objc.io | objc 中国

App 架构

技术预览版

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

版本 0.1 (2018 年 5 月)

ObjC 中国
在中国地区独家翻译和销售授权

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

关于本书

> **注意：**本书和相关视频还处于编辑阶段，并没有正式发布，最终的内容可能会有所改变。对于本书中现存的错误和不足，我们深表歉意！

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。从已经经过广泛验证的模式，到还处于实验阶段的模式，我们所挑选的模式有：

- → 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，并探讨它们在实践中的使用方式。

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的约束，任何一种模式都有可能是最佳选择。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工具，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。或者说，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每种模式在解决问题时的优势，以及使用类似策略在**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，看看这些思想如何跨越不同的模式并解决问题。

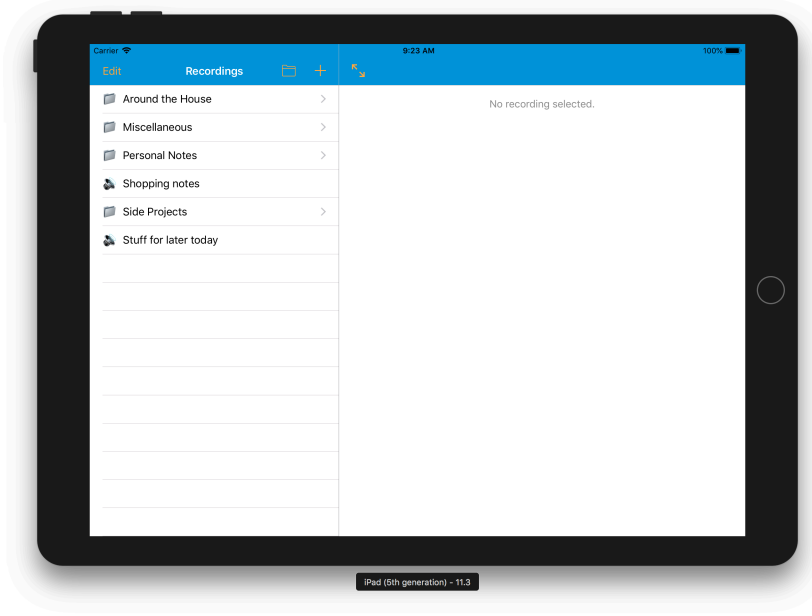
本书将会为你展示，对于每个问题，app 的架构包含了多种解决方案。当被正确实现后，所有的解决方案最终都将为用户呈现出同样的结果。这也就是说，app 架构的选择所关乎到的是我们程序员自己的幸福。我们想要在背后解决的潜在问题是什么？我们需要个别单独考虑的问题又有哪些？我们在何处需要自由度？又在何处需要稳定性？我们在哪里需要抽象？我们又在哪里需要简化？通过一次次地思考这些问题，我们对使用的架构进行选择。

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文本的 alert 弹框组成。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 在副栏进行显示。

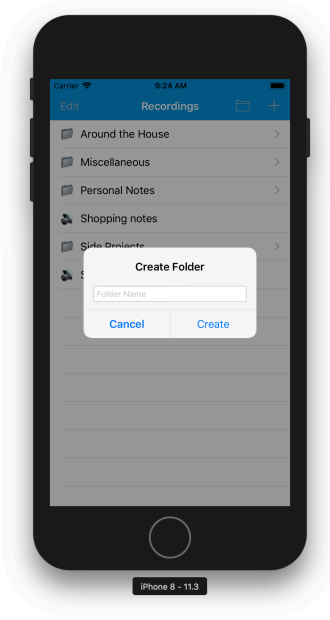
在选择本书的示例 app 时，我们列了一个判断标准，录音 app 满足了表里的所有项目。对于展示架构模式来说，它足够复杂；而放到一本书中时，它又足够简单。app 中存在导航关系，其中的一些视图是实时更新的，而不像很多 app 中那样只存在静态视图。对模型层，我们实现了持续化的存储，每个变更都会被自动保存到磁盘。我们还为其中的两个版本加上了网络的支持。这个 app 对 iPhone 和 iPad 都做了适配，最后，我们还支持冷启动的状态恢复。

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湮没。我们的录音 app 恰到好处地在两个极端之间做到了平衡。

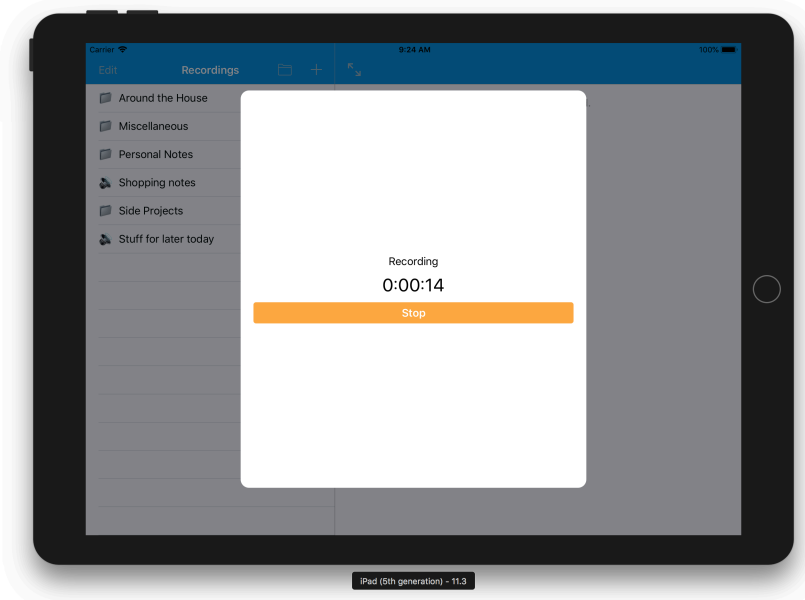


App 开始时会显示一个根目录的 FolderViewController。这个 controller 存在于一个 navigation controller (导航控制器) 中，而这个 navigation controller 又是一个 split view controller 的 master controller。在 iPhone 上，这个 controller 将以全屏模式被展示，而在 iPad 上，它将会被显示在屏幕左侧 (这是 iOS 上 UISplitViewController 的标准行为)。

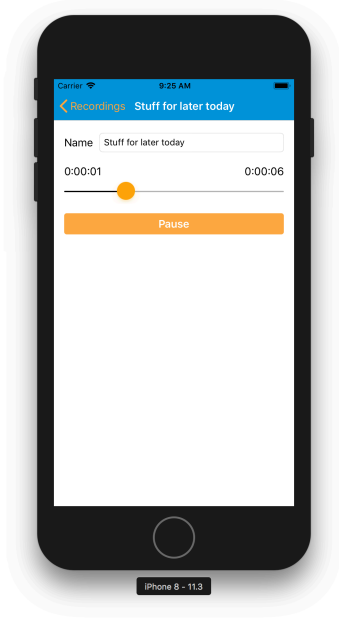
一个文件己可以包含具体的录音文件以及其他的文件夹。FolderViewController 允许我们添加新的文件夹和录音，或者删除已有的项目。



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

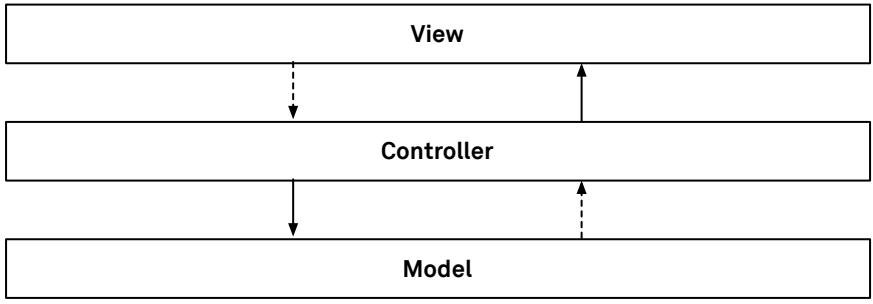
1

> **注意：**本书和相关视频还处于编辑阶段，并没有正式发布，最终的内容可能会有所改变。对于本书中现存的错误和不足，我们深表歉意！

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 model 层可见，并允许用户交互，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。在树干部分是整个屏幕，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标签这样的单一 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，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。

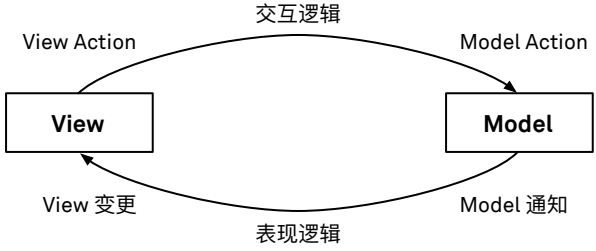
定义一个 model 层的最重要的理由是，它为我们的程序提供一个表述事实的单一来源，这会让逻辑清晰，行为正确。这样一来，我们的程序便不会被应用框架中的实现细节所支配。

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 app 的 Android，macOS 或者 Windows 版本中。

App 的本质是反馈回路

View 层和 model 层需要交流。所以，也就存在两者之间的连接。假设 view 层和 model 层是被清晰地分开，而且不存在不可解耦的联结的话，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**。

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。在其他一些架构中，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**。当某个树突对象或者中间层对象只能够通过 model 发出的通知进行创建和更新 (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项的。

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，让逻辑可以在信息在部件之间传输时得以表达。

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 (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 smodel 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

> **注意：**本书和相关视频还处于编辑阶段，并没有正式发布，最终的内容可能会有所改变。对于本书中现存的错误和不足，我们深表歉意！

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，采用绑定的方式来在 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 来在 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。

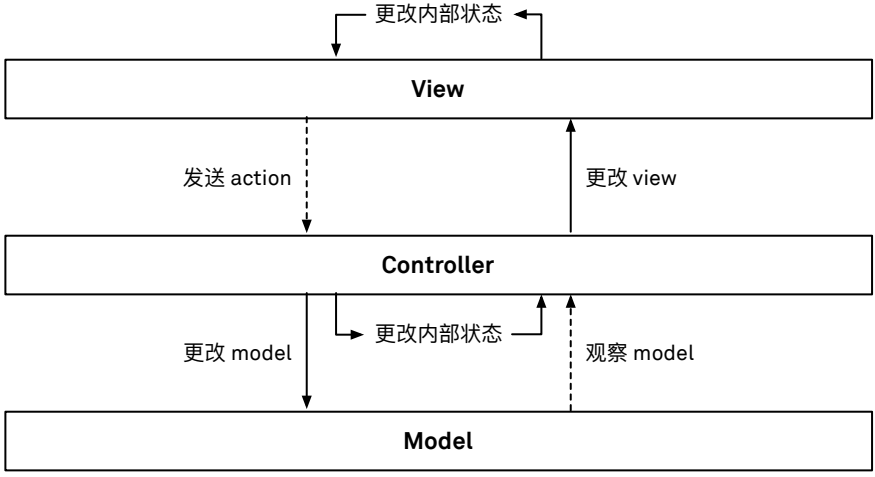
当然，除了这五种模式以外，iOS 的 app 设计模式还有非常多的选择。在讨论每种模式的时候，我们为你揭示我们选择这种模式进行介绍的原因。在本章的最后，我们也会简单讨论没有被我们选中在本书中进行介绍的一些其他模式。

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，准备所有用来展示的数据，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。

如果我们在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 (通过 storyboard 或者代码设置的) 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 可以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 可以按需要被存储在 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 书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，来测试结果是否是我们想要的效果。

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实现所使用的与 MVC 协作的方式，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，我们之后会看到，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，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+C)

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。

相对于 MVC，对于 MVVM 的定义部分，它在每个场景中使用 **view-model** 来描述场景中的表现逻辑和交互逻辑。

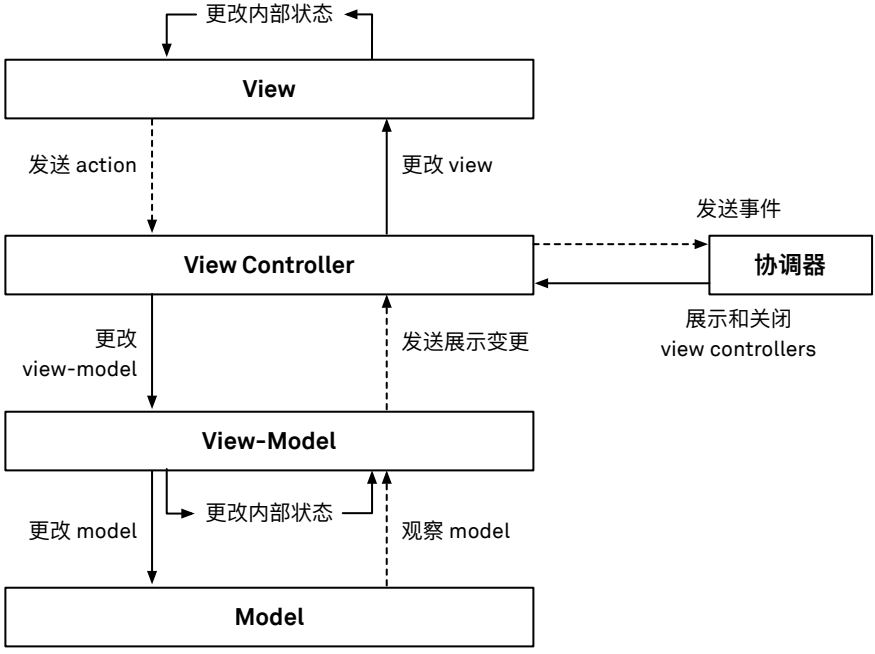
View-model 在编译期间不包含对 view 或者 controller 的引用。它暴露出一系列属性，用来描述每个 view 在显示时应有的值。把一系列变换运用到底层的 model 对象后，就能得到这些最终可以直接设置到 view 上的值。实际将这些值设置到 view 上的工作，则由预先建立的绑定来完成，绑定会保证当这些显示值发生变化时，把它设定到对应的 view 上去。响应式编程是用来表达这类声明和变换关系的很好的工具，所以它天生就适合 (虽说不是严格必要) 被用来处理 view-model。在很多时候，整个 view-model 都可以用响应式编程绑定以声明式的方式进行表达。

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：现在不需要关心如何展示其他的 view controller 了。因此，它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。

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 负责观察 model，并将 model 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，但也可以使用任何的观察机制。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 层级将有协调器进行管理。

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。

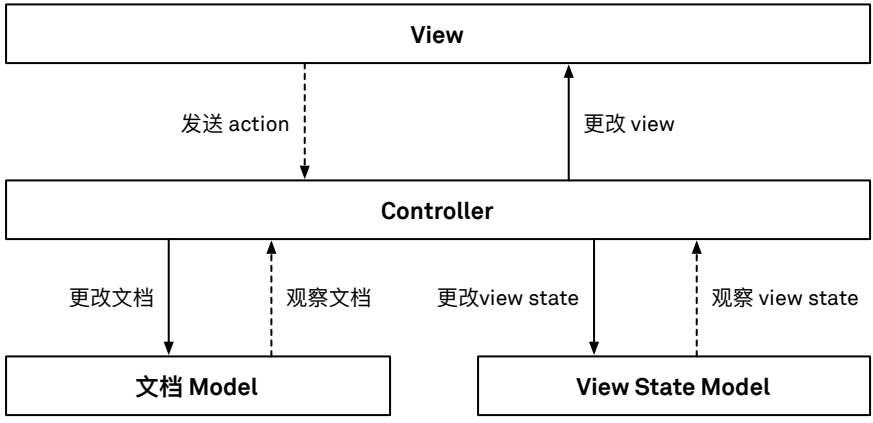
MVVM 和我们在 MVC 历史中提到的 MVP 模式非常类似. 不过，在 Cooper 和 Peters 的论述中，MVVM 中 view 和 view-model 的绑定需要明确的框架支持，但 presenter 是通过传统的手动方式来传递变化。

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 的更古老的模式，它们在 Cocoa 和其他平台上已经存在了数十年之久。

Model-View-Controller+ViewState

MVC+VS 是为标准的 MVC 带来单项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，我们明确地将所有的 view state 定义和表示在一个新的 model 对象中，我们把它叫做 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，这样变更的通讯非常直接。在表现或者交互逻辑部分，我们从来不从 view 中去读取 view state ，而是从 view state model 中去获取它们：



结果所得到的图标和 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，是一个独立的 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，这使得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，所有的变更都要通过文档 model 和 view state model。

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 构建整个 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，我们可以书写两种测试。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。

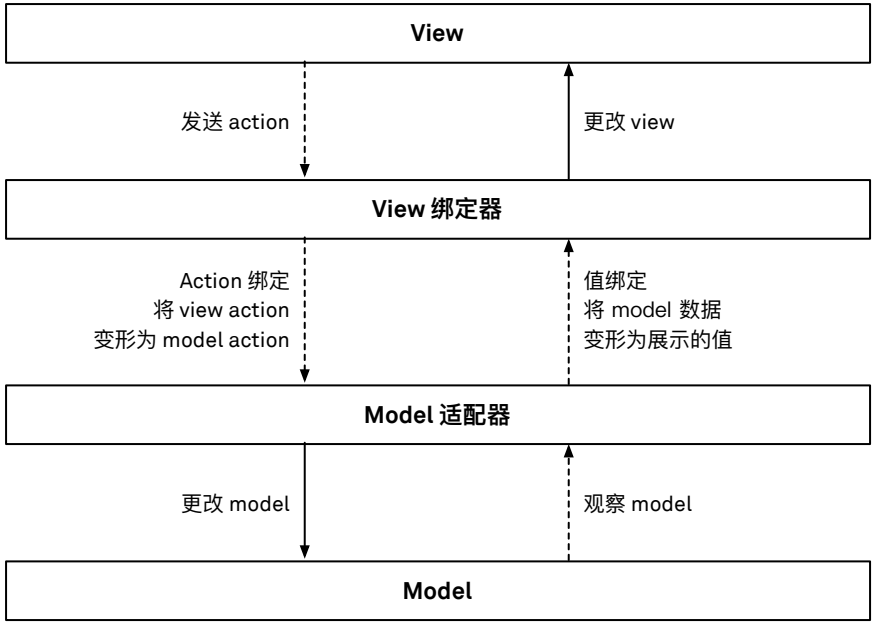
View 绑定器是 view 类 (或者 view controller 类) 的封装类：它构建 view，并且为它暴露出一个绑定绑定列表。一些绑定为 view 提供数据 (比如，一个标签的文本)，另一些从 view 中发出事件 (比如，按钮点击或者导航变更)。

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 也是一个声明式的模式：你声明你的 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 Cocoaview 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，输出端会与一个 model 适配器相连接，view 事件会被通过绑定进行变形，成为 model 适配器可以理解的一条消息。这条消息之后被 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 和 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，用于试验并有一些奇怪。我们在这儿包含它的意义在于，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一些经验教训：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 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。

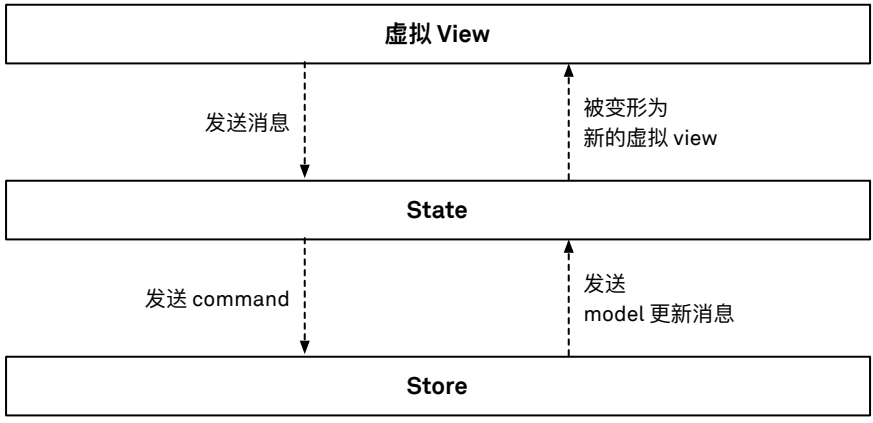
Elm 架构 (TEA)

TEA 和 MVC 有着根本上的不同。在 TEA 中，model 和所有的 view state 被集成称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) 将比较虚拟 view 层级和 UIView 层级，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 框架中的驱动 (driver) 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，这些命令会被驱动所执行。在我们的例子中，最重要的命令是更改存储中的内容，存储反过来又被驱动所持有的订阅者监听。这些订阅者可以触发消息改变状态，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 (也就是驱动)。运行时系统拥有状态，存储是一个单例。

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。驱动可以接收这些消息，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)，比如我们想在存储中进行的改动。驱动会截获该命令并执行它。TEA 让 view 不可能直接对状态或者存储进行更改。

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，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，因为驱动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命令。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 来展示了两种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。将网络作为 model 层的扩展，我们将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已经被用在发布 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 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。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。

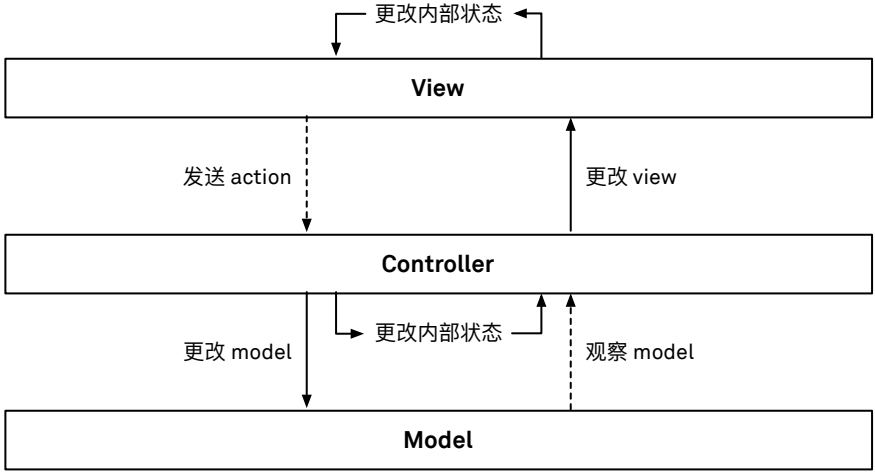
Model-View-Controller

3

> **注意：**本书和相关视频还处于编辑阶段，并没有正式发布，最终的内容可能会有所改变。对于本书中现存的错误和不足，我们深表歉意！

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 层是作为主心骨来参与形成 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，并且避免依赖程序的其他部分。正因如此，将其他两部分组合起来成为完整的程序，就是 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；它们将保持独立可重用。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：它将特定 app 相关的知识传达给程序中的其他部件。

存储在 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。

这就是 view controller 作为身份对象的设置方式。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 (最上层分割 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。录音被从外界设定，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。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)
    }
```

```
    }
}
```

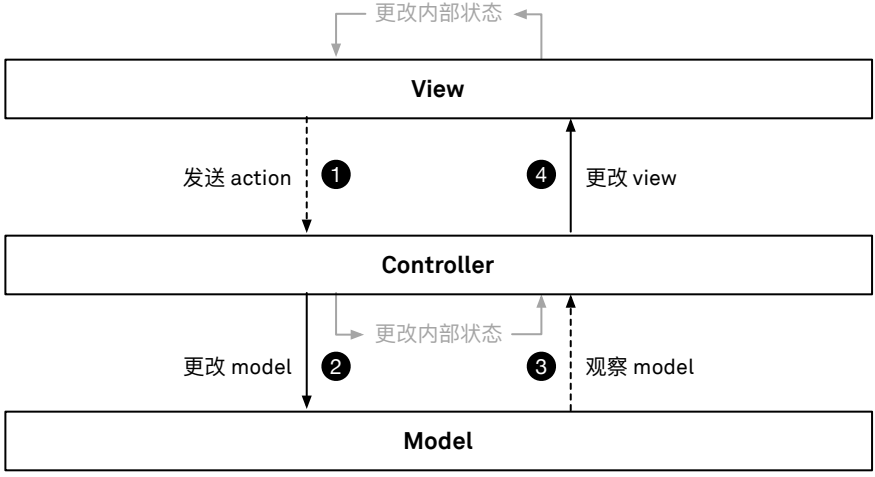
在对 uuidPath 进行解码后，FolderViewController 必须检查条目是否依然存在于存储之中，这样它才能将 folder 属性设置为该条目。如果该条目已经不在存储之中，那么 FolderViewController 必须尝试将它自身从导航 controller 的 view controller 列表中移除。

更改 Model

在最广泛的 MVC 的说明中，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。

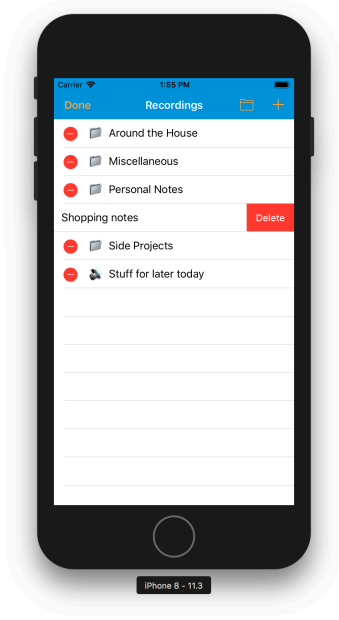
观察者模式是在 MVC 中维持 model 和 view 分离的关键。这种方式的优点在于，不论变更究竟是源自哪里 (比如，view 事件、后台任务或者网络)，我们都可以确信 UI 是和 model 数据同步的。而且，在遇到变更请求时，model 将有机会拒绝或者修改这个请求：



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 data source 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。然后它告诉存储应该保存数据，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存储的 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 为存储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)
}
```

上一步里在存储上调用的 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

当存储更改的通知到达时，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，都重新加载列表的数据。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，也就是发生变化的行号，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，它直接使用的是 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 chapter 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：值类型在赋值时将发生赋值，因此存储中对 folder 的更改不会影响到 view controller 中的 folder，我们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。

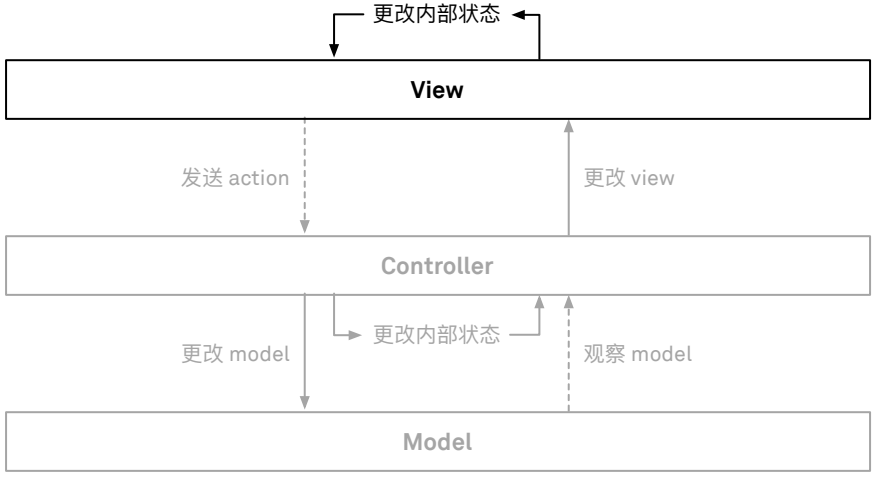
更改 View State

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。

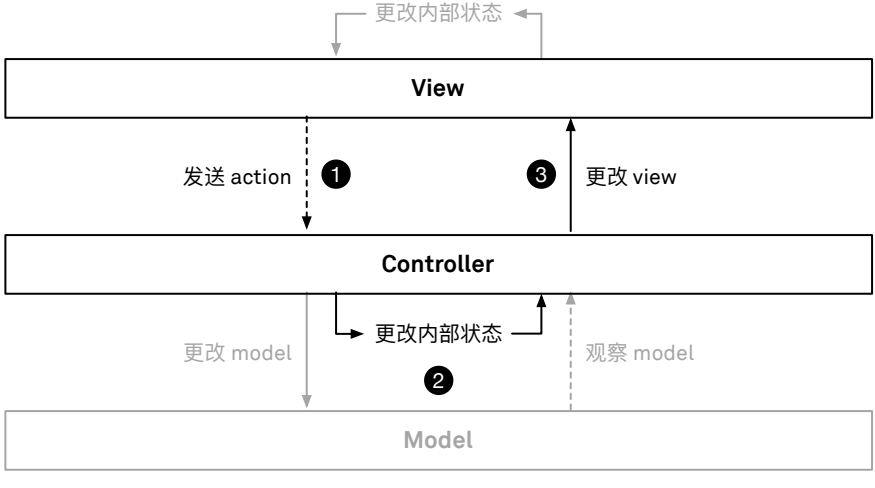
在 MVC 中，这些被我们统称为 view state 的 “其他” 状态没有被包含在模式的描述中。依照传统的面向对象的原则，任意的对象都可以拥有内部状态，这些对象也不需要将内部状态的变化传达给程序的其余部分。

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，反过来，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只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起怪了。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，也就是一些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，这种测试需要一个存储对象，所以我们可以创建一个 (只在内存中存在的) 不包含 URL 的存储，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 是存储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。构建了存储之后，我们现在需要一个使用该存储的 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存储的 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 中进行了删除。

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 推入导航栈时被满足。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，是与框架和类重度集成的，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正确实现提供任何检查或者内部机制。解决方法是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 Controlle

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会开始变得困难。

不过关于大的 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，并且限制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来正确更新某个值。比如文件夹 view controller 中的 folder，是在父文件夹 view controller 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 实际上无法比通知的方式更加稳定。由于它需要每个被观察的属性都声明为 dynamic，这也让它在 Swift 中远远没有在 Objective-C 中流行。

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**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在存在空隙：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，而不需要对存储本身进行任何更改，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]
  }
}
```

存储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。但是它们依然是与 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，它恩可以只是场景所拥有的子对象。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。

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是，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。这样的变更能我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，不适用 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 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。这样依赖，所有的 view controller 就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。虽然并不是很容易就能看到重构的可能性，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 是一个可重用的 controller 对象，它被配置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, numberOfRowsInSection 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)"
  })
```

在我们的小型的示例 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) {
    textLabel!.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有一行而已。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，我们希望你意识到，虽然 MVC 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。

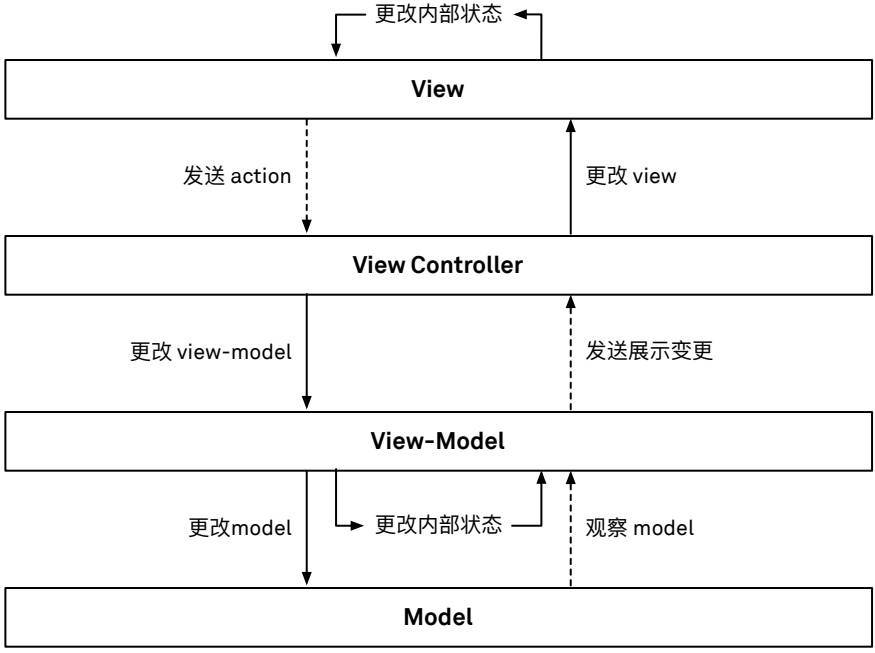
当然啦，有很多其他方式，可以让 app 设计中我们所要面临的挑战发生改变。在接下来的章节里，我们会讨论一系列替代的设计模式，它们中的每一个都从不同的方面解决了 app 设计中的某些问题。所有那些模式都包含了很多想法，你可以将这些想法运用在 MVC 程序中，来务实地解决特定的问题。你也可以通过它们来创造你自己的混合模式或者编程风格。

Model-View-ViewModel+协调器 (MVVM-C)

4

> **注意：**本书和相关视频还处于编辑阶段，并没有正式发布，最终的内容可能会有所改变。对于本书中现存的错误和不足，我们深表歉意！

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。

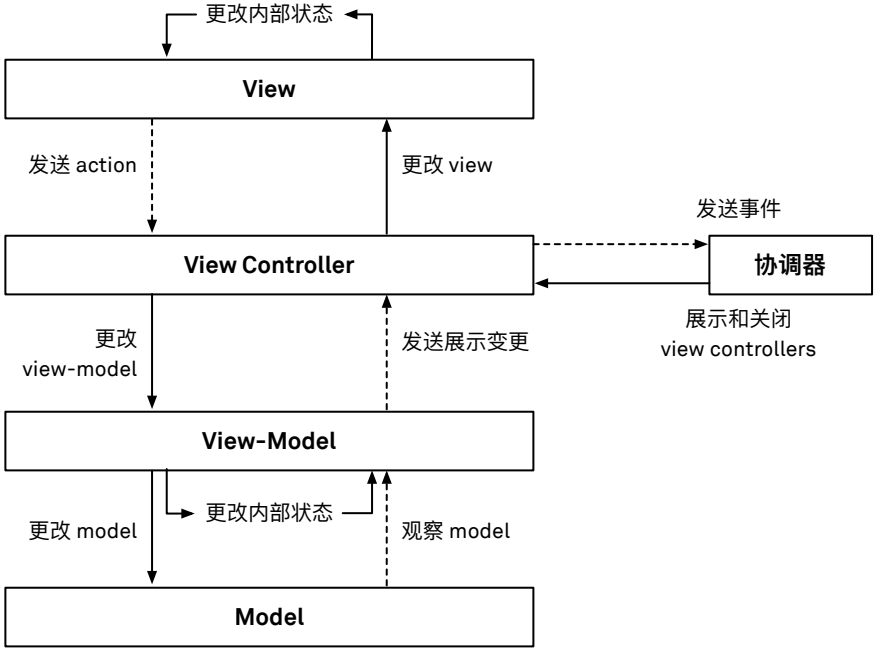
两者结合起来，对 MVC 的两个最大的被人诟病的地方进行了修正。第一项通过把 model 相关的观察和变形从 controller 层移除出去，减少了 controller 所需要承担的责任。第二项为场景

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来为示例 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，我们认为就算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，我们在本章的后面的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，协调器并非是强制存在的部件，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的特定场景的使用了。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，这么做会很有帮助。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种部件，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 和 skipWhile (和 dropFirst 与 dropWhile 类似) 以及 take and 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，将对 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存储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 的目的是将所有的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：

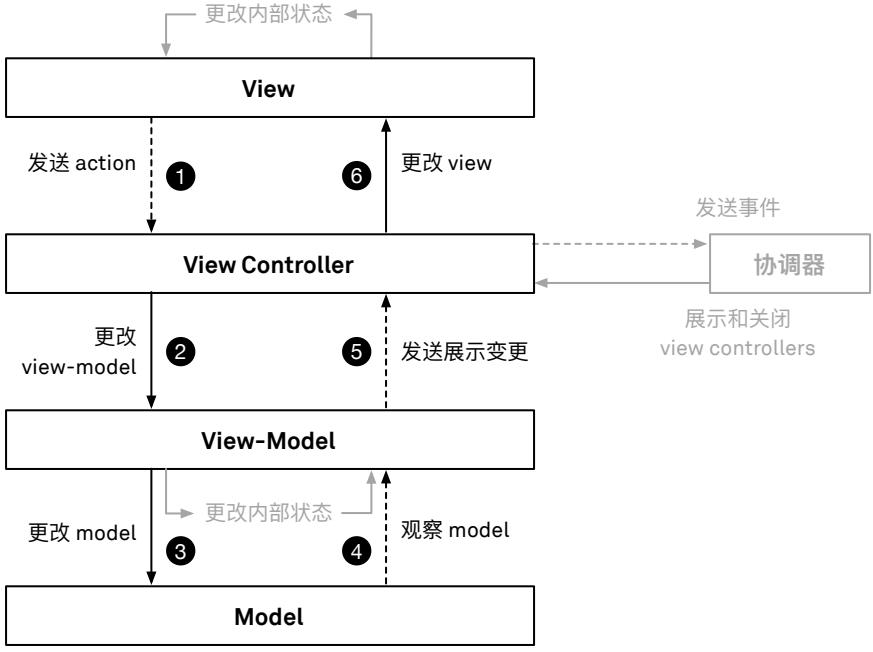
```
// FolderViewController
override func encodeRestorableState(with coder: NSCoder) {
  super.encodeRestorableState(with: coder)
  coder.encode(viewModel.folder.value.uuidPath, forKey: .uuidPathKey)
}

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，并在存储上调用 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。

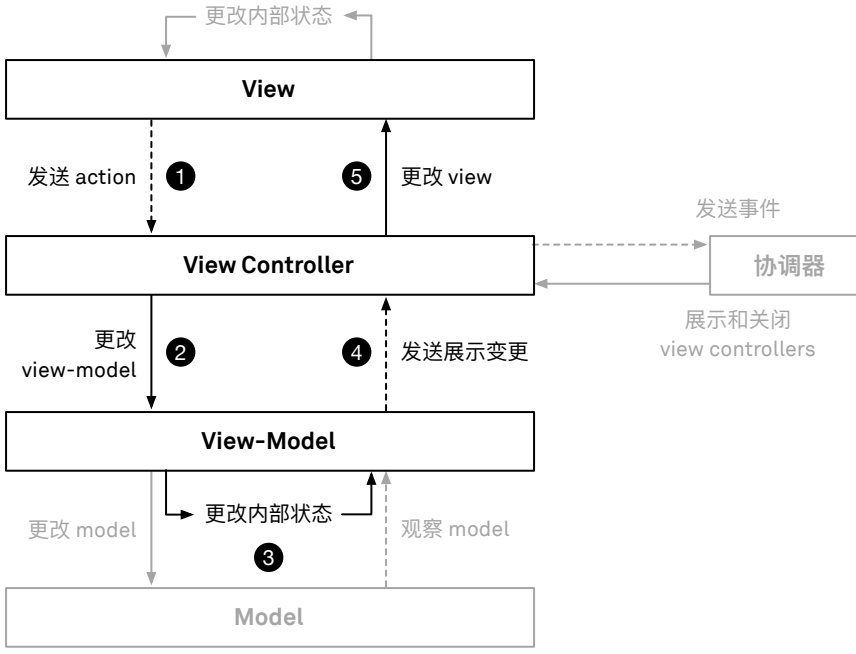
下面，我们将通过两个不同的例子来看看如何更改 view state：

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 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

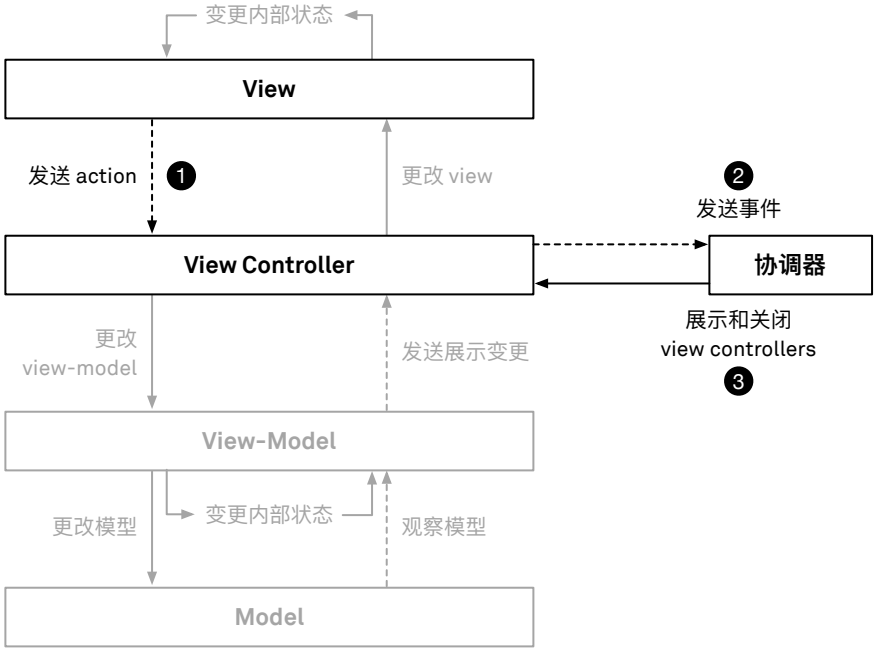
在 PlayViewController 的 viewDidLoad 中，我们对 view-model 的 playButtonTitle 可观察量和播放按钮标题进行了响应式绑定：

```
// 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) 的全面的支持，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 通过定义明确从 view 层解耦出来的 view-model 这一清晰的接口，改变了集成的特性。相比与测试整个 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。

然后，MVVM 的接口测试和 MVC 的集成测试不论在范围，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，但是它们难以书写和维护。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，而不是将测试缩小为每个测试针对一个函数。

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 通常包含了要求 controller 必须非常简单的规则 (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存储的构建，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存储，包括了一些文件夹的引用；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，构建接口并将传递输入，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。这类测试和可观测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，而不是在存储上以外，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

相比于 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 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，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 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及交互逻辑移动到围绕着数据流构建的隔离的类中，可以解决 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

响应式编程是一种有它自己的权衡的技术，它对有的人很具有吸引力，但是也有人对它避之唯恐不及。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，包括暴露出给导航栏标题和 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 可观察量是通过对 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。接下来，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，我们通过观察存储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，它会都调用 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。如果有影响，我们必须要处理两种情况：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，响应式编程在让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直接将值设置到 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，类似于使用 Rx 进行绑定那样，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 的播放 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 将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，我们必须要提出自己的机制。

学习到的经验教训

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被应用到 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) - 使用例指的是那些对主 model 进行切片准备，并且简化执行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，它们会更加清晰。

网络

5

> **注意：**本书和相关视频还处于编辑阶段，并没有正式发布，最终的内容可能会有所改变。对于本书中现存的错误和不足，我们深表歉意！

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。这个缺陷将由第二种网络的实现方式来解决。

和 controller 拥有网络的的版本一样，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，就和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 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，在当为 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：它最近一次网络请求所得到的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
  }
}
```

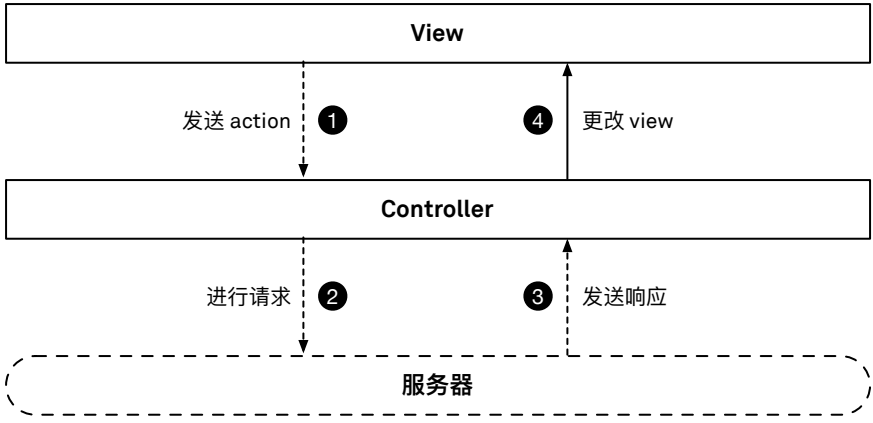
首先，我们将文件夹的状态设置为 .loading，这可以阻止我们在前一个请求还在途中的时候再次请求同样的数据。然后我们执行请求，更新文件夹的内容，将状态设置为 .loaded，并且重载 table view。在此期间，我们还更新了 table view controller 的刷新控件的状态。

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研究了：

```
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...
  }
}
```

实际的删除是一个网络请求：

```
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化的：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，有时会立即发生，有时要花上好几秒。为了简介明了，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，所以我们也许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，或者是实现一个处理和服务器交互的操作的网络服务类。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 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。我们本意就想让 model 拥有网络的版本要比 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，并且在将变更提交给服务器时，处理可能发生的冲突。

我们在下面会仔细查看实现的时候，会把注意力集中在相比于 controller 拥有网络的方式，网络是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 (使用与控制器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，就好像它们仅仅只是本地数据一样，网络则发生在 model 层的背后。

对于我们 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对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才不至于丢失。

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.backgroundView?.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。

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)。