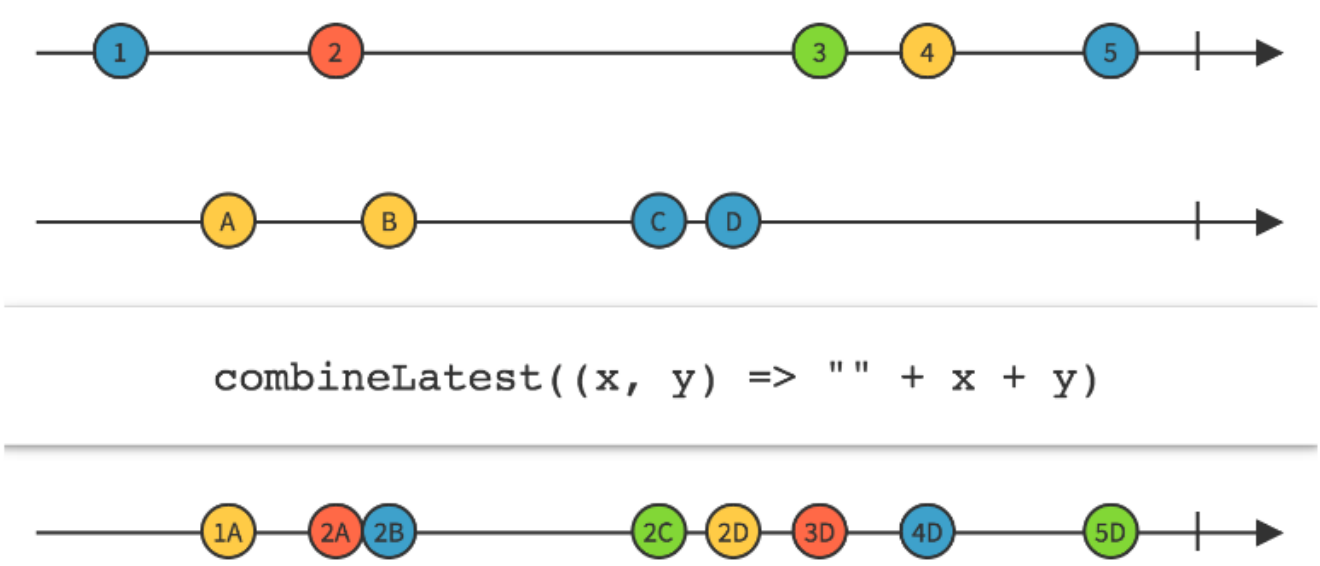
Rxjava_操作符3_组合操作符

目录
[TOC]

1.CombineLatest操作符

当两个Observables中的任何一个发射了数据时，使用一个函数结合每个Observable发射的最近数据项，并且基于这个函数的结果发射数据



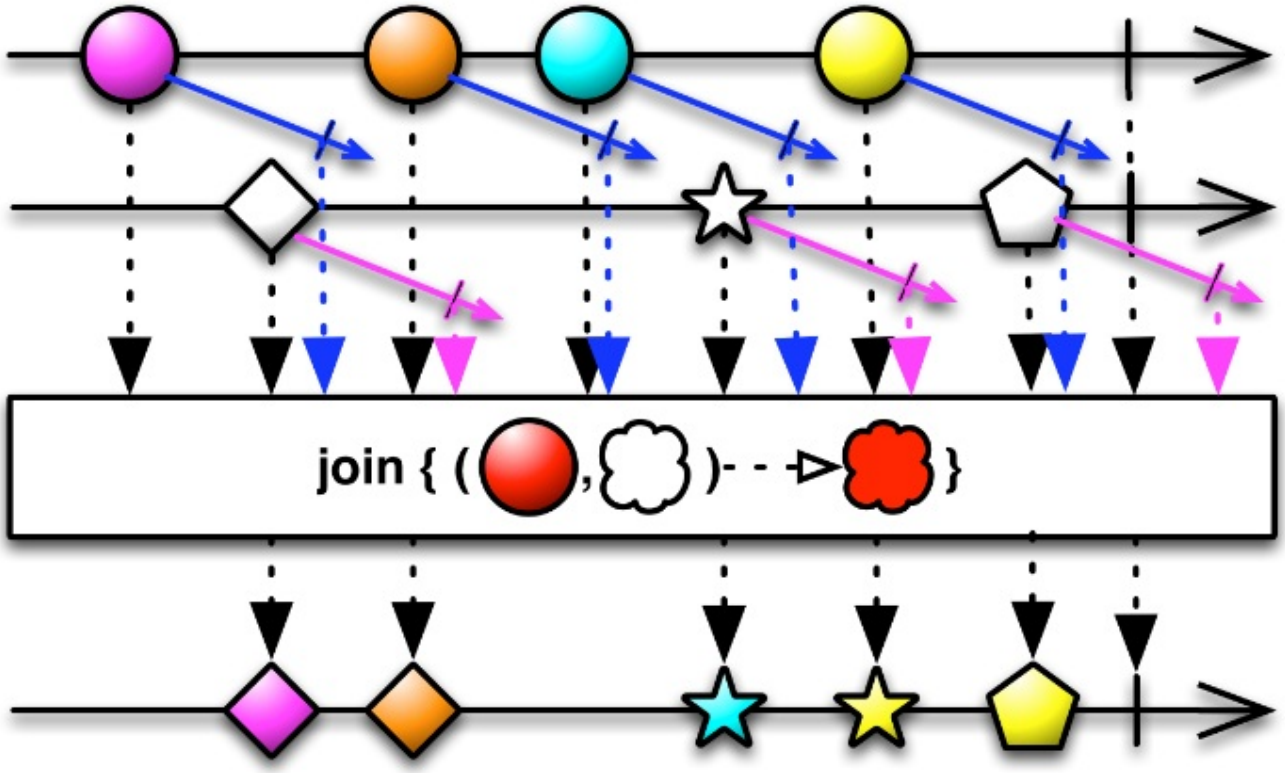

```
//产生0,5,10,15,20数列
        Observable<Long> observable1 = Observable.interval(0, 1000, TimeUnit.MI
LLISECONDS)
                .map(new Func1<Long, Long>() {
                    @Override
                    public Long call(Long aLong) {
                        return aLong * 5;
                    }
                }).take(5);

        //产生0,10,20,30,40数列
        Observable<Long> observable2 = Observable.interval(500, 1000, TimeUnit.
MILLISECONDS)
                .map(new Func1<Long, Long>() {
                    @Override
                    public Long call(Long aLong) {
                        return aLong * 10;
                    }
                }).take(5);
```

```
        Observable.combineLatest(observable1, observable2, new Func2<Long, Long
, Long>() {
            @Override
            public Long call(Long aLong, Long aLong2) {
                return aLong+aLong2;
            }
        }).subscribe(new Subscriber<Long>() {
            @Override
            public void onCompleted() {
                System.out.println("Sequence complete.");
            }

            @Override
            public void onError(Throwable e) {
                System.err.println("Error: " + e.getMessage());
            }

            @Override
            public void onNext(Long aLong) {
                System.out.println("Next: " + aLong);
            }
        });
```

```
运行结果如下：
Next: 0
Next: 5
Next: 15
Next: 20
Next: 30
Next: 35
Next: 45
Next: 50
Next: 60
Sequence complete.
```

2.join操作符

join操作符类似于combineLatest操作符。任何时候，只要在另一个Observable发射的数据定义的时间窗口内，这个Observable发射了一条数据，就结合两个Observable发射的数据。，流程图如下：



Join操作符结合两个Observable发射的数据，基于时间窗口（你定义的针对每条数据特定的原则）选择待集合的数据项。你将这些时间窗口实现为一些Observables，它们的生命周期从任何一条Observable发射的每一条数据开始。当这个定义时间窗口的Observable发射了一条数据或者完成时，与这条数据关联的窗口也会关闭。只要这条数据的窗口是打开的，它将继续结合其它Observable发射的任何数据项。你定义一个用于结合数据的函数。

`join(Observable, Func1, Func1, Func2)` 我们先介绍下join操作符的4个参数：

`Observable`：源Observable需要组合的Observable,这里我们姑且称之为目标Observable；

`Func1`：接收从源Observable发射来的数据，并返回一个Observable，这个Observable的声明周期决定了源Obsrvable发射出来的数据的有效期；

`Func1`：接收目标Observable发射来的数据，并返回一个Observable，这个Observable的声明周期决定了目标Obsrvable发射出来的数据的有效期；

`Func2`：接收从源Observable和目标Observable发射出来的数据，并将这两个数据组合后返回。

```
//产生0,5,10,15,20数列
        Observable<Long> observable1 = Observable.timer(0, 1000, TimeUnit.MILLI
SECONDS)
                .map(new Func1<Long, Long>() {
                    @Override
                    public Long call(Long aLong) {
                        return aLong * 5;
                    }
                }).take(5);
```

```
        //产生0,10,20,30,40数列
        Observable<Long> observable2 = Observable.timer(500, 1000, TimeUnit.MIL
LISECONDS)
                .map(new Func1<Long, Long>() {
                    @Override
                    public Long call(Long aLong) {
                        return aLong * 10;
                    }
                }).take(5);

        observable1.join(observable2, new Func1<Long, Observable<Long>>() {
            @Override
            public Observable<Long> call(Long aLong) {
                //使Observable延迟600毫秒执行
                return Observable.just(aLong).delay(600, TimeUnit.MILLISECONDS)
;
            }
        }, new Func1<Long, Observable<Long>>() {
            @Override
            public Observable<Long> call(Long aLong) {
                //使Observable延迟600毫秒执行
                return Observable.just(aLong).delay(600, TimeUnit.MILLISECONDS)
;
            }
        }, new Func2<Long, Long, Long>() {
            @Override
            public Long call(Long aLong, Long aLong2) {
                return aLong + aLong2;
            }
        }).subscribe(new Subscriber<Long>() {
            @Override
            public void onCompleted() {
                System.out.println("Sequence complete.");
            }

            @Override
            public void onError(Throwable e) {
                System.err.println("Error: " + e.getMessage());
            }

            @Override
            public void onNext(Long aLong) {
                System.out.println("Next: " + aLong);
            }
        });
```

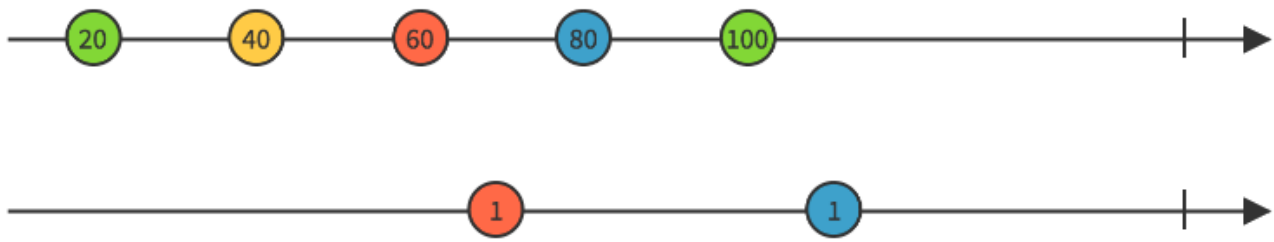
运行结果如下：

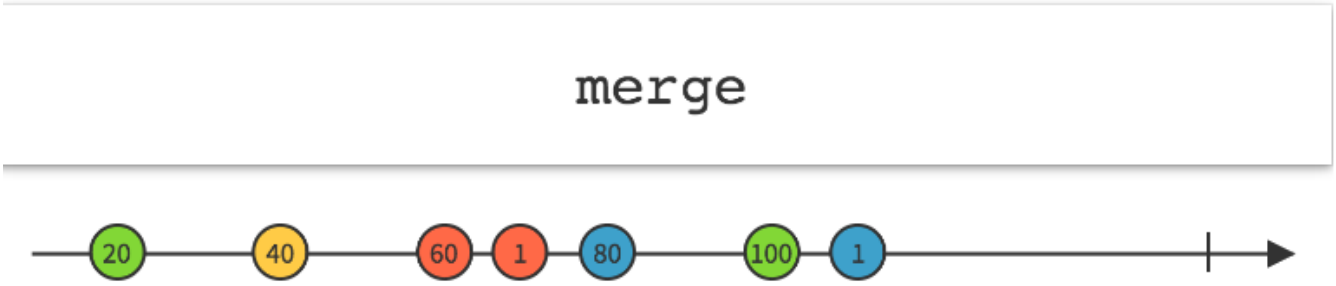
```
Next: 0
```

```
Next: 5
Next: 15
Next: 20
Next: 30
Next: 35
Next: 45
Next: 50
Next: 60
Sequence complete.
```

3.merge操作符

合并多个Observables的发射物





使用Merge操作符你可以将多个Observables的输出合并，就好像它们是一个单个的Observable一样。

Merge可能会让合并的Observables发射的数据交错（有一个类似的操作符Concat不会让数据交错，它会按顺序一个接着一个发射多个Observables的发射物）。

正如图例上展示的，任何一个原始Observable的onError通知会被立即传递给观察者，而且会终止合并后的Observable。

```
//产生0,5,10,15,20数列
        Observable<Long> observable1 = Observable.timer(0, 1000, TimeUnit.MILLI
SECONDS)
                .map(new Func1<Long, Long>() {
                    @Override
                    public Long call(Long aLong) {
                        return aLong * 5;
                    }
                }).take(5);
```

```
        //产生0,10,20,30,40数列
        Observable<Long> observable2 = Observable.timer(500, 1000, TimeUnit.MIL
LISECONDS)
                .map(new Func1<Long, Long>() {
                    @Override
                    public Long call(Long aLong) {
                        return aLong * 10;
                    }
                }).take(5);

        Observable.merge(observable1, observable2)
                .subscribe(new Subscriber<Long>() {
                    @Override
                    public void onCompleted() {
                        System.out.println("Sequence complete.");
                    }

                    @Override
                    public void onError(Throwable e) {
                        System.err.println("Error: " + e.getMessage());
                    }

                    @Override
                    public void onNext(Long aLong) {
                        System.out.println("Next:" + aLong);
                    }
                });
```

```
运行结果如下：
Next:0
Next:0
Next:5
Next:10
Next:10
Next:20
Next:15
Next:30
Next:20
Next:40
Sequence complete.
```

栗子之二

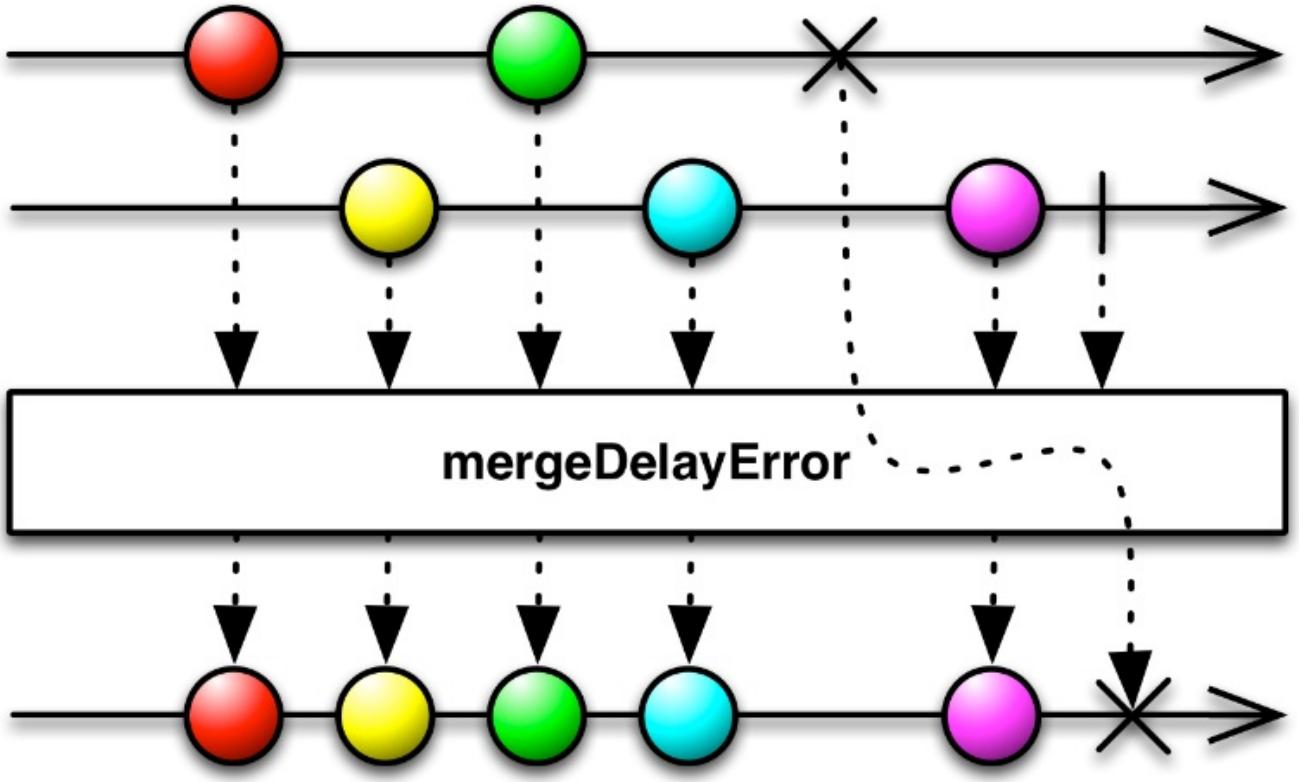
```
//作者：张磊(BaronZhang)
//链接：https://zhuanlan.zhihu.com/p/22039934

String[] letters = new String[]{"A", "B", "C", "D", "E", "F", "G", "H"};
Observable<String> letterSequence = Observable.interval(300, TimeUnit.MILLISECO
```

```
NDS)
        .map(new Func1<Long, String>() {
            @Override
            public String call(Long position) {
                return letters[position.intValue()];
            }
        }).take(letters.length);

Observable<Long> numberSequence = Observable.interval(500, TimeUnit.MILLISECOND
S).take(5);

Observable.merge(letterSequence, numberSequence)
        .subscribe(new Observer<Serializable>() {
            @Override
            public void onCompleted() {
                System.exit(0);
            }

            @Override
            public void onError(Throwable e) {
                System.out.println("Error:" + e.getMessage());
            }

            @Override
            public void onNext(Serializable serializable) {
                System.out.print(serializable.toString()+" ");
            }
        });
```

```
A 0 B C 1 D E 2 F 3 G H 4
```

mergeDelayError操作符

从merge操作符的流程图可以看出，一旦合并的某一个Observable中出现错误，就会马上停止合并，并对订阅者回调执行onError方法，而mergeDelayError操作符会把错误放到所有结果都合并完成之后才执行，其流程图如下：



Javadoc: merge(Iterable))
Javadoc: merge(Iterable,int))
Javadoc: merge(Observable[]))
Javadoc: merge(Observable,Observable)) (接受二到九个Observable)

```
//产生0,5,10数列,最后会产生一个错误
        Observable<Long> errorObservable = Observable.error(new Exception("this
is end!"));
        Observable < Long > observable1 = Observable.timer(0, 1000, TimeUnit.MI
LLISECONDS)
                .map(new Func1<Long, Long>() {
                    @Override
                    public Long call(Long aLong) {
                        return aLong * 5;
                    }
                }).take(3).mergeWith(errorObservable.delay(3500, TimeUnit.MILLI
SECONDS));

        //产生0,10,20,30,40数列
        Observable<Long> observable2 = Observable.timer(500, 1000, TimeUnit.MIL
LISECONDS)
                .map(new Func1<Long, Long>() {
                    @Override
                    public Long call(Long aLong) {
                        return aLong * 10;
                    }
                }).take(5);

        Observable.mergeDelayError(observable1, observable2)
```

```
                .subscribe(new Subscriber<Long>() {
                    @Override
                    public void onCompleted() {
                        System.out.println("Sequence complete.");
                    }

                    @Override
                    public void onError(Throwable e) {
                        System.err.println("Error: " + e.getMessage());
                    }

                    @Override
                    public void onNext(Long aLong) {
                        System.out.println("Next:" + aLong);
                    }
                });
```

```
运行结果如下：
Next:0
Next:0
Next:5
Next:10
Next:10
Next:20
Next:30
Next:40
Error: this is end!
```

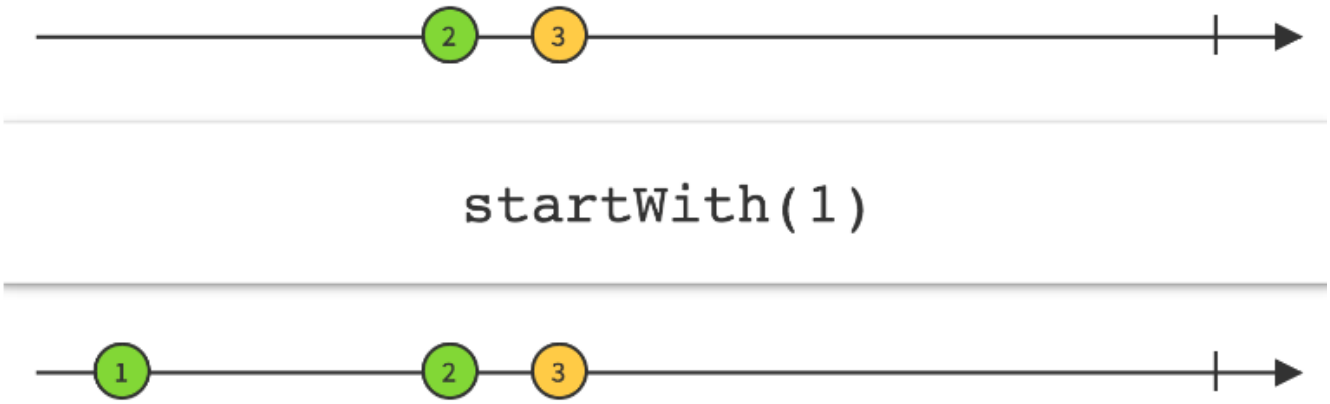
mergeWith

merge是静态方法，mergeWith是对象方法，举个例子，Observable.merge(odds,evens)等价于odds.mergeWith(evens)。

4.startWith操作符

在数据序列的开头插入一条指定的项

如果你想要一个Observable在发射数据之前先发射一个指定的数据序列，可以使用StartWith操作符。（如果你想一个Observable发射的数据末尾追加一个数据序列可以使用Concat操作符。）



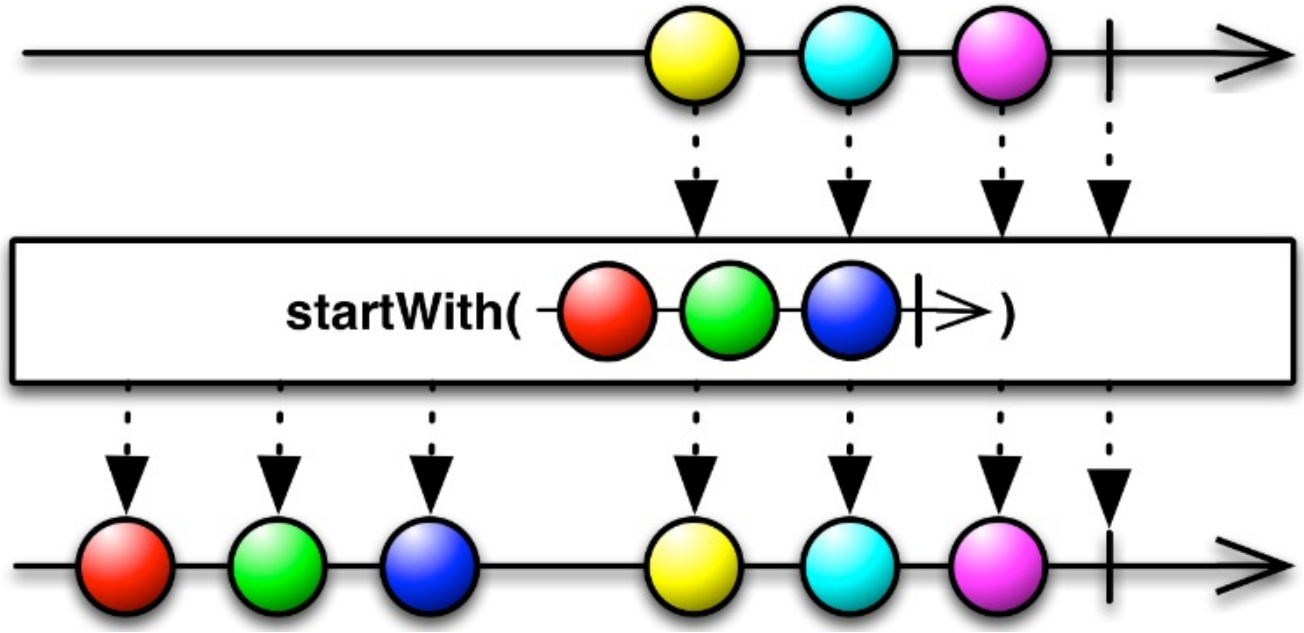
可接受一个Iterable或者多个Observable作为函数的参数。

Javadoc: startWith(Iterable))
Javadoc: startWith(T)) (最多接受九个参数)

```
Observable.just(10,20,30).startWith(2, 3, 4).subscribe(new Subscriber<Integer>(
) {
            @Override
            public void onCompleted() {
                System.out.println("Sequence complete.");
            }

            @Override
            public void onError(Throwable e) {
                System.err.println("Error: " + e.getMessage());
            }

            @Override
            public void onNext(Integer value) {
                System.out.println("Next:" + value);
            }
        });
```

```
运行结果如下：
Next:2
Next:3
Next:4
Next:10
Next:20
Next:30
Sequence complete.
```

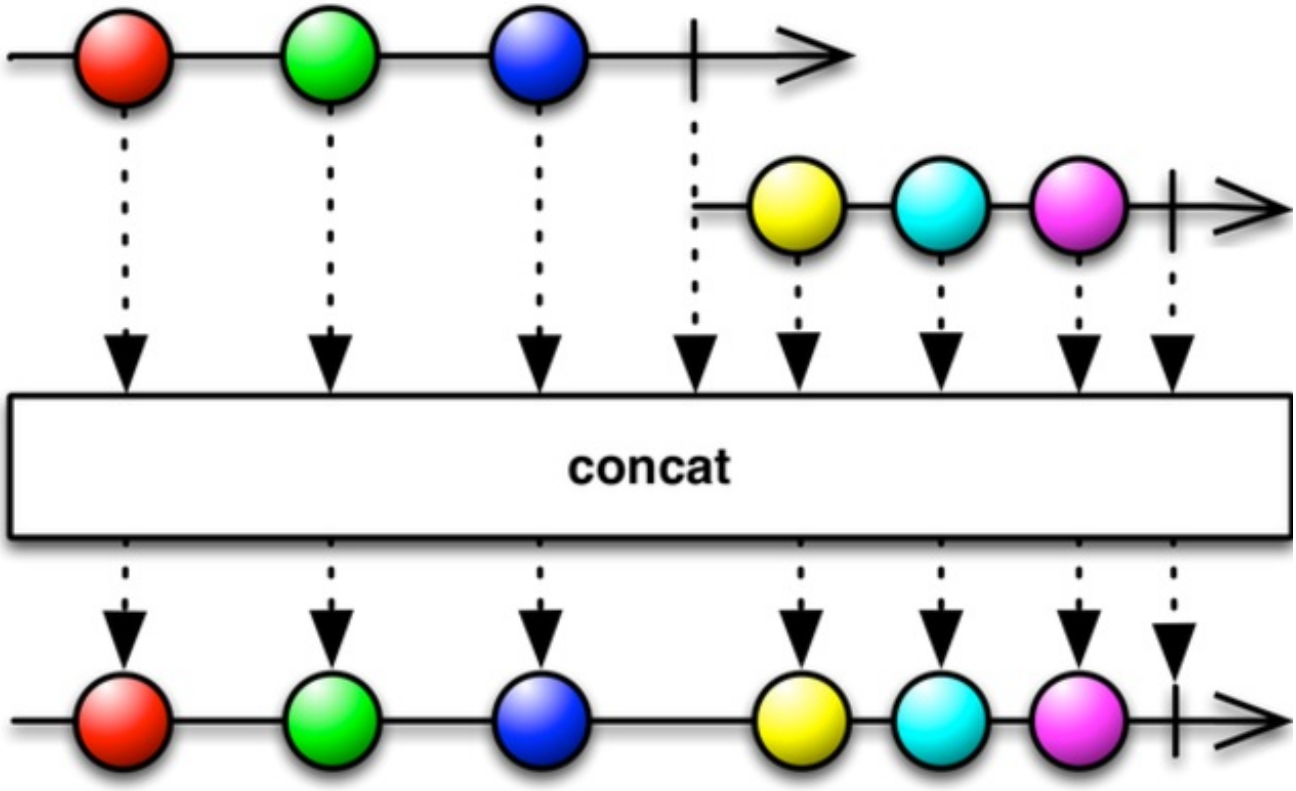
你也可以传递一个Observable给startWith，它会将那个Observable的发射物插在原始Observable发射的数据序列之前，然后把这个当做自己的发射物集合。这可以看作是Concat的反转。



5.Concat操作符

用于将多个obserbavle发射的的数据进行合并发射，concat严格按照顺序发射数据，前一个Observable没发射玩是不会发射后一个Observable的数据的。它和merge、startWitch和相似，不同之处在于：

merge:合并后发射的数据是无序的；
startWitch:只能在源Observable发射的数据前插入数据。




```
//这里我们将前面Merge操作符的例子拿过来，并将操作符换成Concat，然后我们看看执行结果：

    String[] letters = new String[]{"A", "B", "C", "D", "E", "F", "G", "H"};
```

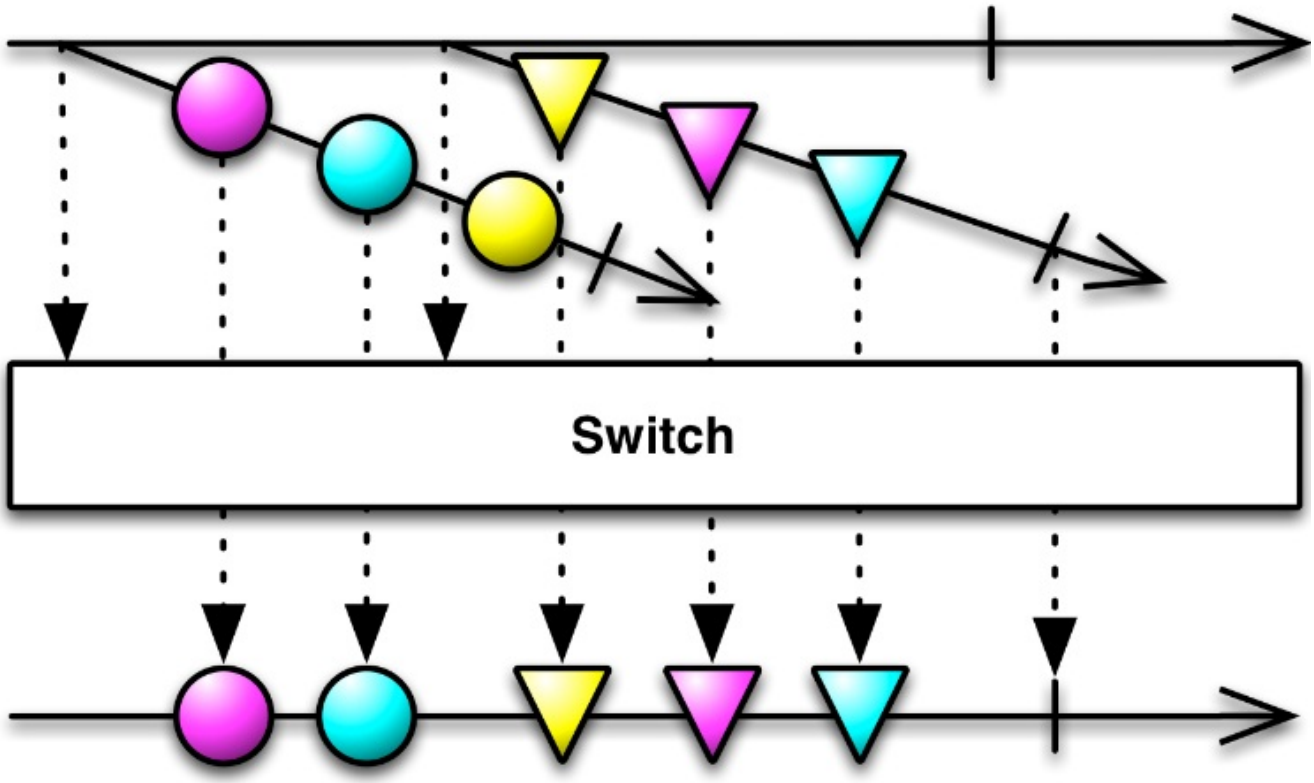
```
    Observable<String> letterSequence = Observable.interval(300, TimeUnit.MILLI
SECONDS)
            .map(new Func1<Long, String>() {
                @Override
                public String call(Long position) {
                    return letters[position.intValue()];
                }
            }).take(letters.length);

    Observable<Long> numberSequence = Observable.interval(500, TimeUnit.MILLISE
CONDS).take(5);

    Observable.concat(letterSequence, numberSequence)
            .subscribe(new Observer<Serializable>() {
                @Override
                public void onCompleted() {
                    System.exit(0);
                }

                @Override
                public void onError(Throwable e) {
                    System.out.println("Error:" + e.getMessage());
                }

                @Override
                public void onNext(Serializable serializable) {
                    System.out.print(serializable.toString() + " ");
                }
            });
```

```
    A B C D E F G H 0 1 2 3 4
```

6.switchOnNext操作符

将一个发射多个Observables的Observable转换成另一个单独的Observable，后者发射那些Observables最近发射的数据项

switchOnNext操作符是把一组Observable转换成一个Observable，转换规则为：对于这组Observable中的每一个Observable所产生的结果，如果在同一个时间内存在两个或多个Observable提交的结果，只取最后一个Observable提交的结果给订阅者



当原始Observable发射了一个新的Observable时（不是这个新的Observable发射了一条数据时），它将取消订阅之前的那个Observable。这意味着，在后来那个Observable产生之后到它开始发射数据之前的这段时间里，前一个Observable发射的数据将被丢弃（就像图例上的那个黄色圆圈一样）。

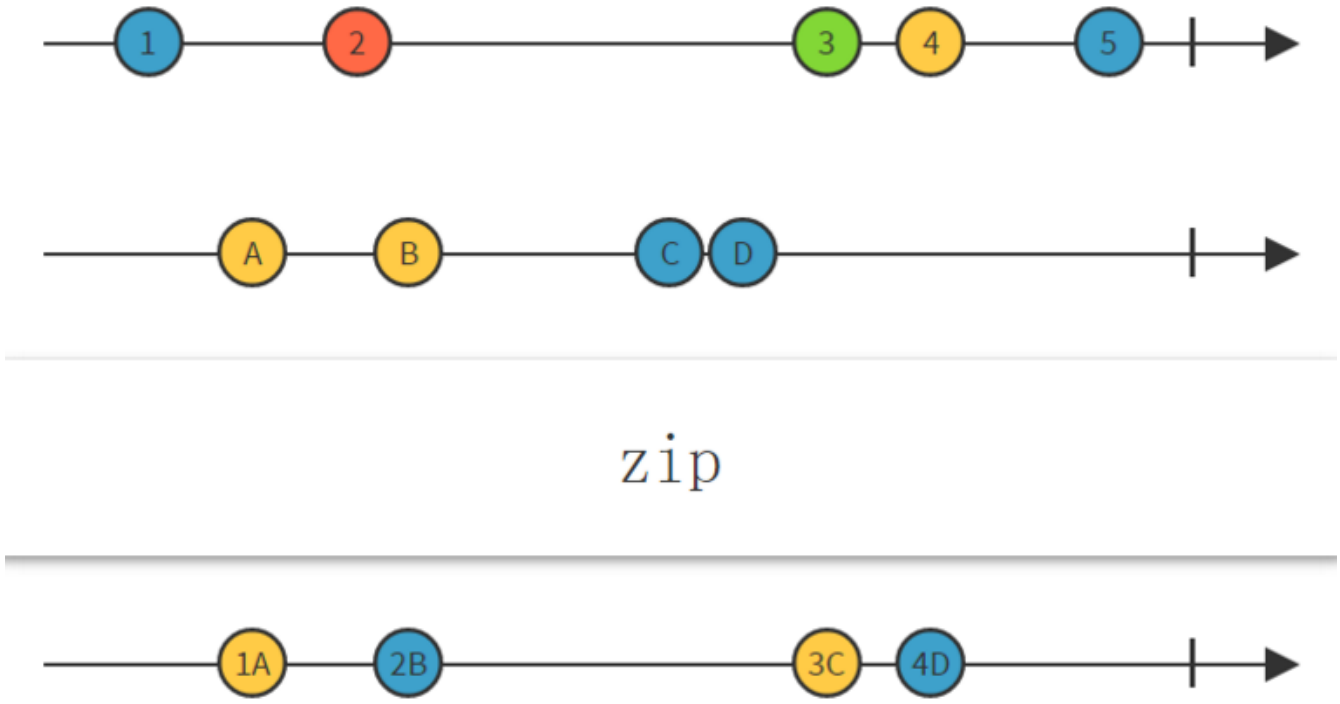
```
//每隔500毫秒产生一个observable
        Observable<Observable<Long>> observable = Observable.timer(0, 500, Time
Unit.MILLISECONDS).map(new Func1<Long, Observable<Long>>() {
            @Override
            public Observable<Long> call(Long aLong) {
                //每隔200毫秒产生一组数据（0,10,20,30,40)
                return Observable.timer(0, 200, TimeUnit.MILLISECONDS).map(new
Func1<Long, Long>() {
                    @Override
                    public Long call(Long aLong) {
                        return aLong * 10;
                    }
                }).take(5);
            }
        }).take(2);

        Observable.switchOnNext(observable).subscribe(new Subscriber<Long>() {
            @Override
            public void onCompleted() {
                System.out.println("Sequence complete.");
            }

            @Override
            public void onError(Throwable e) {
```

```
                System.err.println("Error: " + e.getMessage());
            }

            @Override
            public void onNext(Long aLong) {
                System.out.println("Next:" + aLong);
            }
        });
```

```
运行结果如下：
Next:0
Next:10
Next:20
Next:0
Next:10
Next:20
Next:30
Next:40
Sequence complete.
```

7.zip操作符

通过一个函数将多个Observables的发射物结合到一起，基于这个函数的结果为每个结合体发射单个数据项。



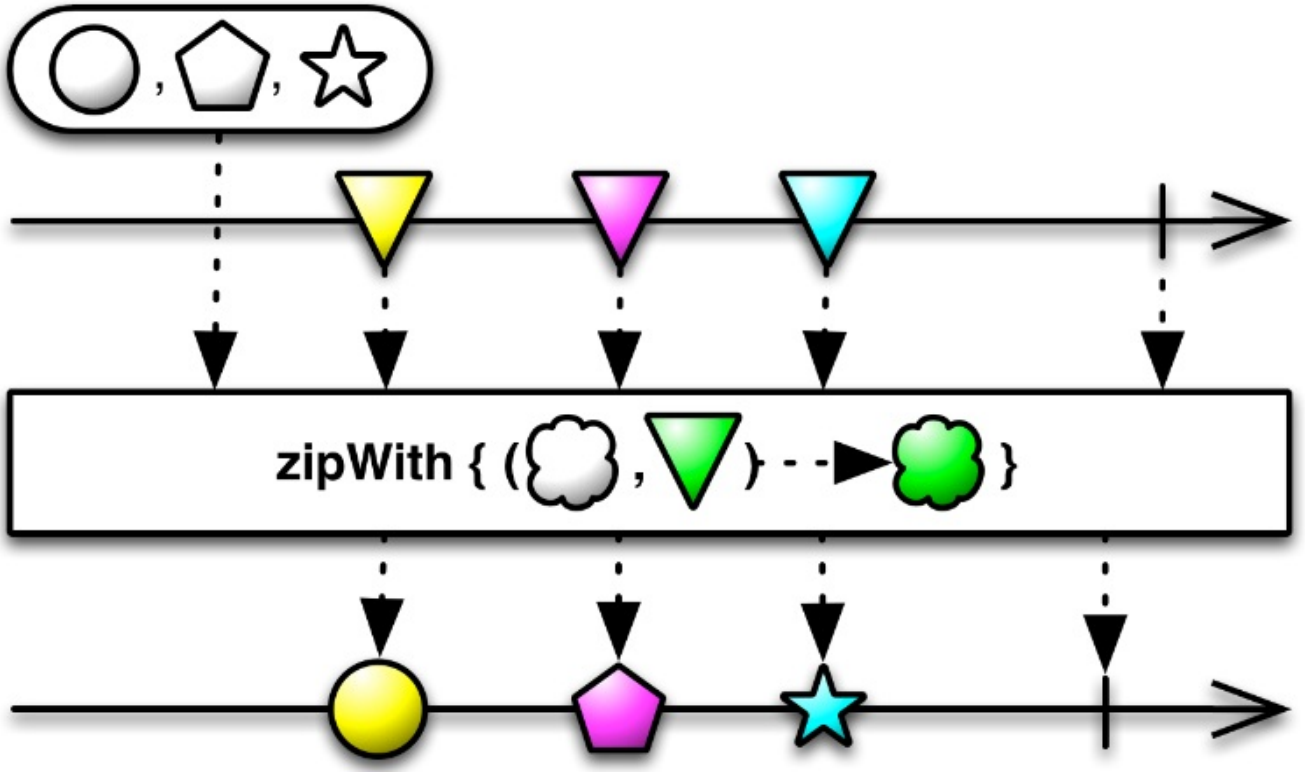
用来合并两个Observable发射的数据项，根据Func2函数生成一个新的值并发射出去。当其中一个Observable发送数据结束或者出现异常后，另一个Observable也将停在发射数据。

```
String[] letters = new String[]{"A", "B", "C", "D", "E", "F", "G", "H"};
Observable<String> letterSequence = Observable.interval(120, TimeUnit.MILLISECONDS)
        .map(new Func1<Long, String>() {
            @Override
            public String call(Long position) {
                return letters[position.intValue()];
            }
        }).take(letters.length);

Observable<Long> numberSequence = Observable.interval(200, TimeUnit.MILLISECONDS).take(5);

Observable.zip(letterSequence, numberSequence, new Func2<String, Long, String>() {
    @Override
    public String call(String letter, Long number) {
        return letter + number;
    }
}).subscribe(new Observer<String>() {
    @Override
    public void onCompleted() {
        System.exit(0);
    }

    @Override
    public void onError(Throwable e) {
        System.out.println("Error:" + e.getMessage());
    }

    @Override
    public void onNext(String result) {
        System.out.print(result + " ");
    }
});
```

程序输出：

```
A0 B1 C2 D3 E4
```

zipWith



zipWith操作符总是接受两个参数，第一个参数是一个Observable或者一个Iterable。

Javadoc: zipWith(Observable,Func2))
Javadoc: zipWith(Iterable,Func2))