响应式编程与Combine

1.异步编程与响应式编程

相比同步程序而言，其实异步操作在客户端开发中更为常见。举个例子：

1.数据请求，请求结束后处理展示数据。2.等待用户点击某个按钮。3.下载图片、音视频到本地。4.UI动画，动画完成后处理某些事情。5.某个SDK处理事件（比如文字识别，OCR等等），处理完后回调。.....

而在处理这些事情的时候我们往往会用三个东西1.闭包回调（block）2代理3.通知。但是你再仔细想想，在异步编程中，不论采用闭包，delegate 还是 notification，实际上都是在当前的时间节点上预先设置好特定的逻辑，去处理未来会发生的事件。所以抛开不同 API 的定义所产生的表象，异步编程的本质是响应未来的事件流。

那么，我们完全完全可以用一种通用的方式来，统一这个事情，让事件发生这个核心概念凸显出来。而这个统一的结果就是响应式编程，其核心思路如下：

事件发布-------->操作变形-------->订阅使用。

异步操作在合适的时机发布事件，这些事件带有数据，使用一个或多个操作来处理这些事件以及内部的数据。在末端，使用一个订阅者来“消化”这个事件和数据，并进一步驱动程序的其他部分 (比如 UI 界面) 的运行。上面这些对于事件和数据的操作，以及末端的订阅，都是在事件发生之前完成的。一开始我们就将这些设定好，之后它可以以预设的方式响应源源不断发生的事件流。

2.Combine基本介绍

Combine是什么?

WWDC 2019 上 Apple 公布了声明式全新界面框架 SwiftUI，以及配套的响应式编程框架 Combine。

与SwiftUI一样，Combine最低支持iOS13。但是，发布之后社区热情高涨，自发完成了一个Combine的开源版库OpenCombine。 而OpenCombine只需要你在Xcode 10.2和Swift 5.0或更高版本上运行就好。（所以理论上最低可以支持到iOS8.0）。所以想用Combine又怕兼容版本太高就尽情的使用OpenCombine吧！

但其实响应式编程在iOS上已经掀起过不少热潮了，例如OC下的ReactiveCocoa Swift下的Rx套件RxSwift、RxCocoa，那么为什么官方出一个Combine大家就这么激动呢。究其原因我觉得很简单。1.相比于三方的优秀实现，官方版本更加稳定（bug少，不需要引入大量代码，更好的底层优化，性能更强。）2.这类官方框架的出现代表着官方对响应是编程的认可。

Combine的基本介绍

相对于前面总结的响应式思路，Combine恰好有三个最重要的角色：负责发布事件的 Publisher，负责订阅事件的 Subscriber，以及负责转换事件和数据的 Operator。即：

Publisher-------->Operator-------->Subscriber

下面简单看下 这些抽象的协议定义源码：

Publisher

```
public protocol Publisher {
     // 发布值类型，关联类型，订阅值要与之相同
    associatedtype Output
    // 发布值错误类型 不发布错误就用 Never
    associatedtype Failure : Error
    // 用于订阅者订阅
    func receive<S>(subscriber: S) where S : Subscriber, Self.Failure == S.Failure,
Self.Output == S.Input
}
```

从定义也可以知道，其实就是Publisher其实就2个事情，一个发布值，一个接受订阅。

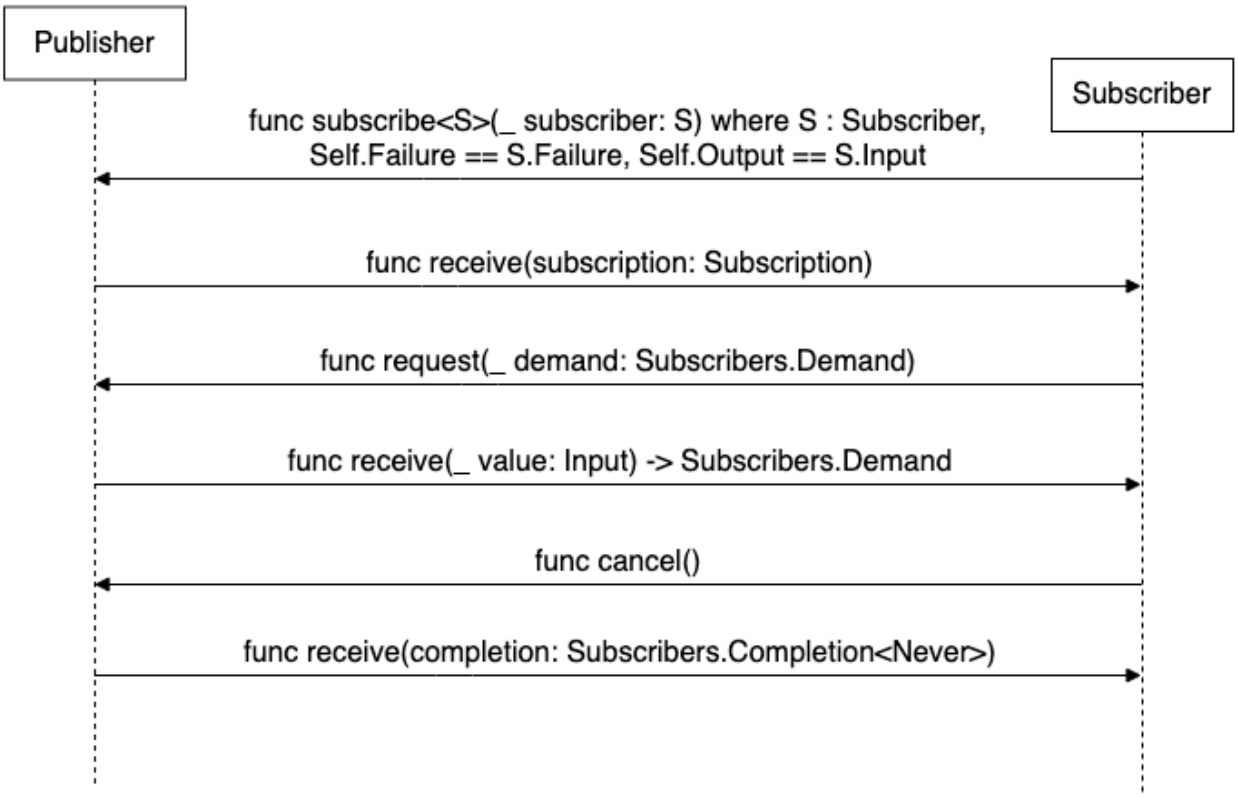
Subscriber

```
public protocol Subscriber : CustomCombineIdentifierConvertible {
  // 订阅者要接收的值的类型 要与发布值相同
    associatedtype Input
    associatedtype Failure : Error
    // 告知订阅者成功订阅了发布者
    func receive(subscription: Subscription)
    // 告知订阅者，发布者已经发布了一个元素
    func receive(_ input: Self.Input) -> Subscribers.Demand
   // 告知订阅者，发布者已经结束了发布，可能是正常结束，也可能是因为发生了错误
    func receive(completion: Subscribers.Completion<Self.Failure>)
}

public protocol Subscription : Cancellable, CustomCombineIdentifierConvertible {
    // 通知发布者，它可以向订阅者发送一个或多个值
    func request(_ demand: Subscribers.Demand)
}
```

从上面可以看到负责订阅事件的订阅者的定义。其实本质也是为了接受事件和处理事件。

把他们穿起来整个流程如下：



1. Subscriber 被绑定到 Publisher 上。
2. Publisher 创建订阅对象(subscription)，并将订阅对象发送给 Subscriber。
3. Subscriber 通过订阅对象将需求发送给 Publisher，告知需要多少个值。
4. Publisher 根据需求，将内容发送给 Subscriber。
5. Subscriber 通过订阅对象来向 Publisher 请求取消订阅（当然这个步骤不是必须的）。
6. Publisher 向 Subscriber 发送完成内容（也可能是错误内容）。

至此整个Combine的核心处理逻辑就分析清楚了。

当然我们还有个重要角色漏掉了就是Operator。

Operator

```
extension Publisher {
    public func scan<T>(_ initialResult: T, _ nextPartialResult: @escaping (T,
Self.Output) -> T) -> Publishers.Scan<Self, T>
}

let buttonClicked: AnyPublisher<Void, Never>
buttonClicked.scan(0) { value, _ in value + 1 }
```

这里看一个简单的Operator定义，这里有个例子是在点击后传入一个初始值，后续点击后直接返回点击的次数。

在 Combine 框架中，类似的用来操作数据和事件的函数还有很多，它们大多数都以函数式的形式出现，来对原有的 Publisher 进行变形等逻辑操作。这些操作符在响应式异步编程中担任的就是 Operator 的角色。

每个 Operator 的行为模式都一样：它们使用上游 Publisher 所发布的数据作为输入，以此产生的新的数据，然后自身成为新的 Publisher，并将这些新的数据作为输出，发布给下游。通过一系列组合，我们可以得到一个响应式的 Publisher 链条：当链条最上端的 Publisher 发布某个事件后，链条中的各个 Operator 对事件和数据进行处理。在链条的末端我们希望最终能得到可以直接驱动 UI 状态的事件和数据。这样，终端的消费者可以直接使用这些准备好的数据，而这个消费者的角色由 Subscriber 来担任。

3.Combine的案例分析

字典数组转Publisher

```
// 这里定义一个函数帮助我们 查看
public func check<P: Publisher>(
    _ title: String,
    publisher: () -> P
) -> AnyCancellable
{
    print("----- \(title) -----")
    defer { print("") }
    return publisher()
        .print()
        .sink(
            receiveCompletion: { _ in},
            receiveValue: { _ in }
        )
}
check("Map"){
[1,2,3]
    .publisher
    .map { $0 * 2 }
}
```

这里个例子就很好的解释了数组转publisher，然后订阅的过程。打印日志如下

```
----- Map -----
receive subscription: ([2, 4, 6])
request unlimited
receive value: (2)
receive value: (4)
receive value: (6)
receive finished
```

从这里也可以看出来，确实过程跟我们之前说的是一样的。

通知与定时器

Foundation 中的 NotificationCenter 和Timer提供了创建 Publisher 的辅助 API

定义如下：

```
// 通知名，通知对象
public func publisher(for name: Notification.Name, object: AnyObject? = nil) ->
NotificationCenter.Publisher
public static func publish(every interval: TimeInterval, tolerance: TimeInterval? =
nil, on runLoop: RunLoop, in mode: RunLoop.Mode, options: RunLoop.SchedulerOptions? =
nil) -> Timer.TimerPublisher
```

```
NotificationCenter.default.publisher(for: testNotification).sink { _ in
    print("hello wold!")

}
```

```
var timer:Timer.TimerPublisher = Timer.publish(every: 1, on: .main, in: .default)
timer.sink {_ in
    print("hello wold!")
}
timer.connect()
```

上面是简单的例子

KVO

```
let textfield = UITextField.init(frame: CGRect(x: 100, y: 250, width: 100, height: 50))
let textLabel = UILabel.init(frame: CGRect(x: 100, y: 350, width: 100, height: 50))
textfield.publisher(for: \.text,options: NSKeyValueObservingOptions.new).assign(to:
\.text, on: textLabel)
```

其中assig也是一种订阅方式

```
public func assign<Root>(to keyPath: ReferenceWritableKeyPath<Root, Self.Output>, on
object: Root) -> AnyCancellable
```

按钮的点击事件

在这个案例分析前需要分析一个另外的对象Subject,我们可以看下他的定义

```
public protocol Subject : AnyObject, Publisher {
    func send(_ value: Self.Output)
    func send(completion: Subscribers.Completion<Self.Failure>)
    func send(subscription: Subscription)
}
```

从定义上就可以看出来他就是一个遵循Publisher协议的对象，而且他多了3个方法，用于发送信号。而Combine内置提供了两种常用的 Subject 类型，分别是 PassthroughSubject 和 CurrentValueSubject。下面我们就要用到其中一个PassthroughSubject。

```
 let publisher2 = PassthroughSubject<Int, Never>()
    publisher2.send(1)
    print("开始订阅")
    publisher2.sink(
        receiveCompletion: { complete in
            print(complete)
        },
        receiveValue: { value in
            print(value)
        }
    )
    publisher2.send(2)
    publisher2.send(completion: .finished)
```

PassthroughSubject 简单地将 send 接收到的事件转发给下游的其他 Publisher 或 Subscriber。它的特性是从订阅开始才接受信号。然后我们想想按钮的点击事件是不是可以用这个来封装呢?

于是很自然的想到如下代码

```
let button = UIButton.init(frame: CGRect(x:100, y: 100, width: 100, height: 100))
  button.backgroundColor = .yellow;
  button.addTarget(self, action: #selector(testAction), for: .touchUpInside)
  self.view.addSubview(button);

 let buttonClicked = PassthroughSubject<Void, Never>()
 let  test = buttonClicked
      .scan(0) { value, _ in value + 1 }
      .map { String($0) }
      .sink {
          print("Button pressed count: \($0)")

      }

@objc func testAction() {
      buttonClicked.send()
  }
```

到这里你可能会觉得这跟之前好像也没啥区别啊，感觉没什么用。OK我们再封装下：

```
class TestButton : UIButton {

    lazy var buttonPublisher:PassthroughSubject<Void, Never> = {
        self.addTarget(self, action: #selector(testAction), for: .touchUpInside)
        return PassthroughSubject<Void, Never>()
    }()

    func getPublisher() ->  PassthroughSubject<Void, Never> {
        return buttonPublisher
```

```
    }

    @objc func testAction() {
        buttonPublisher.send()
    }
}

let button2 = TestButton.init(frame: CGRect(x: 100, y: 300, width: 100, height: 100))
    button2.backgroundColor = .blue
    self.view.addSubview(button2);
    let test2 = button2.getPublisher().scan(0) { value, _ in value + 1 }
        .map { String($0) }
        .sink {
            print("Button2 pressed count: \($0)")

        }
```

这样看是不是就很舒服了。这就是响应式编程带来的修改与变化。

总结

看到这里，可能很多小伙伴会迫不及待的说，Combine真好用，我们搞起来。但是等等，其实分享的前面我已经提过了：**WWDC 2019 上 Apple 公布了声明式全新界面框架 SwiftUI，以及配套的响应式编程框架 Combine。** Combine最强大的点在于与SwiftUI配套使用，在SwiftUI里有很多相应的配套封装。那UIKit呢，对不起没有。如果你要用，就像我的案例里一样需要自己封装。那这时候有更好的选择吗？有的老牌框架RxSwift。所以我这里建议是如果要兼容iOS13以前的，还是只能用RxSwift配合它的伙伴RxCocoa在UIKit里的兼容还是不错的。如果你兼容iOS13以上的那么我觉得可以直接干SwiftUI+Combine。毕竟SwiftUI性能上是优于UIKit的，当然Combine理论上也优于RxSwift，毕竟都是官方维护的。当然，如果你大部分处理数据，与一些简单的绑定那么就算你兼容的版本比较低Combine依然是很好的选择。

参考：SwiftUI与Combine编程