


Time Based Operators

I. Buffering operators

1. Basic with Timer

- Các bước để tạo 1 observable với timer 1 cách thủ công.

+ Setup cho timer:

```
let elementsPerSecond = 1
let maxElement = 5
let replayedElement = 1 |  Initializa
let replayDelay: TimeInterval = 3
```

+ Tạo 1 custom observable với toán tử
create()

```
let observable = Observable<Int>.create { (observer) -> Disposable in
    var value = 1
```

+ Tạo 1 timer trên main queue

```
let source = DispatchSource.makeTimerSource(queue: .main)
```

+ Đặt closure cho vòng lặp timer

```
source.setEventHandler {
    if value <= maxElement {
        observer.onNext(value)
        value += 1
    } else {
        observer.onCompleted()
    }
}
```

- + Tạo schedule để lên kế hoạch emit dữ liệu

```
source.schedule(  
    deadline: .now(),  
    repeating: 1.0 / Double(elementsPerSecond),  
    leeway: .nanoseconds(0)  
)
```

- + Bắt đầu chạy timer:

```
source.resume()
```

- + Dừng timer khi mà dispose observable

```
return Disposables.create {  
    source.suspend()  
}  
}
```

- + Thực hiện chạy observable:

```
DispatchQueue.main.asyncAfter(deadline: .now()) {  
    observable  
        .subscribe(onNext: { (value) in  
            print("🔴 : \(value)")  
        }, onCompleted: {  
            print("🔴 Completed")  
        }, onDisposed: {  
            print("🔴 Disposed")  
        })  
        .disposed(by: bag)  
}
```

- Thực hiện subscribe:

```
DispatchQueue.main.asyncAfter(deadline: .now()) {
    observable
        .subscribe(onNext: { (value) in
            print("🔴 : \(value)")
        }, onCompleted: {
            print("🔴 Completed")
        }, onDisposed: {
            print("🔴 Disposed")
        })
        .disposed(by: bag)
}
```

● Output

```
🔴 : 1
🔴 : 2
🔴 : 3
🔴 : 4
🔴 : 5
🔴 Completed
🔴 Disposed
```

2. Replaying past elements

- `replay(_)`: Tạo 1 observable với buffer để lưu lại giá trị trước đó mà observable source đã emit. Kích thước của buffer tùy theo tham số truyền vào
- `replayAll`: Giống như toán tử `replay` nhưng không giới hạn kích thước của buffer
- Toán tử này thuộc kiểu `Connectable Observable` nên ta phải gọi `connect()` để nối

observable này với observable source.

- Tạo 1 observable mà sẽ replay lại observable ở ví dụ 1.

```
let replaySource = observable.replayAll()
// let replaySource = observable.replay(replayedElement)

//replay observable
DispatchQueue.main.asyncAfter(deadline: .now() + replayDelay) {
    replaySource
        .subscribe(onNext: { (value) in
            printValue("🟦 : \(value)")
        }, onCompleted: {
            print("🟦 Completed")
        }, onDisposed: {
            print("🟦 Disposed")
        })
        .disposed(by: bag)
}
```

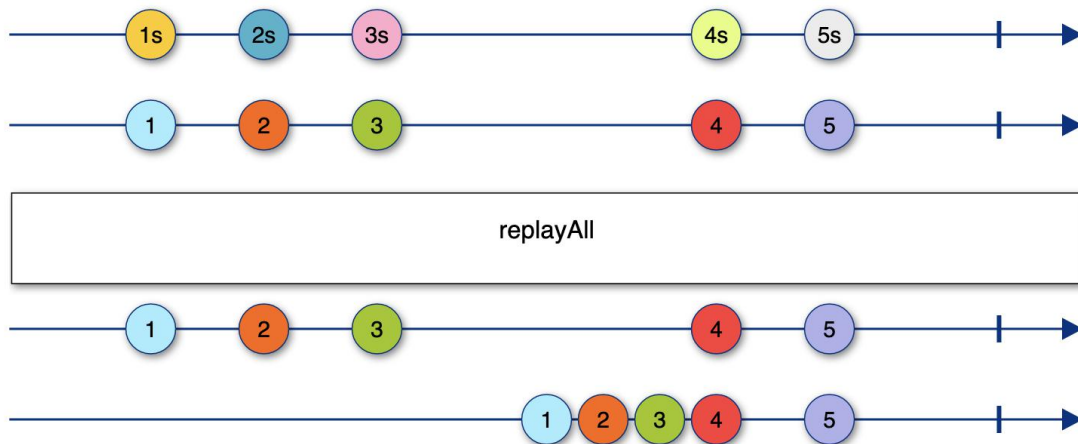
- Nối observable với observable gốc

```
replaySource.connect()
```

- Output:

```
🟥 : 1 --- at 02.9280
🟥 : 2 --- at 03.9230
🟥 : 3 --- at 04.9230
🟦 : 1 --- at 05.8230
🟦 : 2 --- at 05.8250
🟦 : 3 --- at 05.8260
🟦 : 4 --- at 05.8270
🟥 : 4 --- at 05.9240
🟦 : 5 --- at 06.8250
🟥 : 5 --- at 06.9230
```

- Sơ đồ marble



3. Controlled buffering

- buffer : toán tử dùng cho công việc tạo và quản lý bộ đệm, có các tham số:
 - + timeSpan: thời gian mà bộ đệm sẽ hết hạn.
 - + count: kích thước của bộ đệm.
 - + scheduler: luồng thực thi.
- Setup:

```
let bufferTimeSpan = RxTimeInterval.milliseconds(4_000)
let bufferMaxCount = 4

let source = PublishSubject<String>()

//observing the observable
source
    .subscribe(onNext: { value in
        printValue("🔴: \(value)")
    })
    .disposed(by: bag)
```

- Tạo bộ đệm cho observable:

```

source
    //the buffer is an array of String
    .buffer(
        timeSpan: bufferTimeSpan,
        count: bufferMaxCount,
        scheduler: MainScheduler.instance
    )
    .map { $0.count }
    .subscribe(onNext: { value in
        printValue("🟡: \$(value)")
    })
    .disposed(by: bag)

```

- Trình tự chạy:

```

DispatchQueue.main.asyncAfter(deadline: .now() + 4) {
    source.onNext("A")
    source.onNext("B")
    source.onNext("C")
    source.onNext("D")
}

```

- Output:

```

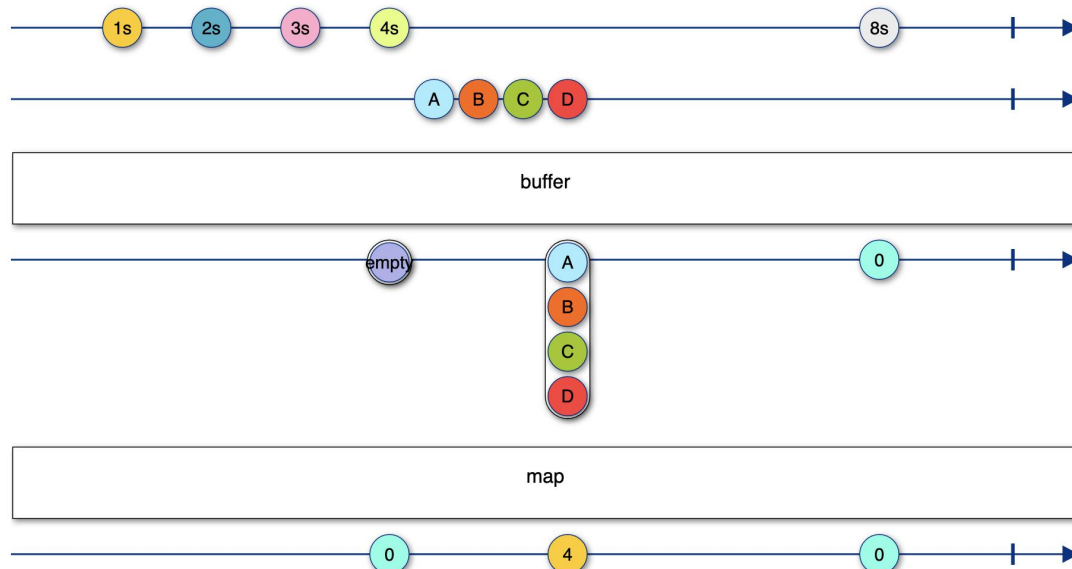
🟡: 0 --- at 40.6880
🟢: A --- at 40.6940
🟢: B --- at 40.6970
🟢: C --- at 40.6980
🟢: D --- at 40.6990
🟡: 4 --- at 40.6990
🟡: 0 --- at 44.7010
🟡: 0 --- at 48.7040
🟡: 0 --- at 52.7060

```

- + Ở giây thứ 4 bộ đếm được gọi và trả về count hiện tại

- + Kế tiếp observable bắt đầu phát 4 element.
- + Bộ đệm giữ lại các element vào trong buffer.
- + Bộ đệm đạt đến mức tối đa nên giải phóng và phát giá trị cho subscriber là 1 mảng chứa 4 phần tử.

● Sơ đồ marble:



4. Window

- Tương tự như buffer nhưng thay vì kiểu dữ liệu là 1 mảng thì toán tử window trả về 1 observable.
- Sửa lại:

```

source
//the buffer is now an observable sequence of [String]
.window(timeSpan: bufferTimeSpan, count: bufferMaxCount, scheduler: MainScheduler.instance)
//turning the buffer into an array of string
.flatMap { (observable) -> Observable<[String]> in
    observable
        .scan(into: []) { (array, element) in
            array.append(element)
        }
}
.subscribe(onNext: { (value) in
    printValue("🟡: \"${value}\"")
})
.disposed(by: bag)

```

- Trình tự chạy:

```

DispatchQueue.main.asyncAfter(deadline: .now() + 4) {
    source.onNext("1")
    source.onNext("2")
    source.onNext("3")
    source.onNext("4")
    source.onNext("5")
}

```

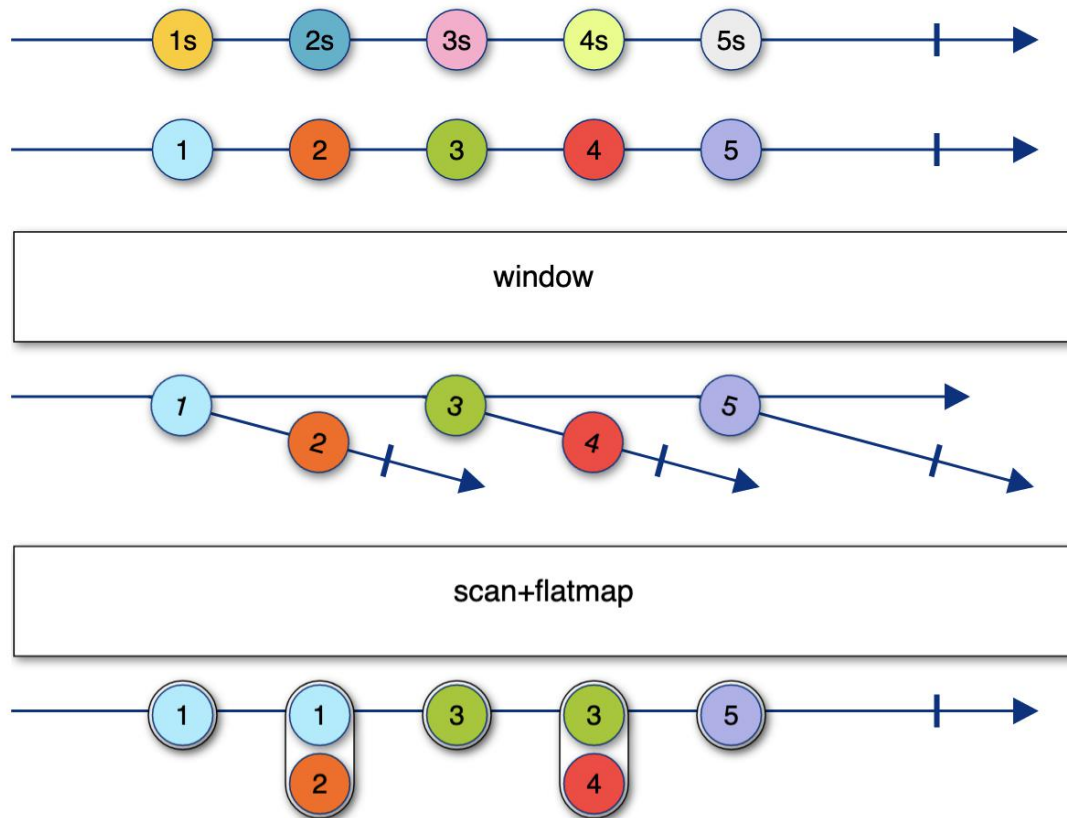
- Output:

```

🔴: 1 --- at 58.4680
🟡: ["1"] --- at 58.4740
🔴: 2 --- at 58.4770
🟡: ["1", "2"] --- at 58.4780
🔴: 3 --- at 58.4790
🟡: ["3"] --- at 58.4790
🔴: 4 --- at 58.4800
🟡: ["3", "4"] --- at 58.4800
🔴: 5 --- at 58.4810
🟡: ["5"] --- at 58.4810

```

- Sơ đồ marble:



II. Time-shifting operators

1. Delayed subscriptions

- **delaySubscription:** Dùng để trì hoãn việc subscription cho các observer theo thời gian mà truyền vào cho tham số.
- Các giá trị mà observable emit trong khi mà đang trì hoãn thì sẽ bị bỏ qua
- Setup:

```

func delayedSubscription() -> DispatchSourceTimer {
    let source = PublishSubject<String>()

    source
        .delaySubscription(.seconds(2), scheduler: MainScheduler.instance)
        .subscribe(onNext: { value in
            printValue("🔴: \(value)")
        })
        .disposed(by: bag)

    var count = 1
    let temp = DispatchSource.timer(interval: 1.0, queue: .main) {
        source.onNext("\(count)")
        count += 1
    }

    return temp
}

```

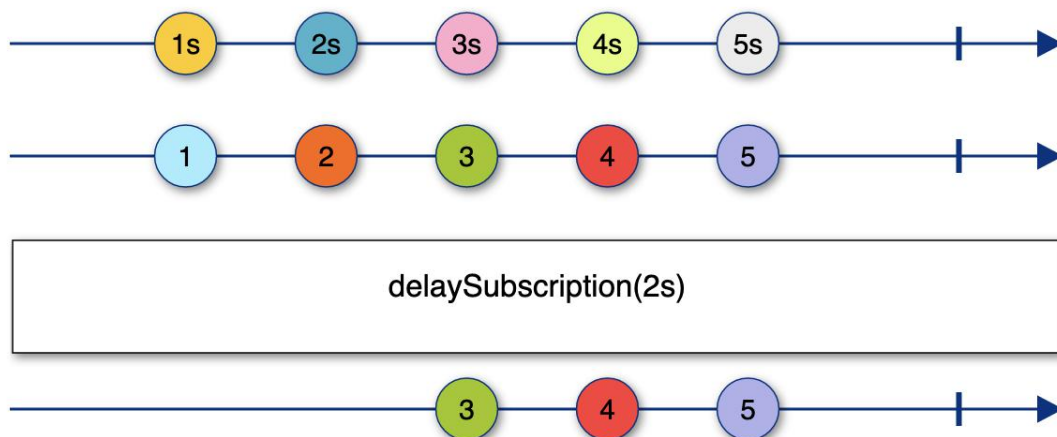
- Output:

```

🔴: 3 --- at 55.7980
🔴: 4 --- at 56.7990
🔴: 5 --- at 57.7990
🔴: 6 --- at 58.7990
🔴: 7 --- at 59.7980
🔴: 8 --- at 00.7990

```

- Sơ đồ marble



2. Delayed elements

- `delay(_:)` : Giống như `delaySubscription` nhưng trì hoãn việc đưa element cho các subscriber.

- Sửa lại:

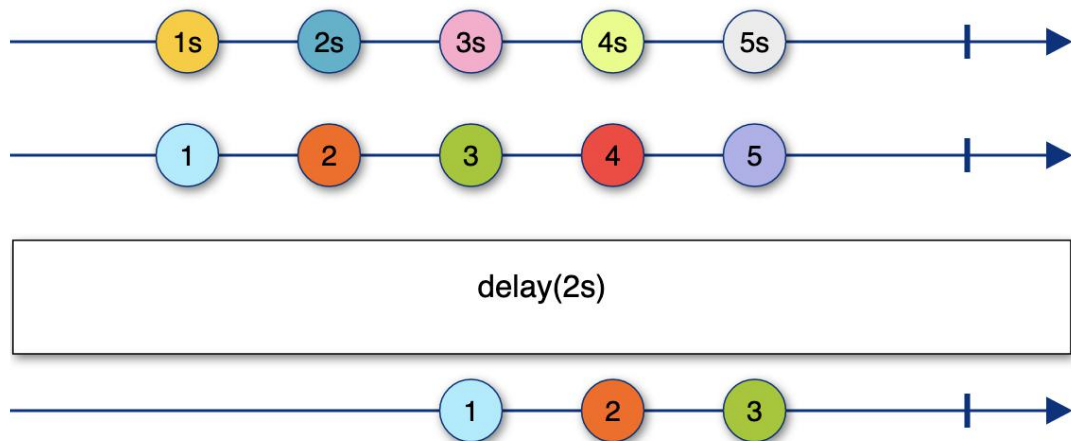
```
let source = PublishSubject<String>()
source
    .delay(.seconds(2), scheduler: MainScheduler.instance)
    .subscribe(onNext: { value in
        printValue("🔴: \(value)")
    })
    .disposed(by: bag)

var count = 1
var temp = DispatchSource.timer(interval: 1, queue: .main) {
    printValue("emit: \(count)")
    source.onNext("\(count)")
    count += 1
}
```

- Output:

```
emit: 1 --- at 24.1910
emit: 2 --- at 25.1420
emit: 3 --- at 26.1420
🔴: 1 --- at 26.1960
emit: 4 --- at 27.1420
🔴: 2 --- at 27.2000
emit: 5 --- at 28.1420
🔴: 3 --- at 28.2030
emit: 6 --- at 29.1410
🔴: 4 --- at 29.2050
emit: 7 --- at 30.1420
🔴: 5 --- at 30.2080
```

- Sơ đồ marble:



III. Timer operators

1. interval(_:scheduler:)

- Hoạt động giống như toán tử timer của swift mà sẽ thực hiện 1 dòng for lặp mỗi khoảng thời gian nhất định.
- Setup làm lại replay:

```
let source = Observable<Int>.interval(RxTimeInterval.seconds(1), scheduler: MainScheduler.instance)
let replay = source.replay(2)

DispatchQueue.main.asyncAfter(deadline: .now()) {
    source
        .subscribe(onNext: { value in
            printValue("🔴: \(value)")
        }, onCompleted: {
            print("🔴 Completed")
        }, onDisposed: {
            print("🔴 Disposed")
        })
        .disposed(by: bag)
}

DispatchQueue.main.asyncAfter(deadline: .now() + 2) {
    replay
        .subscribe(onNext: { value in
            printValue("🔵: \(value)")
        }, onCompleted: {
            print("🔵 Completed")
        }, onDisposed: {
            print("🔵 Disposed")
        })
        .disposed(by: bag)
}

replay.connect()
```

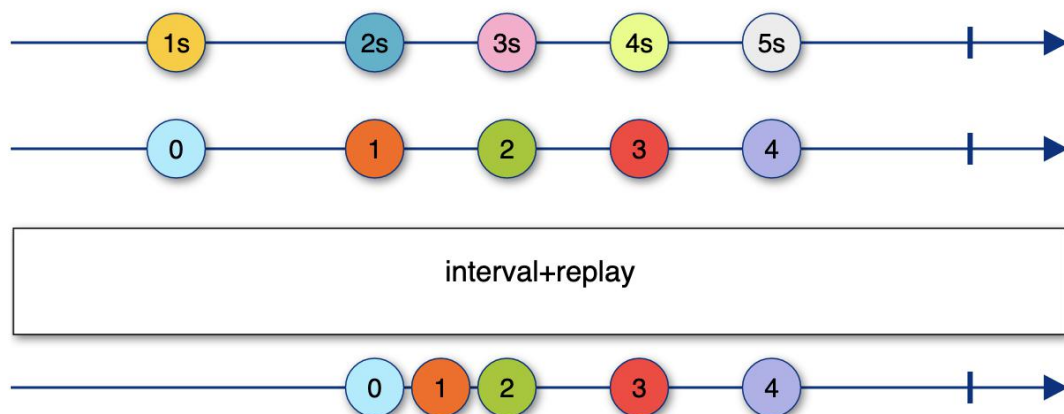
- Output:

```

● : 0 --- at 57.6890
● : 0 --- at 58.6130
● : 1 --- at 58.6150
● : 1 --- at 58.6880
● : 2 --- at 59.6140
● : 2 --- at 59.6880
● : 3 --- at 00.6140
● : 3 --- at 00.6880
● : 4 --- at 01.6140

```

- Sơ đồ marble



2. timer

- Hoạt động giống như interval nhưng chỉ emit 1 lần xong rồi kết thúc.
- Thời gian emit dựa theo tham số truyền vào.
- Ví dụ:

```

let source = Observable<Int>.timer(RxTimeInterval.seconds(3), scheduler: MainScheduler.instance)

DispatchQueue.main.asyncAfter(deadline: .now()) {
    source
        //delay by 2 more second.
        .flatMap{ _ in
            | source.delay(RxTimeInterval.seconds(2), scheduler: MainScheduler.instance)
        }
        .subscribe(onNext: { value in
            printValue("🔴: \(value)")
        }, onCompleted: {
            print("🔴 Completed")
        }, onDisposed: {
            print("🔴 Disposed")
        })
        .disposed(by: bag)
}

```

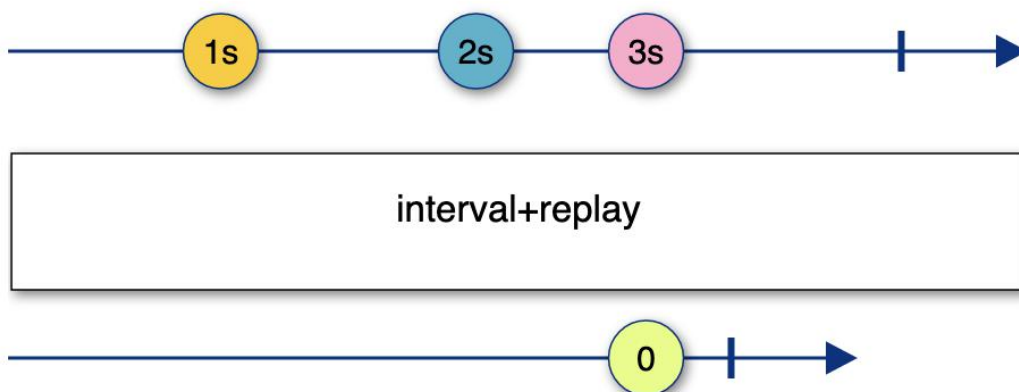
- Output:

```

🔴: 0 --- at 40.5110
🔴 Completed
🔴 Disposed

```

- Sơ đồ marble:



3. Timeout

- Toán tử này dùng để kết thúc 1 observable nếu mà observable không emit trong 1 khoảng thời gian nhất định.

- Toán tử này kết thúc observable bằng cách throw `TimeoutError`.
- Setup:

```
let source = PublishSubject<Int>()
source
  .timeout(RxTimeInterval.seconds(5), scheduler: MainScheduler.instance)
  .subscribe(onNext: { value in
    printValue("🔴: \(value)")
  }, onCompleted: {
    print("🔴 Completed")
  }, onDisposed: {
    print("🔴 Disposed")
  })
  .disposed(by: bag)

DispatchQueue.main.asyncAfter(deadline: .now() + 1) {
  source.onNext(1)
}
DispatchQueue.main.asyncAfter(deadline: .now() + 2) {
  source.onNext(2)
}
DispatchQueue.main.asyncAfter(deadline: .now() + 6) {
  source.onNext(3)
}
```

- Output:

```
🔴: 1 --- at 08.6710
🔴: 2 --- at 09.7710
🔴: 3 --- at 14.1710
Unhandled error happened: Sequence timeout.
🔴 Disposed
```

- Sơ đồ marble:

