# 响应式思维 (Thinking Reactively) | Ben Lesh



Ben Lesh 是 RxJS 库的领导者和布道者，提倡使用响应式思维来抽象逻辑和编写程序，现就职于 Google 。而本文则是对他的一篇研报的记录，该研报是在 AngularConnect 会议中汇报的。研报首先从一个实例开始谈起：

## 实例： Drag & Drop

### 内容

每次 **在目标上按下鼠标（mousedown）** ，开始监听 **页面上鼠标移动（mousemove）** 直到 **鼠标弹起（mouseup）**

相关概念（基础函数）定义：

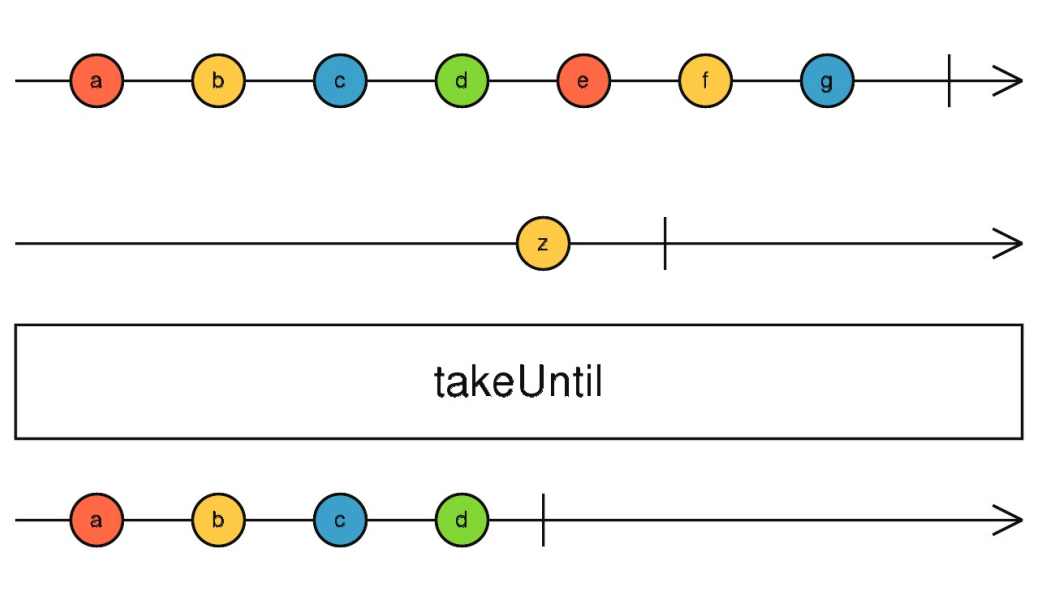
```
const target = document.querySelect('#target')

const targetMouseDown$ = Observable.fromEvent(target, 'mousedown')

const docMouseMove$ = Observable.fromEvent(target, 'mouseover')

const docMouseUp$ = Observable.fromEvent(target, 'mouseup')
```

> 1. 变量后 $ 表示该变量是 Observable 。
> 2. Observable 本质是一个函数，后面 Ben 会解释。

分析

1. 页面上鼠标移动（mousemove）直到鼠标弹起（mouseup） =>

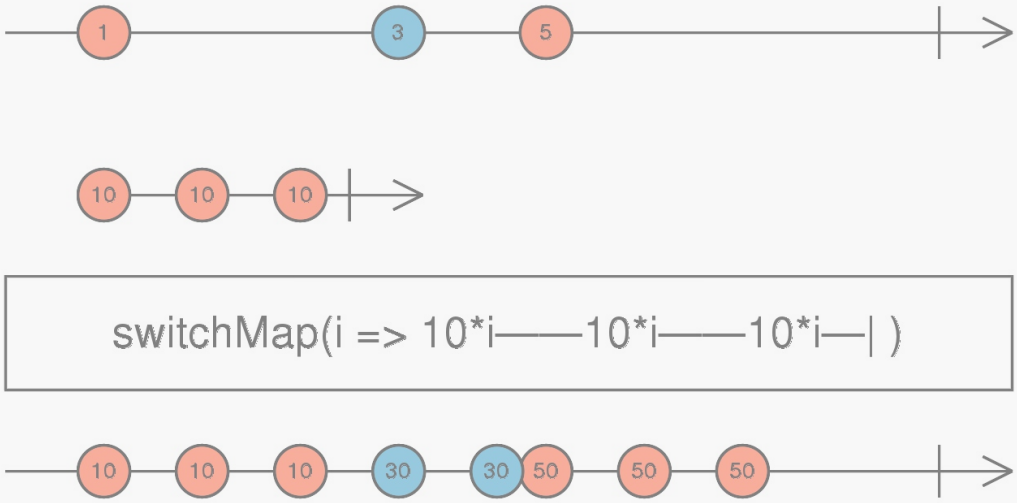
```
docMouseMove$.takeUntil(docMouseUp$)
```

> 操作符 takeUntil 使得 docMouseMove$ 持续推送数据，直到 docMouseUp$ 推送一个通知（数据）后停止。附上 takeUntil 弹珠图：
>
> 
> 

2. 每次在目标上按下鼠标（mousedown），开始监听 =>

```
const dragDrop$ = targetMouseDown$.switchMap(() =>
  docMouseMove$.takeUntil(docMouseUp$)
)
```

> 操作符 switchMap 将 targetMouseDown$ 推送的值传入进内部函数（该例不传入推送值，推送仅仅作为通知使用），然后 **切换** 至 docMouseMove$ 并压平输出（例中 targtMouseDown$ 仅仅推送一次，内部函数仅执行一次，因此无须压平）。附上 switchMap 弹珠图：




小结

解决问题的思路应该是从后向前推导，确认好每个事件流，根据问题组织事件流。

## 进一步了解响应式思维

流变量和非流变量 (自己臆想的，慎看)

> 在系统中流变量都是 Observable。

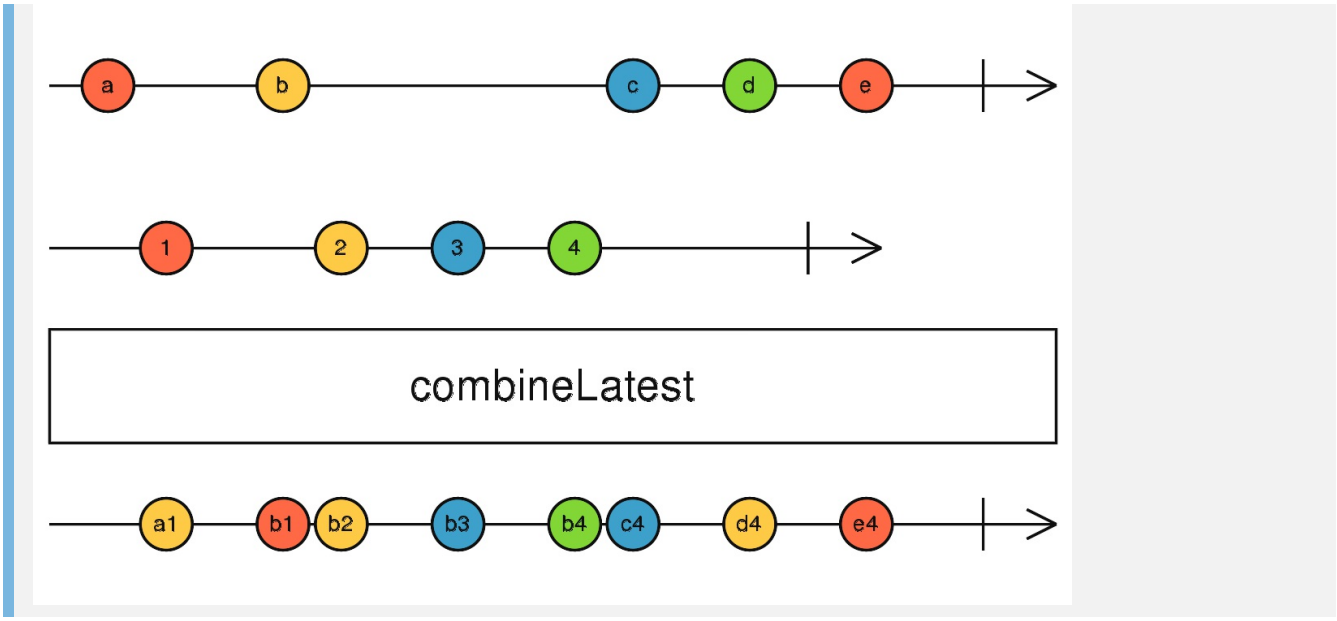
变量顾名思义是变化的，根据广义、狭义相对论可知，变化是针对参照物来说的（提高点 X 格），而非流变量的参照物是整个程序和时间轴。但是如果将参照物改为某一行代码，非流变量随时间是不变的，只是一个占位符。

```
var c = a + b // 站在这里，发现 c 一直不变。
doSometing(c) // 对非流变量的操作，仅调用一次
```

流变量则是一种指向流（ stream ）的标识符。

```
var c$ = a$.combineLatest(b$, (a, b) => a + b) // 站在这里感受下涓涓细流
c$.subscribe(doSomething) // 对流变量的操作，回调多次
```

> 操作符 combineLatest 对任一个 Observable 推送的值，都与其他 Observable 最后值融合。具体融合方法以函数形式给出。惯例附上 combineLatest 弹珠图：




没有了操作符，Observable 就是 。。。

```
// 名字就是标识符，可以改成 asy.ok(...)
promise.then(successFn, errorFn)

// 名字就是标识符，可以改成 asy.ok(...)
observable.subscribe(nextFn, errorFn, completeFn)
```

大概 60+ 的操作符，请查看官网

## 重点：解密 Observable

### Observable 内部是什么？

- 搅乱脑汁的复杂异步
- 应用于航空科技的算法
- 黑魔法
- 独角兽

以上是 Ben 总结的。

### Observable 仅仅是一个函数

1. Observable 有一个名为 observer 的参数：

```
const myObservable = observer => {}
```

2. observer 对象会有几个方法：

```
const myObservable = observer => {
  let i = 0
  const id = setInterval(() => {
    observer.next(i++) // next 方法
    if (i === 10) observer.complete() // complete 方法
  }, 200)
}
```

3. Observable 会返回销毁逻辑：

```
const myObservable = observer => {
  let i = 0
  const id = setInterval(() => {
    observer.next(i++) // next 方法
    if (i === 10) observer.complete() // complete 方法
  }, 200)
  return () => clearInterval(id) // 用于终止订阅
}
```

4. 调用 Observable 函数的同时订阅了你的 observer

```
const myObservable = (observer) => {
  let i = 0
  const id = setInterval(() => {
    observer.next(i++) // next 方法
    if (i === 10) observer.complete() // complete 方法
  }, 200)
  return () => clearInterval(id) // 用于终止订阅
}

// 订阅你的 observer
const teardown = myObservable({
  next(x) {console.log(x)},
  error(err) {console.error(err)},
  complete() {console.info('done)}
})

// 1 秒后取消订阅
setTimeout(() => {
  teardown()
}, 1000)
```

操作符也是一个函数

操作符函数吃进一个 Observable 吐出一个 Observable :

```
const operator = InputObservable => OutputObservable
```

1. 将 OutputObservable 变量展开成函数形式：

```
const operator = (InputObservable) => {
  return (OutObserver) => {...}
}
```

2. 操作符是一个函数，她的参数是 Observable 并且输出也是 Observable ， 也就是通过 InputObservable 构建 OutputObservable ：

```
const operator = InputObservable => {
  return OutObserver => {
    return InputObservable(InObserver)
  }
}
```

3. Observable 就是一个拥有 observer 参数的函数，而 observer 对象的形式是约定好的：

```
const observer = {
  next: (data) => {...},
  error: (err) => {...},
  complete: () => {...}
}
```

也可以短方法声明：

```
const observer = {
  next(data) {...},
  error(err) {...},
  complete() {...}
}
```

4. 构建 InObserver 和 OutObserver 之间的映射关系（操作符是 Observable 和 Observer 之间的操作，而此时还没有给出映射函数，所以 InObserver 和 OutObserver 其实现在还没有变化 ）：

```
const operator = InputObservable => {
  return OutObserver => {
    return InputObservable({
      next(data) {
```

```
        OutObserver.next(data)
      },
      error(err) {
        OutObserver.error(err)
      },
      complete() {
        OutObserver.complete()
      }
    })
  }
}
```

> 不难看出 next(data) { OutObserver.next(data)} 等同于 next = OutObserver.next, error 和 complete 类似，意味着此时 InObserver 等于 OutObserver。

5. 最后添加操作推送数据的映射函数：

```
const operator = (InputObservable, mapFn) => {
  return OutObserver => {
    return InputObservable({
      next(data) {
        OutObserver.next(mapFn(data))
      },
      error(err) {
        OutObserver.error(err)
      },
      complete() {
        OutObserver.complete()
      }
    })
  }
}
```

单独分析 next 方法来观察 InObserver 和 OutObserver 的关系：

```
function InObserver.next(data) {
  let newData = mapFn(data)
  OutObserver.next(newData)
}
```

6. 验证。现在把之前创建的 myObservable 和 operator 应用到程序中：

```
const source = operator(myObservable, x => x + '!')

const teardown = source({
```

```
  next(data) {
    console.log(data)
  },
  error(err) {
    consol.log(err)
  },
  complete() {
    console.log('done')
  }
})

// 4 秒后取消订阅
setTimeout(() => {
  teardown()
}, 4000)
```

输出的结果：

```
0!
1!
2!
...
```

> 这里的结果有问题，因为收到 complete 推送后，并没有取消订阅，因此上面代码设置了显式的取消订阅过程。Ben 在另外的研报中详细介绍了使用 safeObserver 解决上述问题。

7. 酷，来几个串行操作。

```
const source = operator(operator(myObservable, x => x + '!'), x => x + '?')

const teardown = source({
  next(data) {
    console.log(data)
  },
  error(err) {
    consol.log(err)
  },
  complete() {
    console.log('done')
  }
})

// 4 秒后取消订阅
setTimeout(() => {
  teardown()
}, 4000)
```

太繁琐了，想想就头痛：

```
const source = operator(operator(opserator(observable, mapFn), mapFn), mapFn)
```

8. 把 Observable 函数用类来包裹（注意仅仅是把 Observable 函数打包进类里，并不是把 Observable 函数转化成类），操作符作为类的方法，这样便可以使用链式写法调用操作符了：

```
class Observable {
  constructor(observableFn) {
    this.subscribe = observableFn // 好记忆的标识 subscribe
  }
}

const myObservable = new Observable((observer) => {...})

const teardown = myObservable.subscribe({
  next(data) { console.log(data) },
  error(err) { consol.log(err) },
  complete() { console.log('done') }
})
```

9. 添加 map 操作符到类中：

```
class Observable {
  constructor(observableFn) {
    this.subscribe = observableFn // 好记忆的标识 subscribe
  }

  map(mapFn) {
    return new Observable(observer => {
      return this.subscribe({
        next(data) {
          observer.next(mapFn(data))
        },
        error(err) {
          observer.error(err)
        },
        complete() {
          observer.complete()
        }
      })
    })
  }
}
```

现在使用链式写操作符试试：

```
myObservable
  .map(x => x + '!')
  .map(x => x + '?')
  .map(x => x + '.')
  .subscribe({
    next(data) {
      console.log(data)
    }
  })
```

小结

- Observable 就是函数。
- 因为函数仅在调用时执行，所以 Observable 是惰性的。
- 操作符也是函数，输入是 Observable，输出也是 Observable 。
- 链式操作就是连接每个操作的 observer 。

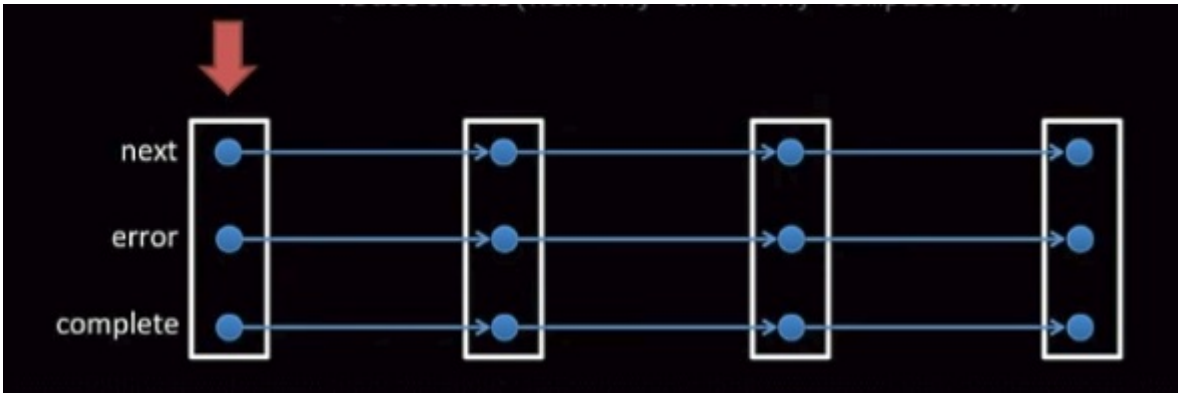
# 图示

## 正常推送数据

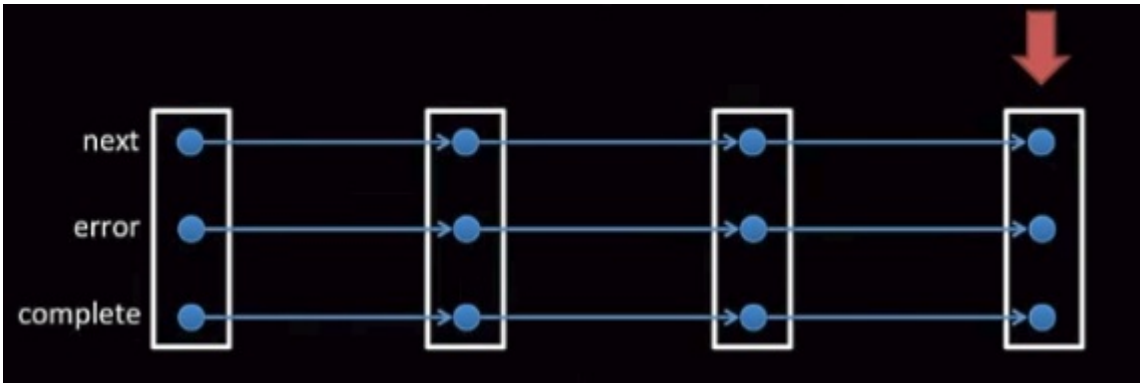
下面将 Observable 的运行过程可视化，先给出 Observable 的订阅实例：

```
Observable.interval(1000) // like setInterval
  .filter(x => x % 2 === 0)
  .map(x => x + x)
  .subscribe(next, error, complete)
```

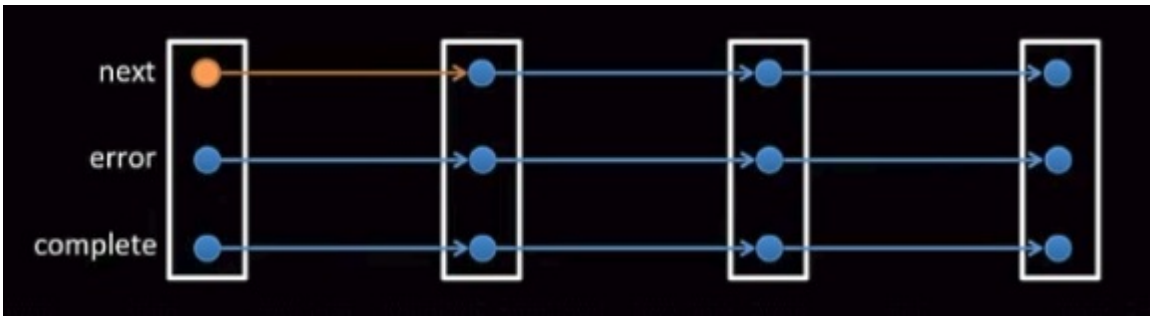
1. 开始是数据的产生者

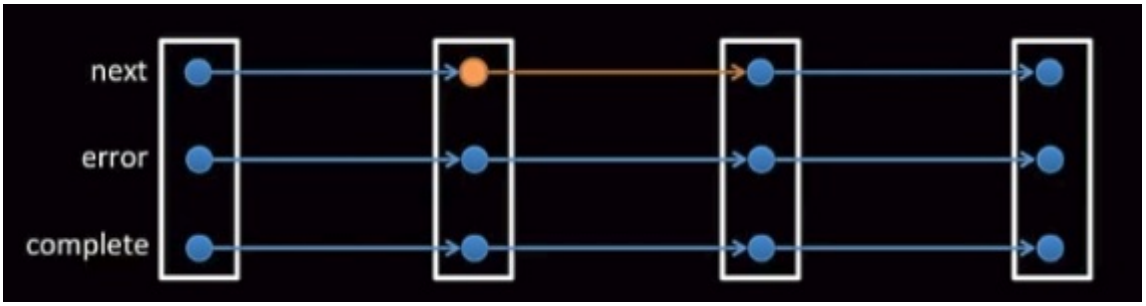


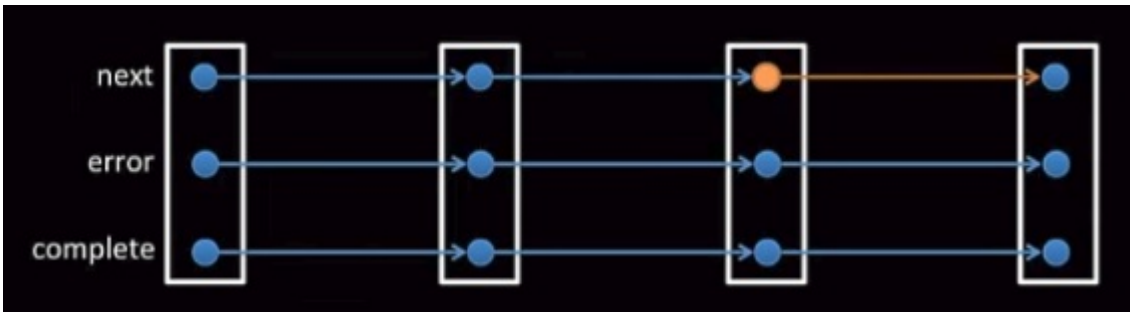

2. 流程的最后是你的回调处理，也就是数据的消费者

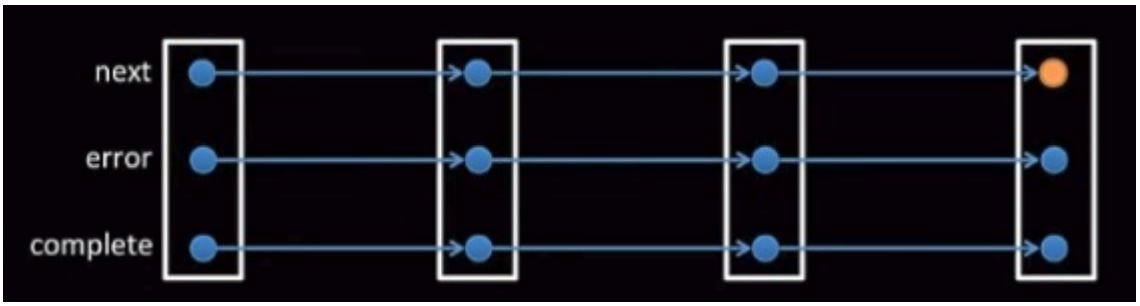



3. 最初推送的是 0 ， 每一步的图示：

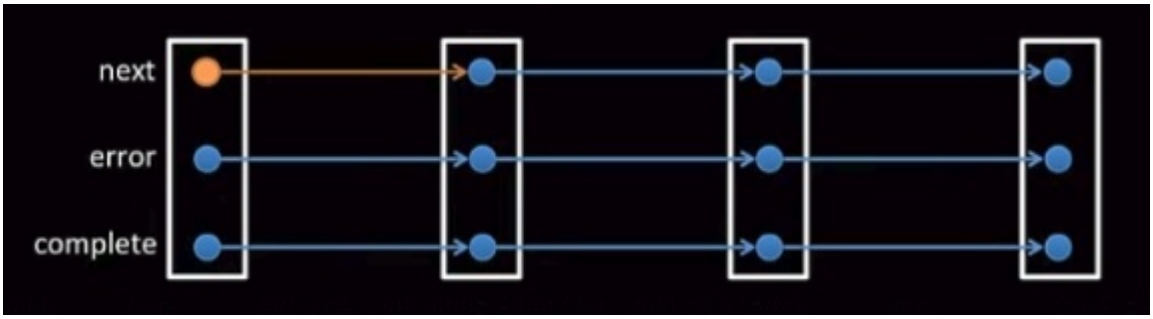


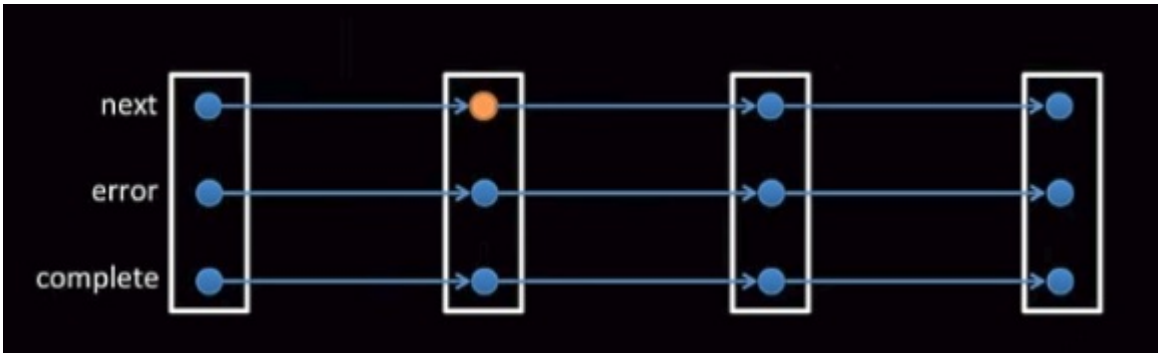




4. 然后是推送 1 ：




然后就没有然后了， 1 被 filter 过滤掉了。

### 重点：异常处理

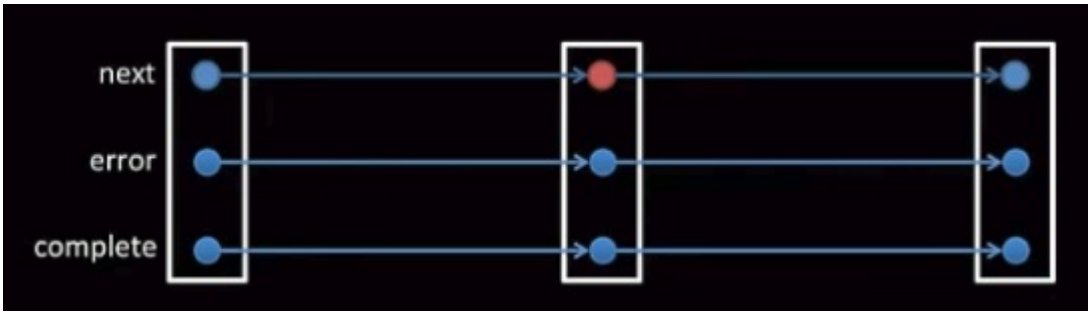
异常处理过程可能会给我们带来一些疑惑，主要是因为以下几个事件影响：

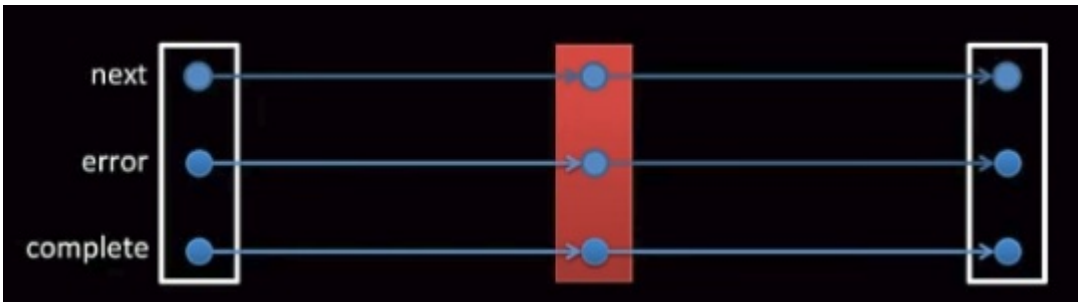
- error() 被调用
- complete() 被调用
- 取消订阅

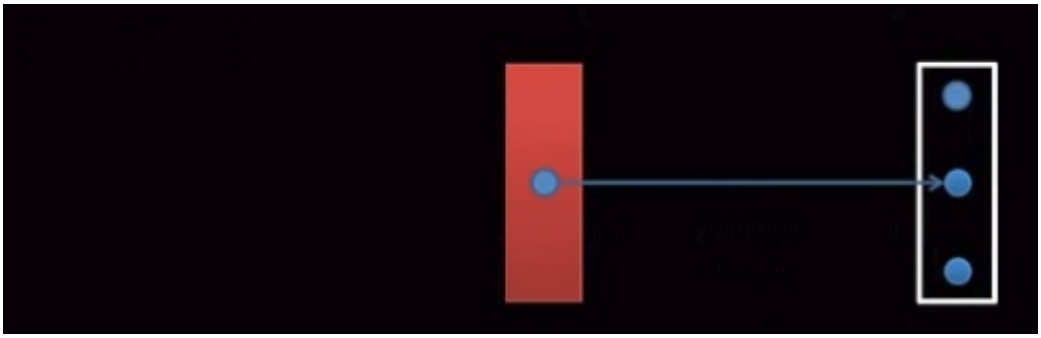
这些事件发生后， Observable 将不再推送数据。继续给出实例：

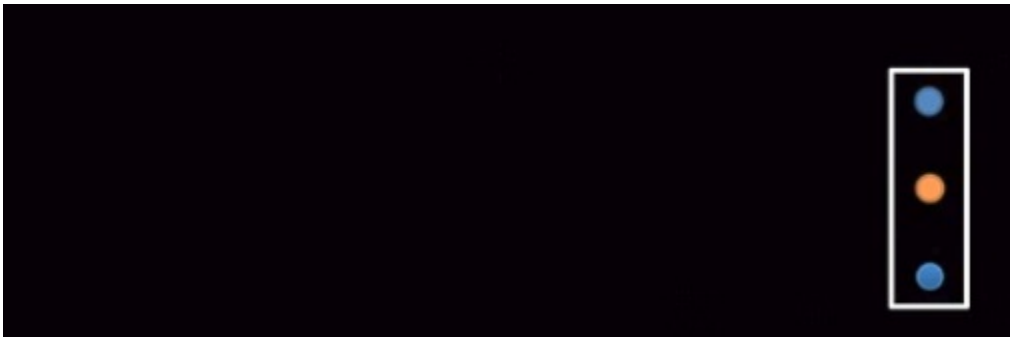
```
Observable.interval(1000)
  .map(x => {
    if (x === 1) {
      throw new Error('haha')
    }
    return x
  })
  .subscribe(next, error, complete)
```

1. 处理推送 0 的过程略过。
2. 生产者推送 1 是，抛出了异常：



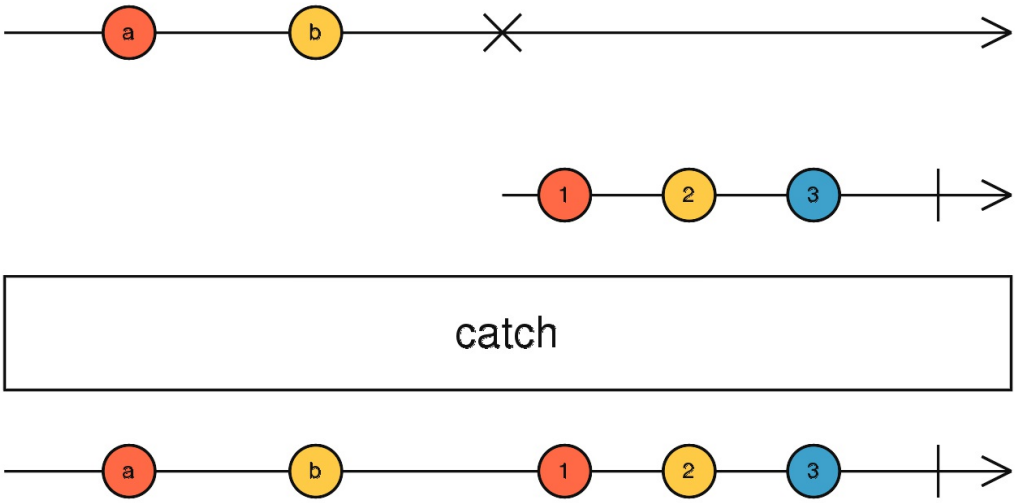


> 当抛出异常后，Observable 不会继续推送数据（取消订阅），而消费者将会使用 error() 处理异常。
>
> 这个还有个问题，Ben 在他的专栏里提到过，多播场景的错误捕获。

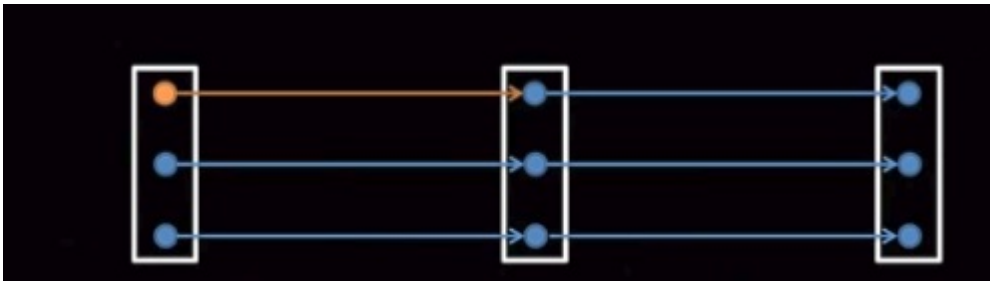
3. 当异常抛出后，Observable 就挂掉了，如果还想继续推送如何实现？答案是：创建 observer 分支。

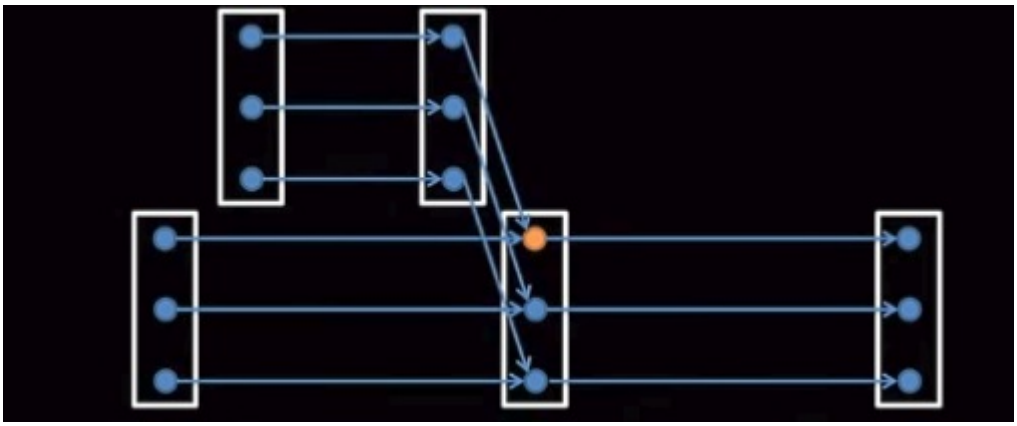
```
Observable.interval(10000)
  .switchMap(() => this.http.get(url).catch(err => Observable.empty()))
  .subscribe(data => render(data))
```

先来学习操作符 catch ，它会捕获和处理 Observable 推送的异常，并返回一个新的 Observable 或者继续抛出异常。附上弹珠图：

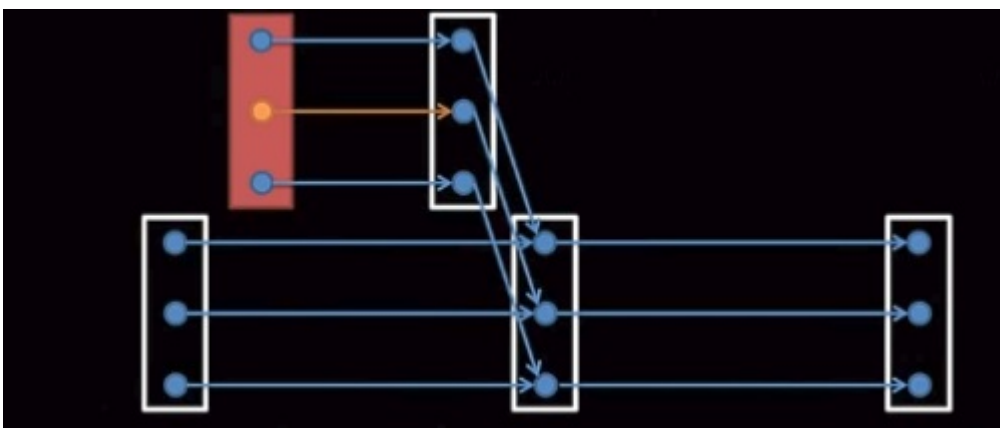



图示具体流程：

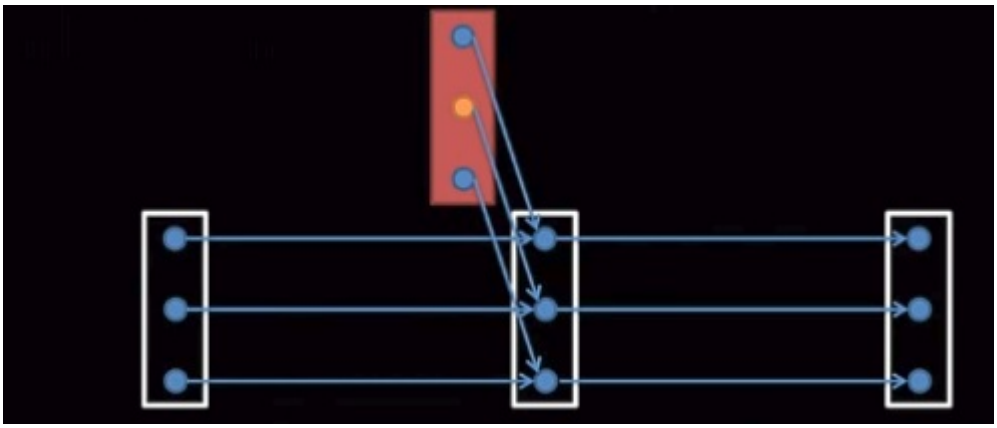




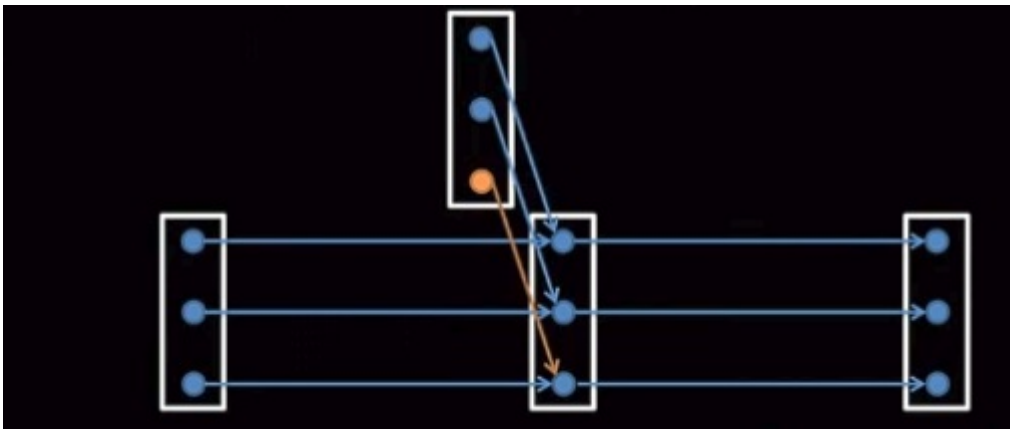
网络不好，查询过程超时。



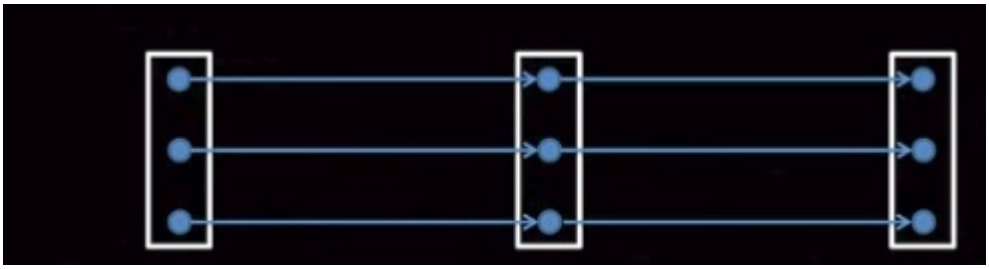
取消 Ajax Observable 的订阅



转化异常到新的 Observable



取消 Observable.empty 的订阅



小结：

- 创建另外一个 observer 链
- 使用 catch 增强这个链的鲁棒性
- 守护了原始的 observer 链

## 响应式思维的适用场景

Ben 在研报的最后分析了响应式思维的使用场景，这里简单的将 PPT 页翻译，具体的实际应用还是需要在编程中发觉和选择的。

- 将多个事件融合在一起
- 添加延时

- 客户端限制流量
- 协调异步任务
- 需要注销机制

## 总结

最后将该文的具体内容概述为以下 6 个方面。

- 逆向思维
- 任何的变量都可以被观察
- Observable 是函数
- Observer 链处理计算
- 调用 error() 会终止 observer 链
- 尽情使用 Rx 。

本文是在我学习 RxJS 过程中为了加强记忆和便于理解而记录的，里面添加了大量的个人学习倾向，局限于个人知识面有限，难免有不当和错误之处，欢迎大家批评指导。