**Rxswift**

**Part 1: Cách dùng cơ bản**

Rxswift và RxCocoa là 1 phần của ReactiveX (đã được viết trên nhiều nền tảng và ngôn ngữ khác).

RxSwift là 1 framework để tương tác với swift, trong khi RxCocoa là 1 framework làm cho Cocoa APIs sử dụng được trên iOS và OS X.

**Observables và Observers**

* Observable emit notification of change
* Observer subscribes to an Observable and gets notified when that Observable has changed.

1 observable có thể có nhiều observers. Khi observer thay đổi nó sẽ thông báo cho tất các observers của nó.

Có 2 loại observable:

* Cold: Observable chỉ bắt đầu thực hiện khi có observer subcribe. Dùng trong các loại: Async operations, HTTP Connections, TCP connections, streams.
* Hot: observavle luôn thực hiện bất kể không có observer subcribe nào. Dùng trong các loại: Variables, properties, constants, tap coordinates, mouse coordinates, UI control values, current time.

**DisposeBag**

DisposeBag được RxSwift cung cấp để tránh ARC và memory management. Khi denit() được gọi từ 1 object holds DisposeBag, mỗi disposable Observer sẽ tự động unsubscribed observable.

**Chú ý:**

* Không nên tạo 1 instance của dispose bag và chia sẻ nó cho các class khắc (bởi khi dispose bag deinit thì nó sẽ huỷ các subcriber đã đang ký với dispose bag đó)
* Nên thêm các subcriptions vào dispose bag, kể cả khi ta biết chắc là các sequences này sẽ terminate
* Luôn sử dụng .disposed(by: disposeBag) ngay cả khi nó không thực sự cần thiết

**Using**

didset của Array object trong swift chỉ được gọi khi toàn bộ array đó thay đổi còn nếu dữ liệu của 1 phần tử trong nó thay đổi thì nó sẽ k nhảy vào didset. Để theo dõi được sự thay đổi của dữ liệu của 1 object trong 1 mảng ta có thể sử dụng BehaviorRelay của RxSwift

let chocolates: BehaviorRelay<[Chocolate]> = BehaviorRelay(value: [])

* với method asObservable() của BehaviorRelay ta có thẻ theo dõi sự thay đổi của BehaviorRelay.
* Để thay đổi dữ liệu của BehaviorRelay phải thông qua menthod accept()
* Truy cập dữ liệu thông qua .value

Using in Table View

Khởi tạo table view data

* Observeable.just(\_:) để khai báo là dữ liệu sẽ không bị thay đổi nhưng nó vẫn có thể truy cập đến như là 1 Observable.
* Properties của just: có thể bind dữ liệu đến tableview
* Gọi .rx để truy cập đến extensions của RxCocoa
* Từ .rx ta gọi đến menthod Items(cellIdentifier: cellType:) để khai báo các thông tin về cell của Table View

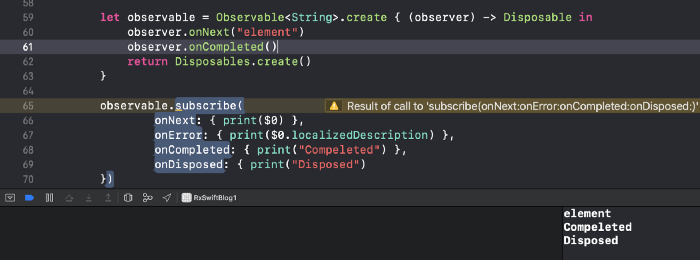
Khởi tại các action handler

* Gọi modelSelected(\_:) từ .rx nó sẽ trả về 1 observable

**Observable LifeCycle**

Observable có thể emit 3 loại events:

* .next: event mang theo dữ liệu mới nhất.
* .completed: hoàn thành. Sau khi emit event này sẽ không có event nào đc emit nữa.
* .error: lỗi. Sau khi emit event này sẽ không có event nào đc emit nữa.
* .dispose: để bỏ theo dõi. Observabl sẽ không emit event nữa. (.complete và .error sẽ tự gọi đến method này)



**Creating Observable**

* Obserable.create(): tạo ra obserable với các event do ta quy định.

let observable = Observable<String>.create { (observer) -> Disposable in  
DispatchQueue.global(qos: .default).async {  
Thread.sleep(forTimeInterval: 10)  
observer.onNext("Hello dummy 🐣")  
observer.onCompleted()  
}  
return Disposables.create()  
}

* Observable.deffered: giúp cho các observable như just, from, of,… thực thi code khi nó được subcribe. Nghĩa là mỗi lần subcribe nó sẽ tạo 1 observable mới

**var** observable1: Observable<Date> {

**return** Observable.deferred {

**let** date = Date()

print("obs 1: ", Thread.current)

**return** .just(date)

}

Tại mỗi thời điểm observable sẽ trả về kết quả khác nhau

* Observable.just(): tạo ra observable với chỉ 1 phần tử. Khi observable này được gọi nó sẽ bắn ra 1 event next và 1 event complete hoặc error
* let disposeBag = DisposeBag()
* let observable = Observable.just("this is element")
* observable.subscribe { element in
* print(element)
* }
* .disposed(by: disposeBag)

Kết quả:

* next(this is element)
* completed
* Observable.of(): tạo ra observable với 1 chuỗi các phẩn tử cho trước. Khi observable này được gọi nó sẽ bắn ra các event next có trứa các phẩn tử có trong chuỗi và 1 event complete hoặc error
* let disposeBag = DisposeBag()
* let observable = Observable.of(2, 5, 1, 3, 6, 4)
* observable.subscribe { element in
* print(element)
* }
* .disposed(by: disposeBag)

Kết quả:

* next(2)
* next(5)
* next(1)
* next(3)
* next(6)
* next(4)
* completed
* Observable.from(): giống với of nhưng dữ liệu của observable này là 1 mảng
* let disposeBag = DisposeBag()
* let observable = Observable.from([2, 5, 1, 3, 6, 4])
* observable.subscribe { element in
* print(element)
* }
* .disposed(by: disposeBag)

Kết quả:

* next(2)
* next(5)
* next(1)
* next(3)
* next(6)
* next(4)
* completed
* Observable.empty(): tạo ra observable với giá trị rỗng. Khi được gọi nó sẽ bắn ra sự kiện completed
* let disposeBag = DisposeBag()
* let observable = Observable<Void>.empty()
* observable.subscribe(
* onNext: { element in
* print(element)
* },
* onError: {error in
* print(error)
* },
* onCompleted: {
* print("onCompleted")
* },
* onDisposed: {
* print("onDisposed")
* })
* .disposed(by: disposeBag)

Kết quả:

* onCompleted
* onDisposed
* Observable.never(): không bắn ra bất cứ event nào
* let disposeBag = DisposeBag()
* let observable = Observable<Void>.never()
* observable.subscribe(
* onNext: { element in
* print(element)
* },
* onError: {error in
* print(error)
* },
* onCompleted: {
* print("onCompleted")
* })
* .disposed(by: disposeBag)

Kết quả:

* //không emit bất cứ event nào.
* Obvervable.range(start:, count:): tạo ra observable với 1 giá trị ban đầu và số lượng event tuần tự được tạo ra.
* let disposeBag = DisposeBag()
* let observable = Observable.range(start: 1, count: 10)
* observable.subscribe { element in
* print(element)
* }
* .disposed(by: disposeBag)

Kết quả:

* next(1)
* next(2)
* next(3)
* next(4)
* next(5)
* next(6)
* next(7)
* next(8)
* next(9)
* next(10)
* completed

Lưu ý:

* Khi sử dụng observable phải sử dụng cùng với DisposeBag nếu k có thể sẽ bị retain cycle
* Khi sử dụng create observable phải gửi event complete hoặc error để kết thúc observe
* Create và deffered đều giúp code bên trong observable thực thi lại thời điểm nó được subcribe và mỗi lần sẽ tạo ra observable mới.

**Operators**



Các loại operators:

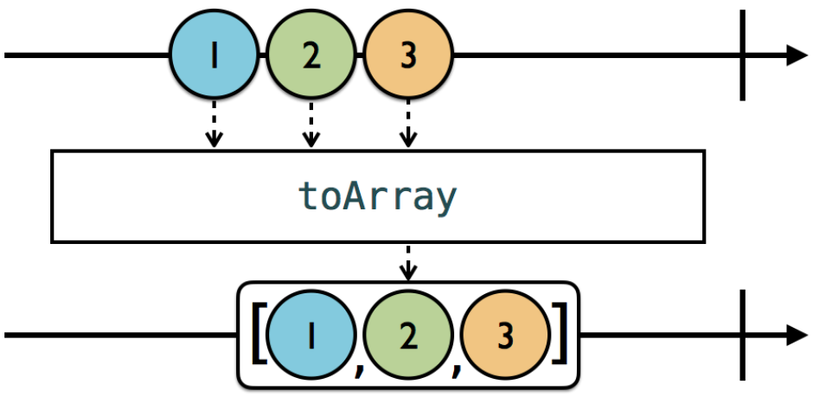
* **Creating Observables:** gồm các phép toán dùng để khởi tạo observable.
* **Transforming Observables:** gồm các phép toán dùng để biến đổi các giá trị được phát ra bởi observable.
* **Filtering Observables:** gồm các phép toán dùng để chọn lọc các giá trị được phát ra bởi observable.
* **Combining Observables:** gồm các phép toán dùng để nhóm nhiều observables lại thành một observable.
* **Error Handling Operators:** gồm các phép toán dùng để thông báo lỗi từ observable.
* **Observable Utility Operators:** gồm các ultility hữu ích.
* **Conditional and Boolean Operators**: gồm các phép toán điều kiện.
* **Mathematical and Aggregate Operators:** gồm các phép toán tính toán số lượng các giá trị được phát ra.
* **Connectable Observable Operators:** gồm các phép toán có khả năng connect được observable.
* **Operators to Convert Observables:** gồm các phép toán dùng để convert observable thành object hoặc data structure.

|  |  |  |
| --- | --- | --- |
| * map * flatMap * flatMapLatest * withLatestFrom | * filter * debounce * retry * skip | 1. catchError 2. take 3. concat 4. merge |

**Transforming Operators:**

1. **toArray()**

Dùng để chuyển đổi 1 observable gồm các phần tử riêng lẻ thành một mảng các phần tử. nó emit ra event next chứa mảng các phần tử kia.



// 1

let observable = Observable.of("A", "B", "C")

// 2

print("element of observable.toArray(): ")

observable.toArray()

.subscribe(onNext: {

print($0)

}).dispose()

// 3

print("element of observable: ")

observable

.subscribe(onNext: {

print($0)

}).dispose()

Kết quả

element of observable.toArray():

["A", "B", "C"]

element of observable:

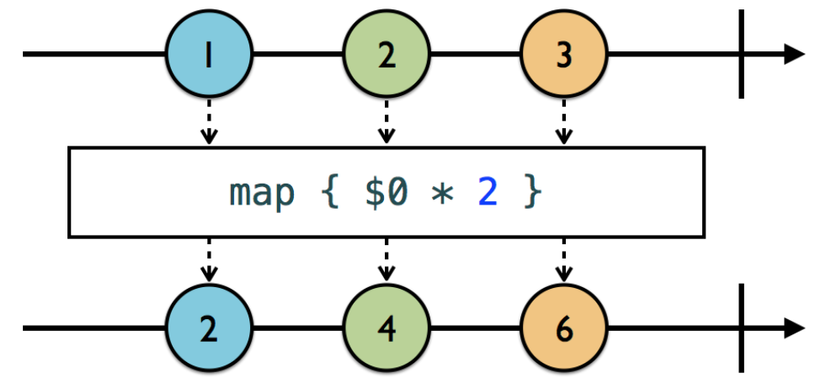
A

B

C

1. **Map**

Dùng để biến đổi dữ liệu (giá trị hoặc kiểu dữ liệu) được phát ra bởi Observable.



Observable.of(1, 2, 3, 4, 5, 6, 7, 8, 9)

.map {

$0 \* 10

}

.subscribe(onNext: {

print($0)

})

.dispose()

Kết quả in ra như sau:

10

20

30

40

50

60

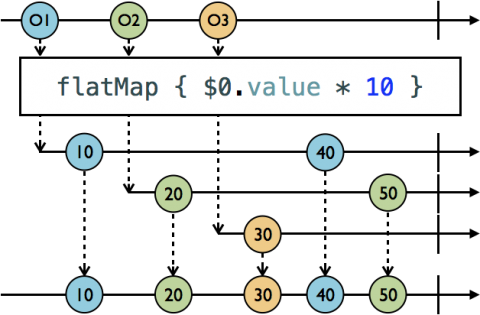
70

80

90

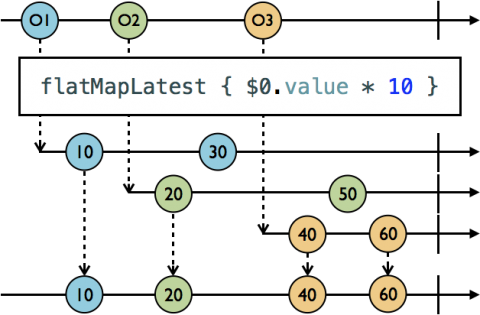
1. **flatMap()**

Projects each element of an observable sequence to an observable sequence and merges the resulting observable sequences into one observable sequence (bản thân các element cũng phải observe được)



1. **flatMapLatest**

Projects each element of an observable sequence into a new sequence of observable sequences and then transforms an observable sequence of observable sequences into an observable sequence producing values only from the most recent observable sequence



1. **Scan**

Giống với **reduce**. Nhưng sẽ emit sau mỗi lần scan

myButton.rx\_tap.scan(false) { lastState, newValue in

return !lastState

}

.subscribeNext {value in

print("tap: \(value)")

}

Kết quả

tap: true

tap: false

tap: true

tap: false

tap: true

False là gía trị mặc định

LastState là giá trị trước

newValue trong trường hợp này = {}

let numbers = [0, 1, 2 , 3, 4, 5, 6].toObservable()

numbers.scan([]) { lastSlice, newValue in

return Array(lastSlice + [newValue]).suffix(3)

}

.subscribeNext {value in

print("last 3: \(value)")

}

last 3: [0]

last 3: [0, 1]

last 3: [0, 1, 2]

last 3: [1, 2, 3]

last 3: [2, 3, 4]

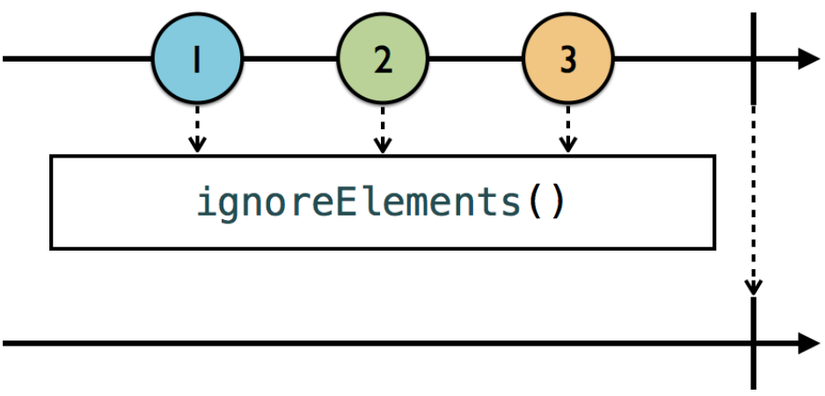
last 3: [3, 4, 5]

last 3: [4, 5, 6]

**Filtering Operators**

1. **ignoreElements**

Ignore các event .next và chỉ gửi các event .completed hoặc .error



Ví dụ:

let strikes = PublishSubject<String>()

let disposeBag = DisposeBag()

strikes

.ignoreElements()

.subscribe { \_ in

print("You're out!")

}

.disposed(by: disposeBag)

strikes.onNext("X")

strikes.onNext("X")

strikes.onNext("X")

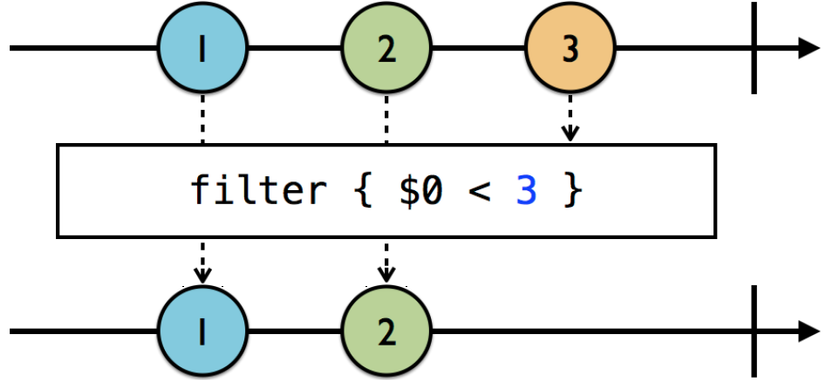
strikes.onCompleted()

Kết quả:

You're out!

1. **Filter**

Lọc các element của observable



Observable.of(1, 2, 3, 4, 5, 6)

.filter { integer in

integer % 2 == 0

}

.subscribe(onNext: {

print($0)

})

.dispose()

Kết quả:

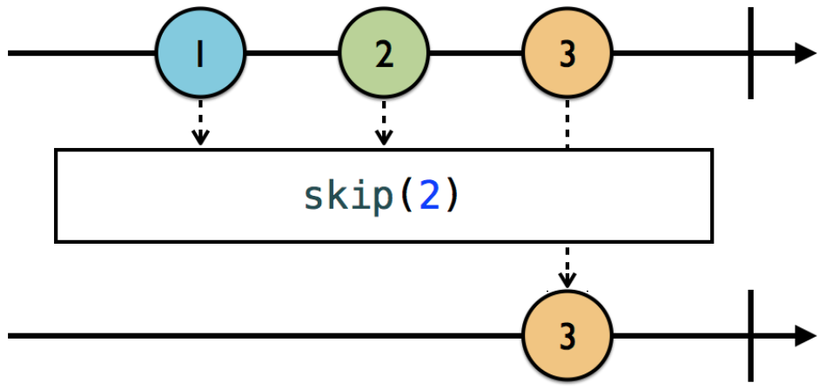
2

4

6

1. **Skip**

Skip(n) bỏ qua n phần tử đầu tiên



Observable.of("A", "B", "C", "D", "E", "F")

.skip(3)

.subscribe(onNext: {

print($0)

})

.dispose()

Kết quả:

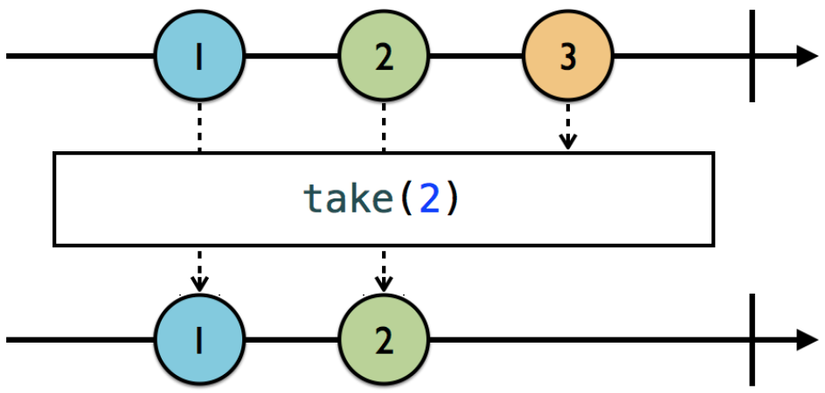
D

E

F

1. **Take**

Take giữ lại n phần tử đầu tiên và bỏ qua các phần tử phía sau:



Observable.of(1, 2, 3, 4, 5, 6, 7, 8)

.tcake(5)

.subscribe(onNext: {

print($0)

})

.dispose()

Kết quả:

1

2

3

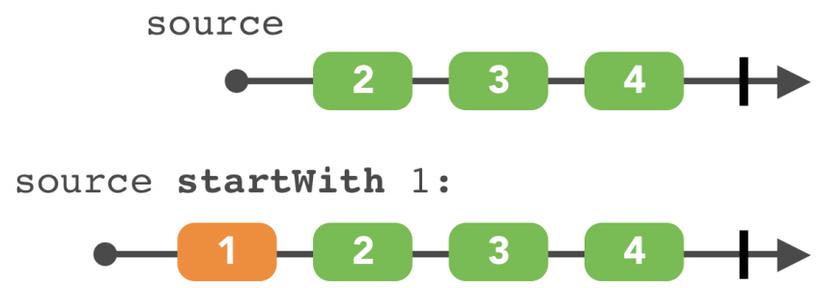
4

5

**Combining operators**

1. **startWith**

startWith nối vào đầu của observable một giá trị khởi tạo



let numbers = Observable.of(2, 3, 4)

let observable = numbers.startWith(1)

observable.subscribe(onNext: { value in

print(value)

})

Kết quả:

1

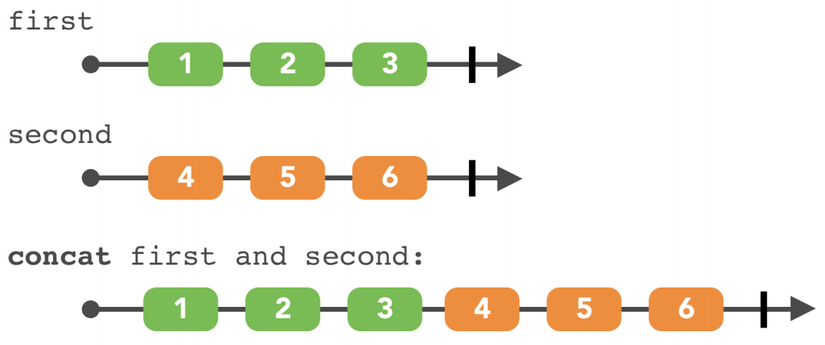
2

3

4

1. **Concat**

Giúp nối các observable cùng kiểu thành 1 observable



let first = Observable.of(1, 2, 3)

let second = Observable.of(4, 5, 6)

let observable = Observable.concat([first, second])

observable.subscribe(onNext: { value in

print(value)

}).dispose()

Kết quả:

1

2

3

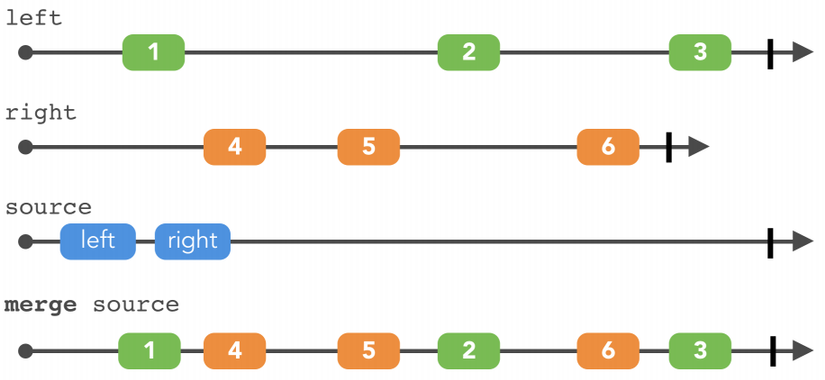
4

5

6

1. **Merge**

Kết hợp nhiều observable bằng cách gộp các emit lại thành một observable. Complete khi tất cả các observable con đều complete, khi có observable con emit error thì observable sẽ replay error và k gửi emit nữa.



let left = PublishSubject<Int>()

let right = PublishSubject<Int>()

let source = Observable.of(left, right)

let disposable = source.merge().subscribe { (event) in

print(event)

}

left.onNext(1)

right.onNext(3)

left.onNext(2)

right.onNext(4)

disposable.dispose()

Kết quả:

next(1)

next(3)

next(2)

next(4)

Thử thêm đoạn code sau vào trước dòng disposable.dispose():

left.onError(MyError.anError)

right.onNext(5)

Kết quả sẽ in ra như sau:

next(1)

next(3)

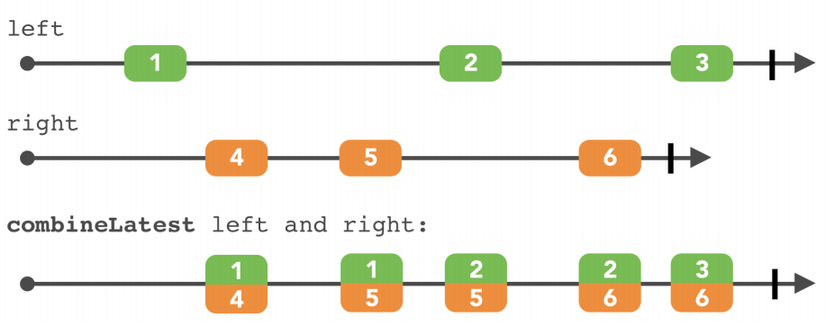
next(2)

next(4)

error(anError)

1. **CombineLastest**

Mỗi lần observable được combine emit 1 value, nó sẽ ghép value này với value mới nhất từ các observable khác được combine.



Ví dụ:

let left = PublishSubject<Int>()

let right = PublishSubject<Int>()

let disposable = Observable.combineLatest(left, right) { (left, right) -> String in

return "\(left) - \(right)"

}.subscribe(onNext: { (event) in

print(event)

})

left.onNext(1)

right.onNext(3)

left.onNext(2)

right.onNext(4)

right.onNext(5)

disposable.dispose()

Kết quả:

1 - 3

2 - 3

2 - 4

2 - 5

1. **withLatestFrom()**

A.withLatestFrom(B)

Sẽ lấy data mới nhất của observer B để emit

vd:

example(of: "withLatestFrom") {

**let** button = PublishSubject<Void>()

**let** textField = PublishSubject<String>()

**let** observable = button.withLatestFrom(textField)

**let** dispose1 = observable.subscribe(onNext: { value **in**

print("1)", value)

})

**let** observable2 = textField.sample(button)

**let** dispose2 = observable2.subscribe(onNext: { (value) **in**

print("2)", value)

})

textField.onNext("Par")

button.onNext(())

textField.onNext("Pari")

textField.onNext("Paris")

button.onNext(())

button.onNext(())

dispose1.dispose()

dispose2.dispose()

}

Out put:

1) Par

2) Par

1) Paris

2) Paris

1) Paris

Chú ý:

* So sánh flatmap vs map

**Share()**

Define streams that share resources among their subcribers. (tạo ra 1 execution trung cho các subcriber và dựa vào scope để quyết định xem có load lại execution hay không trong trường hợp subcriber nhảy từ 0 lên 1)

Share cung cấp 2 properties là replay và scope

* Replay: số lượng element sẽ được emit đến subcribers khi nó được subcriber (vd: số lượng element của observer là 5 mà replay của nó là 2 thì chỉ có 2 element được emit đến các subcriber mới)
* Scope: có 2 giá trị là .whileConnected và .forever:
* .whileConnected: khi số lượng subcriber nhảy từ 1 xuống 0 nó sẽ clean cache của shared stream và execute lại khi có subcriber mới.
* .forever: cache của stream sẽ không bị clear kể cả khi số lượng subcriber nhảy từ 1 xuống 0. (không khuyến khích sử dụng).

**Subject**

Subject vừa có thể là observable sequence vừa có thể là observer.

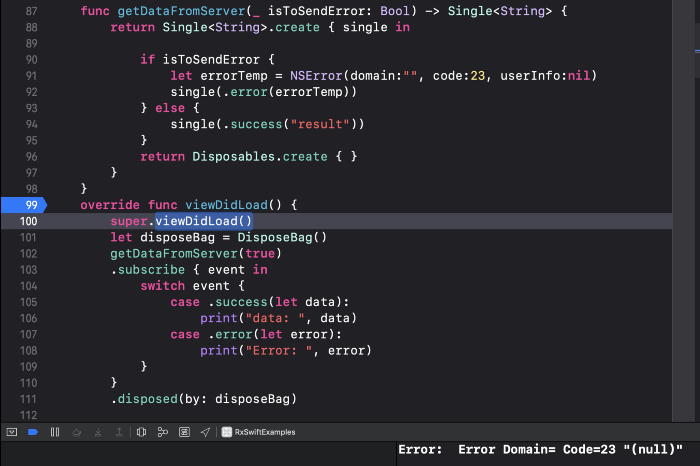
* **PublishSubject:** chỉ emit new next events cho subcribers, nghĩa là các element được thêm vào trước khi subcribe sẽ không được emit
* **BehaviorSubject:** được khởi tạo với starting value, nó sẽ gửi next event có chứa element mới nhất đến các subcribers. Lưu ý if any subject type is terminated, it will reemit its stop event, whether that is a completed or a new event to its subscribers.
* **ReplaySubject:** (ReplaySubject<Int>.create(bufferSize: 2)) cho phép khởi tạo buffersize cho observable, buffersize là số lượng element cuối cùng sẽ emit đến subcriber ( ví dụ thứ tự emit đến là 1,2,3 mà buffesize là 2 thì các subcriber sau đó chỉ nhận 2 giá trị là 2,3) (khi có emit error thì emit error không được tính là element)
* **PublishRelay:** wrap a PublishSubject. Sự khác biệt của nó vs PublishSubject là nó sẽ không bị terminated (dùng accept thay cho onNext)
* **BehaviorRelay**: wrap a BehaviorSubject. Sự khác biển của nó với BehaviorSubject là nó sẽ không bị terminated. Và ta có thẻ lấy giá trị hiện tại của nó thông qua biến value

**Traits**

**RxSwift**

1. **Single**

Là 1 biến thể của observable, nó chỉ e emit 1 element hoặc emit error. Nó sử dụng .success(value) để thay thế cho .next và completed

1 trường hợp điển hình để sử dụng Single là để thực hiện HTTP Request (chỉ return response hoặc error). 

1. **Completable**

Trả về completed hoặc trả về error

Sử dụng khi:

* Không quan tâm đến result của completion
* Updating a cache for instance
* UPDATE/PUT network call that resulted with success indication only.
* You have to perform operation when network connection is re-established.

1. **Maybe**

Là sự kết hợp của completable và single. Nó có thể emit .completed, .success(value), error

**RxCocoa**

1. **Driver**

Là trait đầy đủ nhất trong các traits.

* Driver luôn thực hiện trên main thread
* Không emit error nên thích hợp làm việc với UI
* Có shares side effects (share(replay: 1, scope: .whileConnected))

Có chức năng bắt error xem chi tiết ở dưới

1. **Signal**

Giống với Driver nhưng nó không emit event cuối cùng đến các subscription

* Driver luôn thực hiện trên main thread
* Không emit error nên thích hợp làm việc với UI
* Có shares side effects (share(scope: .whileConnected))
* Không replay elements

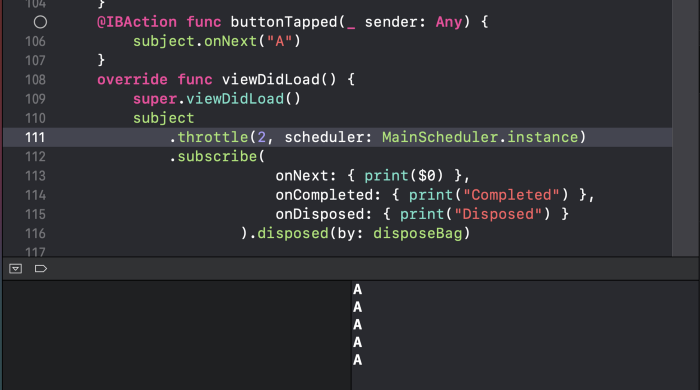
**Custom (do)**

‘do’ để register callback cái mà ReactiveX sẽ gọi khi có event diễn ra trong Observable

* onNext: emit cho mỗi element trong observable sequence
* onError: emit khi có error và sẽ terminate observer
* onComplete: emit khi observable completed
* onSubcribe: emit trước khi subcribe observable
* onSubcribed: emit sau khi subcribe observable
* onDispose: emit sau khi subcription bị disposed.

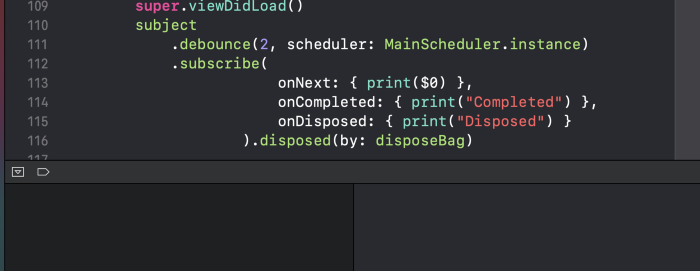
**Throttle**

Đảm bảo sẽ không có element nào được emit sau ít nhất 1 khoảng thời gian nào đó.



**Debounce**

Ignore element từ observable sequence nếu element đó được emit liên tục trong 1 khoảng thời gian được quy định, thì chỉ event được emit cuối cùng được thực thi.



Hữu dụng

* Khi muốn xác định khi nào người dùng ngừng nhập text

**Scheduler**

* **MainScheduler**: main thread
* **CurrentThreadScheduler**: thread hiện tại
* **SerialDispatchQueueScheduler**: đảm bảo rằng nếu concurrrent dispatch queue được truyền vào thì nó sẽ được chuyển đổi về serial queue
* **ConcurrentDispatchQueueScheduler**:chạy trên concurrent dispatch queue
* **OperationQueueScheduler**: thực hiện NSOperationQueue, thường dùng cho background chạy concurrent, và chúng ta set được số lượng concurrent: maxConcurrentOperationCount.

**observeOn:** chuyển thread (queue) khi observer nhận được notification

observable  
 .**observeOn**(MainScheduler.instance)  
 .subscribe(onNext: { (data) in]  
 **//Update UI on Main Thread**  
 })  
.addDisposableTo(disposeBag)

**subcribeOn:** chuyển thread (queue) khi bắt đầu xử lý observable

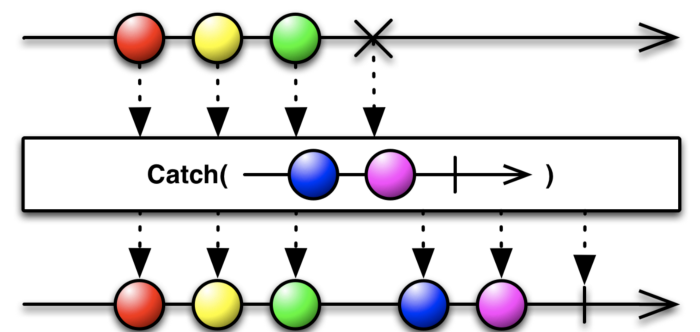
**// Scheduler**  
let bgScheduler = ConcurrentDispatchQueueScheduler(qos: .background)sequence  
 .**subcribeOn**(bgScheduler)  
 .map { n in  
 print("This is performed on the background scheduler")  
 }  
 .observeOn(MainScheduler.instance)  
 .map { n in  
 print("This is performed on the main scheduler")  
 }

Note tìm hiểu:

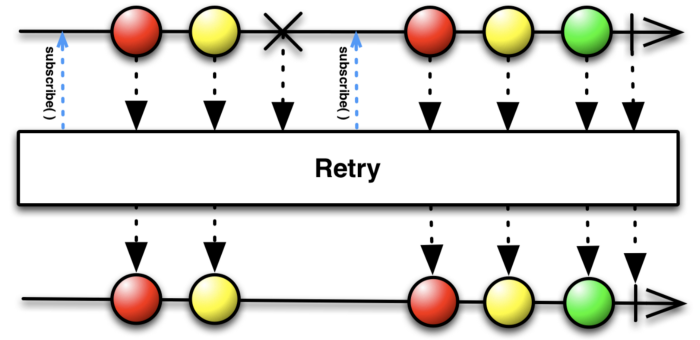
* Check gửi request: vd: gõ @huy nhanh quá thì chỉ gửi @huy thôi (throttle)
* Tìm example về Rx

**Error Handler**

1. Khi ta subcribe 1 observable mà nó emit error thì các subcriber sẽ bị huỷ. Để tránh việc đó ta sẽ emit ra các **Observable<Event<()>>** thông qua **.materialize()**
2. Khi sử dụng **.****catchErrorJustReturn(())** thì nếu xảy ra lỗi thì nó sẽ emit 1 giá trị và sau đó sẽ emit complete và các subcriber vẫn sẽ bị huỷ (có nghĩa là bắt lỗi được 1 lần)
3. Sử dụng **asDriver(onErrorJustReturn:)** cũng tương tự như .**catchErrorJustReturn(())**. Nhưng nó sẽ trả về 1 driver
4. CatchError: bắt các lỗi từ onError và tiếp tục sequence, khi có error catchError sẽ trả về 1 observable mới



1. **Retry():** mặc khi bắt được onError sẽ execute lại đến khi được thì thôi hoặc thực hiện 1 số phần nhất định (vd: retry(3) thực hiện lại 3 lần nếu vẫn lỗi thì thôi, nếu lần 1 k có lỗi thì thôi k lập lại nữa).



1. **RetryWhen**

Bắt emit onError từ source observable và trả về 1 observable(b1). Thời gian và số lần retry sẽ do b1 đó quyết định (khi b1 emit thì sẽ tiến hành retry, khi b1 complete thì source observable complete, khi b1 emit error thì souce observable emit error).

**let** maxAttempts = 9

**let** retryHandler: (Observable<Error>) -> Observable<Int> = { e **in**

**return** e.enumerated().flatMap({ (arg) -> Observable<Int> **in**

**let** (attempt, error) = arg

**if** attempt >= maxAttempts - 1 {

**return** Observable.error(error)

} **else** **if** **let** casted = error **as**? ApiController.ApiError, casted == .invalidKey {

**return** ApiController.shared.apiKey.filter {$0 != ""}.map { **\_** **in** **return** 1 }

} **else** **if** (error **as** NSError).code == -1009 {

**return** RxReachability.shared.status.filter {

**return** $0 == .online

}.map { **\_** **in**

**return** 1 }

}

print("logger: == retrying after \(attempt + 1) seconds ==")

**return** Observable<Int>.timer(Double(attempt + 1), scheduler: MainScheduler.instance)

.take(1)

})

}

1. **ThrowError**

**enum** ApiError: Error {  
 case noInternetConnection  
 case serverError  
}return session.rx.response(request: request).map() { response, data in  
 if 200 ..< 300 ~= response.statusCode {  
 return JSON(data: data)  
 } else if 400 ..< 500 ~= response.statusCode {  
 throw ApiError.cityNotFound   
 //return Observable.error(ApiError.cityNotFound)} else {  
 throw ApiError.serverError  
 //return Observable.error(ApiError.serverError)  
 }  
}//Check error  
**onError**: { [weak self] e in  
  
}

**RxCocoa**

Chú ý:

1. Textview vs textfield

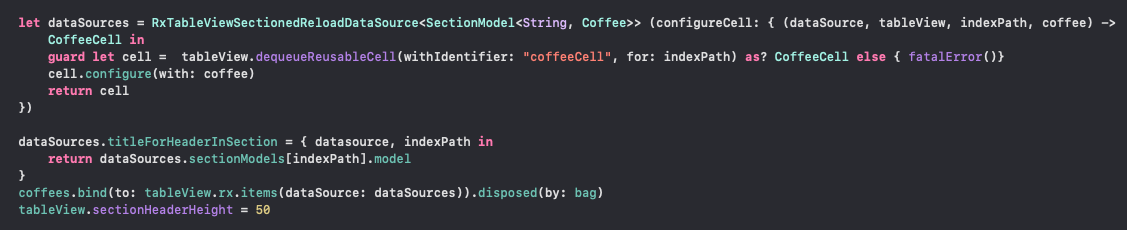
Cách emit event rx.text của textview và textfield khác nhau (rx.text emit khi có 1 trong các controlEvent emit đối với textfield và là didchange,… đói với textview)

* Đối với textfield khi ta set textfield.text thì rx.text sẽ không emit event còn đối với textview thì có emit

1. Tableview

* Muốn sử dụng multable section với rx ta phải cài thêm thư viện RxDataSources

Observable sẽ emit ra các sectionmodel, ta sẽ bind các session đó đến datasource của table view

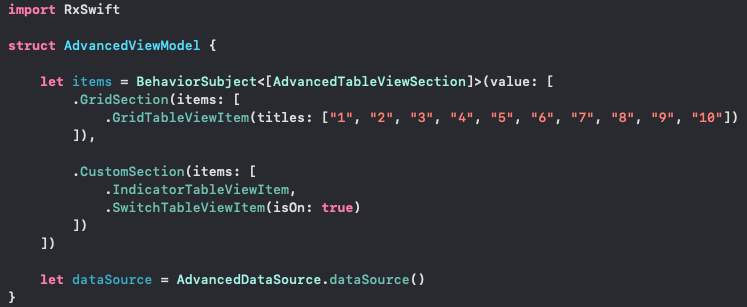


* Ta có thể custom sectionmodel thông qua protocol SectionModelType

Custom SectionModel



Bắn observable để bắn dữ liệu cho tableview



Khi sử dụng **RxTableViewSectionReloadDataSource** thì khi observable bắn dữ liệu của tableview emit thì nó sẽ gọi hàm reload() -> Không tạo animation

* Nếu muốn tạo các animation khi thay đổi sanh sách các cell trong tableview ta phải sử dụng **RxTableViewSectionAnimatedDataSource** (Nó sẽ tự động insert, delete, update cell dựa trên observable)

Các section phải là **AnimatableSectionModel** hoặc custom qua **AnimatableSectionModelType**

Các item của **AnimatableSectionModel** phải implement **IdentifiableType** (của framework **Differentiator**) biến **identity** của các item không được phép giống nhau.

Nhớ gán biến **canEditRowAtIndexPath** để cho phép thực hiện các action của cell

Khi muốn thay đổi danh sách cell ta chỉ cần emit event mới từ observable



* Việc sử dụng search bar để filter dữ liệu của multable section (sẽ rất phức tạp nếu filter trực tiếp từ SectionModel) tốt hơn là nên filter trực tiếp đối tượng và gán vào SectionModel

Chú ý:

1. Bản chất bind(onNext) là subcribe( onNext)
2. Các thuộc tính get-only của rx ui có thể dùng bind để thay đổi
3. Nếu trong các cell của table view hoặc collection view có sử dụng dispose bag thì phải tạo dispose bag ở trong cell và tạo mới ở prepare for reuse

Câu hỏi:

1. tại sao dùng RxSwift

<https://medium.com/@VincentVuVNG/d%C3%B9ng-th%C6%B0-vi%E1%BB%87n-rxswift-%C4%91%E1%BB%83-c%E1%BA%A3i-ti%E1%BA%BFn-hi%E1%BB%87u-su%E1%BA%A5t-cho-d%E1%BB%B1-%C3%A1n-ios-c%E1%BB%A7a-b%E1%BA%A1n-ph%E1%BA%A7n-1-bbf71e2d0e52>

1. tại sao subject lại concat được với observable

do bản thân subject cũng là observable

1. So sánh flatmap vs map

Link:

Part 1:

* <https://www.raywenderlich.com/1228891-getting-started-with-rxswift-and-rxcocoa>

Creating observable:

* <http://adamborek.com/creating-observable-create-just-deferred/>
* <https://viblo.asia/p/rxswift-cac-cach-khoi-tao-observable-trong-rxswift-aWj53pkPK6m>

Operator

* <https://www.raywenderlich.com/682-rxswift-transforming-operators>
* <https://www.raywenderlich.com/677-rxswift-transforming-operators-in-practice#toc-anchor-012>
* <https://viblo.asia/p/rxswift-operator-trong-rxswift-gGJ59vp9KX2>

Share

* <https://medium.com/gett-engineering/rxswift-share-ing-is-caring-341557714a2d>

Subject

* <https://medium.com/@navdeepsingh_2336/rxswift-foundation-part-2-observables-and-subjects-b143f983e757#:~:text=Subject%20is%20a%20special%20type,emitted%20to%20the%20subject%20subscribers>
* <https://medium.com/@aliakhtar_16369/rxswift-part-2-f96b9490f814>

Traits

* <https://medium.com/@aliakhtar_16369/rxswift-part-2-f96b9490f814>

Custom (do), Throttle, Debound:

* <https://medium.com/@aliakhtar_16369/rxswift-part-3-cad06fb1adf8>

rxCocoa

* tableview: <https://medium.com/better-programming/how-to-implement-uitableview-and-uicollectionview-with-rxdatasources-1afcd68729bf>