理顺 RxSwift 核心知识

首先，让我们忘掉什么 Observable, Observer, 直接回归问题的本质。

基本需求: viewModel 里的一个网络请求，服务端下发一组数据，viewController 里收到正确数据，刷新UI。

如果没有RxSwift，代码可能是这样的:

```
func request(_ success: @escaping ((String) -> ())) {
    // 假设这是一个网络请求
    DispatchQueue.main.asyncAfter(deadline: .now() + 2) {
        success("我是一条数据")
    }
}
--------------
// vc 中就是这样的
viewModel.request { data in
    print(data)
}
```

问题是，如果我有多个UI状态变更需要这组数据，且这些变更毫不相关，那在vc 里面是不是要多次调用 request方法?
这时候我们可能会写出这样的代码

```
var dataHandler1: ((String) -> Void)?
var dataHandler2: ((String) -> Void)?
var dataHandler3: ((String) -> Void)?
// request 变成了这样
func request() {
    // 假设这是一个网络请求
    DispatchQueue.main.asyncAfter(deadline: .now() + 2) {
        self.dataHandler1?("我是一条数据")
        self.dataHandler2?("我是一条数据")
        self.dataHandler3?("我是一条数据")
    }
}
--------------
// vc 中变成了这样
viewModel.dataHandler1 = { data in
```

```
    // 执行UI1状态变更
}
viewModel.dataHandler2 = { data in
    // 执行UI2状态变更
}
...
```

这样太丑了，这些笨重的 handler 做的事差不多是一样的，为什么不把他们放到一个数组统一处理？另外，viewModel 只是负责提供请求数据的，他根本不需要知道究竟有多少个 handler，每一个 handler 具体做了什么。
所以，我们应该自己封装一个 handler 的管理类，这个类里面存放了一组 handler，并且每一次网络请求成功回调后，执行所有的 handler。

一个最简单的事件序列，大概就是这样的：

```
// 传入的事件类型，先暂时忽略 error，completed 等类型
// 这里只关注 next 事件
enum TestEvent<Element> {
    case next(Element)
}

class TestSubject<Element> {
    // handler 事件变成了 Observer
    typealias Observer = (TestEvent<Element>) -> Void

    init() {

    }
    // 存放 Observer 的数组
    var observers: [Observer] = []
    // on 被调用，事件产生，遍历 observers， 执行每一个 observer
    func on(_ event: TestEvent<Element>) {
        dispatch(event)
    }
    // 只要新增一个 observer，就把他加入到事件序列里
    func subscribe(_ observer: @escaping Observer) {
        observers.append(observer)
    }
    // 清空事件序列，清空之后，之前添加的 observer 全部失效，但是不耽误
    // 之后的 observer 添加，之后会聊到 DisposeBag 的实现机制
    func dispose() {
```

```
        observers.removeAll()
    }

    private func dispatch(_ event: TestEvent<Element>) {
        observers.forEach {
            $0(event)
        }
    }
}
```

这样，viewModel 和 vc 的代码变得更加清晰和可维护,对于vc中的UI状态管理也更加统一

```
// ViewModel
let testSubject = TestSubject<String>()

func request() {
    let timer = Observable<Int>.interval(.seconds(1), scheduler: MainScheduler.ins
tance)
    timer.subscribe(onNext: { [weak self] time in
        self?.testSubject.on(.next("发射"))
    }).disposed(by: bag)
}
-----------
// vc
viewModel.testSubject.subscribe { event in
    if case let .next(element) = event {
        print("observer1 --- \(element)")
    }
}
viewModel.testSubject.subscribe { event in
    if case let .next(element) = event {
        print("observer2 --- \(element)")
    }
}
```

像 TestSubject 这样既可以作为观察者产生事件，又可以作为被观察者处理事件的，在 RxSwift 里，可以参考 PublishSubject，BehaviorSubject

现实世界的 Observable, Observer 和 Disposable

让我们从一个 Observable的创建，订阅，和销毁逐步分析

```
let testObservable = Observable<String>.create {
    observer -> Disposable in
    observer.onNext("1111")
    return Disposables.create {
        print("销毁了")
    }
}
let testDisposable = testObservable.subscribe(onNext: { value in
    print(value)
})

testDisposable.disposed(by: bag)
```

首先我们思考三个问题：

1. creat 闭包内容是何时被调用的?
2. subscribe 方法的 onNext事件是何时被调用的?
3. Dsiposable 是何时清除不再需要的资源的?

第一个问题: creat 闭包内容是何时被调用的?

首先看一下 creat 函数

```
extension ObservableType {
    public static func create(_ subscribe: @escaping (AnyObserver<Element>) -> Disposable) -> Observable<Element> {
        AnonymousObservable(subscribe)
    }
}
// 参数是一个 以 AnyObserver 作为参数, 以 Disposable 作为返回值的闭包, 返回值就是一个 Observable
// 函数内返回了一个 AnonymousObservable
```

下面看一下 AnonymousObservable 到底做了什么

```
final private class AnonymousObservable<Element>: Producer<Element> {
    typealias SubscribeHandler = (AnyObserver<Element>) -> Disposable
    // 首先, 把 creat 函数传进来的 闭包 作为一个存储属性保存了下来
    let subscribeHandler: SubscribeHandler
    // 保存闭包
```

```
    init(_ subscribeHandler: @escaping SubscribeHandler) {
        self.subscribeHandler = subscribeHandler
    }
    // 其次是一个 run 函数继承自父类，run 函数何时被调用不清楚，只能查看父类
    override func run<Observer: ObserverType>(_ observer: Observer, cancel: Cancel
able) -> (sink: Disposable, subscription: Disposable) where Observer.Element == El
ement {
        let sink = AnonymousObservableSink(observer: observer, cancel: cancel)
        let subscription = sink.run(self)
        return (sink: sink, subscription: subscription)
    }
}
```

AnonymousObservable 是 Observable的子类，继承关系是: AnonymousObservable -> Producer -> Observable -> ObservableType -> ObservableConvertibleType

接着，我们查看一下 Producer(为了方便阅读代码，我会把源码中和事件逻辑不相关的全部删掉，如果对源码感兴趣，可以自行查阅)

```
class Producer<Element>: Observable<Element> {
    override init() {
        super.init()
    }

    override func subscribe<Observer: ObserverType>(_ observer: Observer) -> Dispo
sable where Observer.Element == Element {
       let disposer = SinkDisposer()
       // 我们只需要关注这一行即可
       let sinkAndSubscription = self.run(observer, cancel: disposer)
       disposer.setSinkAndSubscription(sink: sinkAndSubscription.sink, subscriptio
n: sinkAndSubscription.subscription)
       return disposer
    }
    // AnonymousObservable 重写的就是这个方法，忽略父类的run，直接分析
    // AnonymousObservable 的 run 函数即可
    func run<Observer: ObserverType>(_ observer: Observer, cancel: Cancelable) ->
(sink: Disposable, subscription: Disposable) where Observer.Element == Element {
        rxAbstractMethod()
    }
}
```

因为 Producer 继承自 Observable，遵守了 ObservableType 协议，所以它实现了 subscribe 方

法，subscribe 方法调用了子类 AnonymousObservable 的 run 方法（一定要记住这里 run 的调用时机，待会儿会回来解释）。
那么，subscribe 方法又是在何时调用的呢?
来到 ObservableType+Extensions.swift 文件

```
extension ObservableType {

    public func subscribe(
        onNext: ((Element) -> Void)? = nil,
        onError: ((Swift.Error) -> Void)? = nil,
        onCompleted: (() -> Void)? = nil,
        onDisposed: (() -> Void)? = nil
    ) -> Disposable {
            let disposable: Disposable

            if let disposed = onDisposed {
                disposable = Disposables.create(with: disposed)
            }
            else {
                disposable = Disposables.create()
            }

            let observer = AnonymousObserver<Element> { event in

                switch event {
                case .next(let value):
                    onNext?(value)
                case .error(let error):
                    onError?(error)
                    disposable.dispose()
                case .completed:
                    onCompleted?()
                    disposable.dispose()
                }
            }
            return Disposables.create(
                self.asObservable().subscribe(observer),
                disposable
            )
    }
}
```

当testObservable 调用了 subscribe 函数后

```
testObservable.subscribe(onNext: { value in
    print(value)
})
```

函数内部帮我们创建了一个 AnonymousObserver对象，在初始化时传递进来一个闭包，并持有这个闭包_eventHandler，AnonymousObserver是匿名观察者，用于存储，和处理事件。
然后来到本函数的核心代码

```
self.asObservable().subscribe(observer)
```

还记得Producer的 subscribe 函数吗？self.asObservable返回的是 Producer对象自己，Producer的subscribe 函数就是在这里调用的！然后这里执行了Producer的子类 AnonymousObservable 的 run 函数。再回到 run 函数

```
override func run<Observer: ObserverType>(_ observer: Observer, cancel: Cancelable
)
-> (sink: Disposable, subscription: Disposable)
where Observer.Element == Element {
        let sink = AnonymousObservableSink(observer: observer, cancel: cancel)
        // AnonymousObservableSink的 run 函数往后看
        let subscription = sink.run(self)
        return (sink: sink, subscription: subscription)
    }
```

AnonymousObservableSink 是专门为 AnonymousObservable而服务的 sink,sink 可以理解成管道，是将Observable的事件传递给Observer的桥梁。（后面几乎所有继承自Observable的特殊类型都是这么设计的）

```
final private class AnonymousObservableSink<Observer: ObserverType>: Sink<Observer
>, ObserverType {
    typealias Element = Observer.Element
    typealias Parent = AnonymousObservable<Element>

    // state
    private let isStopped = AtomicInt(0)

    override init(observer: Observer, cancel: Cancelable) {
```

```
        super.init(observer: observer, cancel: cancel)
    }

    func on(_ event: Event<Element>) {
        switch event {
        case .next:
            if load(self.isStopped) == 1 {
                return
            }
            self.forwardOn(event)
        case .error, .completed:
            if fetchOr(self.isStopped, 1) == 0 {
                self.forwardOn(event)
                self.dispose()
            }
        }
    }

    func run(_ parent: Parent) -> Disposable {
        parent.subscribeHandler(AnyObserver(self))
    }
}
```

AnonymousObservableSink 的 run 函数真正调用了AnonymousObservable 所持有的subscribeHandler，同时给闭包传入了一个 AnyObserver(self)（AnyObserver的初始化方法，参数是一个遵守了 ObserverType 协议的类型，也就是AnonymousObservableSink自己），还记得subscribeHandler 是什么吗？没错，是creat函数的参数，闭包内容是在这里被执行的，也就是下面的代码

```
let testObservable = Observable<String>.create {
    observer -> Disposable in
    observer.onNext("1111")
    return Disposables.create {
        print("销毁了")
    }
}
```

可以看到，此时此刻，observer 的 onNext 方法被执行了，这个observer就是刚刚那个AnyObserver（self）。至此，我们解决了第一个问题: "***creat 闭包内容是何时被调用的?*** "

第二个问题：subscribe 方法的 onNext事件是何时被调用的?

接下来，我们应该着眼于 AnyObserver 的初始化方法

```
public struct AnyObserver<Element> : ObserverType {
    public typealias EventHandler = (Event<Element>) -> Void
    private let observer: EventHandler

    public init(eventHandler: @escaping EventHandler) {
        self.observer = eventHandler
    }

    // 外面的 “AnyObserver (self) ”就是调用的这个方法
    public init<Observer: ObserverType>(_ observer: Observer) where Observer.Eleme
nt == Element {
        // 这里比较绕，需要仔细想一想
        // AnyObserver (self) 传进来的 observer 是 AnonymousObservableSink对象
        // 这句代码的意思是把 AnonymousObservableSink 的 on 方法传给了
        // observer 变量，因为这个 observer 本身是一个 (Event<Element>) -> Void 类型，
        // AnonymousObservableSink 的 on 方法本质上也是 (Event<Element>) -> Void 类型
        self.observer = observer.on
    }
    // observer.onNext("")执行后，执行这个方法，具体定义参照 ObserverType 的代码
    // on 方法直接执行了 observer 这个闭包代码（千万别忘了 observer 其实就是 AnonymousObse
rvableSink 的 on 方法）
    public func on(_ event: Event<Element>) {
        self.observer(event)
    }

    public func asObserver() -> AnyObserver<Element> {
        self
    }
}
```

当 creat 闭包中的 observer.onNext("")执行后，会来到 AnyObserver 的 on 方法，参照 ObserverType 这个协议中的定义(不管是 onNext, onCompleted 还是 onError 都是直接调用了 on 方法)

```
public protocol ObserverType {
    associatedtype Element

    func on(_ event: Event<Element>)
```

```
}
extension ObserverType {
    public func onNext(_ element: Element) {
        self.on(.next(element))
    }

    public func onCompleted() {
        self.on(.completed)
    }

    public func onError(_ error: Swift.Error) {
        self.on(.error(error))
    }
}
```

此时的 on 方法，正是 AnonymousObservableSink 的 on 方法，代码如下

```
final private class AnonymousObservableSink<Observer: ObserverType>: Sink<Observer
>, ObserverType {

    func on(_ event: Event<Element>) {
        switch event {
        case .next:
            if load(self.isStopped) == 1 {
                return
            }
            self.forwardOn(event)
        case .error, .completed:
            if fetchOr(self.isStopped, 1) == 0 {
                self.forwardOn(event)
                self.dispose()
            }
        }
    }

    func run(_ parent: Parent) -> Disposable {
        parent.subscribeHandler(AnyObserver(self))
    }
}
```

这个方法调用了父类 Sink 的 forwardOn 函数，forwardOn 函数真正调用了我们 subscribe 方法内生成的那个 AnonymousObserver（匿名监听者）的 on 方法。

而 event 参数就是 creat 函数内的 onNext 事件。

```
class Sink<Observer: ObserverType>: Disposable {
    // ...
    final func forwardOn(_ event: Event<Observer.Element>) {
        if isFlagSet(self.disposed, 1) {
            return
        }
        self.observer.on(event)
    }
}
```

目前为止，我们追到了 AnonymousObserver的 on 方法，AnonymousObserver 没有 on 方法，只有一个 onCore 方法，继续去父类 ObserverBase 找， 父类的 on 方法调用了子类onCore，onCore 调用了 eventHandler，并且把 上一步 forwardOn方法传入的 event 传进了闭包。

```
final class AnonymousObserver<Element>: ObserverBase<Element> {
    typealias EventHandler = (Event<Element>) -> Void

    private let eventHandler : EventHandler

    init(_ eventHandler: @escaping EventHandler) {
        self.eventHandler = eventHandler
    }

    override func onCore(_ event: Event<Element>) {
        self.eventHandler(event)
    }
}

class ObserverBase<Element> : Disposable, ObserverType {
    private let isStopped = AtomicInt(0)

    func on(_ event: Event<Element>) {
        switch event {
        case .next:
            // 暂时忽略 load 和 fetchOr 这样的判断方法
            // 他们与observer的调用逻辑无关
            if load(self.isStopped) == 0 {
                self.onCore(event)
            }
        case .error, .completed:
```

```
            if fetchOr(self.isStopped, 1) == 0 {
                self.onCore(event)
            }
        }
    }

    func onCore(_ event: Event<Element>) {
        rxAbstractMethod()
    }

    func dispose() {
        fetchOr(self.isStopped, 1)
    }
}
```

AnonymousObserver 的 eventHandler 不就是最初 ObservableType 拓展中的 subscribe 方法内生成的 AnonymousObserver 的初始化参数吗！

```
public func subscribe(
        onNext: ((Element) -> Void)? = nil,
        onError: ((Swift.Error) -> Void)? = nil,
        onCompleted: (() -> Void)? = nil,
        onDisposed: (() -> Void)? = nil
    ) -> Disposable {
            let disposable: Disposable

            if let disposed = onDisposed {
                disposable = Disposables.create(with: disposed)
            }
            else {
                disposable = Disposables.create()
            }

            let observer = AnonymousObserver<Element> { event in
                //上面的 eventHandler被执行，  其实就是调用了这个代码块
                switch event {
                case .next(let value):
                    // 到这里，才真正的调用了我们最外层的
                    // subscribe (onNext: { // 收到信号后... })
                    onNext?(value)
                case .error(let error):
                    if let onError = onError {
                        onError(error)
```

```
                }
                // error 或者 completed 发出
                // 资源立即被释放
                disposable.dispose()
            case .completed:
                onCompleted?()
                disposable.dispose()
            }
        }
        return Disposables.create(
            self.asObservable().subscribe(observer),
            disposable
        )
    }
```

至此，我们解决了第二个问题: "subscribe 方法的 onNext事件是何时被调用的"。

> *追踪Observable的订阅过程也在侧面验证了两个问题:*
> *第一: 每增加一个Observer，Observable的creat方法的闭包就会被调用一次，联想到 share 操作符，共享事件结果。（造成的负面影响，如果有多个订阅，附加操作会被调用多次）*
> *第二:如果没有Observer来订阅 Observable，creat 方法的闭包是不会被执行的。*

第三个问题: Dsiposable 是何时清除不再需要的资源的?

- 一般的，序列如果发出了 error 或者 completed 事件，所有内部资源都会被释放，不需要我们手动释放(这个在前面 ObservableType+Extensions.swift 的subscribe 方法里面多次提到了)
- 如果我们需要提前释放这些资源或取消订阅的话，那我们可以对返回的 Disposable 调用 dipose 方法
- 官方推荐使用 DisposeBag,来管理订阅的生命周期，一般是把资源加入到一个全局的 DisposeBag 里面，它跟随着页面的生命周期，当页面销毁时 DisposeBag 也会随之销毁，同时 DisposeBag 里面的资源也会被一一释放。

由于时间关系，dispose 方法具体是如何执行的，本次分享就不再深入追踪源码了。我们只看一下常用的 DisposeBag 具体是怎么实现的。

```
extension Disposable {
    // 把当前disposabel加入到一个bag
    public func disposed(by bag: DisposeBag) {
        bag.insert(self)
    }
```

```
}

// DisposeBase 不需要关心(资源调试用的)
public final class DisposeBag: DisposeBase {

    private var lock = SpinLock()

    // 存放着所有被添加进来的 Disposable
    private var disposables = [Disposable]()
    private var isDisposed = false

    public override init() {
        super.init()
    }

    public func insert(_ disposable: Disposable) {
        self._insert(disposable)?.dispose()
    }

    private func _insert(_ disposable: Disposable) -> Disposable? {
        self.lock.performLocked {
            if self.isDisposed {
                return disposable
            }

            self.disposables.append(disposable)

            return nil
        }
    }
    private func dispose() {
        let oldDisposables = self._dispose()

        for disposable in oldDisposables {
            // 这里才真正的释放了所有的资源
            disposable.dispose()
        }
    }

    private func _dispose() -> [Disposable] {
        self.lock.performLocked {
              // 首先保存了一个 disposables 的临时变量
            let disposables = self.disposables
            // 随后清空 disposables
```

```
            self.disposables.removeAll(keepingCapacity: false)
            // isDisposed 设为 true（用于判断 bag 的资源是否被清空）
            self.isDisposed = true
            // 返回刚才的临时变量，具体的 dispose 是在后面进行的
            return disposables
        }
    }

    deinit {
        self.dispose()
    }
}
```

总结以下几点:

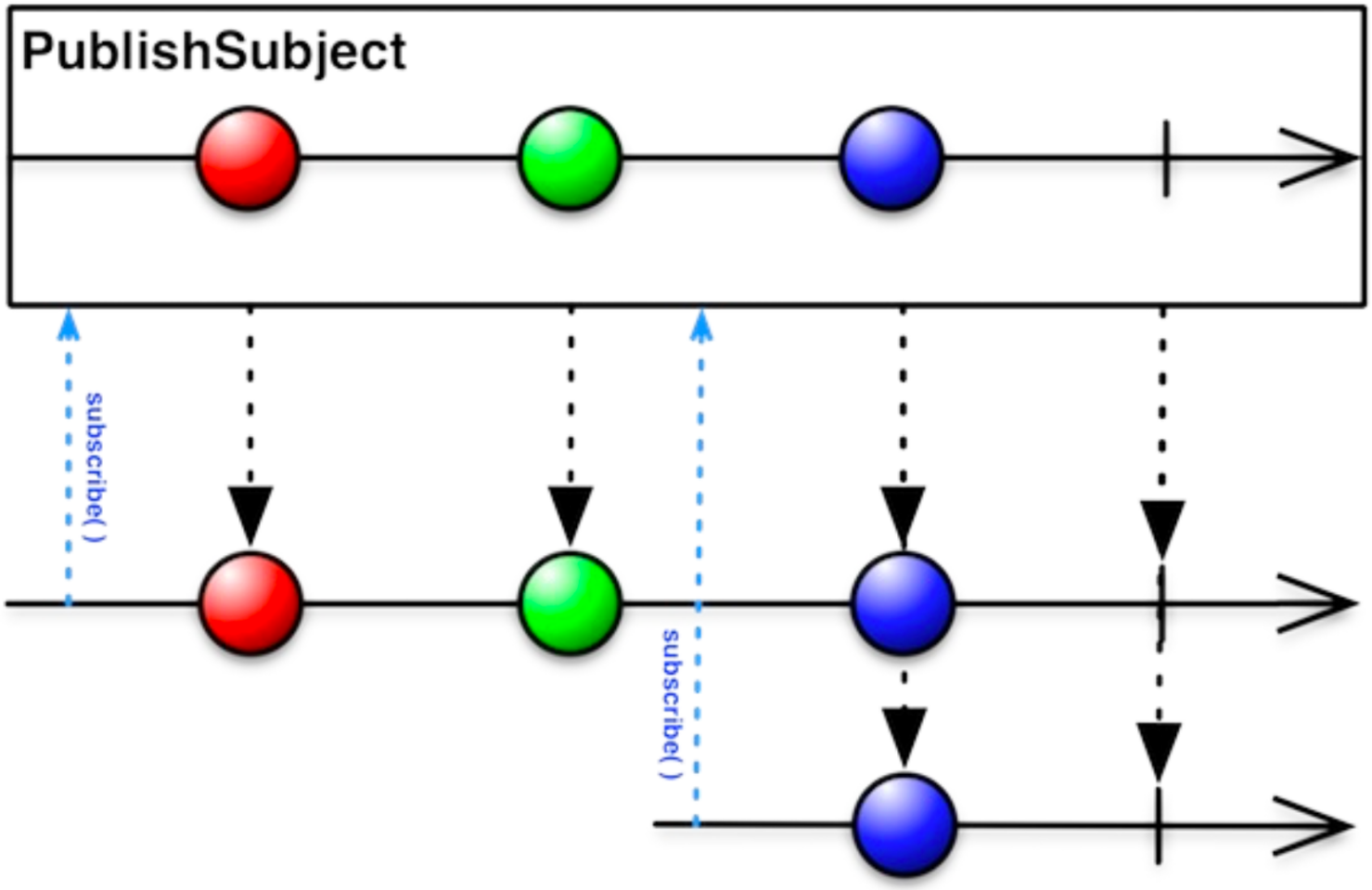
1. DisposeBag 有一个 disposables 属性， disposables 保存了所有被添加进来的 disposable。
2. insert 方法把 disposable 添加进 disposables数组中。
3. dispose 方法的具体执行时机是在 DisposeBag 的实例对象被销毁的时候。
4. DisposeBag 对象的 dispose 方法做了两件事，第一，清空 disposables 数组。 第二，遍历 disposables 数组，对每一个元素执行他自己的 dispose 方法。

RxSwift 实用篇

接下来聊一聊适应项目中不同场景的特殊的 Observable,以及那些特征序列

PublishSubject

正如本次分享开头提到的 TestSubject ， 他既可以作为事件序列被 Observer 监听，又可以作为 Observer 发出 on 方法产生事件序列。PublishSubject 将对观察者发送订阅后产生的元素，而在订阅前发出的元素将不会发送给观察者。




```
let disposeBag = DisposeBag()
let subject = PublishSubject<String>()

subject.subscribe { print("Subscription: 1 Event:", $0) }
 .disposed(by: disposeBag)

subject.onNext(" ")
subject.onNext(" ")

subject.subscribe { print("Subscription: 2 Event:", $0) }
.disposed(by: disposeBag)

subject.onNext("")
subject.onNext("")
```

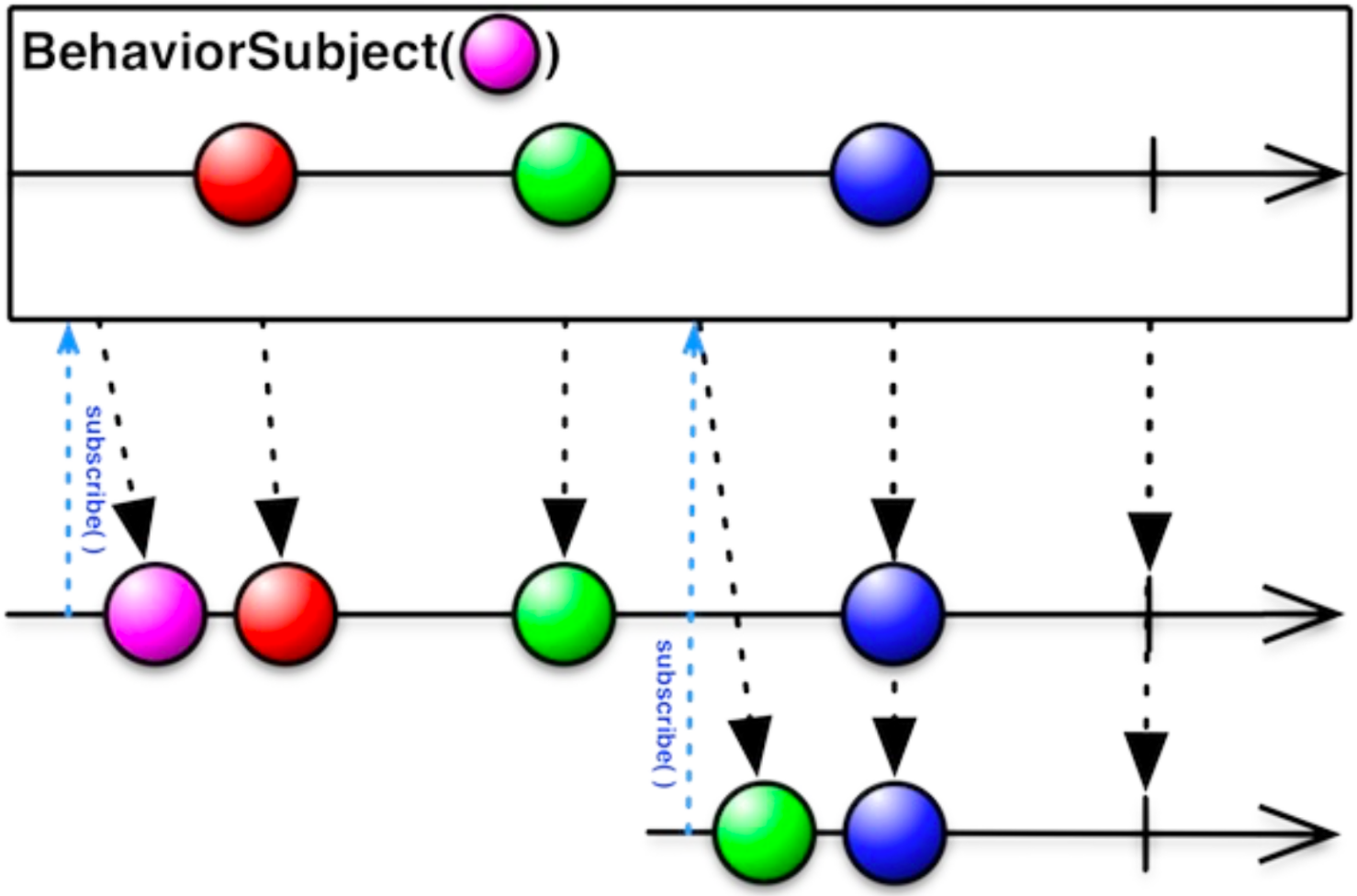
输出结果

```
Subscription: 1 Event: next( )
```

```
Subscription: 1 Event: next( )
Subscription: 1 Event: next()
Subscription: 2 Event: next()
Subscription: 1 Event: next()
Subscription: 2 Event: next()
```

BehaviorSubject

BehaviorSubject 的实现方式和 PublishSubject 其实基本上是一样的，唯一的区别就是当观察者对 BehaviorSubject 进行订阅时，它会将源 Observable 中最新的元素发送出来(如果不 存在最新的元素，就发出默认元素)。然后将随后产生的元素发送出来。



BehaviorSubject 的实现细节很有意思，他持有了一个属性 element，并且每次接收到 next 信号，就把新的元素赋值给他。所以每次我们接收到的回放都是这个 element 值。具体 BehaviorSubject 是如何做到回放的呢?非常简单，就是在他每次收到订阅（subscribe）的时候，主动调一次

```
observer.on(.next(self.element)) // 相当于把上次的 element 又发射了一次
```

RxRelay

PublishRelay

他就是 PublishSubject 去掉了 completed 和 error 事件，同样的，既是可监听序列，也是观察者。

```
public final class PublishRelay<Element>: ObservableType {
    private let subject: PublishSubject<Element>
    // 省掉了 PublishSubject 的 onError, onCompleted 事件
    public func accept(_ event: Element) {
        self.subject.onNext(event)
    }

    public init() {
        self.subject = PublishSubject()
    }

    /// Subscribes observer
    public func subscribe<Observer: ObserverType>(_ observer: Observer) -> Disposa
ble where Observer.Element == Element {
        self.subject.subscribe(observer)
    }

    /// - returns: Canonical interface for push style sequence
    public func asObservable() -> Observable<Element> {
        self.subject.asObservable()
    }
}
```

BehaviorRelay

BehaviorRelay 就是 BehaviorSubject 去掉终止事件 onError 或 onCompleted,参考 BehaviorSubject

Single

Single 要么只能发出一个元素，要么产生一个 error 事件,因为 single 的 subscribe 方法接收的事件类型 SingleEvent 是一个地地道道的 Swift 库中的 Result 枚举。如果他被某一个观察者 Observer 所监听，那么在收到 success 事件的时候，这个事件序列就已经结束了，所以它只能发射一个元素。

```
switch event {
```

```
case .success(let element):
    // 结束了
    observer.on(.next(element))
    observer.on(.completed)
case .failure(let error):
    observer.on(.error(error))
}
```

一个最直白的应用场景，就是执行 HTTP 请求，返回一个 Result 结果。

Completable

Completable 要么只能产生一个 completed 事件，要么产生一个 error 事件。
Completable 适用于那种你只关心任务是否完成，而不需要在意任务返回值的情况。它和 Observable 有点相似。
仔细想一想，Completable 其实在很多场景下都很有用，比如某一个定时器，如果超过了多长时间，就不再执行回调。此时使用 Completable 一方面语义更加清晰，另一方面在事件发出后，Completable 所引用的资源就已经被全部销毁了。下面是我们最新需求的一个例子,可以感受一下Completable的威力:

```
let speed = Observable<CGFloat>.create { (observer) -> Disposable in
    // 这里是伪代码，地图开启连续定位
    // 地图发出一个速度序列
    // observer.onNext(model.speed)

    /**
     return
     WSHitchDriverMapManager.shareInstance().rx
         .observeWeakly(Int.self, "remainingSpeed")
         .subscribe(onNext: { speed in
             observer.onNext(speed)
         })
     */


    return Observable<Int>.interval(.seconds(1), scheduler: MainScheduler.instance
).subscribe(onNext: { element in
        var speed = element
        // 第10秒以后 把 速度置为 0
        if element > 10 {
            speed = 0
        }
```

```
        print("speed = \(speed)")
        observer.onNext(CGFloat(speed))
    })
    // throttle 可以控制采集频率，两秒钟采集一次
}.throttle(.seconds(2), scheduler: MainScheduler.instance)

var lastSentTime: Date?
var pastTime: TimeInterval = 0

let completable = Completable.create { event -> Disposable in

    let dispose = speed.subscribe(onNext: { value in
        guard value == 0.0 else { return }

        if let lastSendingTime = lastSentTime {
            pastTime = Double(Date().timeIntervalSinceNow) - Double(lastSendingTim
e.timeIntervalSinceNow)
        } else {
            lastSentTime = Date()
        }

        print("pastTime = \(pastTime)")
        if pastTime >= 15.0 { // 速度为0的持续时间超过了 300 秒，发出事件
            event(.completed)
        }
    })
    // 我们把dispose加进去，是为了在 completable 销毁之后，把 speed 也销毁掉
    // 事实上，这么做也是非常符合需求的
    return Disposables.create { dispose.dispose() }
}

completable.subscribe(onCompleted: {
    // completable 序列即将被销毁，所持有的资源被清空
    print("您已在原地停留15秒~")
}).disposed(by: bag)
```

Maybe

Maybe 介于 Single 和 Completable 之间，它要么只能发出一个元素，要么产生一个 completed 事件，要么产生一个 error 事件。

适用场景: 如果遇到那种可能需要发出一个元素，又可能不需要发出时，就可以使用 Maybe。
这个我就想不到具体使用场景了~

Driver

Driver 主要是为了简化 UI 层的代码。不过如果我们遇到的序列具有以下特征，也可以使用它

- 不会产生 error 事件
- 一定在 MainScheduler 监听(主线程监听)
- 共享附加作用(自动实现了 share 方法)

driver 可以在日常开发中为我们避免很多麻烦，比如

1. 有的事件序列遇到 error 事件就结束了，但是我们不希望他结束，此时将 Obaervable 转成 Driver，调用 asDriver函数，会强制要求我们传一个默认值，以防止在遇到 error 的时候序列被销毁。
2. 在一个异步网络请求的回调中，我们拿到回调结果往往需要在主线程做一系列操作，driver 会默默的帮我们回到主线程。
3. 如果同时多个 observer 订阅了 driver，Observable 的附加代码只会调用一次(比如 request 只会请求一次)

RxSwift 涉及到的东西太多太多了，时间太短，本次就到这里吧👋~

还有两大块儿待探索，未完待续...

Scheduler

Operator(操作符)

下次一定~~😄