浙江大學

硕 士 研 究 生 读 书 报 告

题目 函数响应式编程之 ReactiveCocoa 框架

作者姓名	郭志强
作者学号	21651160
指导教师	李启雷
学科专业	移动互联网和游戏开发技术
所在学院	软件学院
提交日期	二〇一六 年十二月

The ReactiveCocoa Framework of Functional Reactive Programming

A Dissertation Submitted to
Zhejiang University
in partial fulfillment of the requirements for
the degree of
Master of Engineering

Major Subject: Software Engineering
Advisor: Qilei Li

By
Zhiqiang Guo
Zhejiang University, P.R. China
2016

摘要

本文以 iOS 下的开源框架 ReactiveCocoa（RAC）为起点，研究了响应式编程（RP）以及其变体函数响应式编程（FRP）。RAC 基于函数响应式编程，它将传统响应事件的方式统一，为事件定义了标准接口，这使得更容易对事件流进行过滤、连接和组合。在传统的 iOS 开发中，都是通过 action、delegate、KVO 以及 callback 等方式来进行事件的响应，但这些方式都会让开发者过多陷入实现细节，大量的状态变量、额外的处理过程都使代码结构复杂化。RAC 的主旨是让代码更简洁易懂，逻辑可以用清晰的管道、流式语法来表示，开发人员不用过多陷入实现细节而更注重业务逻辑，使得应用更容易理解。RAC 试图解决三个问题：传统 iOS 开发过程中，状态以及状态之间依赖过多的问题；传统 MVC 架构的问题：Controller 比较复杂，可测试性差；提供统一的消息传递机制。理解 RAC 的关键是理解响应式编程，在理解 RP 的过程中，最困难的是以 RP 的方式思考，意味着要放弃命令式且带状态的(Imperative and stateful)编程习惯，并且要强迫大脑以一种不同的范式来工作。如果能熟练地运用 RAC，这会让 iOS 开发变得更快捷，因为在开发过程中更注重的是业务逻辑，业务逻辑能很简单地转化为 RAC 方式下的代码。特别是开发高互动性的 Webapps、Mobile apps 时，运用 RAC 框架的优势更加显著，这些应用包含大量与数据事件相关的 UI 事件，而 RAC 能轻松地用极少的代码实现复杂的 UI 交互。

关键词：iOS，ReactiveCocoa，响应式编程，FRP，RP

Abstract

Based on ReactiveCocoa (RAC) which is an open source framework in iOS, this paper studies Reactive Programming (RP) and its variant Functional Reactive Programming (FRP). RAC bases on Functional Reactive Programming that unifies the traditional way of reacting to events and defines a standard interface for events, making it easier to filter, connect, and combine event streams. In the traditional iOS development, through the action, delegate, KVO and callback and other ways to react to events. But these methods will force developers to fall into too many implementation details, a large number of state variables and additional processing procedure make the code to complex. RAC's main purpose is to make the code more simple and easy to understand, and logic can be expressed in clear pipelines and streaming syntax. Developers do not have to fall into the implementation details too much, but focus on business logic which makes the application easier to understand. RAC attempts to solve three problems: the state and too much dependence between the states problems in the traditional iOS development process; traditional MVC architecture problems: Controller is much complex and testability is poor problems; provide a unified messaging mechanism. The key to understanding RAC is to understand Reactive Programming. The most difficult way to understand RP is to think in terms of RP, which means giving up the imperative and stateful programming habit and forcing the brain to a different paradigm to work. If you can skillfully use RAC, which will make iOS development faster, because in the development process is more focused on business logic, business logic can be easily converted to RAC mode code. Especially when developing highly interactive Webapps and Mobile apps, the advantages of using the RAC framework are more remarkable. These applications contain a large number of UI events related to data events, and RAC can easily implement complex UI interaction with very little code.

Keywords：iOS, ReactiveCocoa, Reactive Programming, FRP, RP

目录

1 响应式编程概念

1.1 什么是 RP

RP（Reactive Programming）即响应式编程，在理解 RP 的过程中，最困难的是以 RP 的方式思考，意味着要放弃命令式且带状态的(Imperative and stateful)编程习惯，并且要强迫你的大脑以一种不同的范式来工作。

网络上有一大堆对新手难以理解的解释和定义，如 Wikipedia 上通常都是些非常笼统和理论性的解释，而 Stackoverflow 上的一些规范的回答显然也不适合新手来参考，Reactive Manifesto 看起来也只像是拿给你的 PM 或者老板看的东西，微软的 Rx 术语"Rx = Observables + LINQ + Schedulers" 也显得太过沉重，而且充满了太多微软式的东西，反而给我们带来更多疑惑。"Reactive"和"Propagation of change"这样的术语并没有传达任何有意义的概念。

1.2 RP 是使用异步数据流进行编程

RP 是一种编程范式，它与异步数据流进行交互[1]。事件总线（Event buses）或者点击事件（Click Events）等本质上就是异步事件流（Asynchronous event stream），它们可以被监听处理，可以观察它们的变化做出一些反应（do some side effects）。RP 的大致思路是：除了点击（Click）和悬停（Hover）等事件，其它任何事物都可以被创建数据流（Data stream）。变量、用户输入、属性、cache、数据结构等都可以被创建为一个 stream 。

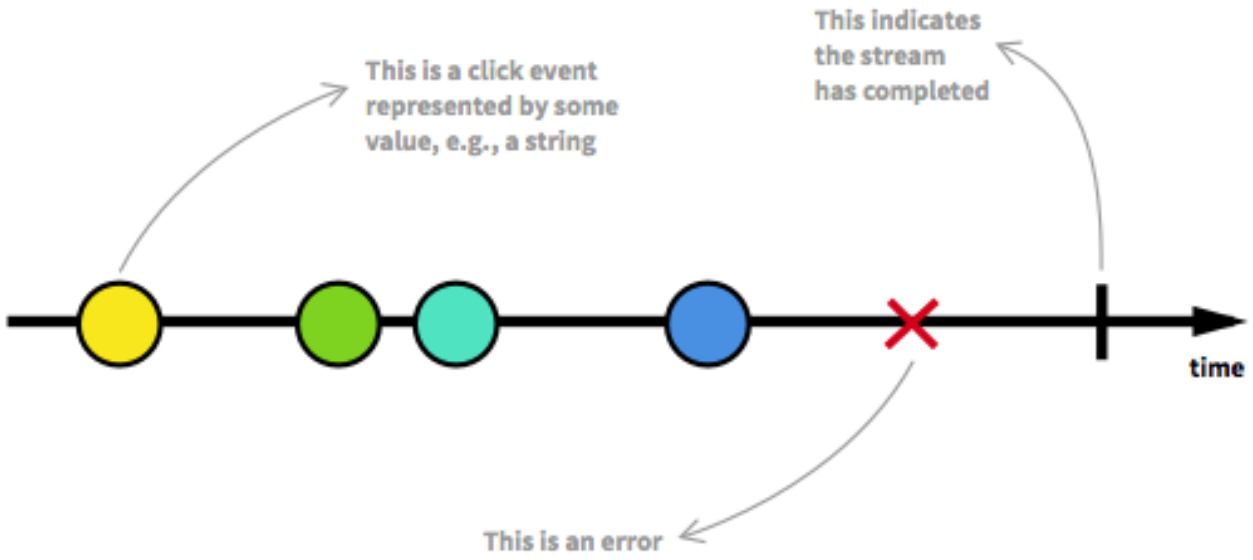

图 1-1 “Click Button” Event Stream

在创建出数据流后，还可以用函数取处理，比如结合（combine）、创建（create）、过滤（filter）等函数。这是函数式编程，它与响应式编程结合形成一个变体—FRP（函数响应式编程）。

数据流也可以作为另一个数据流的输入，每个数据流可以接受一个或多个数据流输入；还可以合并（merge）两个数据流；从一个数据流过滤（filter）出感兴趣的事件生成新的数据流；也可以将一个数据流的值映射（map）到新的数据流中。

1.3 点击事件流

数据流是 RP 的核心[2]，这里从“点击一个按钮”事件流来进行简单介绍。数据流是一个按时间排序的即将发生事件的序列（Ongoing events ordered in time），如图 1-1，它可以发出（emit）三种不同的事件：某种类型的值（value）事件、错误（error）事件和完成（completed）事件。

捕获这三个事件是以异步的方式，分别为三个事件定义事件处理函数，当事件被发出便执行相应的处理函数，一般情况只需要定义处理值事件的处理函数。如何知道哪个事件被发出了，需要去监听数据流，这也被成为订阅（Subscribing）。那么定义的事件处理函数就是观察者（Observer），被订阅的数据流就是被观察者（Observable），这其实就是观察者模式（Observer Design Pattern）。

图 1-1 也可以用 ASCII 码的方式表示：

```
--a---b-c---d---X---|->

a, b, c, d are emitted values

X is an error

| is the 'completed' signal

---> is the timeline
```

1.4 简单的计数器数据流

接下里是一个计数器数据流的实现，它能记录一个按钮点击次数。一般的 RP 库中，每个数据流都包含 map、filter、scan 等方法。比如 clickStream.map(f)，会在原 clickStream 的基础上返回新的数据流，不会去修改原数据流，即不可变性（immutability）。

这里进行了链式调用 clickStream.map(f).scan(g)，用 ASCII 表示：

```
clickStream: ---c----c--c----c------c-->

                vvvvv map(c becomes 1) vvvv
```

```
                    ---1----1--1----1------1-->
                    vvvvvvvv scan(+) vvvvvvvv
counterStream: ---1----2--3----4------5-->
```

map(f)根据 f 函数把原数据流中的值映射到新数据流，这里是把每次点击全部映射为 1 。 scan(g) 根据 g 函数把数据流中的值聚合成新的值， new_value = g(accumlated,current)，这里的 g 函数是一个累加函数。通过这两次数据流转换，最后的 counterStream 会在每次点击后将值发送给观察者，观察者实际得到的就是点击的总次数。

1.5 获取双击事件数据流

当需要获取包含双击事件的数据流，或许还需要考虑三击（Triple clicks），甚至是连击（Mutiple clicks）。如果用传统的命令式带状态的方式去实现，代码必定是混乱难以理解的，一定有一堆变量来保存状态，还有一堆计算时间间隔的代码。

在 RP 中却只需要 4 行代码就能实现这个功能，首先用图的方式表示这个实现过程。如图 1-2，灰色的方框是用来转换数据流的函数。

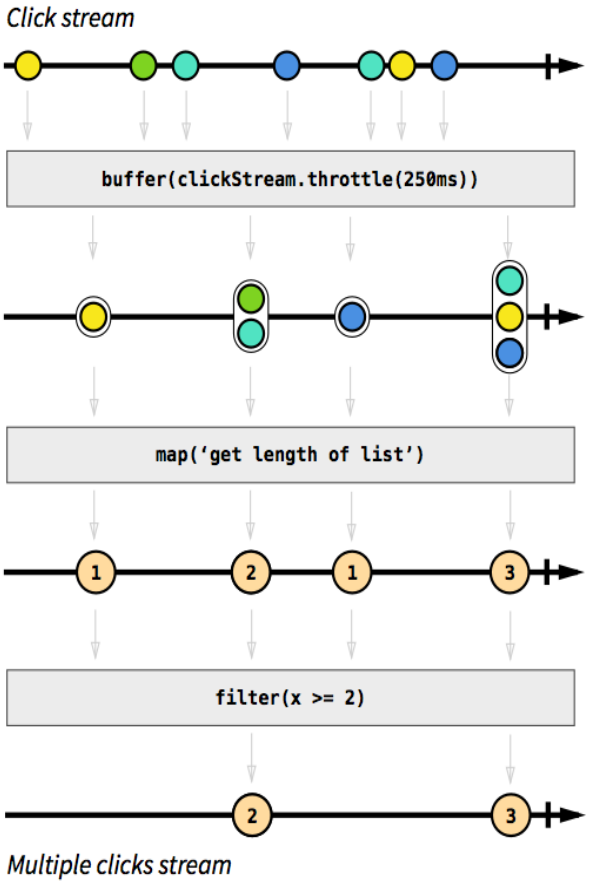

图 1-2 获取双击事件数据流过程

首先将点击事件分类放进列表（list），而这个分类的依据就是 250ms 内无事件发生（event silence，无事件时段，上一个发生事件到下一个发生事件间的时段），通过 buffer(stream.throttle(250ms))将点击事件隔开，然后将每一段的一个或多个点击事件添加进列表。这一步得到图 1-2 中第二条数据流，其中有 4 个列表，每个列表间隔大于等于 250ms。然后使用 map('get length of list')将列表映射为整数，即每个列表的长度。最后使用 filter(x >= 2)将整数过滤。

通过上述步骤，生成了需要的数据流，然后可以订阅（监听）这个数据流做出相应的反应。从这个小功能体现出 RP 的强大之处，它简洁、逻辑清晰，没有一堆状态变量，也没有复杂的计算代码。

1.6 RP 的优点

RP 提高了代码的抽象层级，在开发中只需要关注业务逻辑中那些相互依赖的事件，不用纠结于大量的实现细节，这使得 RP 的代码逻辑更加的清晰、更容易理解。

在开发高互动性的 Webapps、Mobile apps 时，RP 优势更加显著，这些应用包含大量与数据事件相关的 UI 事件。以前的网页交互很简单，只需要在用户填完一个很长的表单后，将其内容提交给后端，在填写表单过程中交互很少。而现在的网页、app 都需要大量的交互，最基本就是用户名、密码，这些东西需要实时地反应给用户。这意味着现在的应用存在大量的交互事件，如果使用命令式带状态的方式去处理，要处理太多实现细节、太多状态变量，而用 RP 的方式显然更加清晰、简单。

2 函数响应式编程框架 ReactiveCocoa

2.1 概念

利用函数式编程（Functional Programming）的思想和方法（函数、高阶函数）来支持 RP 就是函数响应式编程（FRP，Functional Reactive Programming），而 iOS 开发中常用的框架 ReactiveCocoa 便支持 FRP。

ReactiveCocoa (RAC)是一个 Objective-C 的框架,由 Github 开源的一个应用于 iOS 和 OS 开发的新框架，它的灵感来自函数响应式编程（FRP）。RAC 提供了单一且统一的方法来处理异步行为，包括 delegate 方法、block 回调、target-action 机制、notification 和 KVO

RAC 常用于处理异步事件、驱动数据源，作为一个 iOS 开发者，每一行代码几乎都是在响应某个事件，例如按钮的点击、收到网络消息、属性的变化或者用户位置的变化（通过 CoreLocation）。但是这些事件都用不同的方式来处理，比如 action、delegate、KVO、callback 等。RAC 为事件定义了一个标准接口（信号 signal，用 RACSignal 类表示），从而可以使用一些基本工具来更容易的连接、过滤和组合。

RAC 为事件提供了很多处理方法，而且利用 RAC 处理事件很方便。可以把要处理的事情和监听的事情的代码放在一起，这样非常方便管理，就不需要跳到对应的方法里。在处理时不需要考虑调用顺序，直接考虑结果，把每一次操作都写到一系列嵌套的方法中，使代码高聚合方便管理。

RAC 的主旨是让代码更简洁易懂，逻辑可以用清晰的管道、流式语法来表示，很容易理解这个应用到底干了什么。

2.2 RAC 的优点

2.2.1 试图解决的问题

RAC 试图解决：传统 iOS 开发过程中，状态以及状态之间依赖过多的问题；传统 MVC 架构的问题：Controller 比较复杂，可测试性差；提供统一的消息传递机制。

2.2.2 传统 iOS 开发过程中，状态以及状态之间依赖过多的问题

开发过程中，一个界面元素的状态很可能受多个其它界面元素或后台状态的影响。例如，在用户帐户的登录界面，通常会有 2 个输入框（分别输入帐号和密码）和一个登录按钮。如果要加入一个限制条件：当用户输入完帐号和密码，并且登录的网络请求还未发出时，确定按钮才可以点击。通常情况下，需要监听这两个输入框的状态变化以及登录的网络请求状态，然后修改另一个控件的 enabled 状态。

◆ 传统的写法见附录 1 中代码-1

RAC 通过引入信号（Signal）的概念，来代替传统 iOS 开发中对于控件状态变化检查的 target-action 模式或代理模式（delegate）。因为 RAC 的信号是可以组合（combine）的，所以可以轻松地构造出另一个新的信号出来，然后将按钮的 enabled 状态与新的信号绑定。

◆ 代码见附录 1 中代码-2

可以看到，在引入 RAC 之后，以前散落在 action-target 或 KVO 的回调函数中的判断逻辑被统一到了一起，从而使得登录按钮的 enabled 状态被更加清晰地表达了出来。
除了组合（combine）之外，RAC 的信号还支持链式（chaining）和过滤（filter），以方便将信号进行进一步处理。

2.2.3 试图解决 MVC 框架的问题

对于传统的 Model-View-Controller 的框架，Controller 很容易变得比较庞大和复杂。由于 Controller 承担了 Model 和 View 之间的桥梁作用，所以 Controller 常常与对应的 View 和 Model 的耦合度非常高，这同时也造成对其做单元测试非常不容易，对 iOS 工程的单元测试大多都只在一些工具类或与界面无关的逻辑类中进行。

RAC 的信号机制很容易将某一个 Model 变量的变化与界面关联，所以非常容易应用 Model-View-ViewModel 框架。通过引入 ViewModel 层，然后用 RAC 将 ViewModel 与 View 关联，View 层的变化可以直接响应 ViewModel 层的变化，这使得 Controller 变得更加简单，由于 View 不再与 Model 绑定，也增加了 View 的可重用性。

因为引入了 ViewModel 层，所以单元测试可以在 ViewModel 层进行，iOS 工程的可测试性也大大增强了。

2.2.4 统一消息传递机制

iOS 开发中有着各种消息传递机制，包括 KVO、Notification、delegation、block 以及 target-action 方式，各种消息传递机制使得开发者在做具体选择时感到困惑。RAC 将传统的 UI 控件事件进行了封装，使得以上各种消息传递机制都可以用 RAC 来完成。

◆ 示例代码见附录 1 中代码-3

2.3 ReactiveCocoa 类关系图

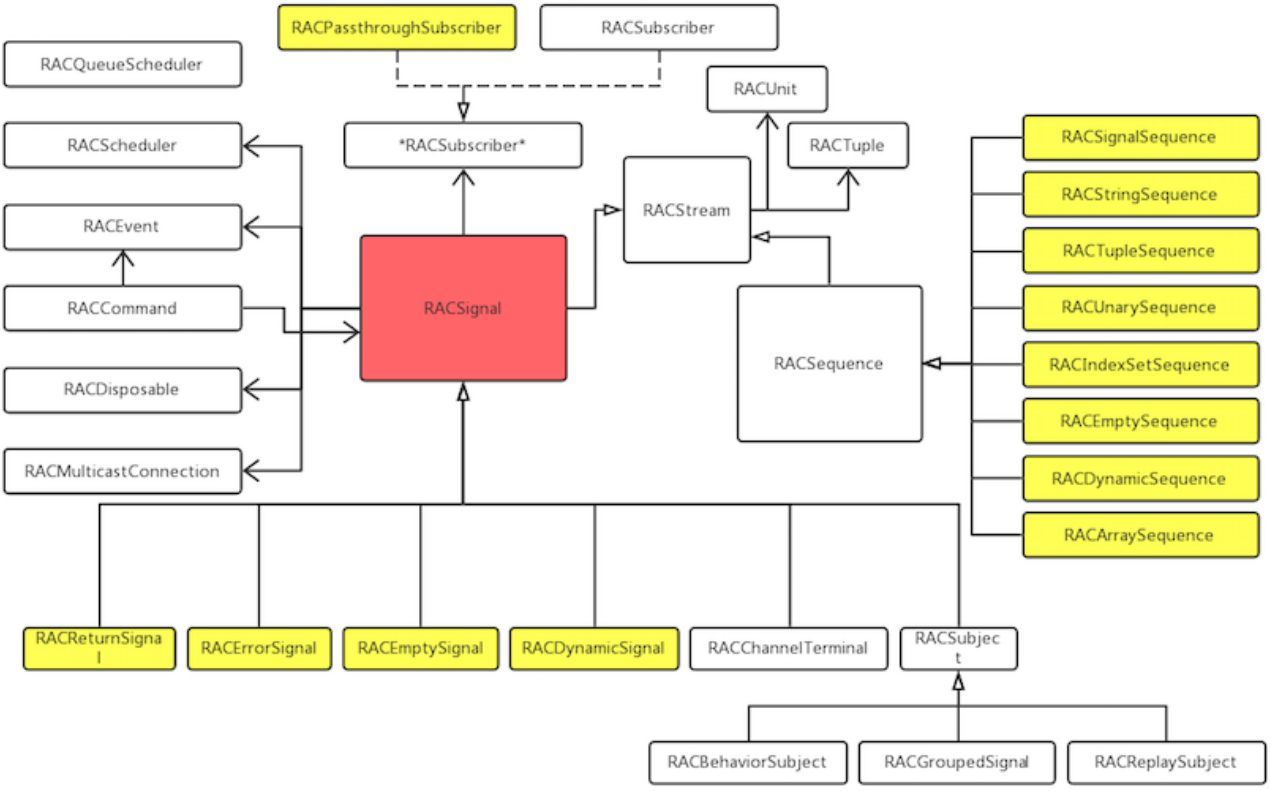

图 2-1 ReactiveCocoa 类图

图 2-1 中红色类 RACSignal 是最重要的类，而黄色类是私有类。

2.4 重点类解释

◆ Streams—RACStream

streams 代表任意的值，其值会随着事件发生变化，由 RACStream 类表示。值可能马上可用，或者在将来某一段时间可用，但必须按顺序获取，也就是说在获取到第一个值之前，是不可能获取到第二个值的。

streams 是一个构造因果关系的结构（ Monad），它可以实现在基本的初始值上进行复杂的操作运算（ filter， map， reduce 等）。

streams 不会被经常使用，大多情况下表现为 signal 和 sequences，即 Signal 和 Sequence 是由 stream 继承。

◆ Signals—RACSignal

Signals 由 RACSignal 类表示，Signals 一般表示将来被传递的数据。当信号接收到数据时，值就会通过 signal 被发送出去，它推送数据给订阅者，用户必须订阅信号才能获取到它的值。

Signals 发送 3 种不同类型的事件给它的订阅者：

1.next：next 事件是 stream 提供了一个新值。而 RACStream 类的方法也只能在这个值上进行操作运算。和 cocoa 的集合不同的是，它可以包含一个 nil 值。

2.error：error 事件表示在信号完成之前发生了错误。这个事件包含了一个 NSError 类型的错误，这个值不会在 RACStream 类里存储。

3.completed：表示信号已经成功地完成了。这个值也不会在 RACStream 类里存储。

◆ Subscription—RACSubscriber

subscriber（订阅者）表示等待或者能够等待信号发送事件的任意对象。在框架中使 RACSubscriber 协议表示，也即任意实现了 RACSubscriber 协议的对象都可以是订阅者。可以通过调用 -subscribeNext: error: completed: 方法来创建订阅， RACStream 和 RACSignal 类的大多操作也会自己创建订阅。

订阅会对 Signals 对象引用计数加 1，当信号发送错误或者完成事件后，会自动被处理，不需要用户关心内存管理。当然，用户也可以手动处理。

◆ Subjects—RACSubject

subject 由 RACSubject 类表示，它是可以手动控制的信号。subject 可以想成是 signal 的变体，就像 NSMutableArray 相对于 NSArray 一样。它们是非 RAC 的代码和 RAC 代码之间的桥梁，因此，非常有用。

有些 subject 提供了额外的功能，特别地， RACReplaySubject 用于缓存事件，将来有订阅者时，可以将缓存的数据发送给订阅者，比如，当请求一个网络时，服务器数据已经返回，但其它的数据还没有准备好，这时，先要将网络请求结果缓存，等其它数据准备好时，再订阅，而不至于网络请求数据丢失。

◆ Commands—RACCommand

command 由 RACCommand 类表示， 它创建并订阅响应 action 的信号。通常 command 是由 UI 触发，比如一个按钮被点击时。当 command 被触发时，控件会自动被禁用。

◆ Connections—RACMulticastConnection

connection 由 RACMulticastConnection 类表示。它是在任意数量的订阅者之间共享订阅。默认情况下，信号没有订阅时，是冷信号；当有订阅者被添加进来后，信号才会开始工作，变为热信号。

对于每个订阅，数据值都会被延迟重新计算。否则，如果信号有副作用或者任务开销很大时，可能就会产生问题。

connection 通过 RACSignal 的 publish 或者 multicast 方法创建，一旦连接，信号就变为热信息，订阅会保持激活状态直到所有的订阅连接被取消。

◆ Sequences—RACSequence

sequence 由 RACSequence 类表示。Sequence 是一种集合，类似于 NSArray。和数组不同，为了改进性能，在 sequence 里的值会延迟计算，只有需要输出值时，结果才会被计算。 sequences 不能包含 nil 值。

对于大多数据的集合类， RAC 添加了-rac_sequence 方法来生成 RACSequence 类进行操作。

◆ Disposables—RACDisposable

disposable 由 RACDisposable 类表示，用于取消订阅或者清理资源。Disposable 大多情况下用于取消信号的订阅 。

◆ Schedulers—RACScheduler

scheduler 由 RACScheduler 类表示，它是信号执行任务时所在的队列（ queue）或者信号执行完后将结果放到队列里执行，可以认为就是 gcd 里的 queues。

Scheduler 支持取消操作，而且它总是串行地执行任务，这有利于避免死锁。RACScheduler 有时候也类似 NSOperationQueue，但它不允许任务间相互依赖。

◆ Value types

RAC 提供了以下类用于表示各种值：

1.RACTuple： 一个比较少数量值，大小固定的，可以包含 nil 的集合。一般用于表示多个 streams 的聚合值。

2.RACUnit ： 一个单例的空值，表示 stream 不包含有意义的值。

3.RACEvent ： 代表任意的信号事件。

◆ Asynchronous Backtraces

由于基于 RAC 的代码经常和异步相关，因此，为了更方便调试， RAC 支持捕获当前的 asynchronous backtraces（异步调用栈）。

在 OS X 系统上，支持捕获所有代码，包括系统库。而 iOS，只支持 RAC 及工程内的代码。

2.5 工作原理

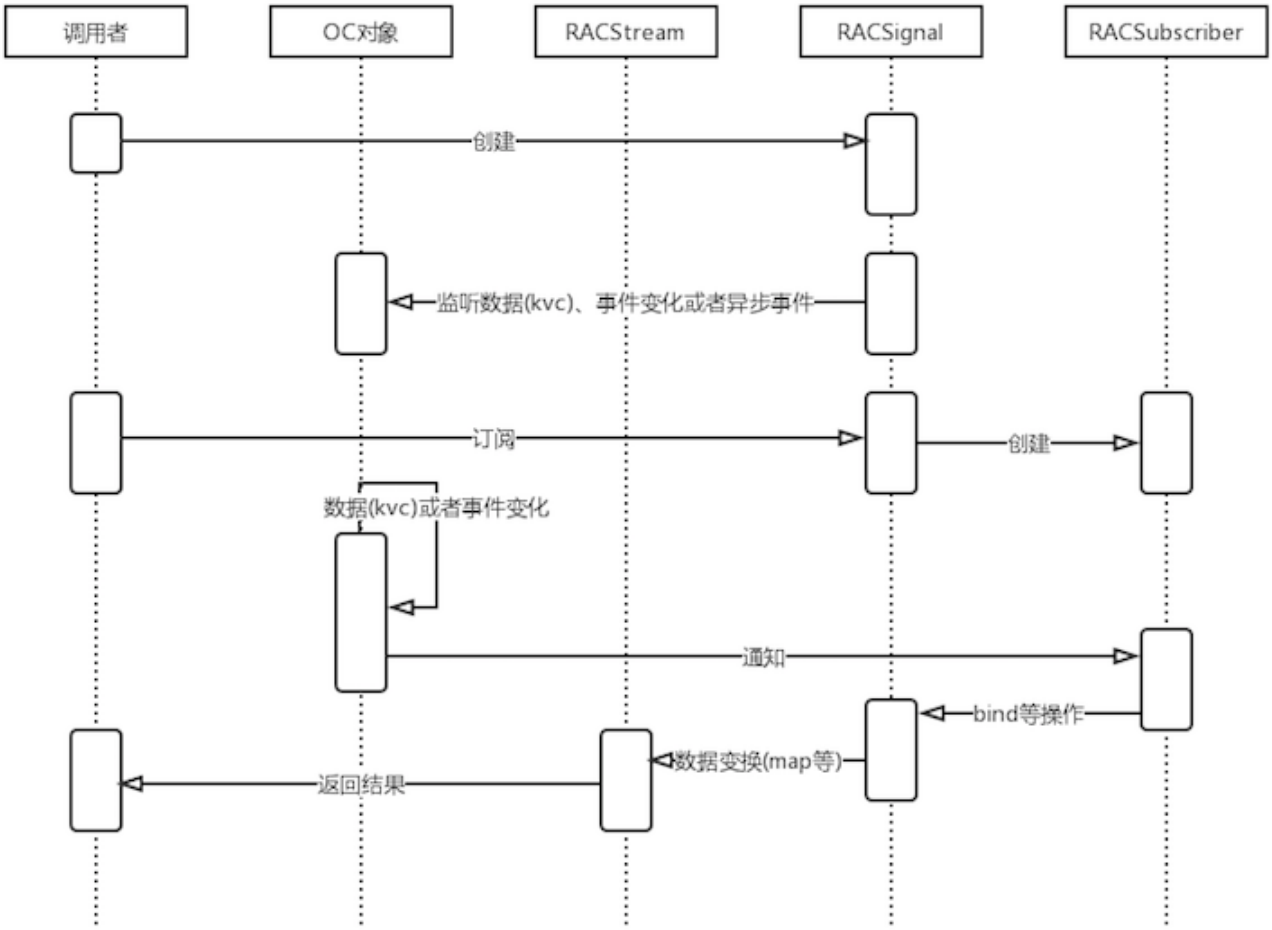

图 2-2 工作原理

2.6 重要 API 示例

2.6.1 Signal

信号类,一般表示将来有数据传递，只要有数据改变，信号内部接收到数据，就会马上发出数据。

注意：

1.信号类(RACSiganl)，只是表示当数据改变时，信号内部会发出数据，它本身不具备发送信号的能力，而是交给内部一个订阅者去发出。

2.默认一个信号都是冷信号，也就是值改变了，也不会触发，只有订阅了这个信号，这个信号才会变为热信号，值改变了才会触发。

3.如何订阅信号：调用信号 RACSignal 的 subscribeNext 就能订阅

◆ 示例代码见附录 1 中代码-4

2.6.2 RACSubscriber

表示订阅者的意思，用于发送信号，这是一个协议，不是一个类，只要遵守这个协议，并且实现方法才能成为订阅者。通过 create 创建的信号，都有一个订阅者，帮助他发送数据。

2.6.3 RACDisposable

用于取消订阅或者清理资源，当信号发送完成或者发送错误的时候，就会自动触发它。使用场景：不想监听某个信号时，可以通过它主动取消订阅信号。

2.6.4 RACSubject

信号提供者，自己可以充当信号，又能发送信号。使用场景:通常用来代替代理，有了它，就不必要定义代理了。

2.6.5 RACReplaySubject

重复提供信号类，RACSubject 的子类。RACReplaySubject 与 RACSubject 区别：RACReplaySubject 可以先发送信号，再订阅信号，RACSubject 就不可以。

使用场景一：如果一个信号每被订阅一次，就需要把之前的值重复发送一遍，使用重复提供信号类。使用场景二：可以设置 capacity 数量来限制缓存的 value 的数量,即只缓充最新的几个值。

- RACSubject 和 RACReplaySubject 简单使用见附录 1 中代码-5
- RACSubject 替换代理见附录 1 中代码-6

2.6.6 RACTuple、RACSequence

RACTuple：元组类，类似 NSArray，用来包装值。RACSequence：RAC 中的集合类，用于代替 NSArray，NSDictionary,可以使用它来快速遍历数组和字典。

2.6.7 RACCommand

RAC 中用于处理事件的类，可以把事件处理、事件中的数据传递过程包装到这个类中，他可以很方便的监控事件的执行过程。

2.6.8 RACMulticastConnection

在创建信号时，为避免多次调用创建信号中的 block 而造成副作用，可以使用这个类解决。使用注意：RACMulticastConnection 通过 RACSignal 的-publish 或者-muticast:方法创建。

2.6.9 RACScheduler

RAC 中的队列，用 GCD 封装的。

2.6.10 RACUnit

表示stream 不包含有意义的值,也就是看到这个，可以直接理解为 nil。

2.6.11 RACEvent

把数据包装成信号事件(signal event)，它主要通过RACSignal的-materialize来使用。

2.7 常用方法

- 代替代理：rac_signalForSelector;
- 代替KVO：rac_valuesAndChangesForKeyPath，用于监听对象属性的改变;
- 监听事件：rac_signalForControlEvents;
- 代替通知：rac_addObserverForName;
- 监听文本框文字改变：rac_textSignal，只要文本框发出改变就会发出这个信号;
- 合并多个 signal：rac_liftSelector:withSignalsFromArray:signals，多次请求获得数据后才进行操作。当传入的Signals(信号数组)，每一个 signal 都至少 sendNext 过一次，就会去触发第一个selector参数的方法。有几个信号，参数一方法就几个参数，每个参数对应信号发出的数据;

- 示例代码见附录1中代码-7

3 小结

掌握 ReactiveCocoa 框架的核心是理解响应式编程，抛弃命令式且带状态的编程习惯，习惯使用异步数据流进行编程。在理解 RP 的基础上，以清晰地管道、流式语法来开发。RAC 将传统的事件响应方式统一，提供了统一的接口，以 signal 的方式使得代码简洁易懂、逻辑清晰。

使用 RAC 不用花过多精力关注实现细节、状态变量，当对业务逻辑完成分析后可以轻松地转换为 RAC 框架下的代码，这让开发更加快捷。面对高交互性应用开发，RAC 比传统开发更具优势，因为 RAC 能以很少的代码清晰地完成复杂交互。

熟练运用 RAC 会在目前的 iOS 开发中发挥很大作用，使用的过程中也会加深对响应式编程的理解，响应式编程是一个更加接近自然语言的编程方式，所以花精力去学习 RAC 框架肯定大有益处。

参考文献

[1] Czaplicki E. Elm: Concurrent FRP for Functional GUIs[J]. 2012.

[2] Czaplicki E, Chong S. Asynchronous functional reactive programming for GUIs[C]// ACM SIGPLAN Symposium on Programming Language Design & Implementation. ACM, 2013:411-422.

附录

附录 1：代码示例

代码-1

```
static void *ObservationContext = &ObservationContext;
(void)viewDidLoad {
    [super viewDidLoad];

    [LoginManager.sharedManager
        addObserver:self
        forKeyPath:@"loggingIn"
        options:NSKeyValueObservingOptionInitial
        context:&ObservationContext];
    [self.usernameTextField
        addTarget:self
        action:@selector(updateLogInButton)
        forControlEvents:UIControlEventEditingChanged];
    [self.passwordTextField
        addTarget:self
        action:@selector(updateLogInButton)
        forControlEvents:UIControlEventEditingChanged];
}

- (void)updateLogInButton {
    BOOL textFieldsNonEmpty =
        self.usernameTextField.text.length > 0
        && self.passwordTextField.text.length > 0;
    BOOL readyToLogIn =
        !LoginManager.sharedManager.isLoggingIn
        && !self.loggedIn;
    self.logInButton.enabled =
        textFieldsNonEmpty && readyToLogIn;
}

- (void)observeValueForKeyPath:(NSString *)keyPath
                      ofObject:(id)object
                      change:(NSDictionary *)change
                      context:(void *)context {
    if (context == ObservationContext) {
        [self updateLogInButton];
    } else {
        [super observeValueForKeyPath:keyPath
                              ofObject:object
                              change:change
                              context:context];
    }
}
```

代码-2

```
RAC(self.logInButton, enabled) = [RACSignal
    combineLatest:@[
        self.usernameTextField.rac_textSignal,
        self.passwordTextField.rac_textSignal,
        RACObserve(LoginManager.sharedManager, loggingIn),
        RACObserve(self, loggedIn)
        ] reduce:^(NSString *username, NSString *password, NSNumber *
loggingIn, NSNumber *loggedIn) {

        return @(username.length > 0 && password.length > 0 && !
loggingIn.boolValue && !loggedIn.boolValue);
    }];
```

代码-3

```
// KVO
[RACObserve(self, username) subscribeNext:^(id x) {
    NSLog(@"成员变量 username 被修改成了: %@", x);
}];

// target-action
self.button.rac_command = [[RACCommand alloc] initWithSignalBlock:
^RACSignal *(id input) {
    NSLog(@"按钮被点击");
    return [RACSignal empty];
}];

// Notification
[NSNotificationCenter.defaultCenter addObserver:self
        selector:@selector(keyboardDidChangeFrameNotificationHandler:)
        name:UIKeyboardDidChangeFrameNotification object:nil];
```

代码-4

```
// RACSignal使用步骤:
// 1.创建信号 + (RACSignal *)createSignal:(RACDisposable *
//             (^)(id<RACSubscriber> subscriber))didSubscribe
// 2.订阅信号,才会激活信号. - (RACDisposable *)subscribeNext:(void (^)(id x))nextBlock
// 3.发送信号 - (void)sendNext:(id)value


// RACSignal底层实现:
// 1.创建信号, 首先把didSubscribe保存到信号中, 还不会触发。
// 2.当信号被订阅, 也就是调用signal的subscribeNext:nextBlock
// 2.2 subscribeNext内部会创建订阅者subscriber, 并且把nextBlock保存到subscriber中。
// 2.1 subscribeNext内部会调用siganl的didSubscribe
// 3.siganl的didSubscribe中调用[subscriber sendNext:@1];
// 3.1 sendNext底层其实就是执行subscriber的nextBlock

// 1.创建信号
RACSignal *siganl = [RACSignal createSignal:
                ^RACDisposable *(id<RACSubscriber> subscriber) {

    // block调用时刻: 每当有订阅者订阅信号, 就会调用block。
```

```objc
// 2.发送信号
    [subscriber sendNext:@1];
    //如果不在发送数据，最好发送信号完成，
    //内部会自动调用[RACDisposable disposable]取消订阅信号。
    [subscriber sendCompleted];

    return [RACDisposable disposableWithBlock:^{

        //block调用时刻：当信号发送完成或者发送错误，
        //就会自动执行这个block,取消订阅信号。
        // 执行完Block后，当前信号就不在被订阅了。

        NSLog(@"信号被销毁");

    }];
}];

// 3.订阅信号,才会激活信号.
[siganl subscribeNext:^(id x) {
    // block调用时刻：每当有信号发出数据，就会调用block.
    NSLog(@"接收到数据:%@",x);
}];
```

代码-5

```objc
// RACSubject使用步骤
// 1.创建信号 [RACSubject subject]，跟RACSiganl不一样，创建信号时没有block。
// 2.订阅信号 - (RACDisposable *)subscribeNext:(void (^)(id x))nextBlock
// 3.发送信号 sendNext:(id)value

// RACSubject:底层实现和RACSignal不一样。
// 1.调用subscribeNext订阅信号，只是把订阅者保存起来，并且订阅者的nextBlock已经赋值了。
// 2.调用sendNext发送信号，遍历刚刚保存的所有订阅者，一个一个调用订阅者的nextBlock。

// 1.创建信号
RACSubject *subject = [RACSubject subject];

// 2.订阅信号
[subject subscribeNext:^(id x) {
    // block调用时刻：当信号发出新值，就会调用.
    NSLog(@"第一个订阅者%@",x);
}];
[subject subscribeNext:^(id x) {
    // block调用时刻：当信号发出新值，就会调用.
    NSLog(@"第二个订阅者%@",x);
}];

// 3.发送信号
[subject sendNext:@"1"];
```

```objc
// RACReplaySubject使用步骤:
// 1.创建信号 [RACSubject subject]，跟RACSiganl不一样，创建信号时没有block。
// 2.可以先订阅信号，也可以先发送信号。
// 2.1 订阅信号 - (RACDisposable *)subscribeNext:(void (^)(id x))nextBlock
// 2.2 发送信号 sendNext:(id)value

// RACReplaySubject:底层实现和RACSubject不一样。
// 1.调用sendNext发送信号，把值保存起来，然后遍历刚刚保存的所有订阅者，一个一个调用订阅者的nextBlock。
// 2.调用subscribeNext订阅信号，遍历保存的所有值，一个一个调用订阅者的nextBlock

// 如果想当一个信号被订阅，就重复播放之前所有值，需要先发送信号，在订阅信号。
// 也就是先保存值，在订阅值。

// 1.创建信号
RACReplaySubject *replaySubject = [RACReplaySubject subject];

// 2.发送信号
[replaySubject sendNext:@1];
[replaySubject sendNext:@2];

// 3.订阅信号
[replaySubject subscribeNext:^(id x) {

    NSLog(@"第一个订阅者接收到的数据%@",x);
}];

// 订阅信号
[replaySubject subscribeNext:^(id x) {

    NSLog(@"第二个订阅者接收到的数据%@",x);
}];
```

代码-6

```objc
// 需求:
// 1.给当前控制器添加一个按钮，modal到另一个控制器界面
// 2.另一个控制器view中有个按钮，点击按钮，通知当前控制器

//步骤一：在第二个控制器.h，添加一个RACSubject代替代理。
@interface TwoViewController : UIViewController
@property (nonatomic, strong) RACSubject *delegateSignal;
@end

//步骤二：监听第二个控制器按钮点击
@implementation TwoViewController
- (IBAction)notice:(id)sender {
    // 通知第一个控制器，告诉它，按钮被点了

     // 通知代理
     // 判断代理信号是否有值
    if (self.delegateSignal) {
        // 有值，才需要通知
        [self.delegateSignal sendNext:nil];
    }
}
@end
```

```objc
//步骤三：在第一个控制器中，监听跳转按钮，给第二个控制器的代理信号赋值，并且监听。
@implementation OneViewController
- (IBAction)btnClick:(id)sender {

    // 创建第二个控制器
    TwoViewController *twoVc = [[TwoViewController alloc] init];

    // 设置代理信号
    twoVc.delegateSignal = [RACSubject subject];

    // 订阅代理信号
    [twoVc.delegateSignal subscribeNext:^(id x) {

        NSLog(@"点击了通知按钮");
    }];

    // 跳转到第二个控制器
    [self presentViewController:twoVc animated:YES completion:nil];
}
@end
```

代码-7

```objc
// 1.代替代理
// 需求：自定义redView,监听redView中按钮点击
// 之前都是需要通过代理监听，给redView添加一个代理属性，点击按钮的时候，通知代理做事情
// rac_signalForSelector:把调用某个对象的方法的信息转换成信号，就要调用这个方法，就会发送信号。
// 这里表示只要redV调用btnClick:,就会发出信号，订阅就好了。
[[redV rac_signalForSelector:@selector(btnClick:)] subscribeNext:^(id x) {
    NSLog(@"点击红色按钮");
}];

// 2.KVO
// 把监听redV的center属性改变转换成信号，只要值改变就会发送信号
// observer:可以传入nil
[[redV rac_valuesAndChangesForKeyPath:@"center"
    options:NSKeyValueObservingOptionNew
    observer:nil
    ] subscribeNext:^(id x) {

    NSLog(@"%@",x);

}];

// 3.监听事件
// 把按钮点击事件转换为信号，点击按钮，就会发送信号
[[self.btn rac_signalForControlEvents:UIControlEventTouchUpInside
    ] subscribeNext:^(id x) {

    NSLog(@"按钮被点击了");
}];

// 4.代替通知
// 把监听到的通知转换信号
[[[NSNotificationCenter defaultCenter]
    rac_addObserverForName:UIKeyboardWillShowNotification
    object:nil
    ] subscribeNext:^(id x) {
    NSLog(@"键盘弹出");
}];
```

```objc
// 5.监听文本框的文字改变
[_textField.rac_textSignal subscribeNext:^(id x) {

   NSLog(@"文字改变了%@",x);
}];

// 6.处理多个请求，都返回结果的时候，统一做处理。
RACSignal *request1 = [RACSignal createSignal:
    ^RACDisposable *(id<RACSubscriber> subscriber) {

    // 发送请求1
    [subscriber sendNext:@"发送请求1"];
    return nil;
}];

RACSignal *request2 = [RACSignal createSignal:
    ^RACDisposable *(id<RACSubscriber> subscriber) {
    // 发送请求2
    [subscriber sendNext:@"发送请求2"];
    return nil;
}];

// 使用注意：几个信号，参数一的方法就几个参数，每个参数对应信号发出的数据。
[self rac_liftSelector:@selector(updateUIWithR1:r2:)
    withSignalsFromArray:@[request1,request2]];


}
// 更新UI
- (void)updateUIWithR1:(id)data r2:(id)data1
{
NSLog(@"更新UI%@  %@",data,data1);
}
```