# 关于 MVC 的一个常见的误用

如何避免把 Model View Controller 写成 Massive View Controller 已经是老生常谈的问题了。不管是拆分 View Controller 的功能 (使用多个 Child View Controller)，还是换用“广义”的 MVC 框架 (比如 MVVM 或者 VIPER)，又或者更激进一点，转换思路使用 Reactive 模式或 Reducer 模式，其实所想要解决的问题本质在于，我们要如何才能更清晰地管理“用户操作，模型变更，UI 反馈”这一数据流动的方式。

非传统的 MVC 可以帮助我们遵循一些更不容易犯错的编程范式 (这一点和 Java 很像，使用冗杂的 pattern 来规范开发，让新人也能写出“成熟”的代码)，但是如果不从根本上理解数据流动在 MVC 中的角色，那不过就是末学肤受，迟早会出现问题。

### 例子

举一个非常简单的 View Controller 的例子。假设我们有一个 Table View Controller 来记录 To Do 列表，我们可以通过点击导航栏的加号按钮来追加一个条目，用 Swipe cell 的方式删除条目。我们希望最多同时只能存在 10 条待办项目。这个 View Controller 的代码非常简单，可能也是很多开发者每天会写的代码。包括设置 Playground 和添加按钮等等，一共也就 60 行。我将它放到了[这个 gist 中](https://gist.github.com/onevcat/4042d4d0f156b986e4755a7d4370bb9c)，你可以全部复制下来扔到 Playground 里查看效果。

这里简单对比较关键的代码进行一些解释。首先是模型定义：

// 定义简单的 ToDo Modelstruct ToDoItem {

let id: UUID

let title: String

init(title: String) {

self.id = UUID()

self.title = title

}}

然后我们使用 UITableViewController 的子类进行待办事项的展示和添加：

class ToDoListViewController: UITableViewController {

// 保存当前待办事项

var items: [ToDoItem] = []

// 点击添加按钮

@objc func addButtonPressed(\_ sender: Any) {

let newCount = items.count + 1

let title = "ToDo Item \(newCount)"

// 更新 `items`

items.append(.init(title: title))

// 为 table view 添加新行

let indexPath = IndexPath(row: newCount - 1, section: 0)

tableView.insertRows(at: [indexPath], with: .automatic)

// 确定是否达到列表上限，如果达到，禁用 addButton

if newCount >= 10 {

addButton?.isEnabled = false

}

}}

接下来，处理 table view 的展示，这部分内容乏善可陈：

extension ToDoListViewController {

override func tableView(\_ tableView: UITableView, numberOfRowsInSection section: Int) -> Int {

return items.count

}

override func tableView(\_ tableView: UITableView, cellForRowAt indexPath: IndexPath) -> UITableViewCell {

let cell = tableView.dequeueReusableCell(withIdentifier: cellIdentifier, for: indexPath)

cell.textLabel?.text = items[indexPath.row].title

return cell

}}

最后，实现滑动 cell 删除的功能：

extension ToDoListViewController {

override func tableView(\_ tableView: UITableView, trailingSwipeActionsConfigurationForRowAt indexPath: IndexPath) -> UISwipeActionsConfiguration? {

let deleteAction = UIContextualAction(style: .destructive, title: "Delete") { \_, \_, done in

// 用户确认删除，从 `items` 中移除该事项

self.items.remove(at: indexPath.row)

// 从 table view 中移除对应行

self.tableView.deleteRows(at: [indexPath], with: .automatic)

// 维护 addButton 状态

if self.items.count < 10 {

self.addButton?.isEnabled = true

}

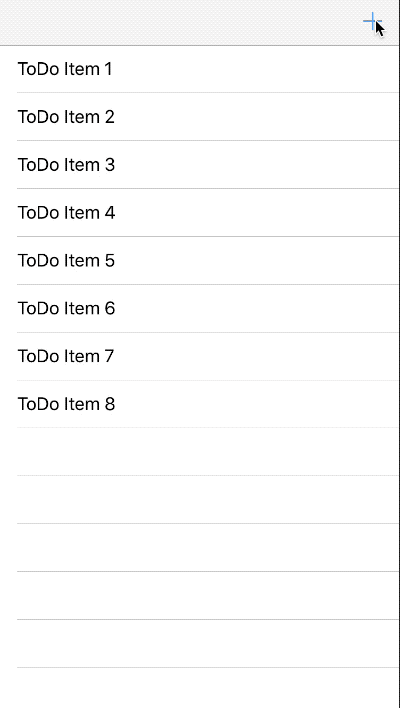
done(true)

}

return UISwipeActionsConfiguration(actions: [deleteAction])

}}

效果如下：



看起来一切正常工作！不过，你能看出有什么问题吗？抑或说你觉得这段代码已经完美无瑕了？

### 风险

简单来说，这也已经是对 MVC 的误用了。上面的代码存在着这些潜在问题：

1.Model 层“寄生”在ViewController 中

在这段代码中，View Controller 里的 items 充当了 model。

这导致了几个问题：我们难以从外界维护或者同步 items 的状态，添加和删除操作被“绑定”在了这个 View Controller 里，如果你还想通过其他 View Controller 维护待办列表的话，就不得不考虑数据同步的问题 (我们会在稍后看到几个具体的这方面的例子)；另外，这样的设置导致 items 难以测试。你几乎无法为添加/删除/修改待办列表进行 Model 层的测试。

2.违反数据流动规则和单一职责规则

如果我们仔细思考，会发现，用户点击添加按钮，或者侧滑删除 cell 时，在 View Controller 中其实发生了这些事情：

* 1. 维护 Model (也就是 items)
  2. 增删 table view 的 cell
  3. 维护 addButton 的可用状态

也就是说，UI 操作不仅导致了 Model 的变更，还同时导致了 UI 的变化。理想化的数据流动应该是单向的：UI 操作 -> 经由 View Controller 进行模型更新 -> 新的模型经由 View Controller 更新 UI -> 等待新的 UI 操作，而在例子中，我们变成了“经由 View Controller 进行模型更新以及 UI 操作”。虽然看起来这是很不起眼的变更，但是会在项目复杂后带来麻烦。

也许你现在并不觉得有什么问题，让我们来假设一些情景，你可以思考一下如何实现吧。

#### 场景一

首先来看看待办条目的编辑，我们可能需要一个详情页面，用来编辑某个待办的细节，比如为 ToDoItem 添加上 date，location 和 detail 这类的属性。另外，PM 和用户也许希望在详情页面中也能直接删除这个正在编辑的待办。

以现在的实现来看，一个很朴素的想法是新建 ToDoEditViewController，然后设置 delegate 来告诉 ToDoListViewController 某个 ToDoItem 发生了变化，然后在 ToDoListViewController 进行对 items 进行操作：

protocol ToDoEditViewControllerDelegate: class {

func editViewController(\_ viewController: ToDoEditViewController, editToDoItem original: ToDoItem, to new: ToDoItem)

func editViewController(\_ viewController: ToDoEditViewController, remove item: ToDoItem)}

// 在 ToDoListViewController 中extension ToDoListViewController: ToDoEditViewControllerDelegate {

func editViewController(\_ viewController: ToDoEditViewController, remove item: ToDoItem) {

guard let index = (items.index { $0.id == item.id }) else { return }

items.remove(at: index)

let indexPath = IndexPath(row: index, section: 0)

tableView.deleteRows(at: [indexPath], with: .automatic)

if self.items.count < 10 {

self.addButton?.isEnabled = true

}

}

func editViewController(\_ viewController: ToDoEditViewController, editToDoItem original: ToDoItem, to new: ToDoItem) {

//...

}}

有一部分和之前重复的代码，虽然可以通过将它们提取成像是 removeItem(at index: Int) 这样的方法，但是并不能改变非单一功能的问题。ToDoEditViewController 本身也无法和 items 通讯，因此它扮演的角色几乎就是一个“专用”的 View，一旦脱离了 ToDoListViewController，则“难堪重任”。

#### 场景二

另外，纯单机的 app 已经跟不上时代了，不管是 iCloud 同步还是自建服务器，我们总是想要一个后端来为用户跨设备保存列表，这是一个非常可能的增强。在现有架构下，把从服务器获取已有条目的逻辑放到 ToDoListViewController 也是很自然的想法：

override func viewDidLoad() {

super.viewDidLoad()

//..

NetworkService.getExistingToDoItems().then { items in

self.items = items

self.tableView.reloadData()

if self.items.count >= 10 {

self.addButton?.isEnabled = false

}

}}

这种简单的实现面临很多挑战，是我们在实际 app 中不得不考虑的：

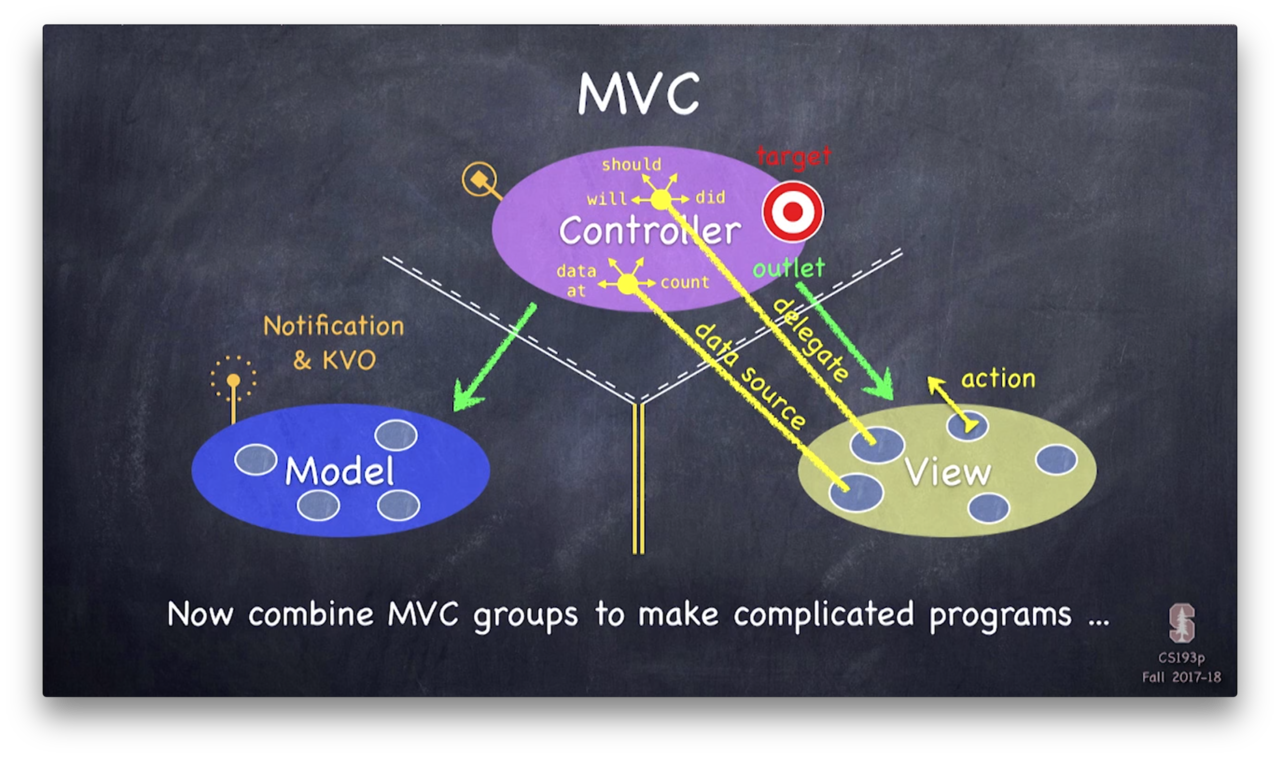
1. 是不是应该需要在 getExistingToDoItems 过程中 block 掉 UI，否则用户在请求完成前所添加的条目将被覆盖。
2. 在添加和删除条目的时候，我们都需要进行网络请求，另外我们也需要根据请求返回的状态更新添加按钮的状态。
3. Block 用户输入将让 app 变为没网无法使用，不进行 block 的话则需要考虑数据同步的问题。
4. 另外，我们需不需要在没网时依然让用户可以进行增加或删除，并缓存操作，等到有网时再将这些缓存反映给服务器。
5. 如果需要实现 4，那么还要考虑操作结果导致超出条目最大数量限制的错误处理，以及多设备间数据冲突处理的问题。

是不是突然感觉有些头大？

### 改善

这些问题的来源其实都是我们为了“省事”，选择了一个**不那么有效的 Model**，以及**存在风险的数据流动方式**。或者说，我们没有正确和严格地使用 MVC 架构。

关于 MVC，斯坦福的 CS193p Paul 老师有一张非常经典的图，相信很多 iOS 的开发者也都看过：



我们的例子中，我们等于把 Model 放到了 Controller 里，而且 Model 也无法与 Controller 进行有效的通讯 (图中的 Notification & KVO 部分)。这导致 Controller 承载了太多的功能，这往往是光荣地迈向 Massive View Controller 的第一步。

#### 单独的 Model

当务之急是将 Model 层提取出来，为了说明简单，暂时先只考虑纯本地的情况：

extension ToDoItem: Equatable {

public static func == (lhs: ToDoItem, rhs: ToDoItem) -> Bool {

return lhs.id == rhs.id

}}

class ToDoStore {

static let shared = ToDoStore()

private(set) var items: [ToDoItem] = []

private init() {}

func append(item: ToDoItem) {

items.append(item)

}

func append(newItems: [ToDoItem]) {

items.append(contentsOf: newItems)

}

func remove(item: ToDoItem) {

guard let index = items.index(of: item) else { return }

remove(at: index)

}

func remove(at index: Int) {

items.remove(at: index)

}

func edit(original: ToDoItem, new: ToDoItem) {

guard let index = items.index(of: original) else { return }

items[index] = new

}

var count: Int {

return items.count

}

func item(at index: Int) -> ToDoItem {

return items[index]

}}

当然，为了一步到位，也可以直接把上面的 NetworkService 加上，写成异步 API，例如：

func getAll() -> Promise<[ToDoItem]> {

return NetworkService.getExistingToDoItems()

.then { items in

self.items = items

return Promise.value(items)

}}

func append(item: ToDoItem) -> Promise<Void> {

return NetworkService.appendToDoItem(item: item)

.then {

self.items.append(item)

return Promise.value(())

}}

为了好看，这里用了一些 [PromiseKit 的东西](https://github.com/mxcl/PromiseKit)，如果你不熟悉 Promise，也不用担心，可以将它们简单地看作 closure 的形式就好，这并不会影响继续阅读本文：

func getAll(completion: @escaping (([ToDoItem]?, Error?) -> Void)?) {

NetworkService.getExistingToDoItems { response, error in

if let error = error {

completion?(nil, error)

} else {

self.items = response.items

completion?(response.items, nil)

}

}}

这样，我们就可以将 items 从 ToDoListViewController 拿出来。对单独提取的 Model 进行测试变得非常容易，纯 Model 的操作与 Controller 无关，ToDoEditViewController 也不再需要将行为 delegate 回 ToDoListViewController，编辑条目的 View Controller 可以通过成为了真正意义上的 View Controller，而不止是 ToDoListViewController 的“隶属 View”。

单独的 ToDoStore 作为模型带来的另一个好处是，因为它与具体的 View Controller 分离了，在进行持久化时，我们可以有更多的选择。不论是从网络获取，还是保存在本地的数据库，这些操作都不必 (也不应写在 View Controller 中)。如果有多种数据来源，我们可以轻松地创建类似 ToDoStoreCoordinator 或者 ToDoStoreDataProvider 这样的类型。既可以满足单一职责，也易于覆盖完整的测试。

#### 单向数据流动

接下来，将数据流动按照 MVC 的标准进行梳理就是自然而然的事情了。我们的目标是避免 UI 行为直接影响 UI，而是由 Model 的状态通过 Controller 来确定 UI 状态。这需要我们的 Model 能够以某种“非直接”的方式向 Controller 进行汇报。按照上面的 MVC 图，我们使用 Notification 来搞定。

对 ToDoStore 进行一些改造：

class ToDoStore {

enum ChangeBehavior {

case add([Int])

case remove([Int])

case reload

}

static func diff(original: [ToDoItem], now: [ToDoItem]) -> ChangeBehavior {

let originalSet = Set(original)

let nowSet = Set(now)

if originalSet.isSubset(of: nowSet) { // Appended

let added = nowSet.subtracting(originalSet)

let indexes = added.compactMap { now.index(of: $0) }

return .add(indexes)

} else if (nowSet.isSubset(of: originalSet)) { // Removed

let removed = originalSet.subtracting(nowSet)

let indexes = removed.compactMap { original.index(of: $0) }

return .remove(indexes)

} else { // Both appended and removed

return .reload

}

}

private var items: [ToDoItem] = [] {

didSet {

let behavior = ToDoStore.diff(original: oldValue, now: items)

NotificationCenter.default.post(

name: .toDoStoreDidChangedNotification,

object: self,

typedUserInfo: [.toDoStoreDidChangedChangeBehaviorKey: behavior]

)

}

}

// ...}

这里添加了 ChangeBehavior 作为“提示”，来具体告诉外界 Model 中发生了什么。diff 方法通过比较原始 items 和当前 items 来确定发生了哪种 ChangeBehavior。最后，使用 items 的 didSet 来发送 Notification。

由于 Swift 的数组是值类型，对于 items 的元素增加，删除，修改或者整体变量替换，都会触发 didSet 的调用。Swift 的值语义编程带来了很大的便利。

在 ToDoListViewController，现在只需要订阅这个通知，然后根据消息内容进行 UI 反馈即可：

class ToDoListViewController: UITableViewController {

override func viewDidLoad() {

super.viewDidLoad()

//...

NotificationCenter.default.addObserver(

self,

selector: #selector(todoItemsDidChange),

name: .toDoStoreDidChangedNotification,

object: nil)

}

private func syncTableView(for behavior: ToDoStore.ChangeBehavior) {

switch behavior {

case .add(let indexes):

let indexPathes = indexes.map { IndexPath(row: $0, section: 0) }

tableView.insertRows(at: indexPathes, with: .automatic)

case .remove(let indexes):

let indexPathes = indexes.map { IndexPath(row: $0, section: 0) }

tableView.deleteRows(at: indexPathes, with: .automatic)

case .reload:

tableView.reloadData()

}

}

private func updateAddButtonState() {

addButton?.isEnabled = ToDoStore.shared.count < 10

}

@objc func todoItemsDidChange(\_ notification: Notification) {

let behavior = notification.getUserInfo(for: .toDoStoreDidChangedChangeBehaviorKey)

syncTableView(for: behavior)

updateAddButtonState()

}}

Notification 本身有很长的历史，是一套基于字符串的松散 API。这里通过扩展和泛型的方式，由 .toDoStoreDidChangedNotification，.toDoStoreDidChangedChangeBehaviorKey 和 post(name:object:typedUserInfo) 以及 getUserInfo(for:) 构成了一套更 Swifty 的类型安全的 NotificationCenter 和 userInfo 的使用方式。如果你感兴趣的话，可以参看[最后的代码](https://gist.github.com/onevcat/9e08111cebb1967cb96a737ed40f9f14)。

最后，我们可以把之前用来维护 table view cell 和 addButton 状态的代码都删除了。用户操作 UI 唯一的作用就是触发模型的更新，然后模型更新通过通知来刷新 UI：

class ToDoListViewController: UITableViewController {

// 保存当前待办事项

// var items: [ToDoItem] = []

// 点击添加按钮

@objc func addButtonPressed(\_ sender: Any) {

// let newCount = items.count + 1

// let title = "ToDo Item \(newCount)"

// 更新 `items`

// items.append(.init(title: title))

// 为 table view 添加新行

// let indexPath = IndexPath(row: newCount - 1, section: 0)

// tableView.insertRows(at: [indexPath], with: .automatic)

// 确定是否达到列表上限，如果达到，禁用 addButton

// if newCount >= 10 {

// addButton?.isEnabled = false

// }

let store = ToDoStore.shared

let newCount = store.count + 1

let title = "ToDo Item \(newCount)"

store.append(item: .init(title: title))

}}

extension ToDoListViewController {

override func tableView(\_ tableView: UITableView, trailingSwipeActionsConfigurationForRowAt indexPath: IndexPath) -> UISwipeActionsConfiguration? {

let deleteAction = UIContextualAction(style: .destructive, title: "Delete") { \_, \_, done in

// 用户确认删除，从 `items` 中移除该事项

// self.items.remove(at: indexPath.row)

// 从 table view 中移除对应行

// self.tableView.deleteRows(at: [indexPath], with: .automatic)

// 维护 addButton 状态

// if self.items.count < 10 {

// self.addButton?.isEnabled = true

// }

ToDoStore.shared.remove(at: indexPath.row)

done(true)

}

return UISwipeActionsConfiguration(actions: [deleteAction])

}}

现在，不妨再考虑一下上一节中场景一 (编辑条目) 和场景二 (网络同步) 的需求，是不是觉得结构会清晰很多呢？

1. 我们现在有了一个单独的可以测试的 Model 层，通过简单的 Mock，ToDoListViewController 也可以被方便地测试。
2. UI 操作 -> 经由 Controller 进行模型变更 -> 经由 Controller 将当前模型“映射”为 UI 状态，这个数据流动方向是严格可预测的 (并且应当时刻牢记需要保持这个循环)。这大大减少了 Controller 层的负担。
3. 由于模型层不再被单一 View Controller 持有，其他的 Controller (不单指像是编辑用的 Edit View Controller 这样的视图控制器，也包括比如负责下载的 Controller 等等这类数据控制器) 也可以操作模型层。在此同时，所有的模型结果会被自动且正确地反应到 View 上，这为多 Controller 协同工作和更复杂的场景提供了坚实的基础。

这个例子的修改后的最终版本[可以在这里找到](https://gist.github.com/onevcat/9e08111cebb1967cb96a737ed40f9f14)。

#### 其他选项

MVC 本身的概念相当简单，同时它也给了开发者很大的自由度。Massive View Controller 往往就是利用了这个自由度，“随意”地将逻辑放在 controller 层所造成的后果。

有一些其他架构选择，最常用的比如 MVVM 和响应式编程 (比如 RxSwift)。MVVM 可以说几乎就是一个 MVC，不过通过 View Model 层来将数据和视图进行绑定。如果你写过 Reactive 架构的话，可能会发现我们在本文中 MVC 的 Controller 层的通知接收和 Rx 的事件流非常相似。不同之处在于，响应式编程“借用”了 MVVM 的思路，提供了一套 API 将事件流与 UI 状态进行绑定 (RxCocoa)。

这些“超越” MVC 的架构方式无一例外地加入了额外的规则和限制，提供了相对 MVC 来说更小的自由度。这可以在一定程度上规范开发者的行为，提供更加统一的代码 (当然代价是额外的学习成本)。完全理解和严格遵守 MVC 的思想，我们其实也可以将 MVC 用得“小而美”。第一步，就从避免文中这类常见“错误”开始吧~