

RXSwift - Time-based Operator

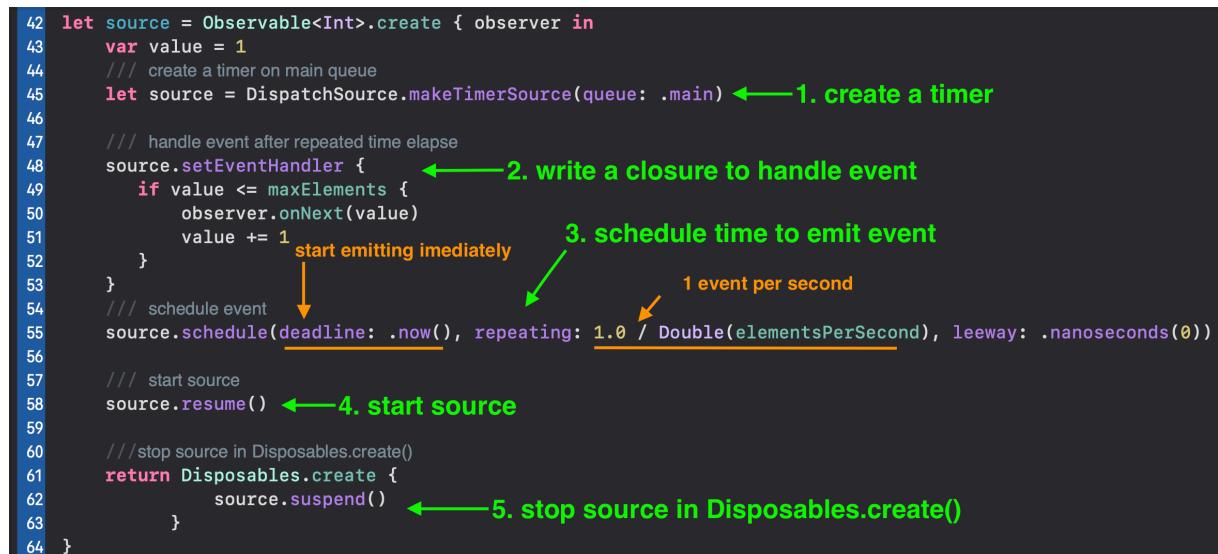
✓ Buffering operators

- deal with buffering
- either replay past elements to new subscribers, or buffer them and deliver them in bursts.
- they allow you to control how and when past and new elements get delivered.
- Basic steps:

EX: create an observable can emit element by time: 1 event per 1 second

```
let elementsPerSecond = 1
let maxElements = 5
let replayedElements = 1
let replayDelay: TimeInterval = 3
```

```
42 let source = Observable<Int>.create { observer in
43     var value = 1
44     /// create a timer on main queue
45     let source = DispatchSource.makeTimerSource(queue: .main) ← 1. create a timer
46
47     /// handle event after repeated time elapse
48     source.setEventHandler {
49         if value <= maxElements {
50             observer.onNext(value)
51             value += 1
52         }
53     }
54     /// schedule event
55     source.schedule(deadline: .now(), repeating: 1.0 / Double(elementsPerSecond), leeway: .nanoseconds(0)) ← 3. schedule time to emit event
56
57     /// start source
58     source.resume() ← 4. start source
59
60     /// stop source in Disposables.create()
61     return Disposables.create {
62         source.suspend() ← 5. stop source in Disposables.create()
63     }
64 }
```



```
DispatchQueue.main.async { //Subscribe in main  
    observable  
        .subscribe(onNext: { value in  
            print("🔵 : ", value)  
        }, onCompleted: {  
            print("🔵 Completed")  
        }, onDisposed: {  
            print("🔵 Disposed")  
        })  
        .disposed(by: bag)  
  
}
```

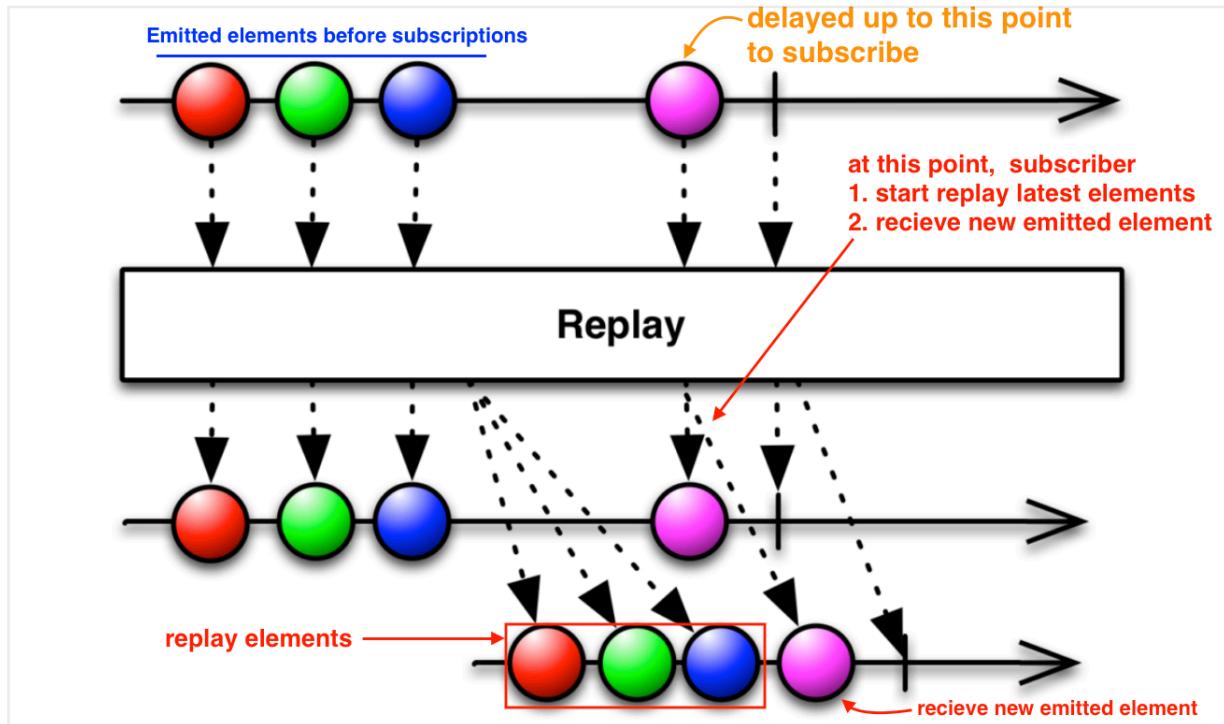
>>>>Basic Timer Example:

```
🔴 : 1 --- at 43.4760  
🔴 : 2 --- at 44.4750  
🔴 : 3 --- at 45.4760  
🔴 : 4 --- at 46.4760  
🔴 : 5 --- at 47.4750
```

Elements are printed out
one by one

** NOTE: We don't code like above as it may contains unexpected error

- Replaying past elements



- replay:
 - o we need to provide it a specific number of buffer element to replay it to subscriber
 - o Any saved elements are the latest elements that Observable has emitted

EX:

replay the latest element before subscriber start subscribing

```

88
89 let source1 = Observable<Int>.create { observer in
90     var value = 1
91     /// create a timer on main queue
92     let source = DispatchSource.makeTimerSource(queue: .main)
93
94     /// handle event after repeated time elapse
95     source.setEventHandler {
96         if value <= maxElements {
97             observer.onNext(value)
98             value += 1
99         }
100    }start emitting immediately
101   /// schedule event
102   source.schedule(deadline: .now(), repeating: 1.0 / Double(elementsPerSecond), leeway:
103                   .nanoseconds(0))
104
105   /// start source
106   source.resume()
107
108   ///stop source in Disposables.create()
109   return Disposables.create {
110       source.suspend()
111   }

```

```

113 // Create a Observable Replay
114 let replaySource = source1.replay(1)
115
116 //Start subscribe after 3 seconds delay
117 DispatchQueue.main.asyncAfter(deadline: .now() + 3) {
118     replaySourcewait for 3 sec before start
119     .subscribe(onNext: { value in
120         printValue("● : \(value)")
121     }, onCompleted: {
122         print("● Completed")
123     }, onDisposed: {
124         print("● Disposed")
125     })
126     .disposed(by: disposeBag)
127 }
128 //** NOTE: because it belongs to ConnectableObservable, therefore, to execute, we must connect
129 // to observable by calling .connect()
130 replaySource.connect()

```

>>>>Replaying past elements Example:

**assume red dot used for counting second

● : 1 --- at 34.5970	← start emitting element #1, #2, #3
● : 2 --- at 35.5960	
● : 3 --- at 36.5960	← start subscribing
● : 3 --- at 37.5220	← replay 1 latest element that had just been emitted ('3') before subscription
● : 4 --- at 37.5320	← receive new emitted element from now on
● : 4 --- at 37.5960	
● : 5 --- at 38.5330	← receive new emitted element from now on
● : 5 --- at 38.5960	
● : 6 --- at 39.5320	← receive new emitted element from now on
● : 6 --- at 39.5960	
● : 7 --- at 40.5320	← receive new emitted element from now on

- replayAll:
 - replay all emitted elements before subscriber start subscribing

```

88
89 let source1 = Observable<Int>.create { observer in
90     var value = 1
91     /// create a timer on main queue
92     let source = DispatchSource.makeTimerSource(queue: .main)
93
94     /// handle event after repeated time elapse
95     source.setEventHandler {
96         if value <= maxElements {
97             observer.onNext(value)
98             value += 1
99         }
100    }
101    /// schedule event
102    source.schedule(deadline: .now()+1, repeating: 1.0 /
103                    Double(elementsPerSecond), leeway: .nanoseconds(0))
104
105    /// start source
106    source.resume()
107
108    /// stop source in Disposables.create()
109    return Disposables.create {
110        source.suspend()
111    }

```

wait for 1 sec before emitting events

```

134 let replaySource2 = source1.replayAll()
135 DispatchQueue.main.asyncAfter(deadline: .now() + 3) {
136     replaySource2
137         .subscribe(onNext: { value in
138             printValue("🟡 : \(value)")
139         }, onCompleted: {
140             print("🟡 Completed")
141         }, onDisposed: {
142             print("🟡 Disposed")
143         })
144         .disposed(by: disposeBag)
145 }
146 replaySource2.connect()

```

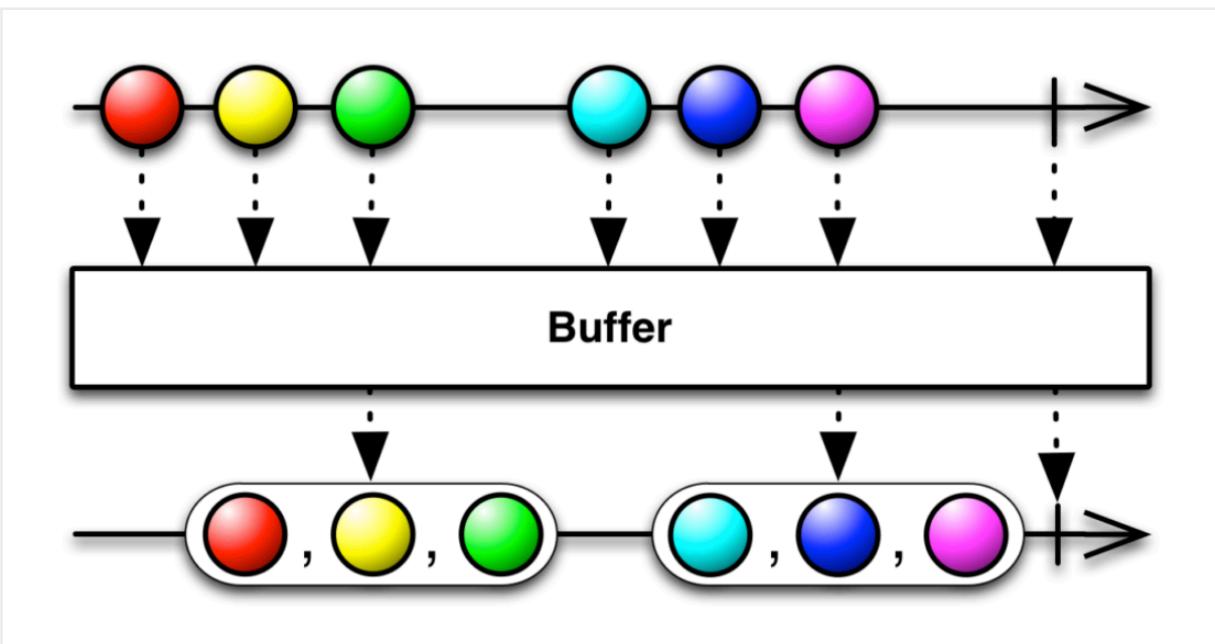
start subscribing
after 3 sec

>>>>Replaying past elements Example:

Red dot is used to count second

● : 1 --- at 24.6850	
● : 2 --- at 25.6820	← after 1s, event #1 & #2 have been emitted
● : 3 --- at 26.6820	← After 3s, start subscribing
● : 1 --- at 27.6170	← Replay ALL elements (2 elements) that had been
● : 2 --- at 27.6180	emitted before subscription
● : 3 --- at 27.6190	← receive new emitted element
● : 4 --- at 27.6820	
● : 4 --- at 28.6190	← receive new emitted element
● : 5 --- at 28.6840	
● : 5 --- at 29.6170	← receive new emitted element
● : 6 --- at 29.6820	
● : 6 --- at 30.6170	← receive new emitted element

- Controlled buffering



EX:

```

156 let bufferTimeSpan = RxTimeInterval.milliseconds(4000) // 4 seconds time span
157 let bufferMaxCount = 2 max elements we can emitted to subscriber durrring the time span time span for emitting event to subscriber
158
159 let source2 = PublishSubject<String>()
160
161 source2
162   //1. manage buffering
163   // + timeSpan : buffering spanning time
164   // + count : maximum number of elements that a buffer can store
165   // + scheduler : select which thread to execute
166   .buffer(timeSpan: bufferTimeSpan, count: bufferMaxCount, scheduler: MainScheduler.instance)
167   //2. AFTER timespan it will emits an event (independent with source2), so need to be modify a bit to read current number elements in the buffer
168   .map{ $0.count } ←buffer returns a Array containing emitted event's values. In this case is [String] type
169   .subscribe(onNext: { (value) in
170     printValue("璇 : \(value)")
171   })
172   .disposed(by: disposeBag)
173
174 DispatchQueue.main.asyncAfter(deadline: .now() + 5) {
175   source2.onNext("A")
176   source2.onNext("B") ←emits 5 elements after 5s
177   source2.onNext("C")
178   source2.onNext("D")
179   source2.onNext("E")
180 }
181

```

Giải thích code:

- Sẽ có khung thời gian **4s để nhận event hoặc đóng không làm gì cả.**
- Khi đến **thời gian 4s** cho phép Source **nhận event**, Source sẽ emit events cho subscriber.
 - o Lưu ý là số event được emit cũng sẽ bị giới hạn bởi **bufferMaxCount**.
 - o Source không thể emit nhiều hơn số lượng cho phép
 - o Nếu có event được phát ra:
 - ◆ nếu số event \leq buffer_max_count :
 - emit tất cả event rồi delay tiếp
 - ◆ nếu số event $>$ buffer_max_count :
 - chỉ emit đúng số event cho phép (max_count). Còn như event dư lại sẽ được emit đợt sau
 - Sau khi emit xong thì delay tiếp
 - o Nếu không có event được phát ra:
 - emit 1 event rỗng, không chứa dữ liệu
- Khi đến **thời gian 4s đóng**:
 - o Sẽ không nhận phát bất cứ event nào trong thời gian này

red dot is used to count time

```

: 1 --- at 38.3900
: 2 --- at 39.3860
: 3 --- at 40.3850
: 4 --- at 41.3850
: 0 --- at 42.2880 ← there's no emitted event at this point
: 5 --- at 42.3850 ← after 5s, we emits 5 events
: 2 --- at 43.2890 ← Souce take max 2 events for each time as buffer_max_count = 2
: 2 --- at 43.2910 ← => Events are splitted to 2, 2, 1
: 6 --- at 43.3860 ← => Event [ 'A' , 'B' ] are emitted together => Source print 2 as 2 emitted events
: 7 --- at 44.3850 ← => Event [ 'C' , 'D' ] are emitted together => Source print 2 as 2 emitted events
: 8 --- at 45.3860 ← **CANNOT emit 5th event 'E' because it exceed the time span limit
: 9 --- at 46.3860 ← => have to wait for next time to emit
: 1 --- at 47.2930 ← Source emits 'E'
: 10 --- at 47.3860
: 11 --- at 48.3860
: 12 --- at 49.3860
: 13 --- at 50.3850
: 0 --- at 51.2950 ← No more emitted event
: 14 --- at 51.3860 ← => Source emit an event with '0' indicating no event are emitted
: 15 --- at 52.3850 ← Keep going like that.....

```

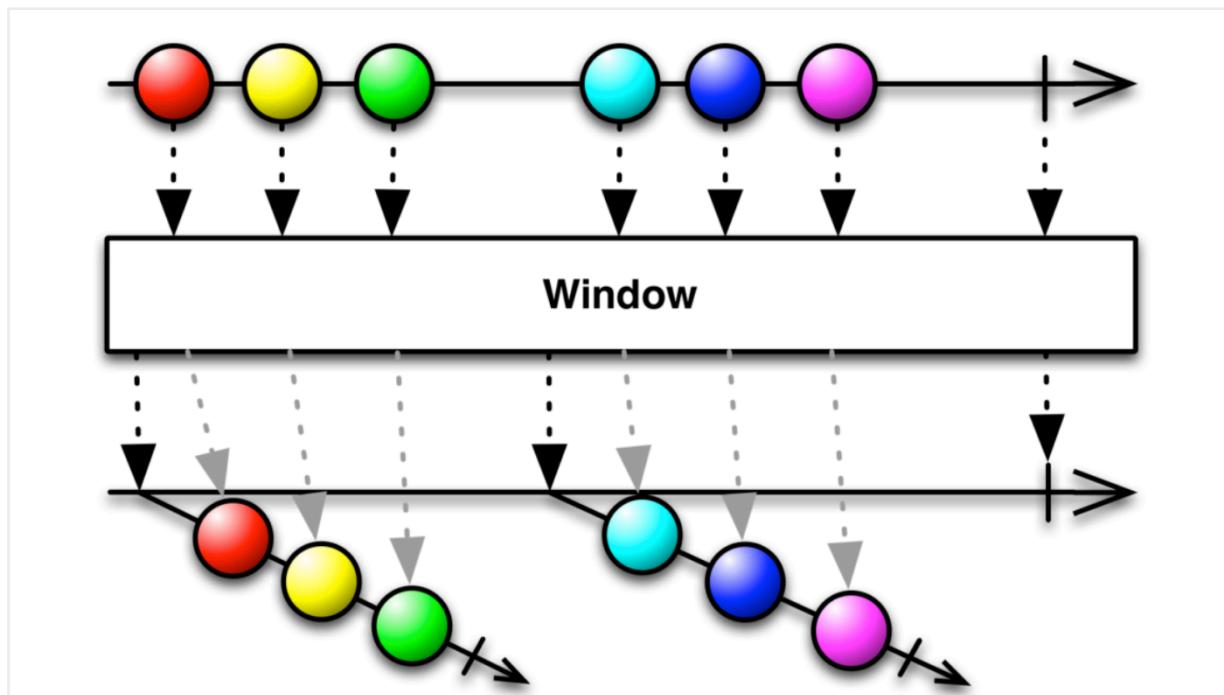
time span for 4s

time span for 4s

time span for 4s

time span for 4s

- window:
 - similar to *buffer*
 - BUT in *window*, the buffer data type is a observable, not an Array type like *buffer*



```

207 source2
208     //source.window : Observable<String>
209     .window(timeSpan: bufferTimeSpan, count: bufferMaxCount, scheduler: MainScheduler.instance)
210     //convert Observable<String> to Array String with flat map
211     .flatMap({ obs -> Observable<[String]> in
212         // because it's an Observable, we scan all its emitted element into 1 single value
213         obs.scan(into: []) { $0.append($1) }
214     })
215     .subscribe(onNext: { (value) in
216         printValue("⌚ \(value)")
217     })
218     .disposed(by: disposeBag)
219

```

>>>>Controlled buffering Example:

● : 1 --- at 57.2250
● : 2 --- at 58.2230
● : 3 --- at 59.2230
⌚ ["A"] --- at 00.2070
⌚ ["A", "B"] --- at 00.2090
⌚ ["C"] --- at 00.2110
⌚ ["C", "D"] --- at 00.2120
⌚ ["E"] --- at 00.2140
⌚ ["E", "F"] --- at 00.2160
⌚ ["G"] --- at 00.2180
● : 4 --- at 00.2220
● : 5 --- at 01.2230
● : 6 --- at 02.2230
● : 7 --- at 03.2220
● : 8 --- at 04.2220

✓ Time-shifting operators

- Delayed subscription:
 - delaying register a subscription to certain amount of time.
 - Of course, any emitted element before the subscription will not be received

```

239
240 let source3 = PublishSubject<String>()
241
242 source3
243     // delay subscription for 2 second at start
244     .delaySubscription(RxTimeInterval.seconds(2), scheduler: MainScheduler.instance)
245     .subscribe(onNext: { value in
246         printValue("🤡 : \(value)")
247     })
248     .disposed(by: disposeBag)
249
250 var count = 1
251 // for every 1s, emit event once.
252 /// **NOTE:
253 /// + the 1st element we received is 3
254 /// + if we don't do :timer.suspend(), it will never end
255 var timer = DispatchSource.timer(interval: 1.0, queue: .main) {
256     source3.onNext("\(count)")
257     count += 1
258 }

```

>>>>Delayed subscription Example:

 : 1 --- at 40.5420  : 2 --- at 41.5380  : 3 --- at 42.4400  : 3 --- at 42.5370  : 4 --- at 43.4400  : 4 --- at 43.5370  : 5 --- at 44.4410  : 5 --- at 44.5380  : 6 --- at 45.4400  : 6 --- at 45.5370	 ← delay for 2s before start subscribing
---	--

- Delayed elements
 - delay receiving elements



delay



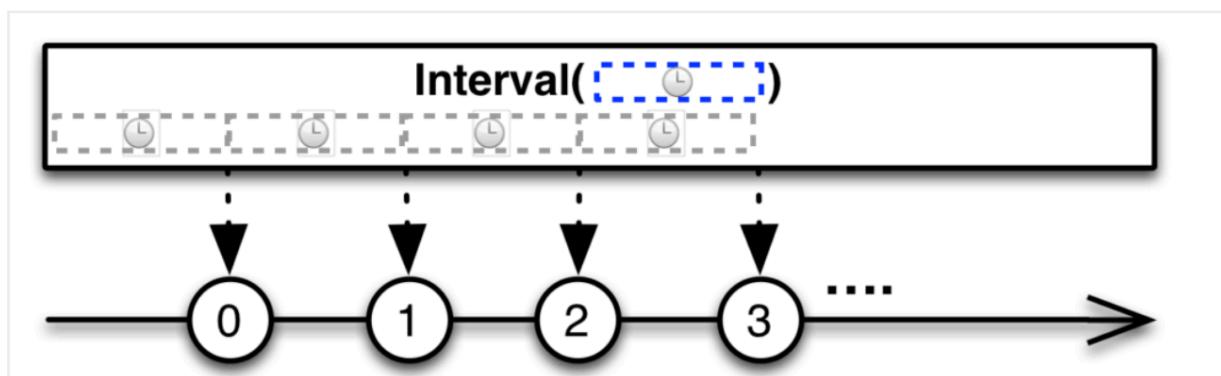
```
268
269 let source4 = PublishSubject<String>()
270
271 source4
272     // delay elements for 2 second
273     .delay(RxTimeInterval.seconds(2), scheduler: MainScheduler.instance)
274     .subscribe(onNext: { value in
275         printValue("⌚ receive: \(value)")
276     })
277     .disposed(by: disposeBag)
278
279 var count4 = 1
280 var timer4 = DispatchSource.timer(interval: 1.0, queue: .main) {
281     printValue("emit: \(count4)")
282     source4.onNext("\(count4)")
283     count4 += 1
284 }
```

```
>>>>Delayed elements Example:
```

```
emit: 1 --- at 48.3160
emit: 2 --- at 49.2290
emit: 3 --- at 50.2280
🤖 recieve: 1 --- at 50.3210
emit: 4 --- at 51.2290
🤖 recieve: 2 --- at 51.3230
emit: 5 --- at 52.2290
🤖 recieve: 3 --- at 52.3260
emit: 6 --- at 53.2290
🤖 recieve: 4 --- at 53.3280
emit: 7 --- at 54.2280
🤖 recieve: 5 --- at 54.3290
emit: 8 --- at 55.2280
🤖 recieve: 6 --- at 55.3310
emit: 9 --- at 56.2280
🤖 recieve: 7 --- at 56.3320
```

✓ Timer operators

- Interval
 - Another way to do DispatchSource.timer()



```

290 // for every 1s, emits 1 element
291 let source5 = Observable<Int>.interval(RxTimeInterval.seconds(1), scheduler: MainScheduler.instance)
292 let replay5 = source5.replay(2)
293
294 DispatchQueue.main.asyncAfter(deadline: .now()) {
295     source5
296         .subscribe(
297             onNext: { value in
298                 printValue("😈 : \(value)")
299             }, onCompleted: {
300                 print("😈 Completed")
301             }, onDisposed: {
302                 print("😈 Disposed")
303             })
304         .disposed(by: disposeBag)
305 }
306 }
307 replay5.connect()
308

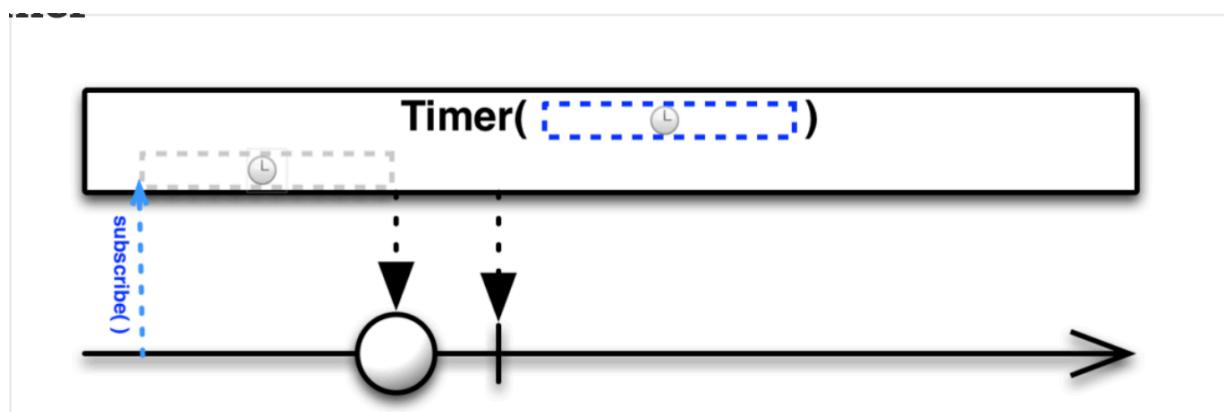
```

```

😈 : 0 --- at 43.4900
😈 : 1 --- at 44.4900
😈 : 2 --- at 45.4890
😈 : 3 --- at 46.4890
😈 : 4 --- at 47.4890

```

- Timer:
 - similar to interval
 - set a timer, and when time comes, emit the value



```
316 let source6 = Observable<Int>.timer(RxTimeInterval.seconds(3), scheduler: MainScheduler.instance)
317
318 /// **NOTE: it only emits 1 element then terminate**
319 source6
320     /// To make it emits more than 1, we use flatMap()
321     /// Here in flatMap(), we continue delaying for 2s more before subscriber can receive events
322     .flatMap({ _ in
323         source6.delay(RxTimeInterval.seconds(2), scheduler: MainScheduler.instance)
324     })
325     .subscribe { value in
326         printValue("👽 : \(value)")
327     } onError: { error in
328         printValue("👽 error")
329     } onCompleted: {
330         printValue("👽 completed")
331     } onDisposed: {
333         printValue("👽 disposed")
334     }
335     .disposed(by: disposeBag)
336
```

```
👽 : 0 --- at 34.7200
👽 completed --- at 34.7240
👽 disposed --- at 34.7240
```

- Timeout:
 - Similar to request API, if there are no emits left, it will automatically terminate

```
342 let source7 = PublishSubject<Int>()
343
344 source7
345     /// ** time set for 5s. After 5s, terminate if doesn't receive anymore emits**
346     .timeout(RxTimeInterval.seconds(5), scheduler: MainScheduler.instance)
347     .subscribe { value in
348         printValue("🐶 : \(value)")
349     } onError: { error in
350         printValue("🐶 error")
351     } onCompleted: {
352         printValue("🐶 completed")
353     } onDisposed: {
354         printValue("🐶 disposed")
355     }
356     .disposed(by: disposeBag)
357
358 DispatchQueue.main.asyncAfter(deadline: .now() + 1) {
359     source7.onNext(1)
360 }
361
362 DispatchQueue.main.asyncAfter(deadline: .now() + 2) {
363     source7.onNext(2)
364 }
365
366 DispatchQueue.main.asyncAfter(deadline: .now() + 6) {
367     source7.onNext(3)
368 }
369
370 //**NOTE: this exceed 5s rule, "4" emits after 6s, therefore, subscriber won't receive this as source has already terminated**
371 DispatchQueue.main.asyncAfter(deadline: .now() + 12) {
372     source7.onNext(4)
373 }
374 }
```

```
🐶 : 1 --- at 07.8260
🐶 : 2 --- at 08.8260
🐶 : 3 --- at 12.8260
🐶 error --- at 17.8280
🐶 disposed --- at 17.8290
```