

RxJava使用教程

RxJava一路走来，已经是第三个版本了~现在百分之99.99还在使用RxJava 2.X

概念

1、RX的含义

Rx，指的是Reactive Extension，一种响应式扩展编程方式。

1.1、Reactive：

一种面向数据流和变化传播的[声明式](#)编程范式。这意味着可以在编程语言中很方便地表达静态或动态的数据流，而相关的计算模型会自动将变化的值通过数据流进行传播。

1.2、Extension：

扩展。通过使用不同的操作符,像玩乐高积木一样,随心所欲的拼出不同的模型。



2、RxJava基本知识

2.1、基类

在 RxJava 3 可以发现有几个基类（跟RxJava 2是一致的吧）：

- io.reactivex.Flowable：发送0个N个的数据，支持Reactive-Streams和背压
- io.reactivex.Observable：发送0个N个的数据，不支持背压，
- io.reactivex.Single：只能发送单个数据或者一个错误
- io.reactivex.Completable：没有发送任何数据，但只处理 onComplete 和 onError 事件。
- io.reactivex.Maybe：能够发射0或者1个数据，要么成功，要么失败。

2.2、上流、下游

在RxJava,数据以流的方式组织。也就是说, Rxjava包括一个源的数据流, 数据流后跟着消费者的零个到多个消费数据流步骤。

```
source
    .operator1()
    .operator2()
    .operator3()
    .subscribe(consumer)//此处要改写
```

在上文代码中, 对于operator2来说, 在它前面叫做上游(流), 在它后面的叫做下游(流)。

2.3、流的对象

在RxJava中, 常见的emits, item, event, signal, emission, data and message都被认为在数据流中被传递的数据对象。

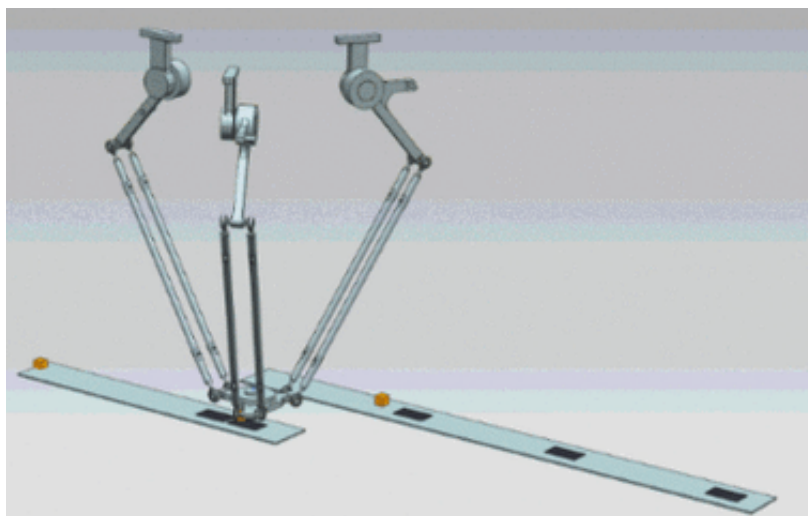
2.4、线程调度器 (Schedulers)

对于我们Android开发来说, 最喜欢的就是它简洁切换线程的操作。

- Schedulers.computation(): 适合运行在密集计算的操作, 大多数异步操作符使用该调度器。
- Schedulers.io():适合运行I/O和阻塞操作.
- Schedulers.single():适合需要单一线程的操作
- Schedulers.trampoline(): 适合需要顺序运行的操作

在不同平台还有不同的调度器, 例如Android的主线程: `AndroidSchedulers.mainThread()`

```
Flowable.range(1, 10)
    .observeOn(Schedulers.computation())
    .map(v -> v * v)
    .blockingSubscribe(System.out::println);
```



2.5、背压(Backpressure)

当数据流通过异步的步骤执行时，可能上游发送速度太快，下游处理不过来导致阻塞。为了避免这种情况，要么缓存上流的数据，要么抛弃数据。为此，RxJava带来了backpressure的概念。背压是一种流量的控制步骤，在不知道上流还有多少数据的情形下控制内存的使用，表示它们还能处理多少数据。

支持背压的仅有Flowable类，Observable，Single，Maybe and Completable不支持。

####

操作符

实用操作符

1、ObserveOn

指定观察者的线程，例如在Android访问网络后，数据需要主线程消费，那么将观察者的线程切换到主线就需要ObserveOn操作符。每次指定一次都会生效。

2、subscribeOn

指定被观察者的线程，即数据源发生的线程。例如在Android访问网络时，需要将线程切换到子线程。多次指定只有第一次有效。

3、doOnEach

数据源（Observable）每发送一次数据，就调用一次。

4、doOnNext

数据源每次调用onNext() 之前都会先回调该方法。

5、doOnError

数据源每次调用onError() 之前会回调该方法。

6、doOnComplete

数据源每次调用onComplete() 之前会回调该方法

7、doOnSubscribe

数据源每次调用onSubscribe() 之后会回调该方法

8、doOnDispose

数据源每次调用dispose() 之后会回调该方法

其他的见官网吧，不难

[实用操作符](#)

对数据源过滤操作符

主要讲对数据源进行选择 and 过滤的常用操作符

1、skip（跳过）

可以作用于Flowable,Observable，表示源发射数据前，跳过多少个。例如下面跳过前四个：

```
Observable<Integer> source = Observable.just(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);

source.skip(4)
    .subscribe(System.out::print);
```

打印结果：5678910

```
Observable<Integer> source = Observable.just(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);

source.skipLast(4)
    .subscribe(System.out::print);
```

打印结果：1 2 3 4 5 6

skipLast(n)操作表示从流的尾部跳过n个元素。

2、debounce（去抖动）

可作用于Flowable,Observable。在Android开发，通常为了防止用户重复点击而设置标记位，而通过RxJava的debounce操作符可以有效达到该效果。在规定时间内，用户重复点击只有最后一次有效，

```
Observable<String> source = Observable.create(emitter -> {
    emitter.onNext("A");

    Thread.sleep(1_500);
    emitter.onNext("B");

    Thread.sleep(500);
    emitter.onNext("C");

    Thread.sleep(250);
    emitter.onNext("D");

    Thread.sleep(2_000);
    emitter.onNext("E");
    emitter.onComplete();
});

source.subscribeOn(Schedulers.io())
    .debounce(1, TimeUnit.SECONDS)
    .blockingSubscribe(
```

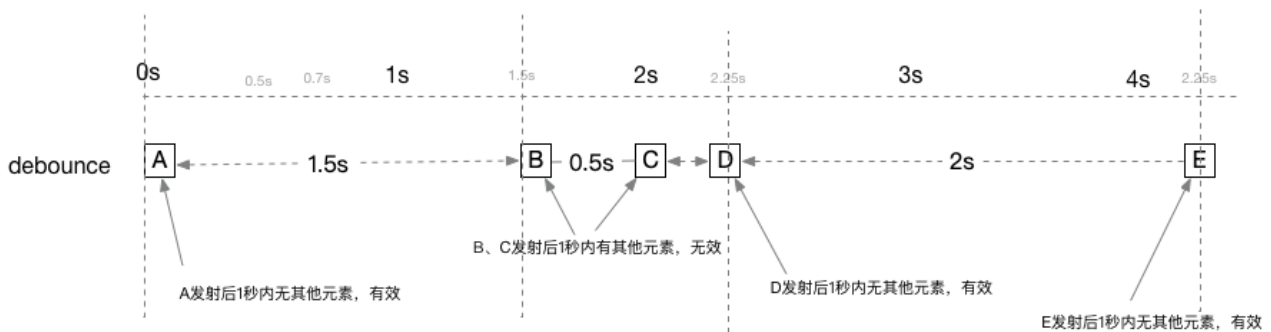
```

item -> System.out.print(item+" "),
Throwable::printStackTrace,
() -> System.out.println("onComplete"));

```

打印: A D E onComplete

上文代码中，数据源以一定的时间间隔发送A,B,C,D,E。操作符debounce的时间设为1秒，发送A后1.5秒并没有发射其他数据，所以A能成功发射。发射B后，在1秒之内，又发射了C和D,在D之后的2秒才发射E,所有B、C都失效，只有D有效；而E之后已经没有其他数据流了，所有E有效。



debounce操作符示例图

3、distinct（去重）

可作用于Flowable,Observable，去掉数据源重复的数据。

```

Observable.just(2, 3, 4, 4, 2, 1)
    .distinct()
    .subscribe(System.out::println);

// 打印:2 3 4 2 1
Observable.just(1, 1, 2, 1, 2, 3, 3, 4)
    .distinctUntilChanged()
    .subscribe(System.out::print);
//打印: 1 2 1 2 3 4

```

distinctUntilChanged()去掉相邻重复数据。

4、elementAt（获取指定位置元素）

可作用于Flowable,Observable，从数据源获取指定位置的元素，从0开始。

```
Observable.just(2,4,3,1,5,8)
    .elementAt(0)
    .subscribe(integer ->
        Log.d("TAG","elmentAt->" + integer));
```

打印: 2

```
Observable<String> source = Observable.just("Kirk", "Spock", "Chekov", "Sulu");
Single<String> element = source.elementAtOrNull(4);
```

```
element.subscribe(
    name -> System.out.println("onSuccess will not be printed!"),
    error -> System.out.println("onError: " + error));
```

打印: onSuccess will not be printed!

elementAtOrNull: 指定元素的位置超过数据长度, 则发射异常。

5、filter (过滤)

可作用于 Flowable,Observable,Maybe,Single。在filter中返回表示发射该元素, 返回false表示过滤该数据。

```
Observable.just(1, 2, 3, 4, 5, 6)
    .filter(x -> x % 2 == 0)
    .subscribe(System.out::println);
```

打印: 2 4 6

6、first(第一个)

作用于 Flowable,Observable。发射数据源第一个数据, 如果没有则发送默认值。

```
Observable<String> source = Observable.just("A", "B", "C");
Single<String> firstOrDefault = source.first("D");
firstOrDefault.subscribe(System.out::println);
```

打印: A

```
Observable<String> emptySource = Observable.empty();
Single<String> firstOrNull = emptySource.firstOrNull();
firstOrNull.subscribe(
    element -> System.out.println("onSuccess will not be printed!"),
    error -> System.out.println("onError: " + error));
```

打印: onError: java.util.NoSuchElementException

和firstElement的区别是first返回的是Single, 而firstElement返回Maybe。firstOrNull在没有数据会返回异常。

7、last(最后一个)

last、lastElement、lastOnError与first、firstElement、firstOnError相对应。

```
Observable<String> source = Observable.just("A", "B", "C");
Single<String> lastOrDefault = source.last("D");
lastOrDefault.subscribe(System.out::println);
//打印:c

Observable<String> source = Observable.just("A", "B", "C");
Maybe<String> last = source.lastElement();
last.subscribe(System.out::println);
//打印:c

Observable<String> emptySource = Observable.empty();
Single<String> lastOnError = emptySource.lastOnError();
lastOnError.subscribe(
    element -> System.out.println("onSuccess will not be printed!"),
    error -> System.out.println("onError: " + error));
// 打印: onError: java.util.NoSuchElementException
```

8、ignoreElements & ignoreElement（忽略元素）

ignoreElements 作用于Flowable、Observable。**ignoreElement**作用于Maybe、Single。两者都是忽略掉数据，返回完成或者错误时间。

```
Single<Long> source = Single.timer(1, TimeUnit.SECONDS);
Completable completable = source.ignoreElement();
completable.doOnComplete(() -> System.out.println("Done!"))
    .blockingAwait();
// 1秒后打印: Done!

Observable<Long> source = Observable.intervalRange(1, 5, 1, 1,
    TimeUnit.SECONDS);
Completable completable = source.ignoreElements();
completable.doOnComplete(() -> System.out.println("Done!"))
    .blockingAwait();
// 五秒后打印: Done!
```

9、ofType（过滤掉类型）

作用于Flowable、Observable、Maybe、过滤掉类型。

```
Observable<Number> numbers = Observable.just(1, 4.0, 3, 2.71, 2f, 7);
Observable<Integer> integers = numbers.ofType(Integer.class);
integers.subscribe((Integer x) -> System.out.print(x+" "));
//打印:1 3 7
```

10、sample

作用于Flowable、Observable，在一个周期内发射最新的数据。

```
Observable<String> source = Observable.create(emitter -> {
    emitter.onNext("A");

    Thread.sleep(500);
    emitter.onNext("B");

    Thread.sleep(200);
    emitter.onNext("C");

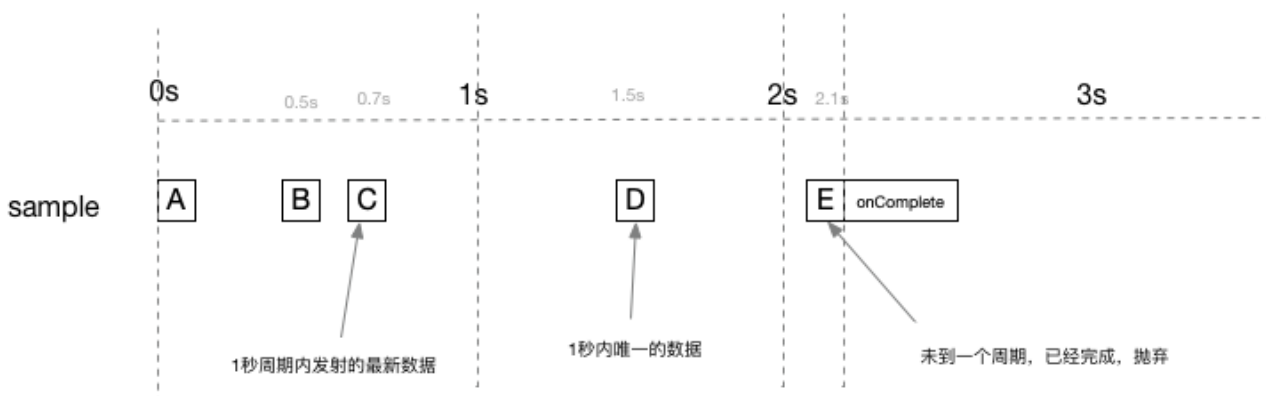
    Thread.sleep(800);
    emitter.onNext("D");

    Thread.sleep(600);
    emitter.onNext("E");
    emitter.onComplete();
});

source.subscribeOn(Schedulers.io())
    .sample(1, TimeUnit.SECONDS)
    .blockingSubscribe(
        item -> System.out.print(item+" "),
        Throwable::printStackTrace,
        () -> System.out.print("onComplete"));

// 打印: C D onComplete
```

与debounce的区别是，sample是以时间为周期的发射，一秒又一秒内的最新数据。而debounce是最后一个有效数据开始。



Sample操作符示例图

11、throttleFirst & throttleLast & throttleWithTimeout

作用于Flowable、Observable。throttleLast与smample一致，而throttleFirst是指定周期内第一个数据。throttleWithTimeout与debounce一致。

```
Observable<String> source = Observable.create(emitter -> {
    emitter.onNext("A");

    Thread.sleep(500);
    emitter.onNext("B");

    Thread.sleep(200);
    emitter.onNext("C");

    Thread.sleep(800);
    emitter.onNext("D");

    Thread.sleep(600);
    emitter.onNext("E");
    emitter.onComplete();
});

source.subscribeOn(Schedulers.io())
    .throttleFirst(1, TimeUnit.SECONDS)
    .blockingSubscribe(
        item -> System.out.print(item+" "),
        Throwable::printStackTrace,
        () -> System.out.print(" onComplete"));
//打印:A D onComplete

source.subscribeOn(Schedulers.io())
    .throttleLast(1, TimeUnit.SECONDS)
    .blockingSubscribe(
        item -> System.out.print(item+" "),
        Throwable::printStackTrace,
        () -> System.out.print(" onComplete"));

// 打印:C D onComplete
```

12、throttleLatest

之所以拿出来单独说，我看不懂官网的解释。然后看别人的文章：throttleFirst+throttleLast的组合？开玩笑的吧。个人理解是：如果源的第一个数据总会被发射，然后开始周期计时，此时的效果就会跟throttleLast一致。

```
Observable<String> source = Observable.create(emitter -> {
    emitter.onNext("A");
```

```
Thread.sleep(500);
emitter.onNext("B");

Thread.sleep(200);
emitter.onNext("C");

Thread.sleep(200);
emitter.onNext("D");

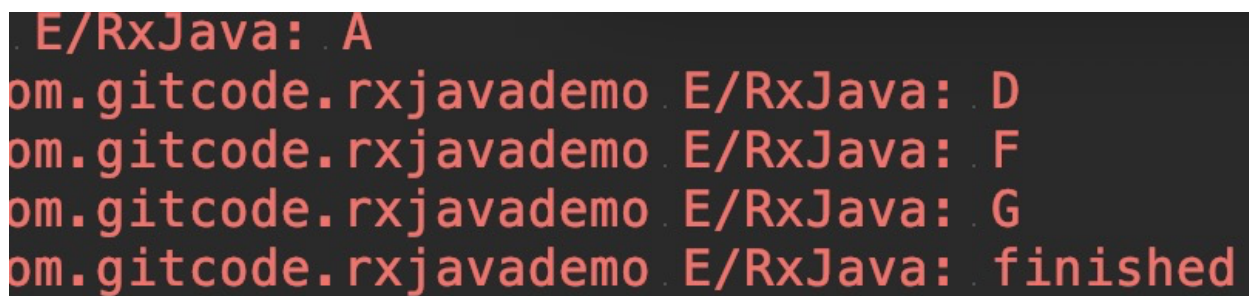
Thread.sleep(400);
emitter.onNext("E");

Thread.sleep(400);
emitter.onNext("F");

Thread.sleep(400);
emitter.onNext("G");

Thread.sleep(2000);
emitter.onComplete();
});
source.subscribeOn(Schedulers.io())
    .throttleLatest(1, TimeUnit.SECONDS)
    .blockingSubscribe(
        item -> Log.e("RxJava",item),
        Throwable::printStackTrace,
        () -> Log.e("RxJava","finished"));
```

打印结果：

A terminal window with a dark background and red text. The output shows a sequence of log messages from the package 'com.gitcode.rxjavademo'. The first line is 'E/RxJava: A'. This is followed by four lines, each with a package name and a log message: 'com.gitcode.rxjavademo E/RxJava: D', 'com.gitcode.rxjavademo E/RxJava: F', 'com.gitcode.rxjavademo E/RxJava: G', and 'com.gitcode.rxjavademo E/RxJava: finished'.

```
com.gitcode.rxjavademo E/RxJava: A
com.gitcode.rxjavademo E/RxJava: D
com.gitcode.rxjavademo E/RxJava: F
com.gitcode.rxjavademo E/RxJava: G
com.gitcode.rxjavademo E/RxJava: finished
```

13、take & takeLast

作用于Flowable、Observable，take发射前n个元素;takeLast发射后n个元素。

```
Observable<Integer> source = Observable.just(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);

source.take(4)
    .subscribe(System.out::print);
//打印:1 2 3 4

source.takeLast(4)
    .subscribe(System.out::println);
//打印:7 8 9 10
```

14、timeout（超时）

作用于Flowable、Observable、Maybe、Single、Completable。后一个数据发射未在前一个元素发射后规定时间内发射则返回超时异常。

```
Observable<String> source = Observable.create(emitter -> {
    emitter.onNext("A");

    Thread.sleep(800);
    emitter.onNext("B");

    Thread.sleep(400);
    emitter.onNext("C");

    Thread.sleep(1200);
    emitter.onNext("D");
    emitter.onComplete();
});

source.timeout(1, TimeUnit.SECONDS)
    .subscribe(
        item -> System.out.println("onNext: " + item),
        error -> System.out.println("onError: " + error),
        () -> System.out.println("onComplete will not be printed!"));

// 打印:
// onNext: A
// onNext: B
// onNext: C
// onError: java.util.concurrent.TimeoutException:
//         The source did not signal an event for 1 seconds
//         and has been terminated.
```

连接操作符

通过连接操作符，将多个被观察数据（数据源）连接在一起。

1、startWith

可作用于Flowable、Observable。将指定数据源合并到另外数据源的开头。

```
Observable<String> names = Observable.just("Spock", "McCoy");
Observable<String> otherNames = Observable.just("Git", "Code", "8");
names.startWith(otherNames).subscribe(item -> Log.d(TAG, item));

//打印:
RxJava: Git
RxJava: Code
RxJava: 8
RxJava: Spock
RxJava: McCo
```

2、merge

可作用于所有数据源类型，用于合并多个数据源到一个数据源。

```
Observable<String> names = Observable.just("Hello", "world");
Observable<String> otherNames = Observable.just("Git", "Code", "8");

Observable.merge(names, otherNames).subscribe(name -> Log.d(TAG, name));

//也可以是
//names.mergeWith(otherNames).subscribe(name -> Log.d(TAG, name));

//打印:
RxJava: Hello
RxJava: world
RxJava: Git
RxJava: Code
RxJava: 8
```

merge在合并数据源时，如果一个合并发生异常后会立即调用观察者的onError方法，并停止合并。可通过mergeDelayError操作符，将发生的异常留到最后处理。

```
Observable<String> names = Observable.just("Hello", "world");
Observable<String> otherNames = Observable.just("Git", "Code", "8");
Observable<String> error = Observable.error(
    new NullPointerException("Error!"));
Observable.mergeDelayError(names, error, otherNames).subscribe(
    name -> Log.d(TAG, name), e -> Log.d(TAG, e.getMessage()));
```

```
//打印:  
RxJava: Hello  
RxJava: world  
RxJava: Git  
RxJava: Code  
RxJava: 8  
RxJava: Error!
```

3、zip

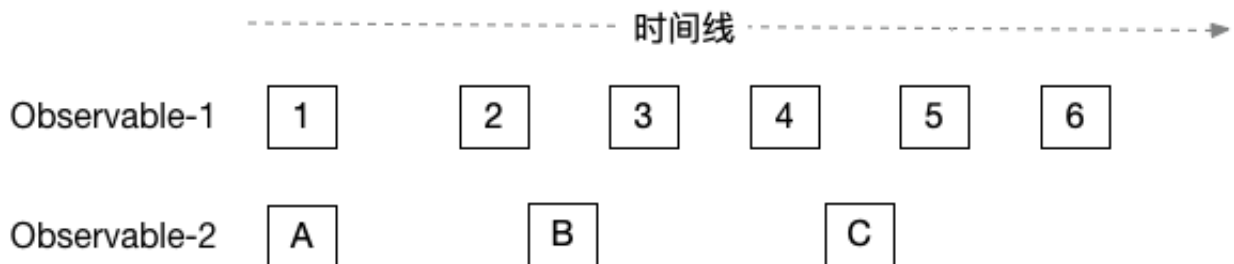
可作用于Flowable、Observable、Maybe、Single。将多个数据源的数据一个一个的合并在一起哇。当其中一个数据源发射完事件之后，若其他数据源还有数据未发射完毕，也会停止。

```
Observable<String> names = Observable.just("Hello", "world");  
Observable<String> otherNames = Observable.just("Git", "Code", "8");  
names.zipWith(otherNames, (first, last) -> first + "-" + last)  
    .subscribe(item -> Log.d(TAG, item));  
  
//打印:  
RxJava: Hello-Git  
RxJava: world-Code
```

4、combineLatest

可作用于Flowable, Observable。在结合不同数据源时，发射速度快的数据源最新item与较慢的相结合。

如下时间线，Observable-1发射速率快，发射了65，Observable-2才发射了C, 那么两者结合就是C5。



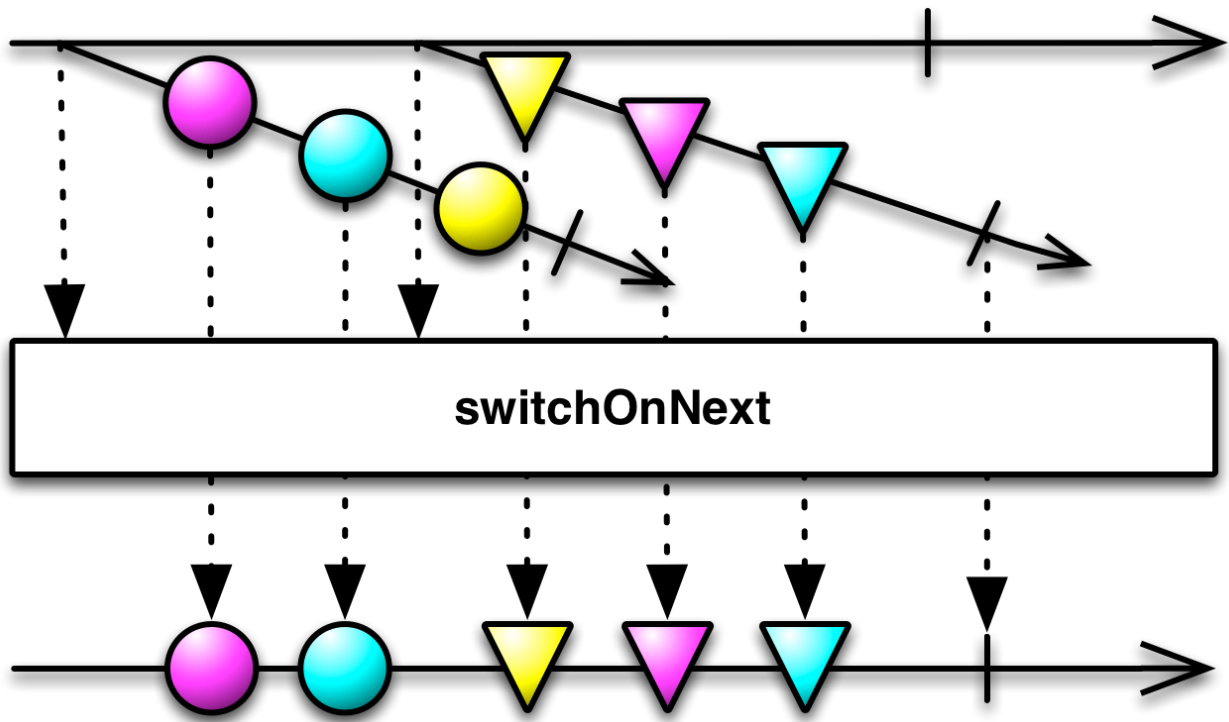
输出：C5

输出：B3

输出：A1

5、switchOnNext

一个发射多个小数据源的数据源，这些小数据源发射数据的时间发生重复时，取最新的数据源。



变换操作符

变化数据源的数据，并转化为新的数据源。

1、buffer

作用于Flowable、Observable。指将数据源拆解含有长度为n的list的多个数据源，不够n的成为一个数据源。

```
Observable.range(0, 10)
    .buffer(4)
    .subscribe((List<Integer> buffer) -> System.out.println(buffer));

// 打印:
// [0, 1, 2, 3]
// [4, 5, 6, 7]
// [8, 9]
```

2、cast

作用于Flowable、Observable、Maybe、Single。将数据元素转型成其他类型,转型失败会抛出异常。

```
Observable<Number> numbers = Observable.just(1, 4.0, 3f, 7, 12, 4.6, 5);

numbers.filter((Number x) -> Integer.class.isInstance(x))
    .cast(Integer.class)
    .subscribe((Integer x) -> System.out.println(x));
// prints:
// 1
// 7
// 12
// 5
```

3、concatMap

作用于Flowable、Observable、Maybe。将数据源的元素作用于指定函数后，将函数的返回值有序的存在新的数据源。

```
Observable.range(0, 5)
    .concatMap(i -> {
        long delay = Math.round(Math.random() * 2);

        return Observable.timer(delay, TimeUnit.SECONDS).map(n -> i);
    })
    .blockingSubscribe(System.out::print);

// prints 01234
```

4、concatMapDelayError

与concatMap作用相同，只是将过程发送的所有错误延迟到最后处理。

```
Observable.intervalRange(1, 3, 0, 1, TimeUnit.SECONDS)
    .concatMapDelayError(x -> {
        if (x.equals(1L)) return Observable.error(new IOException("Something
went wrong!"));
        else return Observable.just(x, x * x);
    })
    .blockingSubscribe(
        x -> System.out.println("onNext: " + x),
        error -> System.out.println("onError: " + error.getMessage()));

// prints:
// onNext: 2
// onNext: 4
// onNext: 3
// onNext: 9
// onError: Something went wrong!
```

5、concatMapCompletable

作用于Flowable、Observable。与contactMap类似，不过应用于函数后，返回的是CompletableSource。订阅一次并在所有CompletableSource对象完成时返回一个Completable对象。

```
Observable<Integer> source = Observable.just(2, 1, 3);
Completable completable = source.concatMapCompletable(x -> {
    return Completable.timer(x, TimeUnit.SECONDS)
        .doOnComplete(() -> System.out.println("Info: Processing of item \"" +
x + "\" completed"));
});

completable.doOnComplete(() -> System.out.println("Info: Processing of all
items completed"))
    .blockingAwait();

// prints:
// Info: Processing of item "2" completed
// Info: Processing of item "1" completed
// Info: Processing of item "3" completed
// Info: Processing of all items completed
```

6、concatMapCompletableDelayError

与concatMapCompletable作用相同，只是将过程发送的所有错误延迟到最后处理。

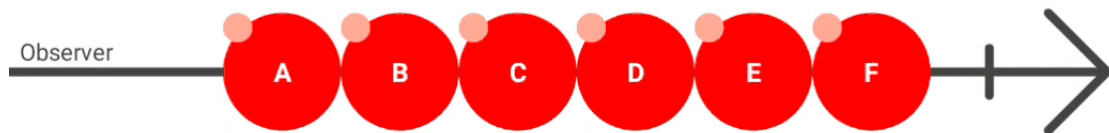
```
Observable<Integer> source = Observable.just(2, 1, 3);
Completable completable = source.concatMapCompletableDelayError(x -> {
    if (x.equals(2)) {
        return Completable.error(new IOException("Processing of item \"" + x +
"\\" failed!"));
    } else {
        return Completable.timer(1, TimeUnit.SECONDS)
            .doOnComplete(() -> System.out.println("Info: Processing of item
\\" + x + "\" completed"));
    }
});

completable.doOnError(error -> System.out.println("Error: " +
error.getMessage()))
    .onErrorComplete()
    .blockingAwait();

// prints:
// Info: Processing of item "1" completed
// Info: Processing of item "3" completed
// Error: Processing of item "2" failed!
```


just()

concatMap()



8、flatMap

作用于Flowable、Observable、Maybe、Single。与contactMap类似，只是contactMap的数据发射是有序的，而flatMap是无序的。

```
Observable.just("A", "B", "C")
    .flatMap(a -> {
        return Observable.intervalRange(1, 3, 0, 1, TimeUnit.SECONDS)
            .map(b -> '(' + a + ", " + b + ')');
    })
    .blockingSubscribe(System.out::println);

// prints (not necessarily in this order):
// (A, 1)
// (C, 1)
// (B, 1)
// (A, 2)
// (C, 2)
// (B, 2)
// (A, 3)
// (C, 3)
// (B, 3)
```

9、flatMapXXX 和 contactMapXXX

太多了，减少篇幅，大家感兴趣自己查阅官网吧。功能与flatMap和contactMap类似。

10、flattenAsFlowable & flattenAsObservable

作用于Maybe、Single，将其转化为Flowable，或Observable。

```
Single<Double> source = Single.just(2.0);
Flowable<Double> flowable = source.flattenAsFlowable(x -> {
    return List.of(x, Math.pow(x, 2), Math.pow(x, 3));
});

flowable.subscribe(x -> System.out.println("onNext: " + x));

// prints:
// onNext: 2.0
// onNext: 4.0
// onNext: 8.0
```

11、groupBy

作用于Flowable、Observable。根据一定的规则对数据源进行分组。

```
Observable<String> animals = Observable.just(
    "Tiger", "Elephant", "Cat", "Chameleon", "Frog", "Fish", "Turtle",
    "Flamingo");

animals.groupBy(animal -> animal.charAt(0), String::toUpperCase)
    .concatMapSingle(Observable::toList)
    .subscribe(System.out::println);

// prints:
// [TIGER, TURTLE]
// [ELEPHANT]
// [CAT, CHAMELEON]
// [FROG, FISH, FLAMINGO]
```

12、scan

作用于Flowable、Observable。对数据进行相关联操作，例如聚合等。

```
Observable.just(5, 3, 8, 1, 7)
    .scan(0, (partialSum, x) -> partialSum + x)
    .subscribe(System.out::println);

// prints:
// 0
// 5
// 8
// 16
// 17
// 24
```

13、window

对数据源发射出来的数据进行收集，按照指定的数量进行分组，以组的形式重新发射。

```
Observable.range(1, 4)
    // Create windows containing at most 2 items, and skip 3 items before
    // starting a new window.
    .window(2)
    .flatMapSingle(window -> {
        return window.map(String::valueOf)
            .reduce(new StringJoiner(", ", "[", "]"), StringJoiner::add);
    })
    .subscribe(System.out::println);

// prints:
// [1, 2]
// [3, 4]
```

错误处理操作符

1、onErrorReturn

作用于Flowable、Observable、Maybe、Single。但调用数据源的onError函数后会回到该函数，可对错误进行处理，然后返回值，会调用观察者onNext()继续执行，执行完调用onComplete()函数结束所有事件的发射。

```
Single.just("2A")
    .map(v -> Integer.parseInt(v, 10))
    .onErrorReturn(error -> {
        if (error instanceof NumberFormatException) return 0;
        else throw new IllegalArgumentException();
    })
    .subscribe(
        System.out::println,
        error -> System.err.println("onError should not be printed!"));

// prints 0
```

2、onErrorReturnItem

与onErrorReturn类似，onErrorReturnItem不对错误进行处理，直接返回一个值。

```
Single.just("2A")
    .map(v -> Integer.parseInt(v, 10))
    .onErrorReturnItem(0)
    .subscribe(
        System.out::println,
        error -> System.err.println("onError should not be printed!"));

// prints 0
```

3、onExceptionResumeNext

可作用于Flowable、Observable、Maybe。onErrorReturn发生异常时，回调onComplete()函数后不再往下执行，而onExceptionResumeNext则是要在处理异常的时候返回一个数据源，然后继续执行，如果返回null，则调用观察者的onError()函数。

```
Observable.create((ObservableOnSubscribe<Integer>) e -> {
    e.onNext(1);
    e.onNext(2);
    e.onNext(3);
    e.onError(new NullPointerException());
    e.onNext(4);
})
    .onErrorResumeNext(throwable -> {
        Log.d(TAG, "onErrorResumeNext ");
        return Observable.just(4);
    })
    .subscribe(new Observer<Integer>() {
        @Override
        public void onSubscribe(Disposable d) {
            Log.d(TAG, "onSubscribe ");
        }
    })
```

```

        @Override
        public void onNext(Integer integer) {
            Log.d(TAG, "onNext " + integer);
        }

        @Override
        public void onError(Throwable e) {
            Log.d(TAG, "onError ");
        }

        @Override
        public void onComplete() {
            Log.d(TAG, "onComplete ");
        }
    });

```

结果：

```

/RxJava: onSubscribe
/RxJava: onNext 1
/RxJava: onNext 2
/RxJava: onNext 3
/RxJava: onErrorResumeNext
/RxJava: onNext 4
/RxJava: onComplete

```

onExceptionResumeNext操作符也是类似的，只是捕获Exception。

4、retry

可作用于所有的数据源，当发生错误时，数据源重复发射item，直到没有异常或者达到所指定的次数。

```

boolean first=true;

Observable.create((ObservableOnSubscribe<Integer>) e -> {
    e.onNext(1);
    e.onNext(2);

    if (first){
        first=false;
        e.onError(new NullPointerException());
    }
})

```

```

    })

    .retry(9)
    .subscribe(new Observer<Integer>() {
        @Override
        public void onSubscribe(Disposable d) {
            Log.d(TAG, "onSubscribe ");
        }

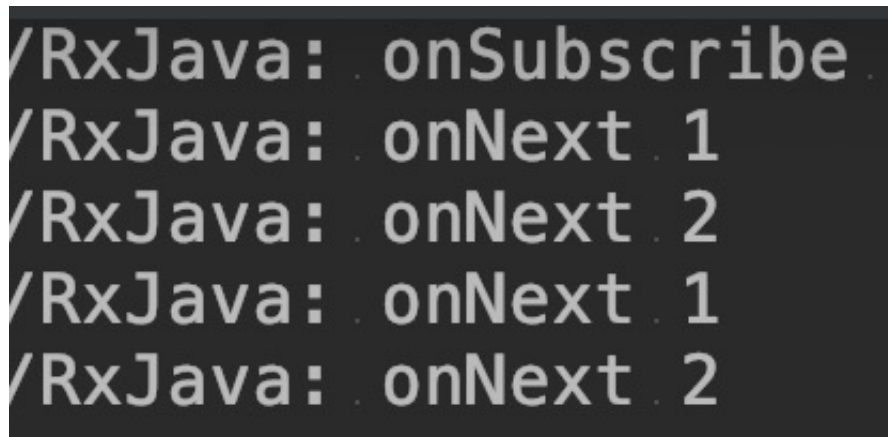
        @Override
        public void onNext(Integer integer) {
            Log.d(TAG, "onNext " + integer);
        }

        @Override
        public void onError(Throwable e) {
            Log.d(TAG, "onError ");
        }

        @Override
        public void onComplete() {
            Log.d(TAG, "onComplete ");
        }
    });

```

结果：



```

/RxJava: onSubscribe
/RxJava: onNext 1
/RxJava: onNext 2
/RxJava: onNext 1
/RxJava: onNext 2

```

5、retryUntil

作用于Flowable、Observable、Maybe。与retry类似，但发生异常时，返回值是false表示继续执行(重复发射数据)，true不再执行,但会调用onError方法。

```

Observable.create((ObservableOnSubscribe<Integer>) e -> {
    e.onNext(1);
    e.onNext(2);
    e.onError(new NullPointerException());
    e.onNext(3);

```

```

        e.onComplete();
    })

    .retryUntil(() -> true)
    .subscribe(new Observer<Integer>() {
        @Override
        public void onSubscribe(Disposable d) {
            Log.d(TAG, "onSubscribe ");
        }

        @Override
        public void onNext(Integer integer) {
            Log.d(TAG, "onNext " + integer);
        }

        @Override
        public void onError(Throwable e) {
            Log.d(TAG, "onError ");
        }

        @Override
        public void onComplete() {
            Log.d(TAG, "onComplete ");
        }
    });

```

结果：

```

/RxJava: onSubscribe
/RxJava: onNext 1
/RxJava: onNext 2
/RxJava: onError

```

retryWhen与此类似，但其判断标准不是BooleanSupplier对象的getAsBoolean()函数的返回值。而是返回的 Observable或Flowable是否会发射异常事件。

总结

太多操作符太累了，看得心好累。还是根据需要查阅文档才是正确的姿势。本文写的操作符只是冰山一角，更多请参阅官网。