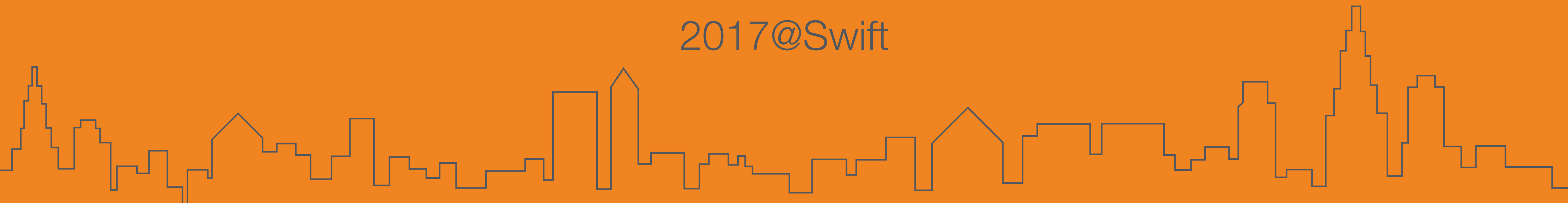




Lightweight reactive-api for MVVM

王文權

2017@Swift



自我介绍



@aaaron7



只做一件事

Reactive 思想能够解决很多问题

VS

Reactive 技术栈陡峭的学习曲线

当一个FP小白打算开始使用 Rx

Functional Reactive Programming

Observable

ReactiveSwift

map/merge/combine

React

RxSwift

RxCocoa

SignalProducer

Cold Signal

Promise

ReactiveCocoa

Signal

Stream

Redux

Functional programming

Reactive Programming

学不会怎么办

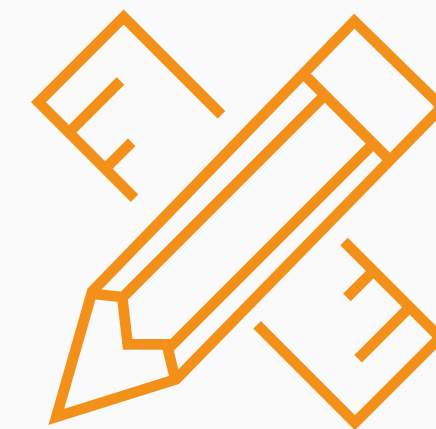


闪电之路

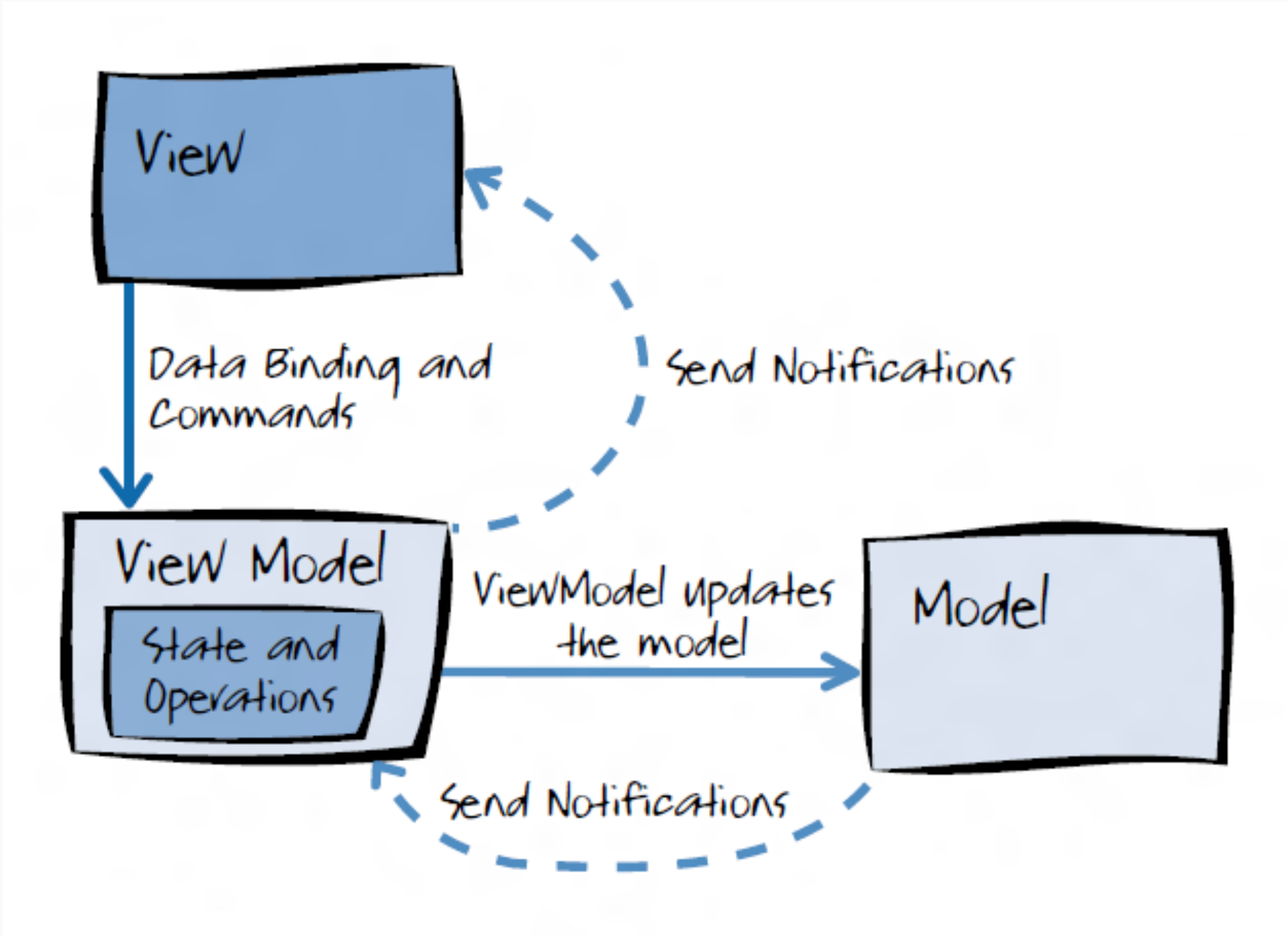


重新认识 MVVM

1



MVVM



Model
和 MVC 中的 Model 一样



View
显示用户界面，接收用户操作，并传递相应的 Command 到 ViewModel.
iOS中，这里实际上是 ViewController



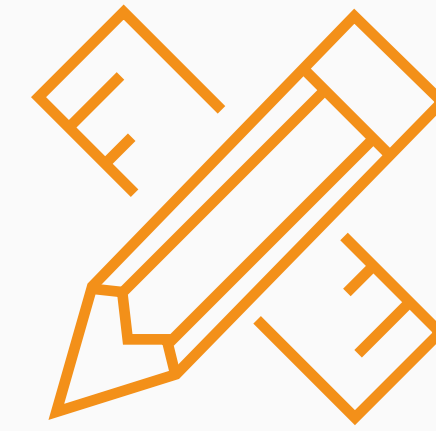
ViewModel
处理具体的业务逻辑，持有 Model，但不持有 View。只处理自己内部的 State。
State 的更改会自动同步到 View 中（数据绑定）

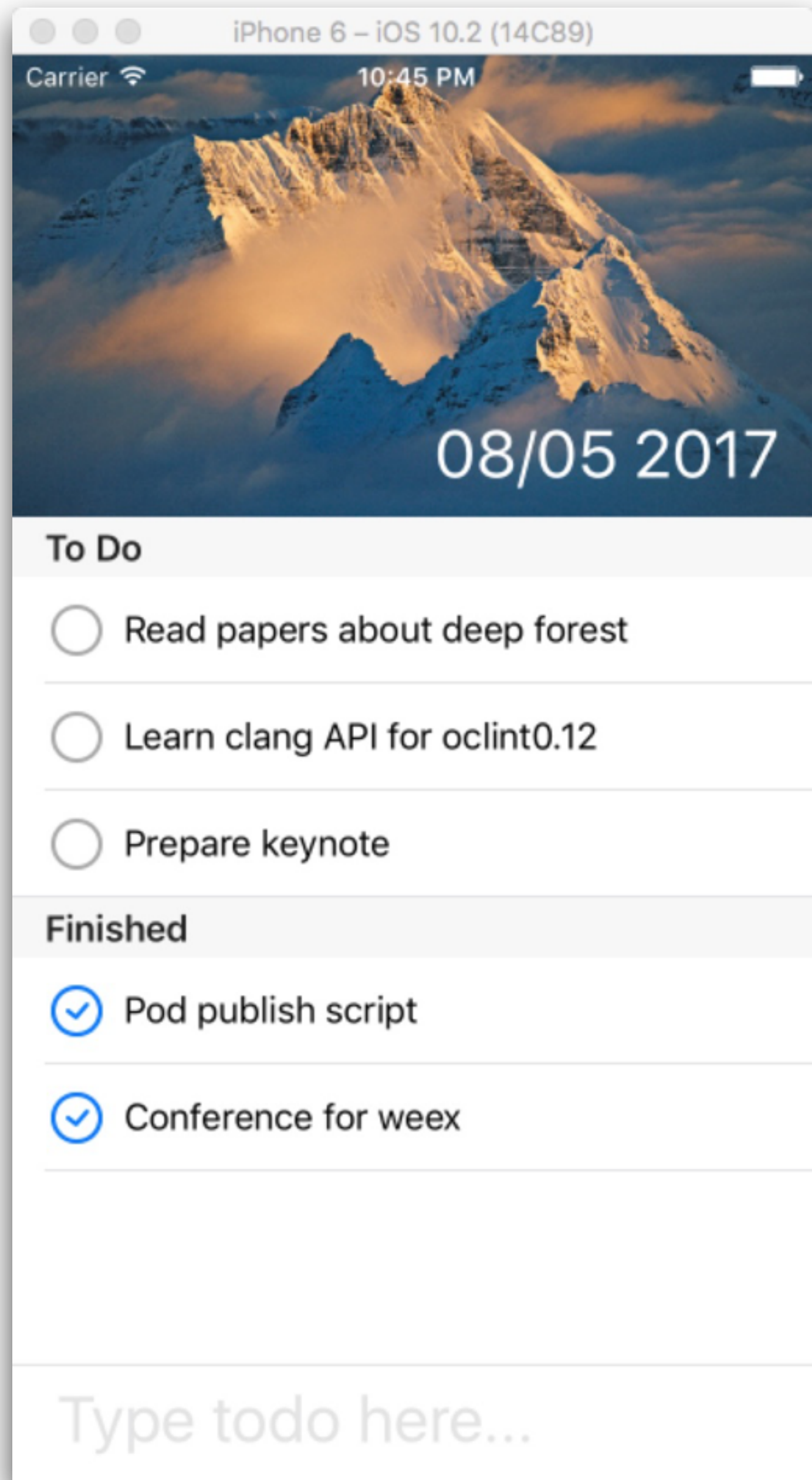
一些澄清

-  MVVM 和 FRP 没有任何关系
-  RxSwift 和 ReactiveCocoa 不能算严格意义上的 FRP, 准确的说是 FP&RP
-  FRP, 指的是把应用响应式思想到函数式编程中。但 Cocoa.....不是函数式
-  Reactive 思想, 非常适合用来解决 MVVM 中, ViewModel的 State 和 View 绑定的问题

一个简单的 App

2





- 1 显示今天的日期
- 2 可以添加 Todo
- 3 点击 Todo 前的 CheckBox, 可以标记完成
- 4 Todo 完成后, 自动去到 Finished 部分
- 5 所有操作, 都需要同步到云端(请求需要加载动画)

MVVM Design



问题来了

- “ViewModel 并不知道 ViewController 的存在”
- “ViewModel 只负责维护自己的 State, 不关心外界”
- “那 ViewModel 内部的更改如何同步到 ViewController”
- “更准确的说: ViewController 如何嗅探到 ViewModel 内部的更改?”



没有一个中间层解决不了的工程问题，如果有，就加两层

尼古拉斯·赵四

Aha, 这个简单

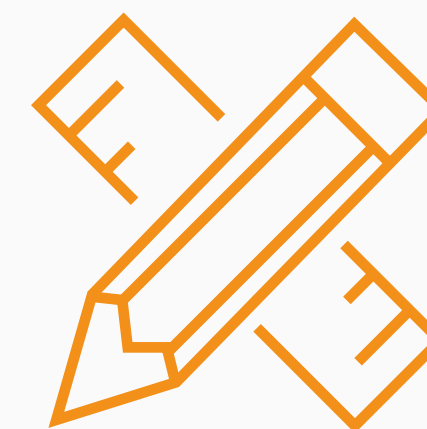
```
protocol ViewModelDelegate
{
    func didUpdateDateText(_ dateText:String)
    func didUpdateTodos(_ todos:[TodoModelItem])
    func didUpdateFinishedTodos(_ todos:[TodoModelItem])
    ...
}
```



永远不要忽视中间层所带来的维护成本和额外的心智负担





轻量级 Reactive 实现 - let.swift

3



Reactive Programming

is programming with asynchronous data streams.

-  核心是观察者模式
-  将一切可变的事物建模为数据流（变量、事件、属性、请求等等）
-  通过对各种各样数据流进行监听、变换来实现复杂的逻辑
-  通俗的来说，对比传统的 OOP，那 Reactive 就是 data-stream oriented programming

let.swift

-  为 MVVM 而生；
-  使用者不需要任何 functional/reactive/frp 的知识，看完这个 keynote 即可；
-  有且只有一个主体 — Signal，用来建模 data stream；

实现 let.swift

```

public class Signal<a> : NSObject
{
    1 public typealias SignalToken = Int
    fileprivate typealias Subscriber = (a) -> Void
    fileprivate var subscribers = [SignalToken:Subscriber]()
    public private(set) var value : a

    let queue = DispatchQueue(label: "com.swift.let.token")

    init(value : a){
        self.value = value
    }

    2 public func subscribeNext(hasInitialValue:Bool = false,
    subscriber : @escaping (a) -> Void) -> SignalToken{

    }

    3 public func bind(to control:NSObject, keyPath:String) ->
SignalToken{
    }

    4 public func update(_ value : a){

    }

    5 public func peek() -> a{
        return value
    }

}

```

1

定义 Subscriber 的类型 (a) -> Void, a 为泛型参数, 代表 Signal 可以容纳任何类型。

2

Signal 的核心方法, subscribeNext, 通过参数中的闭包来订阅该 Signal 的下一更新。

3

bind, 可以把Signal 绑定到某个 control 的 property 上。Signal 更新时会自动更新 Control 的 property。

4

Signal 的核心方法, 通过 update 方法来修改 Signal 的值。并通知所有订阅者。

5

返回 Signal 当前的瞬时值

继续实现 let.swift

```

public func subscribeNext(hasInitialValue: Bool = false,
subscriber : @escaping (a) -> Void) -> SignalToken{
    var token : SignalToken = 0
    queue.sync{
        token = (subscribers.keys.max() ?? 0) + 1
        subscribers[token] = subscriber
        if hasInitialValue{
            subscriber(value)
        }
    }
    return token
}

```

1

调用 subscribeNext 时，我们生成一个 token，并保存闭包。如果设置了 hasInitialValue，则会立刻用当前值调用一次。

```

public func bind(to control: NSObject, keyPath: String) ->
SignalToken{
    _ = self.subscribeNext(hasInitialValue: true,
subscriber: { (v : a) in
    control.setValue(v, forKey: keyPath)
    })
}

```

2

bind 最终也是用 subscribeNext 来实现，属性的同步写使用 NSObject 的 setValue 来实现。

```

public func update(_ value : a){
    queue.sync{
        self.value = value
        for sub in subscribers.values {
            sub(value)
        }
    }
}

```

3

当值更新时，逐个调用 subscriber

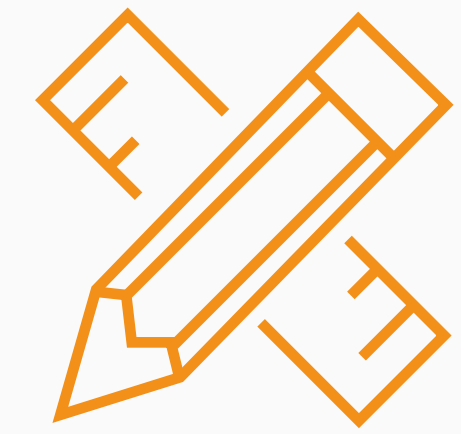
```

public func peek() -> a
{
    return value
}

```

5

Reactive MVVM



使用 Signal 来实现 ViewModel 的 State

```
class HomeViewModel
{
    var todos : Signal<[TodoModelItem]> = Signal(value: [])
    var finishedTodos : Signal<[TodoModelItem]> = Signal(value: [])
    var showIndicator : Signal<Bool> = Signal(value: false)
    var timeValue : Signal<String>
    var todoModel : TodoModel = TodoModel()

    ...
}
```

ViewModel: 两耳不闻窗外事

```

1  func addTodo(content : String, complete : @escaping (Bool)
-> Void)
2  {
    showIndicator.update(true)

    let item = TodoModelItem(timestamp:
Signal<Double>(value : 0), title: Signal<String>(value :
content), status: Signal<TodoStatus>(value : .Normal),
objectId: Signal(value : ""))

    todoModel.sendTodoAsync(todo: item, complete: { (status
: ReqStatus) in
3      self.showIndicator.update(false)
        complete(status == .Success)

        if (status == .Success)
        {
            self.updateTodo()
        }
    })
}

```

```

4  func updateTodo()
{
    todoModel.getAllModelsAsync { (x) in
        self.todos.update(x.filter({ (item) -> Bool in
            item.status.peek() == .Normal
        }))

        self.finishedTodos.update(x.filter({ (item) -> Bool in
            item.status.peek() == .Finished
        }))
    }
}

```

- 1 addTodo是 ViewModel 提供的 Command, 用于添加一个 Todo
- 2 请求开始, 设置 showIndicator 为 true, 代表显示 loading 动画
- 3 请求结束, 设置 showIndicator 为 false, 并刷新一遍 todo list
- 4 请求到 list 后, 取出未完成和已完成的, 赋值到对应的属性中

在ViewController中绑定

```
func setupViews()
{
    ...
    1 viewModel.timeValue.bind(to: self.dateLabel, keyPath:
    "text")
    2 _ = viewModel.finishedTodos.subscribeNext { (v) in
    self.todoTableView.reloadSections(IndexSet(integer:
    1), with: .none)
    }
    3 _ = viewModel.todos.subscribeNext { (v) in
    self.todoTableView.reloadSections(IndexSet(integer:
    0), with: .none)
    }
    4 _ = viewModel.showIndicator.subscribeNext { (x) in
    if x {
        self.activity.startAnimating()
    }else{
        self.activity.stopAnimating()
    }
    }
    ...
}
```

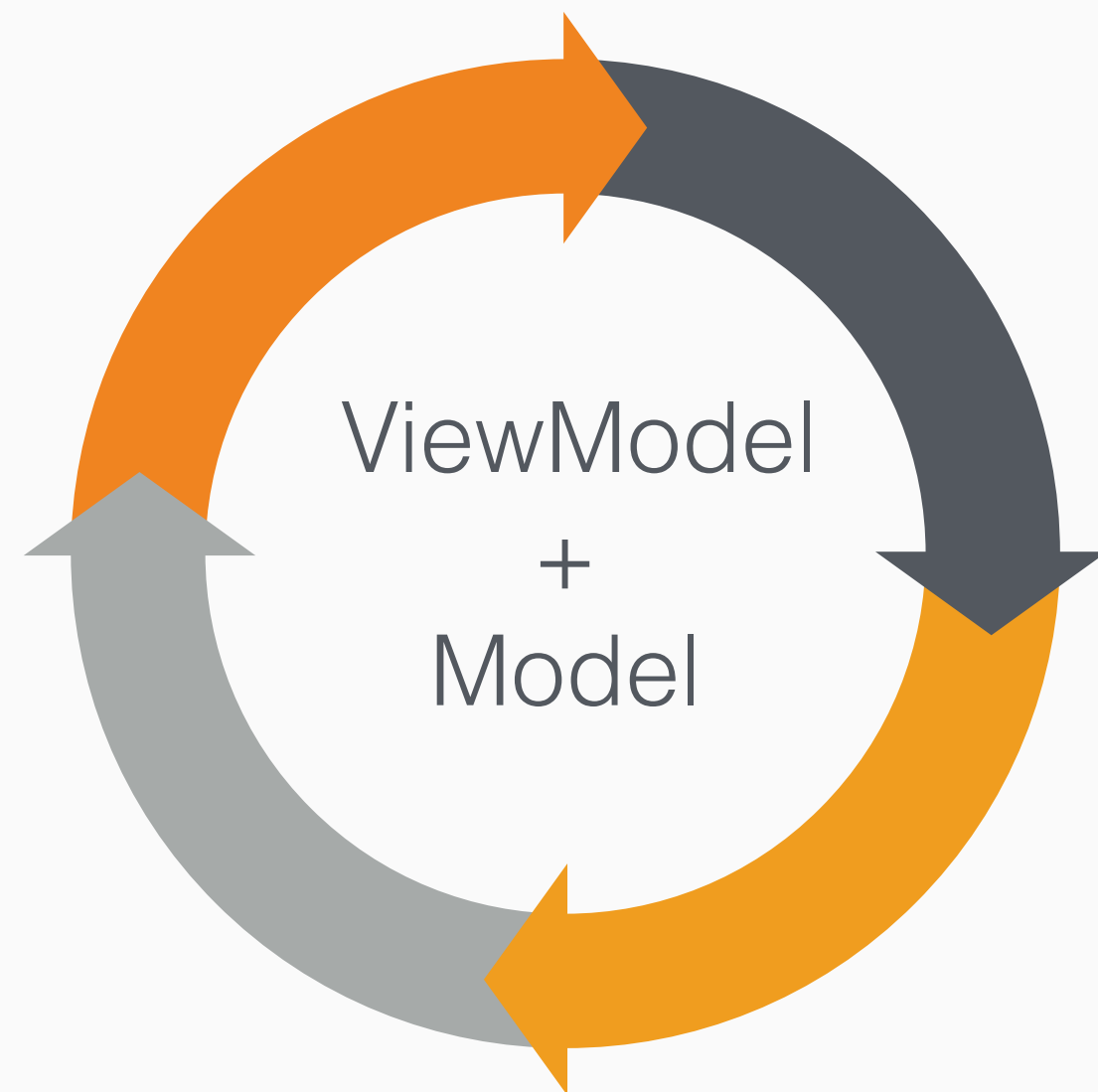
1 直接把 ViewModel 的 timeValue 绑定到 dateLabel.text

2 把 tableView 已完成 section 刷新的行为，绑定到 viewModel.finishedTodos

3 把 tableView 未完成 section 刷新的行为，绑定到 viewModel.todos

4 把 activityIndicator show/hide 的行为，绑定到 viewModel.showIndicator

Reactive MVVM的意义



逻辑的闭环

- 不关心自己被如何使用；
- 完成了绝大多数 UI 相关的业务逻辑；
- 但逻辑上依然与 UI 零耦合；
- UI 渲染层可随意更换；
- UI 逻辑的单元测试变得异常简单；

实现即协议

- 使用 ViewModel, 无需实现/遵循额外的协议; ViewModel 的实现就是协议本身;
- 外部只需绑定 State到具体的控件/控件行为以及根据用户操作, call 相应的 command, 即可完成

```
class HomeViewModel
{
    var todos : Signal<[TodoModelItem]> = Signal(value: [])
    var finishedTodos : Signal<[TodoModelItem]> = Signal(value: [])
    var showIndicator : Signal<Bool> = Signal(value: false)
    var timeValue : Signal<String>
    var todoModel : TodoModel = TodoModel()

    ...

    func updateTodo(){

    }

    func checkedTodo(index : Int, newStatus : TodoStatus){

    }

    func addTodo(content : String, complete : @escaping (Bool) -> Void){

    }
}
```

扩展 let.swift

-  考虑实现 Signal 的 Map, Combine, Filter (轻松)
-  考虑实现订阅频率的流控 (轻松)
-  实现 Promise, 处理异步问题。Signal 本身就是异步模型。(轻松)
-  使用真·FRP (Reactive + Pure functional) 来开发 iOS App。(不是很轻松)

Links & Resource

let.swift & Demo

<https://github.com/aaaron7/let.swift>

Demo Node.js backend

<https://github.com/aaaron7/let.swift.demo.backend>

Blog



THANKS

