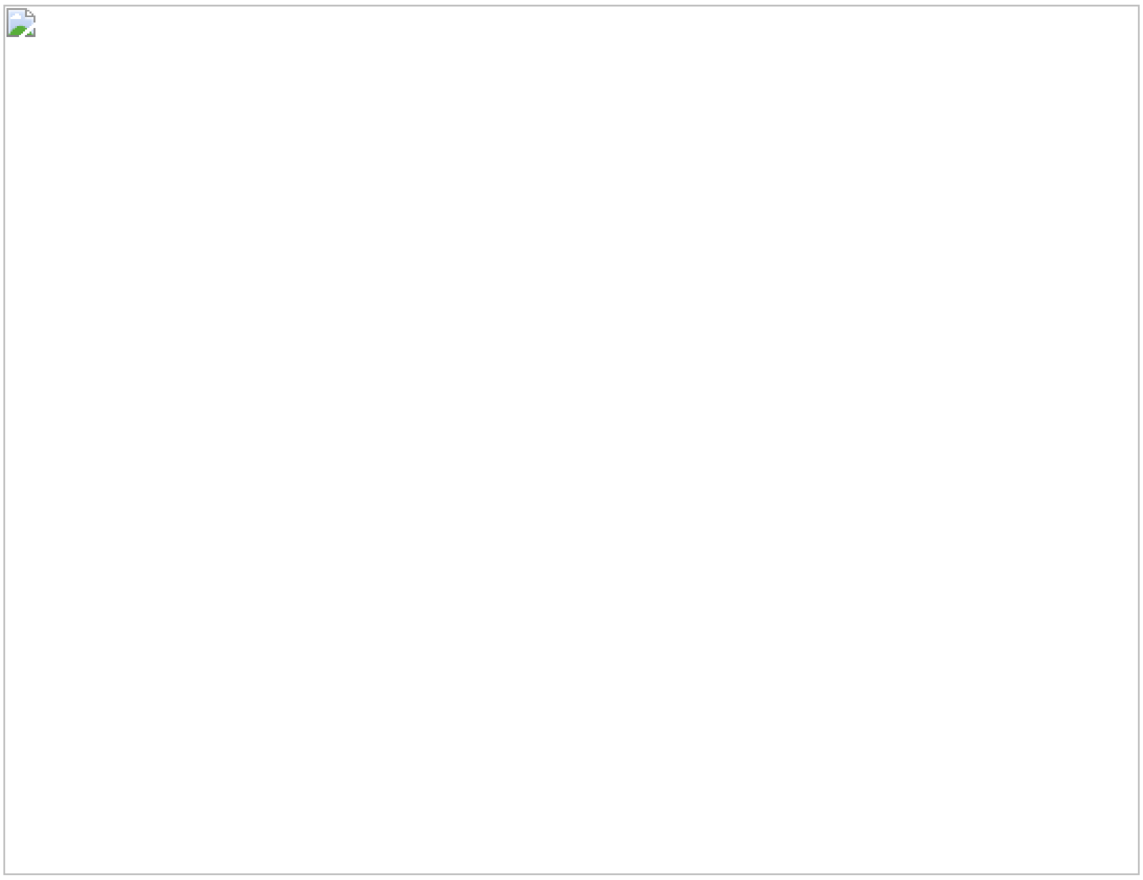
These operators have been proposed but are not part of the 1.0 release of RxJava.

- `chunkify()` — returns an iterable that periodically returns a list of items emitted by the source Observable since the last list
- `fromFuture()` — convert a Future into an Observable, but do not attempt to get the Future's value until a Subscriber subscribes
- `forEachFuture()` — create a futureTask that will invoke a specified function on each item emitted by an Observable
- `forIterable()` — apply a function to the elements of an Iterable to create Observables which are then concatenated
- `fromCancellableFuture()`, `startCancellableFuture()`, and `deferCancellableFuture()` — versions of Future-to-Observable converters that monitor the subscription status of the Observable to determine whether to halt work on the Future
- `generate()` and `generateAbsoluteTime()` — create an Observable that emits a sequence of items as generated by a function of your choosing
- `groupByUntil()` — a variant of the `groupBy` operator that closes any open `GroupedObservable` upon a signal from another Observable
- `multicast()` — represents an Observable as a Connectable Observable
- `onErrorFlatMap()` — instructs an Observable to emit a sequence of items whenever it encounters an error
- `parallel()` — split the work done on the emissions from an Observable into multiple Observables each operating on its own parallel thread
- `parallelMerge()` — combine multiple Observables into a smaller number of Observables, to facilitate parallelism
- `pivot()` — combine multiple sets of grouped observables so that they are arranged primarily by group rather than by set
- `publishLast()` — represent an Observable as a Connectable Observable that emits only the last item emitted by the source Observable

---

chunkify( )

returns an iterable that periodically returns a list of items emitted by the source Observable since the last list

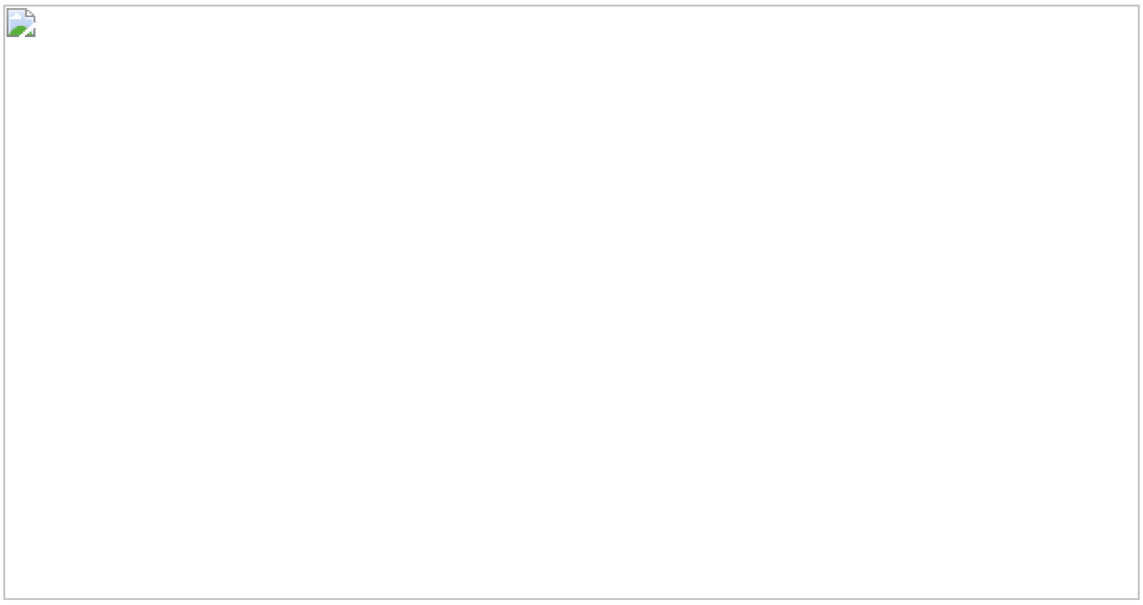


The `chunkify()` operator represents a blocking observable as an Iterable, that, each time you iterate over it, returns a list of items emitted by the source Observable since the previous iteration. These lists may be empty if there have been no such items emitted.

---

fromFuture()

convert a Future into an Observable, but do not attempt to get the Future's value until a Subscriber subscribes

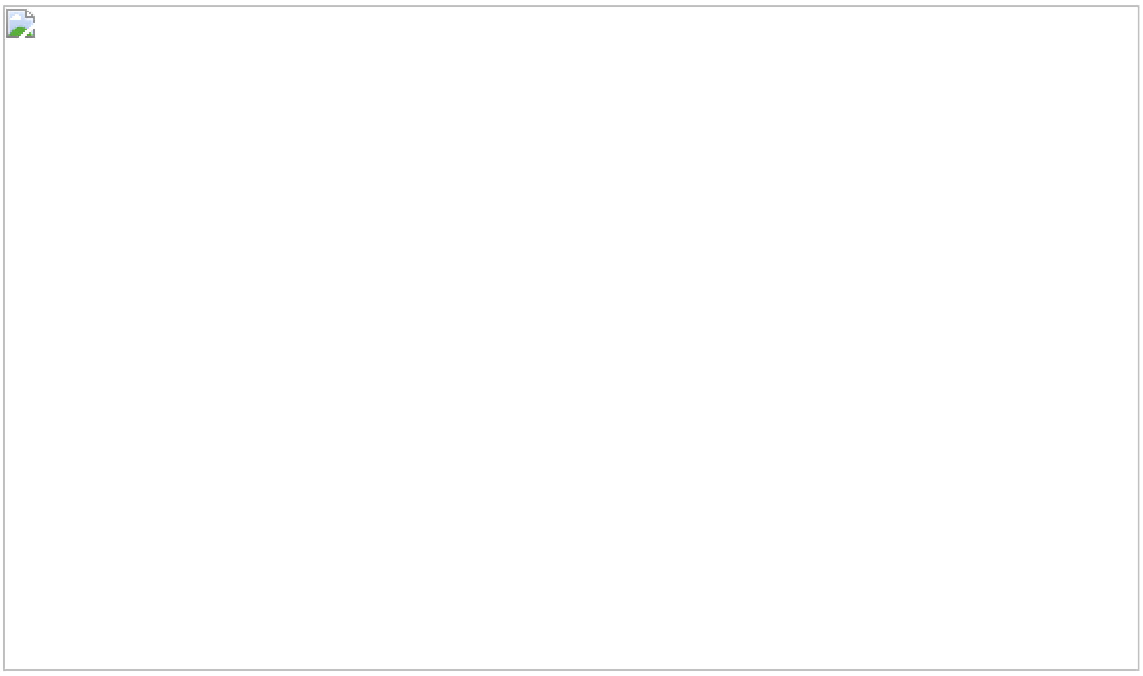


The `fromFuture( )` method also converts a Future into an Observable, but it obtains this Future indirectly, by means of a function you provide. It creates the Observable immediately, but waits to call the function and to obtain the Future until a Subscriber subscribes to it.

---

forEachFuture( )

create a futureTask that will invoke a specified function on each item emitted by an Observable

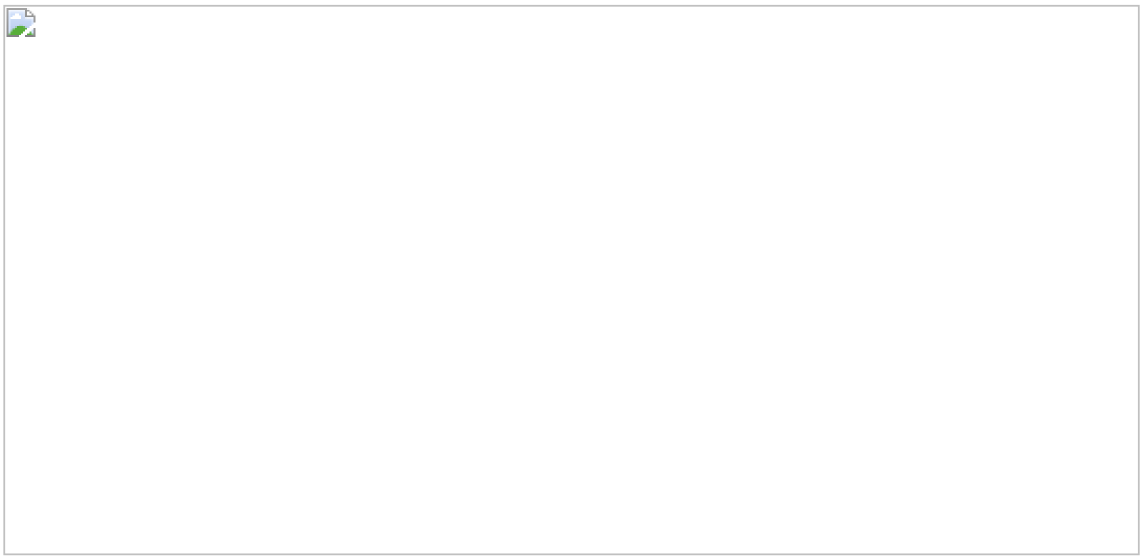


The `forEachFuture( )` returns a `FutureTask` for each item emitted by the source Observable (or each item and each notification) that, when executed, will apply a function you specify to each such item (or item and notification).

---

forIterable( )

apply a function to the elements of an Iterable to create Observables which are then concatenated



`forIterable( )` is similar to `from(Iterable )` but instead of the resulting Observable emitting the elements of the Iterable as its own emitted items, it applies a specified function to each of these elements to generate one Observable per element, and then concatenates the emissions of these Observables to be its own sequence of emitted items.

---

fromCancellableFuture( ), startCancellableFuture( ), and deferCancellableFuture( )

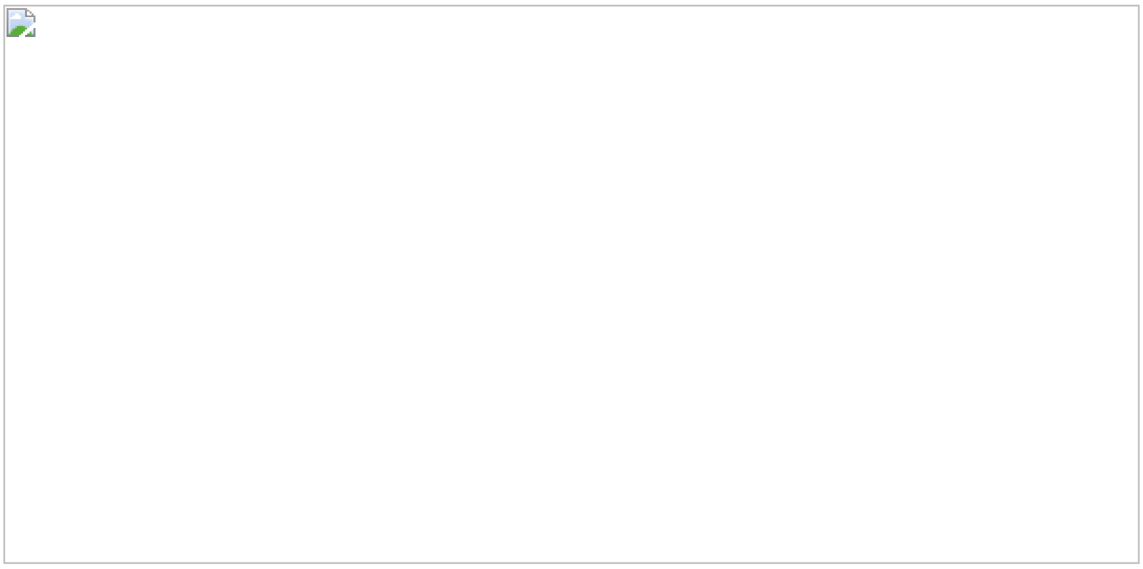
versions of Future-to-Observable converters that monitor the subscription status of the Observable to determine whether to halt work on the Future

If the a subscriber to the Observable that results when a Future is converted to an Observable later unsubscribes from that Observable, it can be useful to have the ability to stop attempting to retrieve items from the Future. The "cancellable" Future enables you do do this. These three methods will return Observables that, when unsubscribed to, will also "unsubscribe" from the underlying Futures.

---

generate( ) and generateAbsoluteTime( )

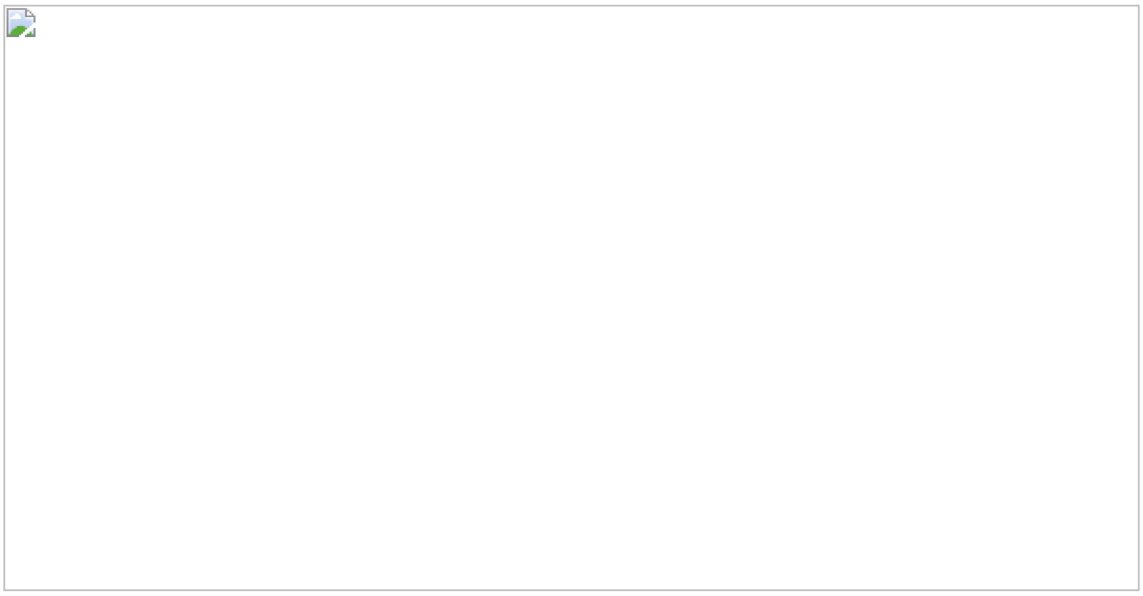
create an Observable that emits a sequence of items as generated by a function of your choosing



The basic form of `generate()` takes four parameters. These are `initialState` and three functions: `iterate()`, `condition()`, and `resultSelector()`. `generate()` uses these four parameters to generate an Observable sequence, which is its return value. It does so in the following way.

`generate()` creates each emission from the sequence by applying the `resultSelector()` function to the current *state* and emitting the resulting item. The first state, which determines the first emitted item, is `initialState`. `generate()` determines each subsequent state by applying `iterate()` to the current state. Before emitting an item, `generate()` tests the result of `condition()` applied to the current state. If the result of this test is `false`, instead of calling `resultSelector()` and emitting the resulting value, `generate()` terminates the sequence and stops iterating the state.

There are also versions of `generate()` that allow you to do the work of generating the sequence on a particular `Scheduler` and that allow you to set the time interval between emissions by applying a function to the current state. The `generateAbsoluteTime()` allows you to control the time at which an item is emitted by applying a function to the state to get an absolute system clock time (rather than an interval from the previous emission).



see also:

- Introduction to Rx: Generate
- Linq: `Generate`
- RxJS: `generate`, `generateWithAbsoluteTime`, and `generateWithRelativeTime`

---

groupByUntil( )

a variant of the `groupBy` operator that closes any open `GroupedObservable` upon a signal from another Observable

This version of `groupBy` adds another parameter: an Observable that emits duration markers. When a duration marker is emitted by this Observable, any grouped Observables that have been opened are closed, and `groupByUntil()` will create new grouped Observables for any subsequent emissions by the source Observable.

Another variety of `groupByUntil()` limits the number of groups that can be active at any particular time. If an item is emitted by the source Observable that would cause the number of groups to exceed this maximum, before the new group is emitted, one of the existing groups is closed (that is, the Observable it represents terminates by calling its Subscribers' `onCompleted` methods and then expires).

---

multicast()

represents an Observable as a Connectable Observable

To represent an Observable as a Connectable Observable, use the `multicast()` method.

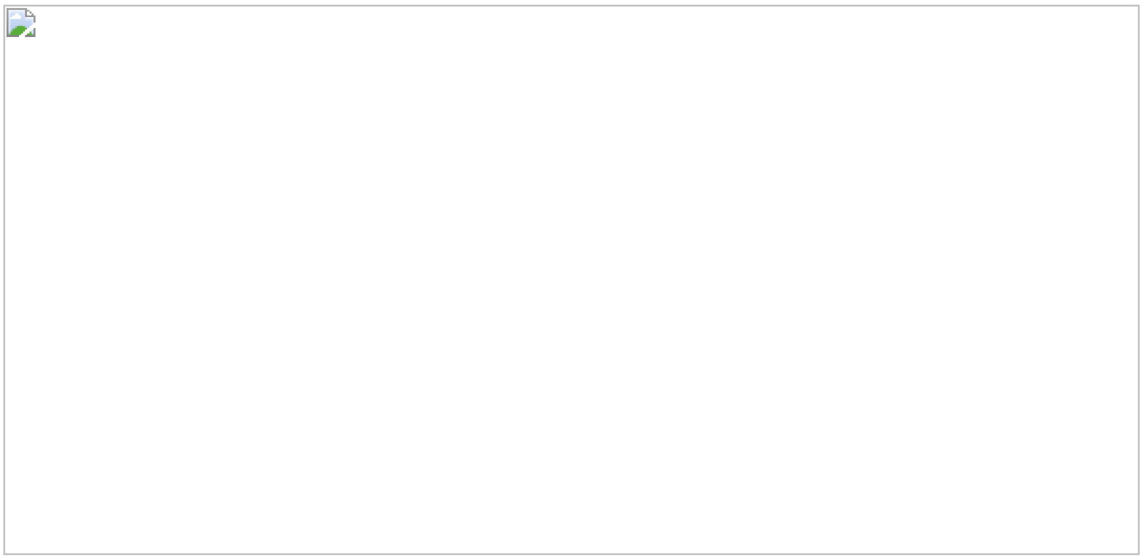
see also:

- javadoc: `multicast(subjectFactory)`
- javadoc: `multicast(subjectFactory, selector)`
- RxJS: `multicast`
- Linq: `Multicast`
- Introduction to Rx: Publish and Connect
- Introduction to Rx: Multicast

---

onErrorFlatMap()

instructs an Observable to emit a sequence of items whenever it encounters an error

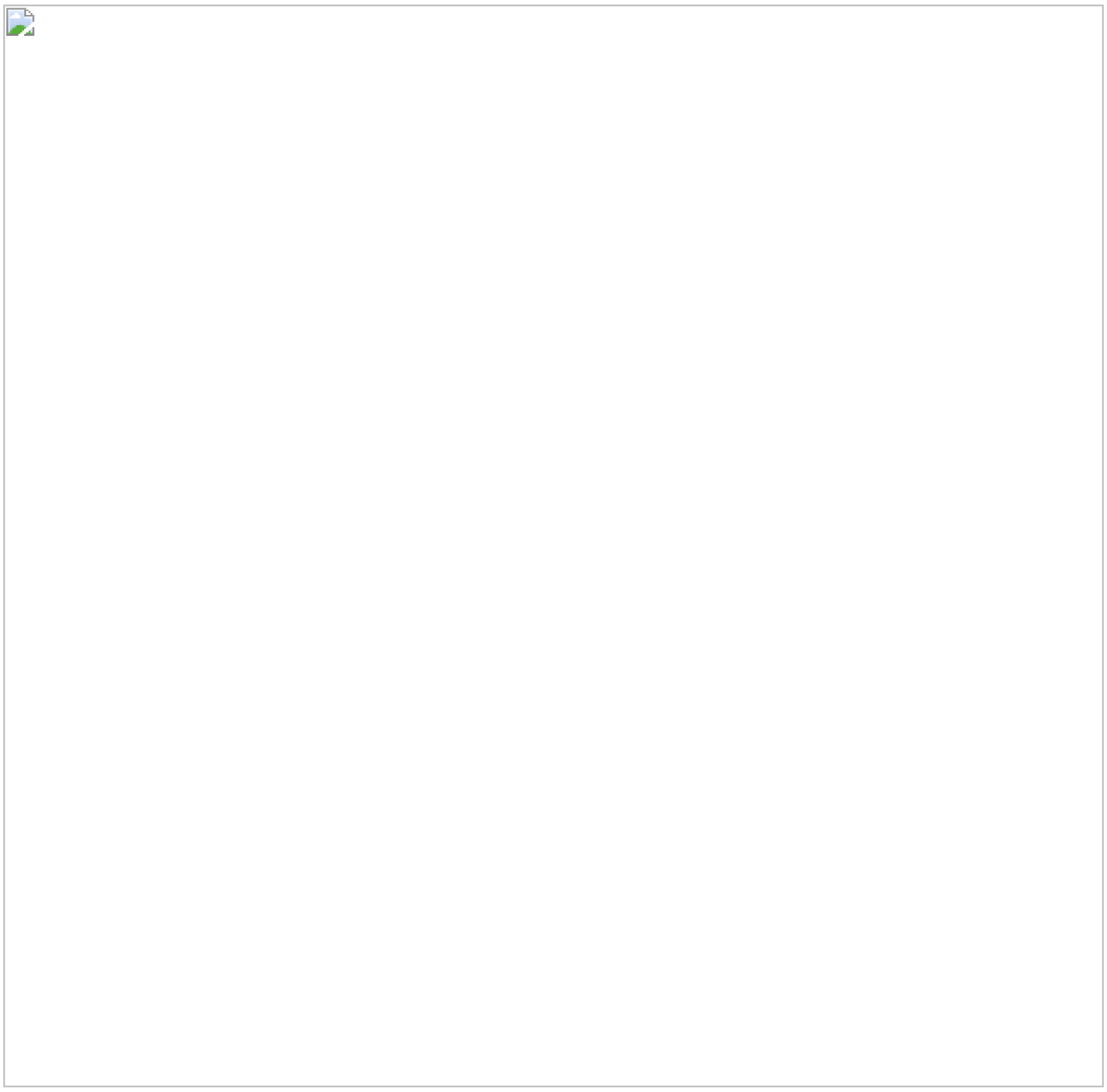


The `onErrorFlatMap()` method is similar to `onErrorResumeNext()` except that it does not assume the source Observable will correctly terminate when it issues an error. Because of this, after emitting its backup sequence of items, `onErrorFlatMap()` relinquishes control of the emitted sequence back to the source Observable. If that Observable again issues an error, `onErrorFlatMap()` will again emit its backup sequence.

The backup sequence is an Observable that is returned from a function that you pass to `onErrorFlatMap()`. This function takes the Throwable issued by the source Observable as its argument, and so you can customize the sequence based on the nature of the Throwable.

Because `onErrorFlatMap()` is designed to work with pathological source Observables that do not terminate after issuing an error, it is mostly useful in debugging/testing scenarios.

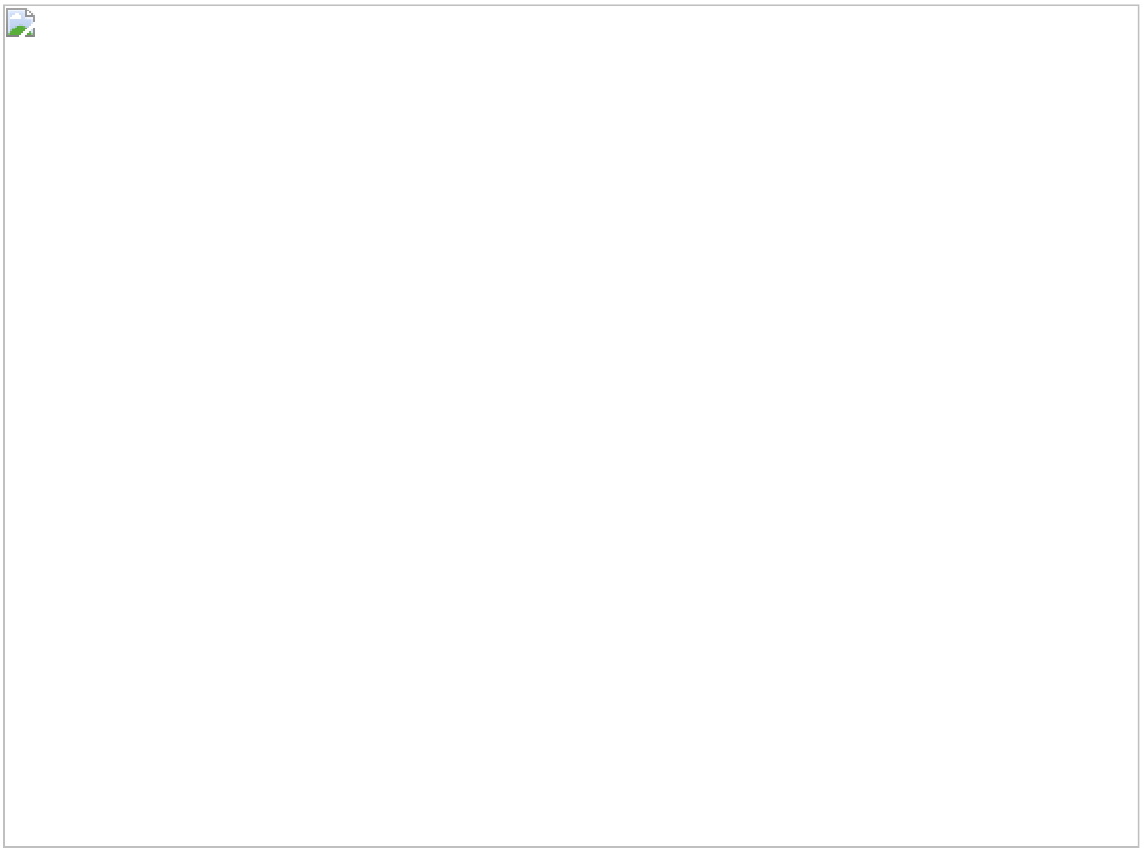
Note that you should apply `onErrorFlatMap()` directly to the pathological source Observable, and not to that Observable after it has been modified by additional operators, as such operators may effectively renormalize the source Observable by unsubscribing from it immediately after it issues an error. Below, for example, is an illustration showing how `onErrorFlatMap()` will respond to two error-generating Observables that have been merged by the `merge()` operator. Note that it will *not* react to both errors generated by both Observables, but only to the single error passed along by `merge()`:



---

parallel( )

split the work done on the emissions from an Observable into multiple Observables each operating on its own parallel thread



The `parallel( )` method splits an Observable into as many Observables as there are available processors, and does work in parallel on each of these Observables. `parallel( )` then merges the results of these parallel computations back into a single, well-behaved Observable sequence.

For the simple "run things in parallel" use case, you can instead use something like this:

```
streamOfItems.flatMap(item -> {
   itemToObservable(item).subscribeOn(Schedulers.io());
});
```

Kick off your work for each item inside `flatMap` using `subscribeOn` to make it asynchronous, or by using a function that already makes asychronous calls.

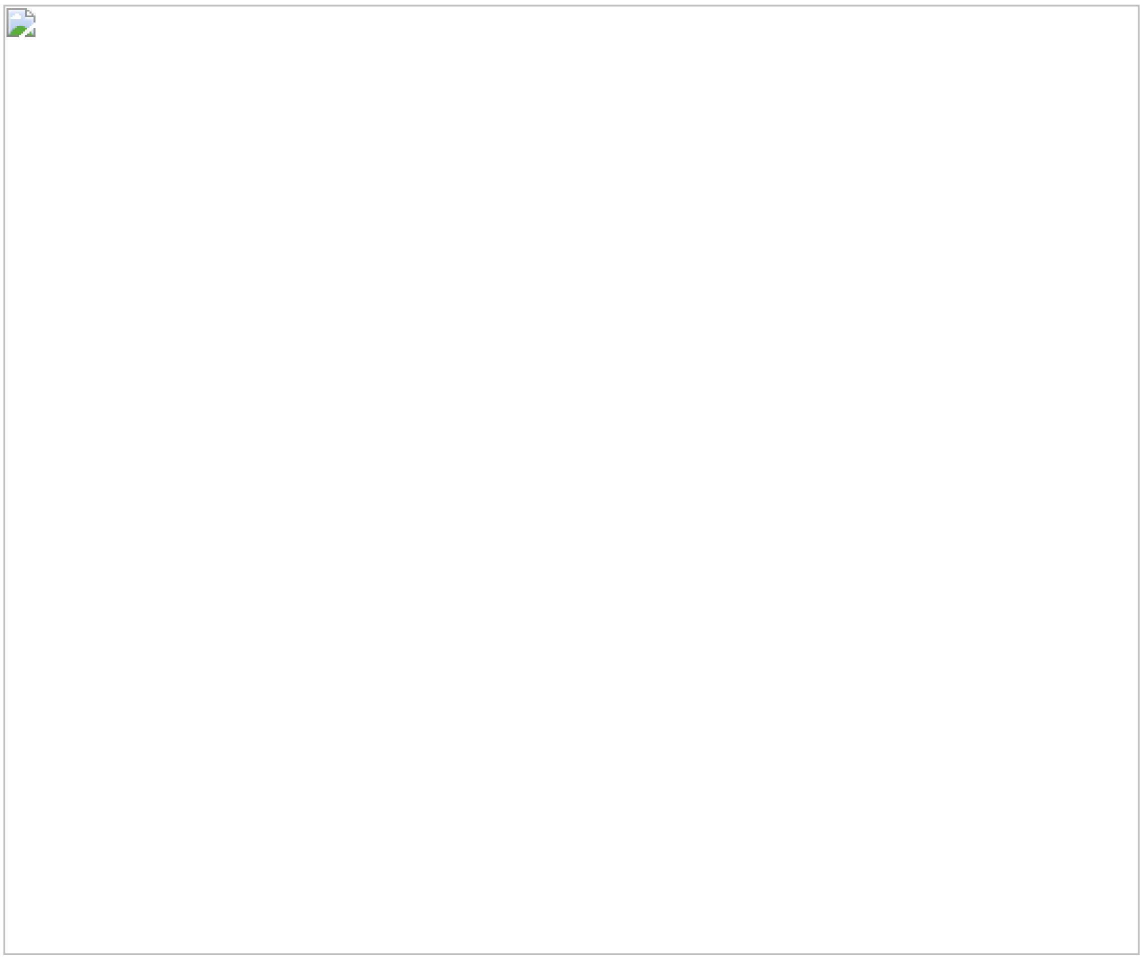
see also:

- RxJava Threading Examples by Graham Lea

---

parallelMerge( )

combine multiple Observables into a smaller number of Observables, to facilitate parallelism

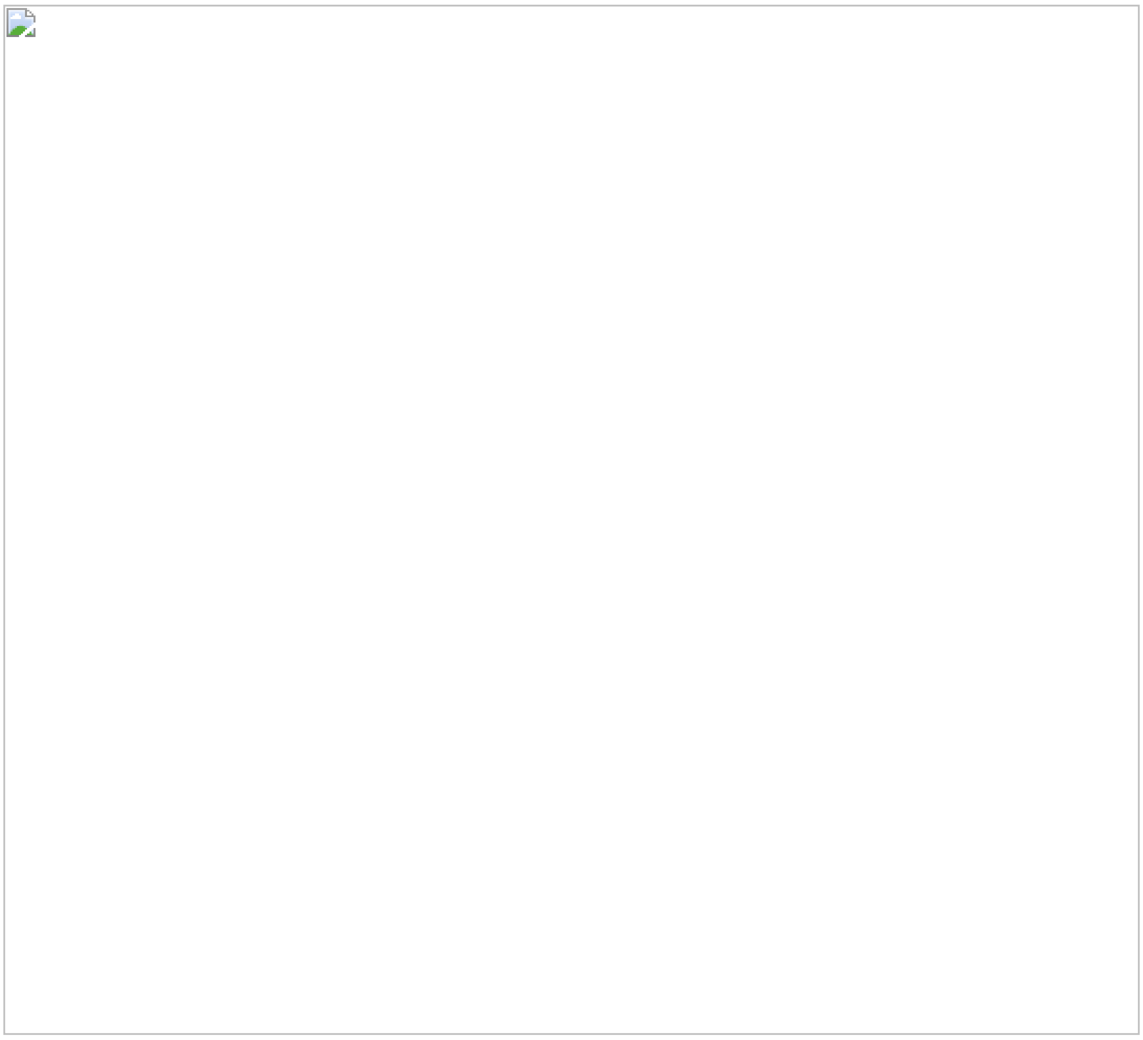


Use the `parallelMerge()` method to take an Observable that emits a large number of Observables and to reduce it to an Observable that emits a particular, smaller number of Observables that emit the same set of items as the original larger set of Observables: for instance a number of Observables that matches the number of parallel processes that you want to use when processing the emissions from the complete set of Observables.

---

pivot( )

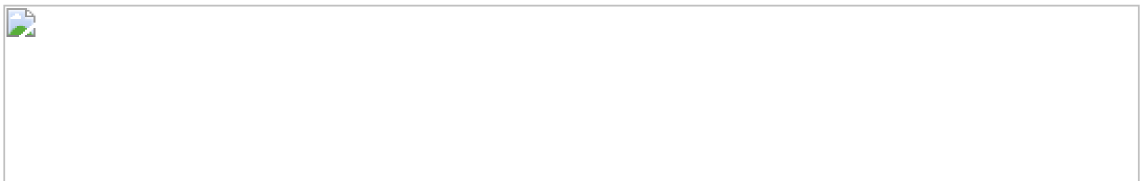
combine multiple sets of grouped observables so that they are arranged primarily by group rather than by set



If you combine multiple sets of grouped observables, such as those created by `groupBy()` and `groupByUntil()`, then even if those grouped observables have been grouped by a similar differentiation function, the resulting grouping will be primarily based on which set the observable came from, not on which group the observable belonged to.

An example may make this clearer. Imagine you use `groupBy()` to group the emissions of an Observable (Observable1) that emits integers into two grouped observables, one emitting the even integers and the other emitting the odd integers. You then repeat this process on a second Observable (Observable2) that emits another set of integers. You hope then to combine the sets of grouped observables emitted by each of these into a single grouped Observable by means of a operator like `from(Observable1, Observable2)`.

The result will be a grouped observable that emits two groups: the grouped observable resulting from transforming Observable1, and the grouped observable resulting from transforming Observable2. Each of those grouped observables emit observables that in turn emit the odds and evens from the source observables. You can use `pivot()` to change this around: by applying `pivot()` to this grouped observable it will transform into one that emits two different groups: the odds group and the evens group, with each of these groups emitting a separate observable corresponding to which source observable its set of integers came from. Here is an illustration:



publishLast()

represent an Observable as a Connectable Observable that emits only the last item emitted by the source Observable

see also:

- RxJS: `publishLast`
- Linq: `PublishLast`
- Introduction to Rx: PublishLast