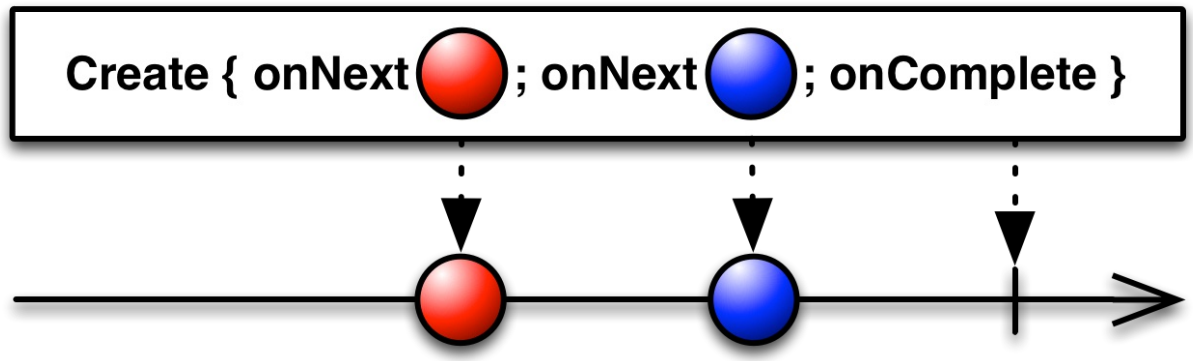
RxJavaOperator 이해와 사용법

manasobi RxJava

Qiita Yuki_Yamada Edited at 2016-10-27

1. 생성

1.1 create



[ReactiveX - Create operator][create] 각각의 Observable의 subscribe에 의해 Observable을 생성하는 오퍼레이터. 비동기처리에서 에러 핸들링이 필요할 때 사용.

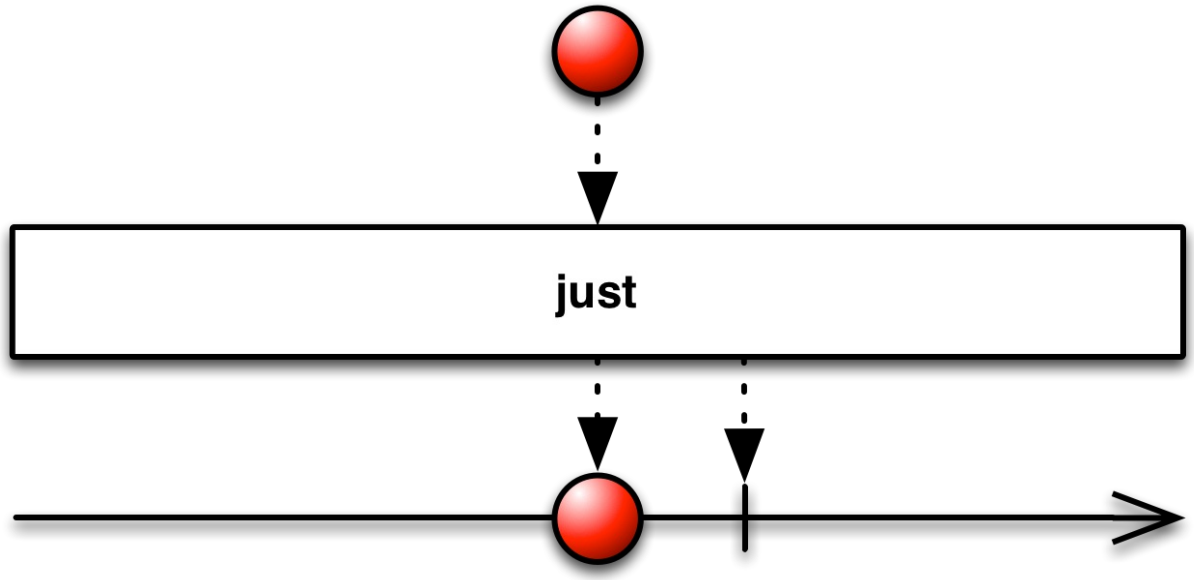
```
Observable<String> observable = Observable.create(
    new Observable.OnSubscribe<String>() {
        @Override
        public void call(Subscriber<? super String> subscriber) {
            try {
                FileInputStream fileInputStream;
                fileInputStream = openFileInput("MyFile.txt");
                byte[] readBytes = new byte[fileInputStream.available()];
                fileInputStream.read(readBytes);
                subscriber.onNext(new String(readBytes));

                fileInputStream = openFileInput("2ndMyFile.txt");
                byte[] readBytes2nd = new byte[fileInputStream.available()];
                fileInputStream.read(readBytes2nd);
                subscriber.onNext(new String(readBytes2nd));

                subscriber.onCompleted();
            } catch (Exception e) {
                subscriber.onError(e);
```

```
            }
        }
    }
```

create를 사용하는 경우는 예최처리가 발생하는 경우라고 생각한다. 자기가 subscribe 발화를 제어하는 느낌이다. 공식 샘플에는 for문에서 observable을 이용하고 있는데 Iterator를 가지고 있지 않는 연속된 데이타를 Observable로 하고 싶은 경우에 사용한다고 할 수 있겠다.

1.2 just

Observable로 하는 값을 직접 지정.



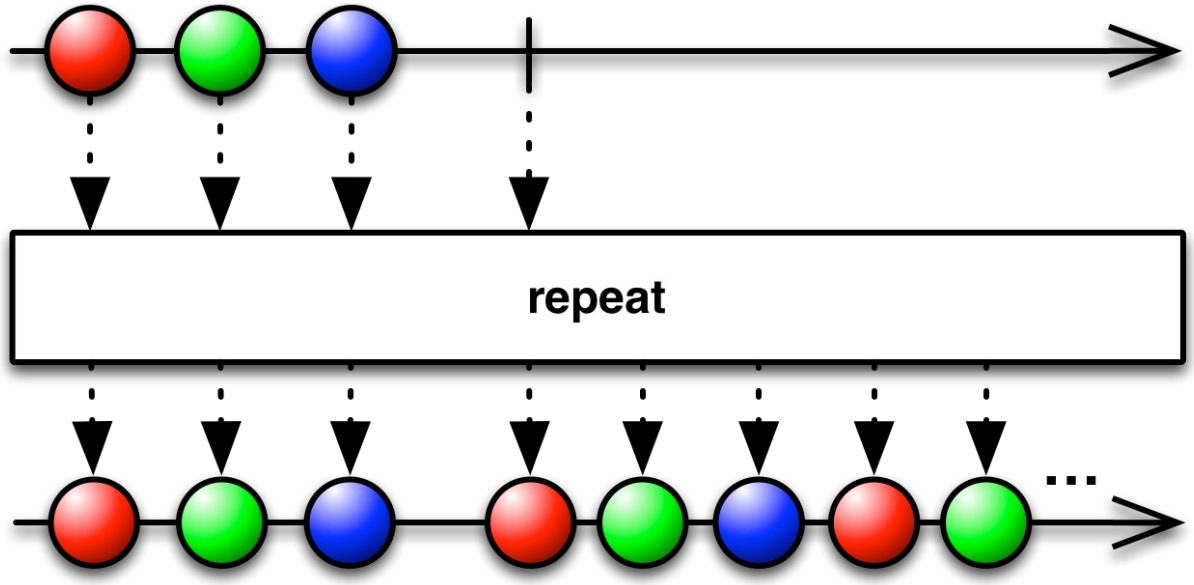
[ReactiveX - Just operator][just] 그림에는 하나밖에 Observable에 흐르고 있지 않지만, 10개까지 지정 가능하다.

```
Observable.just(1,2,3,4,5,6,7,8,9)
```

테스트 등에 사용될 수 있다.

1.3 repeat

onCompleted를 통과했지만 한번더 Subscribe를 해준다.

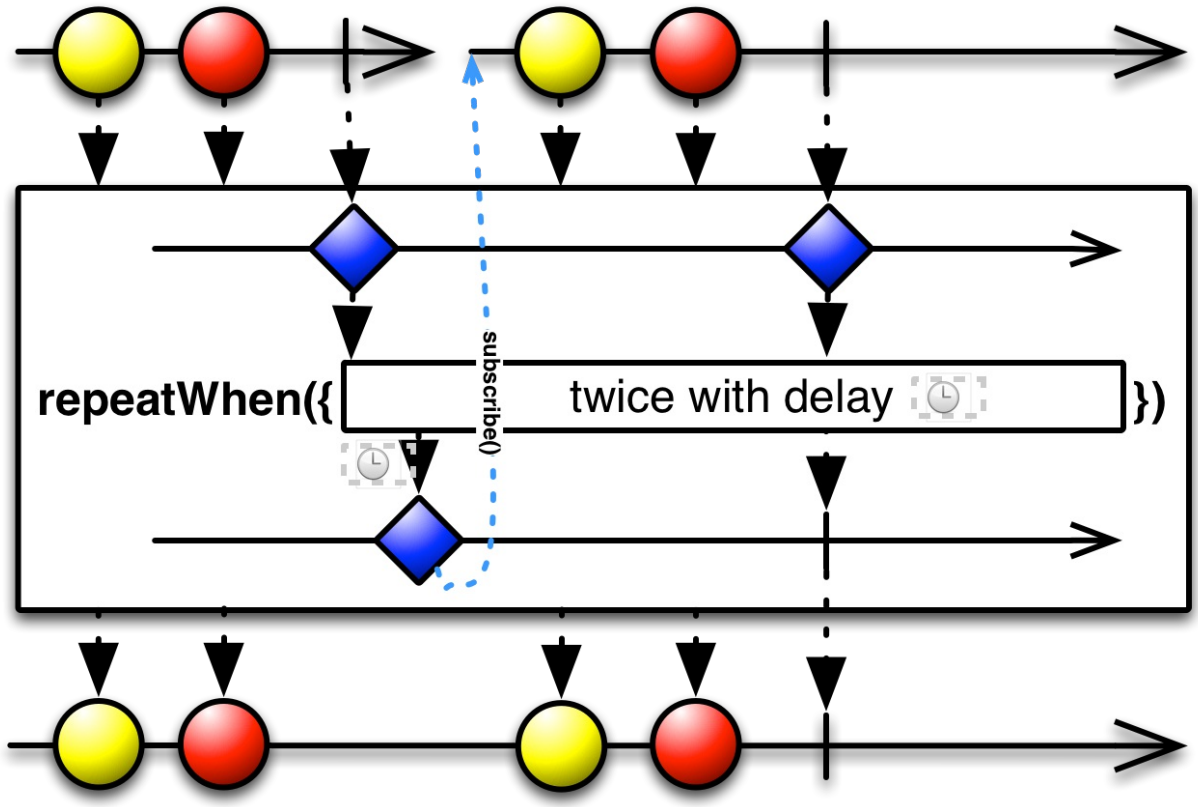


빨강, 초록, 파랑을 Observable로 흐르게하고, repeatOperator를 통과했기때문에 같은 데이터가 반복해서 subscribe된다.

```
Observable.just(1,2,3,4,5,6,7,8,9)
          .repeat(5)
```

1.4 repeatWhen

repeat와 유사 함수로 repeat하는 간격을 지정.

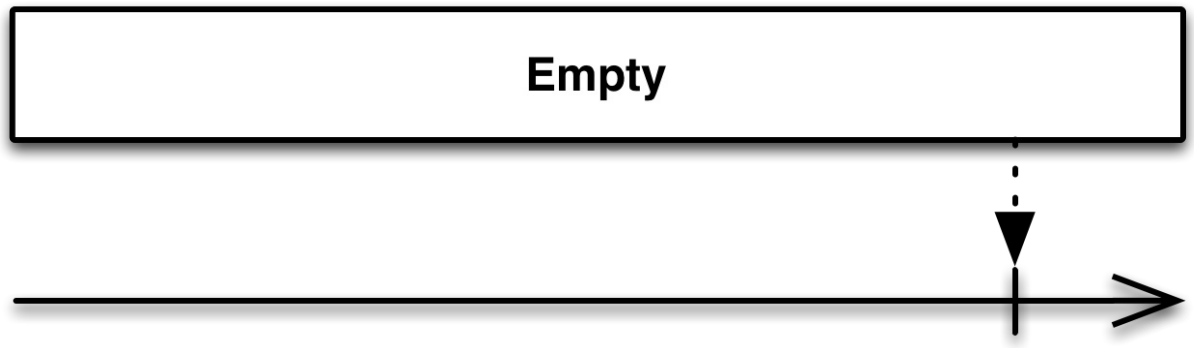


데이터는 노란, 빨간 공. 이것이 막대기에서 종료하면 파란 마름모가 발행되고 delay시간이 포함되어 한번 더 subscribe된다.

```
Observable
    .just(1,2,3,4,5,6,7,8,9)
    .repeatWhen(new Func1<Observable<? extends Void>, Observable<?>>() {
        @Override
        public Observable<?> call(Observable<? extends Void> observable) {
            return observable.delay(5, TimeUnit.DAYS);
        }
    });
```

1.5 Empty/Never/Throw

- Empty는 onComplete만 발행한다.
- Never는 그것조차 호출하지 않는 공백의 Observable이다.
- Throw는 onError를 발행하는 Observable를 발행한다.



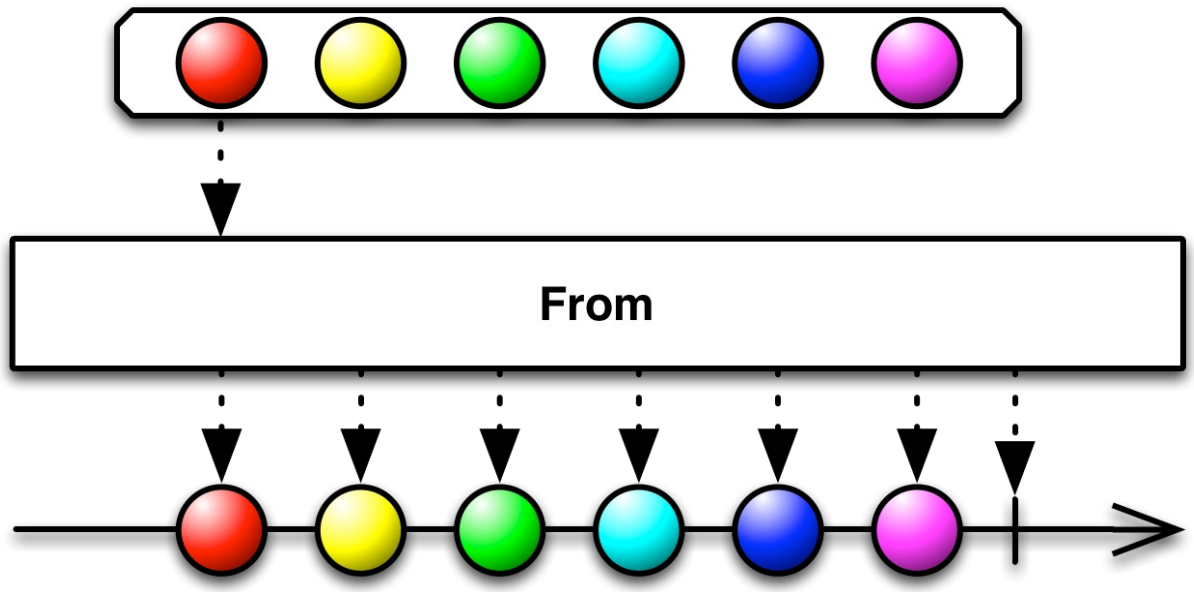
최후에 발화하고 있다. 이것이 OnCompleted이다.

보는 것처럼 발화조차 하지 않는다.

에러의 발화를 하고있다.

1.6 from

Iterator를 가진 오브젝트로부터 Observable을 생성한다.



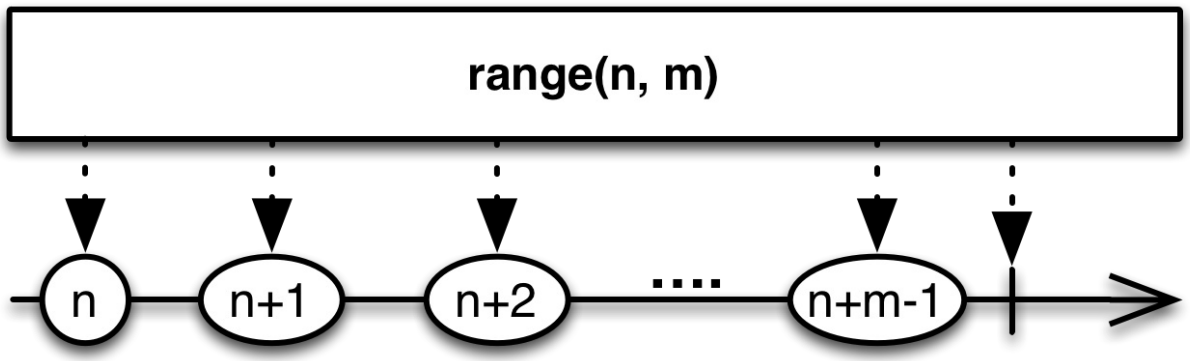

```
String[] array = new String[]{"sasaki","ササキ","ささき","佐々木"};
Observable.from(array)
```

맵으로부터 데이터를 취득하는 경우는 아래와 같다.

```
Observable.from(map.entrySet())
```

1.7 range

int로 시작, 종료를 지정하고 그 사이의 숫자의 Observable을 생성한다.

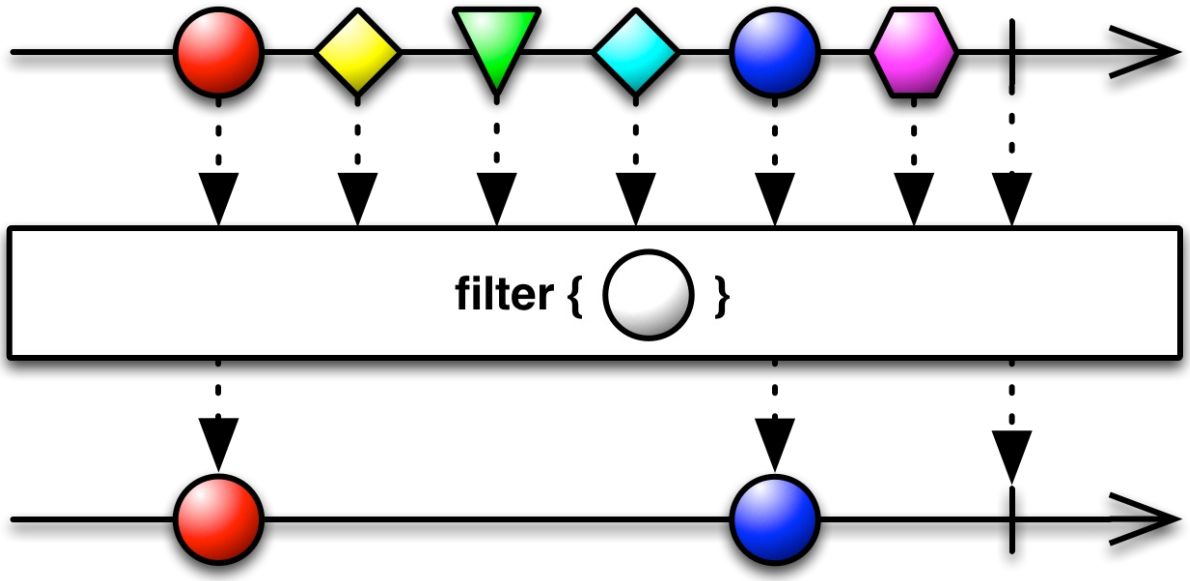



```
Observable.range(1,10);
```

2. 필터링

흘러들어온 데이터에 대하여 onNext를 발행할까 어떨까라는 처리를 주로하는 오퍼레이터이다.

2.1 filter



[ReactiveX - Filter operator][filter] 이 그림에서는 전달된 Observer로부터 동그라미만을 필터링해서, 동그라미 모양만을 onNext한다.

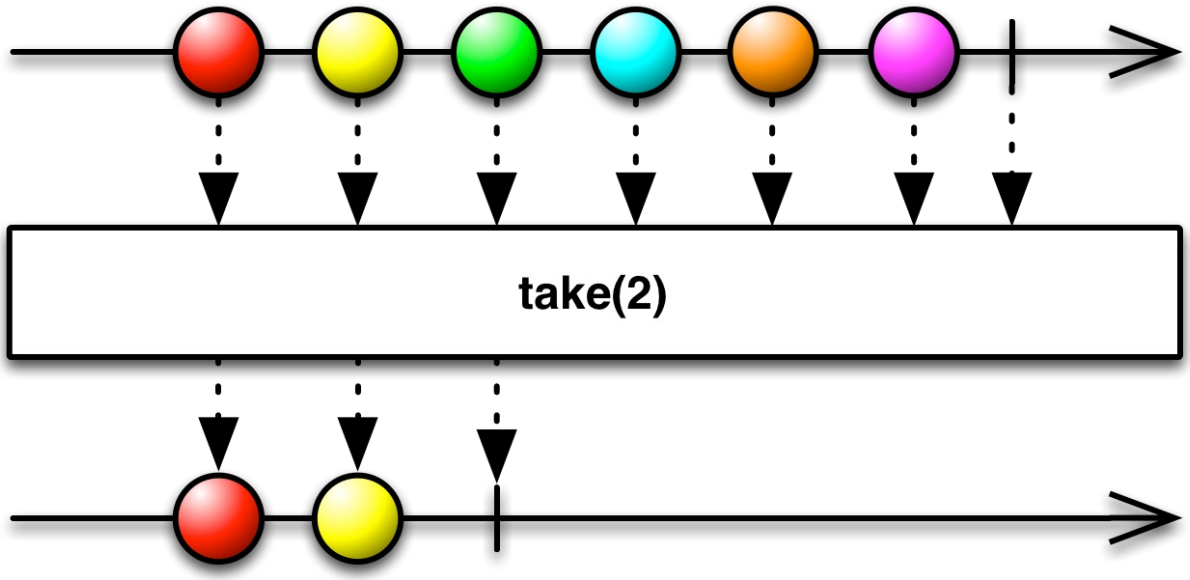
```
Observable.just(1,10,4,21,19,2,13,9)
    .filter(new Func1<Integer, Boolean>() {
        @Override
        public Boolean call(Integer item) {
            return (item < 15);
        }
```

```
    })
```

함수 Func1에서 반환 값이 true이면 그 값이 onNext에 도달한다.

2.2 take

개수를 지정해서 꺼낸다.



[ReactiveX - Take operator][take]

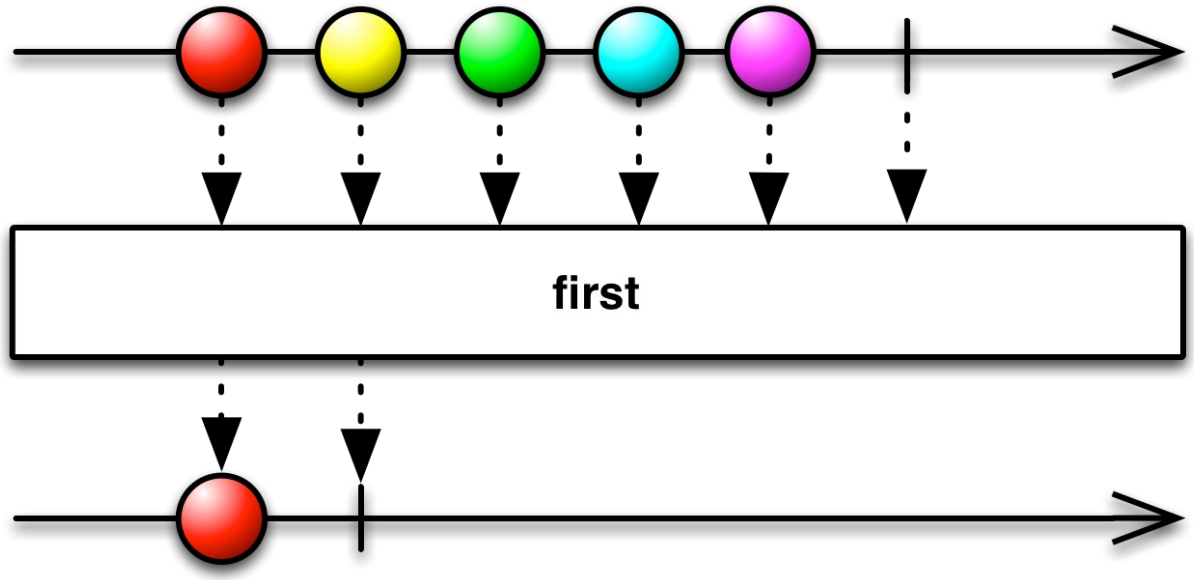
```
Observable.just(1,10,4,21,19,2,13,9)
          .take(2)
```

takeLast를 이용하면 마지막 데이터로부터 데이터를 취득한다.

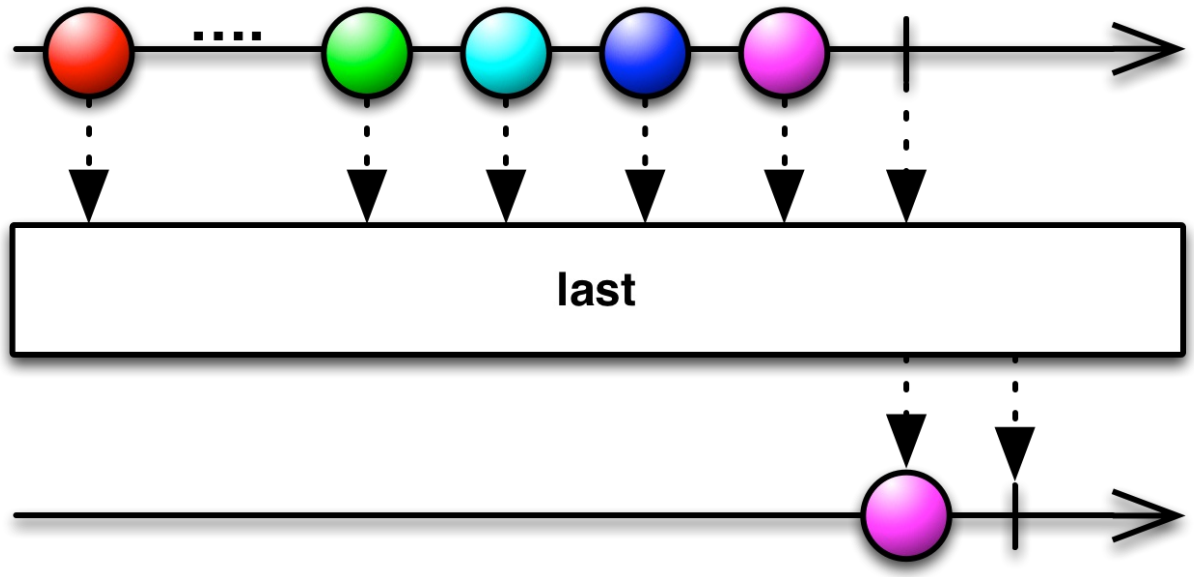
```
Observable.range(1,10)
          .takeLast(3)
```

2.3 first/last/elementAt ..OrDefault

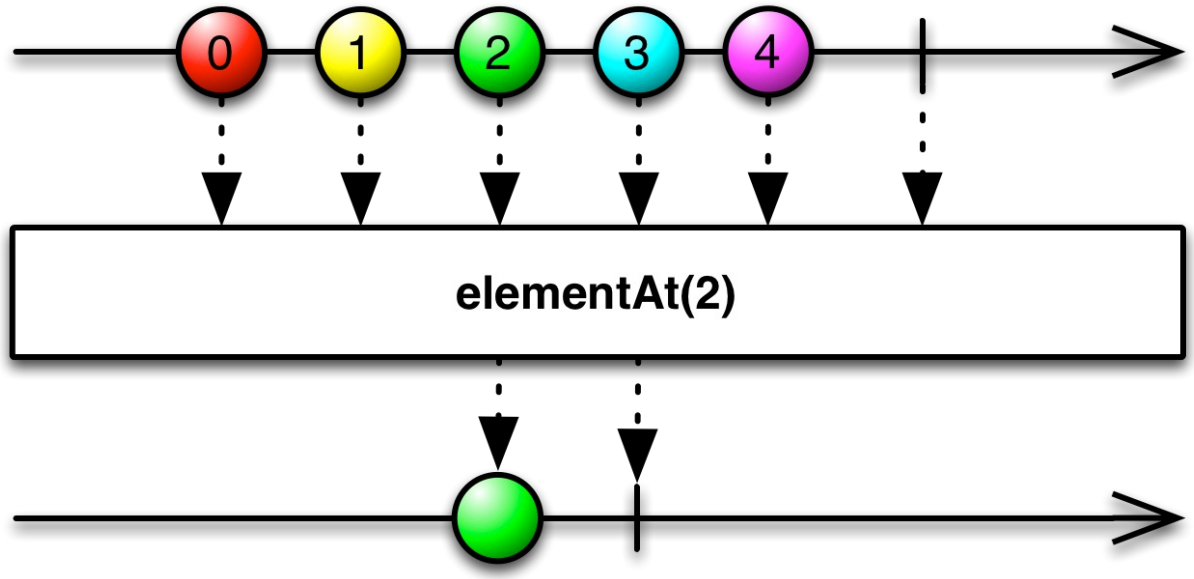
최초(최후, 또는 지정한 곳)만을 취득해서 onNext한다. 요소 하나만을 onNext하는 것이 특징이다.



최초의 빨간색만을 Observable에 흘려보낸다.



제일 마지막 핑크색만을 Observable로 보낸다.

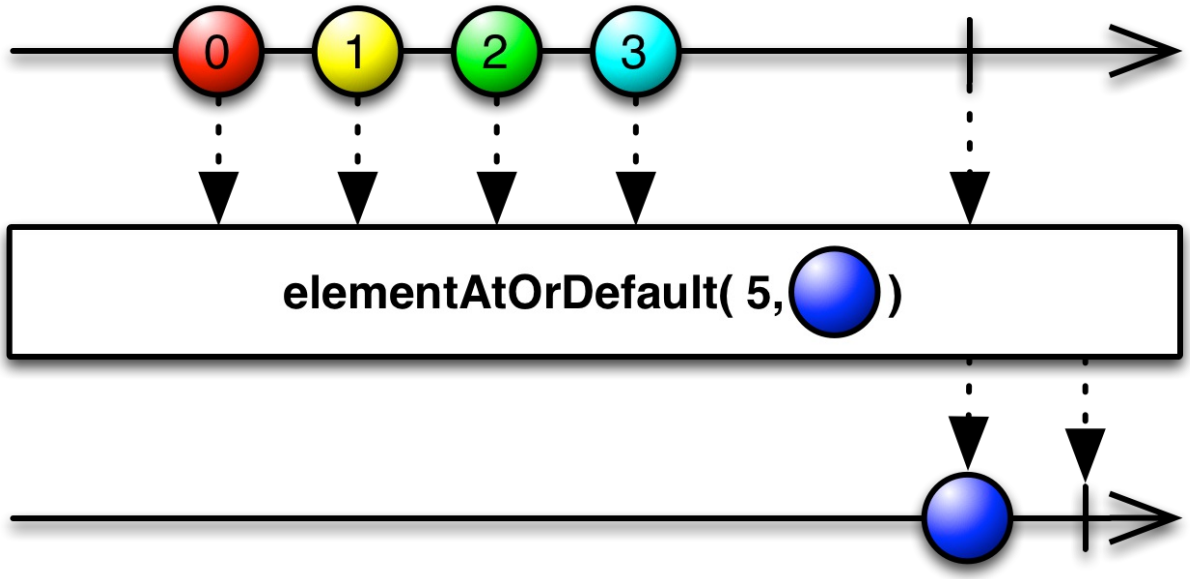


지정한 요소 번호 2번만을 Observable로 보낸다.

```
Observable.just(1,2,3,4,5,6).first();
Observable.just(1,2,3,4,5,6).last();
Observable.just(1,2,3,4,5,6).firstOrDefault(1);
```

OrDefault가 있어서 안전한 설계가 가능하다.

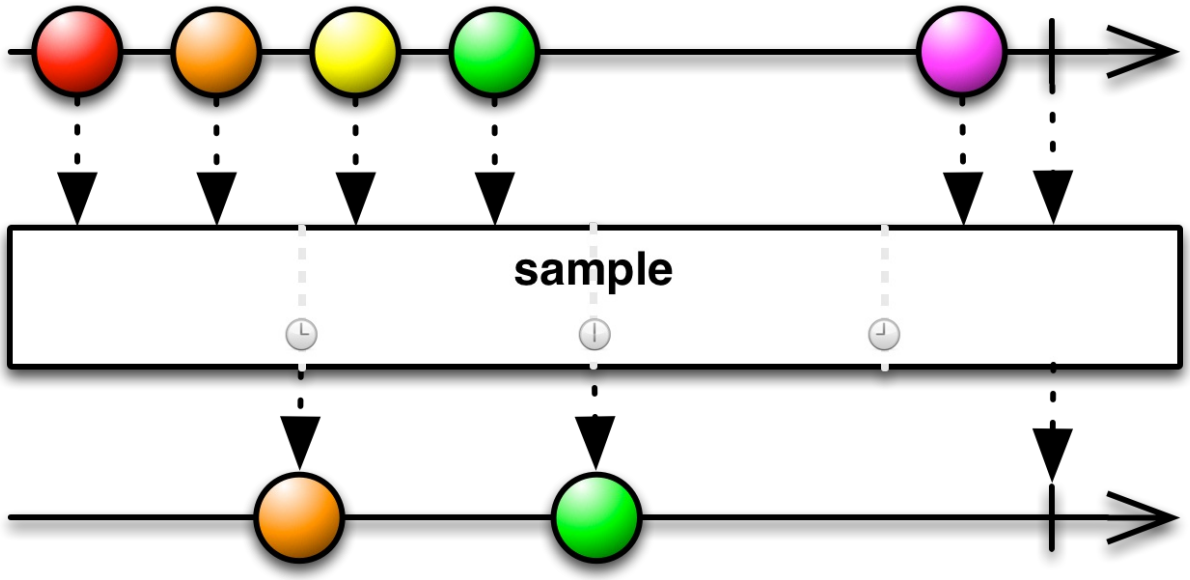
```
Observable.just(1,2,3,4,5,6).elementAt(3);
Observable.just(1,2,3,4,5,6).elementAtOrDefault(2,11);
```



요소 5번이 없더라도 예외가 발생하지않고 Default 값을 Observable에 값을 전달한다.

2.4 sample

일정 시간마다 스트림의 값을 취득하여 전달한다. 인수는 `sample(long,TimeUnit)` 이다.



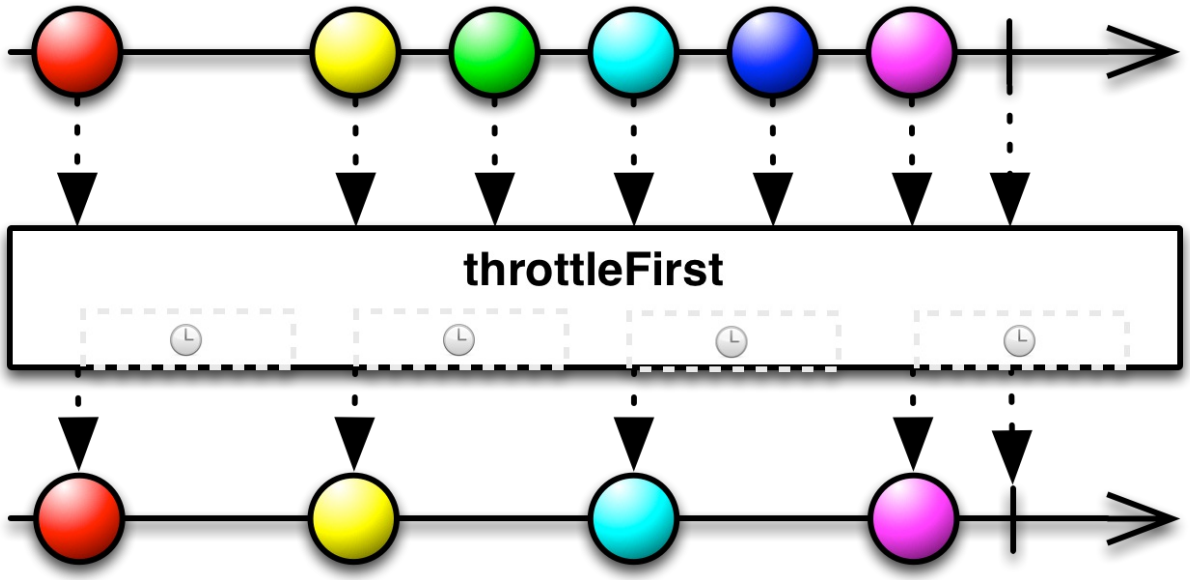
일정 시간이 흐르면 직전 스트림에서 흐르고있던 값을 subscribe한다.

```
Observable.interval(1,TimeUnit.DAYS)
          .sample(2,TimeUnit.DAYS)
```

매일마다 Observable에 데이터가 흘러들어오는 것을 이틀에 한번 onNext한다는 의미이다.

2.5 throttle First/Last

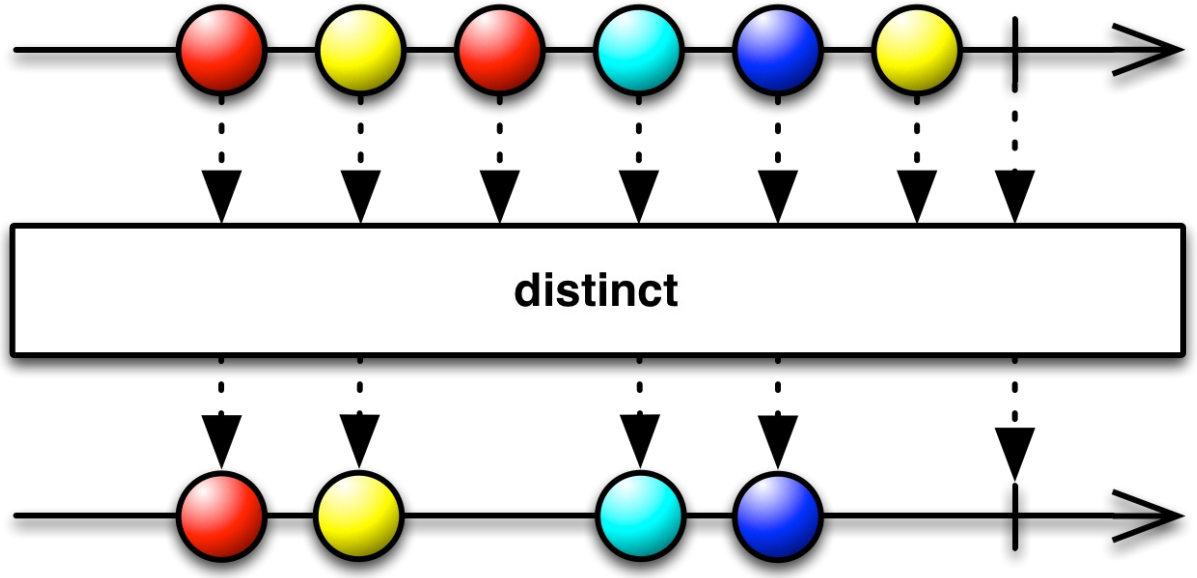
인수는 sample과 같은 `throttleFirst(long,TimeUnit)` 이다. 일정시간, onNext는 하지 않는 경우에 사용한다.




```
Observable.interval(1,TimeUnit.DAYS)
          .throttleFirst(3,TimeUnit.DAYS)
```

2.6 distinct

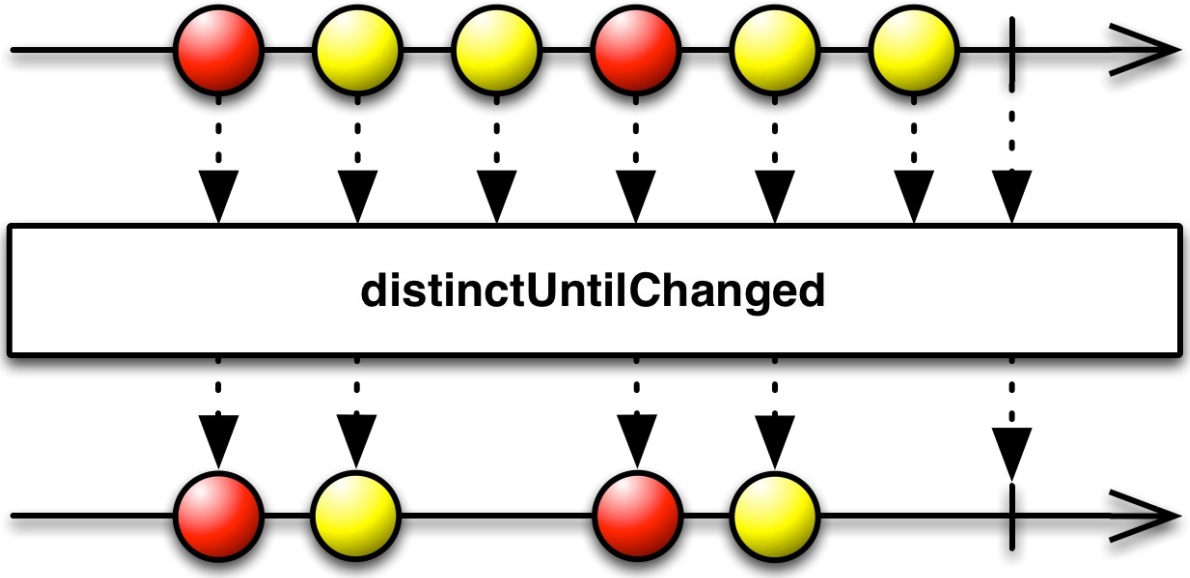
중복은 제거하고 onNext로 흘려보낸다.




```
Observable.just(1,2,2,2,1,3,4,1)
    .distinct()
    .subscribe(
        new Observer<Integer>() {
            @Override
            public void onCompleted() {
                Log.d("onCompleted", "owari");
            }

            @Override
            public void onError(Throwable e) {
                e.printStackTrace();
            }

            @Override
            public void onNext(Integer integer) {
                Log.d("onNext", String.valueOf(integer));
            }
        }
    );
```

2.7 distinctUntilChanged

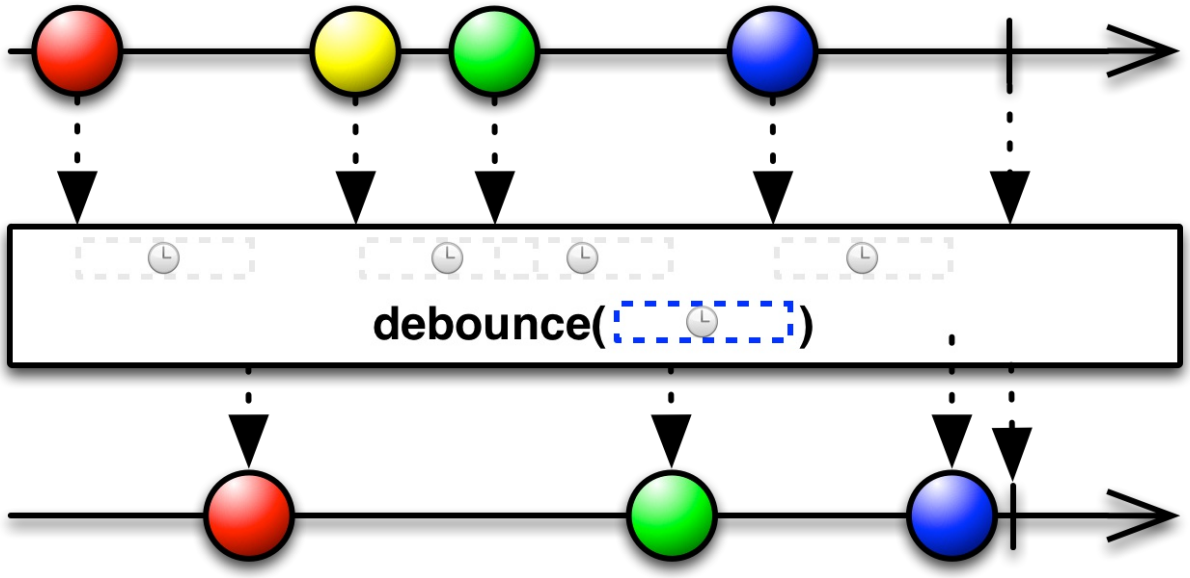
중복이있는 경우에는 하나로 묶는다(여기까지는 Distinct와 동일). 그러나, 다른 값의 입력이 있는 경우에는 별도로 한다.




```
Observable.just(1,2,2,2,1,2,2,3,3,1)
          .distinctUntilChanged()
```

```
[결과]
10-07 21:10:14.459 23959-23959/com.example.yukin.rxjava1st D/onNext: 1
10-07 21:10:14.459 23959-23959/com.example.yukin.rxjava1st D/onNext: 2
10-07 21:10:14.459 23959-23959/com.example.yukin.rxjava1st D/onNext: 1
10-07 21:10:14.459 23959-23959/com.example.yukin.rxjava1st D/onNext: 2
10-07 21:10:14.459 23959-23959/com.example.yukin.rxjava1st D/onNext: 3
10-07 21:10:14.459 23959-23959/com.example.yukin.rxjava1st D/onNext: 1
```

2.8 debounce



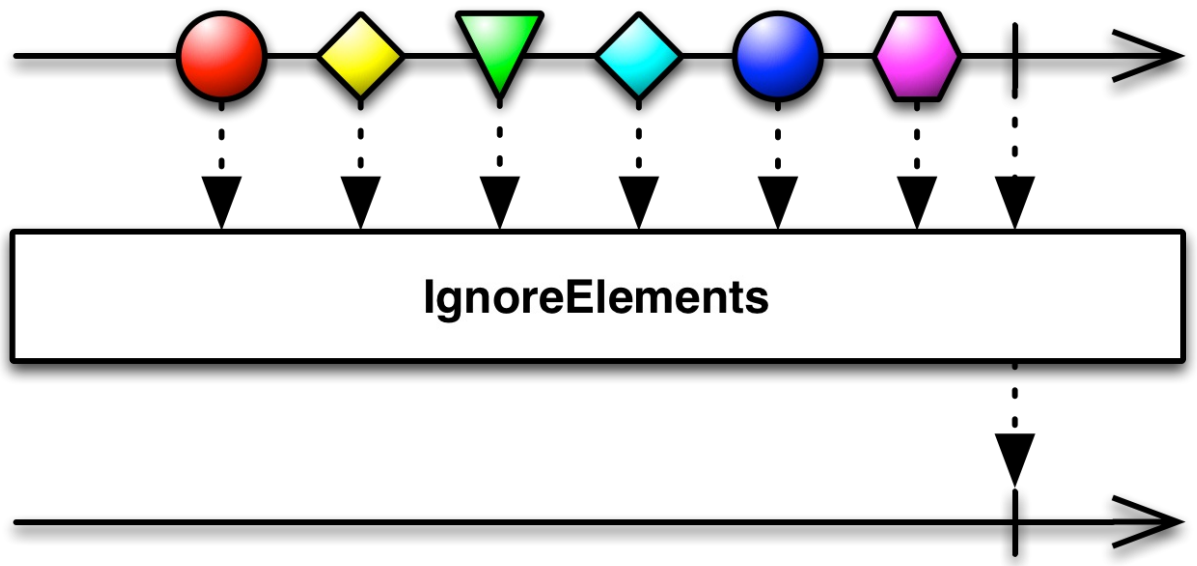
지정한 시간에 데이터가 흘러 들어오지 않으면 직전 데이터에서 이벤트가 발화하는 경우이다.

```
RxView.clicks(button).debounce(2,TimeUnit.SECONDS)
      .subscribe(new Observer<Void>() {
          @Override
          public void onCompleted() {
              Log.d("owari","owari");
          }

          @Override
          public void onError(Throwable e) {

          }

          @Override
          public void onNext(Void aVoid) {
              textView.setText("owari");
          }
      });
```

RxBinding에서 Button의 클릭을 검치한다. 클릭을 할때마다 Observalble이 발행된다. 그러나, debounce에서 2초로 지정되어 있기때문에 2초 이내에 다시 한번 버튼을 클릭하며는 경우에는 onNext하지 않는다.

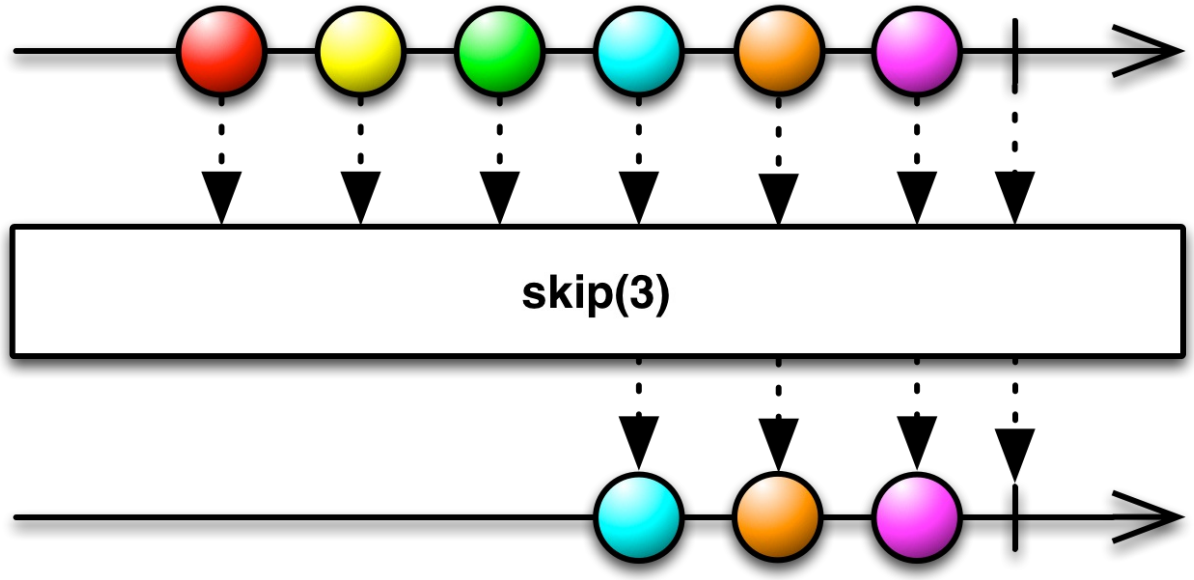
2.9 ignoreElements

어떤 Observable이라도 통과하지 않는다.



onNext도 호출되지 않기때문에, onError, onCompleted만 호출가능하다.

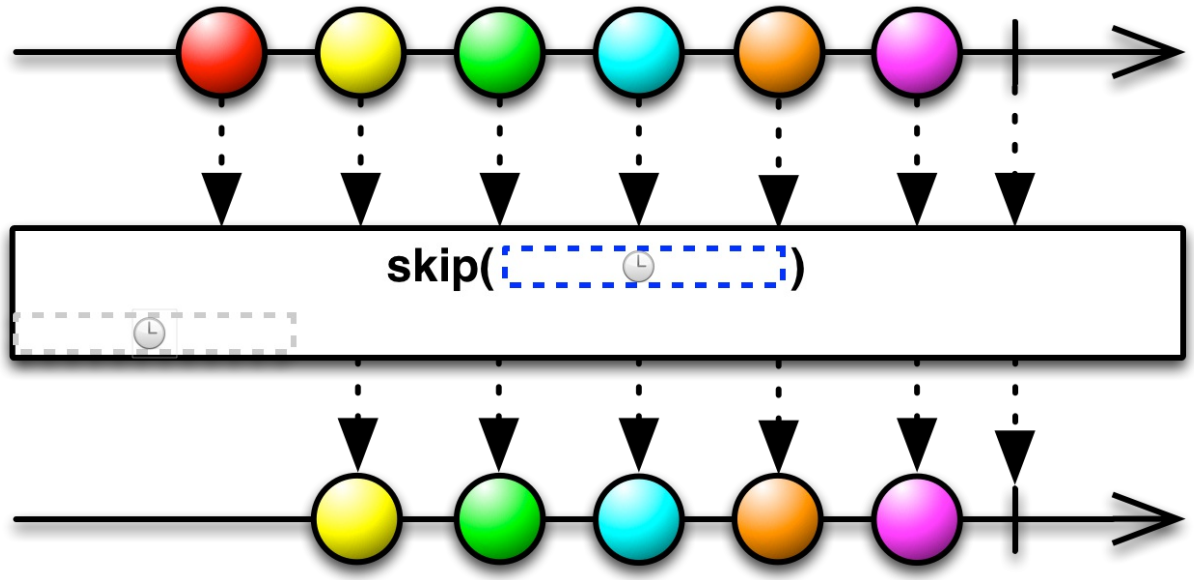
2.10 skip




```
Observable.range(1,10)
          .skip(3)
```

인수로 지정한 수만큼 onNext를 하지않느다. 또, skipLast는 마지막 3개를 스킵한다.

```
Observable.range(1,10)
          .skipLast(3)
```



인수로 시간을 지정한다. 인수로 넘겨진 시간동안에는 onNext하지 않는다.

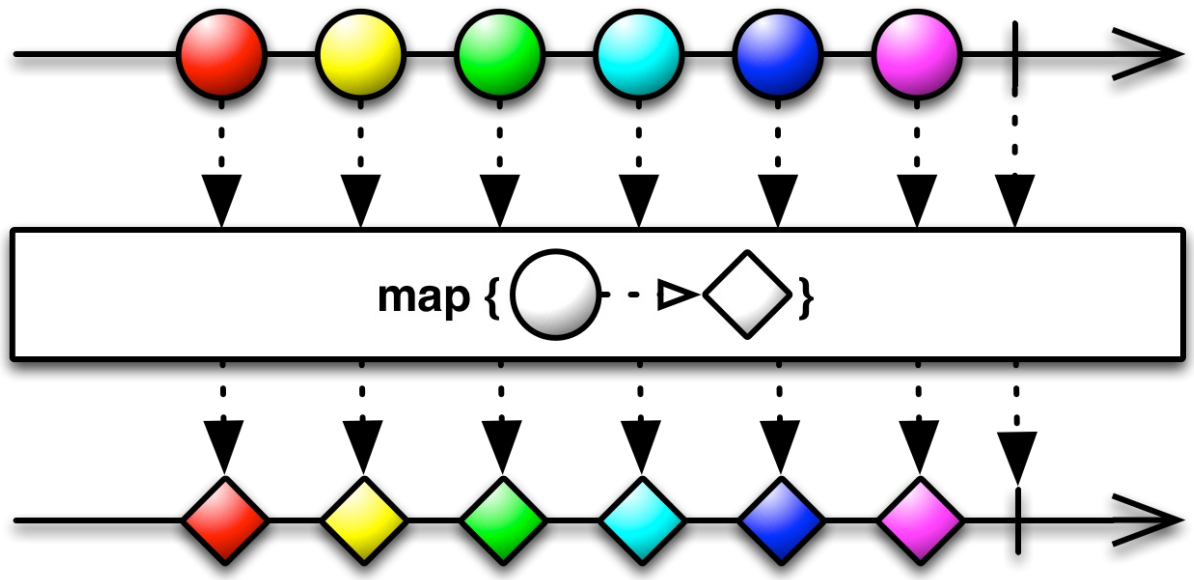
```
//10일간 onNext하지 않는다.
Observable.range(1,11)
          .skip(10,TimeUnit.DAYS)
```

3. 변환

Observable에 흐르고 있는 데이터를 변환해서 subscribe하고는 하는데, 그때 도움이 되는 operator를 소개한다.

3.1 map

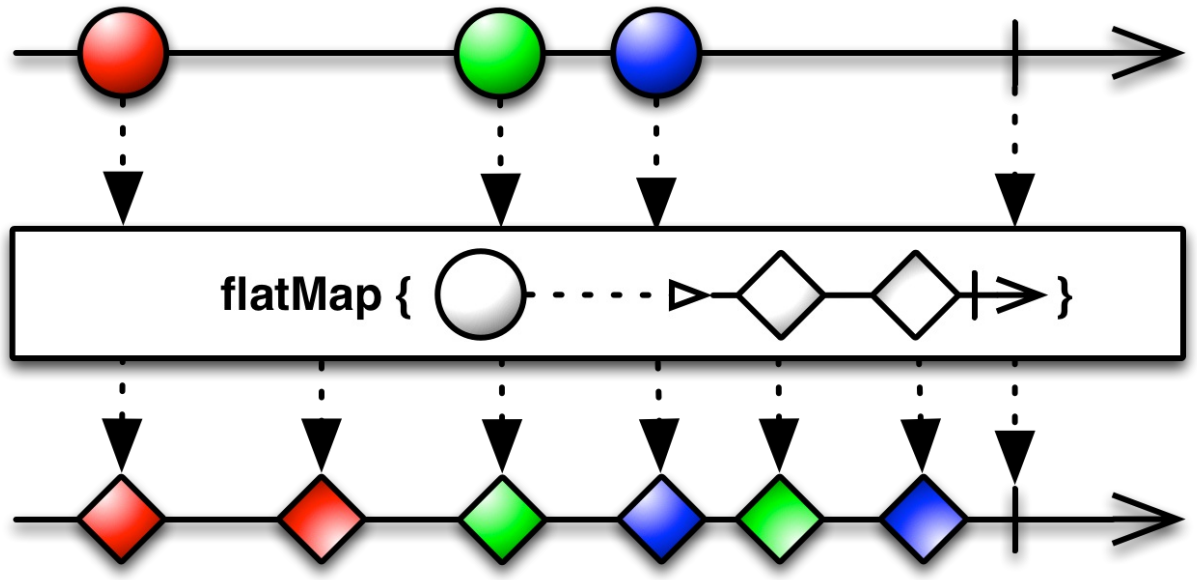
흘러온 데이터를 성형한다.




```
Observable.range(1,10)
          .map(new Func1<Integer, String>() {
              @Override
              public String call(Integer integer) {
                  return integer.toString();
              }
          })
```

3.2 flatMap

하나의 아이템으로부터 복수의 아이템으로 한다던지 하는 경우에 사용한다.



그림에서는 flatMap의 조건이 ○를 ◇ 2개로 분리하고 있다. 그렇기때문에 onNext되는 데이터는「빨간○ 1 개」로부터 「파란◇ 2 개」가 생성된다. flatMap에서는 Observable을 생성하고 있는 것이 핵심이다.

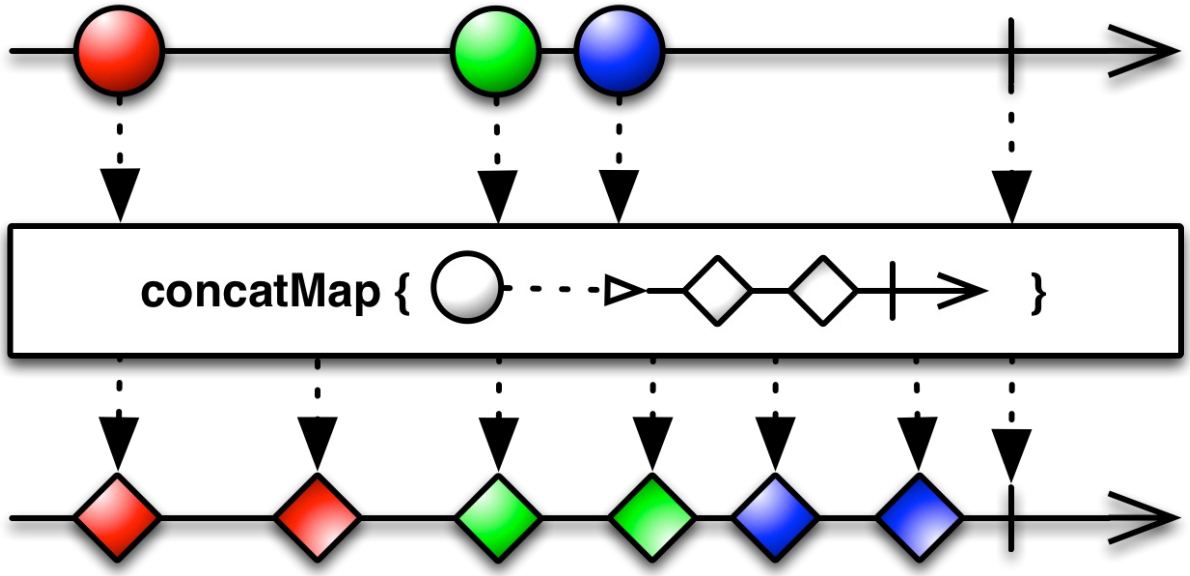
```
String[] place = {"静岡県磐田市", "神奈川県川崎市",
 "千葉県柏市", "石川県金沢市", "宮城県仙台市", "茨城県鹿嶋市"};

Observable.from(place)
          .flatMap(new Func1<String, Observable<String>>() {
                @Override
                public Observable<String> call(String s) {
                    return Observable.from(s.split("県"));
                }
          })
          .filter(new Func1<String, Boolean>() {
                @Override
                public Boolean call(String s) {
                    return s.contains("市");
                }
          })
          .subscribe(new Observer<String>() {
                @Override
                public void onCompleted() {
                }

                @Override
                public void onError(Throwable e) {
                }

                @Override
                public void onNext(String s) {
                    Log.d("city",s);
                }
```

```
        });
```

축구 팀이 있는 시를 현부터 적은 String 배열을 Observable로 한다. 현의 정보는 필요없기때문에 시정보만 취득한다. 우선은 원 데이터로부터 flatMap을 이용해서 '현'을 구분자로 잘라낸 새로운 Observable을 만든다. 요소 수는 이것으로 배로 증가한다. 현의 정보는 필요없기 때문에 "시"가 포함되지 않은 정보는 필터링해서 버린다.

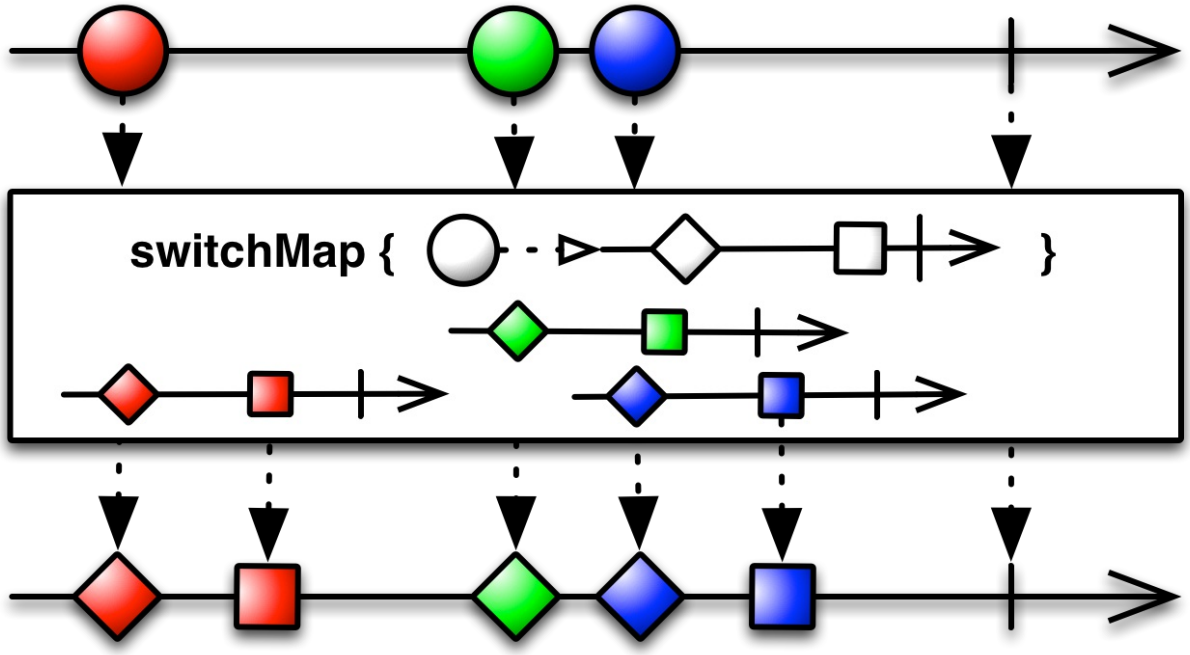
```
10-18 12:44:27.492 2776-2776/com.example.yukin.rxjava1st D/city: 磐田市
10-18 12:44:27.493 2776-2776/com.example.yukin.rxjava1st D/city: 川崎市
10-18 12:44:27.493 2776-2776/com.example.yukin.rxjava1st D/city: 柏市
10-18 12:44:27.493 2776-2776/com.example.yukin.rxjava1st D/city: 金沢市
10-18 12:44:27.493 2776-2776/com.example.yukin.rxjava1st D/city: 仙台市
10-18 12:44:27.493 2776-2776/com.example.yukin.rxjava1st D/city: 鹿嶋市
```

3.3 concatMap



동작방식은 flatMap과 같다. 차이점은 Observable로 흘러들어 왔을때 순서가 보장된다. 직렬조작이다. 반면 flatMap은 순서가 보장되지않는 병렬조작이다.

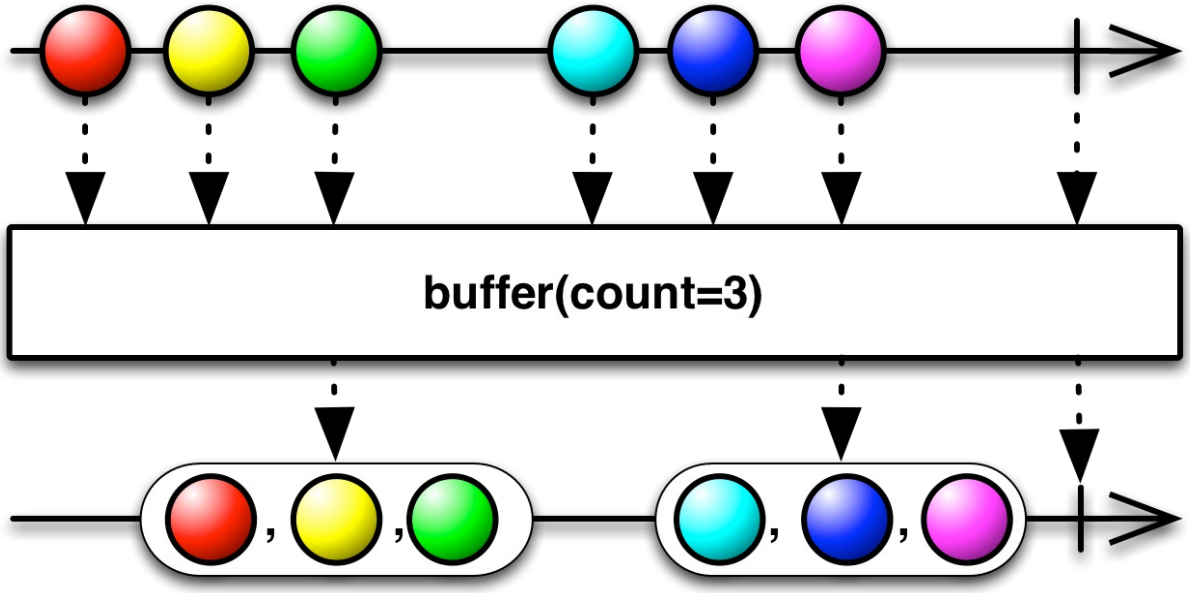
3.4 switchMap



병렬조작이다. 그러나, 인터럽트된 경우에는 더이상 Observable을 생성하지 않게 된다.

3.5 buffer

지정한 수로 Observable의 데이터를 묶어서 리스트로 onNext한다.

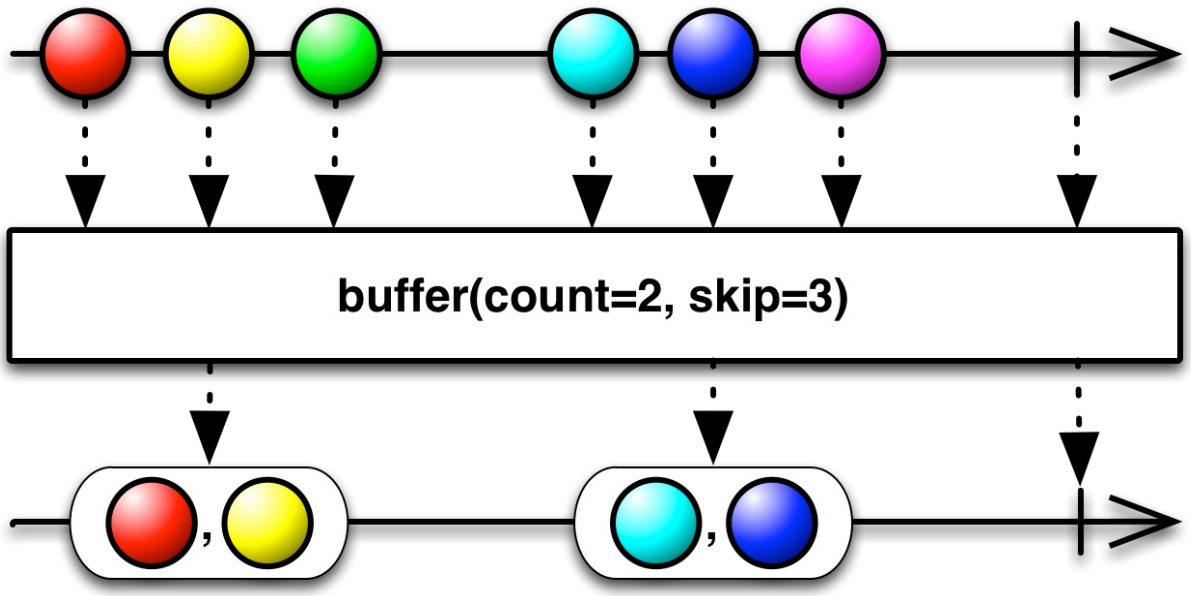



```
Observable.range(1,16)
          .buffer(3)
          .subscribe(new Observer<List<Integer>>() {
              @Override
              public void onCompleted() {
              }

              @Override
```

```
            public void onError(Throwable e) {
                e.printStackTrace();
            }

            @Override
            public void onNext(List<Integer> i) {
                Log.d("next", i.get(i.size()-1).toString());
            }
        });
```

```
10-18 13:24:04.115 6606-6606/com.example.yukin.rxjava1st D/next: 3
10-18 13:24:04.115 6606-6606/com.example.yukin.rxjava1st D/next: 6
10-18 13:24:04.115 6606-6606/com.example.yukin.rxjava1st D/next: 9
10-18 13:24:04.115 6606-6606/com.example.yukin.rxjava1st D/next: 12
10-18 13:24:04.115 6606-6606/com.example.yukin.rxjava1st D/next: 15
10-18 13:24:04.115 6606-6606/com.example.yukin.rxjava1st D/next: 16
```

이 operator는 skip도 있다.

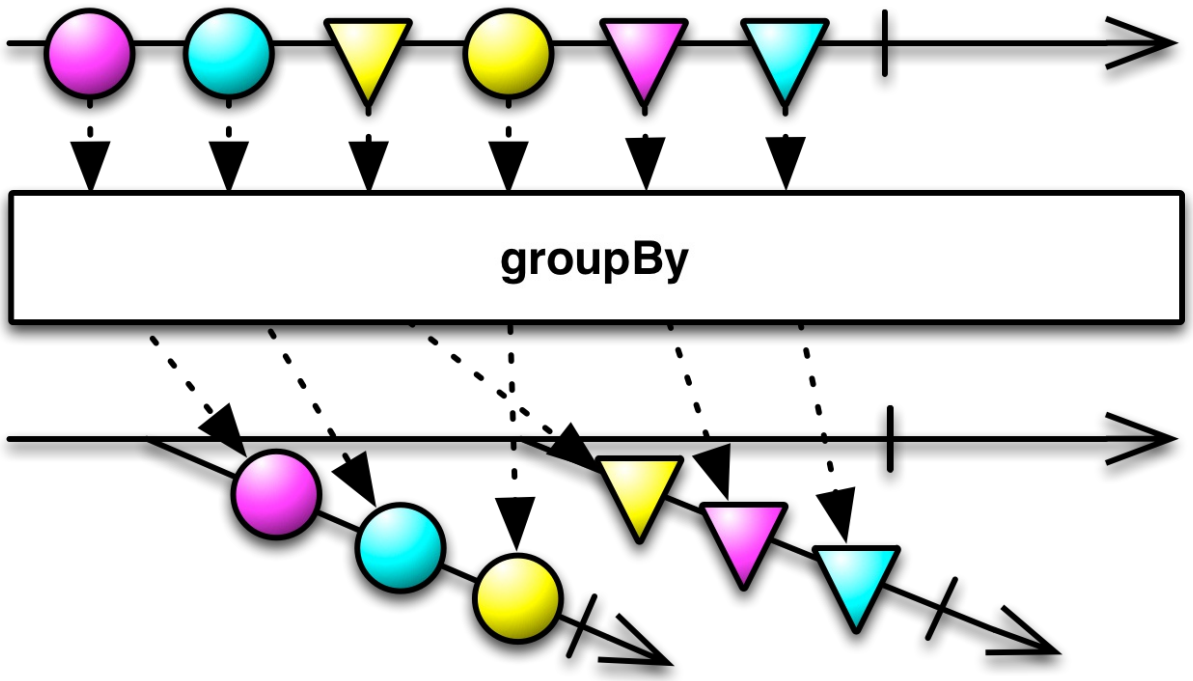



```
10-18 13:38:09.829 6606-6606/com.example.yukin.rxjava1st D/next: 1
10-18 13:38:09.829 6606-6606/com.example.yukin.rxjava1st D/next: 2
10-18 13:38:09.829 6606-6606/com.example.yukin.rxjava1st D/next: 4
10-18 13:38:09.829 6606-6606/com.example.yukin.rxjava1st D/next: 5
10-18 13:38:09.829 6606-6606/com.example.yukin.rxjava1st D/next: 7
10-18 13:38:09.830 6606-6606/com.example.yukin.rxjava1st D/next: 8
10-18 13:38:09.830 6606-6606/com.example.yukin.rxjava1st D/next: 10
10-18 13:38:09.830 6606-6606/com.example.yukin.rxjava1st D/next: 11
10-18 13:38:09.831 6606-6606/com.example.yukin.rxjava1st D/next: 13
10-18 13:38:09.831 6606-6606/com.example.yukin.rxjava1st D/next: 14
10-18 13:38:09.831 6606-6606/com.example.yukin.rxjava1st D/next: 16
```

1 -> 2 -> 3(skip) -> 4 -> 5 -> 6(skip) ...

4. groupBy

조건식을 넘기고 그것에 따라 Observavle을 나눈다.




```
Observable.range(1, 16)
    .groupBy(new Func1<Integer, Integer>() {
        @Override
            public Integer call(Integer integer) {
            return integer % 3;
        }
    })
    .subscribe(new Subscriber<GroupedObservable<Integer, Integer>>() {
        @Override
        public void onCompleted() {
        }

        @Override
        public void onError(Throwable e) {
            e.printStackTrace();
        }

        @Override
        public void onNext(
            final GroupedObservable<Integer, Integer> integerIntegerGroupedO
        integerIntegerGroupedObservable.toList().subscribe(
            new Subscriber<List<Integer>>() {
                @Override
                public void onCompleted() {
                }
```

```
                @Override
                public void onError(Throwable e) {
                    e.printStackTrace();
                }

                @Override
                public void onNext(List<Integer> integers) {
                    for (int item : integers)
                        Log.d(
                            String.valueOf(integerIntegerGroupedObservable.g
                            String.valueOf(item));
                }
            });
        }
    });
```

```
10-18 14:04:03.285 9241-9241/com.example.yukin.rxjava1st D/0: 3
10-18 14:04:03.285 9241-9241/com.example.yukin.rxjava1st D/0: 6
10-18 14:04:03.285 9241-9241/com.example.yukin.rxjava1st D/0: 9
10-18 14:04:03.285 9241-9241/com.example.yukin.rxjava1st D/0: 12
10-18 14:04:03.285 9241-9241/com.example.yukin.rxjava1st D/0: 15
10-18 14:04:03.285 9241-9241/com.example.yukin.rxjava1st D/1: 1
10-18 14:04:03.285 9241-9241/com.example.yukin.rxjava1st D/1: 4
10-18 14:04:03.285 9241-9241/com.example.yukin.rxjava1st D/1: 7
10-18 14:04:03.285 9241-9241/com.example.yukin.rxjava1st D/1: 10
10-18 14:04:03.285 9241-9241/com.example.yukin.rxjava1st D/1: 13
10-18 14:04:03.285 9241-9241/com.example.yukin.rxjava1st D/1: 16
10-18 14:04:03.285 9241-9241/com.example.yukin.rxjava1st D/2: 2
10-18 14:04:03.285 9241-9241/com.example.yukin.rxjava1st D/2: 5
10-18 14:04:03.285 9241-9241/com.example.yukin.rxjava1st D/2: 8
10-18 14:04:03.285 9241-9241/com.example.yukin.rxjava1st D/2: 11
10-18 14:04:03.286 9241-9241/com.example.yukin.rxjava1st D/2: 14
```