

# RxJS直通车第三期



高阶Observable

辅助类操作符

过滤操作符

政法BG 刘灵辉

项目地址



**第一讲**  
**何为RxJS**  
**何为 Rx**

**项目地址**



**第二讲**  
**创建类操作符**  
**合并类操作符**

**项目地址**



**第三讲**  
**高阶Observable**  
**辅助操作符**  
**合并类操作符**

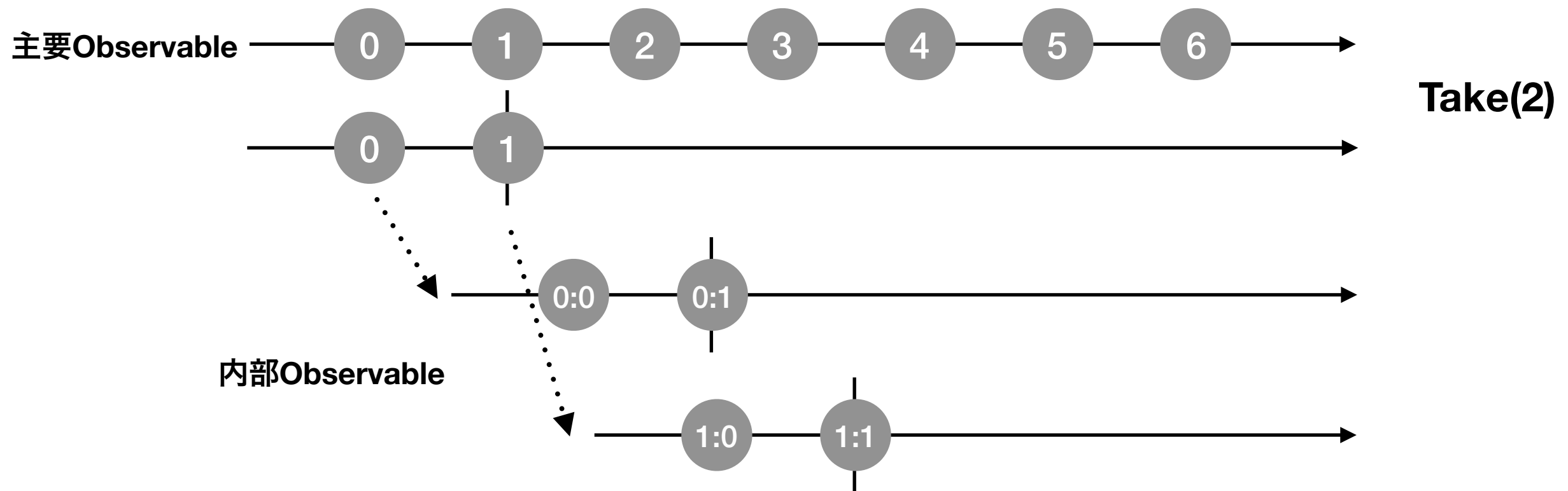
**项目地址**

**配合内容 实践代码**  
**就算没有想好某种技术的应用场景**  
**也能帮助你知晓其设计目的**

# 高阶Observable

- 高阶函数
  - 也是函数
  - 以其他函数作为参数
  - 返回值也是函数
- 低阶Observable（一阶Observable）
  - 之前介绍的都是低阶Observable
- 高阶Observable
  - 产生的数据流是Observable 的 Observable

# 高阶Observable



// 高阶 Observable

```
const source$ = interval(1000).pipe(  
  take(2),  
  map(x => {  
    return interval(1000).pipe(  
      map(y => {  
        return x + ':' + y  
      }),  
      take(2)  
    )  
  })  
)
```

```
const subscription = source$.subscribe(value => {  
  console.log(value);  
})
```

高阶 (外部) Observable的完结  
并不代表内部 Observable 的完结

外部和内部 Observable 的生命周期并不是相同的

# 高阶 Observable 的意义

使用数据流的概念来管理数据流

之前都是用数据流的概念管理数据

实际上可以使用数据流的概念管理数据流

**使用管理数据的方式管理多个 Observable 对象**

# 高阶 合并 操作符

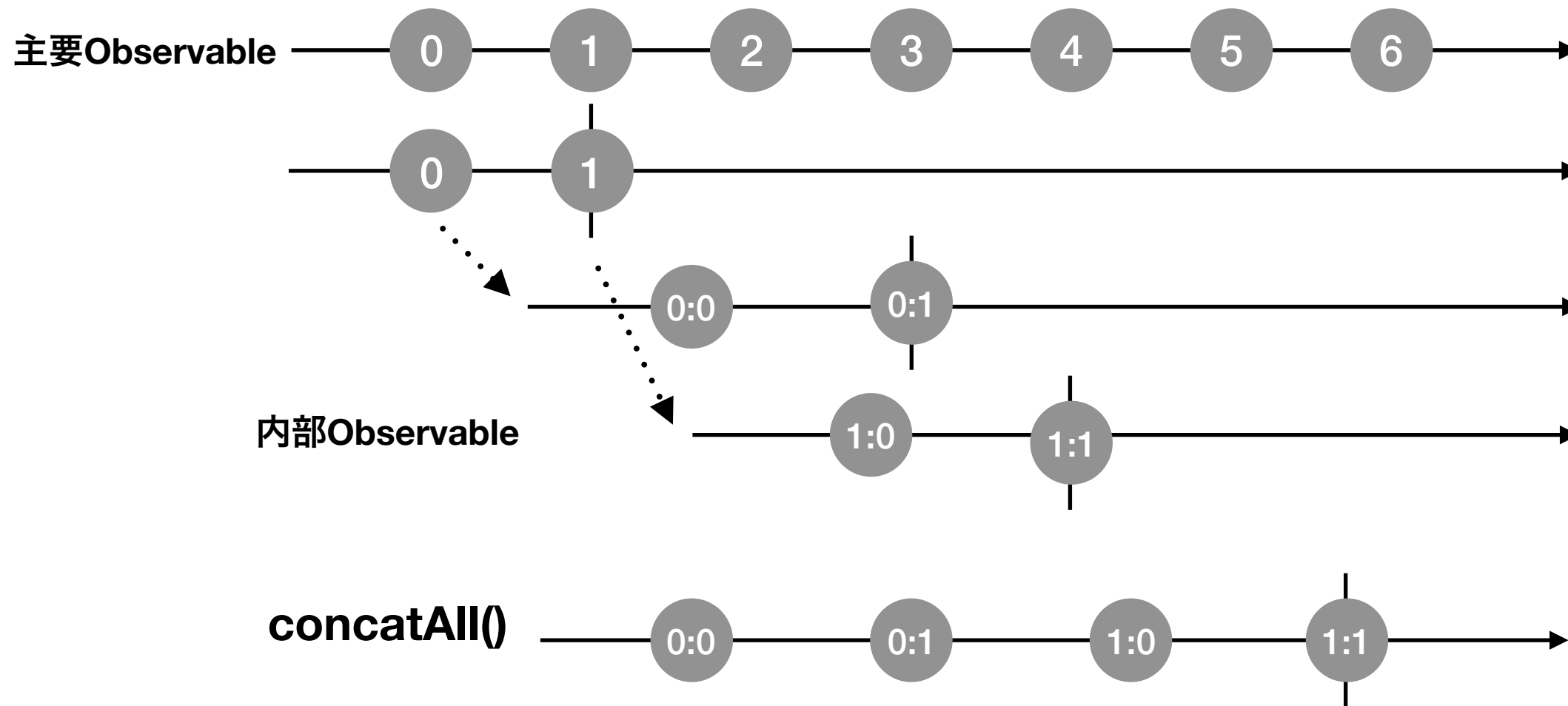
concatAll

mergeAll

zipAll

combineAll

# concatAll 操作符



## 逐个订阅内部 Observable

如果某个正在订阅的内部 Observable 对象没有完结

concatAll 将不会订阅下一个内部 Observable

**可能的内存泄露**

当concatAll 操作符消耗内部 Observable 的速度

跟不上外部 Observable 产生的速度时

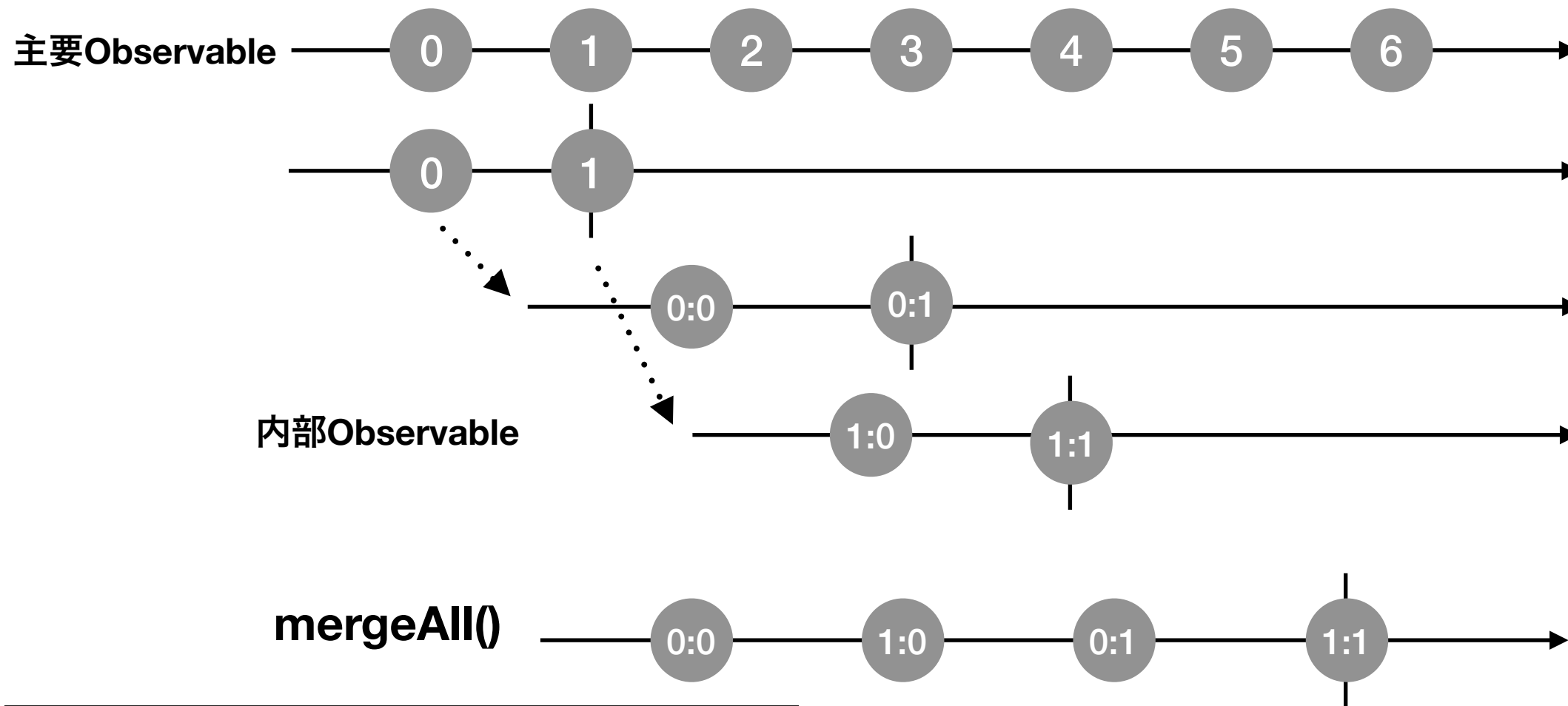
会导致数据积压

数据积压的堆栈过多就会产生内存泄漏

```
// concatAll 操作符
console.clear();
const source$ = interval(100).pipe(
  take(2),
  map(x => interval(1500).pipe(
    map(y => x + ':' + y),
    take(2)
  ))
).pipe(
  concatAll()
);

const subscription = source$.subscribe(value => {
  console.log(value);
});
```

# mergeAll 操作符



```
// mergeAll 操作符
console.clear();
const source$ = interval(1000).pipe(
  take(2),
  map(x => interval(1500).pipe(
    map(y => x + ':' + y),
    take(2)
  ))
).pipe(
  mergeAll()
);

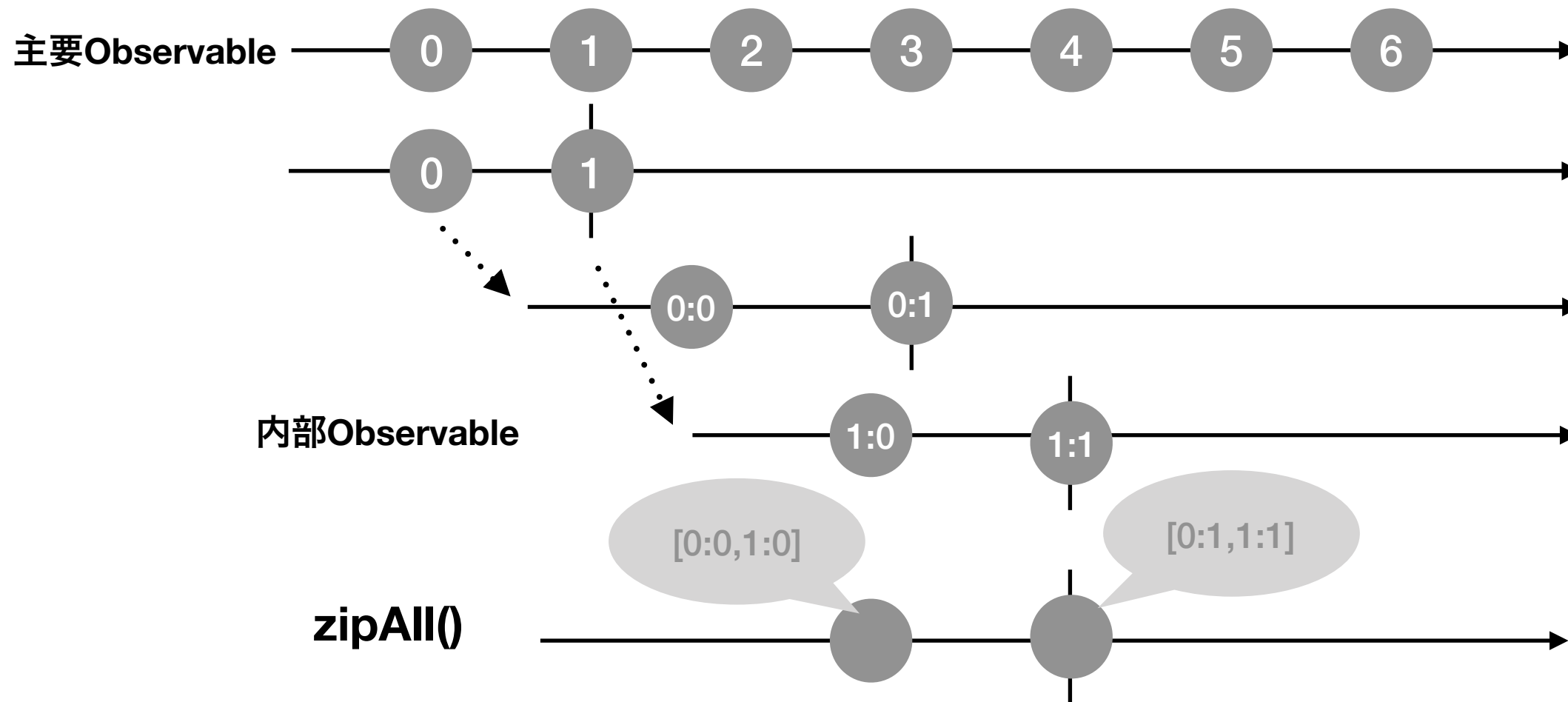
const subscription = source$.subscribe(value => {
  console.log(value);
});

subscription.unsubscribe();
```

与 concatAll 操作符不同  
只要外部(上游) Observable 产生一个内部  
Observable  
mergeAll 操作就会订阅新产生的内部  
Observable  
并按照自然顺序抽取数据



# zipAll 操作符



```
// zipAll 操作符
console.clear();
const source$ = interval(1000).pipe(
  take(2),
  map(x => interval(1500).pipe(
    map(y => x + ':' + y),
    take(2)
  ))
).pipe(
  zipAll()
);

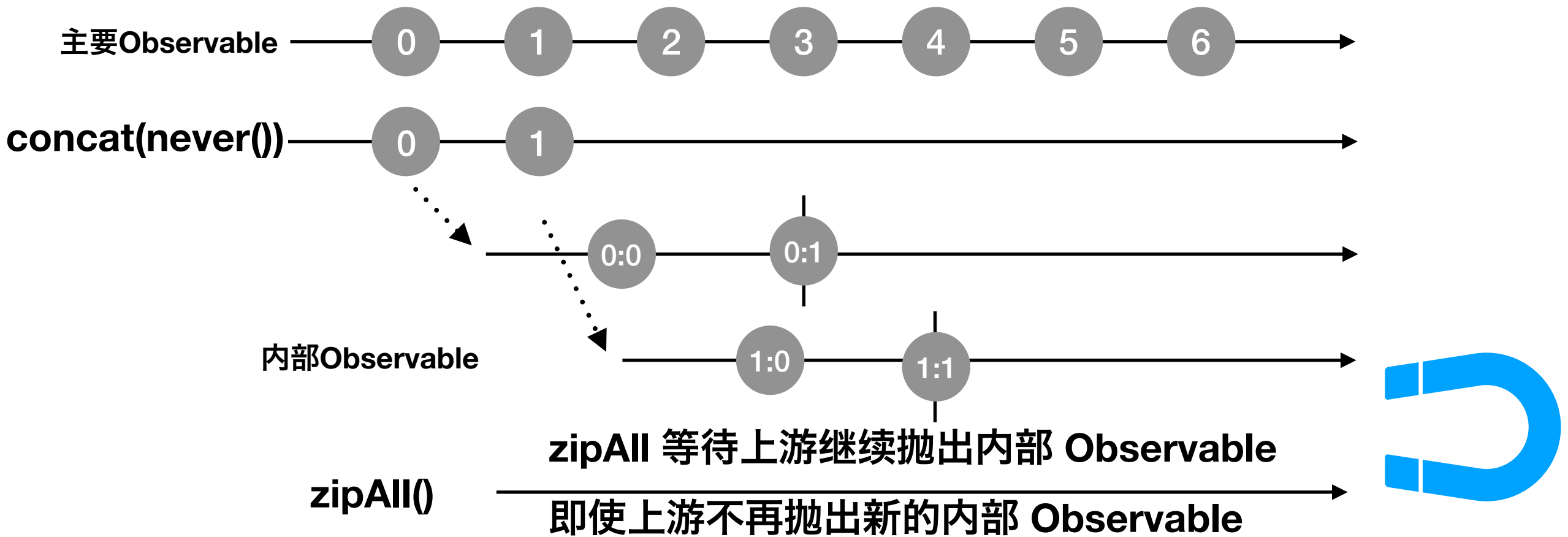
const subscription = source$.subscribe(value => {
  console.log(value);
});

subscription.unsubscribe();
```

上游 Observable 产生两个内部 Observable  
zipAll 将两个内部 Observable 缝合在一起

**需要注意的是**

如果上游 Observable 数据流不会完结  
则zipAll 操作符则不会执行

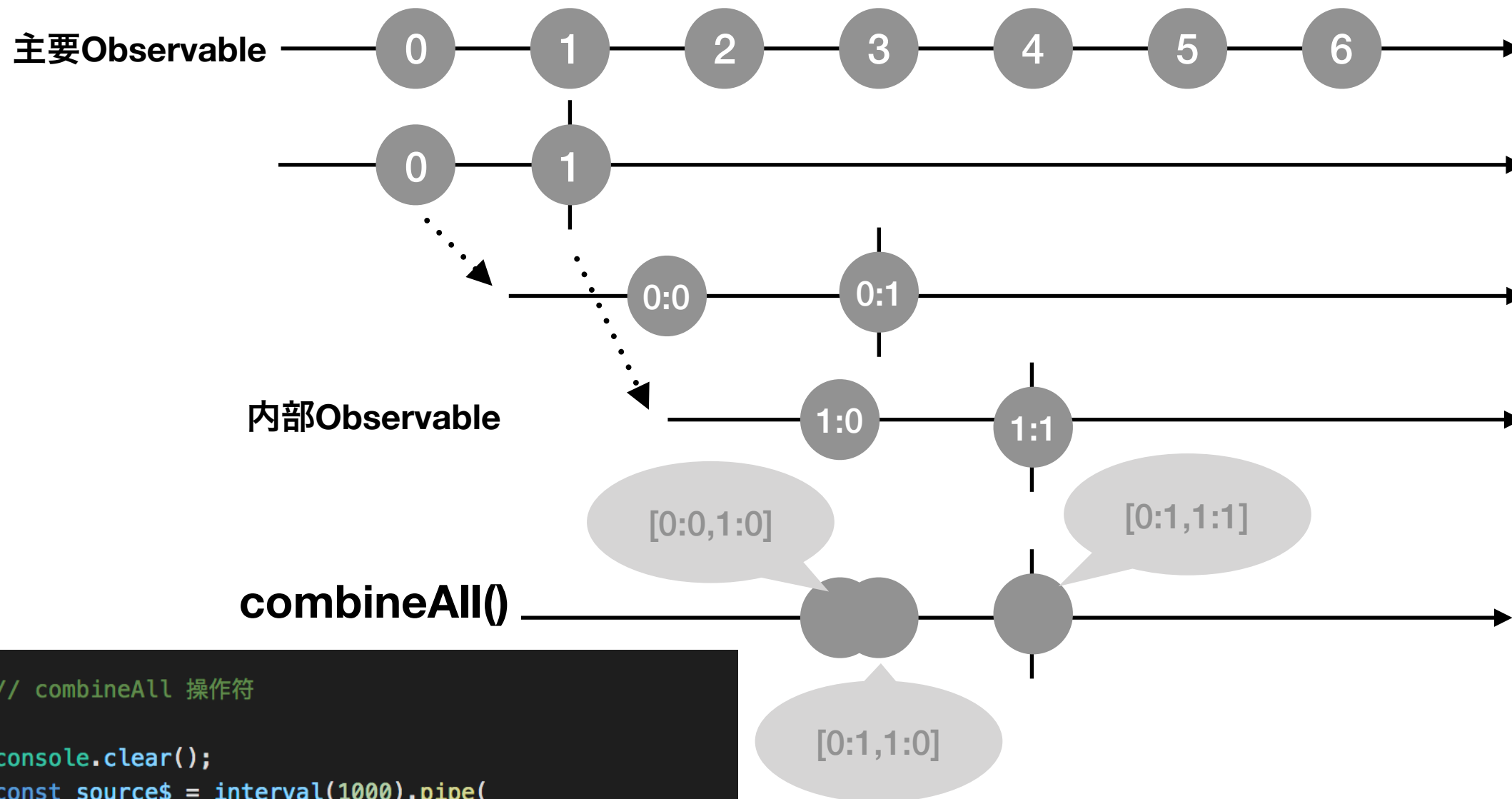


// zipAll 操作符等待上游 Observable 完结

```
const source$ = interval(1000).pipe(  
  take(2),  
  concat(never()),  
  map(x => interval(1500).pipe(  
    map(y => x + ':' + y),  
    take(2)  
  ))  
).pipe(  
  zipAll()  
);  
  
const subscription = source$.subscribe(value => {  
  console.log(value);  
});  
  
subscription.unsubscribe();
```

上游 Observable 链接了 never 数据流  
融合的上游 Observable 不会完结  
zipAll 不会被执行  
订阅的数据流没有任何数据

# combineAll 操作符



// combineAll 操作符

```
console.clear();
const source$ = interval(1000).pipe(
  take(2),
  map(x => interval(1500).pipe(
    map(y => x + ':' + y),
    take(2)
  ))
).pipe(
  combineAll()
);

const subscription = source$.subscribe(value => {
  console.log(value);
});

subscription.unsubscribe();
```

combineAll 与 combineLatest 类似

**注意**

上游 Observable 如果不会完结

combineAll 操作符则不会执行

# 为何没有 withLatestFromAll

- 对于 withLatestFrom 操作符
  - 只有一个输入 Observable 控制产生数据的节奏
  - 其余的 Observable 只提供数据
  - 数据流之间并不是平等的关系
- 如果存在 withLatestFromAll 操作符
  - 则需要保证第一个产生的内部Observable控制节奏
  - 后续产生的内部 Observable 仅提供数据
  - 虽然可以实现，但是对高阶 Observable 内部的内部 Observable 区分对待是不合理的

# 暂停一下

**高阶 Observable 所产生的 内部 Observable**

**可能是同步的 可能是异步的**

**如果上游产生的内部 Observable 数据过快**

**可能就会产生数据积压**

**数据积压就可能导致内存泄漏**

**如何解决可能存在的数据积压呢？**

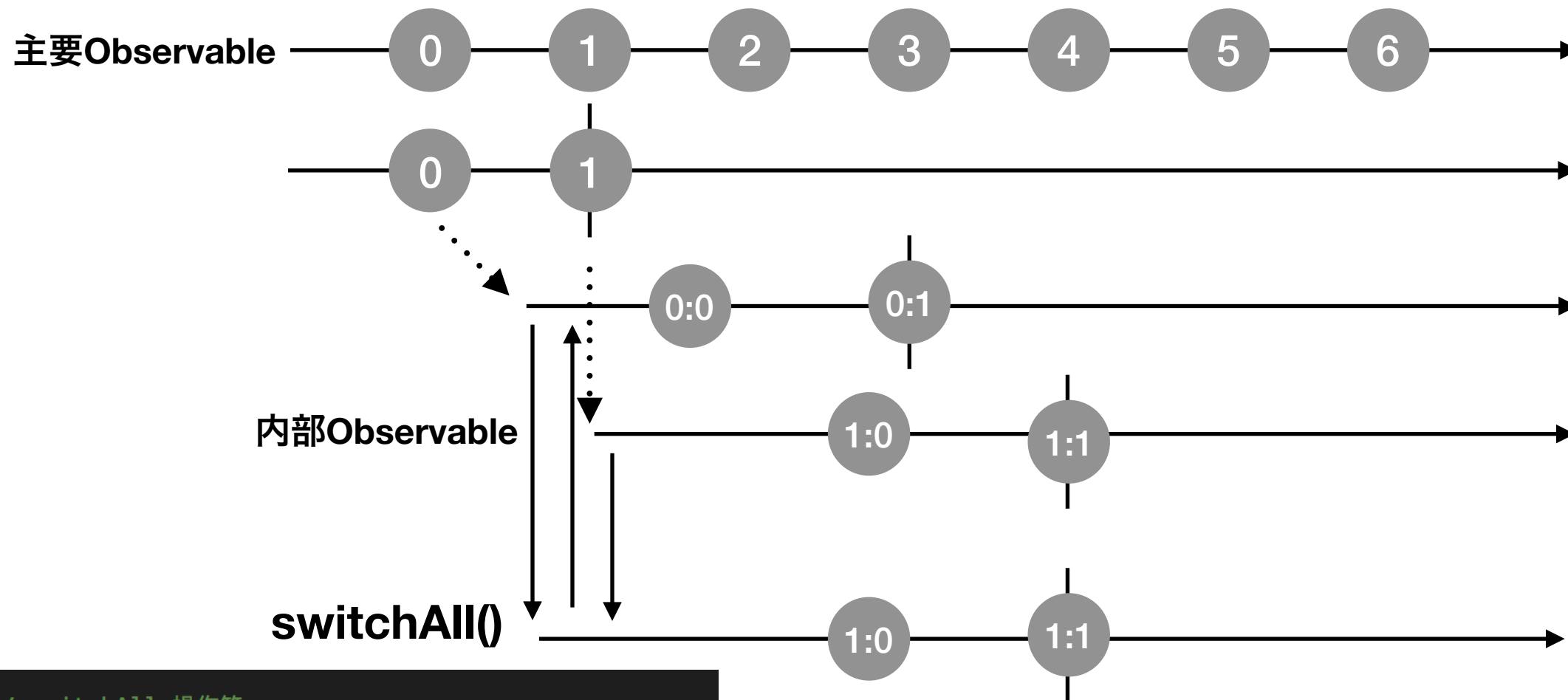
**某些情况下**

**并不需要无损的数据流连接**

**switchAll 操作符**

**exhaust 操作符**

# switchAll 操作符



当高阶 Observable 抛出一个新的内部 Observable

switchAll 操作符将取消

对于之前订阅的 内部Observable的订阅

并重新订阅新的内部 Observable

当上游高阶 Observable 完结

并

当前内部 Observable 完结

的时候 switchAll 操作符产生的数据流完结

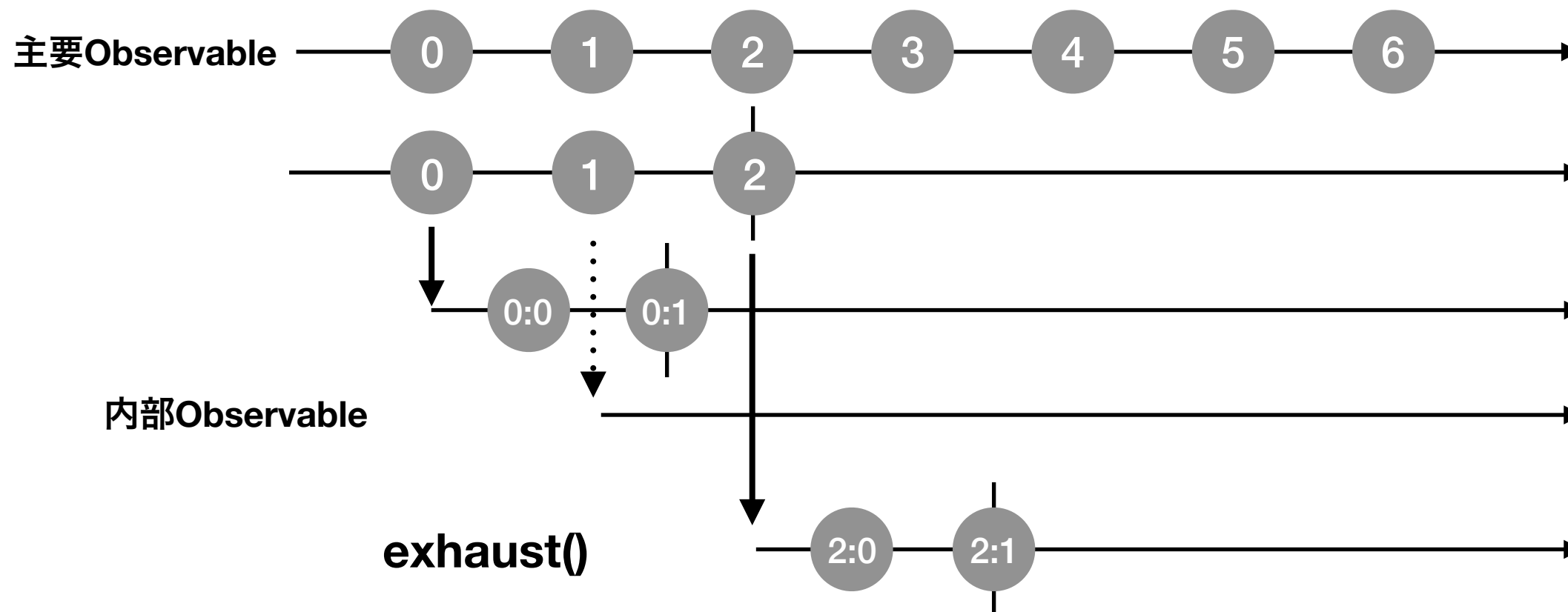
// switchAll 操作符

```
console.clear();
const source$ = interval(1000).pipe(
  take(2),
  map(x => interval(1500).pipe(
    map(y => x + ':' + y),
    take(2)
  ))
).pipe(
  switchAll()
);

const subscription = source$.subscribe(value => {
  console.log(value);
});

subscription.unsubscribe();
```

# exhaust 操作符



```
// exhaust 操作符
console.clear()
const source$ = interval(1000).pipe(
  take(3),
  map(x => interval(700).pipe(
    map(y => x + ':' + y),
    take(2)
  ))
).pipe(
  exhaust()
);

const subscription = source$.subscribe(value => {
  console.log(value);
})
```

在 `exhaust` 操作符的作用下

当上游 `Observable` 产生新的内部 `Observable` 时

操作符会根据 之前的内部 `Observable` 情况做出行动

当 先前的 内部 `Observable` **没有完结时**

则会继续保持对其的订阅

忽视新产生的 `Observable`

而如果先前的 内部 `Observable` **已经完结时**

则订阅新产生的 `Observable`

# 辅助操作符

count 操作符

max/min 操作符

reduce 操作符

every 操作符

find/findIndex 操作符

isEmpty 操作符

defaultIfEmpty 操作符



# 辅助类操作符-数学操作符

**count 操作符**

**max/min 操作符**

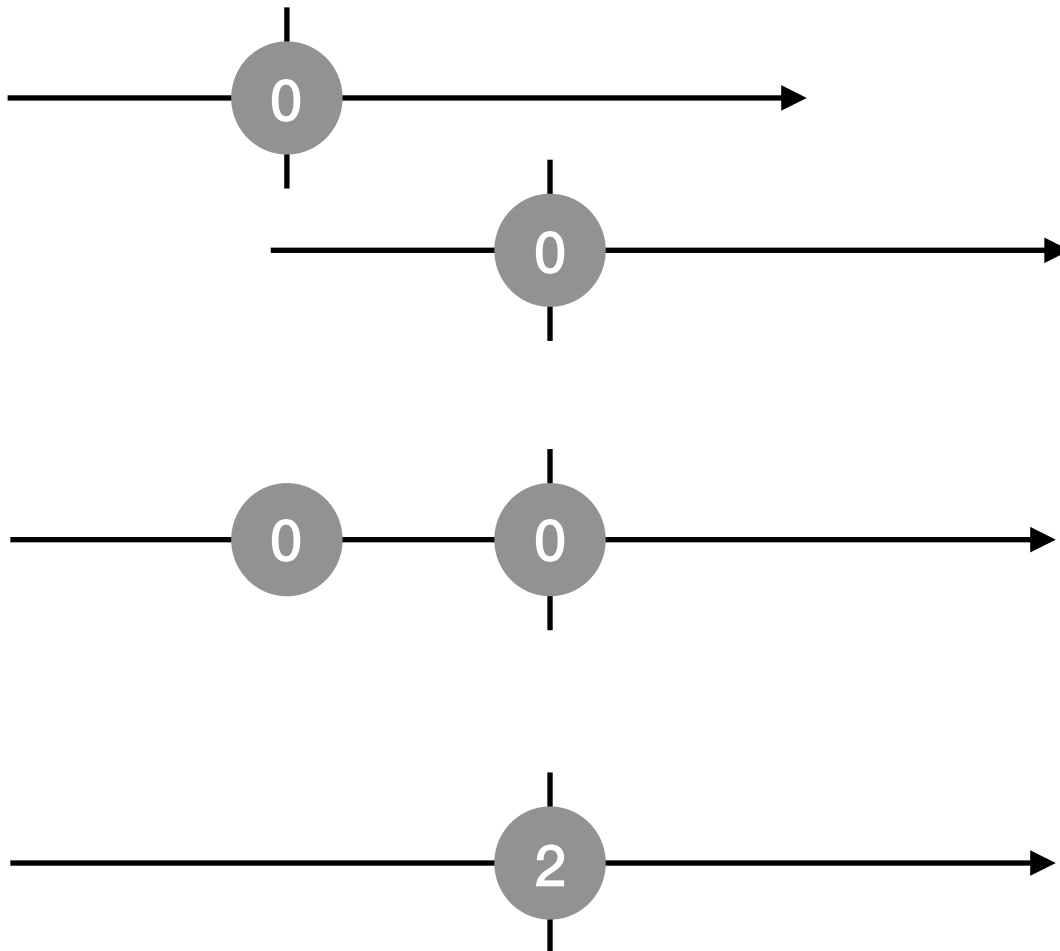
**reduce 操作符**

**以上均为实例操作符**

**只有在上游的数据流完结后才会**

**向下游传递数据**

# count 操作符



统计上游的 Observable 对象

抛出的数据的总数

只有当上游所有的 Observable 对象

完结之后 `count` 操作符

才能给出其结果

`count` 的计算无关数据流的同步异步

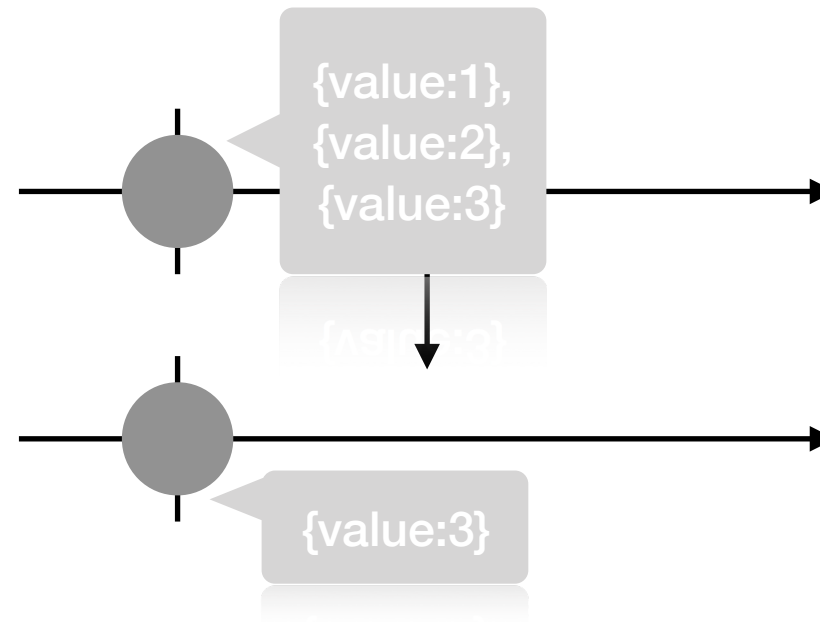
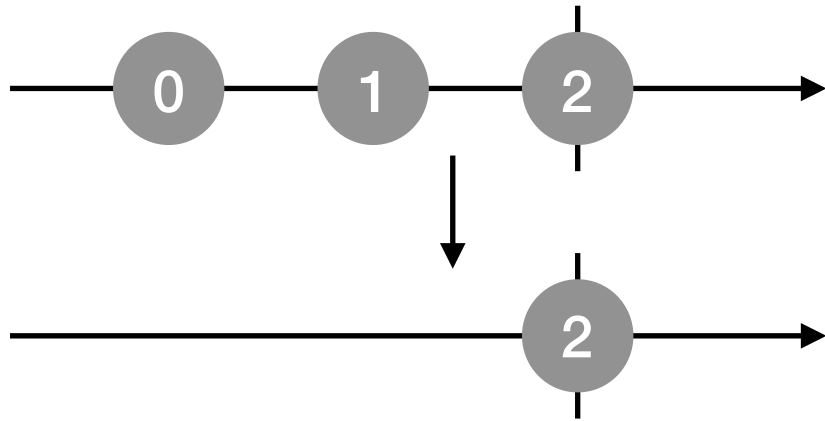
这也是合情合理的

`count()`

```
// count 操作符
console.clear();
const source$ = timer(1000).pipe(
  concat(timer(1000))
).pipe(count())

const subscription = source$.subscribe(value => {
  console.log(value);
})
```

# max/min 操作符



// max/min 操作符

```
console.clear();
const source$ = interval(1000).pipe(
  take(3),
  max()
);
```

```
const subscription = source$.subscribe(value => {
  console.log(value);
})
```

```
const source2$ = of(
  {value:1},
  {value:2},
  {value:3}
).pipe(max((x,y) => x.value - y.value));
```

```
const subscription2 = source2$.subscribe(value => {
  console.log(value);
})
```

如果是可以直接比较的数值类型

**可以直接使用操作符比较**

如果是不能直接进行比较的复杂数据

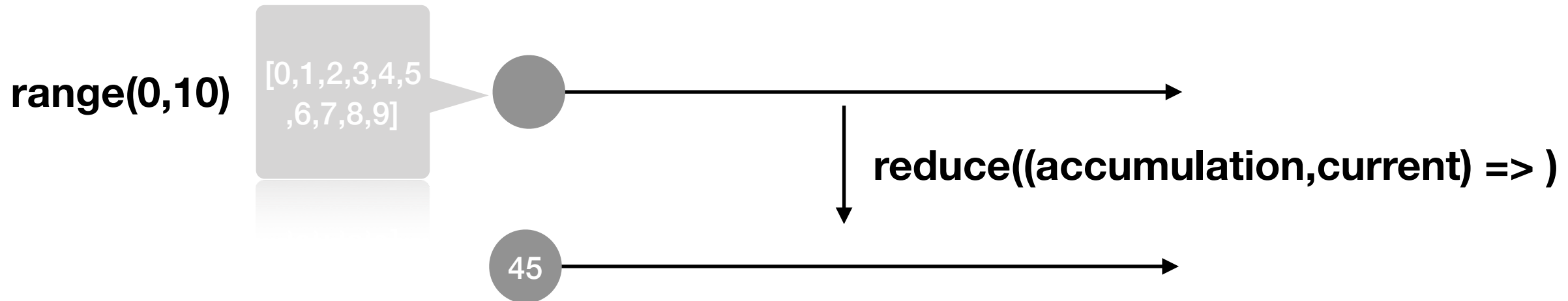
**则需要指定一个用于比较复杂数据的方法**

操作符只有当上游的数据流完结后

才会给出结果

上游的数据流是同步与异步均可

# reduce 操作符/累积操作符



```
// reduce 操作符
console.clear();

const source$ = range(0,10).pipe(
  reduce((acc,curr) => acc + curr, 0)
);

const subscription = source$.subscribe(value => {
  console.log(value);
})

// 使用 reduce 实现一个 average 操作符
```

请自行尝试使用reduce 操作符

创建一个求平均值的操作符

请自行尝试使用 reduce 操作符

实现之前的max/min 操作符

累积从头到尾的结果  
累积操作符接受两个参数  
一个作为累积函数  
一个作为初始值/种子值

累积函数接受两个参数  
一个是既往的累积值  
一个是当前值

js中对于既定数组可以执行 reduce 操作  
RxJS 对于数据流同样可以执行reduce 操作

# 条件 boolean 类操作符

**every 操作符**

**find 操作符**

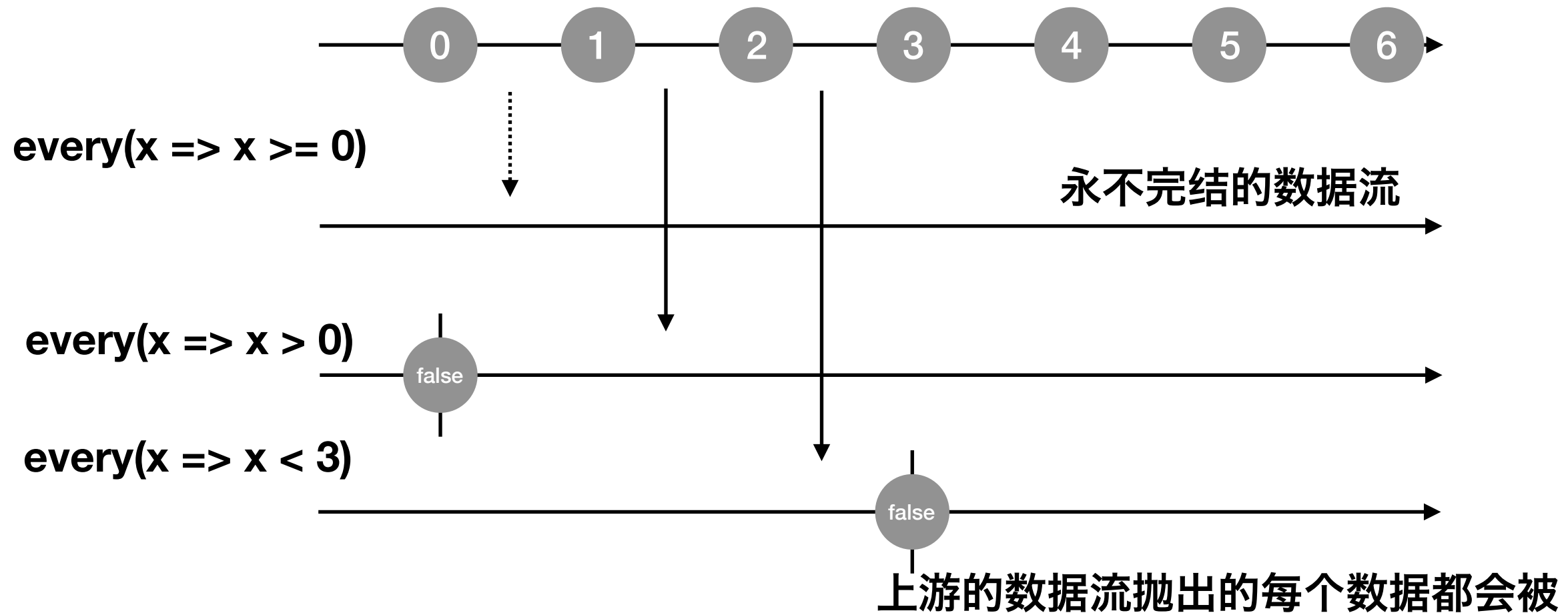
**findIndex 操作符**

**defaultIfEmpty 操作符**

根据上游 Observable 对象的某些条件产生  
一个新的 Observable 对象

根据**自身的判定函数**决定产生的新数据  
同时会记录上游传下来的数据的 **index 属性**  
以及**上游 Observable 数据源头**

# every 操作符



上游的数据流抛出的每个数据都会被  
操作符的判定函数校验

如果所有抛出的数据都符合条件  
则向下游传递一个 true 的数据

如果存在数据不符合条件

则停止订阅上游并向下游抛出一个 false 的数据

**但是要小心**

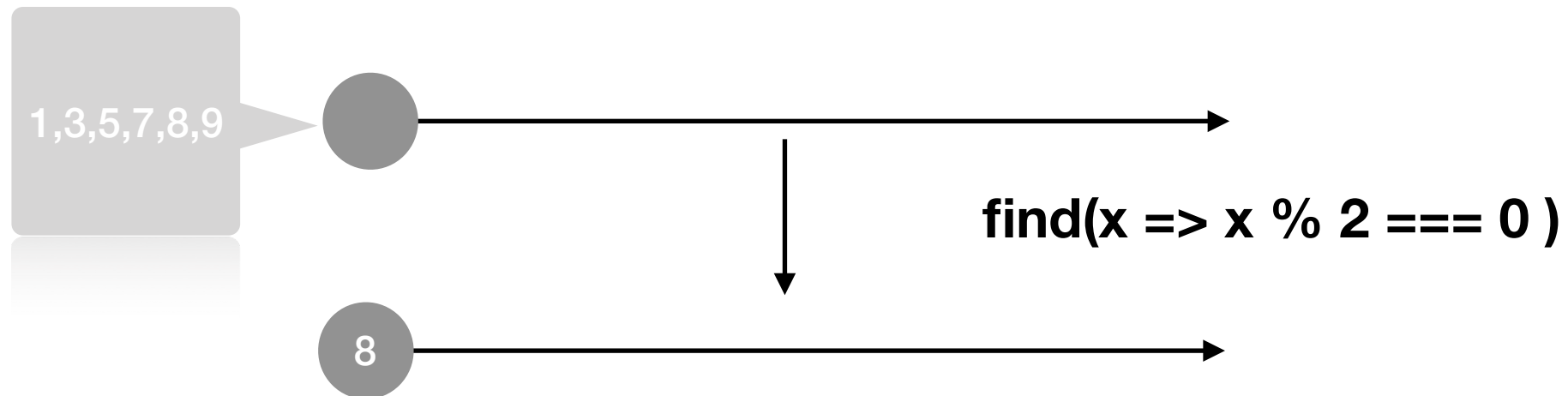
如果上游的数据流是不会完结的数据流

every也可能抛出一个永不完结的 Observable

```
// every 操作符 观察不同判定条件下所产生的数据
console.clear();
const source$ = interval(1000).pipe(
  every(x => x >= 0)
);

// 尝试将不同的判定条件加入上述判定函数中
const subscription = source$.subscribe(value => {
  console.log(value);
})
```

# find 操作符

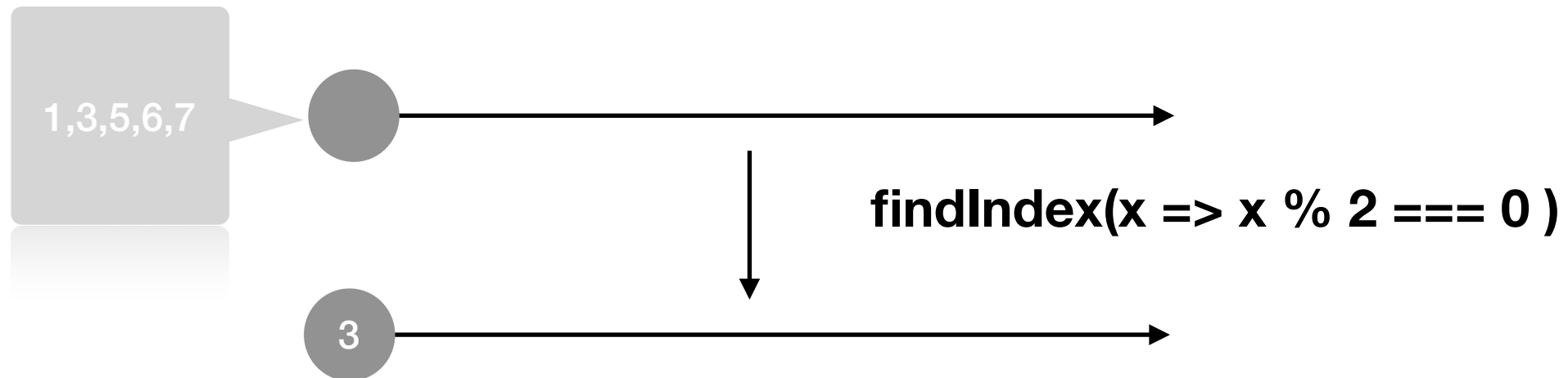


```
// find 操作符
console.clear();
// 如果上游数据抛出的数据没有一个符合判定条件 find 抛出 undefined
const source$ = of(1,3,5,7,8,9).pipe(
  find(x => x % 2 === 0)
);

const subscription = source$.subscribe(value => {
  console.log(value);
});
```

`find` 操作符会根据上游抛出的数据  
找出符合判定条件**第一个**的数据  
并抛出给下游  
如果上游抛出的数据抛出的数据  
没有一个符合条件的数据  
则向下游抛出 **undefined**

# findIndex 操作符



```
// findIndex 操作符
console.clear();
// 如果上游数据抛出的数据没有一个符合判定条件 findIndex 抛出 -1
const source$ = of(1,3,5,6,7).pipe(
  findIndex(x => x % 2 === 0)
);

const subscription2 = source$.subscribe(value => {
  console.log(value);
});
```

findIndex 操作符会根据上游抛出的数据  
找出符合判定条件**第一个**的数据的index  
并抛出给下游  
如果上游抛出的数据抛出的数据  
没有一个符合条件的数据  
则向下游抛出 **-1**

**需要注意**

上游数据流如果不会完结

且一直没有满足条件的数据出现

find/findIndex 操作符都会产生一个永不完结的数据流



如果我希望在获取到符合条件的第一个数据  
并同时获得其index  
应该怎么做呢？

提示：使用 zip 操作符

L276

# isEmpty 操作符

```
// isEmpty 操作符
console.clear()
const source$ = interval(1000).pipe(
  isEmpty()
);

const subscription = source$.subscribe(value => {
  console.log(value); // false
})

const source2$ = empty().pipe(
  isEmpty()
);

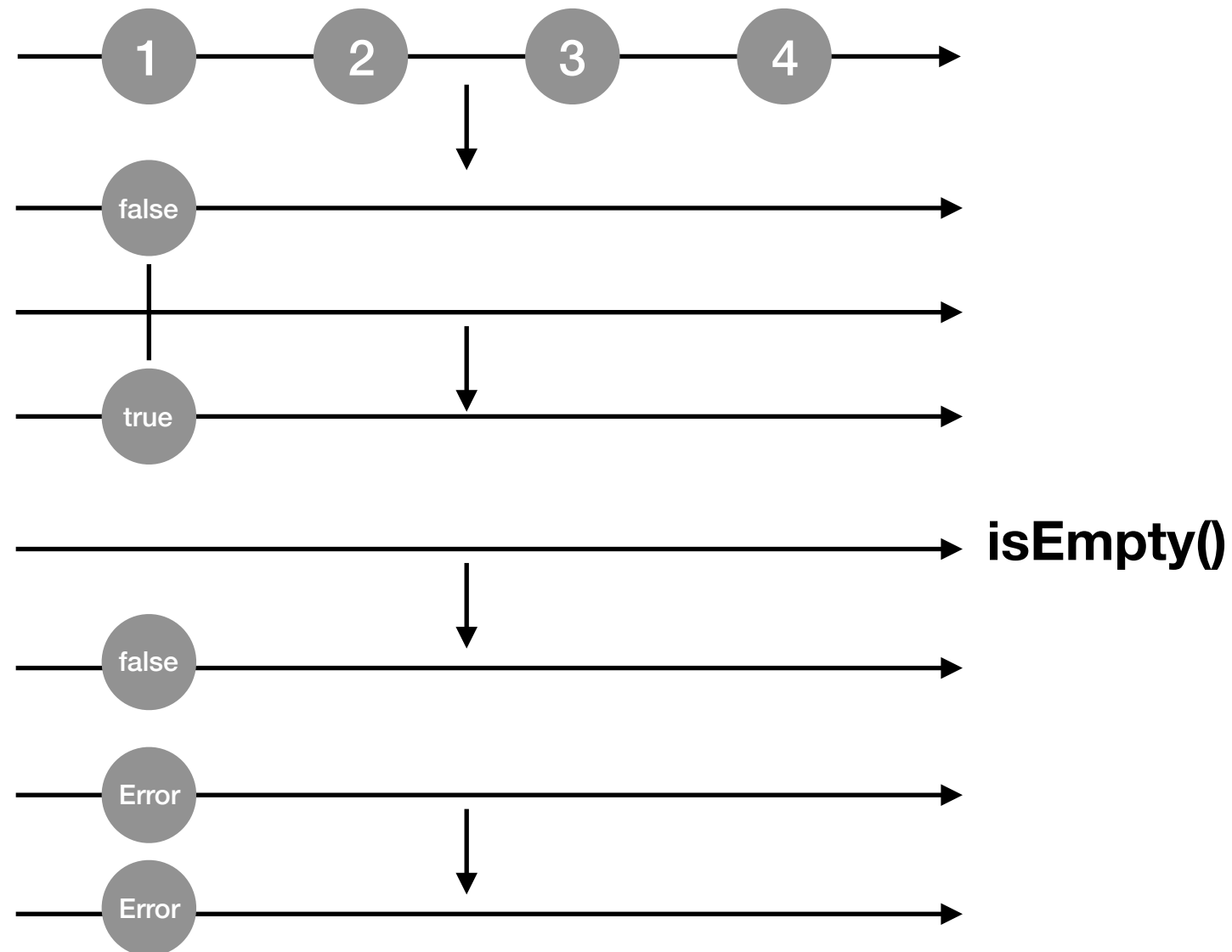
const subscription2 = source2$.subscribe(value => {
  console.log(value); // true
})

const source3$ = throwError('error').pipe(
  isEmpty()
);

const subscription3 = source3$.subscribe(value => {
  console.log(value);
}, error => {
  console.log(error);
})

const source4$ = never().pipe(
  isEmpty()
);

const subscription4 = source4$.subscribe(value => {
  console.log(value);
})
```



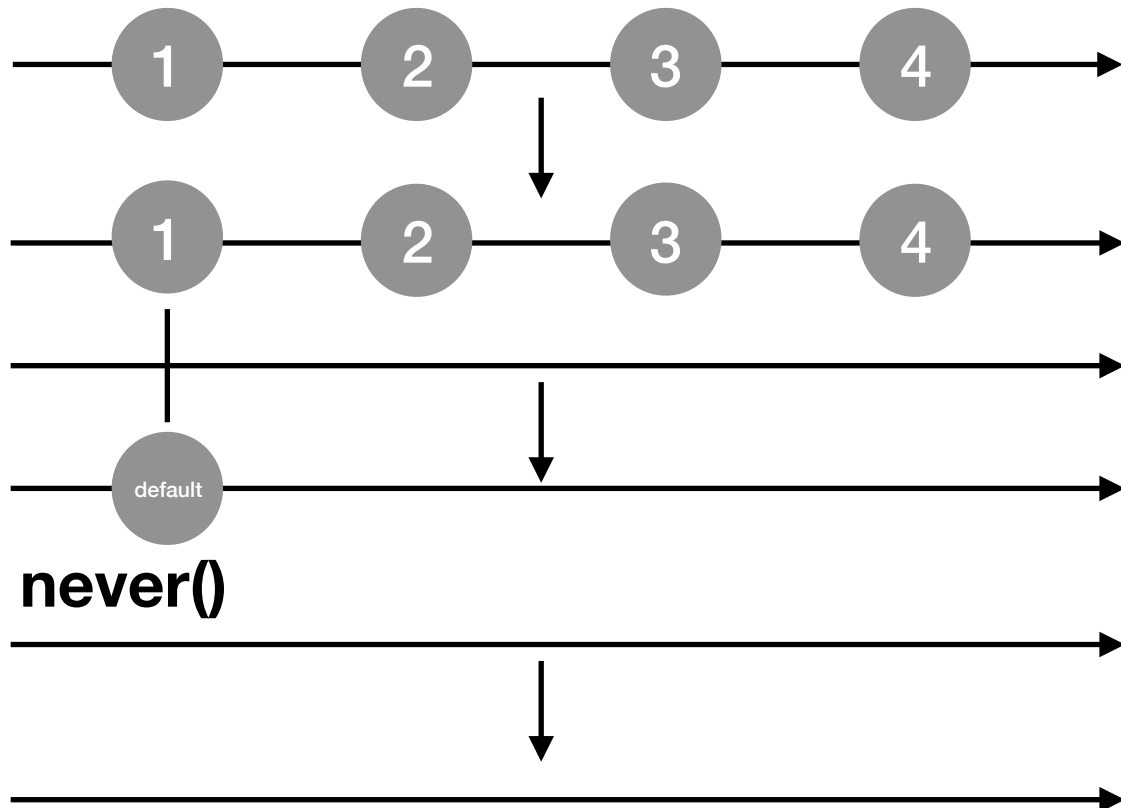
**isEmpty 操作符用于判断上游 Observable 数据流  
是否是 没有抛出任何数据就完结的 Observable  
是则抛出 true 的数据，否则抛出 false  
如果上游抛出一个 error  
操作符则会将 error 向下传递**

# defaultIfEmpty 操作符

defaultIfEmpty('default')

interval()

empty()



```
// defaultIfEmpty 操作符
console.clear();
const source$ = empty().pipe(
  defaultIfEmpty('default value')
);

const subscription = source$.subscribe(value => {
  console.log(value);
})

const source2$ = interval(1000).pipe(
  defaultIfEmpty()
);

const subscription2 = source2$.subscribe(value => {
  console.log(value);
})

const source3$ = never().pipe(
  defaultIfEmpty()
);

const subscription3 = source3$.subscribe(value => {
  console.log(value);
})
```

与 isEmpty 操作符类似

defaultIfEmpty 操作符提供了一个默认值  
当上游的 Observable 没有抛出任何数据就完结时

则抛出该默认值给下游

如果上游抛出了任何数据

则向下游直接抛出数据

如果上游是不完结的数据流

则下游数据流也会是一个永不完结的数据流

# 过滤操作符

上游传递给下游的数据

并不是全部都是下游需要或关注的

这时就需要可以过滤掉不需要的部分  
的操作符

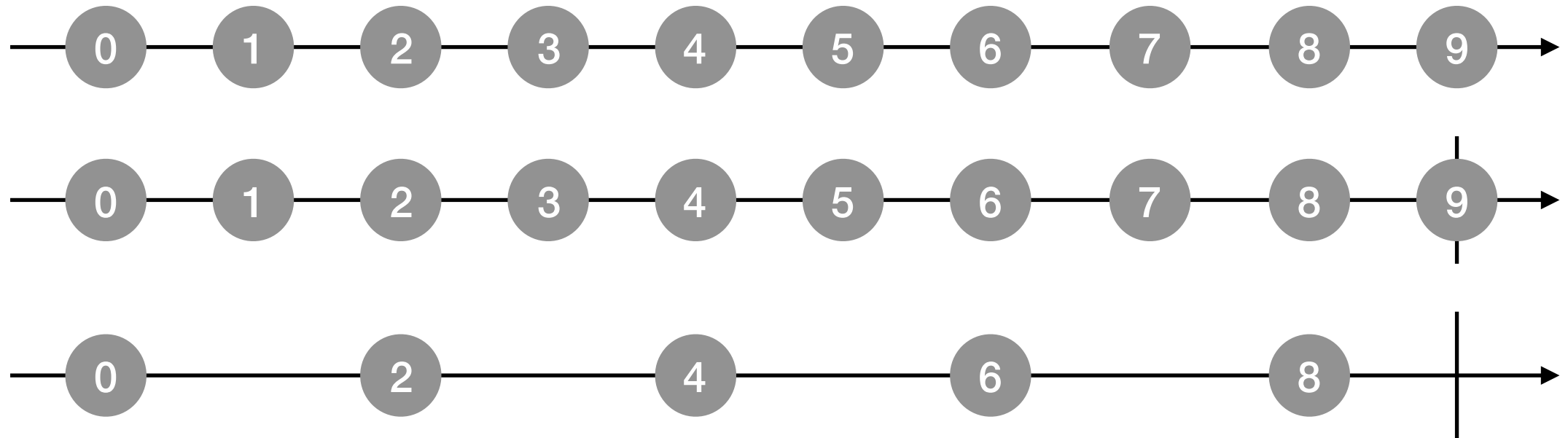
过滤操作符的作用就是对数据流中的每个数据  
都进行判断是否满足给定的条件

如果满足则向下传递

如果不满足则抛弃之

如果需要定制向下游传递的数据流  
还可以通过结果选择函数进行定制化

# filter 操作符



```
// filter 操作符
console.clear();
const source$ = interval(1000).pipe(
  take(10),
  filter(value => value % 2 === 0)
);

const subscription = source$.subscribe(value => {
  console.log(value);
});
```

需要注意的一点是  
filter 操作符所产生的 Observable 对象  
产生数据的时机和上游的 Observable  
**是完全一样的**  
上游同步则产生同步  
上游异步则产生异步

# First 操作符

```
// first 操作符
// 接受两个参数，第一个参数是判定函数，第二个参数是默认值
console.clear();
// 不接受参数时获取上游抛出的第一个数据
const source$ = interval(1000).pipe(
  first()
);

const subscription = source$.subscribe(value => console.log(value));

// 上游不抛出数据也不完结的话 first也会产生一个没有数据也不完结的数据流
const source2$ = never().pipe(
  first()
);

const subscription2 = source2$.subscribe(value => {
  console.log(value);
});

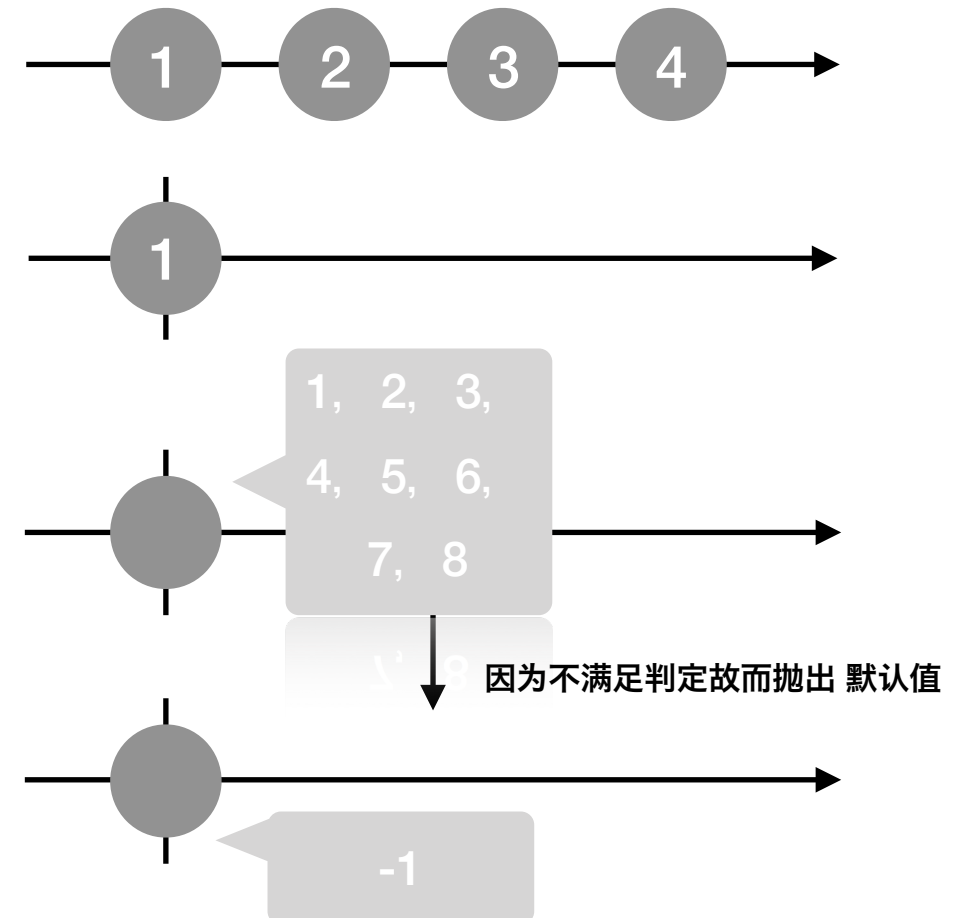
// 上游不抛出数据就完结 first 拿不到任何数据则不会抛出任何数据。
const source3$ = empty().pipe(
  first()
);

const subscription3 = source3$.subscribe(value => {
  console.log('抛出' + value);
});

// 使用两个参数

const source4$ = of(1,2,3,4,5,6,7,8).pipe(
  first(
    value => value % 9 === 0,
    -1
  )
);

const subscription4 = source4$.subscribe(value => {
  console.log(value);
});
```



**需要注意的是**

**v6 版本的 first 操作符只接受两个参数了**

**一个参数为判定函数**

**一个参数为默认值**

**v5 版本中支持的 投影函数 不再支持**

**仅支持 判定函数**

**官方文档尚未更新**

**源码**

# last 操作符

// last 操作符

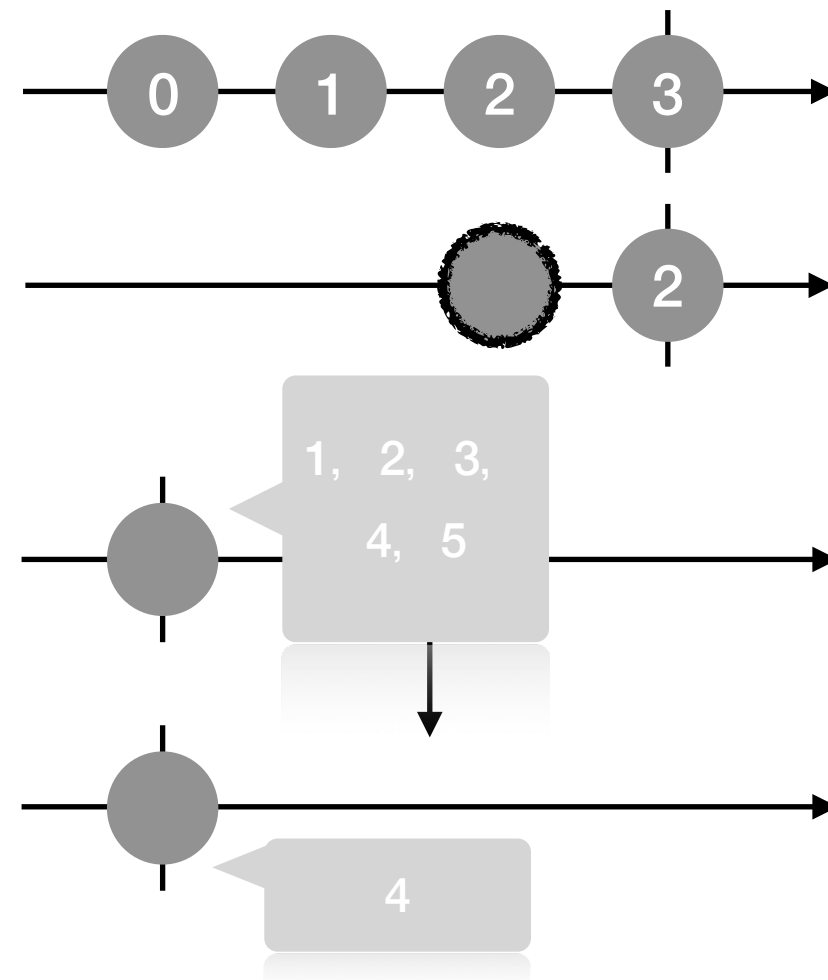
```
const source$ = of(1,2,3,4,5).pipe(  
  last(  
    value => value % 2 === 0,  
    -1  
  )  
);
```

```
const subscription = source$.subscribe(value => {  
  console.log(value);  
});
```

// 上游 Observable 完结后 last 才会抛出数据

```
const source2$ = interval(1000).pipe(  
  take(4),  
  last(  
    value => value % 2 === 0,  
    -1  
  )  
);
```

```
const subscription2 = source2$.subscribe(value => {  
  console.log(value);  
});
```



last 操作符与 first 操作符相似

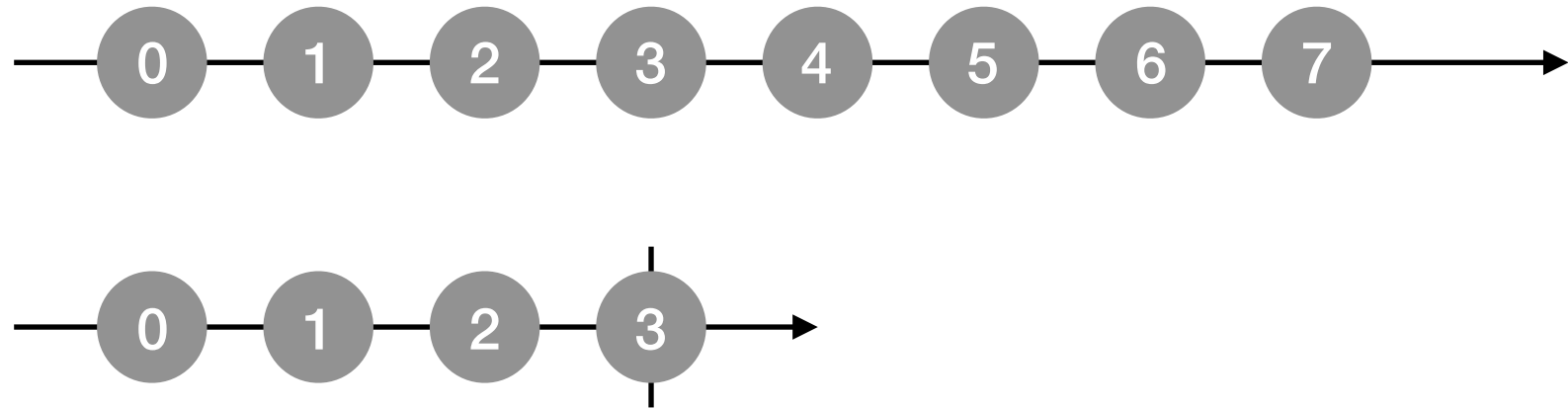
值得注意的地方是

只有当 上游 Observable 数据流**完结后**

last 操作符才会抛出数据

# take 家族 - take 操作符

take(3)



```
// take 操作符
```

```
const source$ = interval(1000).pipe(  
  take(3)  
);
```

```
const subscription = source$.subscribe(value => {  
  console.log(value);  
})
```

从上游拿去指定数量的 数据

无论是demo

还是项目中都是最常见的操作符之一

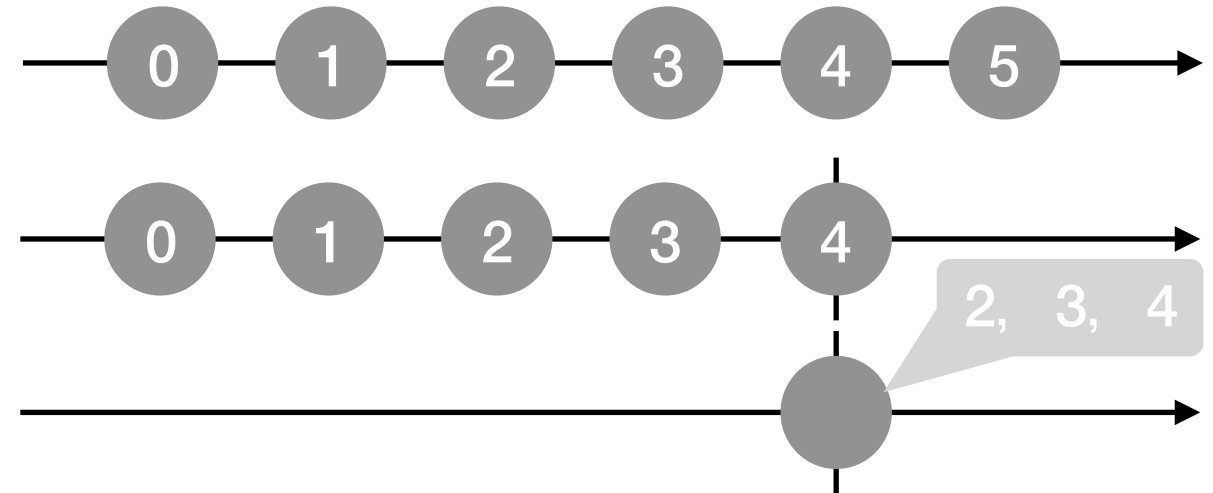
**需要注意**

上游产生一个数据

take 操作符都会立刻将该数据转手给下游



# take 家族 - takeLast 操作符



```
// takeLast 操作符
console.clear();
const source$ = interval(1000).pipe(
  take(5),
  takeLast(3)
);

const subscription = source$.subscribe(value => {
  console.log(value)
});

// 如果上游 Observable 不完结

const source2$ = interval(1000).pipe(
  takeLast(3)
);

const subscription2 = source2$.subscribe(value => {
  console.log(value); // takeLast 产生的数据流永远不会产生数据也不会完结
});

// 如果上游的数据不足 takeLast 传入的参数

const source3$ = interval(1000).pipe(
  take(2),
  takeLast(3)
);

const subscription3 = source3$.subscribe(value => {
  console.log(value); // 抛出上游所有的数据
});
```

**takeLast 只能在上游数据完结时  
才能确定要传递给下游的数据有哪些  
所以其只能一口气抛出所有的数据**

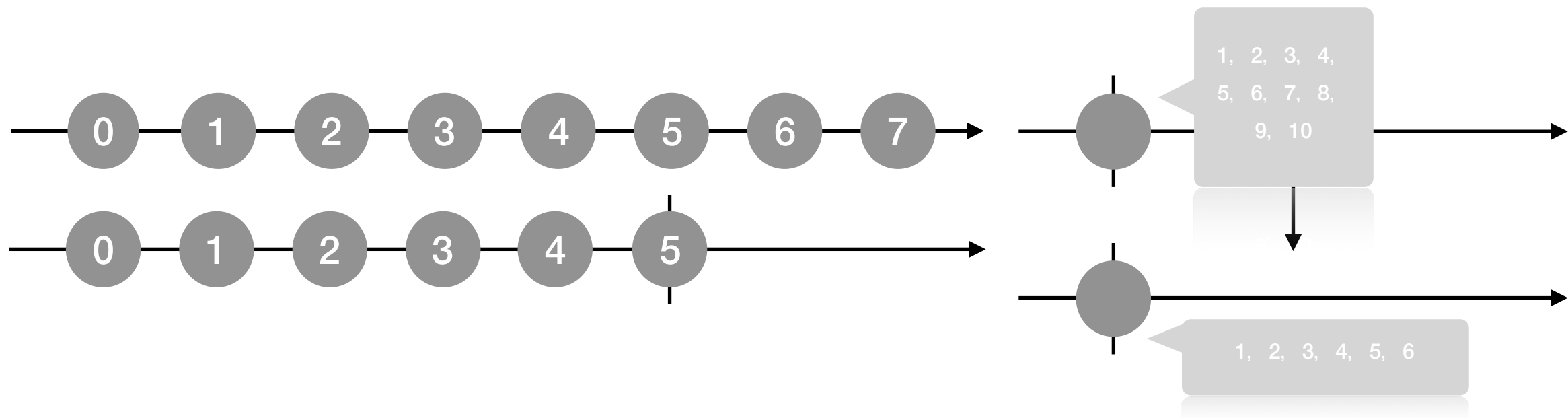
**如果其上游永不完结**

**则其产生的 Observable 对象永远不会产生数据**

**也永远不会完结**

**谁能知道不完结的数据流最后的数据呢？**

# take 家族 - takeWhile 操作符



```
// takeWhile 操作符等待上游
console.clear();
const source$ = range(1,10).pipe(
  takeWhile(
    (value,index) => value % 7 !== 0
  )
);

const subscription = source$.subscribe(value => {
  console.log(value);
});

const source2$ = interval(1000).pipe(
  takeWhile(
    value => (value + 1) % 7 !== 0
  )
);

const subscription2 = source2$.subscribe(value => {
  console.log(value);
});
```

**takeWhile 操作符接受一个判定函数**

**判定函数接受两个参数**

**分别是数据值本身**

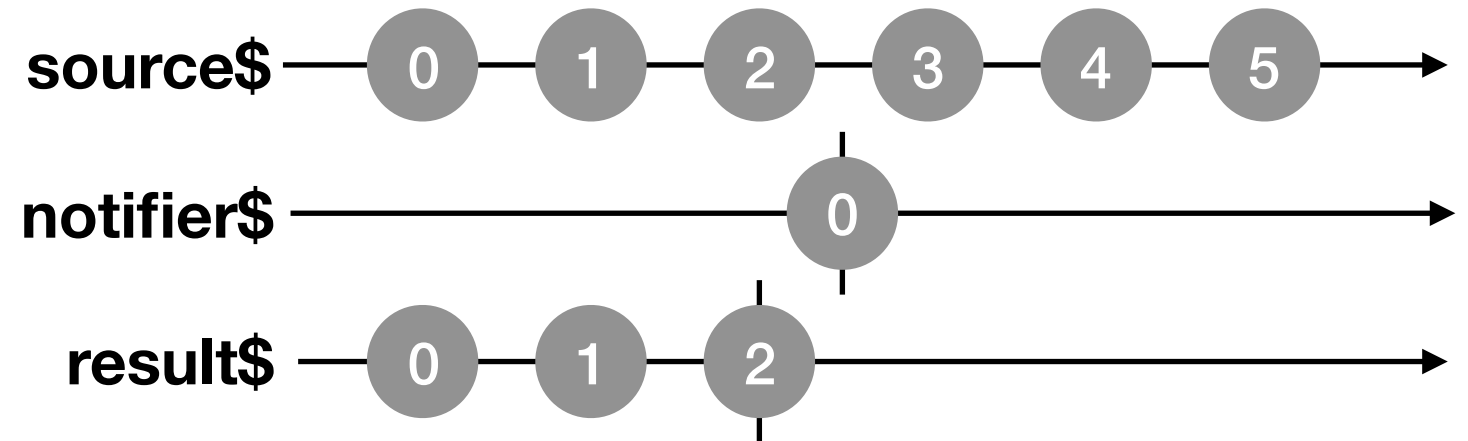
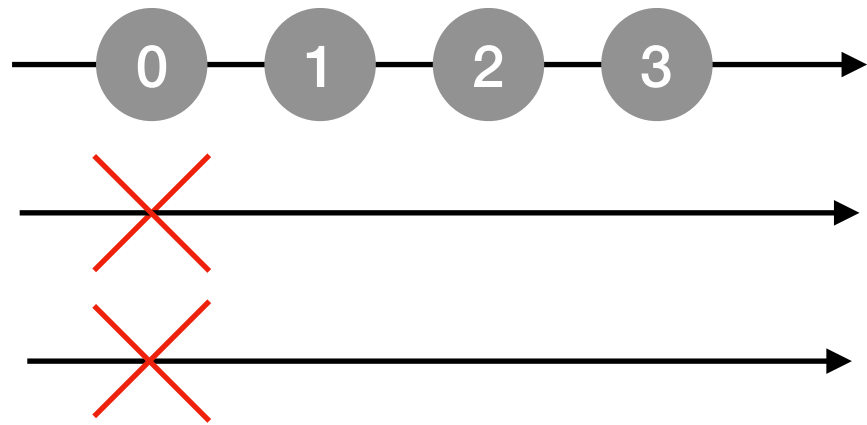
**和 数据的 index**

**takeWhile 会持续吐出上游数据**

**直到 判定函数返回false**

**可以理解为根据自产数据本身  
对自身进行控制的升级版 take 操作符  
会随着上游的节奏向下游抛出数据**

# take 家族 - takeUntil 操作符



// takeUntil 操作符

```
console.clear();
const notifier$ = timer(3500);
const source$ = interval(1000).pipe(
  takeUntil(notifier$)
);

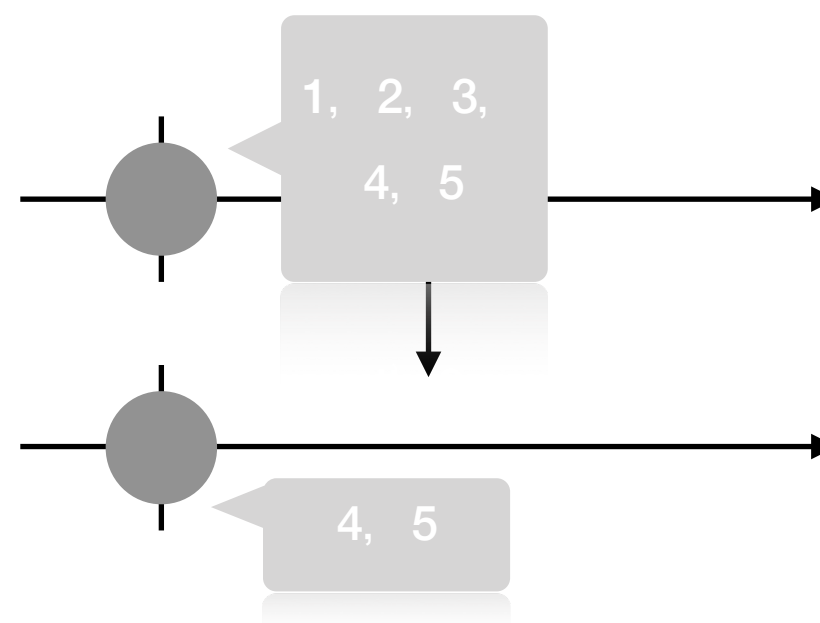
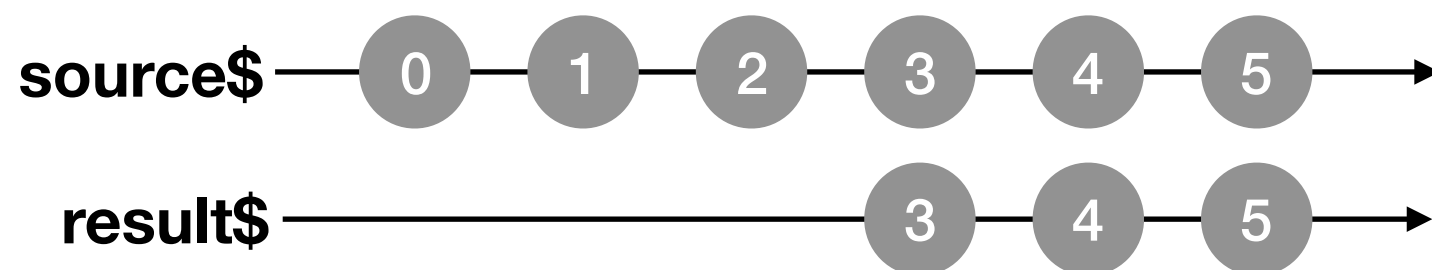
const subscription = source$.subscribe(value => {
  console.log(value);
})

const notifier2$ = throwError('err');
const source2$ = interval(1000).pipe(
  takeUntil(notifier2$)
);

const subscription2 = source2$.subscribe(value => {
  console.log(value);
})
```

之前的take 系列操作符都是通过  
上游抛出的数据  
或事先设定好的时间节点  
**但是 takeUntil 接受**  
**一个 Observable为notifier**  
这就让异步调控成为了可能  
takeUntil 就像一个水龙头一样  
**开关自控**

# skip 家族 - skip 操作符



// skip 操作符

```
const source$ = interval(1000).pipe(  
  skip(3)  
);
```

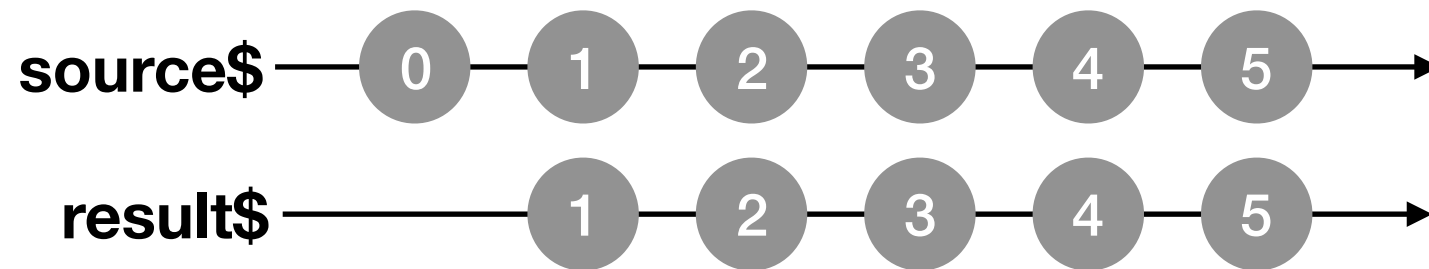
```
const subscription = source$.subscribe(value => {  
  console.log(value);  
});
```

```
const source2$ = of(1,2,3,4,5).pipe(  
  skip(3)  
);
```

```
const subscription2 = source2$.subscribe(value => {  
  console.log(value);  
})
```

与 take 操作符相对应的  
skip 用于跳过 x 个数据  
再拿去上游数据的方式  
下游数据流**跟随**  
上游数据抛出数据的节奏  
如果skip 接受的参数  
**大于**  
上游抛出的数据总量  
则下游数据流**直接完结**

# skipWhile 操作符



操作符会从上游数据抛出的数据中抛弃偶数的部分

如果没有偶数的部分则与上游数据流无异

只会跳过一个偶数而不是全部的偶数

当且仅当判定函数返回 true 时跳过

需要注意的是 skipWhile

是一个体感古怪的操作符

如果上游的数据进入判定函数一直返回true

则一直skip

一旦判定函数返回了false

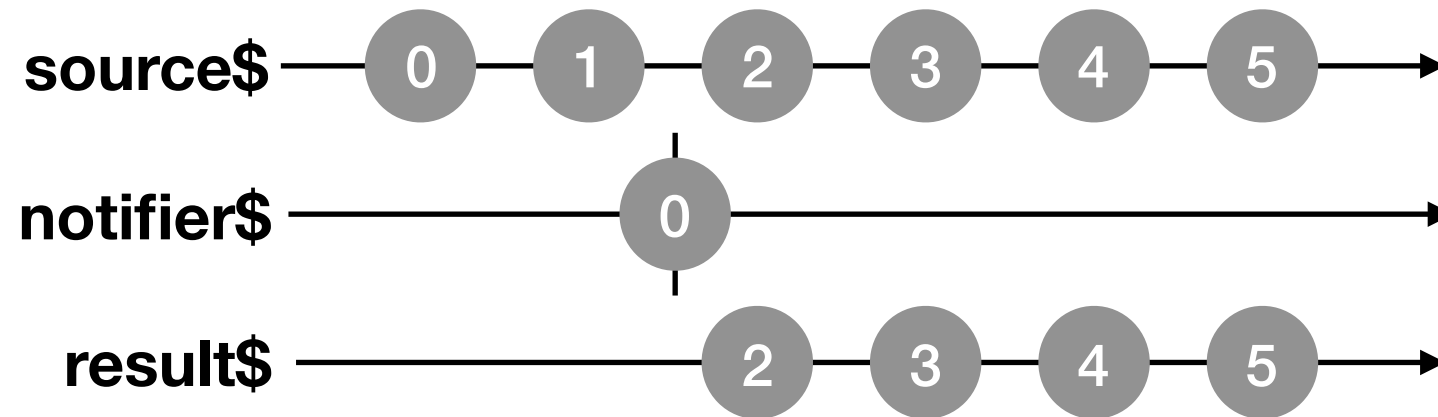
则正常接受上游 Observable 传递的数据

```
// skipWhile 操作符
console.clear();
const source$ = interval(1000).pipe(
  skipWhile(value => value % 2 === 0)
);

const subscription = source$.subscribe(value => {
  console.log(value);
});
```

Returns an Observable that skips all items emitted by the source Observable as long as a specified condition holds true, but emits all further source items as soon as the condition becomes false.

# skip 家族 - skipUntil 操作符



```
// skipUntil 操作符
```

```
console.clear();  
const source$ = interval(1000);  
const notifier$ = timer(2500);
```

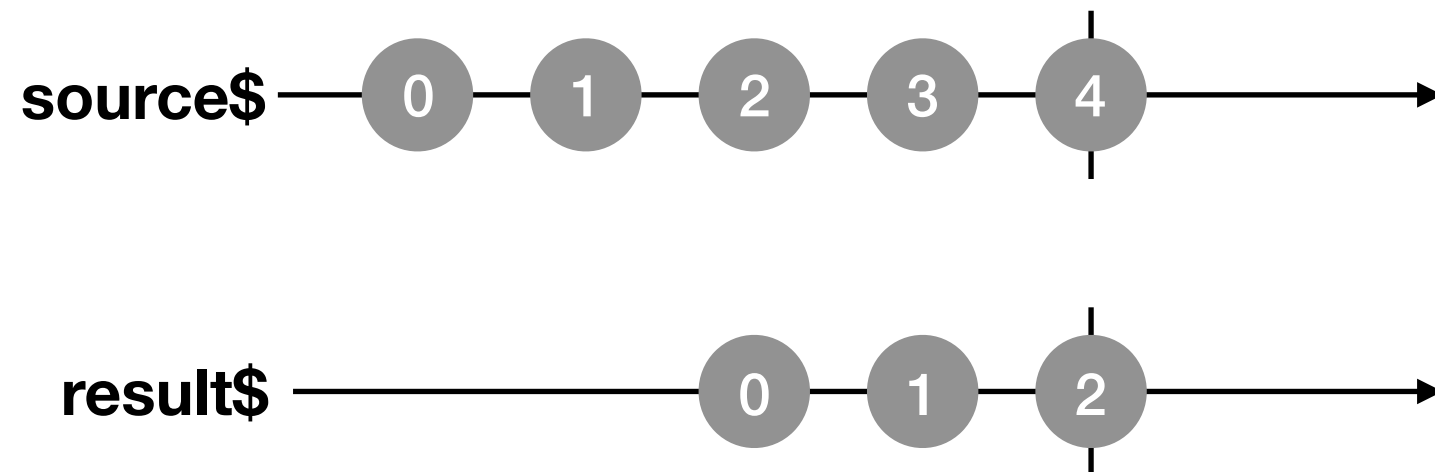
```
const result$ = source$.pipe(  
  skipUntil(notifier$)  
);
```

```
const subscription = result$.subscribe(value => {  
  console.log(value);  
});
```

与takeUntil 操作符一致  
通过 notifier\$ 数据流控制数据  
自己掌握传递数据的时机

爽

# skip 家族 - skipLast 操作符



```
// skipLast 操作符
console.clear();
const source$ = interval(1000).pipe(
  take(5),
  skipLast(2)
);

const subscription = source$.subscribe(value => {
  console.log(value);
});
```

**skipLast 操作符将会维护一个长度为  
传入参数的 队列**  
**优先使用上游抛出的数据充满队列**  
**后续再根据 先进先出的规则将  
序列的头部数据逐个抛出给下游数据流**

**注意**

**这样的操作会造成数据的延迟**

愉悦时间 🧐

玩具地址1 🤔

玩具地址2 🤔





**throttleTime(20)**



**tap(setScroll)**



