让事件流起来

RxJS一小时入门

什么RxJS

- RxJS是通过使用可观察序列来编写异步和基于事件的程序的库
- 核心类型 - Observable
- 周边类型 - Observer, Scheduler, Subject
- 操作符——受Array的map、filter、reduce、every……等方法启发

什么RxJS

- 把RxJS当成事件的Lodash

将Observer模式与Iterator模式和函数式编程相结合，以满足管理事件序列的理想方式的需要

定时输出菲波那契数列项

```
let fibonacci$ = Rx.Observable.interval(400).take(10)
  .scan(x => [x[1], x[0] + x[1]], [0, 1])
  .pluck('0')

fibonacci$.subscribe({
  next: function observer (x) {
    console.log(x)
  }
})
```

RxJS基本概念

- **Observable: 表示未来值或事件的可调用集合的思想**
- Observer: 回调函数的集合，这些回调函数知道如何监听Observable传递的值
- Subscription: 表示Observable的执行，主要用于取消回调
- Subject: 等同于EventEmitter，向多个Observer广播数值或者事件的唯一方法
- Operator: 纯函数，用于实现函数式编程风格，使用map、filter、concat、flatMap……等操作来处理集合
- Scheduler: 控制并发的集中式调试程序，允许我们当运算在setTimeout或requestAnimationFrame……等发生时进行协调

Observable剖析

- 创建Observable
  - Rx.Observable.create
  - 创建操作符（静态方法）——of, from, interval, ...
- Subscribing to Observables
  - 类似于调用一个函数，提供回调处理传递的数据
- Executing the Observable
  - 仅当一个Observable被订阅时才运行的惰性计算
  - next*(error|complete)?
- Disposing Observables
  - unsubscribe

Observable

推送多个数值集合的惰性计算

	单值	多值
拉	函数	迭代器
推	Promise	**Observable**

同步或异步输出任意多值

```
let promise = new Promise((resolve) => {
  setTimeout(() => {
   resolve('foo from Promise')
  }, 1000)
})
let source$ = require('rxjs').Observable.create((observer) => {
  observer.next('foo')
  setTimeout(() => observer.next('bar from Observable'), 2000)
  setTimeout(() => observer.next('bar from Observable again'), 4000)
})
promise.then(console.log)
let subscription = source$.subscribe(console.log)
```

惰性计算

```
let source$ = require('rxjs').Observable.create((observer) => {
  console.log('Observable started')
  observer.next('foo')
  setTimeout(() => observer.next('bar from Observable'), 2000)
  setTimeout(() => observer.next('bar from Observable again'), 4000)
})
setTimeout(() => source$.subscribe(console.log), 3000)
```

Observer

Observable推送的数值的消费者

包含一系列回调函数

```
var observer = {

  next: x => console.log('Observer got a next value: ' + x),

  error: err => console.error('Observer got an error: ' + err),

  complete: () => console.log('Observer got a complete notification')

}

source$.subscribe(observer)
```

Subscription

表示Observable的执行的一次性资源

```
let subscription = source$.subscribe(console.log)

// Later:

subscription.unsubscribe()
```

Subject

一种特殊的Observable

允许将数值组播给多个Observer

Subject既是Observable，又是Observer

```
var subject = new Rx.Subject();
subject.subscribe({
  next: (v) => console.log('observerA: ' + v)
});
subject.subscribe({
  next: (v) => console.log('observerB: ' + v)
});
subject.next(1);
subject.next(2);
```

```
var subject = new Rx.Subject();
subject.subscribe({
  next: (v) => console.log('observerA: ' + v)
});
subject.subscribe({
  next: (v) => console.log('observerB: ' + v)
});
var observable = Rx.Observable.from([1, 2, 3]);
observable.subscribe(subject);
```

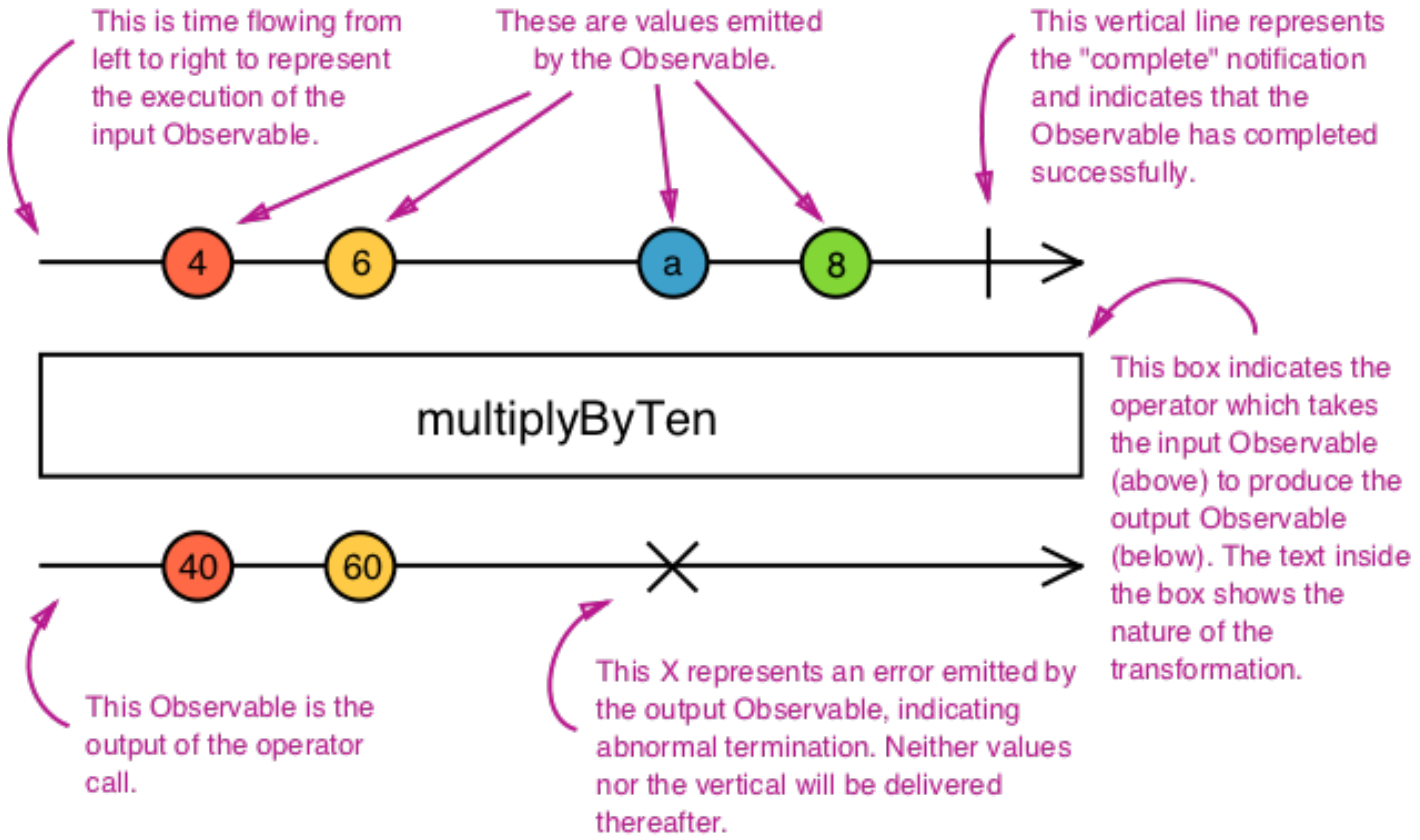
Subject

- 组播Observable，使用multicast operator生成一个拥有connect方法的ConnectableObservable
- 引用计数
- BehaviorSubject - replays one, only before completion
- ReplaySubject - replays many, before or after completion
- AsyncSubject - replays one, only if completed

Operator

返回一个新Observable的Observable方法

- 静态方法
  - create、empty、of、from、fromEvent、interval、timer、bindCallback、……
- 实例方法


This is time flowing from left to right to represent the execution of the input Observable.
These are values emitted by the Observable.
This vertical line represents the "complete" notification and indicates that the Observable has completed successfully.
4
6
a
8
multiplyByTen
This box indicates the operator which takes the input Observable (above) to produce the output Observable (below). The text inside the box shows the nature of the transformation.
40
60
This Observable is the output of the operator call.
This X represents an error emitted by the output Observable, indicating abnormal termination. Neither values nor the vertical will be delivered thereafter.

Marble Diagrams

Operator

- Creation
- Transformation
- Filtering
- Combination
- Multicasting
- Error Handling
- Utility
- Conditional and Boolean
- Mathematical and Aggregate

Scheduler

控制Observable何时执行、通知何时发送

- RxJS uses the least concurrency scheduler principle
- 类型
  - null
  - queue
  - asap
  - async

让事件流起来

- 三连击
- 控制按钮不可点击并倒计时
- 左划右划

Resources

- http://reactivex.io/rxjs
- http://rxmarbles.com
- https://www.learnrxjs.io/