行)
4. 将结果分组，并显示在地图的可见区域 (250 行)
5. 将搜索建议填充到 table view 并进行管理 (500 行)
6. 处理像是搜索建议的 table view 这样的覆盖层的布局 (300 行)

这个例子展示了 view controller 肥大化的三大病因：

1. 管理了超过一个的主 view (地图 view 和搜索建议 table view)。
2. 创建和执行异步任务 (比如获取用户位置)，虽然 view controller 只对任务的结果感兴趣 (本例中，是用户的位置)。
3. Model/专用逻辑 (搜索和处理结果) 在 controller 层被操作。

我们可以通过将主 view 分开到它们自己的更小的 controller 中简化场景的 view 依赖。它们甚至都不需要是 UIViewController 的实例，而只需要是场景所拥有的子对象即可。在父 view controller 中剩下的工作就只有集成和布局了 (而且原来复杂的布局也能够随着 view controller 的简化而被重构)。

我们可以创建工具类来执行像是获取用户位置信息这种异步任务，在 controller 中，唯一需要的代码只是创建任务和回调闭包。

从 app 设计架构的视角来说，最大的问题在于 controller 层中的 model 或者专用逻辑。这些代码缺少一个实际用来执行搜索和收集搜索结果的模型。没错，在 view controller 中确实有一个 `dataStore` 对象，但是代码没有围绕它进行任何抽象，所以它自己并没有帮上什么忙。View controller 实际上还是自己做了所有的搜索和数据处理工作，这些任务本来应该在另外的地方被处理。甚至是像按照可视区域对结果分组这样的操作，也应该由模型或者是在模型和 view controller 之间的转换对象来进行执行。

译者注：将部分 view controller 中的代码抽离出来，在本质上并没有降低整个程序的复杂度。但是，这么做确实降低了 view controller 本身的复杂度。在优化一个 view controller 时，我们可以对照本章中 MVC 的行为框图，来逐一确认某项操作是否真的合适被放在 view controller 中。将非直接相关的部分移出去，是重构工作中最为直接简单，容易实现，同时也是最有效果的步骤

WordPress 的 AztecPostViewController

本书中评估的代码是 6dcf436 版本的 AztecPostViewController.swift。

该文件有 2,703 行，view controller 的职责包括：

1. 通过代码创建子 view (300 行)
2. 自动布局约束和放置标题 (200 行)
3. 管理文章发布流程 (100 行)
4. 设置子 view controller，并处理它们的结果 (600 行)
5. 协调，观察和管理 text view 的输入 (300 行)
6. 显示警告和警告的内容 (200 行)
7. 追踪媒体文件上传 (600 行)

媒体文件的上传是一个专用服务，应该很容易被移动到它自己的模型层服务中去。

同时，剩余代码的 75% 都能够通过改善程序其他部件的接口来移除掉。

在 `AztecPostViewController` 中所处理的模型，服务或者 `child view controller` 都没有任何一个能用一行代码来使用。显示弹窗警告的代码在不同的地方出现了好多次。`Child view controller` 的设置花费了 20 行，而在它结束后，又花了 20 行来进行处理。虽然 `text view` 是自定义的 `Aztec.TextView`，但在 `view controller` 中依然有上百行的代码在调整它的行为。

在这些例子中，其他部件都无法完成的它们自己的工作，而 `view controller` 总是被用来修补它们。其实，这些行为应当尽可能地集成到各自的部件中去。当我们无法更改一个部件的行为时，我们可以围绕这个部件写一个封装，而不是将这些逻辑放到 `view controller` 里去。

Firefox 的 `BrowserViewController`

本书中评估的代码是 98ec57c 版本的 `BrowserViewController.swift`。

这个文件的长度是 2,209 行。这个 `view controller` 包含了 1,000 行以上的代理实现，包括 `URLBarDelegate`, `TabToolbarDelegate`, `TabDelegate`, `HomePanelViewControllerDelegate`, `SearchViewControllerDelegate`, `TabManagerDelegate`, `ReaderModeDelegate`, `ReaderModeStyleViewControllerDelegate`, `IntroViewControllerDelegate`, `ContextMenuHelperDelegate` 和 `KeyboardHelperDelegate`。

它们是什么，它们又做了什么？

基本上来说，`BrowserViewController` 是整个程序的顶层 `view controller`，这些代理实现表示低层级的 `view controller` 使用 `BrowserViewController` 来在程序中进行行为中继。

没错，`controller` 层确实有在程序间传递行为的责任，所以这个例子的问题并不是模型或者专用逻辑的职责被放到了错误的层中。不过这些代理中很多部分其实和 `BrowserViewController` 管理的 `view` 并没有关系 (它们只是想要访问存储在 `BrowserViewController` 中的其他部件或者状态)。

相比于将这个责任放在已经承担了很多其他责任的 `view controller` 上，这些代理回调其实可以全部重新移动到一个专门用来在 `controller` 层进行中继处理的协调器或者抽象的 (非 `view controller` 的) `controller` 上。

使用代码而不是 Storyboard

如果不使用 `storyboard`，我们可以选择用代码来定义我们的 `view` 层级。这样的变更能给我们在构建阶段更多的控制力，其中最大的优点在于，我们可以更好地掌控依赖。

比如，在使用 storyboard 时，没有办法能够保证在 `prepare(for:sender:)` 中所有 view controller 所需要的属性都被设置了。回忆一下，我们已经使用了两种不同的方式进行模型传递：有一个默认值的方式 (比如 `FolderViewController`)，以及使用可选值类型 (比如 `PlayViewController`)。两种方式都没有保证对象一定被传递；如果我们忘记了这件事，代码将默默地继续运行，但是要么值是错误的，要么会是一个空白值。

当我们摆脱 storyboard 时，我们可以在构建过程中得到更多的控制权，并可以让编译器确保必要的参数被正确传递。注意，我们并不需要完全脱离 storyboard，作为起步，我们可以仅仅移除掉所有的 segue，然后手动去运行它们。要创建一个文件夹 view controller，我们可以添加如下静态方法：

```
extension FolderViewController {
    static func instantiate(_ folder: Folder) -> FolderViewController {
        let sb = UIStoryboard(name: "Main", bundle: nil)
        let vc = sb.instantiateViewController(withIdentifier: "folderController")
        as! FolderViewController
        vc.folder = folder
        return vc
    }
}
```

当我们使用 `instantiate` 方法时，编译器将帮助并确保我们提供了 `folder`；这是不可能被忘记的。理想情况下，`instantiate` 应当是一个初始化方法，但是这只有在我们完全将 storyboard 移除的时候才可能办到。

我们可以通过为 view 类添加自定义的初始化方法，或者写一个函数来构建特定的 view 层级的方式，将相同的技术应用到 view 的构建过程中去。总体上，不使用 storyboard 将让我们可以利用 Swift 的所有语言特性：泛型 (比如配置一个泛型的 view controller)，一等函数 (比如，使用函数配置 view 外观或者设置回调)，带有关联值的枚举 (比如依据模型而互斥的状态) 等等。在本书写作的时候，storyboard 并不支持这些特性。

在扩展中进行代码重用

要在不同的 view controller 间共享代码，一个常见的方法是创建一个包含共通功能的父类。然后 view controller 就可以通过子类来获得这些功能了。这种技术可以工作，但是它有一个潜在的不足：我们只能为我们的新类选定单个父类。比如说，我们不能同时继承 `UIPageViewController` 和 `UITableViewController`。这种方式还经常会导致我们常说的**上帝**

view controller 的问题：一个共享的父类包括了项目中全部的共享的功能。这样的类通常会变得非常复杂，难以维护。

在 **view controller** 中共享代码的另一种选择是使用扩展。在多个 **view controller** 中都出现的方法有时候能够被添加到 **UIViewController** 的扩展中去。这样一来，所有的 **view controller** 都能获取这个方法了。比如，我们可以为 **UIViewController** 添加一个显示文本警告的简便方法。

为了扩展能够有用，通常我们需要 **view controller** 具有一些特定的能力。比如，某个扩展可能需要 **view controller** 拥有一个被显示的 **activity indicator**，或者需要 **view controller** 上有某个特定的方法可用。我们可以在协议里确保这些能力。举个例子，我们可以共享处理键盘的代码，这些代码会在键盘显示或隐藏时对 **view** 进行缩放。如果我们使用了自动布局，那么我们可以指定我们需要一个可缩放的底部约束：

```
protocol ResizableContentView {  
    var resizableConstraint: NSLayoutConstraint { get }  
}
```

接下来，我们可以为每个实现了该协议的 **UIViewController** 添加扩展：

```
extension ResizableContentView where Self: UIViewController {  
    func addKeyboardObservers() {  
        // ...  
    }  
}
```

现在，任何一个实现了 **ResizableContentView** 的 **view controller** 同时也获得了 **addKeyboardObservers** 方法。我们可以在想要共享代码但又不想引入子类的其他情况下，使用相同的技术。

利用 Child View Controller 进行代码重用

Child view controller 是在 **view controller** 之间共享代码的另一种选项。比如，要是我们想要在文件夹 **view controller** 的下部显示一个小的播放器，我们可以在文件夹 **view controller** 上添加一个 **child view controller**，这样，播放器的逻辑也被包含了，而且不会弄乱文件夹 **view controller**。相比与在文件夹 **view controller** 中重复一遍相关代码，这种做法要容易得多，也更好维护。

如果我们有某个单个的 view controller，但是它包含两个完全不同的状态，我们也可以将它拆分为两个 view controller (每个 controller 管理一个状态)，并用一个容器 view controller 在这两个 child view controller 之间进行切换。比如，我们可以把 PlayViewController 拆分为两个独立的 view controller：一个负责显示“没有选中的录音”的文本，另一个显示录音。容器 view controller 可以按照状态的不同，在两者之间进行切换。这种方式有两个好处：首先，如果我们将标题 (和其他一些属性) 写为可配置的话，这个空白的 view controller 就可以被重用。其次，PlayViewController 不再需要处理当录音为 nil 时的情况；我们只需要在我们拥有录音的时候创建并展示它就可以了。

提取对象

许多大的 view controller 都有很多角色和职责。虽然想要发现某个地方可以重构并非一件易事，但是通常一个角色或者职责都可以被提取为一个单独的对象。我们发现区分 Apple 所定义的协调 controller (coordinating controller) 和调解 controller (mediating controller) 会很有意义。一个协调 controller 是 app 特定的，而且一般来说是无法重用的 (比如，几乎所有的 view controller 都是协调 controller)。

调解 controller 则是一个可重用的 controller 对象，通过配置，它可以被用来执行特定的任务。比如 AppKit 框架提供了像是 NSArrayController 或者 NSTreeController 这样的类。在 iOS 中，我们可以构建出类似的部件。通常，用来遵守某个协议的代码 (比如文件夹 view controller 中遵守 UITableViewDataSource 的部分)，比较适合被提取为调解 controller。将这些遵守协议的代码抽离到单独的对象中，可以有效减少 view controller 的代码量。首先，我们可以将文件夹 view controller 中 table view 的 data source 不加修改地提取出来：

```
class FolderViewDataSource: NSObject, UITableViewDataSource {
    var folder: Folder

    init(_ folder: Folder) {
        self.folder = folder
    }

    func numberOfSections(in tableView: UITableView) -> Int {
        return 1
    }

    func tableView(_ tableView: UITableView, numberOfRowsInSectionSection section: Int)
        -> Int
    {
```

```

    return folder.contents.count
}

func tableView(_ tableView: UITableView, cellForRowAt indexPath: IndexPath)
-> UITableViewCell
{
    let item = folder.contents[indexPath.row]
    let identifier = item.is Recording ? "RecordingCell" : "FolderCell"
    let cell = tableView.dequeueReusableCell(withIdentifier: identifier,
        for: indexPath)
    cell.textLabel!.text = "\\((item.is Recording) ? "🎤" : "📁") \ \(item.name)"
    return cell
}

func tableView(_ tableView: UITableView,
    commit editingStyle: UITableViewCellEditingStyle,
    forRowAt indexPath: IndexPath)
{
    folder.remove(folder.contents[indexPath.row])
}

func tableView(_ tableView: UITableView, canEditRowAt indexPath: IndexPath)
-> Bool
{
    return true
}
}

```

然后，在文件夹 controller 本身中，我们将 table view 的 data source 设置为这个新的数据源；

```

lazy var dataSource = FolderViewDataSource(folder)

override func viewDidLoad() {
    super.viewDidLoad()
    tableView.dataSource = dataSource
    // ...
}

```

我们还需要对 view controller 的 folder 进行观察，并在它变化时改变 data source 中的 folder。以这些额外的通讯作为代价，我们可以将 view controller 分为两个部分。在这个例子中，这种分离没有带来太大的开销，但是当两个部件紧密耦合在一起，并且需要对很多状态进行通讯和共享的时候，所带来的开销可能会非常大，反而使得事情变得更加复杂。

如果我们有多个类似的对象，我们可能能够将它们泛型化。举例来说，在上面的 FolderViewDataSource 中，我们可以将其中的存储属性从 Folder 变为 [Item] (一个 Item 可以是一个文件夹，也可以是一条录音)。很自然地，下一步就是让 data source 对 Element 类型进行泛型抽象，并将 Item 相关的逻辑移除出去。这就是说，表格单元格的配置 (通过 configure 参数) 和删除逻辑 (通过 remove 参数) 现在是从外面传递进来的：

```
class ArrayDataSource<Element>: NSObject, UITableViewDataSource {
    // ...
    init(_ contents: [Element],
         identifier: @escaping (Element) -> String,
         remove: @escaping (_ at: Int) -> (),
         configure: @escaping (Element, UITableViewCell) -> ()) {
        // ...
    }
    // ...
}
```

想要以我们的文件夹和录音为它进行配置，我们需要下面的代码：

```
ArrayDataSource(folder.contents,
    identifier: { $0 is Recording ? "RecordingCell" : "FolderCell" },
    remove: { [weak self] index in
        guard let folder = self?.folder else { return }
        folder.remove(folder.contents[index])
    }, configure: { item, cell in
        cell.textLabel!.text = "\\((item is Recording) ? "🎧" : "📁") \ \(item.name)"
    })
```

在我们的小型示例 app 中，将代码泛型化并没有给我们带来太多收益。但是在更大一些的 app 中，这种技术可以减少重复代码，让我们能以类型安全的方式进行 cell 重用，并使 view controller 更加简单。

简化 View 配置代码

如果 view controller 需要构建和更新非常多的 view，那么将这部分 view 配置的代码提取出来会很有帮助。特别是对于那些不需要双向通讯的，“设置完后就可以不再关心”的情况，这样做能够简化我们的 view controller。举例来说，当我们有一个很复杂的 tableView(_:cellForRowAtIndexPath:) 时，我们可以将部分代码从 view controller 中移出来：

```
override func tableView(_ tableView: UITableView,
    cellForRowAt indexPath: IndexPath) -> UITableViewCell
{
    let item = folder.contents[indexPath.row]
    let cell = tableView.dequeueReusableCell(withIdentifier: identifier,
        for: indexPath)
    cell.configure(for: item) // 被提取的代码
    return cell
}
```

我们现在已经把所有的逻辑移动到 UITableViewCell 或者一个它的子类中了：

```
extension UITableViewCell {
    func configure(for item: Item) {
        titleLabel!.text = "\(item is Recording) ? "🎧" : "📁" \(item.name)"
    }
}
```

在我们的例子中，将 cell 配置的代码移出来，只是让 view controller 稍微简单了一点点，因为原来的代码也只是一行而已。但是，如果 cell 配置的代码有很多行，就很值得把它们移出去了。我们还可以使用这种模式在不同的 view controller 之间共享配置和布局代码。另外一个好处是，很容易看到，configure(for:) 现在不依赖于 view controller 的任何状态，所有的状态都是通过参数传递进去的，这样一来，cell 就很容易测试了。

总结

MVC 是我们在本书中所讨论的最简单，也是最常用的模式。本书中所展示的其他所有的 app 设计模式，或多或少都是对惯例的破坏。除非你确实知道你想要为你的项目选择一个不那么通俗的路径，否则你可能还是应该使用 MVC 来开始你的项目。

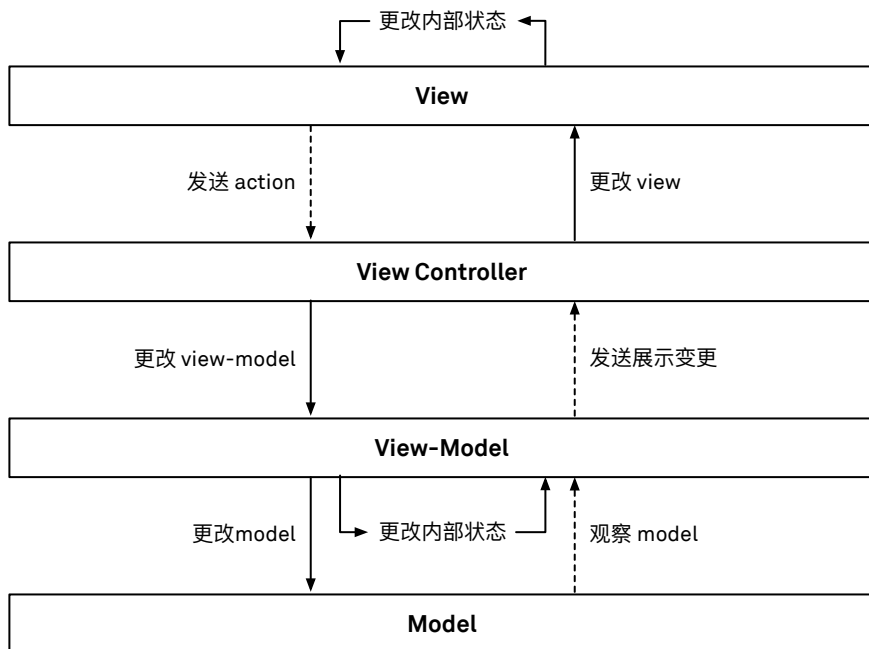
在一个程序员表达对某个不同架构模式的拥护时，他们有时候会贬低 MVC。对于 MVC 的负面评论包括：无法测试的 view controller，view controller 会增长得过大而且无法控制，数据依赖难以管理等。但是，在阅读本章后，我们希望你能意识到，虽然 MVC 确实有它自己的挑战，但是写出清晰简洁，并包含完整测试以及明确对数据依赖进行管理的 view controller，是完全可能的。

当然啦，有很多其他方式，可以让我们在 app 设计时所面临的挑战发生改变。在接下来的章节里，我们会讨论一系列替代的设计模式，它们中的每一个都从不同的方面解决了 app 设计中的某些问题。所有这些模式都包含了很多想法，你可以将这些想法运用在 MVC 程序中，来务实地解决特定的问题。你也可以通过它们来创造你自己的混合模式或者编程风格。

Model-View- ViewModel+协 调器 (MVVM-C)

4

Model-View-ViewModel (MVVM) 是一种基于 MVC 进行改进的模式，它将所有 model 相关的任务 (包括更新 model，观察变更，将 model 变形为可以显示的形式等) 从 controller 层抽离出来，放到新的叫做 view-model 的一层对象中。在通常的 iOS 实现中，view-model 位于 model 和 controller 之间：



和所有好的模式一样，MVVM 所做的不仅仅是把代码移动到新的地方。加入一层新的 view-model 层的目的是双重的：

1. 鼓励将 model 和 view 之间的关系构建为一系列的变形管道。
2. 提供一套独立于 app 框架的接口，但是它在相当程度上代表了 view 应该展示的状态。

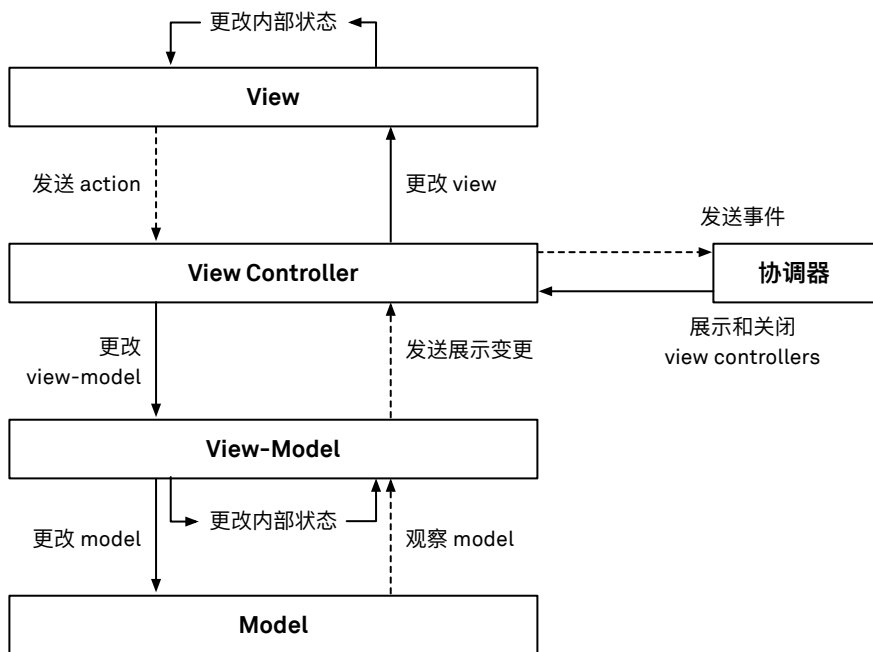
两者结合起来，对 MVC 中两个最大的被人诟病的地方进行了修正。第一项通过把 model 相关的观察和变形从 controller 层移除出去，减少了 controller 所需要承担的责任。第二项为场景的 view state 提供了一套干净的接口，让它可以独立于 app 框架进行测试，而不需要使用 MVC 的集成测试。

为了保持 view 与 view-model 的同步，MVVM 强制使用某种形式的绑定，也就是说，需要一种保证一个对象上的属性与另一个对象上的属性同步的方式。Controller 负责构建这些绑定，将 view-model 所暴露的属性和场景中 view-model 所代表的 view 上的属性关联起来。

我们会使用响应式编程的方式来实现绑定，因为这是 iOS 上的 MVVM 最常用的方式。不过，Cocoa 项目中并没有默认包含响应式编程的方式，而且如果你不熟悉响应式编程的话，响应式的代码阅读起来也会有些困难。请阅读下面 [“响应式编程快速入门”](#) 的部分来了解相关的术语和概念。

我们选择使用响应式编程的方式为示例 app 实现 MVVM，是因为我们相信这种方式能够最好地展示 MVVM 的优势。另外，我们认为无论你最后会不会在你的代码库中使用响应式编程，对它进行学习都是大有裨益的。需要一提的是，我们在本章的后面部分也包括了一节内容，来向你展示在 [不依赖响应式编程](#) 的前提下，或者甚至是只基于 [Foundation 的键值观察 \(KVO\) 的绑定](#) 的基础上，应当如何使用 MVVM 架构。

除此之外，我们示例 app 中的 MVVM 版本还引入了一个协调器 (coordinator) 部件。在 MVVM 中，协调器并非是强制需要的部件，但是它可以帮助 controller 从两个额外的职责中解放出来：controller 可以不再需要管理其他 controller 的展示，它也不再需要去协调 model 数据和 controller 之间的通讯。协调器负责维护 controller 的层级，这样一来，controller 和 view-model 就只需要专注处理它们在特定场景的使用了。Model-View-ViewModel+协调器 (MVVM-C) 的框图如下：



在我们的示例 app 中，协调器是 controller 的 delegate，controller 会把导航行为直接转发给协调器。有人选择把导航行为先转发给 view-model，然后让协调器去观察 view-model，以获取导航事件。如果导航事件是依赖于当前 view 的状态或者 model 数据的话，这么做会有帮助。将导航事件从 controller 中移动到 view-model 里，也会让两者都更容易测试。不过，将导航事件扔到 view-model 里去转一圈并不能让我们的示例 app 获得实际的好处，所以我们选择了更简单的解决方式。

探索实现

MVVM 模式可以被看作是 MVC 的精密版本。我们的实现添加了两部部件，来移除一些 controller 的职责：协调器负责导航功能，view-model 负责了原本在 controller 中的绝大部分逻辑。尽管 MVVM 和 MVC 这两种模式有相似之处，但 MVVM 的实现看上去很多地方还是和 MVC 有根本的不同。

在 MVVM 中 model 和 view 之间的通讯，不像 MVC 那样是单一的观察者模式，而是一组重新构建的数据管道，这是让代码风格看上去不一样的一个主要原因。我们会使用响应式编程来实

现这个数据管道。响应式编程通过一系列的变形阶段来构建数据管道，这看上去和 Swift 基本的控制流语句 (循环，条件，方法调用等) 所构成的构建逻辑截然不同。

为了减少对响应式编程不熟悉的读者带来的困惑，在深入研究 MVVM 中对架构的构建，model 变更和 view state 变更的处理方式之前，我们会先总结一下响应式编程中的重要概念。

响应式编程快速入门

响应式编程是一种用来描述数据源和数据消费端之间数据流动的模式，它将这种流动描绘为一个变形管道。数据被看作一个在管道中流动的序列，我们使用像是 map，filter 和 flatMap 这样的和 Swift 序列或者集合中函数名字相近的函数，来对数据进行变形。

让我们来快速地看一个例子。想象我们有一个 model，上面有一个字符串可选值属性。我们想要对该属性进行观察，并将观察到的值设置到一个 label 上。假设这个属性被声明为 dynamic，我们可以在 controller 的 viewDidLoad 方法中对它使用 KVO 进行观察：

```
var modelObject: ModelObject!
var obs: NSKeyValueObservation? = nil

override func viewDidLoad() {
    super.viewDidLoad()

    obs = modelObject.observe(\ModelObject.value) { [unowned self] obj, change in
        if case let value?? = change.newValue {
            self.textLabel.string = "Selected value is: \(value)"
        } else {
            self.textLabel.string = "No value selected"
        }
    }
}
```

除了需要将 change.newValue 做双层可选值解包以外，所使用的基本是标准的 Cocoa 编程；我们观察该值，当它改变时，我们更新文本 label。

使用 RxSwift 来进行响应式编程的话，我们可以这样来写：

```
var modelObject: ModelObject!
var disposeBag = DisposeBag()
```

```

override func viewDidLoad() {
    super.viewDidLoad()

    modelObject.valueObservable.map { possibleValue -> String in
        if let value = possibleValue {
            return "Selected value is: \(value)"
        } else {
            return "No value selected"
        }
    }.bind(to: self.textLabel.rx.text).disposed(by: disposeBag)
}

```

这里的不同可能最初看起来只是关于美感：

- 我们没有直接观察 `value` 属性，而是读取了一个叫做 `valueObservable` 的属性。
- 我们没有将所有工作在一个单独的回调中完成，而是在中间使用 `map` 做了一次只和自身相关的纯数据变形。
- 我们没有直接设置 `textLabel.text`，而是在链式调用的末端通过 `bind(to:)` 的调用将其绑定在 `label` 的文本上。

为什么这很重要？

相比于在很多地方设置 `textLabel.text` 的值，现在这个 `textLabel` 只会在最后被引用一次。响应式编程让我们从目的地 - 也就是数据的**订阅者** - 开始，一路通过数据变形进行回溯，直到到达原始的数据依赖 - **可观察量** (observable)。通过这么做，数据管道的可观察量，数据变形以及订阅者三者得以分离。

数据变形的部分是响应式编程所能带来的最大优势，但同时它也是学习曲线最为陡峭的部分。也许你在 Swift 的序列或者可选值中已经使用过 `map` 了，在 RxSwift 的版本中，它也以相似的方式工作。RxSwift 中其他一些与 Sequence 等效的函数包括 `filter` (在两者中相同)，`concat` (和 `append(contentsOf:)` 类似)，`skip` 与 `skipWhile` (和 `dropFirst` 与 `dropWhile` 类似) 以及 `take` 与 `takeWhile` (和 `prefix` 与 `prefixWhile` 类似)。

值得讨论一下的是 `flatMapLatest` 这个变形。该变形观察数据源，在数据源每次发出一个值的时候，它使用该值构建，开始，或者选择一个新的可观察量。可能看起来有点让人困惑，不过这个变形可以让我们基于第一个可观察量发出的状态，来订阅第二个可观察量。

另外，RxSwift 中还有一些类型也需要注意：

- **Observable** 是一系列值的串流，我们可以对它们进行变形，订阅，或者将它们绑定到 UI 元素上去。
- **PublishSubject** 是 **Observable** 的一种，我们可以将值**发送**给它，这些值会被发给观察者。
- **ReplaySubject** 和 **PublishSubject** 类似，不过我们可以在没有任何观察者连接上它时就进行值的发送，新的观察者会接收到暂存在“重放”缓冲区上的之前被发送的值。
- **Variable** 是一个对于可设置的值的封装，它能提供一个 **Observable** 属性，这样我们就可以在值每次被设置时观察到它。
- **Disposable** 和 **DisposeBag** 分别用来控制一个或多个订阅的生命周期。当一个 **Disposable** 被销毁或者手动丢弃时，订阅行为就将结束，另外该订阅的所有的可观察量组成部分也将被释放。

要吸收的信息已经很多了。可能我们最好进一步来看看 MVVM。

构建

MVVM 构建的方式和 MVC 的模式很相似：controller 层充分了解程序的结构，它使用这些认知来对所有部件进行构建和连接。相比起 MVC，主要有三个不同：

1. 必须创建 **view-model**。
2. 必须建立起 **view-model** 和 **view** 之间的绑定。
3. **Model** 由 **view-model** 拥有，而不是由 **controller** 所拥有。

要演示第一个不同，我们选择在每个 controller 中构建一个默认的 **view-model**：

```
class FolderViewController: UITableViewController {  
    let viewModel = FolderViewModel()  
    // ...  
}
```

在使用 storyboard 和 segue 时，这个特殊的安排是一种简单、低阻力的方式，可以让我们不需要在 segue 执行期间或者 controller 构建后立即去设置 **view-model**。我们没有在

view-model 生成时用一个 model 对象的引用来对它进行配置，所以我们需要在稍后的时间点进行 model 对象的设置。

在另一种安排中，我们可以在 controller 中用 nil 可选值作为初始值来声明 view-model，把对 view-model 的完整配置放到后面的时间点来进行。我们没有采取这种做法，是因为在录音 app 中它不仅没有解决任何数据传输的问题，而且还在不需要可选类型的地方引入了可选值。不过，如果我们忽略掉 storyboard，而采用手动的方式来初始化 controller 的话，就可以将 view-model 作为参数传递给 controller 了，这样我们就可以同时获得两种安排中最好的一面。

将 View 与初始数据连接

在我们使用的 MVVM-C 模式中，将 model 对象设置给 view-model 的任务是由协调器承担的。协调器本身是一个高层级的 controller 对象，它由 app delegate 在启动的最终阶段进行创建，所以它需要确保在构建初始的 controller 时，数据已经准备妥当：

@UIApplicationMain

class AppDelegate:

UIResponder, UIApplicationDelegate, UISplitViewControllerDelegate {

var coordinator: Coordinator? = nil

func application(_ application: UIApplication,
didFinishLaunchingWithOptions launchOptions:

[UIApplicationLaunchOptionsKey: Any]?)

-> Bool

{

let splitViewController = window!.rootViewController as! UISplitViewController

splitViewController.delegate = self

splitViewController.preferredDisplayMode = .allVisible

coordinator = Coordinator(splitViewController)

return true

}

// ...

}

在协调器自己的初始化方法中，它确保初始数据正确地提供给顶层的 view-model：

final class Coordinator {

```

init(_ splitView: UISplitViewController) {
    // ...
    let folderVC = folderNavigationController.viewControllers.first
        as! FolderViewController
    folderVC.delegate = self
    folderVC.viewModel.folder.value = Store.shared.rootFolder
    // ...
}
// ...
}

```

folderVC.viewModel.folder.value = Store.shared.rootFolder 一行最为重要，在这里，根 FolderViewModel 接收到了它的数据。这也是程序中第一次使用 RxSwift 框架的代码，文件夹 view-model 中的 folder 属性是一个 RxSwift 中的 Variable 值：

```

class FolderViewModel {
    let folder: Variable<Folder>
    // ...
}

```

改变一个 Variable 的值会产生两个效果：首先，文件夹 view-model 的 folder.value 属性会返回我们新设置的值；其次，这个变量所有的观察者都将会收到新设置的值。

想要清楚 view 从根本上是如何接收数据的，我们需要探究文件夹 view-model 的实现。为了观察 folder 变量，view-model 在初始化时创建了依赖该变量的可观察值 folderUntilDeleted：

```

// FolderViewModel
private let folderUntilDeleted: Observable<Folder?>

init(initialFolder: Folder = Store.shared.rootFolder) {
    folder = Variable(initialFolder)
    folderUntilDeleted = folder.asObservable()
    // 每次文件夹变更时
    .flatMapLatest { currentFolder in
        // 通过发出该值开始
        Observable.just(currentFolder)
        // 每次非删除的变更发生时，重新发出文件夹
        .concat(currentFolder.changeObservable.map { _ in currentFolder })
        // 当删除发生时，停止
    }
}

```

```

        .takeUntil(currentFolder.deletedObservable)
        // 在删除后，将当前文件夹设置回 `nil`
        .concat(Observable.just(nil))
    }.share(replay: 1)
}

```

我们为每一行都添加了注释，所以如果你对响应式编程不熟悉的话，可以通过注释来一步步理解代码的大概意思。响应式编程通过序列自身的变形来构建逻辑。这些变形将大量的逻辑以高效的语法进行编码，但是这需要对常见的响应式编程变形进行学习，也使得阅读响应式编程代码的学习曲线更加陡峭。

上面代码的主旨是：我们将 `folder` 这个可观察量，与其他由 `model` 驱动的，可能影响我们 `view` 的逻辑的可观察量，进行了合并。得到的结果是一个新的可观察量 `folderUntilDeleted`，它会在底层文件夹对象发生变化时正确更新，并且在底层文件夹对象被从 `store` 中删除时将自己设置为 `nil`。

`Controller` 并不直接对 `folderUntilDeleted` 可观察量进行观察。`View-model` 的目的是将所有数据转变为可以直接被绑定到 `view` 上的格式。在这里，意味着我们需要从 `folderUntilDeleted` 可观察量发送的文件夹对象中，提取出所需要的文本和其他属性。

比如，在 `FolderViewController` 顶部导航栏上要显示的标题就是在 `FolderViewModel` 中被准备好的。任何时候只要 `folderUntilDeleted` 可观察量发送一个新的值，这个可观察量也会发送一个新的值：

```

// FolderViewModel
var navigationTitle: Observable<String> {
    return folderUntilDeleted.map { folder in
        guard let f = folder else { return "" }
        return f.parent == nil ? .recordings : f.name
    }
}

```

`navigationTitle` 属性的可观察量输出会被绑定到 `controller` 的标题上，`FolderViewController` 中 `viewDidLoad` 方法里的下面的代码做的就是这件事：

```

class FolderViewController: UITableViewController {
    override func viewDidLoad() {
        super.viewDidLoad()
        viewModel.navigationTitle.bind(to: rx.title).disposed(by: disposeBag)
    }
}

```

```

    // ...
}
}

```

这样，我们就有了一个数据的完整管道了：

1. 协调器为每个 controller 的 view-model 设置初始的 model 对象。
2. View-model 将设定值和其他 model 数据及观察量进行合并。
3. View-model 将数据变形为 view 所需要的精确的格式。
4. Controller 使用 bind(to:) 来将准备好的值绑定到各个 view 上去。

实现状态恢复数据

在状态恢复的方法上，MVVM 中为各个 controller 所存储的数据来源于 view-model，而非像 MVC 那样来源于 controller 本身，除此之外，两者所使用的策略大抵相同。所以，在文件夹 controller 上的 encodeRestorableState 和 decodeRestorableState 的实现和 MVC 版本几乎是一样的：

```

// FolderViewController
override func encodeRestorableState(with coder: NSCoder) {
    super.encodeRestorableState(with: coder)
    coder.encode(viewModel.folder.value.uuidPath, forKey: .uuidPathKey)
}

override func decodeRestorableState(with coder: NSCoder) {
    super.decodeRestorableState(with: coder)
    if let uuidPath = coder.decodeObject(forKey: .uuidPathKey) as? [UUID],
        let folder = Store.shared.item(atUUIDPath: uuidPath) as? Folder
    {
        self.viewModel.folder.value = folder
    } else {
        if var controllers = navigationController?.viewControllers,
            let index = controllers.index(where: { $0 === self })
        {
            controllers.remove(at: index)
            navigationController?.viewControllers = controllers
        }
    }
}

```

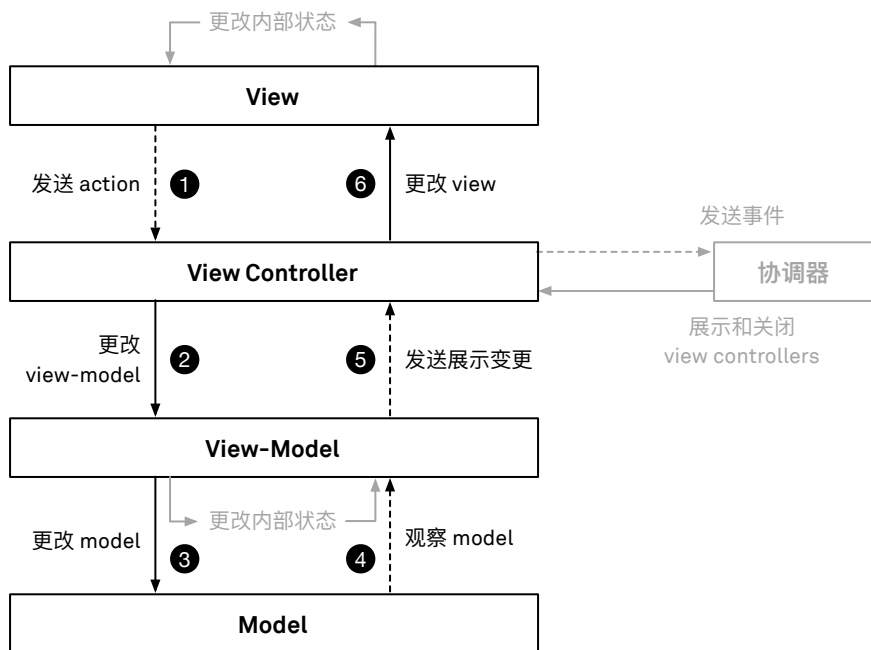


```
}  
}
```

代码中唯一的不同是使用了 `viewModel.folder.value` 而不是 `self.folder`。除开这一点，其他的代码和 MVC 中的完全一致。

更改 Model

MVVM-C 中，从 view 到 model，然后再返回 view 的事件回路和 MVC 中各层之间的路径相比，也很相似。不同在于，在 controller 和 model 之间插入了一层额外的用于协调的 view-model：



下面，我们将会看看从 table view 中删除一个文件夹，并将改动传递到 model 后，再传递回 view 上进行反映，所需要的的实现步骤。

步骤 1: Table View 发送 Action

在我们的 MVVM 实现中, table view 的数据源不是由 storyboard 设置的, 而是通过 RxSwift 的 table view 扩展来提供 table view 所需要的数据。因此, 通常对于数据源的 tableView(_:commit:forRowAt:) 方法调用现在是被封装在 table view 的 modelDeleted 可观察量中的一个事件。

步骤 2: Controller 改变 View-Model

在 MVVM 中, controller 不会自己去调用 model 层的内容。它会把这项工作外包给它的 view-model。在第一步中我们提到过, 我们可以通过 Rx 中 table view 上的 modelDeleted 可观察量获取 table view 删除的通知。我们可以定义这个可观察量, 并且调用 view-model 的 deleteItem 方法:

```
// FolderViewController
override func viewDidLoad() {
    // ...
    tableView.rx.modelDeleted(Item.self)
        .subscribe(onNext: { [unowned self] in
            self.viewModel.deleteItem($0)
        }).disposed(by: disposeBag)
}
```

步骤 3: View-Model 改变 Model

在 view-model 中, 这个变更被直接传递给 model 层:

```
// FolderViewModel
func deleteItem(_ item: Item) {
    folder.value.remove(item)
}
```

Folder 类的 remove 方法将会把指定的条目删除, 并在 store 上调用 save 方法来把变更进行持久化 (这和 MVC 版本是完全一样的):

```
// Folder
func remove(_ item: Item) {
    guard let index = contents.index(where: { $0 == item }) else { return }
}
```

```

        item.deleted()
        contents.remove(at: index)
        store?.save(item, userInfo: [
            Item.changeReasonKey: Item.removed,
            Item oldValueKey: index, Item.parentFolderKey: self
        ])
    }
}

```

保存变更将会触发一个 model 通知，我们会在下一步中使用到它。

步骤 4: View-Model 观察 Model 变更

我们在上面构建的部分做过概论，view-model 将观察它的 folder 变量的变更，并将这个变更和 model 通知一起合并到 folderUntilDeleted 中。每当文件夹的内容发生变化，view-model 都需要更新它所绑定的 table view 中的内容，并将它们显示出来。这是通过 folderContents 可观察量来完成的：

```

// FolderViewModel
var folderContents: Observable<[AnimatableSectionModel<Int, Item>]> {
    return folderUntilDeleted.map { folder in
        guard let f = folder else {
            return [AnimatableSectionModel(model: 0, items: [])]
        }
        return [AnimatableSectionModel(model: 0, items: f.contents)]
    }
}

```

folderContents 将当前文件夹的内容通过一种可以绑定到 table view 的方式发送出去，我们会在下一步中看到。

步骤 5 & 6: Controller 观察 view-model 并更新 view

在 viewDidLoad 中，controller 通过 RxSwift 的 table view 扩展 ([RxDataSources](#) 库中的部分)，将 view-model 的 folderContents 属性和 table view 的数据源绑定起来：

```

// FolderViewController
override func viewDidLoad() {
    // ...
}

```

```
viewModel.folderContents.bind(  
    to: tableView.rx.items(dataSource: dataSource)  
).disposed(by: disposeBag)  
}
```

这就是我们为了让 table view 保持和底层数据同步所需要进行的全部工作了。在底层，Rx 的 table view 扩展将负责全部像是 insertRows 和 deleteRows 这些 table view 的调用。

更改 View State

在 MVVM-C 中，有一部分用户界面的暂时状态是作为 view state 隐式存储在 view 和 controller 层级中的，而另一部分则是显式地由 view-model 进行表示。

理论上来说，view-model 应该是对特定的场景中的 view 的完整状态的表现。不过，在通常实践中，view-model 只负责那些会被 view 行为影响的 view state。举例来说，虽然当前的滚动位置显然是一个 view state，但是它一般不由 view-model 进行表现；它既不依赖于任何的 view-model 属性，也不是 view-model 所依赖的部分。这让 view-model 中所表达的 view state，等效于那些典型的 MVC 架构下的 controller 子类通过属性所保存的 view state。

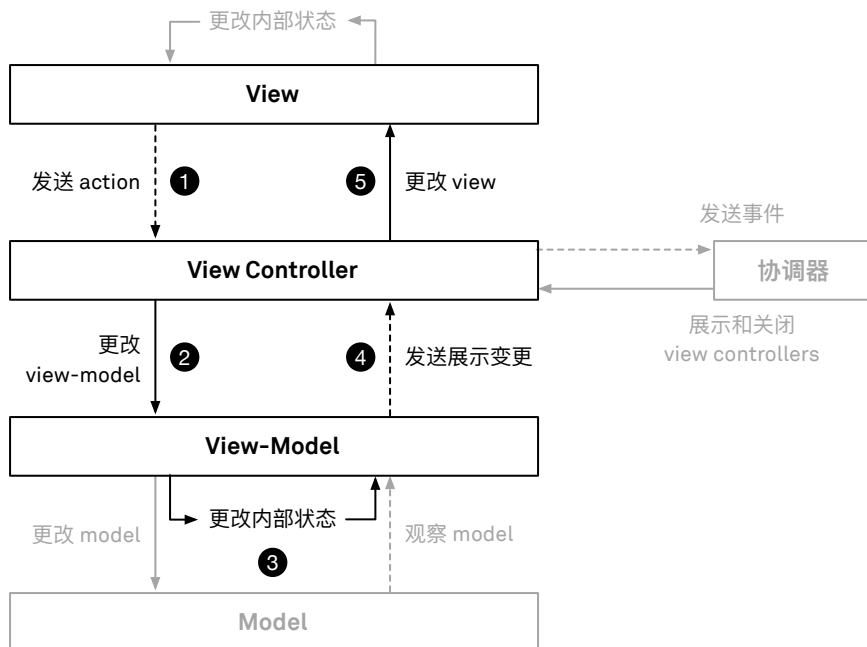
下面，我们将通过两个不同的例子来看看如何更改 view state：

1. 在按下播放按钮时，作为响应，改变它的标题。
2. 在子文件夹被选中时，将一个文件夹 controller 推入导航栈。

在第一个例子中，我们只是对一个已经存在的 view 的属性 (按钮的标题) 进行变更，而在第二个例子中，我们需要对 view 和 controller 层级进行变更。

示例 1：更新播放按钮

如果一个 view state 的变更不会影响到 controller 层级，那么就不需要使用到协调器。Controller 在接收到用户行为后将它转发给 view-model，然后响应式的绑定将必要的 view 更新从 view-model 传递到 view 中。这就是更新播放按钮标题的第一个例子中发生的数据流动：



步骤 1: 按钮向 Controller 发送 Action

第一步和 MVC 的示例 app 中做的一样：我们通过 Interface Builder 将播放按钮与 PlayViewController 的 play 方法连接起来，这样用户按下播放按钮时，play 将被调用。

步骤 2: 播放 Controller 改变 View-Model

play 方法中只有一个调用，那就是向 view-model 的 togglePlay 属性 (它是一个 PublishSubject) 发送一个新的事件：

```
// PlayViewController
@IBAction func play() {
    viewModel.togglePlay.onNext()
}
```

步骤 3: 播放 View-Model 改变它的内部状态

对 view-model 的 `togglePlay` 属性的 `onNext` 调用会将音频播放器的播放状态在播放和暂停之间切换。每当音频播放器的状态被改变时，就向 view-model 内部的 `playState` 可观察量发送一个新的值。从这个可观察量中，我们可以衍生出 `playButtonTitle` 可观察量，这个值映射了可以被设置在按钮标题上的可能的状态：

```
// PlayViewModel
var playButtonTitle: Observable<String> {
    return playState.map { s in
        switch s?.activity {
            case .playing?: return .pause
            case .paused?: return .resume
            default: return .play
        }
    }
}
```

步骤 4 & 5: 播放 Controller 观察 View-Model 并更新 View

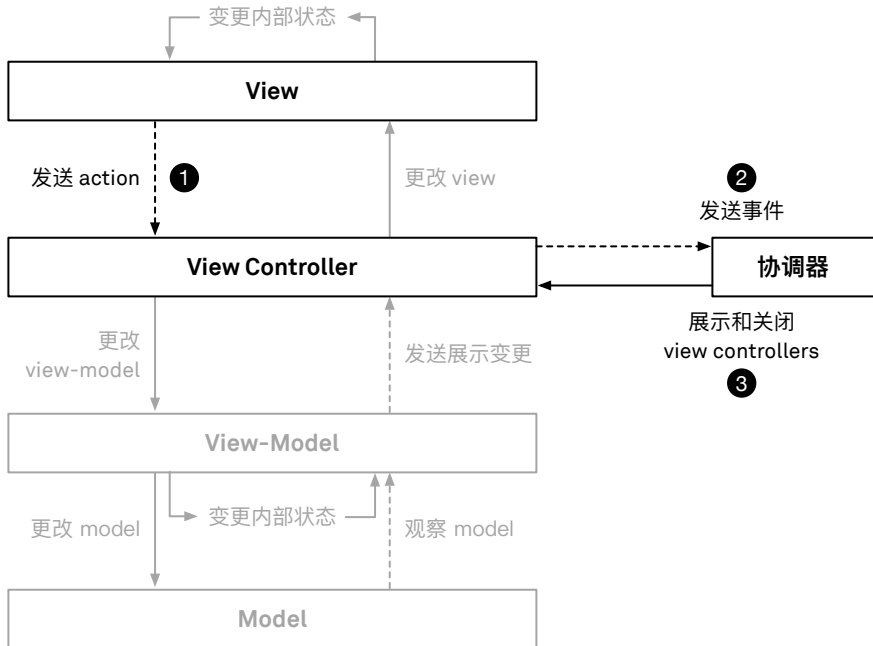
在 `PlayViewController` 的 `viewDidLoad` 中，我们对 view-model 的 `playButtonTitle` 可观察量和播放按钮标题进行了响应式绑定：

```
// PlayViewController
override func viewDidLoad() {
    // ...
    viewModel.playButtonTitle.bind(to: playButton.rx.title(for: .normal))
        .disposed(by: disposeBag)
}
```

这个绑定将会观察 view-model 的播放按钮标题属性，并将它设置到实际的按钮 view 上去。

示例 2: 推入文件夹 View Controller

当我们想要变更 controller 的层级时，就需要使用到协调器了，下面这个例子中就会说明这一点。Controller 接收到用户行为，然后将更新 controller 层级的工作委托给协调器来完成：



步骤 1 & 2: 文件夹 controller 接收选择行为，并调用协调器

在文件夹 controller 的 `viewDidLoad` 中，我们使用 RxSwift 的 `table view` 扩展订阅了 `table view` 选择的事件。当我们接收到事件时，我们将它转发给文件夹 controller 的 `delegate` (这个 `delegate` 就是协调器)：

```
// FolderViewController
override func viewDidLoad() {
    // ...
    tableView.rx.modelSelected(Item.self)
        .subscribe(onNext: { [unowned self] in
            self.delegate?.didSelect($0)
        }).disposed(by: disposeBag)
}
```

步骤 3: 协调器将新的文件夹 controller 推入

协调器的 `didSelect` 方法会检查被选择的条目是不是一个文件夹。如果是，那么它将从 storyboard 中初始化一个新的文件夹 controller。并将其推入导航栈中：

```
extension Coordinator: FolderViewControllerDelegate {  
    func didSelect(_ item: Item) {  
        switch item {  
        case let folder as Folder:  
            let folderVC = storyboard.instantiateFolderViewController(  
                with: folder, delegate: self)  
            folderNavigationController.pushViewController(folderVC,  
                animated: true)  
            // ...  
        }  
    }  
}
```

和 MVC 类似，UIView/UIViewController 层级依然隐式地代表了 view state。但是，controller 通常需要处理的两项工作现在被拿出去了：现在 view-model 会负责将 model 值转换为 view 所需要的数据，而协调器则管理了 controller 的展示流程。

测试

在 MVC 中，我们介绍了**集成测试**的方式。考虑到 view controller，view state 和 view 之间牢固的集成关系，这是实践中唯一可行的测试方式。

在集成测试中，我们不仅仅只测试一个部件的内部逻辑，同时我们也测试了它与其他部件的连接，有时候我们甚至还测试了那些相连的部件。由于集成测试包含了关于框架 (比如 UIKit) 的全面的知识，所以它不是独立的，这在测试失败时导致我们很难确定其原因。另外，使用测试优先的方式来写集成测试并不容易，因为这要求对于程序中所有部件的工作方式都有所了解。

MVVM 通过定义 view-model 这一清晰的接口，明确地将它从 view 层解耦出来，从而改变了集成的特性。相比于测试整个 controller 和所有相连的 view，通过隔离的方式测试 view-model 层，我们可以最小化在测试时必须同时连接的部件的数量。

然而，MVVM 的接口测试和 MVC 的集成测试相比，不论在范围、结构还是细节上，都有很大不同。总体来说，集成测试更偏向功能，但是它们难以编写和维护。另外，由于集成测试中构建

测试场景十分困难，所以典型的集成测试会在单个测试用例中测试场景中的每一个属性，而不是将它们缩小为每个函数对应一个测试的形式。

在 MVVM 里，我们通常只使用接口测试来测试 view-model，而 controller 和 view 则依赖与 Xcode 的 UI 测试或者人工测试。也就是说，controller 中的逻辑不会被接口测试覆盖到。另外，如果我们误解了 UIKit 中 (或者其他在 view 或者 controller 中使用的) 函数的用法，它们也不会被接口测试所覆盖。

所以说，MVVM 通常要求 controller 必须非常简单 (甚至简单到无需考虑)。另外，controller 必须尽可能地使用库提供的绑定方法。在这样的规则的保证下，理想情况中我们就不需要测试 controller 了，因为它没有包含我们自己的任何逻辑。

究竟应该在 controller 中留存多少逻辑，根本上来说是掌握在程序员手上的。RxSwift (或者其他响应式编程框架) 在 iOS 的 MVVM 架构实现中非常流行，是因为它们都含有针对平台的绑定库 (比如，RxCocoa 协助 RxSwift，提供 UIKit 的绑定)。这些绑定库可以将很多代码从 controller 中移除出去，从让程序更接近于理想的 MVVM。

配置测试

View-model 的接口测试的配置中对 store 的构建，和 MVC 的集成测试中所做的一样，不过这次不需要构建 view 树：

```
// FolderViewControllerTests
override func setUp() {
    super.setUp()

    (store, childFolder1, childFolder2) = constructTestingStore()

    viewModel = FolderViewModel(initialFolder: childFolder1)

    viewModel.folderContents.subscribe(onNext: { [weak self] in
        self?.contentsObserved.append($0)
    }).disposed(by: disposeBag)
    viewModel.navigationTitle.subscribe(onNext: { [weak self] in
        self?.titleObserved.append($0)
    }).disposed(by: disposeBag)
}
```

上面的代码创建了测试用的 `store`，包括了一些对文件夹的引用；我们还为其中一个被包含的文件夹创建了 `view-model`；然后立即订阅了 `view-model` 上两个常用的可观察量。这些配置代码展示了接口测试中的经典模式：构建输入，构建用来传递输入的接口，然后从接口中读取结果。

展示测试

对可观察量的测试也遵循通用的模式：对 `view-model` 上所暴露的每一个可观察量，对初始值进行测试，然后执行操作，并测试后续的条件（还可以进行更多的操作并测试更多的条件）。对于导航栏标题的测试就是这种模式的一个例子：

```
// FolderViewControllerTests
func testNavigationTitle() {
    // 测试初始条件
    guard titleObserved.count == 1 else { XCTFail(); return }
    XCTAssertEqual(titleObserved[0], "Child 1")

    // 执行操作
    childFolder1.setName("Another name")

    // 测试后续条件
    XCTAssertEqual(titleObserved.count, 2)
    XCTAssertEqual(titleObserved[1], "Another name")
    // ... 发送更多的操作
}
```

作为比较，MVC 的集成测试并没有针对导航栏标题的测试用例，对于导航栏标题的测试是作为 `testRootFolderStartupConfiguration` 和 `testChildFolderConfigurationAndLayout` 的一部分来进行的，这些测试所针对的分别是两种文件夹 `view` 上的所有显示属性。

行为测试

仅存的另外一种测试是 `view-model` 行为测试。这类测试和可观察量测试的模式很相似：测试初始条件，执行操作，然后测试接下来的条件。对文件夹的删除的测试例子如下：

```
// FolderViewControllerTests
func testFolderDelete() {
    // 测试初始条件
```

```
XCTAssertEqual(contentsObserved.count, 1)

// 执行操作
viewModel.deleteItem(childFolder2)

// 测试后续条件
XCTAssertEqual(contentsObserved.count, 2)
let sections = contentsObserved[1]
XCTAssertEqual(sections.count, 1)
XCTAssertEqual(sections[0].items.first?.uuid, uuid4)
}
```

除开这里的操作是直接作用在 view-model 上，而不是在 store 上以外，这里的测试的结构和前面的测试是一样的。

接口测试 vs. 集成测试

在 MVC 的集成测试中，大部分测试都需要根据被测试的情景进行调整，这让它们形态迥异，而相比起来，view-model 的接口测试就更加统一了。View 集成测试要求关于 view 层级工作方式，以及处理可能异步发生的 view 效果的全面的知识。不过，view-model 测试只需要关于我们自己的 view-model 和它的接口的知识，而且很少需要进行异步的测试。

由于这些原因，相比于 MVC 的集成测试，view-model 接口测试被广泛选择。不过，集成测试要比接口测试的覆盖率要高。比如，在集成测试中，我们不仅可以验证 view-model 被正确绑定到 view 上，还可以验证 view 行为被正确发送到 view-model 或者 model 层。类似地，在集成测试中，我们可以测试那些不依赖 view-model 的 controller 行为 (例如：view 布局或者与其他部件的交互)。

View-model 和 view 之间的空隙，让 app 集成 Xcode 的 UI tests 的需求变得更加迫切。但是 Xcode UI tests 和代码层面的集成测试有大量的重合，如果你已经集成了完整的测试，UI tests 可能就不那么必要了。不过，如果你依赖于 view-model 的接口测试来测试你的 app 逻辑的话，包含 UI tests 的迫切程度就会高得多。

讨论

初步印象来说，因为 MVVM-C 加入了额外的一层来进行管理，看起来是比 Cocoa MVC 模式更加复杂。不过，在实现的层级，如果你能够始终如一地贯彻这个模式，代码会变得更简单一些。

啊，这里说的简单并不意味着容易，只有当你对常见的响应式代码变形熟悉以后，才不会对书写代码感到无从下手，才不会对调试问题感到懊恼沮丧。不过，从令人高兴的一面来说，精心设计的数据管道通常不容易产生错误，在长期来看维护也更容易一些。

MVVM 通过将 model 观察的代码以及其他显示和交互逻辑移动到围绕着数据流构建的隔离的类中，解决了 MVC controller 里不规则的状态交互所带来的有关问题。因为这是 MVC 中最显著的问题，而且会随着 Cocoa controller 的增大而恶化，这个变化在很大程度上缓解了 MVC 中 controller 肥大的问题。但是还有其他一些因素会使得 controller (以及 view-model) 变大，所以为了可持续发展，重构依然还是有需要的。

MVC 中另一个常见问题是，由于观察者模式没有被严格执行，所导致的 model 和 view 之间不同步。View-model 属性和 view 属性之间的响应式绑定解决了这个问题。绑定之所以能解决这个问题，是因为 view 上初始值的设定和它们接下来的更新都经由统一的代码路径完成。

不过引入响应式编程也不是没有缺点的：它是我们理解和写出一个像录音 app 这样的 MVVM app 的最大的障碍。响应式编程框架有着陡峭的学习曲线，而且你需要花上一段时间才能调整到这种编程风格。虽然响应式编程在概念上来说十分优雅，但是由于响应式编程框架依赖高度抽象的变形以及大量的类型，对它们的误用可能导致你的代码无法被人类理解。

较少响应式编程的 MVVM

响应式编程技术有它自己的权衡，有人对它趋之若鹜，但是也有人对它避之不及。对一些程序员来说，MVVM 对这项技术的依赖成为了使用该模式的一种遏制因素。所以，在本节中我们会看看如何在少用或者不用响应式编程的前提下，使用 MVVM 模式。

在原本的 MVVM-C 实现中，我们在 view-model 中使用响应式编程，将它用来把 view-model 的接口和 view 绑定在一起。现在我们会探索两种变种：首先我们会看一看如何不使用任何内部响应式编程来实现 view-model，然后我们会讨论一种响应式绑定的替代方式，来在 view-model 变更是更新 view。

脱离响应式编程的 View-Model 实现

对于这种变种，我们会对文件夹 view-model 进行重构，让它们在不使用响应式编程时也能够正常工作。我们想要保持我们之前已有的相同的 view-model 接口，包括暴露给导航栏标题和 table view 的内容的可观察量。这允许我们在 controller 中保持使用响应式绑定来将 view-model 的输出绑定到对应的 view 上。

在最初的实现里，暴露的可观察量是依据其他的内部可观察量进行实现的。比如，`navigationTitle` 可观察量是通过 `folderUntilDeleted` 可观察量进行映射来创建的：

```
// FolderViewModel
var navigationTitle: Observable<String> {
    return folderUntilDeleted.map { folder in
        // ...
    }
}
```

因为我们想要摆脱像 `folderUntilDeleted` 这样的内部可观察量，我们必须以不同的方式创建可观察量。对于导航栏标题，我们创建一个私有的 `ReplaySubject`，通过它来传递新的值，并且将它作为可观察量暴露给外部世界：

```
// FolderViewModel
var navigationTitle: Observable<String> { return navigationTitleSubject }
private let navigationTitleSubject = ReplaySubject<String>.create(bufferSize: 1)
```

用来为 `table view` 提供数据的 `folderContents` 可观察量也可以用同样的方式进行重构。

接下来的步骤是观察 `model` 层中的变更。如果这个变更影响 `view-model` 所服务的显示时，我们将新的值发送给暴露出的可观察量作为响应。和示例 `app` 的 `MVC` 实现中一样，我们通过观察 `store` 的变更通知来做到这一点：

```
// FolderViewModel
init() {
    NotificationCenter.default.addObserver(self,
        selector: #selector(handleChangeNotification(_:)),
        name: Store.changedNotification, object: nil)
}
```

初始化方法中建立了观察，每次底层数据发生变更时，它都会调用 `handleChangeNotification` 方法。在这个方法中，我们对更改是否影响被显示内容的父文件夹进行判断。如果有影响，我们必须处理两种情况：文件夹被删除，或者文件夹发生变化（比如改名）。如果是前一种情况，我们将空的值发送给可观察量。如果是后一种，我们重置 `view-model` 的 `folder` 属性：

```
// FolderViewModel
@objc func handleChangeNotification(_ notification: Notification) {
    if let f = notification.object as? Folder, f === folder {
```

```

let reason = notification.userInfo?[Item.changeReasonKey] as? String
if reason == Item.removed {
    navigationTitleSubject.onNext("")
    folderContentsSubject.onNext([
        AnimatableSectionModel(model: 0, items: [])
    ])
} else {
    folder = f
}
}
// ...
}

```

如果变更不是一个删除操作，我们将重置 `folder` 属性，来触发它的属性观察者：

```

// FolderViewModel
@objc func handleChangeNotification(_ notification: Notification) {
    // ...
    if let f = notification.userInfo?[Item.parentFolderKey] as? Folder,
        f === folder
    {
        folder = f
    }
}

```

`folder` 属性的属性观察者会把最新的值发送给暴露出来的可观察值：

```

// FolderViewModel
var folder: Folder! {
    didSet {
        let newTitle = folder.parent == nil ? .recordings : folder.name
        navigationTitleSubject.onNext(newTitle)
        folderContentsSubject.onNext([
            AnimatableSectionModel(model: 0, items: folder.contents)
        ])
    }
}

```

对于一个像这样的简单 view-model，我们不使用响应式管道也能很容易地表达内部逻辑。如果我们的 view-model 更复杂，特别是如果它需要处理更多的内部状态的话，响应式编程在实现更为健壮这一点上会更有效果。这一节中 view-model 的完整代码可以在 [GitHub](#) 上找到。

键值观察的 View-Model

在上一节中将 Rx 从 view-model 的实现中移除之后，我们现在要更进一步，看看我们如何完全不使用 Rx 绑定来将 view-model 绑定到 view 上去。在这种变种下，我们对播放 view-model 和 PlayViewController 进行重构，让它们使用基于 KVO 的绑定。

首先，我们将播放 view-model 上的所有 Rx 可观察量属性都加上 @objc dynamic 关键字，让它们可以被 KVO 进行观察：

```
// PlayViewModel
@objc dynamic var navigationTitle: String? = ""
@objc dynamic var hasRecording = false
@objc dynamic var noRecording = true
@objc dynamic var timeLabelText: String? = nil
@objc dynamic var durationLabelText: String? = nil
@objc dynamic var sliderDuration: Float = 1.0
// ...
```

现在我们可以使用原来 MVC 实现中 PlayViewController 的代码，来在被选择的录音改变或者播放状态改变时更新这些属性。唯一的不同在于，我们在原来的实现中是将值设置到 view 上，而现在我们则通过 KVO 可观察属性来设置新的值。举例来说，在 MVC 的 PlayViewController 中，我们使用下面的代码更新播放器的进度 view：

```
// PlayViewController (MVC variant)
func updateProgressDisplays(progress: TimeInterval, duration: TimeInterval) {
    progressLabel?.text = timeString(progress)
    durationLabel?.text = timeString(duration)
    progressSlider?.maximumValue = Float(duration)
    progressSlider?.value = Float(progress)
    updatePlayButton()
}
```

在基于 KVO 的播放 view-model 中，我们可以这样来重用上面的代码：

```
// PlayViewModel (KVO-based)
```

```
func updateProgressDisplays(progress: TimeInterval?, duration: TimeInterval?) {  
    timeLabelText = timeString(progress ?? 0)  
    durationLabelText = timeString(duration ?? 0)  
    sliderDuration = Float(duration ?? 0)  
    sliderProgress = Float(progress ?? 0)  
    updatePlayButton()  
}
```

PlayViewController 的任务是使用 KVO 观察 view-model 的相关属性，并把它们的值转发给各自的 view 属性。对于这项任务，我们使用 Swift 4 的基于键路径的 KVO API 来实现一个简单的 bind 方法：

```
extension NSObjectProtocol where Self: NSObject {  
    func observe<Value>(_ keyPath: KeyPath<Self, Value>,  
        onChange: @escaping (Value) -> () -> NSKeyValueObservation  
    {  
        return observe(keyPath, options: [.initial, .new]) { _, change in  
            guard let newValue = change.newValue else { return }  
            onChange(newValue)  
        }  
    }  
  
    func bind<Value, Target>(_ sourceKeyPath: KeyPath<Self, Value>,  
        to target: Target,  
        at targetKeyPath: ReferenceWritableKeyPath<Target, Value>)  
        -> NSKeyValueObservation  
    {  
        return observe(sourceKeyPath) { target[keyPath: targetKeyPath] = $0 }  
    }  
}
```

observe(_:onChange:) 是对原生的 KVO API 的封装。我们可以在 controller 的属性中存储返回的 NSKeyValueObservation，这样就将观察的生命周期与 controller 的生命周期关联起来了。在此之上，我们还创建了 bind 辅助方法，它负责观察对象上的属性，并将新的值自动设置给另一个对象上的属性。注意我们在观察时使用了 .initial 选项：这就是说，观察者会被立即回调，这可以让设置初始值的代码和对后续变更进行响应的代码得到统一。

在 PlayViewController 的 viewDidLoad 方法中，我们使用上面这个绑定辅助方法，采取类似于 Rx 中进行绑定的方式，将 view-model 的属性绑定到 view 上：


```
// PlayViewController
var observations: [NSKeyValueObservation] = []

override func viewDidLoad() {
    super.viewDidLoad()
    observations = [
        viewModel.bind(\.navigationTitle, to: navigationItem, at: \.title),
        viewModel.bind(\.hasRecording, to: noRecordingLabel, at: \.isHidden),
        viewModel.bind(\.noRecording, to: activeItemElements, at: \.isHidden),
        viewModel.bind(\.timeLabelText, to: progressLabel, at: \.text),
        viewModel.bind(\.durationLabelText, to: durationLabel, at: \.text),
        viewModel.bind(\.sliderDuration, to: progressSlider, at: \.maximumValue),
        viewModel.bind(\.sliderProgress, to: progressSlider, at: \.value),
        viewModel.observe(\.playButtonTitle) { [playButton] in
            playButton!.setTitle($0, for: .normal)
        },
        viewModel.bind(\.nameText, to: nameTextField, at: \.text)
    ]
}
```

这个基于 KVO 的 play view-model 版本和 app 的 MVC 变种中的 PlayViewController 的实现几乎一致。主要不同是 view-model 把要显示的数据设置到可观察的属性上，而原来的 controller 则将它们直接设置到 view 上。

在这种方式中，我们不依赖于响应式框架，但依然可以从可以隔离测试的 view-model 中获益良多。不过，我们也失去了 Rx 绑定所提供的很多便利性，比如和 table view 一起工作时的那些扩展方法。在基于 Rx 的 MVVM 实现中，我们使用了 Rx 的数据源扩展来驱动 table view 的动画，而没有写任何我们自己的动画代码。对于简单的基于 KVO 的绑定，想要在这样的粒度上驱动 table view 的更新，我们必须提出自己的机制。

经验和教训

就算你的代码不采用 MVVM 模式，其中蕴含的一些思想也是可以应用到 Cocoa MVC 中的。

引入额外的层

MVVM 提供了有关将一个抽象使用到任意代码库中的经验。我们可以构建数据管道，来将 (model 中的) 抽象的数据变形为 (view 里的) 特定数据。在 MVVM 里，view-model 是 model 和 controller 之间的单一的管道。不过，我们也可以将这个模式使用到我们程序的其他部分去。下面是一些不同部件间的管道的例子：

- **App-model** - 获取 model 数据 (比如是否存在已经保存的用户凭证) 并将它与系统服务的信息 (比如网络是否可用) 合并，将这个信息作为可观察量提供给其他 view-model 使用。当不需要与系统服务进行交互时，使用**设置-model** (不是一个合并或者变形的 model，而是一个标准的 model 层对象) 来表示 **app-model** 可能会更好。
- **会话-model** (Session-model) - 追踪当前登录会话的细节，可能需要在 view-model 和主 model 之间，或者 view-model 和其他接口之间，进行网络请求的处理。
- **数据流-model** (Flow-model) - 一个类似 model 版本的协调器，用来将导航状态作为数据进行建模，并将导航状态和 model 数据合并，直接为 view-model 提供可观察的 model 数据。
- **使用例** (Use case) - 使用例指的是那些用来对主 model 进行切片准备，并且用来简化所要执行的操作的任意类型的接口或者 model。使用例和 view-model 很像，但是它并不被绑定在一个单独的 controller 上，而是可以在 view-model 之间进行传递或者共享，来在多个 view-model 中提供可重用的功能。当一个 app 有多个 view 显示同样的底层数据时，我们可以使用共通的使用例对象来对从 model 获取数据和将数据写回 model 的操作进行简化。

大多数大型程序最终都会发展到包含这些抽象。我们建议你不要过早地引入额外的抽象层。你需要精心评估某个变更是否能让你的代码变得简单 (比如更容易被理解)，是否能减少错误发生的可能，以及是否更容易维护。如果额外的抽象不能提升你写新代码的能力的话，那么我们就没有理由添加它们。

协调器

协调器是独立于 MVVM 架构之外的一种模式。协调器可以也可以被运用到 Cocoa MVC 中，用来减少 controller 的职责，并对它们解耦。要是没有协调器，一个 controller 往往会需要通过将其他的 controller 推入导航栈，或是以 modal 的方式来显示它们。当使用协调器时，controller 就可以调用协调器上的代码，而不需要直接显示其他 controller 了。

不引入单独的协调器对象，也是可以进行类似的分离的。你可以将那些负责管理展示其他 controller 的 controller 与那些负责管理 UI 的 controller 严格进行区分。如果你想了解更过关于这种协调器模式的变种的信息，可以参考 Dave DeLong 的 [《更好的 MVC》](#) 系列博客。

数据变形

从 MVVM 可以学到的另一点是，将数据变形逻辑从 controller 中抽离出来是很有好处的。Cocoa MVC 中 controller 层的一个职责就是将 model 数据变形为所配置的 view 中所需要的显示数据。通常这意味着将 model 对象上的字符串，数字或者日期转变为可以显示的形式。即使在最简单的情况下，将这边部分代码抽离出来，也可以让 controller 更加整洁，同时易于测试。

在比简单的格式化操作更复杂的情况下，数据变形可能会牵涉到更多的逻辑，这时抽离的好处就会相当明显。比如，当你的 view 依赖于 model 中被变更的内容时，你可能需要将新的数据与老的数据进行比较，或者将若干个 model 对象的数据进行整合并显示。如果能将这些操作从其他用于 view 管理的代码中分离出来的话，它们会更加清晰。

网络

5

在前面两章中，我们从不同角度检视了 MVC 和 MVVM-C 模式：如何创建它们，它们是怎样处理 `model` 变更的，以及它们管理 `view` 状态的方式。但是我们所构建的示例 `app` 中还并没有包含网络层，在本章中，我们会讨论一下如何把网络集成到 `app` 中。

基于 MVC 实现的版本，我们会用两种方式为我们的示例 `app` 添加网络支持。第一种方式是 **controller 拥有网络** - 这会将 `model` 层移除，并让 `view controller` 来处理网络请求。第二种方式是 **model 拥有网络**，它保留了 MVC 版本的 `model` 层，并在它的下方加入了一个网络层。

在 `controller` 拥有网络的版本中，数据直接从网络获取，而非从一个本地存储中取得。从网络获得的数据不会被持久化，而是由 `view controller` 负责将它们以 `view state` 的方式缓存在内存中。这样一来，这个版本的 `app` 就只能在有网络连接的情况下工作了。

对比通过 `model` 来共享数据的方式来说，`controller` 拥有网络的方式让不同 `view controller` 之间进行数据共享变得困难，因为它们之间很大程度上是彼此独立的。现在，新的数据必须要主动地传递给其他依赖它的 `view controller`，而在 `model` 拥有网络的情况下，这些 `view controller` 可以使用观察 `model` 中变更的方式来获取新数据。手动传递数据让在整个 `app` 中保证数据一致变得困难得多。这个缺陷将由 `model` 拥有网络的方式来解决。

和 `controller` 拥有网络的版本一样，`model` 拥有网络的版本也会连接到同样的服务器，但是基准的 MVC 版本中的 `model` 层可以说几乎保持不变，并且还能用来对网络获取到的数据进行离线缓存。另外，现在 `model` 还负责触发和管理网络请求，并按照需要用请求的结果来更新 `model`。`Model` 中的变更，就在没有网络的基础版本中一样，会被 `controller` 使用观察者模式捕获到。

我们在本章中想要着重强调的不是在线和离线的区别，而是网络层由 `controller` 层拥有和由 `model` 层拥有的区别。`Model` 拥有网络的版本中的离线能力，只不过是已经存在的 `model` 层所带来的附加结果。理论上，`controller` 拥有网络的版本同样可以做到离线工作 (虽然已经公认要正确实现会困难得多)，`model` 拥有网络的版本中的存储也可以仅仅只是作为内存中的缓存进行使用。

Controller 拥有网络和 model 拥有网络的录音 `app` 的不同变种都可以在 GitHub 上找到。每个版本还包含了一个配套的 `Mac app`，可以用作 `iOS app` 的服务器。

网络挑战

不管我们使用什么样的架构，在为 `app` 添加网络支持时，总是要面临一系列独特的挑战，包括下面这些：

1. 网络增加了额外的失败的来源；任何从网络获取数据的尝试，都可能由于一系列的理由以失败而告终。我们必须考虑如何优雅地处理这些错误，而且经常还要为展示这些错误创建额外的 UI。
2. 持续从网络监听数据要比观察本地数据困难得多，这往往导致我们需要周期性地手动获取数据。
3. 网络引入了多个客户端更新同一个数据的可能性，这可能导致潜在的冲突。网络上的数据可能会有独立于我们 app 的更改，引用的资源可能会在不通知客户端的情况下直接失效，两个客户端有可能更新了同样的对象，这让服务器必须要从两者间选择出哪一个应该胜出，并且告诉每一个客户端冲突的解决方案。

在本书中，我们很大程度上忽略了这些问题，因为它们并不是某个架构所特有的问题。在这一章里，我们将把注意力集中在如何将网络集成到迄今为止我们所讨论过的架构模式中去。

译者注：其实相比于架构和网络的搭配，如何处理多客户端之间的数据冲突是一门更高深的学问。但是诚如作者所说，本书中还是选择将关注点放到架构这一主题上，而非 Cocoa 网络编程。如果你对 Cocoa 中冲突的出现和处理感兴趣的话，iCloud 关于[解决文档冲突的方式](#)和[相关思考](#)，也许能给你一些启迪。

Controller 持有网络

在 controller 持有网络的实现中，view controller 负责进行网络请求。它们同时也拥有通过这些请求所加载的数据。反过来，这也意味着 app 就没有 model 层了：每个 view controller 管理着它自己的数据。

让 controller 拥有网络，是为一个 app 添加网络支持的很容易的“临时”方案，它能快速给我们结果。虽然我们一般还是建议尽量小心不要使用这种方式，不过在 controller 中写网络确实有一些正确的使用场景（我们在后面会详细讨论），而且它在很多代码库中都很流行。几乎所有的开发者都或多或少写过这样的代码，这也是我们想要把 controller 拥有网络的方式列入讨论的原因。

获取初始数据

对于一个新呈现的 view controller 的首要任务就是获取它的数据。在 controller 拥有网络的方式中，这意味着发起网络请求，并且在数据返回时对 view 进行配置。

在我们的 controller 拥有网络的 app 中，每个 view controller 只维护前一次请求所获取的数据的内存缓存，所以在 app 启动时我们肯定需要执行一次刷新。比如，文件夹 view controller 现在必须加载它想要显示的文件夹的内容。我们在 viewDidLoad 中触发这次刷新：

```
// FolderViewController
override func viewDidLoad() {
    // ...
    reload()
}
```

reload 方法会在新文件还没有加载的时候执行数据获取。为了追踪这个状态，我们为 Folder 类型添加了一个 state 属性。这个属性表示内容现在处于的状态是未加载，正在加载还是已经加载：

```
struct Folder {
    enum State {
        case unloaded
        case loaded
        case loading
    }
    var state: State
    // ...
}
```

和 MVC 版的示例 app 相反，这个 controller 拥有网络的变形中，我们使用结构体来代表文件夹和录音 (这和 MVC+VS 以及 TEA 的变种类似)。我们之所以选择这种方式，是因为我们想要强调数据在内存中被缓存的特性：最近的一次网络请求会得到一个暂时的本地快照，除非我们进行另一次网络请求，它将不会被更新。

在 reload 中，我们检查文件夹的状态，当文件夹还没有被加载时，发起网络请求来获取文件夹的内容：

```
// FolderViewController
@objc func reload() {
    guard folder.state != .loading else { return }

    folder.state = .loading
    refreshControl?.beginRefreshing()
}
```

```

task = URLSession.shared.load(store!.contents(of: folder)) { result in
    self.refreshControl?.endRefreshing()
    guard case let .success(contents) = result else {
        dump(result) // TODO 在产品代码中进行错误处理
        return
    }
    self.folder.contents = contents
    self.folder.state = .loaded
}
}

```

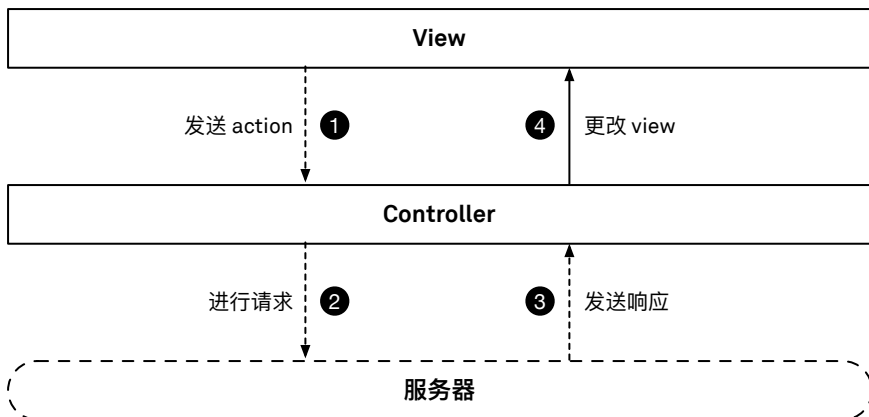
首先，我们将文件夹的状态设置为 `.loading`，这可以阻止我们在前一个请求还在途中的时候再次请求同样的数据。然后我们执行请求，更新文件夹的内容，将状态设置为 `.loaded`，并且重载 `table view`。在此期间，我们还更新了 `table view controller` 的刷新控件的状态。

`URLSession` 上的 `load` 方法是一个自定义扩展，它接受一个描述需要执行的网络请求的资源结构体（我们在[第一期 Swift Talk](#) 中详细讨论了这种方法）。这个对 `URLSession` 的封装在这里不会影响事件流的讨论，不论是直接使用 `URLSession` 还是使用其他任何的网络抽象都会得到同样的步骤。

上面从网络中加载数据的逻辑，要比从本地存储中加载同样的数据的逻辑要复杂得多（加上我们的实现还并不完整，比如错误处理的地方还没写完）。`view controller` 本来就任务繁重，它已经承担了管理 `view` 和与 `app` 框架交互的工作，而这里的网络又为 `view controller` 添加了新的职责。

进行更改

和基础的 MVC 版本不同，我们的 `app` 的 `controller` 拥有网络的版本中，并没有一个 `model` 层；我们不再拥有一个 `store` 类来管理和持久化本地数据。相反，我们 `app` 中所有数据都是暂态的，也就是说，当 `app` 退出时它们不会被保存下来。保存数据的任务被完全外包给了服务器。所以，对数据的更改现在是一个通过网络执行的异步任务。为了描述这会对我们的代码造成什么样的影响，我们来研究删除一个文件夹时所需要的步骤：



步骤 1: Table View 触发 View Controller 上的 Action

这和 MVC 中的方法一样：删除按钮触发 table view 的 data source 中的 `tableView(_:commit:forRowAt:)` 方法。

步骤 2: Controller 发起网络请求

在这个 data source 方法中，我们首先要确保我们正在处理的是一个删除事件，然后我们将条目删除。删除代码根据文件夹和录音的不同而有所区别，不过我们这里只对文件夹的路径进行了研究：

```
override func tableView(_ tableView: UITableView,
    commit editingStyle: UITableViewCellEditingStyle,
    forRowAt indexPath: IndexPath)
{
    guard editingStyle == .delete else { return }
    let item = folder.contents[indexPath.row]
    switch item {
    case .folder:
        // 执行请求...
    }
}
```

实际的删除由一个网络请求来完成：

```

URLSession.shared.load(store.change(.delete, item: item)) { result in
    guard case .success = result else {
        dump(result) // TODO production error handling
        return
    }
    self.deleteItem(item: item)
}

```

注意，直到我们从网络服务收到响应为止，我们都没有改变缓存。这意味着这些变更是事务性的：只有当我们收到网络服务的响应，告诉我们删除成功时，我们才会从缓存中（以及 view 中）将数据删除掉。

在一个产品级的 app 中，我们还会需要处理至少两个附加任务：正确地处理网络错误，以及通过某种提示告诉用户删除正在进行。现在，在网络请求返回时，文件夹仅仅只是消失。根据网络状况的不同，这有时会立即发生，有时要花上好几秒。为了简洁明了，我们将这些任务从例子中去掉了。

步骤 3 & 4: Controller 更新数据及 Table View

在网络的回调中，view controller 的 `deleteItem` 会被调用。这个方法执行两个任务。它会将已删除的文件夹从 view controller 的缓存数据中移除掉，并且将 table view 中对应的行也移除：

```

func deleteItem(item: Item) {
    guard let index = folder.contents.index(where: { $0.uuid == item.uuid })
    else { return }
    folder.contents.remove(at: index)
    tableView.deleteRows(at: [IndexPath(row: index, section: 0)], with: .automatic)
}

```

到这里为止，删除条目就完成了。除了这个操作中异步特性的部分，和基础 MVC 版本相比，主要的不同在于作为用户删除某个特定行的回应，这里我们主动地更新 table view。而在基础版本中，我们指导 store 移除某个特定的条目，并且 table view controller 会根据 store 发出的变更通知来更新自身，在一个 controller 拥有网络的版本中，这些机制都不存在。

讨论

我们在本章前面的地方指出过，对于大多数 app，我们总体上不推荐使用 controller 拥有网络的方式。不过，为了理解在哪些特定情况下 controller 拥有网络的方式可行，在哪些情况下它不可行，我们需要再深入讨论一下这种方式中的权衡。

由于 view controller 拥有从网络获取的数据，这种方式缺少了 model 层。也就是说，由 view controller 所拥有的数据，实际上是 view state。如果 app 中没有其他部分依赖它的话，这些本地管理的 view state 可以良好工作。一旦这些 view state 需要在不同 view controller 中进行共享的话，我们通常会（在 view controller 层级之外）创建一个对象来持有这个状态，并且让它可以被改变和观察。

将这个理论推广到在 view controller 中对网络数据进行本地缓存的情况，我们可以得到以下结论：如果通过 view controller 加载的任何网络数据需要被 app 中其他部分所共享的话，我们需要重新引入 model 层。如果这些数据只是在本地使用的话，那么，原则上来说，我们可以使用 controller 拥有网络的方式，不需要 model 层也能构建 app。

话虽如此，但是还有一些其他问题需要考虑。首先，就算 controller 拥有网络的方式当前来说是可行的，我们需要考虑在近期的未来可能发生的变更。通常，随着 app 功能的增加和复杂度的增长，共享数据的需求有可能出现。因为 model 拥有网络的方式对于简单 app 来说也同样可以良好工作，所以也许我们从一开始就选择对未来更友好的方式会比较明智。

其次，controller 拥有网络的方式将大量的新的职责放到了 view controller 中，包括发起网络请求，处理结果，以及处理网络失败等。由于 view controller 通常已经承担了足够多的责任，我们建议不要把所有这些代码放到 view controller 里，而是将它重构出来，形成某种网络服务。网络服务可以由简单的函数构成，或者也可以通过实现一个处理和服务器交互的操作的网络服务类来完成。注意，这里的网络服务不是我们所说的 model 层，它是一个单纯的封装，用来完成发起请求的操作，这些服务不会持有结果数据。

接下来，我们会看一看 model 拥有网络的示例 app，并且将它与 controller 拥有网络的版本进行一些对比。

Model 拥有网络

与 controller 拥有网络的版本相反，model 拥有网络的版本直接使用原有的 MVC 代码库，而不需要做很多修改。它会添加一个网络服务部件，用来直接和 store 进行对话。因为从网络获取的数据是由 model 层持有的，所以我们将它称为 model 拥有网络。

View controller 中所添加的唯一的职责是开始加载数据 (比如下拉刷新或者当导航至一个新的画面时)。然而, 和上面 controller 拥有网络的方式不同, view controller 不负责处理网络返回的数据。这些数据被插入到 store 中, 和处理本地变更的方式一样, view controller 通过监听 model 变更通知来接收这些变更。

这种方式的一大优势是, 在 app 中分享数据和进行通讯会非常简单。所有的 view controller 从 store 中获取它们的数据, 并且订阅 model 的变更通知。就算是 app 中多个部分独立地显示或者变更相同的数据, 这些数据也会具有一贯性。

我们的 model 拥有网络的代码库增加了一些复杂度。不过, 这不能归咎于设计模式。我们的本意就是让 model 拥有网络的版本比 controller 拥有网络的版本具有更多的能力: 我们使用 store 作为离线缓存, 这让 app 在没有网络连接的时候也能工作。在客户端, 我们可以立即更新数据, 而不用先去请求服务器。由于有这些改进, 我们需要在客户端增加额外的代码, 来追踪未提交的变更, 并且在把变更提交给服务器时, 对可能发生的冲突进行处理。

我们在下面仔细查看实现的时候, 会把注意力集中在它与 controller 拥有网络的方式的区别, 特别是网络如何进行集成的这一重点上面。在其中, 我们只会在必要的时候触及这些追加特性 (和复杂度) 的细节, 因为它们对于 app 架构来说, 并不是关键。

获取初始数据

和 controller 拥有网络的版本类似, 我们在文件夹 view controller 的 viewDidLoad 方法中获取初始数据:

```
// FolderViewController
override func viewDidLoad() {
    // ...
    reload()
}
@objc func reload() {
    task?.cancel()
    refreshControl?.beginRefreshing()
    task = folder.loadContents { [weak self] in
        self?.refreshControl?.endRefreshing()
    }
}
```

在 reload 方法中，我们更新 refresh 控件的状态，然后调用当前文件夹的 loadContents。这会让 model 层从网络刷新文件夹的内容，它会获取服务器上可能发生的所有的变更。Model 拥有网络的方式中，reload 方法比 controller 拥有网络中的实现有了大幅简化：我们触发重载，但是不需要关心处理请求的结果。这部分内容由 model 层完成，我们会在下面的 loadContents 方法里看到这一点：

```
extension Folder {
    @discardableResult
    func loadContents(completion: @escaping () -> () -> URLSessionTask? {
        let task = URLSession.shared.load(contentsResource) { [weak self] result in
            completion()
            guard case let .success(items) = result else { return }
            self?.updateContents(from: items)
        }
        return task
    }
}
```

在上面的代码中，我们（使用与 controller 拥有网络的版本里所用的相同的小型网络抽象）执行实际的网络请求和处理结果。这个请求返回一个 Item（即文件夹或录音）数组，我们用它来更新文件夹的内容。在把来自网络的新内容与具有等待更改的本地项目合并后，updateContents 最终会设置 contents 属性并保存更改：

```
extension Folder {
    func updateContents(from items: [Item]) {
        // ...
        contents = merged
        store?.save(self, userInfo: [Item.changeReasonKey: Item.reloaded])
    }
}
```

就像在基准 MVC 版本中一样，store 在保存后发送通知。进行重载操作的文件夹 view controller 获取这个通知，并通过我们在 MVC 章节 中所描述的正常 model 观察路径，对 table view 进行更新。

进行更改

在 model 拥有网络的版本中对数据进行更改的方式，与基础 MVC 中的方式是一样的。比如，要删除文件夹，我们使用完全相同的步骤：通过 table view 的 data source 方法接收到删除操作，将文件夹从父文件夹里删除，保存变更，然后响应 model 的变更通知并更新 table view (具体可以参考[MVC 一章中模型变更的部分](#))。

这证明了 model 拥有网络的方式，对 controller 层的大部分实现来说是透明的。Controller 与 store 中的数据进行交互，就好像它们仅仅是本地数据一样，网络则发生在 model 层的背后。

对于 model 拥有的网络的实现，我们决定尽可能地保持 MVC 版本的 model 层不变，并添加独立于 store 的网络服务。要从本地 store 获取更改，网络服务将与 view controller 一样观察相同的变更通知：

```
final class Webservice {
  init(store: Store, remoteURL: URL) {
    // ...
    NotificationCenter.default.addObserver(self,
      selector: #selector(storeDidChange(_:)),
      name: Store.changedNotification, object: nil)
  }

  @objc func storeDidChange(_ note: Notification) {
    guard let pending = PendingItem(note) else { return }
    pendingItems.append(pending)
    processChanges()
  }
  // ...
}
```

我们把需要提交给网络服务的变更从通知中提取出来，这包括变更的种类 (比如是一次更新还是一个删除操作)，以及被更改的条目当前的属性。这些信息存储在 PendingItem 结构体中。因为可能有其他的更改正在进行或者正在等待提交，或者当前我们可能没有网络连接，所以我们将这个结构体添加到队列里。最后我们通过调用 processChanges 来启动队列的处理。

processChanges 的实现会从队列中取出第一个项目，尝试将它提交到服务器，并且处理响应。如果响应是一个错误，我们必须决定如何处理冲突。当成功时，我们将该项目移出队列，并发送变更通知，让其他组件有机会对项目的新的提交状态作出响应。

```

// WebService
func processChanges() {
    guard !processing, let pending = pendingItems.first else { return }
    processing = true
    let resource = pending.resource(remoteURL: remoteURL,
        localBaseURL: store.localBaseURL)
    URLSession.shared.load(resource) { [weak self] result in
        guard let s = self else { return }
        if case let .error(e) = result {
            // 错误处理，省略...
        } else {
            s.pendingItems.removeFirst()
            // 发送通知...
        }
        s.processing = false
        s.processChanges()
    }
}

```

这是网络代码的简化版本，你可以在 [GitHub](#) 上找到完整版本。除了上面显示的代码之外，网络服务还会在本地将待处理更改队列进行持久化 (并在初始化时从磁盘加载它)，这样离线时的本地更改也不至于丢失。

想要改进使用离线和在线数据的用户体验，只需要更改 controller 就可以了。比如，我们可以添加强制重载数据的能力，可以通过优雅的方式通知用户发生了错误，或者在屏幕上指示出数据离线/在线的状态。举个例子，我们通过扩充文件夹 view controller 的 data source 方法，来指示某个条目是否具有待提交的本地变更：

```

// FolderViewController
override func tableView(_ tableView: UITableView,
    cellForRowAt indexPath: IndexPath) -> UITableViewCell
{
    // ...
    let cell = tableView.dequeueReusableCell(withIdentifier: identifier,
        for: indexPath)
    cell.backgroundColor = item.latestChange?.color ?? .white
    return cell
}

```

讨论

从架构的角度来看，上述两种方法 (controller 拥有网络和 model 拥有网络) 之间的主要区别在于数据的所有权。在 controller 拥有网络的方式中，view controller 在本地持有数据，而在 model 拥有网络的方式中，model 层中的一些实体 (我们的示例中的 Store 类) 持有数据。

View controller 中的网络代码通常会被认为是反模式的，不过，原则上来说，如果网络过来的数据只作为本地 view state 使用，而不需要与其他部件进行通讯时，在 controller 里写网络代码并不是什么问题。如果我们可以把大块的网络代码正确地提取到辅助函数中去，这种方式甚至不会使我们的 view controller 臃肿很多。真正关键的问题是，这些数据会需要被共享吗？

一旦数据需要被共享，model 拥有网络就编程了天然的选择：由于数据是由 controller 层以外的实体所拥有，它将能够很容易地在整个 app 中进行共享，而不会遭遇数据不一致的问题。这和 view state 很类似；如果某个 view state 只和一个 view 或者一个 view controller 有关，那么在本地存储它是没问题的。但是，一旦其他的组件依赖这个状态，我们就需要将它移动到一个可以被观察的共享实体中去。这样来看，这些状态其实距离独立于 app 框架的共享模型，其实只有一步之遥。

译者注：我们在后面的章节中的架构下，将会看到一些例子，它们的核心思想在于把 view state 当作是 model 的一部分来处理。这使得像是在不同部件间共享状态，或者是实现 app 层级的状态恢复等操作，都变得简单不少。可以说，model 拥有网络的实现方式，为后面“model 拥有状态”的实现，奠定了一些基础。

其他架构下的网络

本章中的两种网络方式不只适用于 MVC，对于像是 MVVM 这样的相关的模式依然适用。区别在于，controller 拥有的网络在 MVVM 中将会变成 view-model 拥有的网络。不过，它们的实现是非常相似的。

然而，在像是 MVC+VS，MAVB，或者 TEA 这样的模式中，controller 拥有网络和 model 拥有网络的差别就不适用了。举个例子，MVC+VS 的前提是，将 view state 明确地作为 model 的一部分来进行表示。由于 controller 拥有网络方式中，数据是 view state 的一部分，所以它将会被立即推入到 model 层中。这样一来，在这个上下文中，区分 controller 拥有网络还是 model 拥有网络，就变得没有意义了。

在 MAVB 和 TEA 中，不存在 view controller，当然 controller 拥有网络的方式也就无法实现。所以，网络代码将很自然地移动到 model 层去，它们会对 app 状态进行更新，这些状态要么是 view 所绑定的目标 (MAVB)，要么是虚拟 view 进行构建的依据 (TEA)。

译者注：如果你是顺序阅读本书的话，我们其实还没有详细介绍 MAVB 和 TEA 的架构方式，所以你可能无法理解这句话的确切涵义。我们建议你可以先暂时跳过这里，在阅读了本书后面关于 MAVB 和 TEA 的章节，并对它们有了基本的了解后，再会来看看这句话。相信你会对网络在这些模式中所扮演的角色和地位，会有更深入的理解。

Model-View-
Controller+
ViewState

6

Model-View-Controller+ViewState (模型-视图-控制器+视图状态, MVC+VS) 是MVC 模式的一种变体。它维护着同样的层次结构, 并且使用像是 notification 和 delegate 这样的相同的平台原生机制来连接各个部件。但它在一个重要的方面有所不同: 它将 view state 视为 model 层的一部分。所有 view state 都由 model 层中一个独立的 store 明确表示。View controller 观察这个 view state 的 store, 并且相应地更新 view 层级。将 view state 变为 model 的一部分, 为我们提供了一种一致的处理 view state 变更的方法, 这和 MVC 中 view state 通讯时所采用的临时的风格是完全不同的。

通过将 view state 建模为单个的结构体, 而不依赖于各种 view 对象上的隐式 view state, 这种架构让使用 view state 书写功能变得更简单。比如, 对整个 view state 进行序列化的工作变得很直接。在某个给定的点上 (比如在调试的时候), 我们可以很容易地监测整个 view state 的情况。

将 View State 作为 Model 的一部分

对于 app 架构的理论分析经常会把 app 的传统角色映射成一个文档编辑器。在一个典型的 MVC 中所定义的 model, 会被表示成一个从磁盘上读取的文档。Model 包含了 app 中的状态, 当用户执行保存操作时, 这些状态将被存储到磁盘上文档的文件中去。我们将会把这个定义叫做**文档 model** (document model)。

导航栈中 view controller 的列表, scroll view 的滑动位置, tab 和 table row 的选择, 未提交的文本框, 以及其他未完成的编辑操作等, 通常不会被存储到文档里, 所以它们是文档 model 之外的东西。我们之前已经将这种文档外的、与 view 相关的状态定义为了 **view state**。

将状态区分为需要进行持久化的文档 model 和暂时的 view state, 会带来两个问题:

1. 在现代的 iOS app 中, view state 是希望被持久化的。
2. 非文档操作无法从 model 抽象中受益。

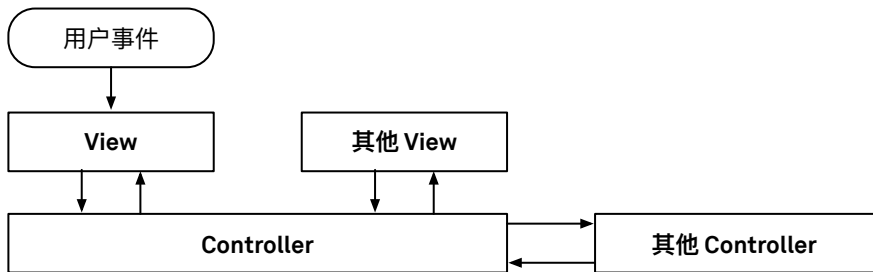
UIKit 提供了一些临时的对 view 进行持久化的方法 (大部分通常是通过 storyboard) 来缓解第一点, 但是对于第二点, 因为 view state 不是 model 的一部分, 所以对此我们无能为力。

让我们来考虑一个复杂的 view action: 点击 table 中的一行, 导致导航发生变化。作为这个 view action 的响应, 会有下面这一连串的步骤:

1. 点击一行通常会更新选中行和 table 的其他一些状态, 来向用户提供反馈。

2. Table view controller 使用选中的行获取模型对象 (并且可能会在之后使用状态来更新这一行)。
3. Table view controller 可能需要更新它自身的本地状态 (比如更新计时器, 迁移的动画, 或者失去用户焦点的 view)。
4. Table view controller 可能需要决定现有的目标 view controller 的细节, 所以它可能需要先读取目标 view controller 的状态, 然后再使用新的值对它进行更新。

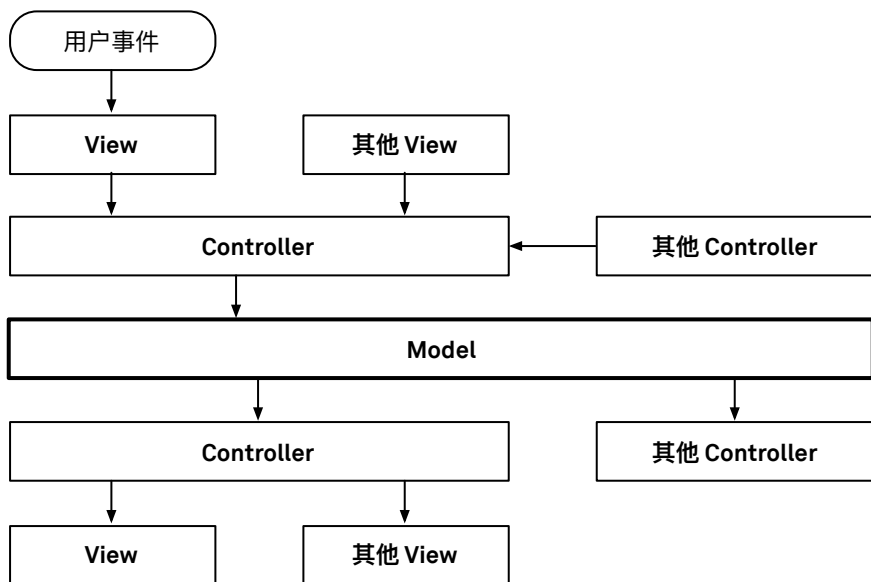
一种可能的数据流图表如下所示:



用户事件会触发一次 view action, 这个 action 被转发给 view controller。View controller 负责处理接下来所有的 action。

在 MVC 中, view state 的 action 中的数据流没有特定的结构, 通常它甚至没有明确的结构。为了正确传递 action, 处于 action 中心的 controller 需要理解所有其他对象, 包括其他 controller 的数据依赖。当所涉及的 view state 不多时, 管理起来不是很困难, 但是随着组件数量的增长, 管理的难度会逐渐增加。让 view state 更新按照正确的顺序更新, 通常会变成一个试错的过程。

如果上面提到的 view action 遵循和 MVC 中的 model action 相同的方式进行处理的话, 图表就会变成这样:



一眼看上去，似乎要做的事情只多不少：图中的节点由四个增加到了九个。然而，我们实际上只不过是原本两次经过自己的图进行了展开，让它变成只有一个方向的图，这就是单向数据流的概念。结果得到的结构中，“我是如何触发这个 action”的问题依然在顶端，“如何展现这个状态”依然位于底部。

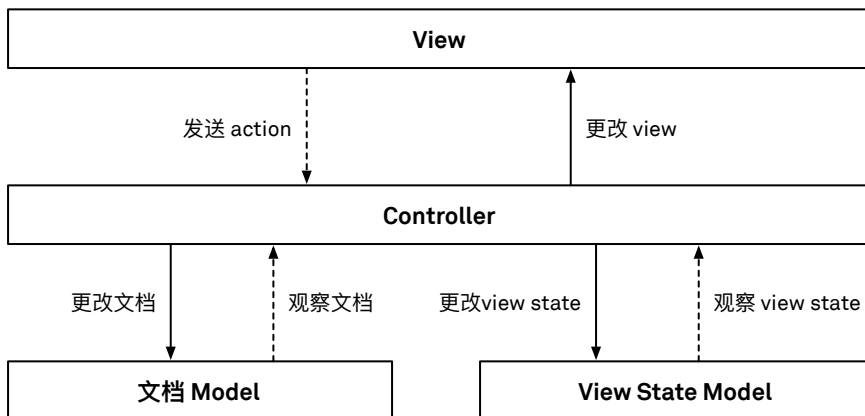
将关注点进行分离会很有帮助，不过这里最值得称道的改进还是数据流。数据现在明确地从顶部流至底部，而且中心的 controller 也不再需要知道次级 controller 中的数据依赖的信息 (它们可以独立地对 model 进行观察)。

使用 model 抽象的其他一些好处包括：

- Model 的干净的接口可以让我们对实现进行替换，而不会影响到暴露给程序其余部分的接口。
- Model 为 action 提供了单一的事实来源，这样，重构和测试可以只关注单一的某个点。
- 由于图中下半部分的代码可以被不同的 action 重用，所以结构的扩展性会更好。

尽管 MVC+VS 提出将 view state 作为 model 层的一部分，但 view state model 还是与文档 model 保持分离，这是因为与文档 model 相比，view state 生命周期稍有不同 (view state 使

用 UIKit 的状态恢复进行持久化，并且可能会在发生崩溃时被清除)。不过，两者都遵循相同的规则：



在 view state 驱动的示例 app 中，view state model 和文档 model 分别由 ViewStateStore 和 DocumentStore 类来表示。DocumentStore 类所扮演的角色与 MVC 版本的 app 中的 Store 类完全相同。

结果所得到的模式和TEA 在结构上有些相似，只不过它拥有一个单独的 view state 和 model，而且它不需要一个将 UIKit 完全抽象出去的框架。在 MVC+VS 里，在 view controller 里观察 view state 并且更新 view 的方式，和 MVC 中已有的处理普通 model 变更的时候使用的方式是一样的。

探索实现

值类型 Model 对象

MVC+VS 版本的录音 app 和基础的 MVC 版本实现相比，有一个很重要的不同：对文档 model 和 view state model，MVC+VS 里都使用值类型来代表。Folder 和 Recording 类型现在都被定义为结构体，SplitViewState 和 FolderViewState 这些 view state 相关的类型也是如此。

我们在进行基本的 MVC 的实现时，使用的是引用类型 (换言之，使用了类)，因为我们认为它更能反映编写 Cocoa 程序的传统方式，因为在 Objective-C 中，引用类型占了主导地位。不过，

使用这种在 model 层和 controller 层之间共享所有权的引用类型，违背了 MVC+VS 旨在体现的单向数据流的精神。

在单向数据流中，状态只能通过订阅来读取。为了强制这一点，我们的 model 只公开订阅 API，而无法直接读取其中的值。使用值类型来对我们的数据建模，可以强化这个概念 - 如果我们使用了引用类型，就可能会存在从共享引用中访问并获取新值的诱惑。

构建

将 view state 和其他模型 model state 看作一致，就意味着我们需要首先构建 view state，然后在构建所触发的观察回调中，我们应该构建 app 的 view。

但是，root view controller 的构造是该规则的一个例外。由于在此模式中，view controller 是观察者，因此在没有构建至少一个 view controller 之前，我们是无法观察 view state model 的。正因如此，我们的录音 app 中的 split view controller 需要在加载 Main.storyboard 文件时被无条件构建。

在我们的实现中，每个 view controller 都有一个 context 属性。这个属性中提供了对文档 model 和 view state model 两者的引用。通过使用 context 属性的方式，可以避免对数据 store 或者协调器使用单例。这个属性必须在 view controller 的 viewDidLoad 函数被调用前进行设置。通常，这发生在构建阶段（对于直接初始化的 view controller 来说），或者发生在 prepare(for:sender:) 函数中（对于通过 segue 创建的 view controller 来说）。

对于 root view controller，这个 context 能够被设置的最早的时间是在 app delegate 的 application(_:didFinishLaunchingWithOptions) 方法中：

```
func application(_ application: UIApplication,
    didFinishLaunchingWithOptions launchOptions:
        [UIApplicationLaunchOptionsKey: Any]?) -> Bool
{
    setContextOnSplitViewIfNeeded()
    // ...
}

func setContextOnSplitViewIfNeeded() {
    let splitVC = window?.rootViewController as! SplitViewController
    if splitVC.context == nil {
        splitVC.context = StoreContext(documentStore: .shared,
```

```

        viewStateStore: .shared)
    }
}

```

Split view controller 中拥有大量的状态，这是因为它同时是 master 和 detail 两侧的导航栈的父级控制器。所以我们选择在 app 的这个版本中，把它从 app delegate 中分离出来，创建一个 SplitViewController 类。因为 split view controller 还需要在启动时加载 master 和 detail view controller (它们分别负责显示根文件夹 view controller 和播放界面 view controller)，我们必须立即将 context 传递给这些子 view controller：

```

// SplitViewController
override func loadView() {
    super.loadView()
    let masterViewController = masterViewNavigationController.topViewController
    as! FolderViewController
    masterViewController.context = context.folderContext(index: 0)
    let detailViewController = (self.viewControllers.last as! UINavigationController)
        .topViewController as! PlayViewController
    detailViewController.context = context
}

```

Detail view controller (播放界面) 的 context 和 split view controller 的 context 是同样的，换言之，它们引用了同一对文档 store 和 view state store。然而，文件夹 view controller 要求 context 还应该包含一个 index 值，这样文件夹 view controller 才能在 view state store 所代表的文件夹 view 的栈中找到自己。

在 MVC+VS 里，所有的 view controller 必须知道如何在 view state store 中定位自己。对于像是 play view controller 这样的唯一的 view controller 来说，这很简单，因为在 view state 中只有唯一一个 playView，所以它可以被直接访问到。但对于导航栈中的文件夹这类非唯一的 controller 来说，我们需要在 context 里传递一个额外的值，用来唯一确定某个 controller 的身份。对于文件夹 view controller，一个索引值就足够让我们在文件夹 view controller 的数组中对它们进行定位了。

要注意的是，像这样传递给 context 的信息，应当只对 view controller 被父级插入到 view 层级中的位置进行描述，以此来定位 view controller 所关联的 view state。Context 不应该包含任何实际的数据 (数据应当存储在 view state 中)。

将 View 与初始数据连接

一旦 context 完成设置，并且 view 被添加到窗口中完成加载后，view controller 必须将它在 view 层级中的状态与 view state 和文档 store 中所描述的状态进行同步。

MVC+VS 的一个关键原则是，我们不仅仅是从 view state 或者文档 model 中**拉取**数据，我们还必须**订阅**这些数据 (这样我们可以保证能接收到未来的变更)。在每一个 view controller 中 (包括 SplitViewController)，这发生在 viewDidLoad 方法里：

```
// SplitViewController
override func viewDidLoad() {
    // ...
    observations += context.viewStateStore.addObserver(actionType:
        SplitViewState.Action.self) { [unowned self] state, action in
        self.state = state
        self.synchronizeMasterNavigationController(action)
        self.synchronizeDetail(action)
        self.synchronizeModalAlerts(action)
    }
}
```

共享的 view state store 上的 addObserver 方法是标准 Cocoa 通知中心上的一个封装。actionType 参数允许我们只接收那些我们感兴趣的 action (每个 view state 变更都和一个描述它的来源的 action 相关联；在这里，我们只想知道来源于 split view controller 的 view state 的变更)。

观察回调会在我们添加观察者的时候被**立即**调用 (这是我们在 MVC 一章中的“改进”部分的内容中所建议的)。这个立即调用的回调将接收到初始的状态值 (一个 SplitViewState 实例)，以及一个 nil 的 action (代表这是一次完整的 view state 重载，而不是一个用户 action)。

SplitViewState 中包含了一些嵌套的结构体值，用来指定 app 中不同组件的 view state：

```
struct SplitViewState: Codable {
    var folderViews: [FolderViewState]
    var playView: PlayViewState
    var recordView: RecordViewState?
    var textAlert: TextAlertState?
    // ...
}
```

由于 split view controller 有许多独立的子树要管理，因此它的 view state 观察处理方法被分为了三个 synchronize... 方法。按照 split view state 中所描述的结构体，它们分别负责创建剩余的 master 层级，detail 层级和 modal 层级。比如，导航栈会在 synchronizeMasterNavigationController 方法中被构建：

```
// SplitViewController
```

```
func synchronizeMasterNavigationController(_ action: SplitViewState.Action?) {  
    switch action {  
        // ... 省略了非 nil 的情况  
    case nil:  
        masterViewNavigationController.viewControllers =  
            state.folderViews.indices.map  
        { index in  
            let fvc = storyboard!.instantiateViewController(  
                withIdentifier: .folderIdentifier) as! FolderViewController  
            fvc.context = context.folderContext(index: index)  
            return fvc  
        }  
    }  
}
```

state.folderViews 数组将被直接转换为在 master navigation controller 中使用的 view controller 数组。这里的代码也展示了把 split view 中的 context 传递给 folder view 的方式。

因为 split view controller 只依赖于 view state，所以它不需要观察文档 store，但是文件夹 view controller 就需要观察文档。如前所述，文档 state 和 view state 非常相似，对于观察逻辑来说，也是一样的：

```
// FolderViewController
```

```
override func viewDidLoad() {  
    // ...  
    observations += context.documentStore.addObserver(actionType:  
        DocumentStore.Action.self) { [unowned self] (store, action) in  
        guard  
            let folderState = self.state,  
            let folder = store[folderState.folderUUID]?.folder  
            else { return }  
        self.folder = folder  
        self.handleStoreNotification(action: action)
```

```
}  
}
```

这里唯一取巧的地方在于，回调闭包使用了 `self.state` (文件夹的 `view state`) 来获取这个 `view controller` 的文件夹的 `UUID`。这是 MVC+VS 中的标准模式：任何 `model` 对象的标识都应该存储在 `view state` 中，而不是传递给 `context` 对象中的 `view controller`。不过，这就意味着对文档 `store` 的观察是依赖于当前的 `view state` 的，这个观察必须在 `view state` 的观察开始之后才能被初始化。

几乎在所有情况下，`view controller` 的 `model` 对象的标识符 (在本例中的文件夹 `UUID`) 在 `view controller` 的生命周期中都不应该更改。但是假设如果你想要它发生改变的话，你需要在 `view state store` 的观察回调里面，重新创建对文档 `store` 的观察。

`Model` 观察回调会在 `handleStoreNotification` 方法中将 `model` 数据应用到 `view` 上去：

```
// FolderViewController  
func handleStoreNotification(action: DocumentStore.Action?) {  
    guard let folder = self.folder else { return }  
    switch action {  
        // ... 省略了非 nil 的情况  
        case nil:  
            title = state?.folderUUID == DocumentStore.rootFolderUUID  
                ? .recordings : folder.name  
            updateCachedSortedUUIDs()  
            tableView.reloadData()  
        }  
    }
```

这里的 `switch` 语句包含着大量的 `case`，它们都是后续变更相关的处理，但是 `nil case` 就是专门用来处理 `view` 的完整重置的，比如当为这个 `view` 设置初始数据时，就会落在这种情况下。因为我们在之前就两个观察者回调中接收到了 `self.state` 和 `self.folder` 的值，我们可以使用这些被缓存的值中的数据。

状态恢复

由于 MVC+VS 始终在 `view state model` 中维护了完整的 `view state`，保存和恢复 `view state` 就变得很简单了。`ViewStateStore` 类中在 `content` 属性里包含了需要持久化的根节点的 `view state` 数据，它是一个 `SplitViewState` 结构体。这个结构体，以及它所包含的所有结构体，都

遵守 Swift 的 Codable 协议，这样它就能在任意时刻从 JSON 中进行反序列化，或者序列化打包为 JSON。

View state 的存储和恢复是由 AppDelegate 触发的：

```
// AppDelegate
```

```
func application(_ application: UIApplication,
    willEncodeRestorableStateWith coder: NSCoder)
{
    guard let data = try? ViewStateStore.shared.serialized() else { return }
    if ViewStateStore.shared.enableDebugLogging {
        print("Encoding for restoration: \(String(decoding: data, as: UTF8.self))")
    }
    coder.encode(data, forKey: .viewStateKey)
}
```

```
func application(_ application: UIApplication,
    didDecodeRestorableStateWith coder: NSCoder)
{
    guard let data = coder.decodeObject(forKey: .viewStateKey) as? Data
    else { return }
    ViewStateStore.shared.reloadAndNotify(jsonData: data)
}
```

当 ViewStateStore 类的 content 属性由于恢复操作而发生变化时，它会发送一个 notifyingStoreReloadNotification 的通知。每个将自己添加为 view state 观察者的 view controller 都会接收到观察回调。和构建时一样，通知中所关联的 action 是 nil。View controller 需要检查自己的状态，并且将它们更新到新的重载状态，理想状况下，只去改变那些需要改变的属性。如上所示，状态恢复的代码路径和初始构建的代码路径是一致的。

更改 Model

当更改文档 model 时，MVC+VS 和 MVC 相比大体上并没有什么不同。两种模式中 view controller 都 (通过 target-action 或者 delegate) 接收 view action，通过方法调用更改 model，然后通过通知来观察变更。不过，由于我们 MVC+VS 的实现使用了值类型的 model 和一种不同的观察方式，它们严格强制了任何对于 model 的读取必须通过观察 model 来进行这一思想，所以还是有所不同。

我们从文件夹 view controller 开始，来看看如何从文件夹里删除一个项目。在示例 app 中，我们通过 storyboard，将文件夹 view controller 设置为了 table view 的 data source。为了处理删除按钮的点击事件，table view 会调用 data source 上的 tableView(_:commit:forRowAt:) 方法：

```
// FolderViewController
```

```
override func tableView(_ tableView: UITableView,
    commit editingStyle: UITableViewCellEditingStyle,
    forRowAt indexPath: IndexPath)
{
    guard editingStyle == .delete, let uuid = uuidAtIndexPath(indexPath)
    else { return }
    context.documentStore.removeItem(uuid: uuid)
}
```

Store 在 removeItem 中对项目确实存在，并且有一个有效的父文件夹进行了验证，进而处理这个调用。然后，它会将项目从父文件夹的本地复制删除，并且调用 Store.content 的 updateValue 来更新 store：

```
// DocumentStore
```

```
func removeItem(uuid: UUID) {
    guard
        let item = content[uuid],
        let parentUUID = item.parentUUID,
        var parentFolder = content[parentUUID]?.folder,
        let oldIndex = parentFolder.sortedChildUUIDs(in: self).index(of: item.uuid)
    else { return }

    parentFolder.removeChild(item.uuid, store: self)
    content.removeValue(forKey: item.uuid)
    content.updateValue(folder(parentFolder), forKey: parentUUID)
    commitAction(Action.removed(parentUUID: parentUUID, childUUID: uuid,
        oldIndex: oldIndex))
}
```

对 commitAction 的调用会保存 store，并且触发更改的通知。FolderViewController 在 handleStoreNotification 中观察到这个变更，并删除对应的行：

```
// FolderViewController
```

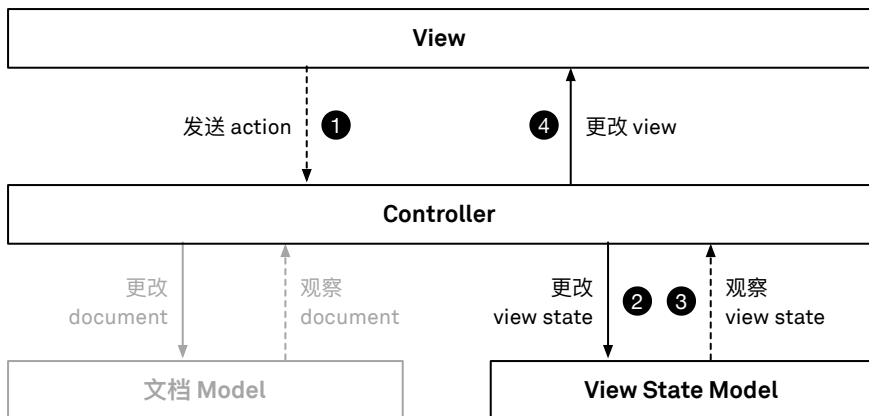
```

func handleStoreNotification(action: DocumentStore.Action?) {
    guard let folder = self.folder else { return }
    switch action {
    case let .removed(parentUUID, _, oldIndex)? where parentUUID == folder.uuid:
        updateCachedSortedUUIDs()
        tableView.deleteRows(at: [IndexPath(row: oldIndex, section: 0)], with: .right)
    // ...
    }
}

```

更改 View State

更改 model 的方式和 MVC 中所做的非常相似，但是 view state 的更改就非常不一样了。对于 view state 的处理方式正是 MVC+VS 这一变体中所不同的地方。在 MVC 中，view state 的变更通常会由本地的 view 或者 view controller 进行处理，而在 MVC+VS 中它们必须要到 view state store 中去转一圈：



我们会通过研究两个不同的例子来探索 view state 的使用方式：切换 play view controller 中的播放按钮的文本，以及通过选择文件夹，将新的文件夹 view controller 推入导航栈。

示例 1: 更新播放按钮

步骤 1: 按钮将 Action 发送给 View Controller

这一步和 MVC 版本中的一样：播放按钮在 storyboard 中通过使用 IBAction 连接到 play view controller 的 play 方法。

步骤 2: View Controller 更改 View State

当 MVC 中的播放按钮被按下时，view controller 将立即调用 audioPlayer 的 togglePlay 方法。我们没有在 audio player 外部追踪这个状态：

```
// PlayViewController
@IBAction func play() {
    audioPlayer?.togglePlay()
    updatePlayButton()
}
```

而在 MVC+VS 中，我们不直接通过 audio player 进行调用，play 方法必须通过 view state store 来传递这个变更。这需要一些步骤，首先我们要调用 view state store 上的 togglePlay：

```
// PlayViewController
@IBAction func play() {
    context.viewStateStore.togglePlay()
}
```

View state store 中 togglePlay 方法的实现负责更新 playView 属性的状态，这个属性代表了 play view controller 的状态。然后使用 commitAction 方法来发送一个 .togglePlay 的通知：

```
// ViewStateStore
func togglePlay() {
    guard var playState = content.playView.playState else { return }
    playState.isPlaying = !playState.isPlaying
    content.playView.playState = playState
    commitAction(PlayViewState.Action.togglePlay)
}
```

步骤 3: View Controller 观察 View State 变更

Play view controller 订阅了 view state 的变更。和我们之前看到的一样，我们在 viewDidLoad 中建立这个观察：

```
// PlayViewController
override func viewDidLoad() {
    // ...
    observations += context.viewStateStore.addObserver(
        actionTypes: PlayViewState.Action.self) { [unowned self] (state, action) in
        self.state = state.playView
        self.handleViewStateNotification(action)
    }
}
```

当 view state 中有任意的 PlayViewState.Action 被提交时，上面的闭包就将被调用。togglePlay 方法提交了一个 PlayViewState.Action.togglePlay，PlayViewController 中的 handleViewStateNotification 方法里对应的 case 将会处理这个通知：

```
// PlayViewController
func handleViewStateNotification(_ action: PlayViewState.Action?) {
    switch action {
    case .togglePlay?:
        if let playState = state.playState {
            updatePlayState(playState)
        }
        audioPlayer?.togglePlay()
    // ...
    }
}
```

步骤 4: View Controller 更新 View

在 updatePlayState 方法中，我们最终可以对 view 层级进行更新：

```
// PlayViewController
func updatePlayState(_ playState: PlayState) {
    progressLabel?.text = timeString(playState.progress)
    durationLabel?.text = timeString(playState.duration)
}
```



```

progressSlider?.maximumValue = Float(playState.duration)
progressSlider?.value = Float(playState.progress)
playButton?.setTitle(
    playState.isPlaying ? .pause : (playState.progress > 0 ? .resume : .play),
    for: .normal)
}

```

示例 2: 推入文件夹 View Controller

步骤 1: Table View 调用 Delegate 方法

在 MVC+VS 中，点击 table view 的 cell 并不会像 MVC 那样直接触发一个和 storyboard 相连的 segue。我们需要使用 table view 的 delegate 方法来接收选择事件。

步骤 2: View Controller 更改 View State

在这个 delegate 方法中，我们通过在 view state 中推入被选择的文件夹来进行更改：

```

// FolderViewController
override func tableView(_ tableView: UITableView,
    didSelectRowAt indexPath: IndexPath)
{
    guard let item = itemAtIndexPath(indexPath) else { return }
    switch item {
    case .folder(let f): context.viewStateStore.pushFolder(f.uuid)
    // ...
    }
}

```

View state store 中的 pushFolder 实现非常直接：

```

// ViewStateStore
func pushFolder(_ uuid: UUID) {
    content.folderViews.append(FolderViewState(uuid: uuid))
    commitAction(SplitViewState.Action.pushFolderView)
}

```

View controller 的 view state 代表了需要推入的文件夹，它将文件夹的 UUID 添加到 split view state 的 folderViews 数组中。由于 split view controller 是包含了文件夹列表的 navigation controller 的父级 controller，所以这次 pushFolder 调用会触发一个 SplitViewState.Action 通知。

步骤 3: View Controller 观察 View State 变更

在初始构建 split view controller 时，我们通过调用 synchronizeMasterNavigationController 添加过观察者，并用它来观察这些 action：

```
// SplitViewController
override func viewDidLoad() {
    // ...
    observations += context.viewStateStore.addObserver(
        actionType: SplitViewState.Action.self) { [unowned self] state, action in
        self.state = state
        self.synchronizeMasterNavigationController(action)
        self.synchronizeDetail(action)
        self.synchronizeModalAlerts(action)
    }
}
```

步骤 4: View Controller 更新 View

在 synchronizeMasterNavigationController 方法中，我们对 view state 的 action 进行判断，并且通过配置并将新的文件夹 view controller 推入导航栈的方式，来对 pushFolderView action 进行响应：

```
// SplitViewController
func synchronizeMasterNavigationController(_ action: SplitViewState.Action?) {
    switch action {
    case .pushFolderView?:
        let fvc = storyboard!.instantiateViewController(
            withIdentifier: .folderIdentifier) as! FolderViewController
        fvc.context = context.folderContext(index: state.folderViews.endIndex - 1)
        masterViewNavigationController.pushViewController(fvc, animated: true)
    // ...
    }
}
```

```
}
```

新的文件夹 `view controller` 现在可以自行找到它需要显示的文件夹了：它会检查自身在导航堆栈中的位置，并查看 `view state`，来确定要显示的文件夹的 UUID (对 `context.viewStateStore.pushFolder` 的调用可以确保这个 UUID 是有效的)。

测试

View State 驱动集成测试

测试 MVC+VS app 的核心方式是集成测试。名义上来说，它和我们在基本 MVC 项目中使用的是同一种方式，但是在实现上却大有不同。

在基本 MVC 项目中，我们构建了一个新窗口，并在它的顶部创建了一个完整的 `view controller` 树和 `view` 的树，然后将这些新的树作为单个集成单元进行测试，并与我们的预期进行对比。在 MVC+VS 中，我们只需要构建根 `view controller`，并将文档 `model` 和 `view state model` 作为 `context` 提供给它。`View controller` 和 `view` 的树将作为 `view state` 的响应被自行创建出来。不论 `view` 层级多么庞大或者复杂，构建过程总是遵循相同的形式：

```
// FolderViewControllerTests
```

```
override func setUp() {  
    super.setUp()  
    documentStore = constructTestingStore()  
    viewStateStore = ViewStateStore()  
  
    let storyboard = UIStoryboard(name: "Main", bundle: Bundle.main)  
    splitViewController = storyboard.instantiateInitialViewController()  
    as! SplitViewController  
    splitViewController.context = StoreContext(documentStore: documentStore,  
        viewStateStore: viewStateStore)  
    splitViewController.loadViewIfNeeded()  
}
```

我们创建两个 `store`，从 `storyboard` 初始化根 `view controller`，设置 `context`，然后加载 `view`。

上面使用了一个默认构建的 `ViewStateStore`，这样我们将会得到一个 app 的默认配置，不过我们也可以构建我们所需要的任意的 `view state`，并使用它来构建 app 的任意可能的配置。比

如，下面这个 view state 中，导航栈上加载了根文件夹和一个子文件夹，在 play view controller 中选择一个条目，并且显示了一个文本弹窗来在子文件夹中创建新的文件夹：

```
let content = SplitViewState(
  folderViews: [
    FolderViewState(uuid: DocumentStore.rootFolderUUID),
    FolderViewState(uuid: uuid1)
  ],
  playView: PlayViewState(uuid: uuid2, playState:
    PlayState(isPlaying: false, progress: 0, duration: 10)),
  textAlert: TextAlertState(text: "Text", parentUUID: uuid2, recordingUUID: nil)
)
viewStateStore.reloadAndNotify(jsonData: try! JSONEncoder().encode(content))
```

这描述了整个 view state，reloadAndNotify 会通知 view controller 层级需要从头来重新构建自己，以满足新的状态。这是 view state 驱动编程的长处：我们可以简单地通过提供文档 state 和 view state 作为根 view controller 的输入，以特定的状态构建出 app 的一个新的实例。

如果我们想要验证对于给定的 view state 和文档 state，view controller 和 view 层级可以被正确构建的话，剩下的测试代码就和 MVC 中的集成测试非常相似了：

```
// FolderViewControllerTests
func testRootTableViewLayout() {
  guard let navController = splitViewController.viewControllers.first
    as? UINavigationController else { XCTFail(); return }
  guard let rootFolderViewController = navController.topViewController
    as? FolderViewController else { XCTFail(); return }
  // ...
  let firstCell = rootFolderViewController.tableView(
    rootFolderViewController.tableView,
    cellForRowAt: IndexPath(row: 0, section: 0))
  XCTAssertEqual(firstCell.textLabel!.text, "👶 Child 1")
  // ...
}
```

前三行代码从 split view controller 开始向下遍历，直到根文件夹 view controller。这是新的代码：在 MVC 中，因为它们是手动被构建的，我们会直接拥有 view controller 的引用。而在 MVC+VS 里，则需要遍历这些自动生成的层级来找到引用。不过，接下来从层级中读取和确认符合预期结构的代码就完全一样了。

当我们研究对变化的响应时，事情就开始变得和 MVC 中的测试有很大不同了。在 MVC 中，我们需要等待动画和其他异步更改完成，然后才能检查 view controller 或 view 的树。这使得测试难以编写，也难以保持可靠。对那些主要影响 view state 的 view action 进行的测试来说，尤为如此。

在 MVC+VS 中，我们可以将“对于特定 view action 的响应所造成的 view state 更改”的测试，与“view 层级中的更新”的测试分离开来。更具体来说，我们可以测试 view state 来确保某个 view action 确实对数据进行了合适的变更(这是 **view action 到 view state 的路径**)，而不需要等待 view action 执行动画并且最终影响 view 树。如果我们想要测试 view 树可以被正确构建并反映 view state 数据的变更，我们可以为它编写单独的测试(这是 **view state 到 view 的路径**)。

我们已经展示过如何测试 **view state 到 view** 这条路径了。前面的 testRootTableViewLayout 对基于特定 view state 来进行构建的 root view controller 进行了测试。就像之前例子中的那样，对 view state 进行替换，我们就可以在程序中测试任意的 view 了。另外，我们还可以通过使用初始的 view state 设置 view，然后改变 view state，并测试 view 的结果是否与预期一致的方式，来对 view state 之间的变化进行测试。

剩下的就是测试 **view action 到 view state** 这条路径了。下面的例子对选择并导航至一个新文件夹进行了测试：

```
// FolderViewControllerTests
```

```
func testSelectedFolder() throws {  
    guard let navController = splitViewController.viewControllers.first  
        as? UINavigationController else { XCTFail(); return }  
    guard let rootFolderViewController = navController.topViewController  
        as? FolderViewController else { XCTFail(); return }  
  
    // 检查初始条件  
    XCTAssertEqual(viewStateStore.content.folderViews.count, 1)  
    // 执行变更  
    rootFolderViewController.tableView(rootFolderViewController.tableView,  
        didSelectRowAt: IndexPath(row: 0, section: 0))  
  
    // 检查最终结果  
    guard viewStateStore.content.folderViews.count == 2 else { XCTFail(); return }  
    XCTAssertEqual(viewStateStore.content.folderViews[1].folderUUID, uuid1)  
}
```

同样地，测试从三个 `guard` 语句开始，它们负责获取根文件夹 `view controller` 的引用。接着我们检查前置状态，触发 `view controller` 上的 `action` 执行变更，然后检查 `view state store` 中的结果。想要在 MVC 中实现同样的测试，我们需要将自身设置为 `navigation controller` 的 `delegate`，调用 `performSegue`，使用 `XCTestExpectation` 等待 `navigation controller` `delegate` 方法告知我们转场已经完成，然后再从 `view controller` 层级中读取变更的部分。

在 MVC+VS 里，通过从 `view state` 中读取值，我们不再需要 `delegate`，也不用进行异步等待，还可以摆脱那些难以编写和维护的代码。我们不再被动画所限制。而且在 **view state 到 view** 这条路径上，当我们从 `view controller` 或者 `view` 层级中进行读取时，所涉及的 `view controller` 和 `view` 都没有被实际添加到窗口上。这带来的好处是，测试不会被像是设备方向或者屏幕尺寸这样的屏幕相关的事情影响，进而出现奇怪的问题。

View State Store 的接口测试？

`View state store` 表现出的特性其实就是第二个 `model` 对象。看起来可能像 MVVM 的 `view-model` 那样，使用接口测试的方式会很不错。然而，`view state` 中并不会含有任何重要的逻辑或者行为，所以实际上它并不是一组可测试的接口。`View state store` 上的 `action` 一般都是简单的 `setter`，或者执行一些 `push` 和 `pop` 这样简单的语句。对于这种简单语义的 `action` 的测试，看起来可能是这样的：

```
func testMyAction() {
    XCTAssert(myStore.myValue == someInitialValue)
    myStore.myValue = newValue
    XCTAssert(myStore.myValue == newValue)
}
```

除非 `myValue` 的自定义 `getter` 或者 `setter` 非常复杂，否则这个测试是没什么用的，因为这里没有对我们的程序中的任何逻辑进行测试。

这是 `view state store` 和 `view-model` 的最本质的区别：`view-model` 是针对 `model` 对象的变形管道，而 `view state store` 仅仅只是一个存储了简单值的可观察量。在 MVC+VS 中，显示和交互这样的 app 逻辑，依然是写在 `view controller` 里的。所以还是需要对 `view controller` 进行自动化测试。对于自动化测试，MVC+VS 应当采用和 MVC 中所使用的集成测试相同的方式进行。

状态日志

由于 view state 很少会包含重要的逻辑，所以它很适合被用于其他目的，比如说序列化。默认情况下，MVC+VS 的 app 可以将 view state 输出到控制台：

Restored ViewState to:

```
{
  "folderViews" : [
    {
      "folderUUID" : "00000000-0000-0000-0000-000000000000",
      "editing" : false,
      "scrollOffset" : 0
    },
    {
      "folderUUID" : "6950BA0E-74F5-4034-9B9F-3D3FEB48B939",
      "editing" : false,
      "scrollOffset" : 0
    }
  ],
  "playView" : {
  }
}
```

上面的日志信息表明了当 app 被打开时，它在导航栈上恢复了两个文件夹 (根文件夹的 UUID 是全零)，play view controller 中则没有东西被选中。

这种类型的信息在构建测试例时非常有用，而且在开发中，当因为设置了错误的状态，或者是对状态的响应不正确时，我们可以通过日志快速理解问题所在。明确的日志还能使那些难以观察的数据中存在的问题显现出来。

日志的关键的优势在于，它可以在我们意识到需要它**之前**就处于激活状态。如果我们无法理解为什么前一个操作行为无法工作，我们可以在 app 还在运行时就快速浏览日志来理解问题所在，而此时调试器也还处于活跃状态。

时间旅行调试

用户界面的时间旅行，指的是对于每个数据驱动的界面变更，我们都可以将所有数据进行快照，并且在随后重放这些数据，来将用户界面恢复到一个之前的状态。这和撤销和重做操作很类似，不过撤销和重做只对传统的 `model` 数据起作用，而时间旅行还会影响像是导航，选择，滑动位置，对话框以及其他更多 `view state` 相关的属性。

在大多数情况下，时间旅行是很可爱的小把戏，和撤销重做不同，时间旅行对用户来说没什么用。但是，在开发阶段，时间旅行可以帮助我们快速验证一个 `app` 是否能干净利索地响应状态恢复，以及所有的 `view state` 是否正确地被 `view state store` 所捕获。

在 `app delegate` 的 `application(_:didFinishLaunchingWithOptions:)` 函数中，我们会看到如下代码：

```
func application(_ application: UIApplication, didFinishLaunchingWithOptions
    launchOptions: [UIApplicationLaunchOptionsKey: Any]?) -> Bool
{
    // ...
    #if false
        DispatchQueue.main.async {
            // 等待主窗口的显示，然后在其上方创建一个新窗口
            self.historyViewController =
                HistoryViewController(nibName: nil, bundle: nil)
        }
    #endif
    return true
}
```

将 `##if false` 改为 `##if true`，时间旅行将被启用。在 `app` 运行的时候，在底部会出现一个滑动条。前后拖动这个滑动条，就可以在 `app` 的所有之前的状态间进行跳转。

注意，我们故意将播放状态和录音状态从时间旅行的历史中去除了，因为它们会在我们尝试拖动滑动条时连续地发出新的状态，这会造成一些问题。

讨论

像本章所描述的这样来处理 `view state` 的变更，比起 `MVC` 中处理简单的 `view state` 变更来说，需要多编写不少代码。之前直接进行的变更，现在被强制要求经由 `view state model` 接口来进

行。好处在于我们可以为我们 app 中的 view state 变更的通讯创建一致的机制。通过在全局 view state 上强制使用观察者模式，结果和 UI 不同步的错误就很不容易再出现了。

原本发生在 view controller 之间的通讯，现在可以通过 view state 来进行。比如，如果我们想要在文件夹 view 的下方添加一个迷你播放器的话，我们可以使用同样的 play state，而不需要在已有的 player view controller 和新增加的迷你播放器之间设置通讯的渠道。View state 被用作唯一的事实来源，两个播放器都可以对它进行订阅。

还有一些其他细微的好处：

- 用户界面的状态恢复不需要额外的代码，而 MVC 需要对每个 view controller 实现编码和解码。
- 我们可以通过 view state 的日志 (将 view state 转换为 JSON 并且在每次发送通知的时候将其打印出来) 来观察 app 的抽象状态，并且调试问题。
- 我们不需要依赖 UIStateRestoring 就可以恢复 app 的状态，这样我们能反复地对 app 的某个特定部分进行调试 (我们在上面的[时间旅行调试](#)一节中描述过这种方法)。
- View state model 为导航和数据流操作提供了一个地方，来为它们添加抽象层，这可以让我们更容易增加特性或者管理复杂度。

追踪 View State 的注意事项

UIKit 的设计中，并没有保证精准报告 view state 的变更。我们并不是每次都能在 UIKit 更新 view 层级之前捕获到 view state 的变化，我们有时候需要更新我们的状态，来使它符合 UIKit 已经作出的变更。在 UIKit 没有能清晰汇报到底发生了什么操作的情况下，这会变得更加复杂。

也许关于这点最困难的例子就是对 navigation controller 的导航栏上的返回按钮进行了处理，特别是当这个 navigation controller 是在一个 split view controller 的 master 侧时 (就像录音 app 中的文件夹 navigation controller 这样)。对于大多数按钮，我们可以截获 action，将这个 action 转发给 view state model，然后只在对它所发出的通知进行响应的时候，才去更改 view 层级。不幸的是，UIKit 不允许我们自定义导航栏上返回按钮的行为，而且在任何一个可以被截取的 action 被触发的时候，navigation controller 的栈已经被更改了，关于发生了什么，UIKit 只提供了很有限的信息。

下面的方法是录音 app 中 split view controller 里的实现，它通过监听 navigation controller 的 delegate 方法，在返回按钮被按下，或者 dismiss 时对 detail view 进行折叠的时候，去对 view state 进行更新。但是，这些方法也将会在 view state 变更的响应中，对 view 层级进行改

变后被调用(比如,当我们推入一个新的 view controller,或者在进行状态恢复时)。我们只有在 action 的来源不是 view state 的情况下,才需要对 view state 进行更新和处理。

```
func navigationController(_ navigationController: UINavigationController,
    didShow viewController: UIViewController, animated: Bool) {
    guard animated else { return }

    let detailOnTop = navigationController.viewControllers.last
    is UINavigationController
    if self.isCollapsed && !detailOnTop {
        // Detail view 被 dismiss, 清除关联的状态
        context.viewStateStore.setPlaySelection(nil, alreadyApplied: true)
    }

    let newDepth = navigationController.viewControllers.count
        - (detailOnTop ? 1 : 0)
    if newDepth < state.folderViews.count {
        // 处理导航栏的返回按钮
        context.viewStateStore.popToNewDepth(newDepth, alreadyApplied: true)
    }
}
```

这种处理非常烦人,不过正如我们所指出的,这大概是我们在 UIKit 中处理 view state 时所会碰到的最困难的情况之一了。TEA 和其他一些拥有声明式 view 层的模式,也会遇到同样地问题:UIKit 的 view state 变更是独立于我们的 view state 或者 model 的。

经验和教训

数据的依赖注入

本书中的每一个 app 的实现都使用了不同的方式来连接每一个 view controller 和它的数据。基本的 MVC 实现中,我们将一个 model 对象设置给了每个 view controller,但是这要求 view controller 能理解 view controller 层级,才能与某个 view controller 或者单例进行对话。MVVM 的实现使用了协调器来提供上下文。MAVB 在一个数据依赖图中,使用响应式编程绑定的方式,来将所有的 view 连接起来。TEA 则通过一个可以自动为函数提供所需数据的 driver,来让所有东西正常运行。

本章中的实现里，我们在每个 view controller 上都使用了一个 context 属性，并要求在 view controller 构建的时候传入这个上下文属性。虽然和基础 MVC 中设定一个 model 对象相比没有特别大的不同，但是它还是展示了一种避免单例的方式。

我们最初在做 MVC+VS 项目的实现的时候，没有使用这个 context 属性。相反，我们让 view controller 直接访问单例版本的文档 store 和 view state store。对于基础的 app 来说，没有什么大问题，但是由于 view 无法从这些单例环境中解耦，所以我们就无法为 app 编写测试了。

确保在 viewDidLoad 之前就已经正确设置了上下文对象，会比使用单例要求更多的工作，但是这是一种更加灵活和简单的方式，我们可以将它添加到包括 MVC 和 MVVM 等任意基于 controller 的模式中去。

全局 View State 通讯

本章中所展示的 MVC+VS 模式有一个前提，那就是作为 model 的一部分的 view state，可以完整表示 UIKit 中的可能范围：整个 app 的 view state 将被 ViewStateStore 所表示。将文档 store 和 view state store 进行组合，就可以对整个 app 进行权威表示，当我们想要更改 view 时，我们对相关的 store 进行更改，然后让 view 以响应式的方式进行更新。

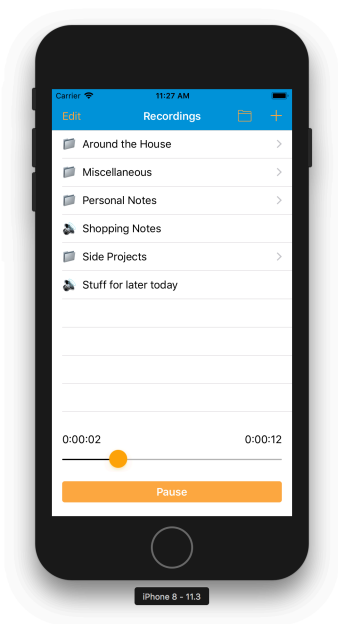
这个思想可以应用在一个更小的范围里，来帮助 MVC 的代码库解决两个常见问题：

1. 在 view controller 之间进行 view state 的通讯。
2. 在单个 view controller 中简化 view state 的本地逻辑。

在本节中，我们来看一个如何解决第一个问题的例子。在 下一节里，我们再研究如何解决第二个问题。

在一般的 iOS app 中，因为 app 中没有其他部分会依赖这些 view state 信息，大部分的 view state 会被本地管理，这是由 view 本身或者是负责展示的 view controller 来完成的。但是，大部分的 app 也有一些必须要在 view controller 之间进行通讯的 view state。这里，我们就可以从 MVC+VS 的 view state store 中进行一些借鉴。

举例来说，和 Apple Music 或者 Spotify 类似，我们想要在屏幕底部集成一个迷你播放器 (这部分内容也在随书的视频中进行了完整的阐释)。这个场景立刻带来了一个问题：在添加迷你播放器之前，因为 app 中没有其他部分依赖播放状态，所以它可以在 PlayViewController 中本地处理。现在要添加一个迷你播放器，我们需要在 detail 侧的 PlayViewController 和迷你播放器的 PlayViewController 之间共享这个播放状态：



和 MVC+VS 把所有 view state 存放到一个 view state store 中类似，我们可以只将音频播放器和相关的播放属性提升为一个全局共享的、可以通过通知观察的播放器类。这个播放器和关联的播放状态的行为将类似于我们的 model。如果我们之后还需要处理除了播放状态以外的其他全局 view state，我们可以将这个 SharedPlayer 类转变为一个更加通用的全局 view state store。

作为开始，我们将播放器和播放状态相关的代码从 play view controller 中拿出来，形成一个共享的播放器类：

```
final class SharedPlayer {
    var duration: TimeInterval { return audioPlayer?.duration ?? 0 }
    var progress: TimeInterval { return audioPlayer?.currentTime ?? 0 }
    var audioPlayer: Player?
    var recording: Recording? {
        didSet { updateForChangedRecording() }
    }
    // ...
}
```

每当播放状态或者当前的录音被改变时，我们发出一个通知。Detail 播放器和迷你播放器的 view controller 都将观察这个通知：

```
final class SharedPlayer {
    // ...

    static let notificationName =
        Notification.Name(rawValue: "io.objc.SharedPlayerStateChanged")
    func notify() {
        NotificationCenter.default.post(name: SharedPlayer.notificationName,
            object: self)
    }

    func updateForChangedRecording() {
        if let r = recording, let url = r.fileURL {
            audioPlayer = Player(url: url) { [weak self] time in
                self?.notify()
            }
            notify()
        } else {
            audioPlayer = nil
            notify()
        }
    }
}
```

updateForChangedRecording 方法必须在三个不同的分支调用 notify：在追踪播放进程的播放器回调中，在 audio player 初始化完成后，以及在我们没有录音的情况下。这里的逻辑可以通过改进底层 Player 类的 API 进一步简化，不过现在来说，这样就够了。

PlayViewController 直接从 shared player 获取当前的状态，然后订阅接下来的更新的通知。这和 MVC 中的文档 store 里的观察者模式是一样的：

```
class PlayViewController: UIViewController, UITextFieldDelegate {
    // ...

    override func viewDidLoad() {
        // ...
        updateDisplay()
        NotificationCenter.default.addObserver(self,
            selector: #selector(playerChanged(notification:)),
```

```

        name: SharedPlayer.notificationName, object: nil)
    }

    @objc func playerChanged(notification: Notification) {
        updateDisplay()
    }

    func updateDisplay() {
        updateControls() // 更新时间标签和播放控件
        if let r = sharedPlayer.recording {
            // ... 显示控件, 设置导航栏标题
        } else {
            // ... 隐藏控件, 显示无录音的标签
        }
    }
    // ...
}

```

通过在观察开始的时候立即发送初始值的方式 (就像我们在 MVC+VS store 上做的那样), 这些代码可以得到改善, 不过为了简明我们将跳过这个部分。

新的 model 包含了我们之前当作 view state 进行处理的那些状态, 对于这个新 model 的观察, 不论屏幕上有多少个可见的播放器, 它们的 view 都可以和播放器的状态同步。当我们需要在多个部件之间共享 view state (比如选择, 网络活动, 或者用户 session 的状态) 时, 我们都可以使用相同的模式来处理。

对上面的 shared player, 我们使用了单例, 因为要是从所有依赖的 view controller 的共通父级 view controller 开始, 再将这个共享的 view state 实例分发传递下去, 会显得有些累赘。不过, 如果你使用了 (我们在 [MVVM-C 一章](#)中所介绍过的) 协调器模式的话, 在一个 view controller 的属性上设置共享的 view state, 或者将它传递到 view controller 的初始化方法中, 就都是轻而易举的事情了。

简化本地 View State

在前面的例子中, 我们借用了 MVC+VS 中的思想, 为多个 view controller 之间进行 view state 通讯创建了一套稳定的机制。然而, 我们还可以把一个简化版的 MVC+VS 应用到 MVC

的单个 view controller 中，通过将状态属性合并为单个的结构体或者枚举属性，并对这个属性进行观察，来在该属性变更的时候对 view 进行更新。

在 play view controller 中，有不少相关的属性是需要同时设定的。每当录音发生改变时，我们需要设置导航的标题，文本框的文本，进度条，以及录音持续时间。因为这些属性是一起的，我们可以将它们集束在一个结构体中：

```
struct DisplayState {  
    var name: String  
    var playState: Player.PlayState  
    var progress: TimeInterval  
    var duration: TimeInterval  
}
```

现在我们可以我们的 view controller 中添加一个持有 DisplayState 的属性。每当它改变时，我们调用 updateViews。注意，这个属性是可选值，要么我们是没有选择录音，要么一个录音和它的所有属性都应该是可用的：

```
var displayState: DisplayState? = nil {  
    didSet {  
        updateViews()  
    }  
}
```

didSet 属性观察是一种简单的方式，用来允许某个可能在多处被改变的变量拥有单个观察者。在这里，displayState 就是这样一个变量。

当我们在单个的 view controller 中有完全不同的两种状态时，我们还可以考虑将这个 view controller 分为两个不同的 view controller：在其中一个 view controller 中，我们可能会显示“无录音”，而在另一个里我们只能用录音和一个播放器来进行配置（这样就不会有可选值属性了）。

updateViews 会根据 displayState 来对 view 层级做相应的变更：

```
private func updateViews() {  
    activeItemElements?.isHidden = displayState == nil  
    noRecordingLabel?.isHidden = displayState != nil  
    title = displayState?.name ?? ""  
}
```

```
nameTextField?.text = displayState?.name ?? ""  
updateProgressDisplays()  
}
```

在最初的 MVC app 中，我们在四个不同的地方对 view controller 的标题进行了设置：storyboard 里进行了一次，updateForChangedRecording 里进行了两次，textFieldDidEndEditing 里进行了一次。现在，我们不再去更改标题，而是更改 displayState 属性就可以了。起初，这看起来好像只是添加了一层间接操作，但是它确实简化了关于这个类的逻辑：要理解所有的 view 是否被正确配置，我们只需要阅读 updateViews 方法。当我们想要改变标题的显示方式时，只需要在一个地方进行修改，而不是在四个地方进行。displayState 的工作方式和 model 值类似：所有通过这个属性进行的变更，都会被一个 didSet 属性观察者进行观察。

这种方式的缺点在于，我们可能执行了额外的操作。现在，当我们只是想要更改录音的名字时，我们还是需要创建新的 DisplayState 值。设置 displayState 属性将会对进度条显示这类无关的属性也进行更新。我们可以尝试查找哪些属性发生了变更，或者像在 MVC+VS 中这样，将发生的 action 包含在通知中。不过这些选项都是在额外开销与正确性之间进行的权衡。根据你的使用场景，你可能会希望向其中的一方进行倾斜。

Model 适配器-View 绑定器

7

本章介绍一种还处于实验阶段的架构，**model 适配器 - view 绑定器** (ModelAdapter-ViewBinder, MAVB)。这种架构尝试解决 MVC 中的一些陷阱，但是它同时也承认 app 从根本上还是要处理可变的 view 层级和可变的 model。MAVB 将可变状态隔离开来，并且在 model 和 view 之间使用清晰定义的观察路径。它还为程序不同部分之间通讯提供了统一的路径：所有东西都通过单向的响应式绑定进行连接。

MAVB 一开始的目的是为了避免在 MVC 中对 view controller 创建子类。它的想法是，我们不去创建 view controller 的子类或者重写里面的方法，而是向 view controller 的构建方法中传入一个闭包，用它描述要重写的方法和需要实现的 delegate 方法。如果能做到这样的话，那我们为什么不通过构建时的参数把 view controller 的所有状态也一并进行配置呢？接下来，如果我们配置了当前的状态，那我们为什么不使用响应式编程来把**未来**的状态也导入到 view controller 里呢？

事实证明，将所有的属性，行为和观察都在 controller 构建的时候传递进去是可行的。这让我们可以完全避免创建 view controller 的子类。实际上来说，我们将一长串规则列表传给 controller，这些规则定义在未来某个特定事件发生时 controller 应该发生什么行为。换句话说，我们通过**声明式**的系统来提供它。

当我们这么做时，会遇到两个有趣的事实：

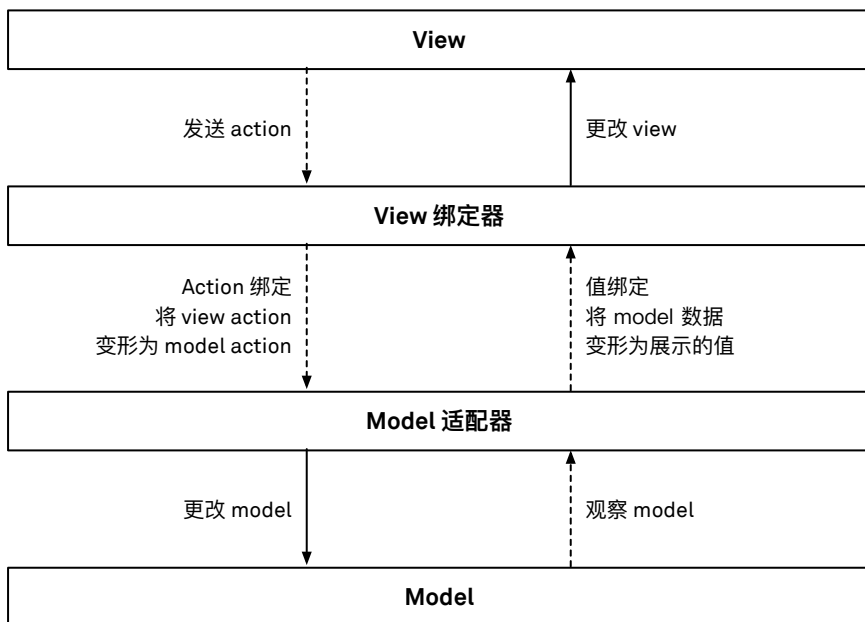
1. 如果我们在构建时将所有行为注入到对象里的话，这个对象就可以自我管理，而不会需要再被访问了。事实上，再次访问这个对象并不是什么好主意，因为这样做我们只会和自我管理的行为发生冲突。
2. View controller 的存在是为了服务 view，而不是它自己。如果我们要做行为注入，我们应该完全绕过 controller，并且将行为注入到我们的 view 中。

这也是 CwlViews 项目的起源。CwlViews 这个框架可以用来在 view 构建时就对所有的属性和行为进行指定，这样在 view 构建完成后，就不再需要访问它了。CwlViews 将 MVC 中 controller 的架构角色**排除掉**。本章中所讨论的 MAVB 模式描述的就是在 CwlViews 框架所创建的条件下，如何编写 app 的方式。

在 MAVB 中，view 层是通过 **view 绑定器**来构建的。View 绑定器是 CwlViews 提供的用来构建 view 的接口，它允许我们在 view 创建时将它们所有的属性和行为进行注入。View 绑定器是一个列表，它包含了闭包，响应式的绑定，以及其他描述 view 行为的属性。我们将这个列表里的东西统称为**绑定**，view 绑定器将根据这个绑定列表，自动按需创建下面的 UIKit view 对象，对它们的所有属性进行配置，然后在其生命周期中自动管理它的所有行为。

Model 适配器是对状态对象的封装，它强制所有对于状态对象的访问都要通过响应式编程来实现。所有对内部对象的变更都必须通过 model 适配器的**信号输入** (我们可以将响应式编程的

observable 绑定到这些属性上) 来完成。相应地，从内部对象中读取任何值和改变的操作，也需要通过 model 适配器的输出**信号** (响应式编程的 observable) 来完成。



其他的模式基本都会把它们的所有逻辑都放到某个层中去，而小一些的 MAVB 模式的 app 会把它们的很大一部分 app 逻辑放在 view 绑定器和 model 适配器层之间，作为响应式编程对于绑定所进行的变形。也就是说，上面的图表中，大部分的逻辑存在于箭头中，而不是方块里。

探索实现

CwlViews 和 View 绑定器快速入门

在我们深入这个设计模式其他部分之前，我们先来快速看看 view 绑定器的构建语法。这是这个模式的核心部分，而且它的语法有一些特别。

如果我们想要一个设置为文本常量值的 UILabel，我们可以使用一个不变量作为 .text 绑定，来构建一个 Label 的 view 绑定器：

```
Label(.text -- "My text label")
```

我们一般不会通过调用 `uiView()` 来手动让 `view` 绑定器构建它的 `view` (我们一般会把 `view` 绑定器添加到一个父 `view` 中, 这个方法会被自动调用), 但是如果我们这么做的话, `view` 绑定器会构建一个 `UILabel`, 并且将它的 `text` 值设置为 “My text label”。

这里 `.text` 是**绑定名**, `--` 是**静态绑定**操作符 (它表示在这个 `view` 的生命周期中, 这个 `text` 值将不会改变), “My text label” 是**绑定参数**。我们可以将这个初始化方式等效于下面这样的标准 Swift 参数语法:

```
Label(text: "My text label")
```

不过, 使用 `.text` 绑定名和一个自定义的操作符, 我们就具有了改用其他操作符的灵活性, 比如像下面这样可以接受一个可变值, 而非数值:

```
Label(.text <-- signalEmittingStrings)
```

`<--` 操作符是 `CwlViews` 中的**值绑定**操作符。它意味着绑定参数所描述的是以信号的形式表示的可变值, 从绑定参数到底层属性的数据流, 通过绑定名进行区分。

在 [MVVM 一章](#)中, 我们使用了 `RxSwift` 来实现响应式编程绑定, 不过 `CwlViews` 框架是依赖 `CwlSignal` 来提供响应式编程绑定的。在大多数情况下, 这个区别并不要紧, 但是要记住, 在 `CwlSignal` 中, “信号” (signal) 这个词在 `RxSwift` 中的同义词是 “observable”, “信号输入” (signal input) 的概念和 `RxSwift` 中的 “subject” 相似 (subject 是位于响应式管道的起始位置的类型, 我们可以向其中发送值)。

我们使用第三种和最后一种绑定操作符, `-->`, 来将信号输入用做绑定参数:

```
TextField(.textDidChange --> inputThatAcceptsStrings)
```

当文本框的文本改变时, 变更通知将会被发送给作为绑定参数的信号输入。

```
Label(.text <-- playState.mediaState.map { timeString($0.progress) })
```

上面一行摘自录音 app 中的代码, 创建了一个 `label`, 用来显示播放器的 `progress`。`playState.mediaState` 是一个 `model` 适配器, 它包含了从播放器发出的最新的状态。直接在 `model` 适配器的默认输出信号进行变形。闭包接受从信号发出的每一个媒体状态结构体中的 `progress` 这个 `time interval` 值, 并将它转变为一个可读的字符串。这个结果字符串的信号将被用来为标签提供文本值。

将 action 从 view 绑定器连接到 model 适配器的方式很相似：

```
BarItem(
    .barButtonItem -- .add,
    .action --> Input().map { _ in .record(inParent: folder.uuid) }.bind(to: store)
),
```

录音 app 中的这一行，表示导航栏的“+”按钮的 action 将会开始一个新的录音。Input() 初始化方法将创建一个新的输入通道 (由一对信号输入和信号对象组成)。map 函数在通道末端的信号追加了一次变形。这个变形闭包忽略按钮被按下时所发出的 void 参数，并返回一个 .record(inParent:) model action 消息。在通道最后，信号被绑定到 store 的默认信号输入上。通道的绑定将消耗信号端，所以这里的返回值是通道末端的输入。最后一步，--> 操作符将输入端和 .action 绑定名合并到一起。

MAVB 中，场景的构建是通过将 view state 的定义与场景中 view controller 和 child view 的构建进行配对来完成的。下面的这个小例子展示了所有常见的特性：

```
struct TextState: StateContainer {
    let value: Var<String>
    init () {
        value = Var("")
    }
    var childValues: [StateContainer] { return [value] }
}

func textFieldController(_ text: TextState) -> ViewControllerConvertible {
    return ViewController(
        .view -- View(
            .layout -- .vertical(
                .view(TextField(
                    .text <-- text.value,
                    .didChange --> Input()
                        .map { content in content.text }
                        .bind(to: text.value)
                )),
            .space(.fillRemaining)
        )
    )
}
```

```
}
```

在上面代码例子中，有两个顶层定义：顶部的 view state 对象 `TextState`，以及底部的 view 绑定器的构造函数 `textFieldController`。

`TextState` 包含一个 `Var<String>`。Var 是 `CwlViews` 库所提供的的一个泛型的 model 适配器类型，它提供了基础的设置/通知行为，并且参与了 view state 的持久化操作。`TextState` 还遵守 `StateContainer` 协议，所以它可以是 view state 树的一部分。我们重写了 `childValues` 的默认实现，让它返回 `[value]`，这将确保 `value` 在 view state 持久化或者 view state 调试的时候被包含在内。

这个场景使用一个 content view 来构建 view controller，content view 包含了一个文本框。代码例子中的 `.vertical`，`.view`，以及 `.space` 并不是绑定名，它们是 content view 布局描述的一部分（一个垂直布局，文本框在顶部，空行填充了 view 的其余部分）。文本框有两个可变绑定：一个是 `.didChange` 绑定，它会确保在文本框内容改变时，对 `text.value` 这个 model 适配器进行更新；另一个 `text` 绑定保证了不论 `text.value` 在程序的任何地方被更新，文本框都会对应地对显示内容进行更新。

构建

和典型的 MVC 相比，MAVB 的启动方式非常不同，这主要有两个原因：

1. 该架构没有 controller 层（没有 `UIApplicationDelegate` 或者 `UIViewController` 子类）。
2. 它不使用 storyboard 或者 nib 文件。

因为典型的 MVC 的启动，通常是由这些组件所决定的，所以将它们拿走，必将导致完全不同的启动流程。

在 MAVB 中，对程序顶层状态进行封装的那些 model 适配器是最先被构建的。`Store` 被封装到一个单独的 model 适配器中。`View state` 是一个 model 适配器的树，这些适配器各自负责封装整个 view 层级中的一个部分。`Split view controller` 的状态是在 view 层级顶部的状态，构建也由此开始：

```
private let store = StoreAdapter(store: Store())  
private let splitVar = Var<SplitState>(SplitState())
```

Store 适配器 (它是一个自定义的 model 适配器) 直接封装底层的 store model 对象。类似地, (CwlViews 库为了实现 view state 的设置和持久化而提供的) Var 对 split 的状态进行封装, 它负责表示 view 层级的抽象状态。

这些对象被传递到一个函数中, 作为参数用来构建 app 的 view 绑定器 (在 MAVB 中, application 对象被当作一个 view 来处理, 这和其他的 Cocoa controller 对象类似):

```
func application(_ splitVar: Var<SplitState>, _ store: StoreAdapter)
-> Application
{
    return Application(
        .window -- Window(
            .rootViewController <-- splitVar.map {
                splitState -> ViewControllerConvertible in
                splitView(splitState, store)
            }
        ),
        .DidEnterBackground --> Input().map { .save }.bind(to: store),
        .willEncodeRestorableState -- { archiver in
            archiver.encodeLatest(from: splitVar)
        },
        .didDecodeRestorableState -- { unarchiver in
            unarchiver.decodeSend(to: splitVar)
        }
    )
}
```

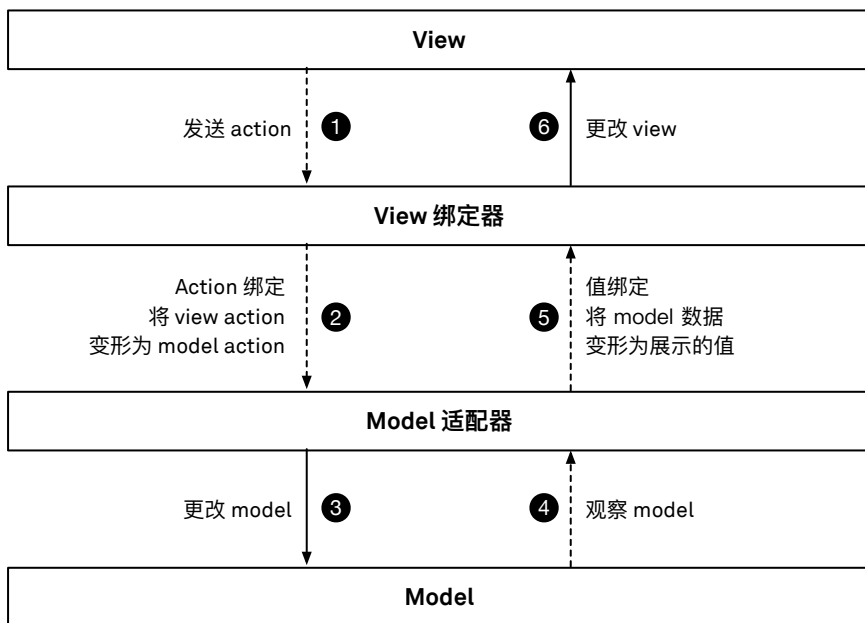
Application view 绑定器通过 .window、.DidEnterBackground、.willEncodeRestorableState 以及 .didEncodeRestorableState 这四个绑定进行配置。

虽然 window 绑定是一个静态绑定, 这个不变性所指的是设置在 application 对象的引用不变, 而不是指 window 的内容不变。当 app 在状态恢复期间调用 .didDecodeRestorableState 时, model 适配器 splitVar 会收到一个变更。在 .rootViewController 绑定中, 我们使用 splitVar.map 观察这个变更, 并将整个 root view controller 替换掉。

更改 Model

将变更运用到底层模型的方式，通常是使用一种称为 **reducer** 的模式来进行的，这是一个签名为 (inout State, Action) -> Notification 的函数，它在自身的隔离上下文中更新状态对象。对于简单一些的 model 适配器，返回的通知 (通常只是一个更新后的 model state 的复制) 将直接通过默认的输出信号进行发送。更复杂的 model 适配器可能需要在发送之前对通知进行进一步的处理。

在这个例子中，我们来看看从一个文件夹中移除某个条目时所需要的步骤：



步骤 1: View Action 传递给绑定器

用户从 table view 中删除 cell，这个 table view 的构建方式如下：

```
.view -- TableView<Item>{  
    // ...  
    .commit --> Input()  
    .compactMap { styleAndRow in styleAndRow.row.data?.uuid }
```



```

        .map { uuid in .removeItem(uuid: uuid) }
        .bind(to: store),
    // ...
)

```

为 commit 指定绑定，会在内部为 table view 创建一个实现了 tableView(_:commit:forRowAt:) 方法的 data source，当 table view 调用该方法时，对被点击的行号的描述，将通过绑定发送出来。

步骤 2: Action 绑定将 View Action 转换为 Model Action

在上面的代码中，compactMap 将 tableView(_:commit:forRowAt:) 方法被调用时所发出的 styleAndRow 转换为该行对应的 UUID。因为 UUID 被一个可选值所包装，所以使用 compactMap 变形来将可选值解包，它只会把非 nil 的结果发送给下一个变形阶段。

下一个阶段是将非 nil 的 UUID 通过 map 变形为一个 .removeItem 的 model action 消息，这个消息随后被发送给 store。

步骤 3: Model 适配器将 Model Action 应用到 Model

StoreAdapter 主要是对内部的 FilteredAdapter 的一层封装。FilteredAdapter 是 CwlViews 中的一个辅助类型，它提供了一个文档 model 适配器中大多数的功能。它包装了底层的状态对象，提供了一个类似 reducer 的闭包，来将消息应用到状态对象上；这个状态可以被观察多次，正如其名，每个观察都带有一个独立的过滤器 (就像底层对象或者被发出的通知流的切片)：

```

struct StoreAdapter: SignalInterface, SignalInputInterface {
    // ...
    init(store s: Store) {
        filteredAdapter = FilteredAdapter(initialState: s) {
            (store: inout Store, message: Message) -> Store.Notification in
                switch message {
                    // ...
                    case .removeItem(let u): return store.removeItem(uuid: u)
                    // ...
                }
        }
    }
}

```

上面展示的是带有内部状态对象 (也就是底层的 store) 的 FilteredAdapter 的初始化方法, 以及将消息应用到状态对象的 reducer 函数。在这个例子里, 我们看到, .removeItem 消息会通过 removeItem(uuid:) 调用来进行处理。

步骤 4: Model 上的 Action 被转变为信号

上面对 store.removeItem(uuid: u) 的调用会立即从 reducer 闭包中返回一个 store 通知。这些返回的通知是通过通知信号从 FilteredAdapter 进行发送的, 但是将信号直接暴露出来的做法并不常见, 使用 FilteredAdapter 的目的在于, 我们想要制造一些针对观察者的需求进行更多自定义的东西。

folderContentsSignal 方法就是一个这种类型的嵌入了 model 的自定义 view 的例子。这个方法被实现在 store 适配器上, 我们给定一个 UUID, 它会输出一个数组变化的串流, 这个流将描述文件夹内容随时间变化的情况, 并经过排序以便显示。

这个实现依赖于 filteredSignal, 这是 filteredAdapter 的主要的过滤后的输出。该函数接收一个可以访问底层 store 的变形函数作为输入, 另外的输入还包括从主 reducer 函数中发出的最近的通知, 以及在可能时用来对更新进行差分比较的状态变量:

```
func folderContentsSignal(_ folderUuid: UUID) -> Signal<ArrayMutation<Item>> {
    return filteredAdapter.filteredSignal(initialValue: []) { (
        items: inout [Item],
        store: Store,
        notification: Store.Notification?,
        next: SignalNext<SetMutation<Item>>>) throws in

        switch notification ?? .reload {
        case .mutation(let m) where m.values.first?.parentUuid == folderUuid:
            next.send(value: m)
        case .reload:
            if let folder = store.content[folderUuid]?.folder {
                next.send(value: .reload(folder.childUuids.compactMap {
                    store.content[$0]
                }))
            } else {
                throw SignalComplete.closed
            }
        default: break
        }
    }
}
```

```

    }.sortedArrayMutation(equate: { $0.uuid == $1.uuid }, compare: { l, r in
        l.name == r.name ? l.uuid.uuidString < r.uuid.uuidString : l.name < r.name
    })
}

```

在上面的代码中，闭包签名的下面的 switch 语句才是重要的部分。这个 switch 语句对 store 发出的两种通知进行了集中处理：

1. .mutation 通知
2. .reload 通知

Store.Notification.mutation 通知对 store 中文件夹里的条目的添加、删除或者更新的变更进行描述。我们对这些通知进行过滤，只关心我们所关注的文件夹的子文件夹里的变更。这些过滤后的变更集合将被不加修改地转发到下一个变形中去。

当我们接收到 Store.Notification.reload 通知时 (在 filteredSignal 第一次被调用，通知为 nil 时，这个通知也会被发出)，我们从 store 中直接获取完整的子文件夹和条目的集合，然后将它包装到 SetMutation.reload 里后再发送出去。

在 filteredSignal 创建的处理阶段完成后，信号变为设置变更后的串流。为了在文件夹 view 中显示，我们想要一个有序的变更后的集合串流，所以我们把结果传递到另一个变形中去，使用 sortedArrayMutation 来完成这次转换。除去其他更改外，这个结果信号还会告诉我们文件夹中移除项目的变更。

步骤 5: 使用值绑定将 Model 适配器事件转换为显示值

上面变更数组所发送的序列，驱动了文件夹 table view 的显示。在我们的 table view 的定义中，我们使用了文件夹内容的信号，并将它绑定到了 table 的数据上：

```

.view -- TableView<Item>{
    // ...
    .tableData <-- store.folderContentsSignal(folder.uuid).tableData(),
    // ...
}

```

tableData 的绑定将从信号中获取变更数组，通过调用 .tableData() 将它们变形为 table row 的变更。当信号发送的是移除通知时，它会被转变为对应的 table 的行的移除变更操作。

步骤 6: 绑定器对 View 进行更改

这一步的实现是在框架内部 (在 view 绑定器中的)

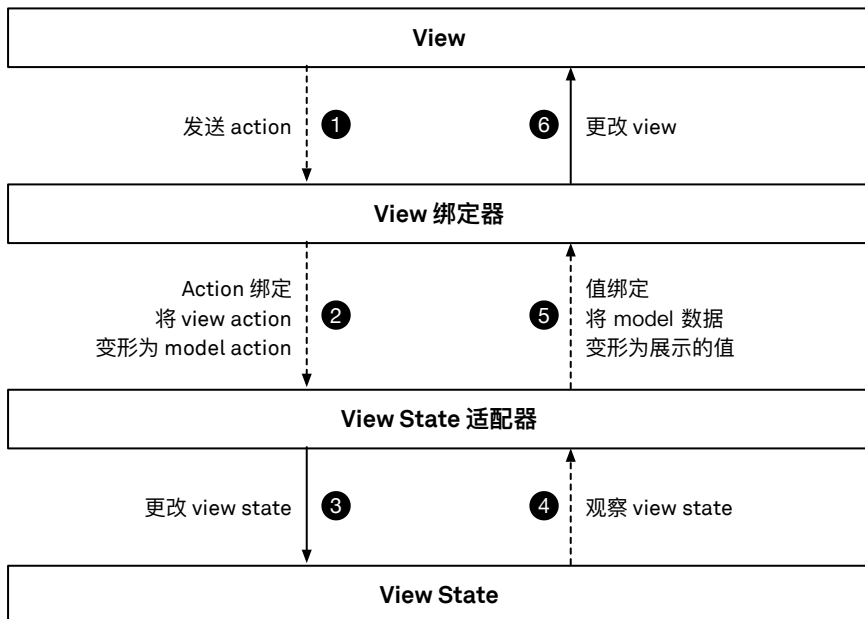
TableView.Storage.applyTableRowMutation 函数里完成的。

每一个通过绑定所接收到的 table 行变更的值, 都将被应用到缓存的 table 数据上去, 变更值的 kind 和 animation 会被观察, 并对应地在 table 上进行动画。在本例中, deleteRows(at:with:) 将被调用, .automatic 动画会作为参数传入。

更改 View State

在本节中, 我们会来看看 MAVB 中更改 view state 的步骤。我们还是使用和前面其他章节中相同的两个例子: 更新播放按钮的标题, 以及将一个新的文件夹 view controller 推入导航栈。

对 view state 的更改和我们上面对 model 的更改一样, 遵循相同的六个步骤。不同之处在于, 我们现在是对 view state 适配器进行操作, 而不是 model 适配器:



View state 适配器和 model 适配器的工作方式基本相似，只不过它包装的是一个 view state 值，而不是 model 对象。

示例 1: 更新播放按钮

当用户点击播放按钮时，按钮的标题应该在“播放”和“暂停”之间切换。在 MAVB 里，这需要相同的六个步骤的两个循环。第一次循环改变 audio player，第二次改变按钮本身。

步骤 1a: 按钮点击被传递给 View 绑定器

这一步由 CwViews 框架来完成。在内部，框架在为一个控件 (比如按钮) 配置 action 的时候，会生成一个 SignalControlEventActionTarget 对象，并在 Control.Preparer.applyBinding 中将它作为控件 action 的 target 进行配置。这个对象会在 action 被触发时向我们发送一个信号。

步骤 2a: 主 Action 被转换为 View State Action

在 action 绑定中，我们忽略这个事件的细节，转而构建一个 .togglePlay 消息：

```
return Button(recordingsAppButtonBindings:
    // ...
    .action(.primaryActionTriggered) --> Input()
        .map { _ in .togglePlay }
        .bind(to: play.mediaControl)
)
```

步骤 3a: View State 适配器更新 Media Control 的 View State

我们将 .togglePlay 消息与 play.mediaControl 绑定在一起，后者是一个控制播放状态的非持久化的 view state：

```
struct PlayState: StateContainer {
    // ...
    let mediaControl = TempVar<PlayerRecorderControl>()
    // ...
}
```

在 MAVB 中，view state 适配器会发送它自己的当前状态的快照。在上面的代码中，mediaControl 属性就是一个 view state 适配器，它所发送的值的类型是 PlayerRecording。要注意，PlayState 本身并不是一个 view state 适配器。它是其父级 view state 适配器的最新的值，playState 属性的定义如下：

```
struct SplitState: StateContainer {  
    // ...  
    let playState: Var<PlayState?>  
    // ...  
}
```

PlayState 和 SplitState 都遵守 StateContainer 协议，这个协议提供了序列化和输出调试日志的支持。还要注意，这些 view state 值的所有属性都是使用 let 进行声明的，唯一可变的只有 view state 适配器自身。

步骤 4a: 适配器对 Media Control View State 的变更进行观察

当我们使用 .bind(to: play.mediaControl) 绑定来设置 mediaControl 的值时，它通过输入绑定来将值发送给 TempVar 这个 view state 适配器，这会将值立即发送到它的输出绑定中去。这一个重新发送的工作是在内部由 TempVar 的实现进行处理的。

步骤 5a: 绑定对 Media Control 信号进行变形

在 audio player 的第二个参数中，我们使用了 TempVar 所输出的信号：

```
AudioPlayer(  
    url: play.url,  
    input: play.mediaControl  
        .mergeWith(AudioSession.shared.isActive.map { .audioSessionActive($0) })  
    // ...  
)  
)
```

步骤 6a: Media Control 被应用到 View (AudioPlayer) 上

AudioPlayer 是一个常规类，它接受绑定作为输出参数，并且只暴露一个 Cancellable 协议。这让它事实上成为了一个 view 绑定器。把一个 audio player 当作 view 可能会有些奇怪，但是在 MAVB 中，服务，以及 model 之外的其他所有东西，都是被当作 view 绑定器来处理的。

我们通过“更改 View State”图表中的六个步骤，现在有了一个 view。不过，它还不是我们想要更新的按钮。为了更新按钮标签，我们需要再次经历所有这六个步骤。

步骤 1b: Media State 中的变更被传递到 View 绑定器中

在内部，AudioPlayer 类用一个 AVAAudioPlayerDelegate 来观察底层的 AVAAudioPlayer，并监测播放状态变更，它会将状态变更通过 output 绑定进行传递。

步骤 2b: Action 绑定将 View Actions 转换为 Model Actions

AudioPlayer 的输出是多点广播 (multicast，发送给多个目标) 的，我们对 play.mediaState 这个目标感兴趣。由于 AudioPlayer 的 output 和 play.mediaState 的目标的类型是相同的，我们可以不需要变形，直接使用 bind：

```
AudioPlayer(  
  // ...  
  output: Input().multicast(  
    Input()  
      .ignoreElements()  
      .catchError { _ in .just(nil) }  
      .bind(to: split.playState),  
    Input().bind(to: play.mediaState)  
  )  
)
```

步骤 3b: View State 适配器更新 Media State

因为 mediaState 是一个 Var，它会立即设置所接收到的值：

```
struct PlayState: StateContainer {  
  // ...
```

```

let mediaState: Var<PlayerRecorderState>
// ...
}

```

步骤 4b: 通过适配器对 Media State 的变更进行观察

Var 会自动将新值发送到它的输出绑定中去。

步骤 5b: 绑定对 PlayState 的信号进行变形

Button 拥有一个 title 绑定，它将会把 mediaState 转换为一个字符串，然后对其进行包装。用于包装的 .normal 会告诉按钮，这个字符串是为控件的 .normal 状态 (而不是 .highlighted 或者 .selected) 而准备的。

```

return Button(recordingsAppButtonBindings:
    .title <-- play.mediaState.map { ps in
        .normal(ps.active ? .pauseLabel : .playLabel)
    },
    // ...
)

```

步骤 6b: View 绑定器更改按钮标签

这一步由 CwViews 框架在内部完成。在 Button.Preparer.applyBinding 中，如果提供了 .title 绑定，那么它将会被观察，当接收到被观察值时，Button 上的 setTitle 会被调用。

示例 2: 推入文件夹 View Controller

当用户在一个文件夹 table view 中点击另一个文件夹的行的时候，这个被选中的文件夹应该在一个新的 view controller 中进行展示。

步骤 1: Table View 选取 Action 被传递给 View 绑定器

这一步在 CwViews 框架内部执行。如果在 table view 上配置了选中某一行的 action，框架将会创建一个 TableView.Delegate 实例，并将它设置为 table 的 delegate。当这个 delegate 接

收到 `tableView(_:didSelectRowAt:)` 时，它会发送一个 `TableRow` 描述作为被设置的绑定的输入。

步骤 2: Action 绑定将 View Action 变形为 Model Action

`.didSelectRow` 绑定将 `folder.selection` 指定为它的绑定参数，当 `tableView(_:didSelectRowAt:)` 这个 `delegate` 方法被调用时，这个参数会接收到一个值：

```
.view -- TableView<Item>{
    // ...
    .didSelectRow --> folder.selection,
    // ...
}
```

在绑定参数中 `selection` 被直接使用，而没有 `map` 或者其他变形。这是因为 `view` 绑定器所发出的是 `TableRow<Item>` 值，而 `selection` 属性正好是一个值为 `TableRow<Item>` 类型的 `TempVar`：

```
struct FolderState: StateContainer {
    // ..
    let selection = TempVar<TableRow<Item>>()
    // ...
}
```

步骤 3: Model 适配器将 Action 应用到 View State 上

Model 适配器 `folder.selection` 是一个 `TempVar` 的实例。`TempVar` 可以从外部进行设置和观察，但是它不会进行持久化。当 `TempVar` 通过它的输入接收到一个新的值时，它将自己的状态也设置为这个新值。

步骤 4 & 5: Action 被变形为适配器上的通知

当一个 `TempVar` 接收到新值时，它会立即将这个新值作为通知从输出信号发送出去。我们使用一个 `compactMap` 变形，来从输出信号 `folder.selection` 中获取文件夹，并在文件夹不为 `nil` 时将它发送出去。结果中的文件夹被转换为一个 `FolderState`，它描述了下一个需要被推入导航栈的文件夹 `view`。这个变形的结果被绑定到 `split view` 的导航栈的 `pushInput` 上：

```

.view -- TableView<Item>{
    // ...
    .cancelOnClose -- [
        folder.selection
        .compactMap { row in row.data?.folder }
        .map { folder in FolderState(folderUuid: folder.uuid) }
        .cancellableBind(to: split.navStack.pushInput),
    // ...
    ]
}

```

我们使用 `cancelOnClose` 连同 `cancellableBind` 一起来确保当 `table view` 被释放时信号也随之失效 (这么做打破了两个 `model` 适配器之间的绑定所可能带来的引用循环)。

步骤 6: UINavigationController 推入新的文件夹 View

`Split view controller` 中的主 `view` 包含了 `master navigation controller`。Navigation controller 指定了一个 `stack` 参数，它绑定了 `split state` 的导航栈。导航栈包含着文件夹状态，每个文件夹状态都可以通过使用 `folderView` 函数转变为 `view controller`：

```

private func primaryView(_ split: SplitState, _ store: StoreAdapter)
-> UINavigationControllerConvertible
{
    return UINavigationController(
        .navigationBar -- navBarStyles(),
        .stack <-- split.navStack.stackMap { return folderView($0, split, store) },
        .poppedToCount --> split.navStack.poppedToCount
    )
}

```

在绑定的右侧是 `stackMap` 方法，它的任务是将文件夹 `view controller` 转变为一个信号，该信号负责发送栈上诸如 `push` 和 `pop` 这样的变化。这个绑定上的栈的变化，将被 `navigation controller` 的 `view` 绑定器所截获。它们会在随后被变形为底层 `navigation controller` 的 `push` 和 `pop` 操作。

测试

MAVB 中的集成测试

在 MVC 中，我们使用集成测试同时对 view controller 和 view 进行测试。这种类型的测试是全面的，但是难以编写 (因为这需要我们对程序、app 以及 view 框架有全面的理解)，而且容易损坏 (因为任意地方的任意改动都会让测试失效)，不过它能被用在任意模式中。

如果我们愿意，我们可以在 MAVB 中使用 MVC 风格的集成测试，我们只需要将 view 绑定器转换为它们的底层的 UIKit 对象，然后通过同样的方式去测试 UIKit 对象就行了。但是，除非我们想要对某个特定的集成任务进行覆盖，这类测试的复杂且易损的特点，让我们更愿意对它敬而远之。

MAVB 中的接口测试

在 MVVM 中，我们使用接口测试的方法对 view-model 进行了测试。接口测试要比集成测试简单，因为我们的测试可以将自定义的观察者连接到 view-model 的可观察量上，而不需要将 view-model 和 view 绑定起来。这可以让我们不需要在测试时构建和管理 view，从而简化了相关的工作。

我们更趋向于选择 MVVM 那样的接口测试，而不是 MVC 那样的集成测试。然而，在 MAVB 模式的要求中，唯一公开的接口是 model 适配器的接口 (包括 view state 的接口)。我们可以通过接口来测试任意的 model，这在本书中的任何模式中都是适用的，但是我们真正想要测试的是 app 逻辑 (也就是 model 和 view 之间的逻辑)。在我们的示例 app 中，view state 接口没有封装重要的 view 的逻辑；View state 主要是一系列可设置的值，它们由集成在绑定中的 view 逻辑进行驱动，而这些绑定在 view 构建的时候就确定了：

```
struct PlayState {  
    let mediaState: Var<MediaState>  
}  
  
func constructLabel(_ play: PlayState) {  
    return Label(.text <-- playState.mediaState.map { timeString($0.progress) })  
}
```

为了让 app 逻辑更容易测试，我们可以将这些变形进行重构，把它们封装到 model 适配器中去：

```

struct PlayState {
  let mediaState: Var<MediaState>
  var labelText: Signal<String> {
    return mediaState.map { state in timeString(state.progress) }
  }
}

func constructLabel(_ play: PlayState) {
  return Label(.text <-- playState. labelText)
}

```

现在变形部分的代码也变成我们的接口了，我们可以像在 MVVM 里那样，使用接口测试的方法对它进行测试了。

一种独特的方式

不过现实是，在 MAVB 中，我们不需要手动将变形逻辑移动到 **view state** 接口中，以此来创建一个可测试的接口。因为在这个架构中已经有 **view** 绑定器这样一层可以测试的部分了，我们并不需要手动创建它。

一个 **view** 绑定器其实是将一系列绑定 (值、信号、信号输入以及函数) 进行存储的一个数组。我们可以将这个数组解包 (unpack)，审查它们的内容，然后将我们找到的绑定作为可测试接口来使用。**consumeBindings** 函数可以用来从绑定器中解包一个绑定。正如其名，一旦绑定被 **consume** (消费) 掉，绑定器就不能再被用来创造实际的实例了。相似地，一旦绑定器创建了一个实例，我们也就不能再 **consume** 这些绑定。所以，这类测试需要在对象被插入到实际的 **view** 或者 **view controller** 层级之前进行。

不管使用哪种接口测试，我们都需要创建一些输入参数来驱动测试。在下面的代码中，我们构建了一个示例 **Store**：

```

func constructTestingStore() -> StoreAdapter {
  let store = Store(relativePath: "store.json", items: [
    Item(Folder(name: "Recordings", uuid: Store.rootFolderUuid,
      parentUuid: nil, childUuids: [uuid1, uuid3])),
    Item(Folder(name: "Child 1", uuid: uuid1,
      parentUuid: Store.rootFolderUuid, childUuids: [uuid2, uuid4])),
    Item(Folder(name: "Child 2", uuid: uuid2,
      parentUuid: uuid1, childUuids: [])),
  ])
}

```

```

        Item(Recording(name: "Recording 1", uuid: uuid3,
            parentUuid: Store.rootFolderUuid)),
        Item(Recording(name: "Recording 2", uuid: uuid4,
            parentUuid: uuid1)),
    ])
    return StoreAdapter(store: store)
}

```

在测试绑定的时候，会牵涉到很多对于绑定结构的构造和遍历，在 `setUp` 方法中加载和遍历 `folderView` 结构的顶层 (它会返回一个 `ViewControllerConvertible`)，会很有帮助：

```

class FolderView: XCTestCase {
    var store: StoreAdapter = StoreAdapter(store: Store(relativePath: nil))
    var rootFolder: FolderState!
    var childFolder: FolderState!
    var split: SplitState!
    var rootViewController: ViewControllerConvertible!
    var childViewController: ViewControllerConvertible!

    override func setUp() {
        super.setUp()

        store = constructTestingStore()
        rootFolder = FolderState(folderUuid: Store.rootFolderUuid)
        childFolder = FolderState(folderUuid: uuid1)
        split = SplitState()
        rootViewController = folderView(rootFolder, split, store)
        childViewController = folderView(childFolder, split, store)
    }
    // ...
}

```

这个设置方法将构建我们的参数，并且使用传入根文件夹，以及传入子文件夹这两种方法来运行 `folderView` 函数。

每个测试剩余的工作就是在绑定中向下遍历，直到找到想要的属性，然后测试它的当前值：

```

func testRootFolderNavigationTitle() throws {
    let viewControllerBindings =

```

```

    try ViewController.consumeBindings(from: rootViewController)
let title =
    try ViewController.Binding.value(for: .title, in: viewControllerBindings)
XCTAssertEqual(title, "Recordings")
}

```

在 MAVB 的很多情况下，因为绑定会确保观察的发生，所以测试一个单独的值就足够了。但是，如果我们对测试变更逻辑感兴趣的话，我们可以去获取绑定参数的信号表现，而不单单只是当前的值。

接下来的例子展示了如何在 store 中文件夹名字的更改前后，对 child view controller 的名称进行测试：

```

func testChildFolderNavigationTitle() throws {
    let bindings = try ViewController.consumeBindings(from: childViewController)
    let childTitle = try ViewController.Binding.signal(for: .title, in: bindings)

    var values = [String]()
    childTitle.subscribeValuesUntilEnd { values.append($0) }

    XCTAssertEqual(values.at(0), "Child 1")
    store.input.send(value: .renameItem(uuid: uuid1, newName: "New name"))
    XCTAssertEqual(values.at(1), "New name")
}

```

最后一种常见的测试是测试 view 的回调。**Action 绑定**会触发回调，在 action 绑定中，绑定参数将会是一个信号输入。我们可以将值发送到这个信号输入中去，然后验证 model state 是否发生了正确的变更。

下面是对“添加文件夹”按钮的测试。这个按钮必须要设置 split view 上的 text alert state (这个状态稍后会通过 split view controller 上创建的绑定，来驱动显示一个文本弹窗)：

```

func testCreateNewFolder() throws {
    // 从 view controller 消耗绑定
    let vcBindings = try ViewController.consumeBindings(from: rootViewController)
    let item = try ViewController.Binding.value(for: .navigationItem, in: vcBindings)

    // 从 navigation item 消耗绑定
    let itemBindings = try NavigationItem.consumeBindings(from: item)
}

```

```

let rightItems = try NavigationItem.Binding.value(for: .rightBarButtonItems,
    in: itemBindings)
guard let addItem = rightItems.value.at(1) else { XCTFail(); return }

// 从 bar button item 消耗绑定
let addBindings = try BarButtonItem.consumeBindings(from: addItem)
let targetAction = try BarButtonItem.Binding.argument(for: .action,
    in: addBindings)
guard case .singleTarget(let actionInput) = targetAction
    else { XCTFail(); return }

var values = [TextAlertState?]()
split.textAlert.subscribeValuesUntilEnd { values.append($0) }
XCTAssert(values.at(0)?.isNil == true)
actionInput.send(value: ())
XCTAssertEqual(values.at(1)?.parentUuid, Store.rootFolderUuid)
}

```

实际的测试在最下面。我们检查了初始状态 (此时应该还没有文本弹窗)，发送 action (模拟一次按钮点击)，然后测试最终状态 (应该显示一个文本弹窗，并以根目录为目标文件夹)。

与 View-Model 接口测试比较

在大多数方面，对 view 绑定器消耗绑定的测试，和对 view-model 上所公开的可观察量的测试的过程非常相似，这是因为在我们思考时，通常把它们两者都定位为对响应式编程的串流的观察。

但是，对于 view-model 而言，接口测试存在一个潜在的间隙：在 view controller 中可能存在着与 view-model 和 view 之间有关的逻辑，我们的接口测试避开了这部分逻辑。在 view controller 中的最终的绑定逻辑，或者在 delegate 里的实现，可能会影响行为，但这将无法被测试捕捉到。另外，view 的构建、布局和其他在 viewDidLoad 中可能进行的处理也会被排除在 view-model 之外，因为在这些时间点上，view-model 还没有被完全构建出来。

使用 view 绑定器消耗绑定，可以确保在被测试的绑定和 view 之间，**没有其他用户代码**。View 绑定器将 view 中包括常量值和构造时就确定的不变量在内的**所有**属性进行封装。因为除了绑定以外，view 的构建不再需要其他任何东西，所以我们可以把绑定看作是 view 的完整表示。

另一种可能性是将 MAVB 的架构改成和 MVVM 类似的结构。举例来说，我们可以将变形部分的代码从绑定中移动到 model 适配器里，或者移动到 model 适配器上方的一个单独的层中。这样一来，我们就可以在独立于 view 绑定器的前提下，测试这些变形逻辑。

讨论

MAVB 有三个主要优点：它的声明式语法结构非常清晰，通讯的模型在各个部分是统一的，model 和 view 之间的联系的定义也很清楚。不过，这些优点同时也带来了两个最大的不足：独特而抽象的语法造成了陡峭的学习曲线，复杂的语法在调试和编辑工具上也造成了麻烦。

声明式语法

MAVB 使用声明式的语法来描述 view 和它们的绑定。这些绑定是一系列不可改变的规则的集合，它们精确描述了 view 构建的方式以及对改变进行响应的方法。在纯 UIKit 中，我们只能使用像是方法调用这样的普通的控制流，来命令式地声明 view 和它们的绑定。我们在本书中使用的一些其他的声明系统有 Auto Layout (描述 view 布局的规则)，响应式编程 (描述数据管道的规则)，以及 [TEA](#)。

我们为 view 绑定器选择了声明式的语法，这是因为我们认为它读起来更容易；我们只需要阅读一个单一的、不可变的绑定列表，就能理解 view 的构建方式和随着时间进行的变化。在纯 UIKit 代码中，事情有所不同，我们通常需要研究 view 构建和变更时的多个方法中的代码。

声明式的系统对可读性的提升大有裨益，但是它们通常也意味着陡峭的学习曲线：我们需要学习这套系统提供的规则，我们需要学习系统如何解释这些规则，我们需要学习如何调试这些描述式的程序，有时候我们还需要理解这些规则可能带来的性能上的问题。

统一的通讯

我们从 MAVB 中所能得到的另一个主要收益是，它带来了一套统一的通讯机制：所有的变化都是通过响应式编程绑定进行通讯的。在 MVC 中，多个不同机制的共用并不鲜见，就算是在一个很小的项目里，我们也会使用诸如通知、回调、方法调用、delegate、target/action 以及 KVO 等多种机制。

单一的通讯机制有助于阅读代码，它也使得在其上方构建可重用的抽象成为可能。我们可以很轻易地看到数据在你的 app 中的流动方式。

Model 和 View 之间的关系

在 MVC 中，model 和 view 之间的关系是被松散定义的。我们可能读取了初始的状态，但是没有观察后续的变更。我们也可能没有将变更应用到合适的 view 上。就算我们没有犯错，在我们的代码中从 model 到 view 遵守这个管道也是相当困难的。

在 MAVB 中，从 model 到 view 的关系是在 view 被构建的地方就清晰定义了的。只需要简单一瞥，我们就能知道有哪些数据被用在 view 中，或者 view action 会对 model 造成什么样的影响。这可以让我们更不容易出现在 observer 以外的地方读取 model 的错误，也让 action 无法正确连接到 model 的概率大幅下降。

MVVM 中的 view-model 描述了一种类似的 model 到 view 的关系。但是整个 model 到 view 的关系是分布在 view-model、controller 和 storyboard 中的，它们的关系并不是那么容易被观测到。

缺点

声明式的程序会比命令式的程序更容易阅读，但是这个效果是主观的。如果你从来没有学习过这套系统的编码结构和规则，那么它会显得非常神秘。如果你对命令式的程序非常执着 (想要自己书写每一个相关的步骤)，那么你可能永远不会去选择使用声明式编程的方式。

本章中所描述的 MAVB 模式依赖了两个大的框架：CwlViews 和 CwlSignal。这两个库都还在活跃开发中，你需要注意它们都还处在实验阶段，并没有准备好用在生产环境里。特别是 CwlViews 库，它在所部署的平台上还欠缺相当多的测试。

就算抛开它们的实现质量，这些库也有相当陡峭的学习曲线。CwlViews 是对 UIKit 进行的全面的包装。虽然 UIKit 中的大部分的属性、action、delegate 方法和其他一些行为都在 CwlViews 中有等价的实现，但是这个关系并非总是一对一，所以需要一定的学习和重定位。要使用 CwlViews，我们必须接受在我们自己的代码和 UIKit 之间添加一层额外的抽象。

当使用 CwlViews 的语法编写代码时，Swift 的错误提示会接近于无用。若是使用得当，CwlViews 语法的代码可以轻松又快速地编译，但是如果我们犯了一些像是忘写逗号，或者使用了错误的类型，或者使用了错误的绑定操作符这样的错误，Swift 通常会高亮第三个参数 (就算实际错误是发生在第二十一个参数)，并且很无能地提示我们也许忘记了“bindings”参数。就算你已经理解了 Swift 在这方面的缺陷，找到问题所在往往还是需要 (使用二分查找) 注释掉一些代码，直到找到症结所在。

因为使用了响应式编程，调试会变得困难得多。LLDB 是为了调试命令式程序而开发的，但是当调试响应式的程序时，调试器进行步入 (step into) 和步出 (step out) 操作时，经常会涉及到大量的响应式编程内部的处理函数。当我们在追踪我们自己的程序逻辑时，我们对这些内部函数并不感兴趣。如果调试器能够在响应式管道之间进行跳转的话会更实用，但是这并不是内建的调试器的工作方式。正因如此，我们需要精心管理程序中的断点，或者使用别的方式来调试我们的代码。

在 MVC 中使用 MAVB

因为 MAVB 看起来十分不同，所以可能看上去不太明显，但是 MAVB 是可以和 MVC 一同工作的。我们可以将任意的 view 绑定器用标准的 Cocoa view 或者 view controller 来替代，或者使用 view 绑定器来在一个更大的 MVC app 中替换掉单个的 view 或 view controller。

举个例子，我们可以将 MAVB 版本的录音 app 中的录音 view 替换成 MVC 风格的 view controller 子类。使用一个和 view 绑定器构造函数接受相同参数的 view controller 子类，可以将录音 view 的构造函数进行替换：

```
func recordView(_ record: RecordState, _ split: SplitState,
    _ store: StoreAdapter) -> ViewControllerConvertible
{
    return RecordViewController(record, split, store)
}
```

这种使用 Cocoa 的 UIViewController 来直接替代 CwlViews 的 ViewController view 绑定器的方式之所以能够工作，是因为在 split view 上的 .modalPresentation 绑定会调用 recordView 函数，而它可以接受任何实现了 ViewControllerConvertible 协议的对象，ViewController 和 UIViewController 都实现了这个协议。该协议只有一个方法，那就是获取一个 UIViewController。对 UIViewController 来说，它只需要返回 self，而对 ViewController，它可以延迟返回所构建的底层实例。

接下来，我们需要 init 的实现，来捕获我们传入的所有可观察量：

```
class RecordViewController: UIViewController {
    let mediaControl: TempVar<PlayerRecorderControl>
    let store: StoreAdapter
    let recorder: AudioRecorder
    let mediaState: Signal<PlayerRecorderState>
    var endpoint: Cancellable?
```

```
// 与 view 绑定器构造函数接受相同参数
// 接受相同参数的 view controller 初始化方法
init(_ record: RecordState, _ split: SplitState, _ store: StoreAdapter) {
    // ...
}
```

对 view controller 来说，对 view 进行构建的代码应该发生在 loadView 方法中。和 MAVB 的例子保持一致，我们应该在这个构建过程中将我们的信号输入和信号与 view 进行绑定 (而不是在 viewDidLoad 或者 MVC 中其他典型的构建后才调用的方法中)。

我们观察信号的输出，并使用 subscribe 来将它应用到 view 的值上。下面的例子展示了将 label 的文本值应用到录音 view 的进度条上的方法：

```
override func loadView() {
    let progressLabel = UILabel()
    endpoint = mediaState.subscribeValues {
        progressLabel.text = timeString($0.progress)
    }
    // ...
}
```

实现 action 要更麻烦一些，我们需要定义一个单独的方法来接收 action 的回调：

```
// ... continued from previous example
let stopButton = UIButton(type: .roundedRect)
stopButton.addTarget(self, action: #selector(stop),
    for: .primaryActionTriggered)

// 其他设置，布局以及组装到单个的 view container 中以发送

self.view = containerView
}

@objc func stop(_ sender: Any?) {
    mediaControl.input.send(value: .stop)
    store.input.send(value: .recordingComplete)
}
```

如果我们将所有的 view 和 view controller 都按照这种 (没有 view 绑定器的) 方式编写的话, 结果会是一种很有趣的混合模式: view state 适配器配合 Cocoa 的 view controller。这有点像是响应式编程版本的 MVC+VS。

经验和教训

View 的构建

在 UIKit 中, view 通常是和 view controller 结合在一起的, 或者是使用一个 UIView 的子类来表现。在 MAVB 里, 我们不去子类化 view, 而是通过现有的 view 将它组合出来。要构建可重用的组件, 我们在一个函数中将这些组合好的 view 用必要的参数包装起来。比如, 下面的函数构建了一行 view, 它的左边是播放器的 progress, 右边是录音的 duration:

```
func progressRow(_ playState: PlayState) -> Layout.Entity {
    return .horizontal(
        .matchedPair(
            .view(
                Label(.text <-- playState.mediaState.map { timeString($0.progress) })
            ),
            .view(Label(
                .text <-- playState.mediaState.map { timeString($0.duration) },
                .textAlignment -- .right
            ))
        )
    )
}
```

当我们不使用 MAVB 时, 我们也可以使用同样的技术。当我们通过代码构建 view 时, 将已经存在的 view 的子类组合起来, 用以构建更复杂的组件, 并用一个函数来包装它的初始化方法, 会让整个过程更加简单。比如说, 下面的类似的代码将构建一个 stack view, 它有两个用来表示录音播放进度的 label:

```
func progressRow(progress: TimeInterval, duration: TimeInterval)
-> UIStackView
{
    let progressLabel = UILabel()
    progressLabel.text = timeString(progress)
```

```

let durationLabel = UILabel()
durationLabel.text = timeString(duration)
durationLabel.textAlignment = .right

let stack = UIStackView(arrangedSubviews: [progressLabel, durationLabel])
stack.axis = .horizontal
return stack
}

```

在最初构建的时候，上面的代码可以正常工作。但是，当要对 label 进行变更时，我们会面临困难：我们必须从 stack view 中找到正确的 subview，把它转换为 label 类型，然后设置文本。在典型的 UIKit 中，我们会将上面的代码用一个类来包装，而不是用一个函数包装，因为用类包装我们才能将 progress 和 duration 公开为可设置的属性。

同样地，我们可以使用 MAVB 的思想。首先我们需要一个响应式的库，比如下面是用 RxSwift 实现的同样的例子：

```

func progressRow(progress: Observable<TimeInterval>,
    duration: Observable<TimeInterval>) -> UIStackView
{
    let progressLabel = UILabel()
    progress.map(timeString).bind(to: progressLabel.rx.text)
        .disposed(by: disposeBag)

    let durationLabel = UILabel()
    duration.map(timeString).bind(to: durationLabel.rx.text)
        .disposed(by: disposeBag)
    durationLabel.textAlignment = .right

    let stack = UIStackView(arrangedSubviews: [progressLabel, durationLabel])
    stack.axis = .horizontal
    return stack
}

```

一旦使用了一个响应式编程库，上面的这种不通过子类化 UIView 的方式来构建可重用组件的技术就很有用了。在这本书中所有的框架里，这种方法能直接和 MVVM，MVC+VS，MAVB 以及 TEA 搭配使用。

资源所有权

对于资源的所有权的处理，MVC 使用了很多方式，而 MVVM 中使用的是一种相当固定的模式，所以来看看 MVVM 中的资源所有权会相对容易。举例来说，如果我们想要对文本进行绑定，我们可能会编写下面的代码：

```
class MyViewController: UIViewController {
    @IBOutlet var textField: UITextField!
    let viewModel = ViewModel()
    var disposeBag = DisposeBag()

    override func viewDidLoad() {
        super.viewDidLoad()
        viewModel.text.bind(to: textField.rx.text).disposed(by: disposeBag)
    }
}
```

这个例子中，维护 disposeBag 并不是很沉重的技术负担，这种模式非常简单，而且经常使用，所以它已经变成使用 RxSwift 进行 MVVM 开发时的本能了。不过不可否认，这还是带来了语法上的开销，而且还会存在一种情况，那就是需要在自身持有 dispose bag 的对象之间建立绑定。在这种时候，如果指定了错误的 dispose bag，那么所有权的管理就会出现问题的。

让我们来看看 CwlViews 的 view 绑定器是如何处理同样的文本观察的：

```
TextField(
    text <-- modelAdapter.text
)
```

这个绑定使用了内部创建的一个 Cancellable 对象，这和 RxSwift 的 Disposable 是等效的，不过我们没有看到这个对象，它也不需要单独的处理。

这里我们使用的解决方案并不是特别复杂：绑定对资源进行描述，并且这个资源在 text field 被创建的时候就与其关联，这样一来，该资源就与 text field 绑定在一起了（在这个例子中，资源会通过 Objective-C runtime 的关联属性 (associated property) 进行存储）。

CwlViews 所提供的 view 绑定器向我们展示了另一种管理的安排方式，所有的资源都在构建的时候与它们所有者相绑定。在 C++ 中，这种安排被叫做**资源获取即初始化** (resource acquisition is initialization, RAII)。RAII 这个名字有些欠考虑，也不太妥当，它的前提是当我们创建一个资源时（本例中，是这个绑定），这个资源的生命周期应该自然地由它的父对象 (view

或者场景) 决定。当资源被交给 view 或者场景时, 它们将获取并负责资源的生命周期。由于历史的局限性, RAII 会和像是区域锁 (scoped locks) 这样的笨拙的概念相关联, 不过当我们对资源的生命周期与对象绑定一无所知时, 它反而能工作得最好。

当我们在构建资源所有者的父对象时, 或者调用这个所有者上的其他方法时, 将资源的构建作为参数来使用的话, 就会导致透明的生命周期管理。如果父对象本身也是一个资源, 而且是作为它自身的父对象的调用的一部分来进行构建的话 (并且这个自身的父对象也是另一个父对象的调用的一部分, 并以此推演), 那么在整个层级上资源管理的部分就都被移除了。

在 MVC 或者 MVVM 这样的 view controller 生命周期不清晰的模式中 (由 storyboard 负责构建, 然后通过调用 viewDidLoad 等许多生命周期方法后才完成全部构建), 维护这样深层级的资源持有是相当困难的。不过我们还是可以做一些细微的改进。

比如, 下面的协议:

```
protocol ScopedOwner {
    var ownedObjects: [Any] { get set }
}

extension ScopedOwner {
    mutating func bind<O: ObservableType, P: ObserverType>(<_ observable: O,
        to observer: P) where O.E == P.E
    {
        let disposable = observable.asObservable().bind(to: observer)
        ownedObjects.append(disposable)
    }
}
```

这让我们可以将最初的 MVVM 例子重写为:

```
class MyViewController: UIViewController, ScopedOwner {
    @IBOutlet var textField: UITextField!
    let viewModel = ViewModel()
    var ownedObjects: [Any] = []

    override func viewDidLoad() {
        super.viewDidLoad()
        self.bind(viewModel.text, to: textField.rx.text)
    }
}
```

```
}
```

和最初的例子相比，并没有太大的不同，但是我们确实通过将所有权的操作编码到整个 `view controller` 的接口中，减少或者去除了所有权误管理的可能性，从而简化了绑定的操作。

声明式的 View 层

和命令式编程不同，声明式编程不使用顺序执行的语句，或者像是 `if` 和 `for` 那样的控制流，相反，它对结构和规则进行定义，包括可能的数据结构，函数的列表，或者其他表达式。这些结构和规则之后被系统所解释，并且执行相应的行为。

在 MAVB 版本的录音 app 中，我们用到了三种不同的声明式系统：

1. `View` 绑定器使用声明式的方法描述 `view` 的行为，然后框架使用这些描述来构建、配置以及管理 `view` 层级。
2. 绑定使用了响应式编程，这是另外一种声明式的系统，我们指定信号变形的方式，而不是手工管理控制流。
3. 布局是通过一种描述水平和竖直方向元素布置的语法来完成的，之后，满足这些描述的 `Auto Layout` 的约束会被创建。

声明式编程在规则比较简单的时候会很有用，相比于执行规则的命令式编程，使用声明式编程更不容易犯错。如果规则挑选得当的话，那些有歧义、未定义、或者存在问题的条件和状态都应该是不可能发生的，因此从设计的层面就可以杜绝 `bug`。

在 [MVVM 一章](#)中，我们已经看到了将响应式编程使用在不同架构上以简化观察和变形管道的方式。响应式编程也可以使用在基本的 `MVC` 上，而不需要 `view-model` 的存在。[TEA](#) 会为我们展示另一种不使用响应式编程，来声明式地实现观察者模式的方式。

在 MAVB app 中使用的布局语法来自于一个单独的文件，`CwlLayout.swift`。它可以通过函数调用中参数的位置，为每个对象推断出四种不同的布局约束 (水平位置，竖直位置，水平尺寸和竖直尺寸)，这么做有效地减少了所需要的代码量。

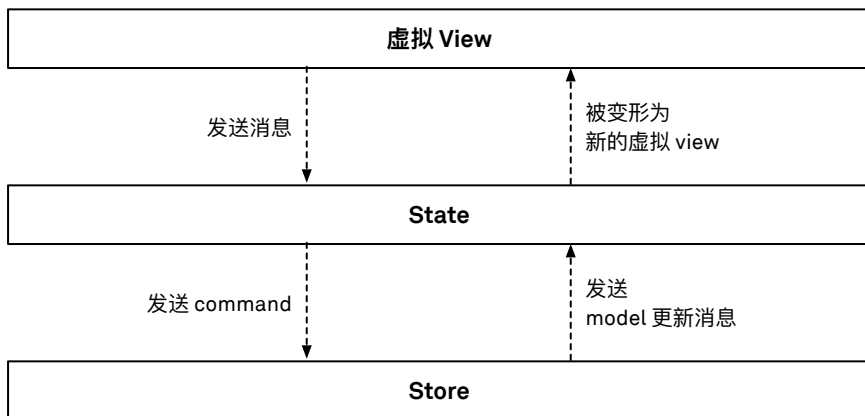
Elm 架构

8

Elm 架构 (The Elm Architecture, TEA) 与 macOS 和 iOS 上的 MVC 以及其他架构有天壤之别。在 TEA 模式中，我们所编写的 app 的每一层，都是由一个监督框架 (也就是 TEA 框架) 所拥有及管理的。在这些层级之间的通讯，以及它们与 UIKit 的交互，也都由框架进行处理，这建立了一道壁垒，来避免我们的代码对这些功能负责。这个监督框架将通常的 UIKit 的命令式及面向对象编程模型隐藏起来，让我们得以通过声明式和函数式的风格来编写 app 代码。

在 iOS 上使用 TEA 的最主要目标，还是想要解决用户界面、model 数据和 app view state 之间不一致这个常见问题。在 MVC 中，为了保持一致性，controller 层级需要处理许多的生命周期和状态变更事件 (构建，view state 变更，model 变更等)，并且小心地将这些变更应用到存在的对象上，来反映 app 的整个状态。TEA 将这些不同的职责进行了简化，将它们归纳到一条单一的路径上：不管 app 中任何地方的任何状态发生改变，app 的整个状态和 model 数据将会被传递到一个函数中，它会构建一个新的、不可变的**虚拟 view 层级**。除了这条路径之外，没有别的方法可以更新 view。这使得用户界面和状态可以时刻保持同步。

换言之，我们的代码不需要处理将特定事件反映到 view 上的工作。我们要做的是，在某个特定状态下对 view 层级进行描述，接下来 TEA 框架将确保在状态改变的任意时候，这个描述都将被遵循。



在上面的图表中，app state 包含了渲染当前用户界面所需要的全部信息 (view state 和 model 数据)。这里的 store 和其他所有实现中的 store 很相似：它负责从磁盘加载数据，以及把变更保存到磁盘中去。

在我们继续之前，我们应该先对这些实现的质量进行澄清：在 iOS 中，还不存在经过证实的，可以用在产品代码中的 TEA 框架。

我们针对录音 app 进行了 TEA 的部分实现，用来说明这个模式要如何使用。不过 TEA 已经在使用 Elm 语言的 web app 中被作为一种设计模式完整验证过了，本书中所使用的是该框架的 iOS 版本，它可以被看作是实验性质的概念验证。如果你想要把它用在生产环境的 app 中，你需要准备好自己对这个框架的实现进行适配，扩展以及维护。我们建议你可以在一个小型的业余项目中尝试一下，去理解它会带来些什么。请一定记得阅读本章中的[讨论部分](#)，来了解这一路上你可能会面临的困难。

除开它的实验性质以及缺少产品级别的框架以外，我们选择在书中介绍 TEA 的原因有两点：

1. TEA 和我们所习惯的 Apple 平台的架构有很大不同，这可以帮助我们拓宽视野，让我们了解一些和传统面向对象的模式完全不同的东西。
2. TEA 中的很多思想对其他的架构和实现都有所启迪；了解这些架构的源头，可以帮助我们更快进入到新的 app 架构的领域中。在 iOS 上的像是 [React Native](#) 这样的模式，在概念上和 TEA 有大量的重叠。即使是 MAVB 和 MVC+VS 这样的架构，也从 TEA 中获得了一些灵感。

探索实现

在本节中，我们先将 TEA 框架的存在看作是理所当然，然后来研究 app 的代码。下面的代码，除了启动流程部分的描述之外，都不会和 UIKit 直接交互。App 代码只需要和我们自己的 model 层级和 TEA 框架进行交互。当我们在本节中提到 view controller 和 view 时，我们所说的不是 UIViewController 和 UIView 的实例，而是它们的虚拟版本，也就是 TEA 框架中的 struct 和 enum。这些 struct 和 enum 都是纯值：它们对 view 层级进行描述，之后这些描述会被用来构建实际的 view 层级。

如果你对 TEA 内部的工作方式感兴趣的话，可以去看看[相关的章节](#)，在那里你可以看到幕后的实现细节。

构建

在我们的实现中，UIApplication 和它的 delegate 是存在于 TEA 框架之外的，所以我们使用标准的 @UIApplicationMain 的行为来构建它们。我们来看看 AppDelegate 顶部的实现，这里是构建开始的地方：

```
@UIApplicationMain
```

```
class AppDelegate: UIResponder, UIApplicationDelegate {
```

```
// ...
let driver = Driver<AppState, AppState.Message>(
    AppState(rootFolder: Store.shared.rootFolder),
    update: { state, message in state.update(message) },
    view: { state in state.viewController },
    subscriptions: { state in state.subscriptions })
// ...
}
```

在这里所构建的 driver，是 TEA 框架所公开的接口。它负责运行 app 的核心更新 loop (消息 — 更新 — 渲染)，driver 持有 AppState 这个 model 值，并将程序的其他部分连接起来。这是框架在 app 中唯一的公开的地方，在构建后，程序的剩余部分就都是在 driver 内部运行的了。

Driver 接受四个参数：

1. 初始的 app 状态 (在录音 app 的例子中，是 AppState 类型的值)。
2. update - 一个 reducer，用来在一个新的消息到达时，对状态进行更新。
3. view - 一个渲染函数，用来从 app 状态中创建虚拟 view 层级。
4. subscriptions - 一个用来获取 app 状态订阅的函数。

subscriptions 参数被用来在 app 状态发生改变时动态地计算订阅。比如说，当我们处于录音状态时，我们会想要订阅录音进度。我们会在本章稍后的部分详细讨论有关订阅的内容。

在 application(_:didFinishLaunchingWithOptions:) 中，root view controller 是通过 driver 的 viewController 属性来进行创建的 (它会通过上面的 view 参数所返回的虚拟 view controller，来创建一个真正的 UIViewController)，它将被用作 view 层级的最根基部分：

@UIApplicationMain

```
class AppDelegate: UIResponder, UIApplicationDelegate {
    func application(_ application: UIApplication,
        didFinishLaunchingWithOptions launchOptions:
            [UIApplicationLaunchOptionsKey: Any]?) -> Bool
    {
        window = UIWindow(frame: UIScreen.main.bounds)
        window?.rootViewController = driver.viewController
        window?.makeKeyAndVisible()
        window?.backgroundColor = .white
        return true
    }
}
```

```

    }
}

```

当框架使用虚拟 view 对 UIKit view 进行构建时，它会设定从 UIKit view 对象到 driver 的调用。当一个 UIKit view 事件发生时，view 对象会将对应的消息发送给 driver，driver 随后把这条消息和当前的状态发送给 update 函数，来对状态进行改变。框架负责对所有未来的更新所带来的 view 层级变化进行维护。

将 View 与初始数据连接

TEA 通过 app 状态构建虚拟 view，当状态改变时，会生成与之前无关的新的虚拟 view。状态和虚拟 view 都是值类型 (enum 和 struct)，所以它们无法持有对彼此的引用。

将初始数据传递给虚拟 view 的方式，和后续更新所采用的方式一样。举例来说，下面是 root view controller 构建的方式：

```

extension AppState {
    var viewController: ViewController<Message> {
        let rootView = SplitViewController<Message> {
            left: { _ in self.master },
            right: self.detail,
            collapseSecondaryViewController: playState == nil,
            popDetail: .popDetail()
        }
        return .splitViewController(rootView, modal: recordModal)
    }
}

```

对于 self.master、self.detail 和 recordModal 的调用，会将 root view controller 的所有可能的子部件都作为虚拟 view 进行构建，这样，返回的 split view controller 将包含**整个**虚拟 view 的树。Driver 将在需要的时候把这些虚拟 view 转变为实际的 UIView 和 UIViewController。

为了说明这些子虚拟 view 是如何创建的，我们来看看 master 的定义：

```

extension AppState {
    var master: NavigationController<Message> {
        let viewControllers: [NavigationItem<Message>] = folders.map { folder in
            let tv: TableView<Message> = folder.tableView(
                onSelect: Message.select,
            )
        }
        return .navigationController(viewControllers)
    }
}

```

```

        onDelete: Message.delete)
    // ...
}
return UINavigationController(viewControllers: viewControllers,
    back: .back, popDetail: .popDetail)
}
}

```

App 状态上的 `folders` 属性用来为导航栈中的 view controller 提供数据：我们将所有文件夹进行映射，并为它们中的每个文件夹创建一个 table view。创建这些 table view 的方法，被定义在 Folder 的扩展中：

```

extension Folder {
    func tableView<Message>(onSelect: (Item) -> Message,
        onDelete: (Item) -> Message) -> TableView<Message>
    {
        return TableView(items: items.map { item in
            let text: String
            switch item {
            case let .folder(folder):
                text = "📁 \(folder.name)"
            case let .recording(recording):
                text = "🎧 \(recording.name)"
            }
            return TableViewCell(identity: AnyHashable(item.uuid),
                text: text, onSelect: onSelect(item), onDelete: onDelete(item))
        })
    }
}

```

TableView 需要一个类型为 TableViewCell 的值的数组。为了创建这些 cell，我们将文件夹中的项目进行 map，把每个项目转变为一个 table view cell，cell 的配置也在这里发生。回到我们在构建主 navigation controller 中的 master 属性里，我们在这里将每个 table view 包装到一个 table view controller 中，然后再包装成一个 navigation 项目 (它负责提供 navigation controller 的标题、导航栏按钮等)。

整个虚拟 view 的层级都是这样构建的：我们使用当前的 app 状态，来构建出可以表示我们所在状态的虚拟 view 的树。

状态恢复

在 TEA 中，状态恢复的工作方式和初始 view 构建或者任意 view 的更新是一样的。每当保存的状态被解码时，driver 会向 AppState 索要虚拟 view 的层级，并将它同步到实际的 view 层级里去。Application delegate 中的状态恢复所做的，仅仅只是将它转发给 driver：

```
func application(_ application: UIApplication,
    willEncodeRestorableStateWith coder: NSCoder) {
    driver.encodeRestorableState(coder)
}
```

```
func application(_ application: UIApplication,
    didDecodeRestorableStateWith coder: NSCoder) {
    driver.decodeRestorableState(coder)
}
```

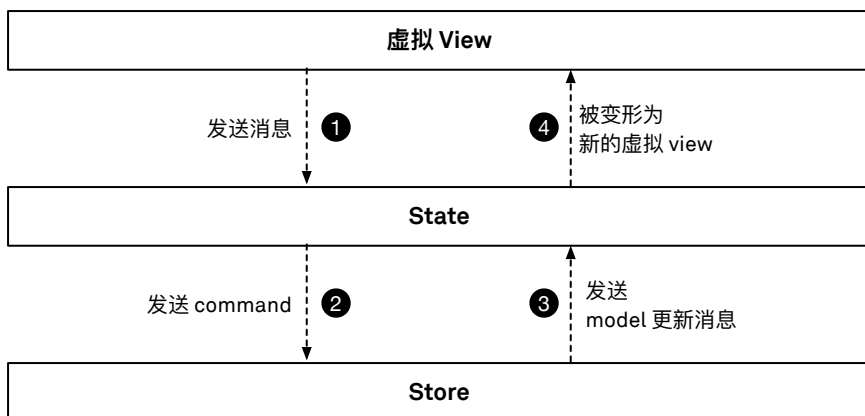
为了让它运转起来，我们的 AppState 还需要遵守 Codable，这样 driver 才能知道如何对它进行编码和解码。在 TEA 的工作方式下，状态恢复特性大体上是免费就能得到的。除了上面的两个 delegate 方法，状态恢复是独立于 UIKit 之外的。

更改 Model

在 TEA 中，底层 UIKit view 中的事件，会作为消息被 driver 接收到。Driver 接下来会将这些消息和 app 状态作为参数，传递给 update 函数 (我们在构建 driver 的时候通过 { state, message in state.update(message) } 进行了配置)。

注意，update 函数不会直接去更改 store 中的数据。如果它想要改变 store，它会返回一个 command。Command 对需要执行的行为进行描述，这可以是一个特定的 store 更新，或者是像网络请求这样的其他任务。Driver 负责解释这些 command 并执行它们。比如，它可能会按照需要对 store 执行变更。在本章后面的部分，我们会看到 command 的工作方式，以及了解为什么它在 TEA 中是很重要的部分。

我们通过订阅来观察 store，当它变更时，会向 update 函数发回消息，来对 app 状态进行更新，使其反映被更改后的 store 数据。之后，基于被更改后的 app 状态，新的虚拟 view 层级将被计算出来：



让我们来看看将一个项目从它的文件夹 view controller 的 table view 中删除的例子，来了解数据流动的方式。

步骤 1: View 将消息发送给 Update 方法

View 拥有关联的 action，我们将它们叫做消息。在 table view cell 的例子中，提交一个删除 action，将触发对 .delete 消息的发送。.delete 消息是定义在 app 状态中的 Message enum 的一部分：

```
extension AppState {  
    enum Message: Equatable {  
        case delete(Item)  
        // ...  
    }  
}
```

AppState.Message 在同一个地方对录音 app 中可以发送和处理的所有可能的消息进行了定义。

在 table view 的设置过程中，我们指定了删除时应该发送的消息：

```
let tv: TableView<Message> = folder.tableView(  
    onSelect: Message.select, onDelete: Message.delete)
```


构建 table view 的方法的第二个参数接受一个函数，这个函数返回的消息将在删除 action 触发的时候被发送。TEA 框架对 table view 的 data source 进行了配置，让它能够自动发送这个消息。Table view 中被删除的项目将被作为参数，传入这个函数中。

步骤 2: 从 Store 中更新触发删除

在我们的 app 中，被 driver 调用的 update 方法是 AppState struct 上的一个 mutating 方法。它对消息进行 switch 操作，对于 .delete 的情况，它返回一个 command：

```
mutating func update(_ msg: Message) -> [Command<Message>] {
    switch msg {
    case .delete(let item):
        return [Command.delete(item)]
    // ...
    }
}
```

因为 update 方法会改变 AppState，所以它被标记为 mutating。事实上，这也是唯一一个可以改变 AppState 的方法。对于 .delete 消息来说，我们仅仅只返回了一个 command，而没有进行任何状态变更。

Command 描述了由 driver 所解释执行的副作用 (比如改变 store 中的数据等)。执行 .delete command 会导致在 store 中这个删除的项目被移除。TEA 框架提供了一些独立于 app 的 command，而我们可以在 Command 的扩展中添加一些与 app 相关的 command (比如 .delete)。

步骤 3: 关于 Store 变更的更新通知

subscriptions 是 AppState 中的一个计算属性，它描述了在任意给定情况下，基于当前状态所想要观察的变更。其中至少包含了对 store 的变更的订阅：

```
extension AppState {
    var subscriptions: [Subscription<Message>] {
        var subs: [Subscription<Message>] = [
            .storeChanged(handle: { .storeChanged($0) })
        ]
        // ...
        return subs
    }
}
```

```

    }
}

```

这个订阅将告诉 TEA 框架，当 `store` 的内容发生变化时，向 `app` 状态发送一个 `storeChanged` 消息。

当 `app` 状态的 `update` 方法被 `.storeChanged` 触发时，它会更新状态中的 `folders` 数组。这个数组包含了需要存在于导航栈中的文件夹。因为文件夹在代码中是 `struct`，我们必须从 `root` 文件夹中获取它们的最新版本，然后将其作为关联值传入 `.storeChanged` 消息中去：

```

mutating func update(_ msg: Message) -> [Command<Message>] {
    switch msg {
    // ...
    case .storeChanged(let root):
        folders = folders.compactMap { root.find($0) }
        // ...
        return []
    }
}

```

要刷新文件夹的值，我们可以已有的文件夹数组进行映射，然后从新的 `root view controller` 中拉取当前值 (`find` 会基于文件夹的 `UUID` 进行递归查找)。我们这里使用 `compactMap` 而不是 `map`，是因为这些文件夹有可能不存在，在这种情况下，`find` 将返回 `nil`。

步骤 4: 从新的 App 状态中计算 View

在前一个步骤里，我们通过设定 `folders` 属性对 `app` 状态进行了改变。TEA 框架会注意到这个状态的改变，并重新计算整个虚拟 `view` 层级。这里我们只展示描述导航栈和它的文件夹 `view controller` 的部分：

```

extension AppState {
    var master: UINavigationController<Message> {
        let viewControllers: [NavigationItem<Message>] = folders.map { folder in
            let tv: TableView<Message> = folder.tableView(onSelect: Message.select,
                onDelete: Message.delete)
            return NavigationItem(title: folder.name,
                // ...
                viewController: .tableViewController(tv))
        }
    }
}

```

```

    return UINavigationController(viewControllers: viewControllers,
        back: .back, popDetail: .popDetail)
}

```

在第一行中的 `master` 属性里，通过映射状态的 `folders` 数组，我们生成了虚拟 view controller。在 `map` 中，每个文件夹的虚拟 table view 都通过下面的 `Folder` 扩展来生成：

```

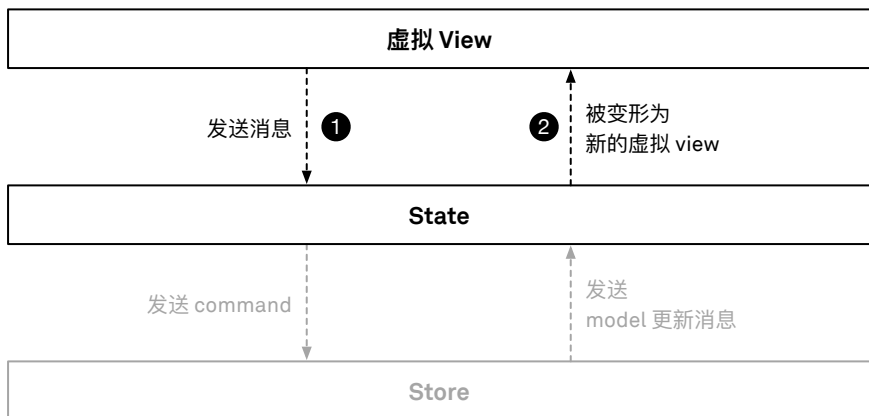
extension Folder {
    func tableView<Message>(onSelect: (Item) -> Message,
        onDelete: (Item) -> Message) -> TableView<Message>
    {
        return TableView(items: items.map { item in
            let text: String
            switch item {
            case let .folder(folder):
                text = "📁 \(folder.name)"
            case let .recording(recording):
                text = "🎧 \(recording.name)"
            }
            return TableViewCell(identity: AnyHashable(item.uuid), text: text,
                onSelect: onSelect(item), onDelete: onDelete(item))
        })
    }
}

```

注意，这些代码所生成的所有的 view 和 view controller 都是虚拟的：它们只是用来描述某种类型的 view 和 view controller 的 struct 或者 enum，而并非所对应的 UIKit 的实际类型。Driver 负责处理这个虚拟的 view 层级，并改变已经存在的 UIView 层级。只有两种层级之间不同的部分才会被改变。这个进行区分的步骤会十分聪明，它能够对 cell 插入和删除添加正确的动画。

更改 View State

在一个 TEA app 中，我们通过发送消息来改变 view state，从而更新 app 状态的 struct，之后 driver 会基于 app 状态，通过一个调用来生成新的虚拟 view 层级：



我们会和其他章节中所做的一样，通过同样的两个例子来展示 view state 的变更：更新播放按钮标题，以及推入一个文件夹 view controller。

示例 1: 更新播放按钮

步骤 1: 播放按钮发送消息

在第一步中，app 状态的 update 方法接受 TEA 框架在点击播放按钮时发送出的消息。我们将这个消息转发给 playerState：

```
extension AppState {
  mutating func update(_ msg: Message) -> [Command<Message>] {
    switch msg {
    case .player(let msg):
      return playState?.update(msg) ?? []
    // ...
    }
  }
}
```

Player state 的 update 方法会处理实际的 .togglePlay 消息：

```
extension PlayerState {
  mutating func update(_ action: Message) -> [Command<AppState.Message>] {
```

```

switch action {
case .togglePlay:
    playing = !playing
    return [Command.togglePlay(player)]
// ...
}
}
}

```

我们反转 `playing` 属性的布尔值，并且返回 `.togglePlay` command。之后 `driver` 解释这个命令，并且更改实际的音频播放状态。

步骤 2: 通过 State 计算新的 View 层级

由于 `app` 状态发生了改变，TEA 框架将从这个改变后的状态中生成新的虚拟 `view` 层级。这并不单单只包括改变的部分，而是一个从 `root view controller` 开始，一路向下到我们的播放按钮（以及其他所有应该显示在屏幕上的元素）的，包含了整个 `view/view controller` 层级的抽象表示。

我们没有展示整个 `view` 的构建，而只是将从 `player state` 的 `play view controller` 的计算作为例子进行说明。`PlayerState` 中实现了一个 `viewController` 属性，它使用内部的 `view` 属性来生成 `player view` 层级：

```

extension PlayerState {
    var viewController: ViewController<Message> {
        return .viewController(view)
    }
}

```

`view` 属性返回一个虚拟 `view`，它是一个虚拟的 `stack view`，包含了播放按钮和正确的标题：

```

extension PlayerState {
    var view: View<Message> {
        // ...
        return View<Message>.stackView(views: [
            .stackView(views: [
                // ...
                .space(height: 20),
                .button(text: playing ? .pause : .play, onTap: .togglePlay),
            ]),
        ])
    }
}

```

```

    )],
    .space(width: nil, height: nil)
  ])
}
}

```

有了这些代码，我们终于完成了整个流程的拼图：当我们创建一个虚拟按钮的 `view` 时，我们也为它指定了点击事件发生时所需要发送的消息 `.togglePlay` (这是 `PlayerState.Message.togglePlay` 的缩略形式)。

示例 2: 推入文件夹 View Controller

在前面的改变播放按钮标题的例子中，我们只是改变了当前层级中已经存在的 `view` 上的一个属性。在这个推入新的文件夹 `view controller` 的例子中，我们会同时更改 `view` 和 `view controller` 层级。不过，使用 TEA 框架的时候，我们所采取的步骤和上一个示例是一样的。

步骤 1: Table View 发送 `.selectFolder` 消息

当我们在 `table view` 中选择一个文件夹 `cell` 时，`AppState` 的 `update` 方法会收到 `.selectFolder` 消息。对于这条消息的对应的状态变更非常简单，我们只需要把选中的文件夹追加到状态的 `folders` 数组中：

```

extension AppState {
  mutating func update(_ msg: Message) -> [Command<Message>] {
    switch msg {
    case let .selectFolder(folder):
      folders.append(folder)
      return []
    // ...
    }
  }
}

```

步骤 2: 通过 State 计算新的 View 层级

在上面的例子中，改变后的状态导致了新的 `view` 层级的计算。我们在[构建一节](#)中已经对文件夹 `table view controller` 的创建方式进行了仔细的描述。

Elm 架构框架

在前面的实现细节一节中，我们看到了 TEA 的代码和像是消息，app 状态，虚拟 view 这些概念一起工作的方式。这些东西唯一的共同之处在于，它们和 UIKit 中的任何已知概念都完全不同。像是 UIView 实例或者 UIViewController 实例、通知、或者 target-action 在这里都不见踪影。这是因为 TEA 框架桥接填补了一般的 UIKit 构建和 TEA 代码所使用的抽象之间所有的空隙。

将这些东西联结起来的核心部件是 Driver 类。Driver 持有 model，将虚拟 view 层级和 UIKit 的 view 层级进行同步，接受 view action，并且执行像是在 store 中持久化数据这样的副作用。为了理解这个框架的实现，我们首先来看看虚拟 view 是如何被渲染成 UIView 和 UIViewController 实例的。

虚拟 View

在我们的实现中，虚拟 view 被定义为 enum。每种 view 都是 enum 中的一个 case：

```
indirect enum View<Message> {
  case _label(Label)
  case _stackView(StackView<Message>)
  // ...
}
```

每个虚拟 view 配置时所需要的数据都通过关联值进行存储。由于大部分虚拟 view 都拥有很多选项，我们将它们打包到 struct 中。比如，StackView struct 看起来是这样的：

```
struct StackView<Message> {
  let views: [View<Message>]
  let axis: UILayoutConstraintAxis
  let distribution: UIStackViewDistribution
  let backgroundColor: UIColor

  init(views: [View<Message>],
       axis: UILayoutConstraintAxis = .vertical,
       distribution: UIStackViewDistribution = .equalCentering,
       backgroundColor: UIColor = .white)
  {
    self.views = views
    self.axis = axis
  }
}
```

```

    self.distribution = distribution
    self.backgroundColor = backgroundColor
}
// ...
}

```

这些虚拟 view 通过一个渲染器 (renderer) 来变形为 UIView 实例：

```

struct Renderer<Message> {
    // ...
    mutating func render(view: View<Message>) -> UIView {
        switch view {
        case let ._label(label):
            let l = UILabel()
            render(label, into: l)
            return l
        case let ._stackView(stackView):
            let views = stackView.views.map { render(view: $0) }
            let result = UIStackView(arrangedSubviews: views)
            render(stackView, into: result)
            return result
        // ...
        }
    }
    // ...
}

```

对于 label 来说，渲染器会生成一个 UILabel 实例，并使用 ._label 的关联值来对它进行配置。对于 stack view，我们首先递归地将所有子 view 进行渲染，然后使用这些 arranged subviews 和其他像是间距和布局方向等选项进行配置。

为了避免每次变更都替换整个 view 和 view controller 层级，框架将在后续更新的时候作出努力，重用那些已经存在的 view。比如，更新一个 label 的代码看起来是这样的：

```

// Renderer
mutating func update(view: View<Message>, into existing: UIView) -> UIView {
    switch view {
    case let ._label(label):
        guard let l = existing as? UILabel else {

```



```

        return render(view: view)
    }
    render(label, into: l)
    return l
    // ...
}
}

```

我们只有在没有 UILabel 实例时才创建一个新的，否则，我们在 `render(_into:)` 方法中将重用已经存在的 label，并且只是去更新它的属性：

```

// Renderer
private func render(_ label: Label, into l: UILabel) {
    l.setContentHuggingPriority(UILayoutPriority.defaultHigh, for: .horizontal)
    l.text = label.text
    l.font = label.font
    l.textAlignment = .center
}

```

发送 View Action

我们以 view 被创建以及跟随事件流的方式作为开始，可以对 TEA 框架是如何响应 view action 并将消息传递给 app 状态的 update 方法这一问题进行研究。举例来说，录音项目的播放按钮定义如下：

```

return View<Message>.stackView(views: [
    .stackView(views: [
        // ...
        .button(text: playing ? .pause : .play, onTap: .togglePlay),
    ]),
    // ...
])

```

当我们创建一个虚拟按钮时，第二个参数将指定在按钮被按下时所发送的消息。渲染器在为这个虚拟按钮创建实际的 UIButton 实例时会用到这个信息：

```

// Renderer
private mutating func render(_ button: Button<Message>, into b: UIButton) {

```

```

b.removeTarget(nil, action: nil, for: .touchUpInside)
if let action = button.onTap {
    let cb = self.callback
    let target = TargetAction { cb(action) }
    strongReferences.append(target)
    b.addTarget(target, action: #selector(TargetAction.performAction(sender:)),
        for: .touchUpInside)
}
// ...
}

```

使用 UIKit 中标准的 target-action 机制，我们把一个内部的 TargetAction 对象作为 UIButton 点击事件的接收者。TargetAction 的目的是将 target-action 事件转换成消息，并把它们转发给一个回调函数。对于所有事件，所有的 view 都使用相同的 callback 函数。这个函数会被 driver 在初始化的时候传递给 render 方法：

```

final class Driver<Model, Message> where Model: Codable {
    init(_ initial: Model,
        update: @escaping (inout Model, Message) -> [Command<Message>],
        view: @escaping (Model) -> ViewController<Message>,
        subscriptions: @escaping (Model) -> [Subscription<Message>],
        initialCommands: [Command<Message>] = [])
    {
        // ...
        strongReferences = view(model)
        .render(callback: self.asyncSend, change: &viewController)
        // ...
    }
    // ...
}

```

asyncSend 方法被用作渲染器的回调，它会切换到主队列，然后调用 run 方法：

```

// Driver
func asyncSend(action: Message) {
    DispatchQueue.main.async { [unowned self] in
        self.run(action: action)
    }
}

```

```
func run(action: Message) {
    assert(Thread.current.isMainThread)
    let commands = updateState(&model, action)
    // ...
}
```

这里是 driver 将消息传递给 app 状态的地方。Driver 在这里调用的 updateState 函数，就是我们在 driver 初始化时所传入的 update 函数。

所有的 view action 都是这样进行处理的。其他的 UIControl 实例也和上面的按钮的例子一样。类似地，像是 table view 这种基于 delegate 的 view，可以通过内部的 delegate 对象来将 table view 的 action 转换为消息，并发送给由 driver 传进来的回调方法。

处理 Command

对于 TEA 框架，我们最后要考察的方面是它如何执行从 update 方法返回的 command。在我们现在的实现中，有和 app 相关的 command (比如从 store 中删除一个条目)，也有一些独立于具体 app 的 command (比如进行网络请求)。

作为例子，我们将介绍从 table view 中删除条目的情况，在这种情况下，app 状态的 update 方法中返回的是一个 .delete command：

```
// AppState
mutating func update(_ msg: Message) -> [Command<Message>] {
    switch msg {
    case .delete(let item):
        return [Command.delete(item)]
    // ...
    }
}
```

Driver 接受从 update 方法中返回的 command：

```
// Driver
func run(action: Message) {
    assert(Thread.current.isMainThread)
    let commands = updateState(&model, action)
    refresh()
```

```

    for command in commands {
        interpret(command: command)
    }
}

func interpret(command: Command<Message>) {
    command.run(Context(viewController: viewController, send: self.asyncSend))
}

```

传入 command 的 run 方法的 context 上下文，可以为我们提供 app 的 root view controller (用来显示一个 modal alert) 以及一个回调，当 command 被执行后，我们使用这个回调来将消息发回给 AppState。

.delete command 的实现很简单，它在 store 上调用 delete：

```

extension Command {
    // ...
    static func delete(_ item: Item) -> Command {
        return Command { _ in
            Store.shared.delete(item)
        }
    }
}

```

.delete command 并没有使用传入的 context，不过有其他 command 会需要用到它。比如，一个执行网络请求的 command，就需要在请求完成后将接收到的数据通过消息的方式发送回去：

```

extension Command {
    // ...
    static func request(_ request: URLRequest,
        available: @escaping (Data?) -> Message) -> Command
    {
        return Command { context in
            URLSession.shared.dataTask(with: request) {
                (data: Data?, response: URLResponse?, error) in
                context.send(available(data))
            }.resume()
        }
    }
}

```

```
}
```

我们已经看到了，在 TEA 框架中并没有什么魔法。Driver 是一个很简单的类，command 的实现也非常直接。最繁重的工作是创建虚拟 view，并编写它们的 render 和 update 方法。这部分会非常复杂，特别是对于像 table view 这样带有动画更新的 view 来说，更是如此。

测试

在 app 测试中，关键点通常是如何找到一个干净的，包含我们想要测试的逻辑的接口。在 TEA 中找到这些干净的接口异常简单 - 当我们在 app delegate 中构造 driver 时，它们就全都显现出来了：

```
@UIApplicationMain
class AppDelegate: UIResponder, UIApplicationDelegate {
    // ...
    let driver = Driver<AppState, AppState.Message>(
        AppState(rootFolder: Store.shared.rootFolder),
        update: { state, message in state.update(message) },
        view: { state in state.viewController },
        subscriptions: { state in state.subscriptions })
    // ...
}
```

在上面示例代码的闭包中，AppState 提供了三个关键的接口，也就是我们需要测试的部分：

1. 渲染虚拟 view (由顶层的 viewController 计算属性提供)
2. 执行变更 (由 update 独自处理)
3. 观察外部影响 (由 subscriptions 计算属性进行描述)

TEA 中的测试所做的仅仅是将这些函数中的一个拿出来，和 AppState 一起，对 app 功能的一小个切片进行测试。比如，给定一个拥有两个条目的 root 文件夹，我们可以测试是否有一个 table view controller 被构建，以及它是否包含了两个条目。这个测试从 AppState 的 viewController 属性开始，遍历虚拟 view 层级，直到相关层。最后，它对属性是否拥有期望的值进行断言。注意我们也对选择 cell 时所发出的消息是否正确进行了测试：

```
func testFolderListing() {
    // 构建 AppState
```

```

let vc = AppState(rootFolder: constructTestFolder()).viewController

// 遍历并且检查层级结构
guard case .splitViewController(let svc, _) = vc else { XCTFail(); return }
let navController = svc.left(nil)
let navItem = navController.viewControllers[0]
XCTAssertEqual(navItem.title, "Recordings")
guard case .tableViewController(let view) = navItem.viewController
    else { XCTFail(); return }

// 检查属性
XCTAssertEqual(view.items.count, 2)

XCTAssertEqual(view.items[0].text, "📁 Child 1")
XCTAssertEqual(view.items[0].onSelect, .selectFolder(folder1))

XCTAssertEqual(view.items[1].text, "🎙️ Recording 1")
XCTAssertEqual(view.items[1].onSelect, .selectRecording(recording1))
}

```

上面测试的结构和 MAVB 实现中的 `testTableData()` 测试十分相似。虽然 `view` 绑定器和虚拟 `view` 的实现非常不同，但它们同样是真实 `view` 的抽象，测试上的相似性也反映了这一点。

在测试文件夹选择时，我们不需要渲染虚拟 `view` 的层级。在前一个测试中，我们已经验证了 `.selectFolder` 消息可以被正确设置到文件夹的 `cell` 上。所以，我们可以从一个已知状态开始，使用 `.selectFolder` 消息来调用 `update` 方法，然后验证正确地发生了状态变化：

```

func testFolderSelection() {
    // 构建 AppState
    var appState = AppState(rootFolder: constructTestFolder())

    // 测试初始条件
    XCTAssertEqual(appState.folders.map { $0.uuid }, [rootUuid])

    // 推入新的文件夹
    let commands = appState.update(
        .selectFolder(Folder(name: "Child 1", uuid: uuid1, items: [])))

    // 测试结果
}

```

```

XCTAssert(commands.isEmpty)
XCTAssertEqual(appState.folders.map { $0.uuid }, [rootUuid, uuid1])
}

```

众所周知，在 MVC 和 MVVM 中，对导航进行测试是非常困难的，因为在这些架构中，不存在对导航状态进行封装的简单的接口。在上面的代码中，我们将测试分为了两个部分：在第一个测试中，我们验证了对于一个给定状态，view 层级可以被正确渲染。而另一个测试，则验证了对于给定的消息，状态可以正确变化。我们不需要测试 view 层级的变化，因为这是由框架所保证的。我们也不需要测试组件被正确连接，因为这部分工作也由框架来完成。

上面测试的结构和 MVC+VS 实现中的 `testSelectedFolder()` 类似。不过，在 MVC+VS 中，我们必须渲染 view 层级，才能更改状态，这是因为我们想要验证 view 层级和状态是相连接的。而在 TEA 中，我们并不需要这么做。

和其他架构模式不同，在 TEA 中对订阅进行测试非常容易。这类测试和 view 层级的测试相似。下面，我们会验证两件事：首先，我们要确保我们订阅了 store 的变化；其次，要验证得到的消息是 `.reloadFolder`。注意，在下面代码中的 `message` 是一个函数，它接受一个文件夹，并返回一条消息：

```

func testFolderStoreSubscription() {
    let appState = AppState(rootFolder: constructTestFolder())
    // 测试 store 订阅
    guard case let .storeChanged(message)? = appState.subscriptions.first
    else { XCTFail(); return }

    // 测试消息是 `.reloadFolder`
    let updatedFolder = Folder(name: "TestFolderName", uuid: uuid5)
    XCTAssertEqual(message(updatedFolder), .storeChanged(updatedFolder))
}

```

和 view 层级测试一样，对订阅的测试也总是遵循同样的模式：给定一个状态，调用 `subscriptions` 方法，然后验证结果值是否正确。我们不需要测试我们是否确实订阅了 Store，`.storeChanged` 的正确发送应该是由框架层级的测试来保证的。我们也不需要验证消息是否被传递，因为这是 driver 的职责。

我们到现在已经编写了三种类型的测试 (view state 测试，update 测试，以及订阅测试)，在此期间我们完全不需要调用任何一个框架的方法或者类。这就是 TEA 的特点，在 TEA 中，我们只需要编写纯函数并且测试输出值。

测试副作用

在 `update` 的测试中，我们验证了返回的 `commands` 数组是空数组。这个数组包含了需要执行的副作用。在我们的这个框架实现中，我们将 `Command` 建模为一个函数，这样我们就能够在提供内建 `command` 的同时，支持开放地进行扩展了。

如果我们想要测试返回的 `command` 是否正确的话，会有两种不同选择。我们可以执行 `command`，不过这通常会需要很多的配置，或者操作本身会很昂贵。或者，我们可以更改 `update` 方法的类型，让它返回一个协议。比如，考虑下面的 `RecordState` 的 `update` 方法：

```
extension RecordState {  
    // ...  
    mutating func update(_ message: Message) -> [Command<Message>] {  
        switch message {  
            // ...  
            case let .save(name: name):  
                guard let name = name, !name.isEmpty else {  
                    // 显示无法保存...  
                    return []  
                }  
                return [Command.saveRecording(name: name, folder: folder,  
                    url: recorder.url)]  
            // ...  
        }  
    }  
}
```

在这个方法中，我们使用了三种不同的 `command`。想要使这些 `command` 可测试，我们首先将它们集中到一个接口中（为了简单，我们只展示了一个 `command`）：

```
protocol CommandProtocol {  
    associatedtype Message  
    // ...  
    static func saveRecording(name: String, folder: Folder, url: URL) -> Self  
}
```

因为这个方法的定义和我们的 `Command` struct 完全一致，所以我们可以直接将 `Command` 声明为遵守该协议，而不需要做任何额外的事情：


```
extension Command: CommandProtocol {}
```

最后，我们可以将 update 方法的返回改为 C 的数组，这里 C 是满足新的协议的泛型类型。当我们使用 update 来配置 driver 时，编译器将会推断出 C 应该和 Command 相等。

```
extension RecordState {  
  // ...  
  mutating func update<C>(_ message: Message) -> [C]  
    where C: CommandProtocol, C.Message == Message {  
    switch message {  
    // ...  
    case let .save(name: name):  
      guard let name = name, !name.isEmpty else {  
        // 我们无法存储...  
        return []  
      }  
      return [C.saveRecording(name: name, folder: folder, url: recorder.url)]  
    // ...  
    }  
  }  
}
```

要进行测试，我们可以创建一个同样满足 CommandProtocol 的 enum。定义比较机械化：对于协议中的每个静态方法，我们定义一个 case。为了简明，我们省略了协议的方法实现，这个实现将构建并返回下面的 case：

```
enum CommandEnum<Message>: CommandProtocol {  
  // ...  
  case _saveRecording(name: String, folder: Folder, url: URL)  
  // ...  
}
```

在测试中，我们现在将结果类型强制为 CommandEnum，来指导编译器返回正确的类型。下面，我们将两个测试合二为一：首先，我们检查如果在没有 name 值的情况下调用 save 的话（由于用户可能没有在弹出的对话框中输入任何文字），不会返回 command。然后我们检查当提供一个值时，可以获取到一个 command，来将录音存储到 store 中，并在磁盘上进行持久化：

```
func testCommands() {  
  var state = RecordState(folder: folder1, recorder: sampleRecorder)
```

```
let commands: [CommandEnum<RecordState.Message>] =
    state.update(.save(name: nil))
XCTAssert(commands.isEmpty)

let commands1: [CommandEnum<RecordState.Message>] =
    state.update(.save(name: "Hello"))
guard case ._saveRecording(name: "Hello",
    folder: folder1, url: sampleRecorder.url)? = commands1.first
else { XCTFail(); return }
}
```

在处理比如像前面一节提到的 `request` 这样的异步 `command` 时，这类操作会变得更加有用。我们可以验证加载了正确的 URL，而且网络的响应被转换为了正确的消息。在另一个的测试中，我们还可以验证对于这个给定的消息，状态正确地发生了变化。

讨论

TEA 的主要好处是不需要可变的 `view` 层级：我们只需要描述对一个给定的 `app` 状态，`view` 层级应该有的样子，而不需要编写 `view` 层级为了要响应某个特定 `action`，而从状态 A 转变为状态 B 的代码。通过对我们的 `view` 进行这样声明式的定义，我们避免了一整类的潜在 `bug`。我们太熟悉那些 `view` 进入无效状态的 `bug` 了：控件被错误地设为无效，弹出的窗口无法关闭，或者加载菊花一直转个不停，也不消失等。这些只是这类错误的一小部分例子。TEA 为 `view state` 在 `app` 中各个部分的更新，建立了一套一致而且容易遵守的机制，这减轻了管理所有状态的压力。

MVC+VS 的实现也想要达到类似的目标，它的做法是将所有的 `view state` 看作是 `model` 层的一部分，并且严格地使用观察者模式来进行 `view state` 的更新。然而，所有的这一切都是通过约定来实现的：我们如果想要的话（或者我们不知道代码库中应该使用的模式时），是可以直接更改 `view` 层级的。TEA 将这个约定更进一步：作为 TEA 框架的用户，我们无法访问到 `view` 的层级。作为 `view action` 或者 `model action` 的响应，我们唯一能做的事情是描述 `view` 层级应有的样子。和 MAVB 不同，TEA 并不需要依赖某个响应式编程库。

译者注：虽然 TEA 确实不需要依赖响应式编程，但是它却带来了一个更大的依赖，那就是 TEA 框架本身。和响应式编程库相比，TEA 框架带给项目的改变更加深入和全面。一旦采用，你甚至就很难再回头，这也是作者在本章中再三提醒现在还不要将这个架构用到大型生产项目中的原因。不过好消息是，TEA 的学习曲线会比大多数响应式编

程框架要来得平缓得多，你不需要知道 `combineLatest` 或者各种 `Subject` 类型的区别，在 TEA 中编程只需要整体地处理各种状态即可。

在 TEA 中，除了 `view` 更新的流程被强制为一致以外，其他的副作用所发生的方式也被强制一致。举例来说，在 `app` 状态的 `update` 方法中，我们不能直接执行网络请求，也不能在它返回时更新 `view` 的层级，要做到这些，唯一的途径是从 `update` 方法中返回一个 `command`；我们并不会去自己执行请求，我们要做的只是提供一个对需要发生的事情进行描述的消息，然后让 `driver` 做这件事。之后，`driver` 会负责执行 `command`，并且在完成后发送一条另外的消息。

不过，TEA 也并非完美无瑕。在 iOS 上实现 TEA 框架的一个问题是，UIKit 并不是为了这样使用而设计的。有一些 UIKit 组件会自主地改变它们的内部状态，而不让我们有机会拦截这个状态的改变（我们只是在一切发生之后才能得到通知）。我们已经在 [MVC+VS](#) 一章中说过这个问题了，而在 TEA 中它又再次出现了。因此，使用 TEA 框架来对 UIKit 进行的抽象将始终无法做到完美。

另外一个主要的挑战是，要如何把动画集成到 TEA 框架中。当对当前的 `app` 状态进行 `view` 层级描述时，我们对具体到达该状态的方式一无所知。不过，这其中往往包含动画等重要信息（比如，某个动画应该在响应用户交互时被播放，而在像是状态恢复这样的，通过程序到达当前状态的时候不播放）。另外，对于（从 `view` 层级 A 动画过渡到层级 B 时）中间的 `view state`，要如何在 TEA 的 `app` 状态 `driver model` 中对其表示，也还尚不清晰。在本书编写的时候，这还是一个未解决的问题，在我们能得到一个良好的解决方式之前，还需要进行更多的研究。

译者注：很多读者可能还有一个疑问，那就是 TEA 的性能如何？在每次状态改变时，就算只是点击了播放按钮，改变了一个小小的 `label` 时，我们都需要根据新的状态重新计算虚拟 `view`。另外，虽然我们在 TEA 框架中实现了一些对 `view` 的重用，但是这种级别的优化是否足以支撑真正规模的 `app`。实际上，大部分的 `app` 并不会需要展示成千上万个 `cell` 的 `table view`，也没有太大的 UI 压力。在这种情况下，虚拟 `view` 的计算基本上不会是性能瓶颈。TEA 在从虚拟 `view` 渲染真实的 `UIView` 时，使用了 `view` 重用的方式，这在一定程度上也可以减少 `view` 的创建（这在很多时候是 UI 性能开销最大的部分）。而像是 `React Native` 这样的框架其实也使用了类似的做法，而实践下来验证的结果还算理想。不过，显然，现在的方式是无法支撑无限扩展的，性能上也会比传统的架构有所损失。TEA 确实需要一些新的想法或者方式，来确保这部分性能的可扩展性。

经验和教训

从 TEA 中我们可以看到，为整个 app 状态明确地建模，并在这个状态发生改变时更新整个层级，会是一个不错的主意。TEA 将这个想法应用到了整个 app 中，将它发挥到了极致，不过，同样的原则也可以只用在某几个特定的界面，或者甚至使用在一个界面中的某个部分里。在 [MVC+VS 一章](#)中，我们谈论了一种关于 view state 的轻量级实现，在本质上来说，想法是一样的。

声明式 View

另一个好处在于，我们可以通过声明式的方法来建立 view。同样地，TEA 通过引入虚拟 view 层来代表各种 view。不过，我们可以将同样的理念用在那些是实现起来更容易，而且经常会用到的组件上去。

在 [Swift Talk #07](#) 中，我们展示了一种用 enum 来声明式定义 stack view 的技术。我们可以通过定义我们想要放入 stack view (或者说，stack view 的 arranged subview) 的那些元素开始：

```
enum ContentElement {
    case label(String)
    case button(String, () -> ())
}
```

然后，我们在 ContentElement 的扩展里实现一个 view 属性，来从这些 enum case 中创建实际的 UIView：

```
extension ContentElement {
    var view: UIView {
        switch self {
            case let .label(text):
                let label = UILabel()
                label.text = text
                return label
            case let .button(title, callback):
                return CallbackButton(title: title, onTap: callback)
        }
    }
}
```

CallbackButton 是对 UIButton 的封装，它可以接受一个回调。你不妨阅读一下 [Swift Talk #07](#) 的文本内容来了解实现的细节。

我们接下来会使用 content element 上的 view 属性，来创建一个 UIStackView 的便捷初始化方法 (convenience initializer)：

```
extension UIStackView {
    convenience init(vertical: [ContentElement]) {
        self.init()
        translatesAutoresizingMaskIntoConstraints = false
        axis = .vertical
        spacing = 10

        for element in elements {
            addArrangedSubview(element.view)
        }
    }
}
```

这可以让我们像这样来快速构建 stack view：

```
let stackView = UIStackView(vertical: [
    .label("Name: \(recording.name)"),
    .label("Length: \(timeString(recording.duration))"),
    .button("Play", { self.audioPlayer.play() }),
])
```

现在这个实现只能显示静态内容。如果我们想要渲染不同的 content element，我们需要创建一个新的 stack view，然后用它来把老的替换掉。对于很多使用场景来说，这不成问题。不过如果 subview 的类型都匹配的话，我们也可以在 UIStackView 上创建一个方法，来依据 content element 数组中的新的数据，更新已经存在的 arranged subview。如果 subview 类型不一致的话，就还是需要使用新的 view 替换掉原来的 subview。

结合 MVC+VS 一章中所述的轻量级 view state 方法，我们可以在一个已经存在的 app 中的一小部分上，近似地实现 TEA 这样的声明式虚拟 view。我们在 [Swift Talk](#) 的 [构建表单库](#) 的系列中，通过一个更复杂的例子展示过这些技术了。

描述副作用

TEA 在处理像是网络请求这样的副作用时所使用的方​​式十分有趣。因为架构被构建时，TEA 并没有给我们在 app 状态的 update 方法中留有直接处理这类任务的方式。因此，TEA 引入了 command 的概念：要执行网络请求，我们需要返回一个描述该请求的 command，然后让 driver 去实际进行执行。

虽然我们在标准的 MVC 代码库中不会面临这样的限制，但是 TEA 的 command 的思想也值得借鉴：我们严格地将要做什么和具体执行任务的代码分开，来使两者不互相纠缠在一起。这在执行异步任务的时候会非常有用：它可以将那些简单的同步代码从难以测试的异步代码中分离出来。我们在 [Swift Talk 的第一集](#)中，就已经介绍了使用这种技术的一个例子了。

这个想法非常简单，我们创建一个类型，它包含了一个网络请求所需要的所有信息，甚至包括如何将 response 数据转变为一个可用的数据类型的函数：

```
struct Resource<A> {  
    let url: URL  
    let parse: (Data) -> A?  
}
```

这个 struct 可以包含更多的信息，比如请求所使用的方法 (GET 或者 POST)，以及 content 的类型 (json 或者 text) 等。这样对于网络请求的抽象，可以让我们很简单地测试解析 (parse) 代码：我们只需要测试 Resource 的 parse 函数是否能对给定输入返回我们所期望的结果就行了，而且这个测试会是同步的。实际发送请求的异步任务现在就只需要编写一次就可以了：

```
final class Webservice {  
    func load<A>(_ resource: Resource<A>, completion: (A?) -> ()) {  
        URLSession.shared.dataTask(with: resource.url) { data, _, _ in  
            let result = data.flatMap(resource.parse)  
            completion(result)  
        }.resume()  
    }  
}
```

虽然这个例子还缺少许多特性 (错误处理，认证等)，但它的原理非常清晰：所有按照请求而不同的东西都被放到 resource struct 中，web service 的 load 方法的所剩下的唯一任务，就是与网络 API 进行交互。