This page shows operators with which you can transform items that are emitted by reactive sources, such as `Observable`s.

Outline

buffer

Available in: ✅ `Flowable`, ✅ `Observable`, ⭕ `Maybe`, ⭕ `Single`, ⭕ `Completable`

ReactiveX documentation: http://reactivex.io/documentation/operators/buffer.html

Collects the items emitted by a reactive source into buffers, and emits these buffers.

buffer example

```
Observable.range(0, 10)
    .buffer(4)
```

```
    .subscribe((List<Integer> buffer) -> System.out.println(buffer));

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

cast

Available in:  `Flowable`,  `Observable`,  `Maybe`,  `Single`,  `Completable`

ReactiveX documentation: http://reactivex.io/documentation/operators/map.html

Converts each item emitted by a reactive source to the specified type, and emits these items.

cast example

```
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
```

concatMap

Available in:  `Flowable`,  `Observable`,  `Maybe`,  `Single`,  `Completable`

ReactiveX documentation: http://reactivex.io/documentation/operators/flatmap.html

Applies the given `io.reactivex.functions.Function` to each item emitted by a reactive source, where that function returns a reactive source, and emits the items that result from concatenating the results of these function applications.

concatMap example

```
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
```

concatMapCompletable

Available in: ✅ `Flowable`, ✅ `Observable`, ⭕ `Maybe`, ⭕ `Single`, ⭕ `Completable`

ReactiveX documentation: http://reactivex.io/documentation/operators/flatmap.html

Applies the given `io.reactivex.functions.Function` to each item emitted by a reactive source, where that function returns a `io.reactivex.CompletableSource`, subscribes to them one at a time and returns a `Completable` that completes when all sources completed.

concatMapCompletable example

```
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
```

concatMapCompletableDelayError

Available in: ✅ `Flowable`, ✅ `Observable`, ⭕ `Maybe`, ⭕ `Single`, ⭕ `Completable`

ReactiveX documentation: http://reactivex.io/documentation/operators/flatmap.html

Applies the given `io.reactivex.functions.Function` to each item emitted by a reactive source, where that function returns a `io.reactivex.CompletableSource`, subscribes to them one at a time and returns a `Completable` that completes when all sources completed. Any errors from the sources will be delayed until all of them terminate.

concatMapCompletableDelayError example

```
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!
```

concatMapDelayError

Available in:      `Flowable`, `Observable`, `Maybe`, `Single`, `Completable`

ReactiveX documentation: http://reactivex.io/documentation/operators/flatmap.html

Applies the given `io.reactivex.functions.Function` to each item emitted by a reactive source, where that function returns a reactive source, and emits the items that result from concatenating the results of these function applications. Any errors from the sources will be delayed until all of them terminate.

concatMapDelayError example

```
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!
```

concatMapEager

Available in: ✅ `Flowable`, ✅ `Observable`, ⭕ `Maybe`, ⭕ `Single`, ⭕ `Completable`

ReactiveX documentation: http://reactivex.io/documentation/operators/flatmap.html

Applies the given `io.reactivex.functions.Function` to each item emitted by a reactive source, where that function returns a reactive source, and emits the items that result from concatenating the results of these function applications. Unlike `concatMap`, this operator eagerly subscribes to all sources.

concatMapEager example

```
Observable.range(0, 5)
    .concatMapEager(i -> {
        long delay = Math.round(Math.random() * 3);

        return Observable.timer(delay, TimeUnit.SECONDS)
            .map(n -> i)
            .doOnNext(x -> System.out.println("Info: Finished processing item " +
x));
        })
        .blockingSubscribe(i -> System.out.println("onNext: " + i));

// prints (lines beginning with "Info..." can be displayed in a different order):
// Info: Finished processing item 2
// Info: Finished processing item 0
// onNext: 0
// Info: Finished processing item 1
// onNext: 1
// onNext: 2
// Info: Finished processing item 3
// Info: Finished processing item 4
// onNext: 3
// onNext: 4
```

concatMapEagerDelayError

Available in: ✅ `Flowable`, ✅ `Observable`, ⭕ `Maybe`, ⭕ `Single`, ⭕ `Completable`

ReactiveX documentation: http://reactivex.io/documentation/operators/flatmap.html

Applies the given `io.reactivex.functions.Function` to each item emitted by a reactive source, where that function returns a reactive source, and emits the items that result from concatenating the results of these function applications. A `boolean` value must be specified, which if `true` indicates that all errors from all sources will be

delayed until the end, otherwise if `false`, an error from the main source will be signalled when the current source terminates. Unlike concatMapDelayError, this operator eagerly subscribes to all sources.

concatMapEagerDelayError example

```
Observable<Integer> source = Observable.create(emitter -> {
    emitter.onNext(1);
    emitter.onNext(2);
    emitter.onError(new Error("Fatal error!"));
});

source.doOnError(error -> System.out.println("Info: Error from main source " +
error.getMessage()))
    .concatMapEagerDelayError(x -> {
        return Observable.timer(1, TimeUnit.SECONDS).map(n -> x)
            .doOnSubscribe(it -> System.out.println("Info: Processing of item \"" +
x + "\" started"));
    }, true)
    .blockingSubscribe(
        x -> System.out.println("onNext: " + x),
        error -> System.out.println("onError: " + error.getMessage()));

// prints:
// Info: Processing of item "1" started
// Info: Processing of item "2" started
// Info: Error from main source Fatal error!
// onNext: 1
// onNext: 2
// onError: Fatal error!
```

concatMapIterable

Available in: ✅ `Flowable`, ✅ `Observable`, ⭕ `Maybe`, ⭕ `Single`, ⭕ `Completable`

ReactiveX documentation: http://reactivex.io/documentation/operators/flatmap.html

Applies the given `io.reactivex.functions.Function` to each item emitted by a reactive source, where that function returns a `java.lang.Iterable`, and emits the items that result from concatenating the results of these function applications.

concatMapIterable example

```
Observable.just("A", "B", "C")
    .concatMapIterable(item -> List.of(item, item, item))
    .subscribe(System.out::print);

// prints AAABBBCCC
```

concatMapMaybe

Available in: ✅ `Flowable`, ✅ `Observable`, ⭕ `Maybe`, ⭕ `Single`, ⭕ `Completable`

ReactiveX documentation: http://reactivex.io/documentation/operators/flatmap.html

Applies the given `io.reactivex.functions.Function` to each item emitted by a reactive source, where that function returns a `io.reactivex.MaybeSource`, and emits the items that result from concatenating these `MaybeSource`s.

concatMapMaybe example

```
Observable.just("5", "3,14", "2.71", "FF")
    .concatMapMaybe(v -> {
        return Maybe.fromCallable(() -> Double.parseDouble(v))
            .doOnError(e -> System.out.println("Info: The value \"" + v + "\" could
not be parsed."))

            // Ignore values that can not be parsed.
            .onErrorComplete();
    })
    .subscribe(x -> System.out.println("onNext: " + x));

// prints:
// onNext: 5.0
// Info: The value "3,14" could not be parsed.
// onNext: 2.71
// Info: The value "FF" could not be parsed.
```

concatMapMaybeDelayError

Available in: ✅ `Flowable`, ✅ `Observable`, ⭕ `Maybe`, ⭕ `Single`, ⭕ `Completable`

ReactiveX documentation: http://reactivex.io/documentation/operators/flatmap.html

Applies the given `io.reactivex.functions.Function` to each item emitted by a reactive source, where that function returns a `io.reactivex.MaybeSource`, and emits the items that result from concatenating these `MaybeSource`s. Any errors from the sources will be delayed until all of them terminate.

concatMapMaybeDelayError example

```
DateTimeFormatter dateFormatter = DateTimeFormatter.ofPattern("dd.MM.uuuu");
Observable.just("04.03.2018", "12-08-2018", "06.10.2018", "01.12.2018")
    .concatMapMaybeDelayError(date -> {
        return Maybe.fromCallable(() -> LocalDate.parse(date, dateFormatter));
    })
```

```
    .subscribe(
        localDate -> System.out.println("onNext: " + localDate),
        error -> System.out.println("onError: " + error.getMessage()));

// prints:
// onNext: 2018-03-04
// onNext: 2018-10-06
// onNext: 2018-12-01
// onError: Text '12-08-2018' could not be parsed at index 2
```

concatMapSingle

    

Available in: `Flowable`, `Observable`, `Maybe`, `Single`, `Completable`

ReactiveX documentation: http://reactivex.io/documentation/operators/flatmap.html

Applies the given `io.reactivex.functions.Function` to each item emitted by a reactive source, where that function returns a `io.reactivex.SingleSource`, and emits the items that result from concatenating these ``SingleSource`s.

concatMapSingle example

```
Observable.just("5", "3,14", "2.71", "FF")
    .concatMapSingle(v -> {
        return Single.fromCallable(() -> Double.parseDouble(v))
            .doOnError(e -> System.out.println("Info: The value \"" + v + "\" could
not be parsed."))

            // Return a default value if the given value can not be parsed.
            .onErrorReturnItem(42.0);
    })
    .subscribe(x -> System.out.println("onNext: " + x));

// prints:
// onNext: 5.0
// Info: The value "3,14" could not be parsed.
// onNext: 42.0
// onNext: 2.71
// Info: The value "FF" could not be parsed.
// onNext: 42.0
```

concatMapSingleDelayError

    

Available in: `Flowable`, `Observable`, `Maybe`, `Single`, `Completable`

ReactiveX documentation: http://reactivex.io/documentation/operators/flatmap.html

Applies the given `io.reactivex.functions.Function` to each item emitted by a reactive source, where that function returns a `io.reactivex.SingleSource`, and emits the items that result from concatenating the results of these function applications. Any errors from the sources will be delayed until all of them terminate.

concatMapSingleDelayError example

```
DateTimeFormatter dateFormatter = DateTimeFormatter.ofPattern("dd.MM.uuuu");
Observable.just("24.03.2018", "12-08-2018", "06.10.2018", "01.12.2018")
    .concatMapSingleDelayError(date -> {
        return Single.fromCallable(() -> LocalDate.parse(date, dateFormatter));
    })
    .subscribe(
        localDate -> System.out.println("onNext: " + localDate),
        error -> System.out.println("onError: " + error.getMessage()));

// prints:
// onNext: 2018-03-24
// onNext: 2018-10-06
// onNext: 2018-12-01
// onError: Text '12-08-2018' could not be parsed at index 2
```

flatMap

Available in:  `Flowable`,  `Observable`,  `Maybe`,  `Single`,  `Completable`

ReactiveX documentation: http://reactivex.io/documentation/operators/flatmap.html

Applies the given `io.reactivex.functions.Function` to each item emitted by a reactive source, where that function returns a reactive source, and emits the items that result from merging the results of these function applications.

flatMap example

```
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)
```

flatMapCompletable

Available in: ✅ `Flowable`, ✅ `Observable`, ⭕ `Maybe`, ⭕ `Single`, ⭕ `Completable`

ReactiveX documentation: http://reactivex.io/documentation/operators/flatmap.html

Applies the given `io.reactivex.functions.Function` to each item emitted by a reactive source, where that function returns a `io.reactivex.CompletableSource`, and returns a `Completable` that completes when all sources completed.

flatMapCompletable example

```
Observable<Integer> source = Observable.just(2, 1, 3);
Completable completable = source.flatMapCompletable(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 "1" completed
// Info: Processing of item "2" completed
// Info: Processing of item "3" completed
// Info: Processing of all items completed
```

flatMapIterable

Available in: ✅ `Flowable`, ✅ `Observable`, ⭕ `Maybe`, ⭕ `Single`, ⭕ `Completable`

ReactiveX documentation: http://reactivex.io/documentation/operators/flatmap.html

Applies the given `io.reactivex.functions.Function` to each item emitted by a reactive source, where that function returns a `java.lang.Iterable`, and emits the elements from these `Iterable`s.

flatMapIterable example

```
Observable.just(1, 2, 3, 4)
    .flatMapIterable(x -> {
        switch (x % 4) {
```

```
            case 1:
                return List.of("A");
            case 2:
                return List.of("B", "B");
            case 3:
                return List.of("C", "C", "C");
            default:
                return List.of();
        }
    })
    .subscribe(System.out::println);

// prints:
// A
// B
// B
// C
// C
// C
```

flatMapMaybe

Available in: ✅ `Flowable`, ✅ `Observable`, ⭕ `Maybe`, ✅ `Single`, ⭕ `Completable`

ReactiveX documentation: http://reactivex.io/documentation/operators/flatmap.html

Applies the given `io.reactivex.functions.Function` to each item emitted by a reactive source, where that function returns a `io.reactivex.MaybeSource`, and emits the items that result from merging these `MaybeSource`s.

flatMapMaybe example

```
Observable.just(9.0, 16.0, -4.0)
    .flatMapMaybe(x -> {
        if (x.compareTo(0.0) < 0) return Maybe.empty();
        else return Maybe.just(Math.sqrt(x));
    })
    .subscribe(
        System.out::println,
        Throwable::printStackTrace,
        () -> System.out.println("onComplete"));

// prints:
// 3.0
// 4.0
// onComplete
```

flatMapObservable

Available in: ○ `Flowable`, ○ `Observable`, ✓ `Maybe`, ✓ `Single`, ○ `Completable`

ReactiveX documentation: http://reactivex.io/documentation/operators/flatmap.html

Applies the given `io.reactivex.functions.Function` to the item emitted by a `Maybe` or `Single`, where that function returns an `io.reactivex.ObservableSource`, and returns an `Observable` that emits the items emitted by this `ObservableSource`.

flatMapObservable example

```
Single<String> source = Single.just("Kirk, Spock, Chekov, Sulu");
Observable<String> names = source.flatMapObservable(text -> {
    return Observable.fromArray(text.split(","))
            .map(String::strip);
});

names.subscribe(name -> System.out.println("onNext: " + name));

// prints:
// onNext: Kirk
// onNext: Spock
// onNext: Chekov
// onNext: Sulu
```

flatMapPublisher

Available in: ○ `Flowable`, ○ `Observable`, ✓ `Maybe`, ✓ `Single`, ○ `Completable`

ReactiveX documentation: http://reactivex.io/documentation/operators/flatmap.html

Applies the given `io.reactivex.functions.Function` to the item emitted by a `Maybe` or `Single`, where that function returns an `org.reactivestreams.Publisher`, and returns a `Flowable` that emits the items emitted by this `Publisher`.

flatMapPublisher example

```
Single<String> source = Single.just("Kirk, Spock, Chekov, Sulu");
Flowable<String> names = source.flatMapPublisher(text -> {
    return Flowable.fromArray(text.split(","))
            .map(String::strip);
});

names.subscribe(name -> System.out.println("onNext: " + name));

// prints:
// onNext: Kirk
// onNext: Spock
```

```
// onNext: Chekov
// onNext: Sulu
```

flatMapSingle

Available in:  `Flowable`,  `Observable`,  `Maybe`,  `Single`,  `Completable`

ReactiveX documentation: http://reactivex.io/documentation/operators/flatmap.html

Applies the given `io.reactivex.functions.Function` to each item emitted by a reactive source, where that function returns a `io.reactivex.SingleSource`, and emits the items that result from merging these `SingleSource`s.

flatMapSingle example

```
Observable.just(4, 2, 1, 3)
    .flatMapSingle(x -> Single.timer(x, TimeUnit.SECONDS).map(i -> x))
    .blockingSubscribe(System.out::print);

// prints 1234
```

Note: `Maybe::flatMapSingle` returns a `Single` that signals an error notification if the `Maybe` source is empty:

```
Maybe<Object> emptySource = Maybe.empty();
Single<Object> result = emptySource.flatMapSingle(x -> Single.just(x));
result.subscribe(
    x -> System.out.println("onSuccess will not be printed!"),
    error -> System.out.println("onError: Source was empty!"));

// prints:
// onError: Source was empty!
```

Use `Maybe::flatMapSingleElement` -- which returns a `Maybe` -- if you don't want this behaviour.

flatMapSingleElement

Available in:  `Flowable`,  `Observable`,  `Maybe`,  `Single`,  `Completable`

ReactiveX documentation: http://reactivex.io/documentation/operators/flatmap.html

Applies the given `io.reactivex.functions.Function` to the item emitted by a `Maybe`, where that function returns a `io.reactivex.SingleSource`, and returns a `Maybe` that either emits the item emitted by this `SingleSource` or completes if the source `Maybe` just completes.

flatMapSingleElement example

```
Maybe<Integer> source = Maybe.just(-42);
Maybe<Integer> result = source.flatMapSingleElement(x -> {
    return Single.just(Math.abs(x));
});

result.subscribe(System.out::println);

// prints 42
```

flattenAsFlowable

    

Available in: `Flowable`, `Observable`, `Maybe`, `Single`, `Completable`

ReactiveX documentation: http://reactivex.io/documentation/operators/flatmap.html

Applies the given `io.reactivex.functions.Function` to the item emitted by a `Maybe` or `Single`, where that function returns a `java.lang.Iterable`, and returns a `Flowable` that emits the elements from this `Iterable`.

flattenAsFlowable example

```
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
```

flattenAsObservable

    

Available in: `Flowable`, `Observable`, `Maybe`, `Single`, `Completable`

ReactiveX documentation: http://reactivex.io/documentation/operators/flatmap.html

Applies the given `io.reactivex.functions.Function` to the item emitted by a `Maybe` or `Single`, where that function returns a `java.lang.Iterable`, and returns an `Observable` that emits the elements from this `Iterable`.

flattenAsObservable example

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

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

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

groupBy

Available in:  `Flowable`,  `Observable`,  `Maybe`,  `Single`,  `Completable`

ReactiveX documentation: http://reactivex.io/documentation/operators/groupby.html

Groups the items emitted by a reactive source according to a specified criterion, and emits these grouped items as a `GroupedObservable` or `GroupedFlowable`.

groupBy example

```
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]
```

map

Available in:  `Flowable`,  `Observable`,  `Maybe`,  `Single`,  `Completable`

ReactiveX documentation: http://reactivex.io/documentation/operators/map.html

Applies the given `io.reactivex.functions.Function` to each item emitted by a reactive source and emits the results of these function applications.

map example

```
Observable.just(1, 2, 3)
    .map(x -> x * x)
    .subscribe(System.out::println);

// prints:
// 1
// 4
// 9
```

scan

Available in:  `Flowable`,  `Observable`,  `Maybe`,  `Single`,  `Completable`

ReactiveX documentation: http://reactivex.io/documentation/operators/scan.html

Applies the given `io.reactivex.functions.BiFunction` to a seed value and the first item emitted by a reactive source, then feeds the result of that function application along with the second item emitted by the reactive source into the same function, and so on until all items have been emitted by the reactive source, emitting each intermediate result.

scan example

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

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

switchMap

Available in:  `Flowable`,  `Observable`,  `Maybe`,  `Single`,  `Completable`

ReactiveX documentation: http://reactivex.io/documentation/operators/flatmap.html

Applies the given `io.reactivex.functions.Function` to each item emitted by a reactive source, where that function returns a reactive source, and emits the items emitted by the most recently projected of these reactive sources.

switchMap example

```
Observable.interval(0, 1, TimeUnit.SECONDS)
    .switchMap(x -> {
        return Observable.interval(0, 750, TimeUnit.MILLISECONDS)
                .map(y -> x);
    })
    .takeWhile(x -> x < 3)
    .blockingSubscribe(System.out::print);

// prints 001122
```

window

Available in:  `Flowable`,  `Observable`,  `Maybe`,  `Single`,  `Completable`

ReactiveX documentation: http://reactivex.io/documentation/operators/window.html

Collects the items emitted by a reactive source into windows, and emits these windows as a `Flowable` or `Observable`.

window example

```
Observable.range(1, 10)

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

// prints:
// [1, 2]
// [4, 5]
// [7, 8]
// [10]
```