

# RxJS 分享第二期-操作符

政法BG-刘灵辉

项目地址

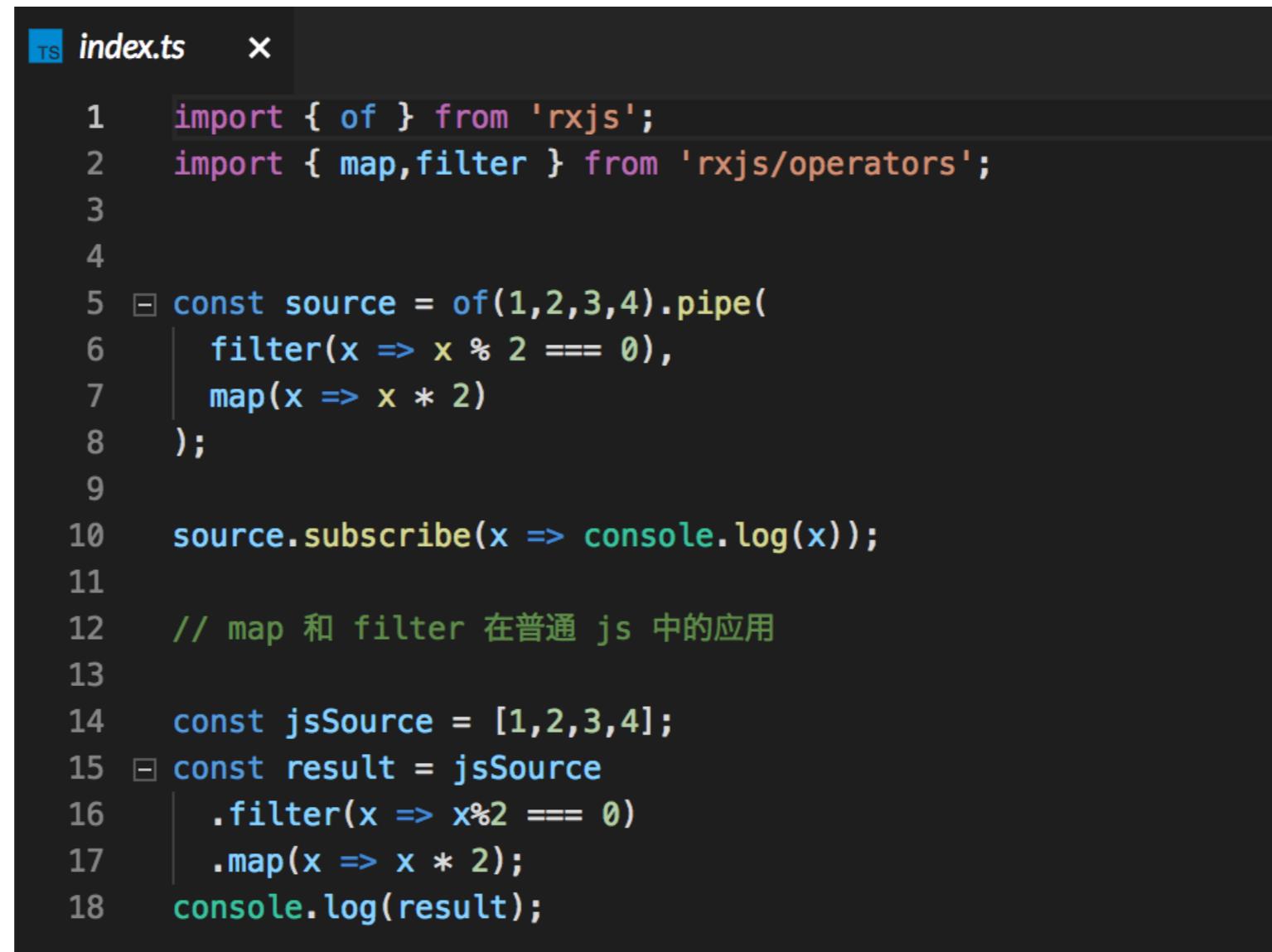
# 本期內容

1. RxJS中的操作符
2. 使用操作符创建RxJS数据流
3. 使用操作符合并RxJS数据流
4. 有趣的案例，希望能激发你的兴趣
5. 薛定谔的总结

# RxJS中的操作符

- Rx的实现与操作符的关系
- 操作符是返回一个Observable对象的函数
  - 根据其他Observable对象返回Observable对象
  - 根据其他类型输入返回Observable对象
  - 凭空创造一个Observable对象

# 简单的示例



The screenshot shows a code editor window titled "index.ts". The code uses RxJS operators to filter and map an array. It demonstrates how RxJS's filter and map operators differ from their counterparts in plain JavaScript.

```
1 import { of } from 'rxjs';
2 import { map, filter } from 'rxjs/operators';
3
4
5 const source = of(1,2,3,4).pipe(
6   filter(x => x % 2 === 0),
7   map(x => x * 2)
8 );
9
10 source.subscribe(x => console.log(x));
11
12 // map 和 filter 在普通 js 中的应用
13
14 const jsSource = [1,2,3,4];
15
16 const result = jsSource
17   .filter(x => x%2 === 0)
18   .map(x => x * 2);
19
20 console.log(result);
```

对JS而言:

filter和map都是数组对象的成员函数  
filter和map返回对象依然是数组对象  
filter和map不会改变原本的数组对象

对RxJS而言:

filter和map都是Observable对象的成员函数  
filter和map返回结果是Observable对象  
filter和map不会改变原本的Observable对象

# 操作符的分类

- 创建类（本次内容）
- 合并类（本次内容）
- 转化类
- 过滤类
- 多播类
- 错误处理类
- 辅助工具类
- 条件分支类
- 数学和合计类

# 静态和实例操作符

- 静态分类
  - 不需要Observable实例就可以执行的函数
  - 大部分的创建类操作符 (of,interval,from,ajax..)
  - `import { xx } from 'rxjs';`
- 实例分类
  - 需要创建好的Observable对象的函数
  - 其他大多数非创建类操作符
  - `import { xxx } from 'rxjs/operators';`

# 创建类操作符



# 创建类操作符

- 能够创造出Observable对象的方法
  - (绝大部分) 不依赖于其他Observable对象
  - 凭空创造
  - 依据其他非Observable数据源创造
    - Dom结构的变更
    - HttpResponse的结果

# 创建同步数据流

- create 操作符
- of 操作符
- range 操作符
- generate 操作符
- repeat 操作符
- empty 操作符
- never 操作符
- throw 操作符

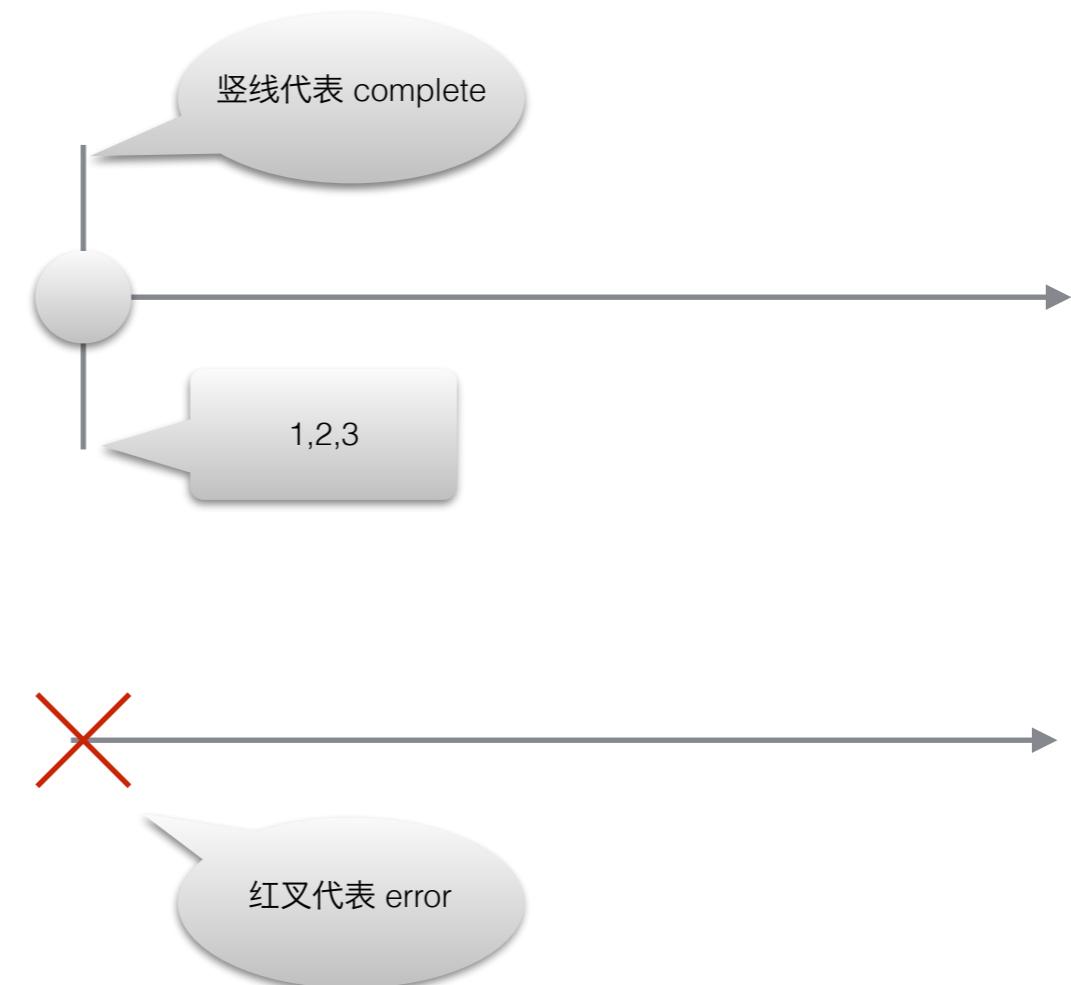
# create 操作符

- create 函数直接调用 Observable 的构造函数
- 静态操作符
- 写demo时可能会用到该操作符
- 业务场景中大多数会使用其他操作符及其组合

```
// create 操作符

var observable = Observable.create(function (observer) {
  observer.next(1);
  observer.next(2);
  observer.next(3);
  observer.complete();
});
observable.subscribe(
  value => console.log(value),
  err => { },
  () => console.log('this is the end')
);

var observable2 = Observable.create(function (observer) {
  observer.error('something wrong');
});
observable2.subscribe(
  value => console.log(value), // there is no value emit
  err => console.log(err),
  () => console.log('this is the end') // there is no complete
);
```



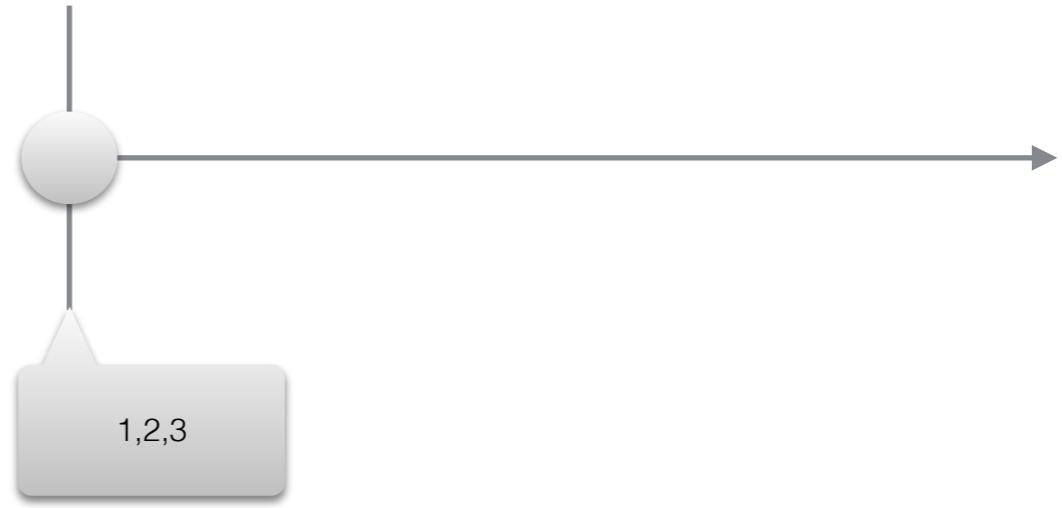
# of 操作符

- 创建指定数据集合的 Observable 对象
- 静态操作符
- 可以使用构造函数实现
- 使用更轻便的 of 操作符实现
- 产生 Cold Observable (对于每个订阅的 Observer 都会 emit 同样的数据)
- 产生的数据之间没有数据间隔，数据是同步抛出的

```
// of 操作符 和 create 操作符的比较
```

```
const ofSource = of(1,2,3);
const createSource = Observable.create(observer => {
  observer.next(1);
  observer.next(2);
  observer.next(3);
  observer.complete();
});

ofSource.subscribe(value => {
  console.log(value);
});
createSource.subscribe(value => {
  console.log(value);
})
```



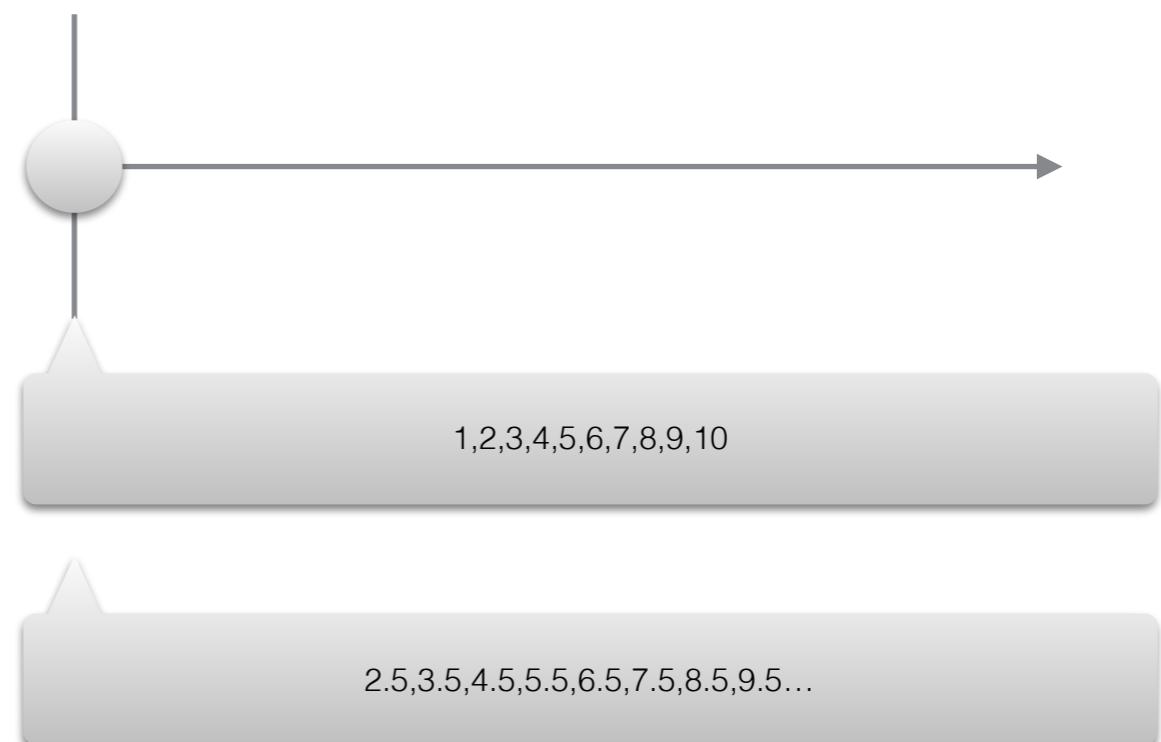
# range 操作符

- 产生一系列连续数字序列的 Observable 对象
- 静态操作符
- 与 of 操作符一样，所有数据同步抛出，无论数据有一千还是一万个数字
- 第一个参数是起始数值(起始数值可以是**非整数**)，第二个参数是期望序列的长度

```
// range 操作符

const rangeSource = range(1, 10);
rangeSource.subscribe(value => {
  console.log(value);
});

const rangeSource2 = range(2.5, 10);
rangeSource2.subscribe(value => {
  console.log(value);
});
```

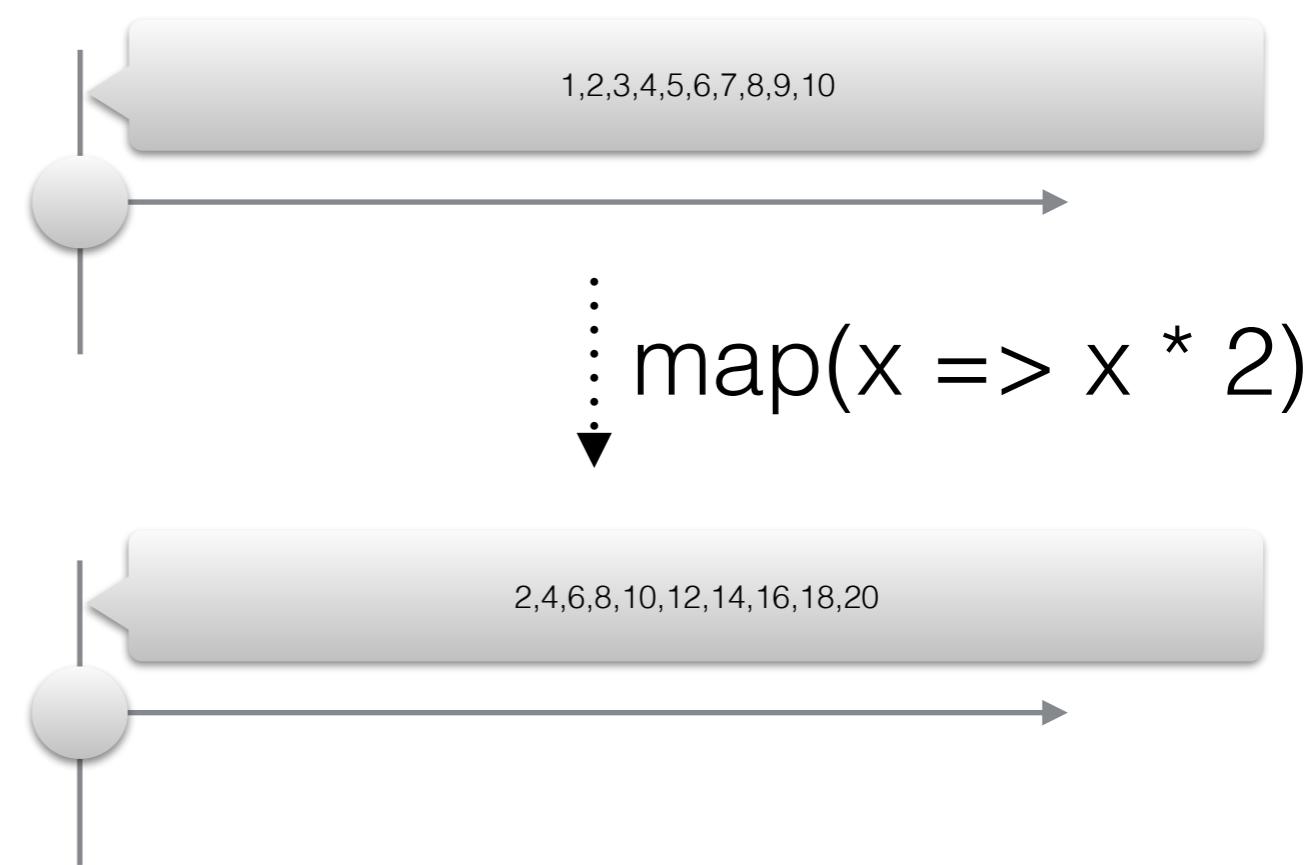


# 如果想使用获得每次递增2的数字序列呢

- 无法直接使用range进行调参实现
- 静态操作符
- RxJS希望每个操作符都保持尽量精简
- 递增2的数字序列可以通过多个操作符组合实现
- range + map

```
// range 操作符 + map 操作符实现 递增2的数字序列

const rangePlusMapSource = range(1,10).pipe(
  map(value => value * 2)
);
rangePlusMapSource.subscribe(value => {
  console.log(value);
});
```



# generate 操作符

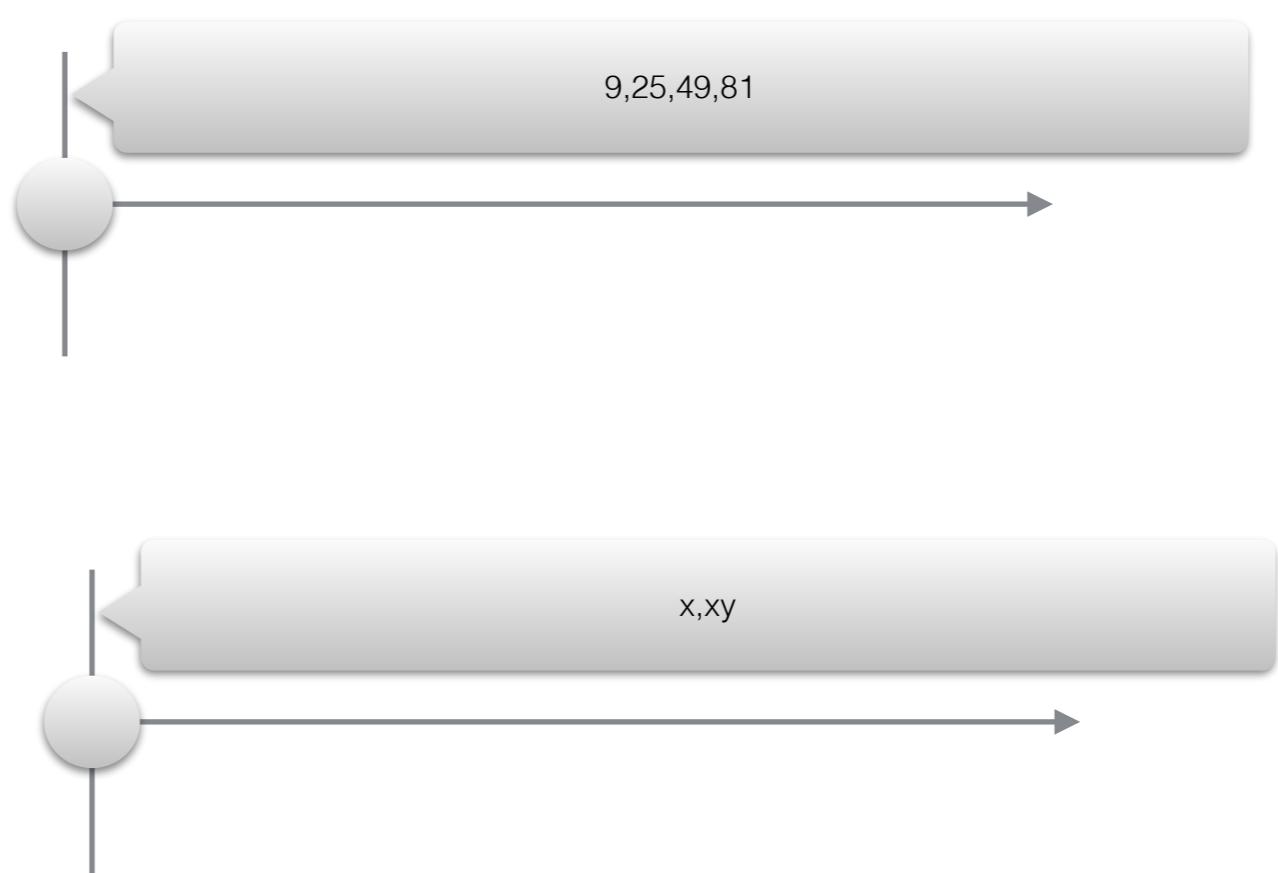
- 创造不连续的数字序列或不仅仅是数字序列
- 静态操作符
- 类似于普通的for循环

```
// generate 操作符
// 一般姿势

const generateSource = generate(
  1,
  value => value < 10,
  value => value + 2,
  value => value * value
)
generateSource.subscribe(value => {
  console.log(value);
})

// 非数字序列姿势

const generateSourceNonNumber = generate(
  'x',
  value => value.length < 3,
  value => value + 'y',
  value => value
);
generateSourceNonNumber.subscribe(value => {
  console.log(value);
})
```



# repeat 操作符

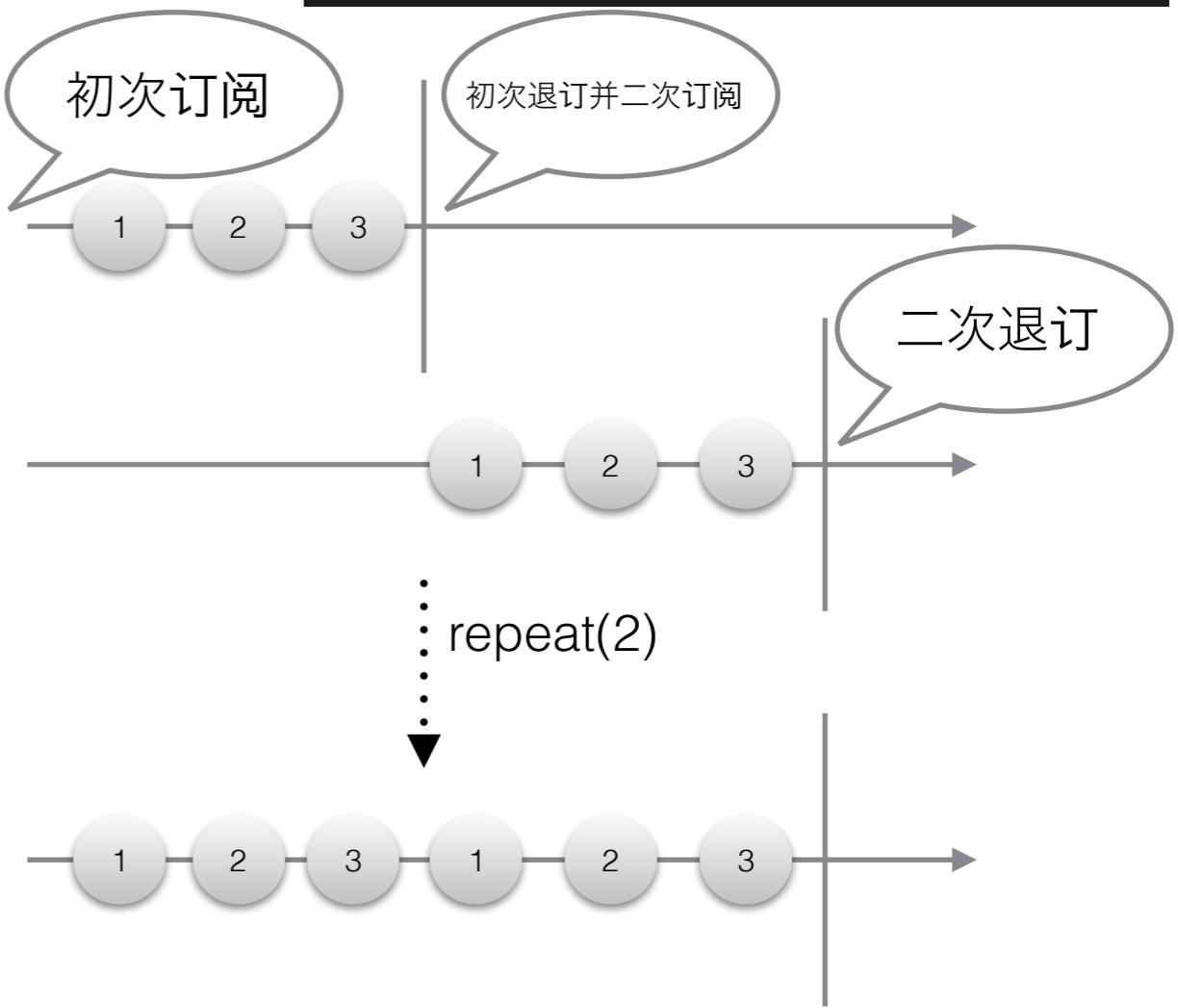
- 重复上游 Observable 的数据若干次
- 实例操作符
- 不会修改上游 Observable
- 上游 Observable **结束后才会进行重复**
- 传入参数为repeat次数，不传入或传入负数为无限循环

```
// repeat 操作符
const repeatSource$ = of(1,2,3).pipe(
  repeat(2)
);
repeatSource$.subscribe(value => {
  console.log(value);
});
```

```
// repeat操作符对complete状态的上游进行操作

const repeatSource2Incomplete$ = Observable.create(observer => {
  setTimeout(() => {
    observer.next(1)
  }, 1000);
  setTimeout(() => {
    observer.next(2)
  }, 2000);
  setTimeout(() => {
    observer.next(3)
  }, 3000);
  setTimeout(() => {
    observer.complete()
  }, 4000);
  return {
    unsubscribe: () => {
      console.log('on subscribe');
    }
  };
}).pipe(
  repeat(2)
);

repeatSource2Incomplete$.subscribe(
  console.log,
  null,
  () => console.log('complete')
);
```



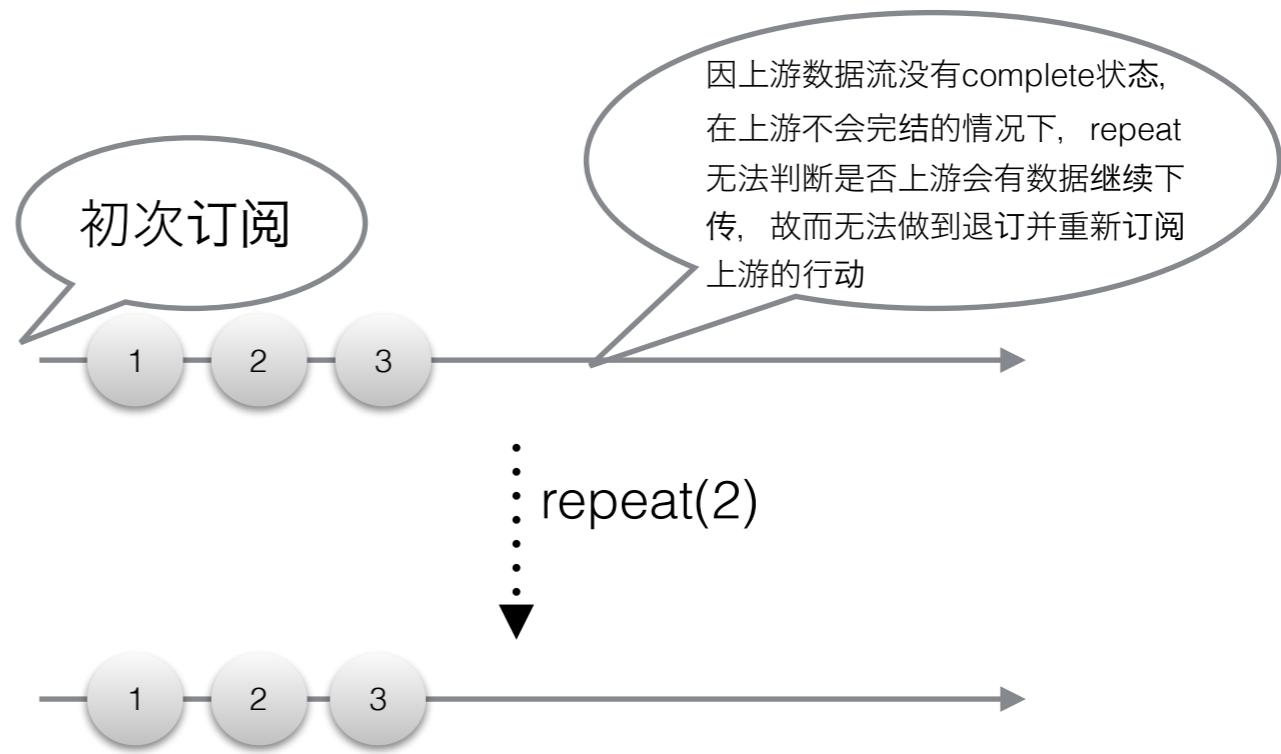
# repeat 操作符

- 若上游 Observable 不 complete 会发生什么？
- 上游的 Observable 可能并不是同步产生的，但是这并不影响 repeat 本身是同步的

```
// repeat操作符对非complete状态的上游进行操作
```

```
const repeatSource2Incomplete2$ = Observable.create(observer => {
  setTimeout(() => {
    observer.next(1)
  }, 1000);
  setTimeout(() => {
    observer.next(2)
  }, 2000);
  setTimeout(() => {
    observer.next(3)
  }, 3000);
  return {
    unsubscribe: () => {
      console.log('on subscribe');
    }
  };
}).pipe(
  repeat(2)
);

repeatSource2Incomplete2$.subscribe(
  console.log,
  null,
  () => console.log('complete')
);
```



# empty 操作符

- 产生一个直接完成的 Observable
- 不接受任何参数
- 不产生任何数据就直接完结
- 是的什么都不产生就完结的数据流

```
// empty 操作符

const emptySource$ = empty();
emptySource$.subscribe(value => {
  console.log(value);
})
```



# throw 操作符

- 产生一个直接抛出错误的 Observable
- 接收 1 - 2 个参数作为 error 的内容
- 不产生任何数据就直接抛出错误
- throw 关键字是 JS 关键字，无法直接导入 throw
- 导入 throwError 代替

```
// error 操作符
const errorSource$ = throwError('throw a error');
errorSource$.subscribe({
  next: val => console.log(val),
  complete: () => console.log('Complete!'),
  error: val => console.log(`Error: ${val}`)
});
```



# never 操作符

- 不像 empty 和 throw 至少产生一个动作, never 不产生任何动作
- 没有数据 emit
- 没有 complete 状态
- 没有 error 状态

```
// never 操作符

const neverSource$ = never();

neverSource$.subscribe({
  next: val => console.log(val),
  complete: () => console.log('Complete!'),
  error: val => console.log(`Error: ${val}`)
});
```

什么都没有

# 创建异步数据流

- interval 操作符
- timer 操作符
- from 操作符
- fromPromise 操作符
- fromEvent 操作符
- ajax 操作符
- repeatWhen 操作符
- defer 操作符

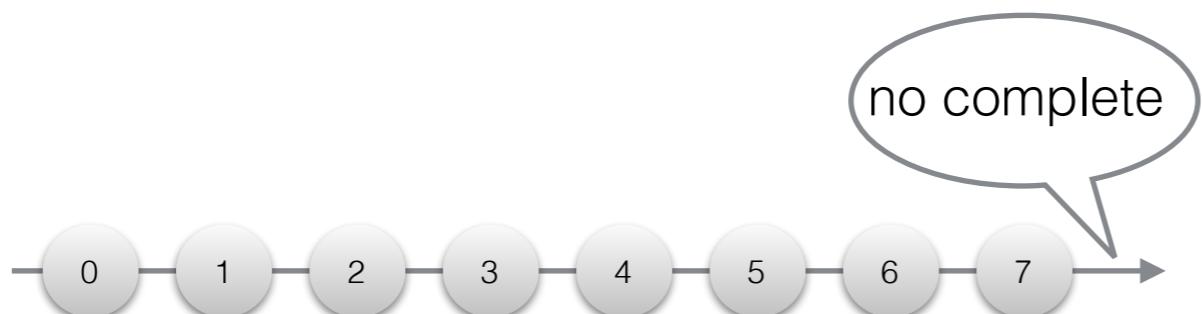
# interval 操作符

- 地位约等于 JS 中的 setInterval()
- 接受数值类型的参数，代表产生数据的间隔毫秒数
- 没有 complete 状态
- 需要使用退订动作停止该序列
- 序列总是从0开始

```
// interval 操作符

const intervalSource$ = interval(1000);
const intervalSubscription = intervalSource$.subscribe(value => {
  console.log(value);
});

// unsubscribe 将会停止 interval 数据序列的继续生成
|
intervalSubscription.unsubscribe();
```



# timer 操作符

- 地位约等于 JS 中的 setTimeout()
- 第一个参数接受数值类型或 Date 类型(若延迟时间为明确的时间点，则使用Date类型传参)
- 第二个参数则会产生一个持续 emit 数据的 Observable 对象，类似于 Interval()
- 即使有第二个参数，第一个参数仍然决定了第一个数据何时被 emit
- 当第二个参数存在时，timer() 的功能则超越了 setTimeout()，且数据序列不会自动 complete

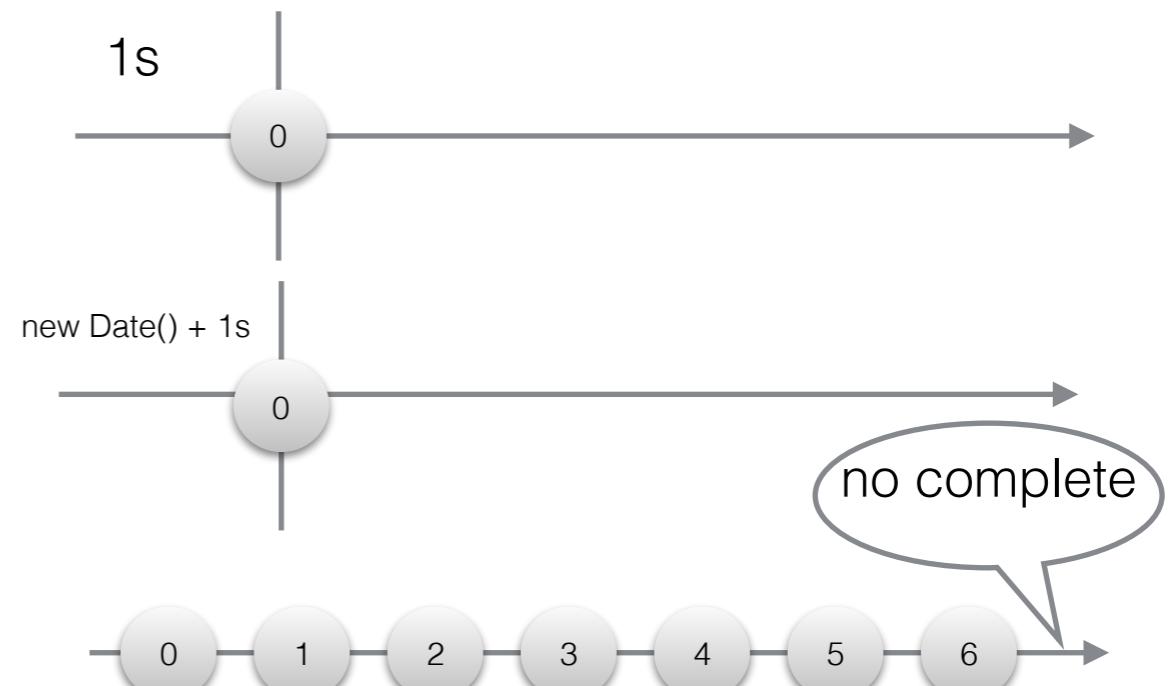
```
// timer 操作符

// 第一个参数传入数值类型

const timerSource1$ = timer(1000);
const timerSubscription = timerSource1$.subscribe(value => {
  console.log(value);
})
// 第一个参数传入Date类型

const now = new Date();
const laterTime = new Date(now.getTime() + 1000);
const timerSource2$ = timer(laterTime);
const timerSubscription2 = timerSource2$.subscribe(value => {
  console.log(value);
});
// 传入第二个参数

const timeSource3$ = timer(1000, 1000);
const timerSubscription3 = timeSource3$.subscribe(value => {
  console.log(value);
})
```



# from 操作符

- 保罗万象的from
- 接受参数 像 Observable 即可
- from 会将传入的参数转化为 Observable
- 可以接收数组, 字符串, Iterator 或 Observable
- 同步亦或是异步根据传入的参数决定

```
// from 操作符

const fromSource1$ = from([1,2,3]);
fromSource1$.subscribe(value => {
  console.log(value);
})

// arguments可以访问所有的参数调用, arguments虽然不是数组但是有length属性及可遍历的能力
function toObservable() {
  return from(arguments);
}

const fromSource2$ = toObservable(1,2,3);
fromSource2$.subscribe(value => {
  console.log(value);
})

const fromSource3$ = from('123');
fromSource3$.subscribe(value => {
  console.log(value);
})

function * generateNumber(max) {
  for(let i= 1; i <= max; ++i) {
    yield i;
  }
}

const fromSource4$ = from(generateNumber(3));

fromSource4$.subscribe(value => {
  console.log(value);
});
```



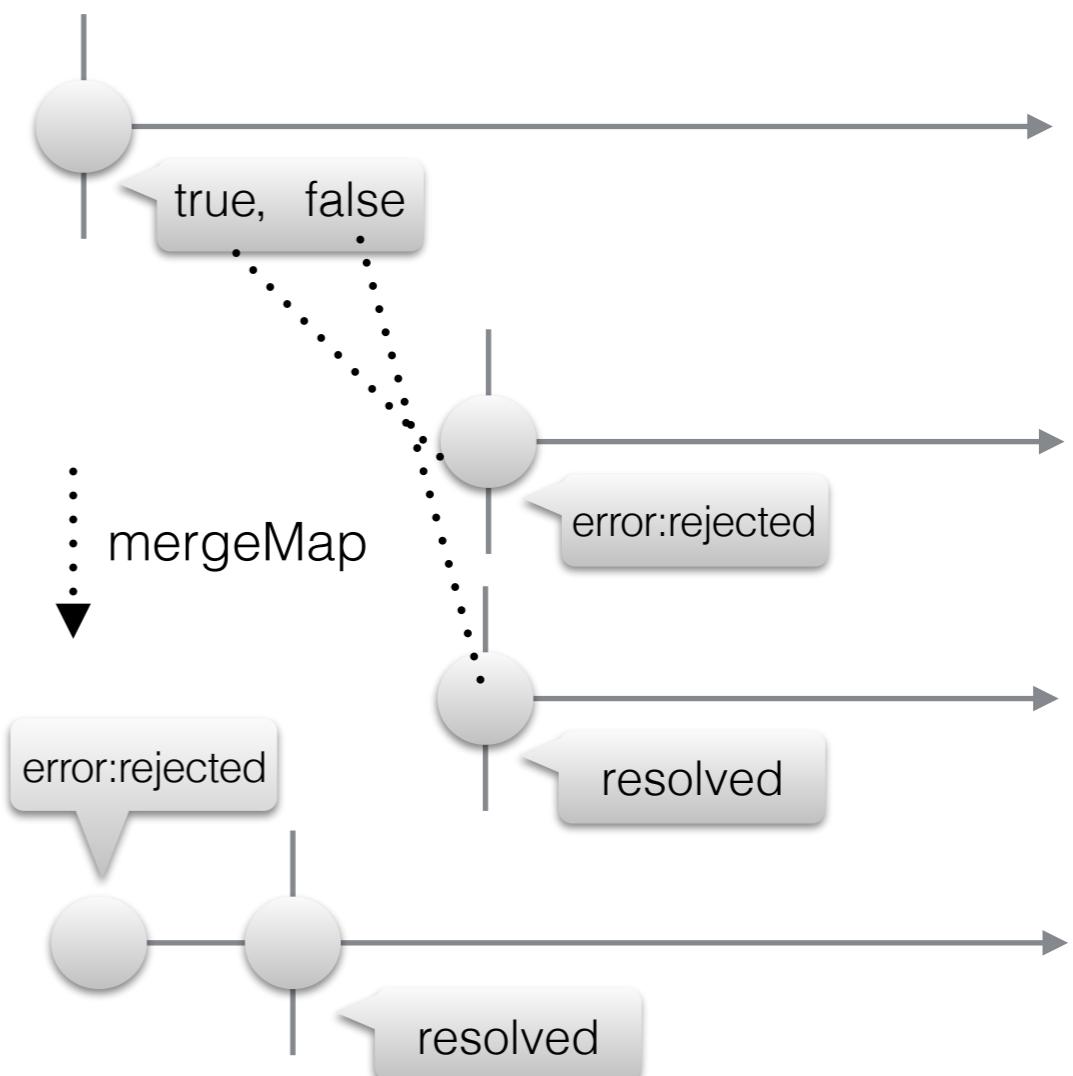
# fromPromise 操作符

- 用于处理异步交接
- 在6.0.3 beta版本后 fromPromise 已经不在作为公开API存在，已被融入from操作符中
- Promise的success 和 reject 状态都代表了 Observable的完结

```
// fromPromise 操作符 -  from 操作符

const myPromise = willReject => {
  return new Promise((resolve, reject) => {
    if (willReject) {
      reject('Rejected!');
    }
    resolve('Resolved!');
  });
};

const source = of(true, false);
const example = source.pipe(
  mergeMap(val =>
    from(myPromise(val)).pipe(
      // 捕获Observable中的error
      catchError(error => of(`Error: ${error}`))
    )
  )
);
// 输出: 'Error: Rejected!', 'Resolved!'
const subscribe = example.subscribe(val => console.log(val));
```



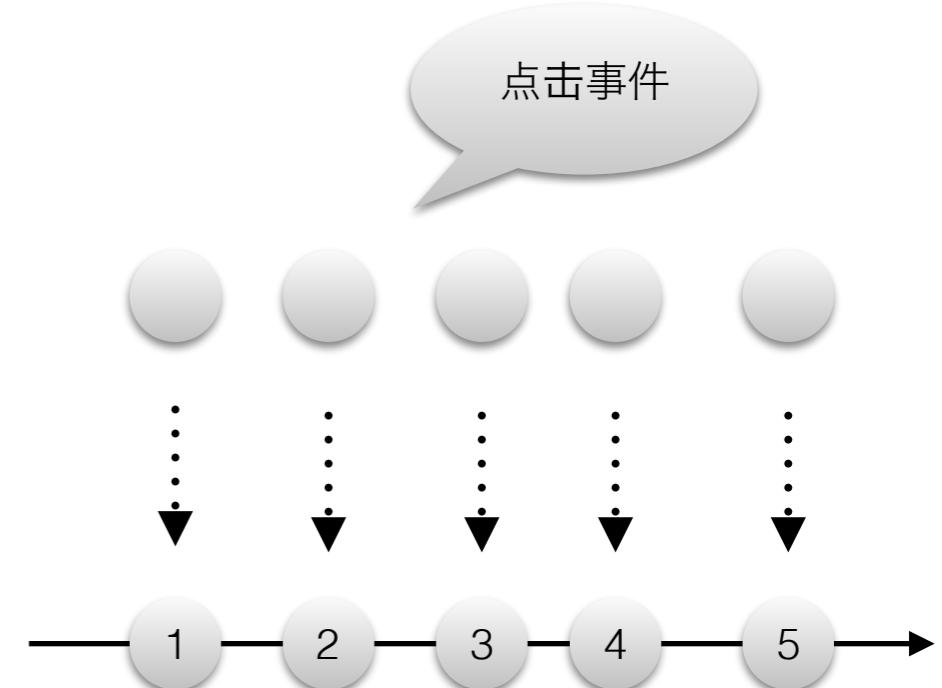
# fromEvent 操作符

- 最常见用于将DOM中的事件转化为Observable对象中的数据
- 在Node服务中常见于使用其获取事件通知
- 异步创建符，因为事件本身是异步的

```
<div>
  <button id="click">Click</button>
</div>
0</span>
```

```
// fromEvent 操作符

let clickCount = 0;
const event$ = fromEvent(document.querySelector('#click'), 'click');
event$.subscribe(() => {
  document.querySelector('#text').innerText = ++clickCount;
})
```



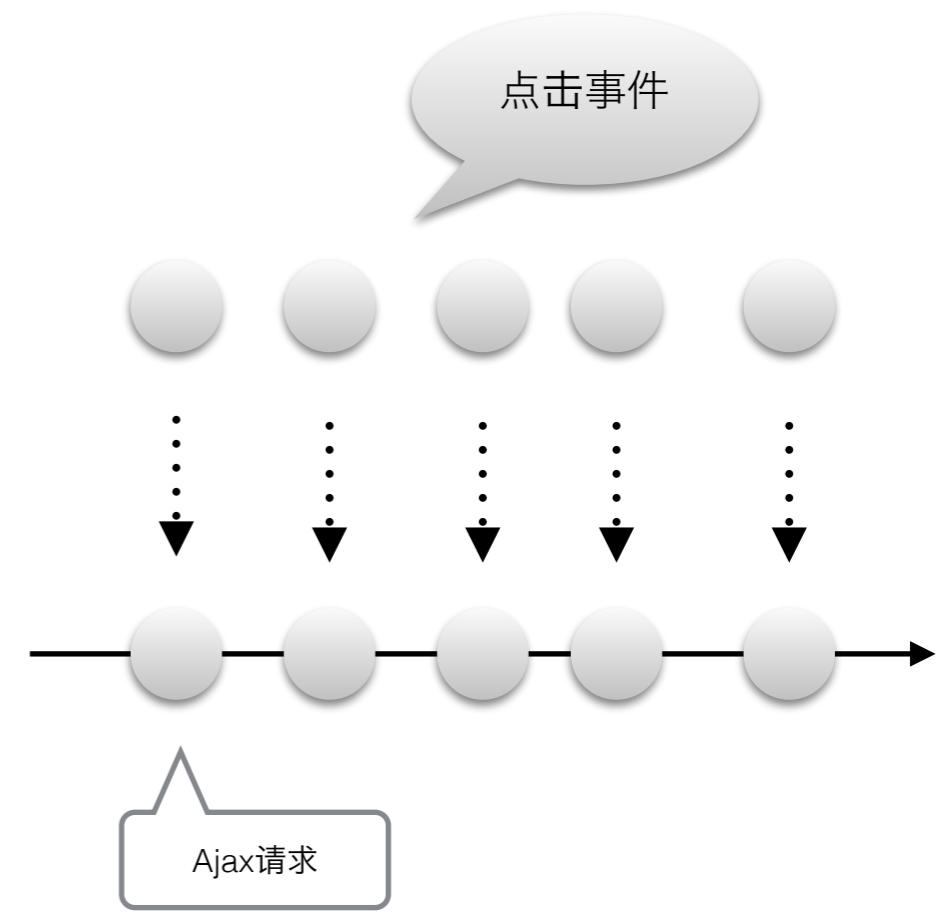
# ajax 操作符

- 顾名思义，将 AJAX 请求的返回结果产生 Observable 对象
- 返回 Angular in Depth 翻译 repo 的 star 数量

```
<div>
  <button id="star">get AiD Star Count</button>
</div>
<span>AiD 的 Star 数为 </span><span id="text">0</span>
```

```
// ajax 操作符

const Source$ = fromEvent((document.querySelector('#star')), 'click')
.subscribe(
() => {
  // console.log(document.querySelector('#text').innerText);
  ajax.getJSON(`https://api.github.com/repos/AngularInDepth/angularindepth`)
.subscribe(value => {
  const starCount = value.stargazers_count;
  document.querySelector('#text').innerText = starCount;
})
})
```



# repeatWhen 操作符

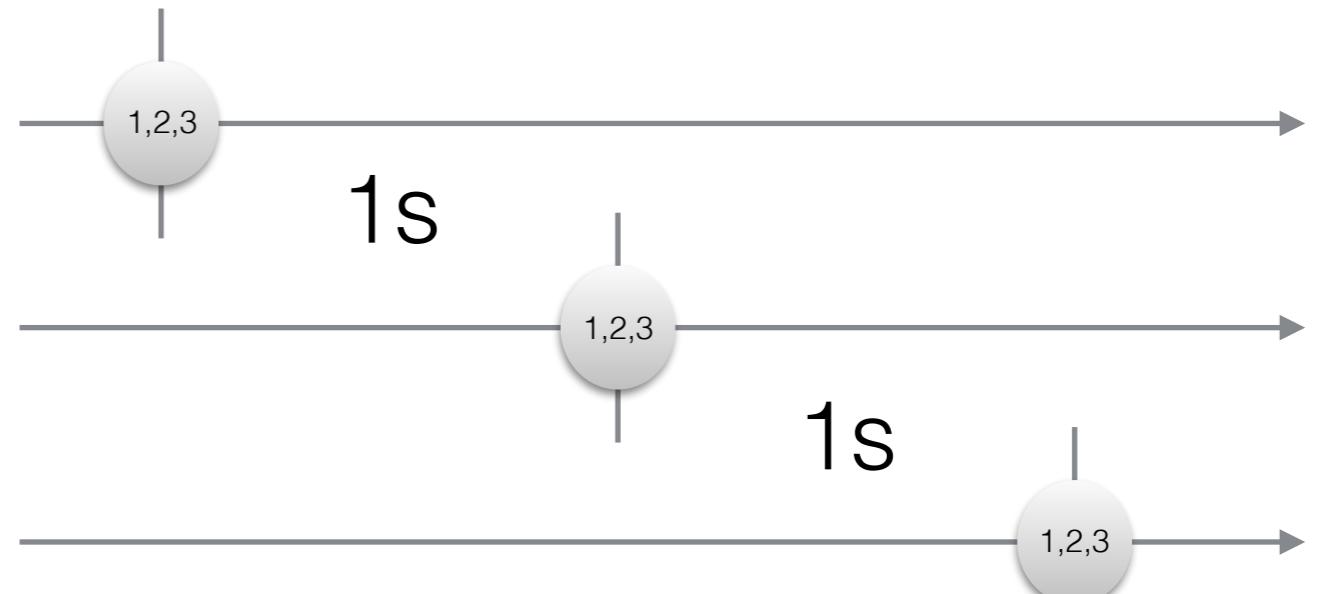
- repeat 可以重复订阅上游的 Observable，但是不能实现对重复订阅的时间节点的控制
- 接受一个函数作为参数，函数在上游产生状态变化时被调用
- 操作符返回一个 Observable 对象作为控制器
- 当作为控制器的 Observable 对象 emit 数据时，操作符就会对上游数据取消订阅并重新订阅
- 实例操作符

```
// repeatWhen 操作符

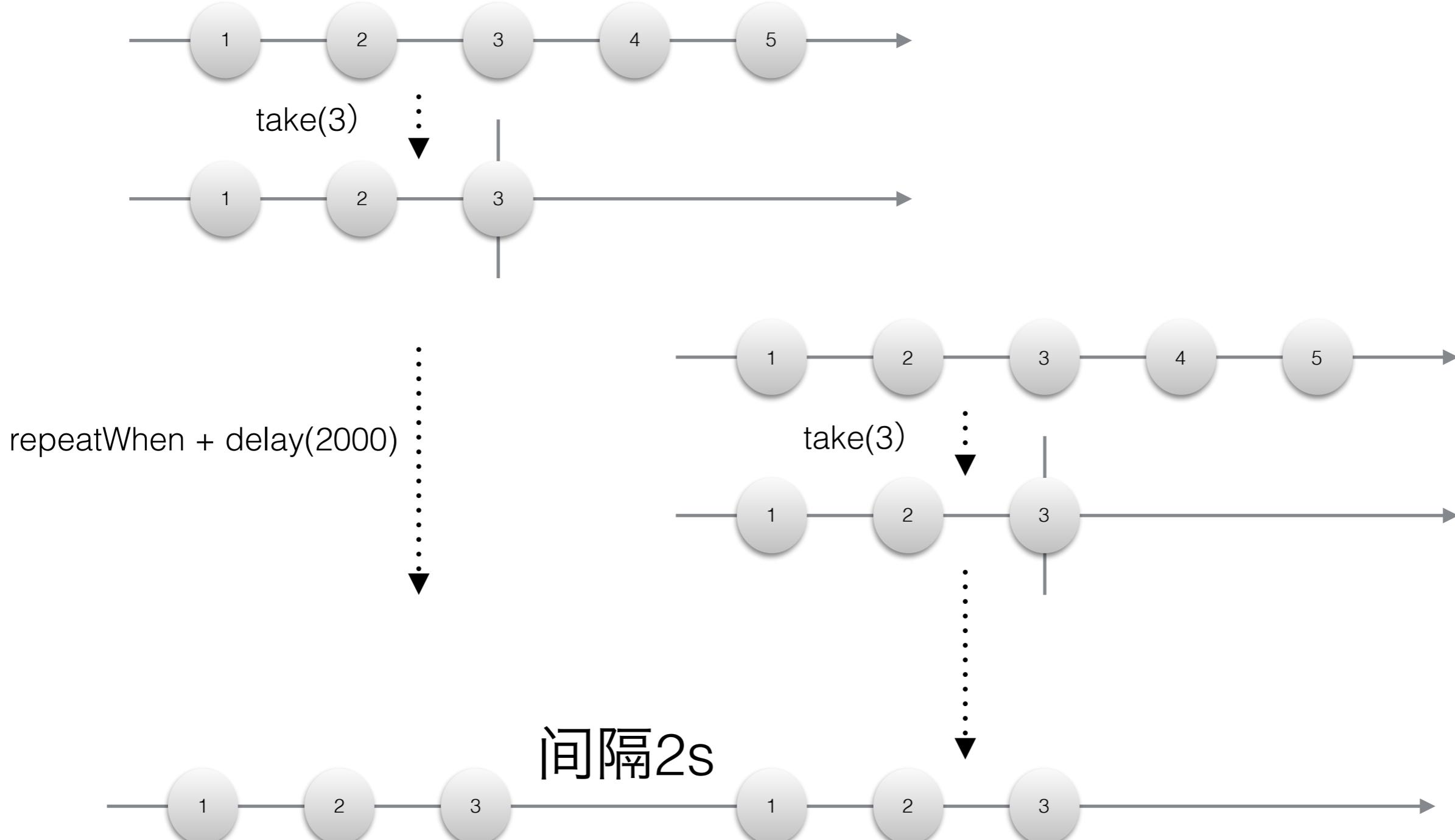
// sync way
const notifier = () => {
  return interval(1000);
}

const ofSource$ = of(1,2,3);
const repeatWhenSource$ = ofSource$.pipe(
  repeatWhen(notifier)
).subscribe(value => {
  console.log(value);
})

// async way
const notifeier2 = (notification$) => {
  return notification$.pipe(delay(2000));
}
const asyncSource$ = interval(1000).pipe(
  take(3)
);
const repeatSource2$ = asyncSource$.pipe(
  repeatWhen(notifeier2)
).subscribe(value => {
  console.log(value);
})
```



# async 方式下的repeatWhen



# defer 操作符

- 以可控的方式在期待的时间点创建 Observable
- 相当于对所渴求的 Observable 进行代理
- 对于同步数据可能没啥大用处
- 但是对于需要从外部资源获得数据的情境，会节省资源

```
// defer 操作符
// sync
const deferSource1$ = defer(() => of(1,2,3));
deferSource1$.subscribe(value => {
  console.log(value);
});
// async
const deferSource2$ = defer(() =>
  ajax.getJSON(`https://api.github.com/repos/AngularInDepth/angularindepth`)
);
deferSource2$.subscribe(value => {
  console.log(value);
});
```



# 创建型操作符总结

- 产生同步数据流只需要考虑产生什么数据
- 产生异步数据流需要考虑产生什么数据
- 产生异步数据流还需要考虑产生数据的时间间隔
- 创建数据流是搭建 RxJS 数据管道这座摩天大楼的基础

# 合并型操作符



# 合并数据流

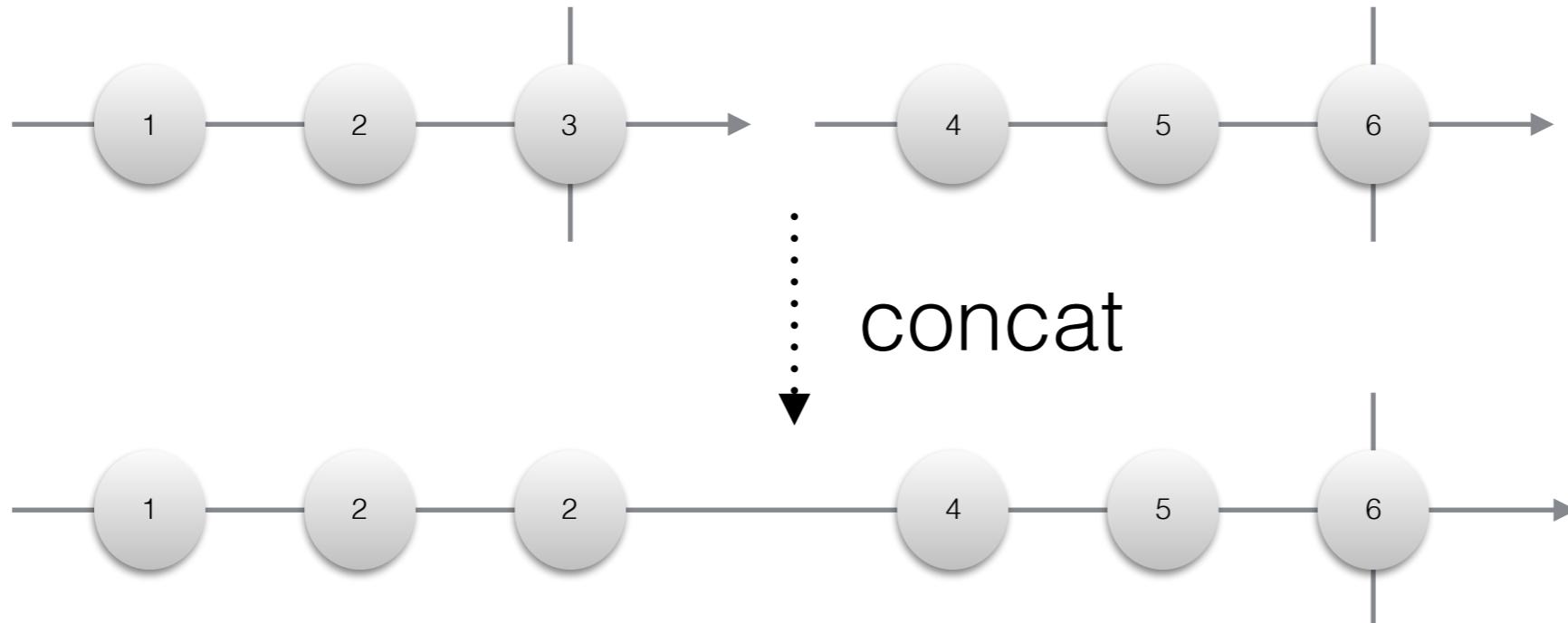
- concat 和 concatAll 操作符
- merge 和 mergeAll 操作符
- zip 和 zipAll 操作符
- combineLatest, combineAll 和 withLatestFrom 操作符
- race 操作符
- startWith 操作符
- forkJoin 操作符
- switch 和 exhaust 操作符
- 高阶 Observable

# 合并的场景

- 合并的场景多种多样
- 可能是有主次关系的合并
- 可能是平等的合并
- 可能是遵循某种规则的合并
- 但是最终都是合并数据流

# concat 操作符

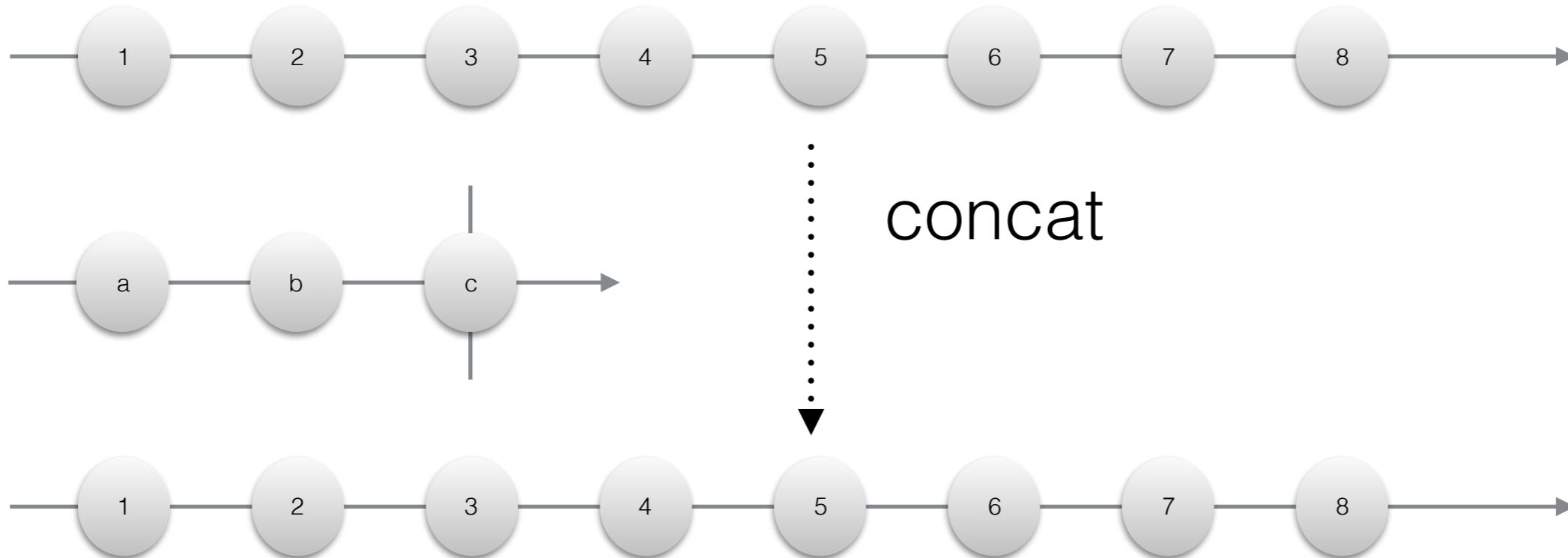
- 将流收尾相连
- 和 JS 中的 concat 方法类似
- 可以作为实例方法，接受上游数据流
- 可以作为静态方法，将数据流作为参数传入方法
- 按照顺序，前一个 observable 完成了再订阅下一个 observable 并发出值。
- 上游如果不完结则不会对下游进行订阅



```
// concat 操作符
// 作为实例操作符
const sourceOne$ = of(1, 2, 3);
// 发出 4,5,6
const sourceTwo$ = of(4, 5, 6);
// 先发出 sourceOne 的值, 当完成时订阅 sourceTwo
const example = sourceOne$.pipe(concat(sourceTwo$));
// 输出: 1,2,3,4,5,6
const subscribe = example.subscribe(val =>
  console.log('Example: Basic concat:', val)
);
// 作为静态操作符 需要注意不能同时同名的引入静态和实例操作符

const sourceThree$ = of(1, 2, 3);
// 发出 4,5,6
const sourceFour$ = of(4, 5, 6);

// 作为静态方法使用
const exampleTwo$ = concat2(sourceOne$, sourceFour$);
// 输出: 1,2,3,4,5,6
const subscribe2 = exampleTwo$.subscribe(val => console.log(val));
```



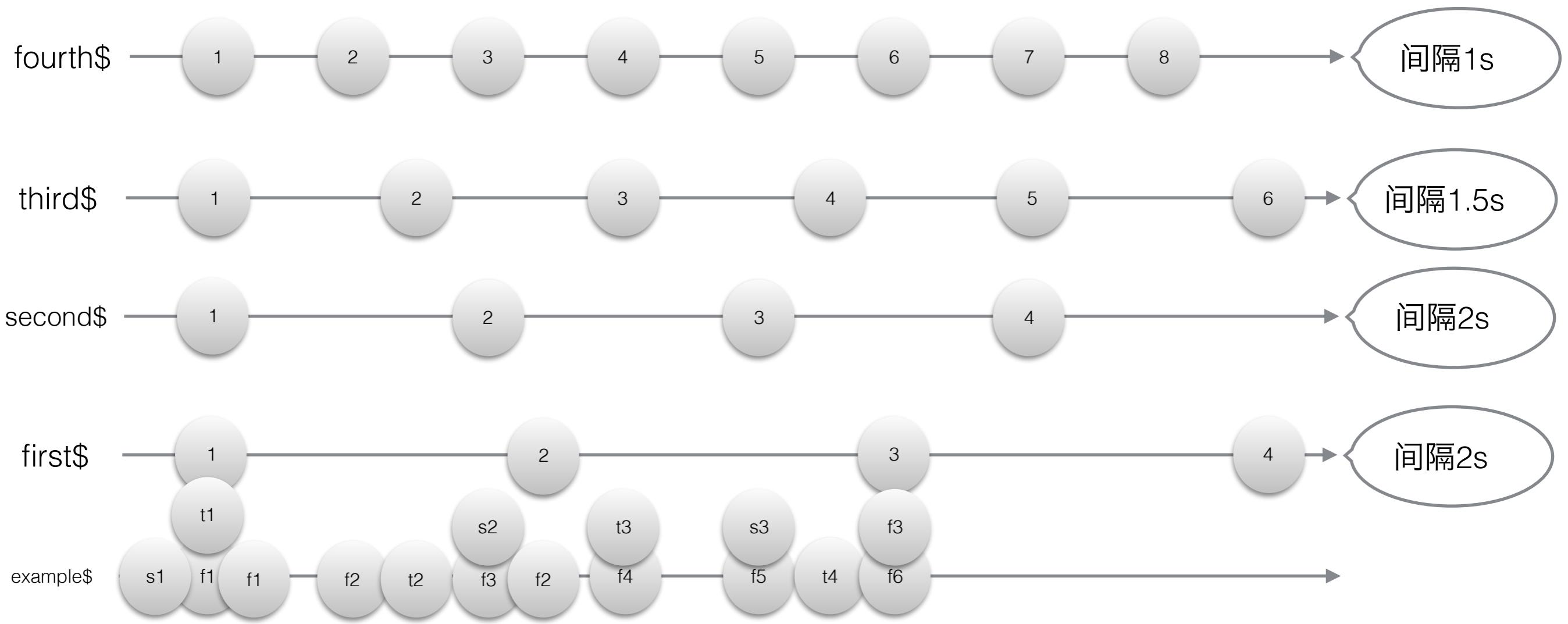
因为上游的内容不会进入complete状态， concat并不会取消订阅上游转而订阅下游的数据流

```
// 对 永不完结的 source 进行 concat

const unlimitedSource$ = concat2(interval(1000), of(['a', 'b', 'c']));
unlimitedSource$.subscribe(value => {
  console.log(value);
})
```

# merge 操作符

- 包含静态和实例两种形态
- 订阅上游的所有 Observable
- 先到先得策略
- 上游的 Observable 若没有全部 complete , 则 merge 后的数据流也没有 complete 状态
- 上游数据流之间的状态不会互相影响
- Merge 可以接受  $> 2$  的数据流
- 最后一个参数如果传入数值类型, 则会限定为 concurrent 参数, 限制同时进行 merge 的数据流
- 被限制的流在之前的流释放出并行空间前不会进入 merge



```
// merge 操作符

const first$ = interval(2500);
// 每2秒发出值
const second$ = interval(2000);
// 每1.5秒发出值
const third$ = interval(1500);
// 每1秒发出值
const fourth$ = interval(1000);

// 从一个 observable 中发出输出值
const example$ = merge(
  first$.pipe(mapTo('FIRST!')),
  second$.pipe(mapTo('SECOND!')),
  third$.pipe(mapTo('THIRD!')),
  fourth$.pipe(mapTo('FOURTH'))
);
// 输出: "FOURTH", "THIRD", "SECOND!", "FOURTH", "FIRST!", "THIRD", "FOURTH"
const subscribe = example$.subscribe(val => console.log(val));
```

上游的数据流并没有进入 complete 状态，故而 merge 之后的数据流也不会进入 complete 状态

如果上游的数据流都是同步产生的呢？  
比如上游分别是

`of(1,2,3)`  
`of(4,5,6)`

那么产生的数据流会是什么样的呢

## merge 的应用场景

- 对于移动端场景中，可能对同一个 DOM 节点并不仅仅监听一个事件
- fromEvent操作符只能一次监听一个事件
- 使用 merge 操作符将不同的 fromEvent 数据流合并成一个并使用统一的 eventHandler 进行处理会更轻松

# 思考一下

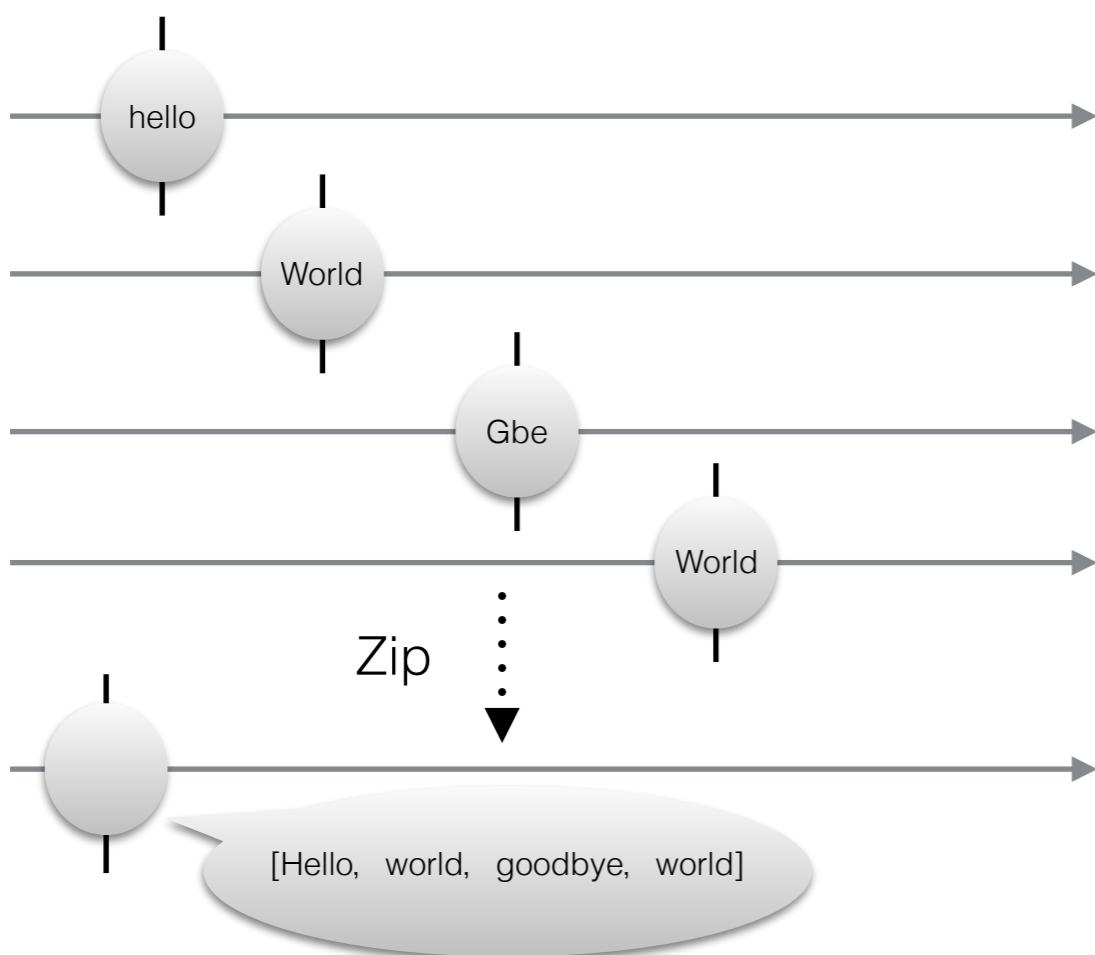
```
// little test

console.log('start');

const obSource1$ = of(1,2,3);
const obSource2$ = interval(1000).pipe(take(2),map(val => 'interval:' + val));
const obSource3$ = timer(0,1000).pipe(take(4),map(val => 'timer:' + val));
obSource1$.subscribe(val => {
  console.log(val);
});
obSource2$.subscribe(val => {
  console.log(val);
});
obSource3$.subscribe(val => {
  console.log(val);
})
setTimeout(() => {
  console.log('setTimeout')
},0);
Promise.resolve().then(function() {
  console.log('promise');
});
console.log('end');
```

## zip 操作符

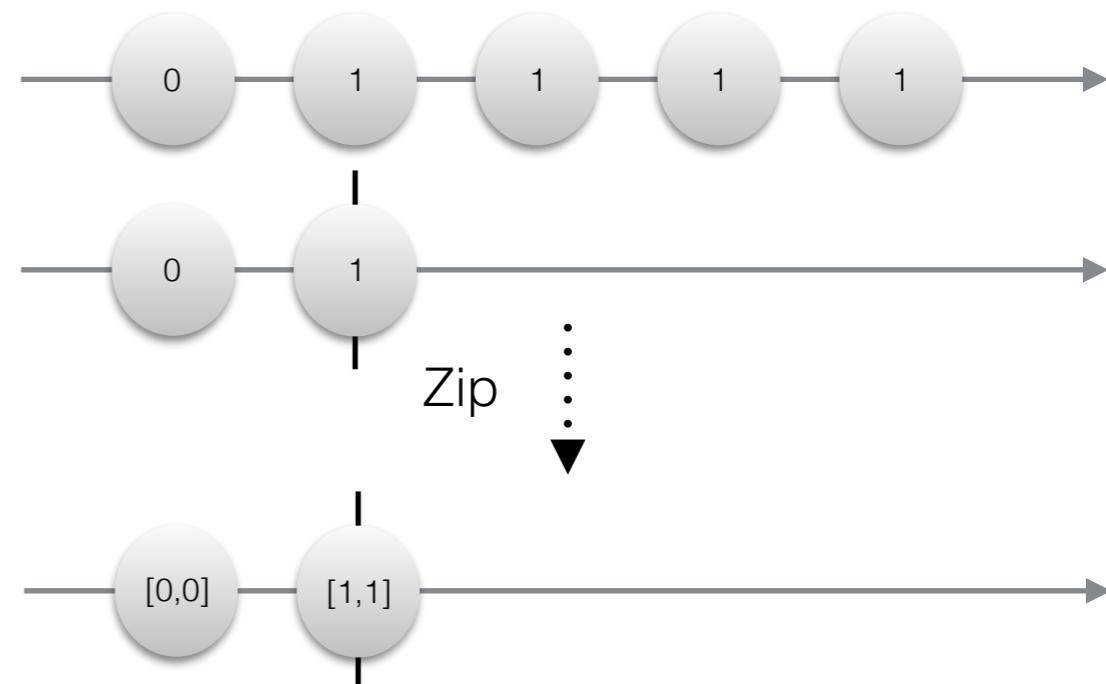
- 对上游的数据进行一对一的配对，将他们的值组成数组进行输出
- 当上游某一条数据流达成了 complete 状态则退订所有上游的订阅并 complete zip 出的数据流
- 支持多数据流共同zip，此时 上游数据流emit 数据最少那一条决定了zip产生的数据个数



```
// zip 操作符

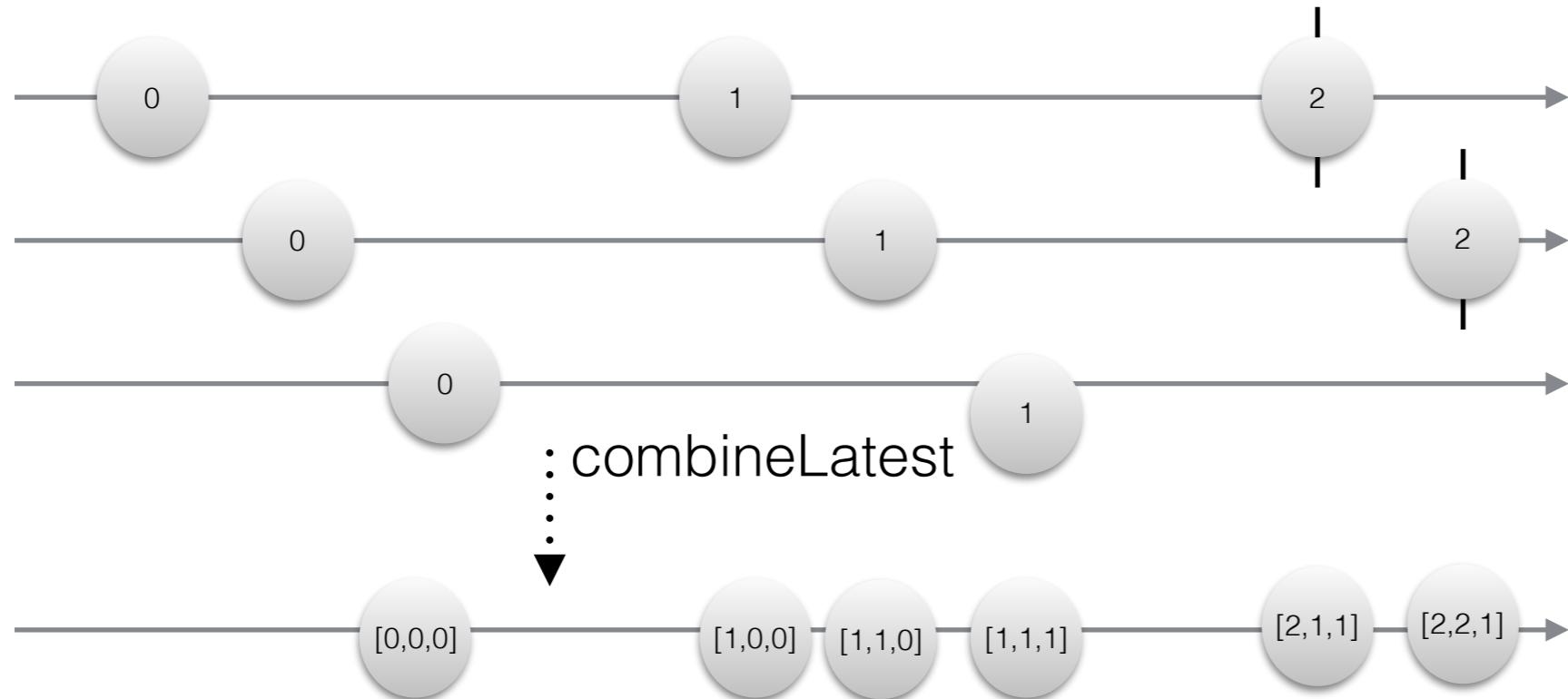
const sourceOne$ = of('Hello');
const sourceTwo$ = of('World!');
const sourceThree$ = of('Goodbye');
const sourceFour$ = of('World!');
// 一直等到所有 observables 都发出一个值，才将所有值作为数组发出
const example$ = zip(
  sourceOne$,
  sourceTwo$.pipe(delay(1000)),
  sourceThree$.pipe(delay(2000)),
  sourceFour$.pipe(delay(3000))
).subscribe(val => console.log(val));
// 输出: ["Hello", "World!", "Goodbye", "World!"]
```

```
const sourceFive$ = interval(1000);
const zipExample$ = zip(
  sourceFive$,
  sourceFive$.pipe(take(2))
).subscribe(val => console.log(val));
```



## combineLatest 操作符

- combine + latest
- 当上游任何一个数据流 emit 一个新的数据时，从所有上游输入的 Observable 对象中拿去最后一次产生的数据（最新数据）并将之组合。
- 想象一个多功能时钟，可以展示当前温度，湿度，时间等信息，每当任何一个数据发生变化时，则时钟会展示出当前的最新数据组合。
- combineLatest 直到每个上游 Observable 都至少发出一个值后才会发出初始值。
- 既有静态方法，又有实例方法
- 接受最后一个参数作为 projection 函数，相当于 pipe(map()) 方法



```
// combineLatest 操作符

const timerSource1$ = timer(1000, 4000);
const timerSource2$ = timer(2000, 4000);
const timerSource3$ = timer(3000, 4000);

const example$ = combineLatest(timerSource1$, timerSource2$, timerSource3$)
  .subscribe(val => {
    const [timerValOne, timerValTwo, timerValThree] = val;
    console.log(
      `Timer One Latest: ${timerValOne},
      Timer Two Latest: ${timerValTwo},
      Timer Three Latest: ${timerValThree}`);
  });
}

const example2$ = combineLatest(
  timerSource1$,
  timerSource2$,
  timerSource3$,
  (one, two, three) => {
    return `Timer One (Proj) Latest: ${one},
           Timer Two (Proj) Latest: ${two},
           Timer Three (Proj) Latest: ${three}`;
  }
).subscribe(val => {
  console.log(val);
})
```

# 测试一下

```
// 测试一下

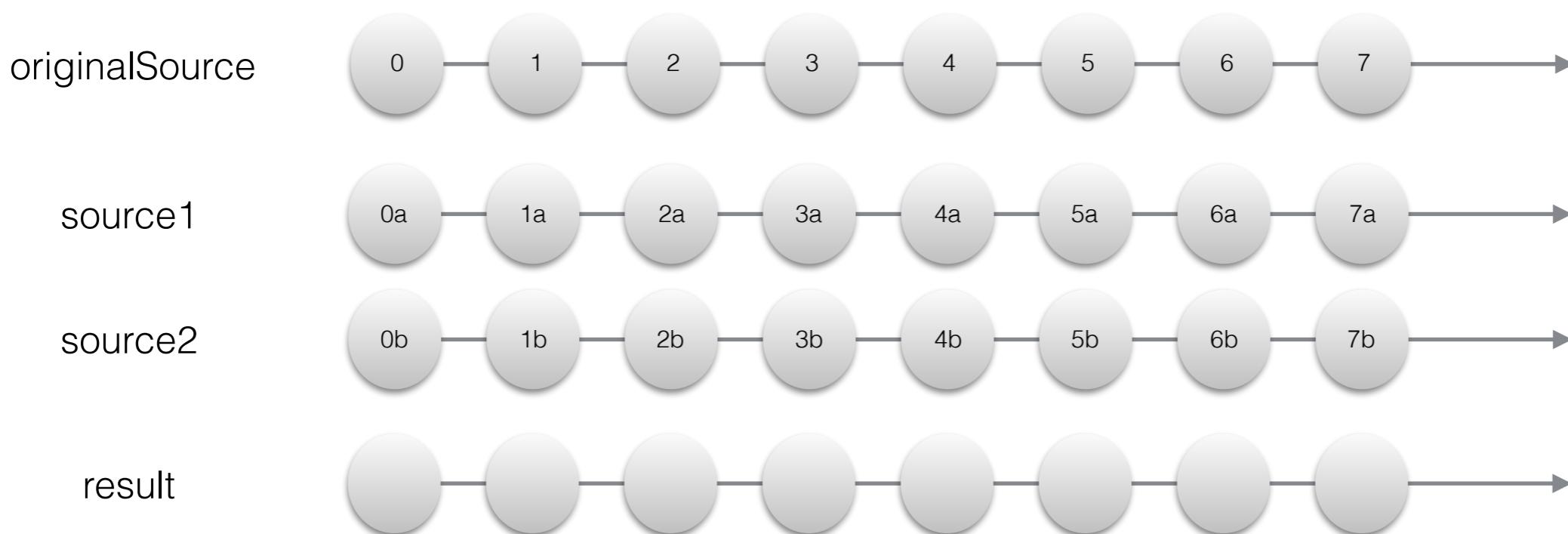
const source1$ = of(1,2,3);
const source2$ = of('a','b','c');
const source3$ = of('x','y');

const result$ = source1$.pipe(
  combineLatest2(source2$,source3$)
).subscribe(val => {
  console.log(val);
})
```

# 注意一个缺陷

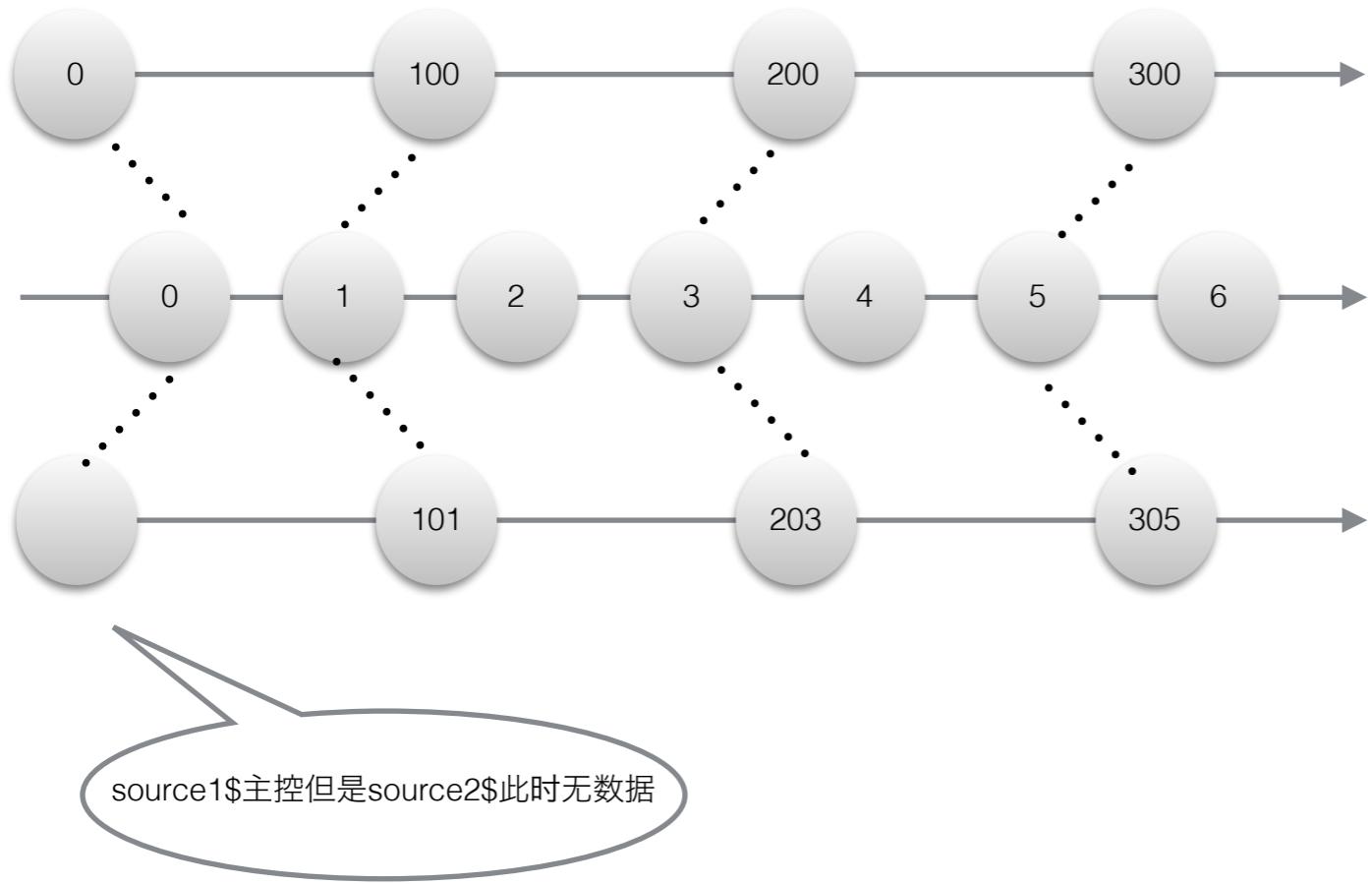
```
// 可能的缺陷

const originalSource$ = interval(1000);
const source1$ = originalSource$.pipe(
  map(val => val + 'a')
);
const source2$ = originalSource$.pipe(
  map(val => val + 'b')
);
combineLatest(source1$,source2$).subscribe(
  val => {
    console.log(val);
  }
)
```



## 引入 withLatestFrom 操作符

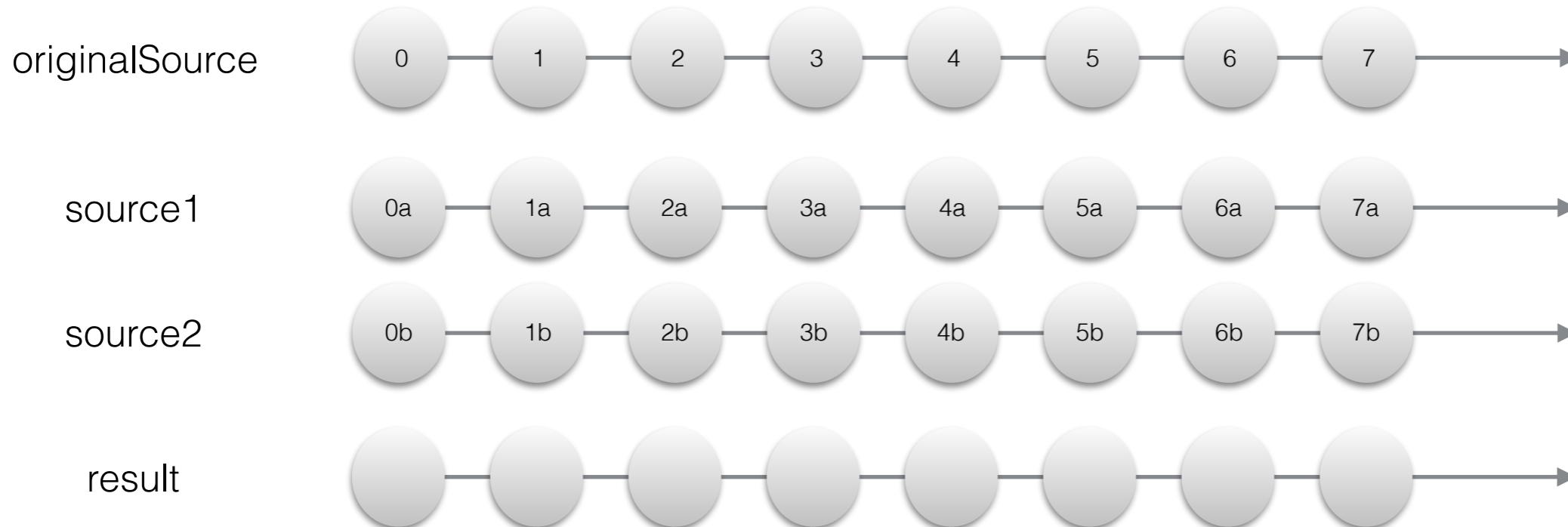
- 只有实例版本
- 功能与 combineLatest 相类似，但是有具象的区别
- 下游的数据流只由上游的数据流驱动而不再有作为参数的数据流掺和进控制部分。
- 换句话说，作为参数的数据流只能提供数据，而不再控制下游数据的节奏



```
// withLatestFrom 操作符

const source1$ = timer(0,2000).pipe(
  map(val => val * 100)
);
const source2$ = timer(500,1000);
// 此处最后一个参数为 projection 函数 等于 pipe(map())
source1$.pipe(
  withLatestFrom(source2$, (a,b) => a + b)
).subscribe(val => {
  console.log(val);
})
```

# 解决combineLatest 带来的错误



```
// 解决 combineLatest 带来的bug

const originalSource$ = interval(1000);
const source1$ = originalSource$.pipe(map(val => val + 'a'));
const source2$ = originalSource$.pipe(map(val => val + 'b'));

source1$.pipe(
  withLatestFrom(source2$)
).subscribe(val => {
  console.log(val);
});
```

# combineLatest 与 withLatestFrom

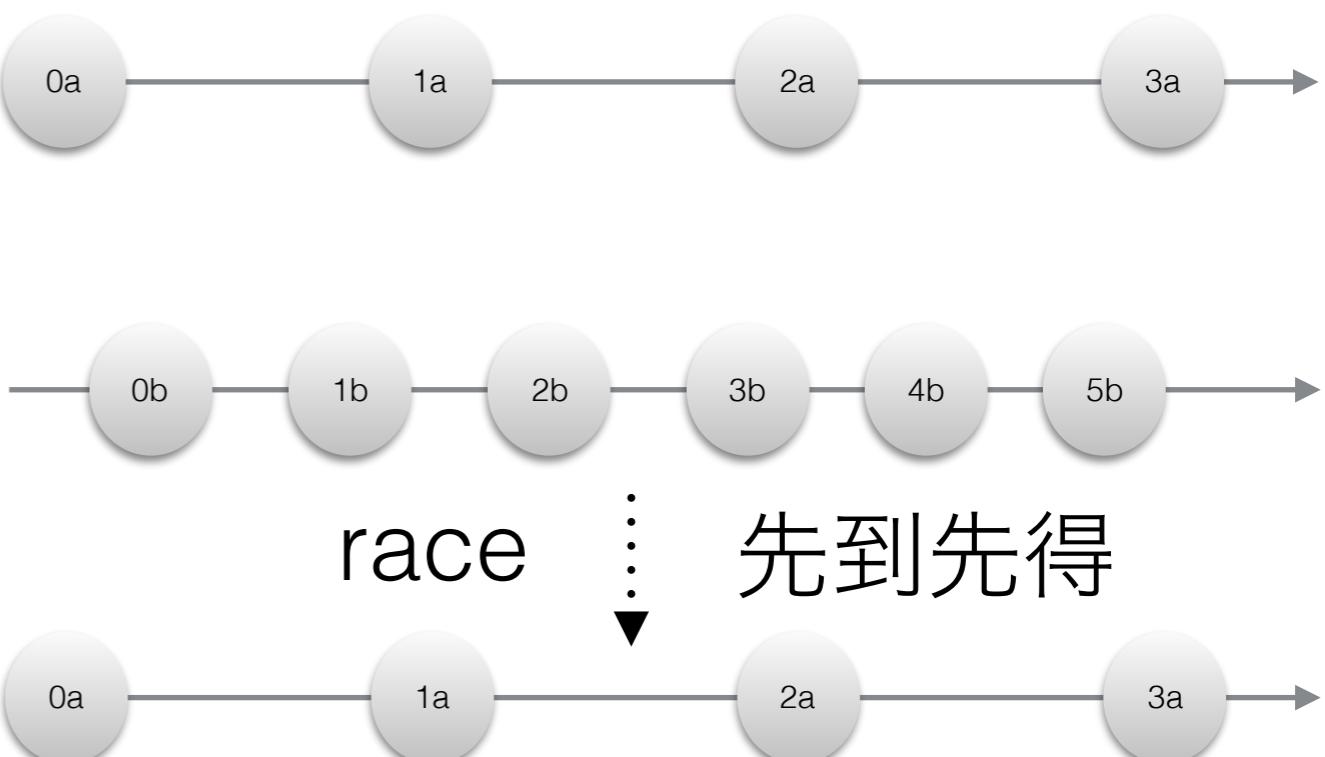
- 对完全独立的 Observable 数据流进行合并，使用 combineLatest 即可
- 如果输入的 Observable 数据流之间存在依赖关系，则 使用 withLatestFrom
- 将一个 Observable 对象 映射 成新的数据流，同时要从 其他 Observable 对象中获取 最新的数据，则使用 withLatestFrom

# race 操作符

- 一百米赛跑，只有第一名有吃蛋糕的权利
- 上游哪一个数据流最先产生数据，其他数据流就会被退订，失去吃蛋糕的权利
- 既有静态形态也有实例形态

```
// race 操作符
const source1$ = timer(0,2000).pipe(map(val => val + 'a'));
const source2$ = timer(500,1000).pipe(map(val => val + 'b'));

race(source1$,source2$).subscribe(val => {
  console.log(val);
})
```



# startWith 操作符

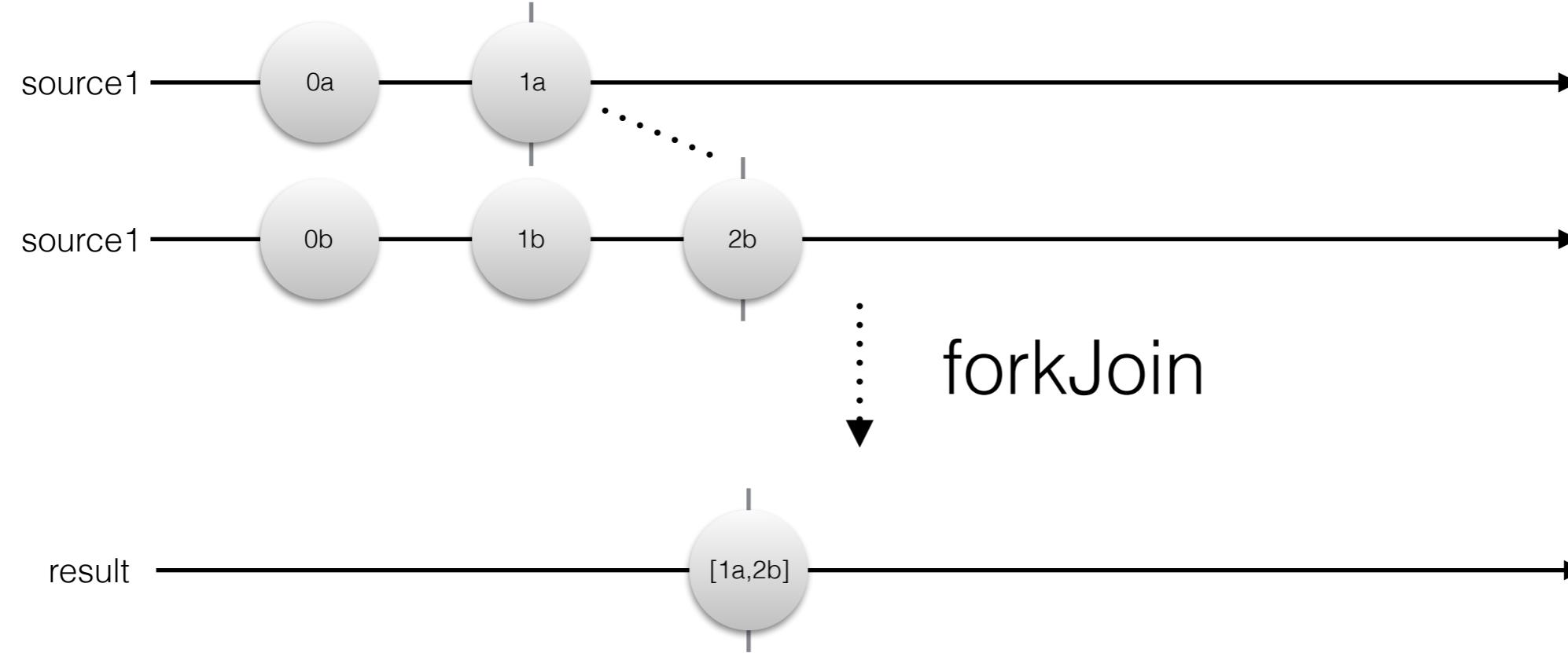
- 只有实例形态
- 在被订阅的那一刻 emit startWith 的参数
- 参数都是同步被 emit 的，如果需要异步地 emit 数据则需要使用 concat
- 某种意义上可以被concat 替换，但是 startWith 提供了更好的链式调用可能

```
// startWith 操作符

const source1$ = timer(0,1000);
source1$.pipe(startWith('i am start')).subscribe(val => console.log(val));
// 上述等同于 of('i am start').concat(source1$)
```

# forkJoin 操作符

- 只有静态形式
- 等到所有输入 Observable 全部完结之后才会将生成的数据传递给下游
- 将所有输入 Observable 的对象都完结后，确定不会有新的数据产生的时候，forkJoin 会将所有输入 Observable 对象产生的最后一个数据合并成给下游的唯一数据
- 换句话说就是 Observable 版本的 Promise.All()



```

// forkJoin 操作符
const source1$ = interval(1000).pipe(
  take(2),
  map(val => val + 'a')
);

const source2$ = interval(1000).pipe(
  take(3),
  map(val => val + 'b')
);

forkJoin(source1$,source2$).subscribe(
  val => console.log(val)
)

```

# 再见之前

[代码地址](#)

[玩具地址](#)

每次input的数值更新后，需要注意的是switchMap会取消之前的订阅并重新订阅新的数据流，你可以当做下面的 marble 图内容在 input 属性值更新后重新来过，此处以 currentnumber = 0, input 属性更新为 endRange = 10 为例

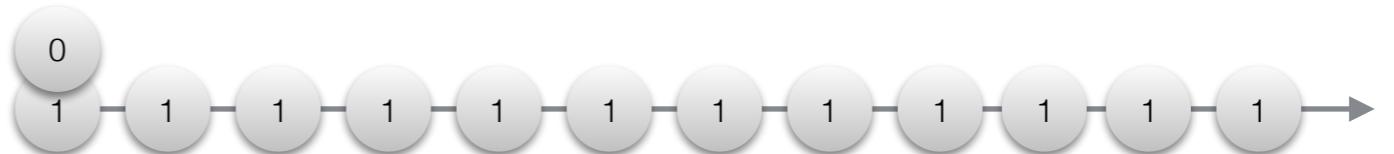
timer(0, this.countInterval), timer时间间隔为20ms



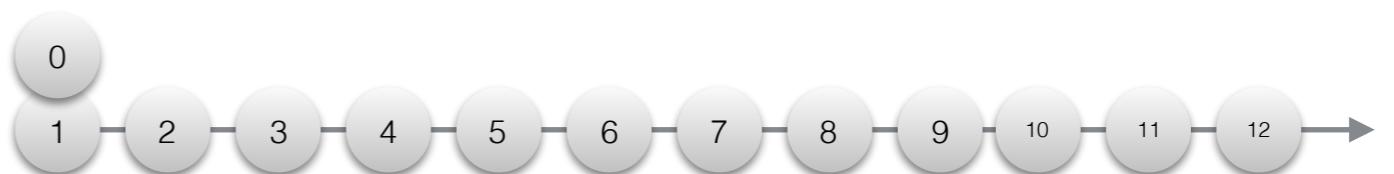
mapTo(this.positiveOrNegative(endRange, this.currentNumber))



startWith(this.currentNumber)



scan((acc: number, curr: number) => acc + curr),



takeWhile(this.isApproachingRange(endRange, this.currentNumber))



⋮ subscribe  
↓

subscribe((val: number) => this.currentNumber = val);

数字就会以20ms为间隔变化了

# 高阶 Observable

下次再见

## concatAll 操作符

- 收集 observables, 当前一个完成时订阅下一个
- 主要用于打平高阶运算符
- 后续介绍了高阶运算符再介绍