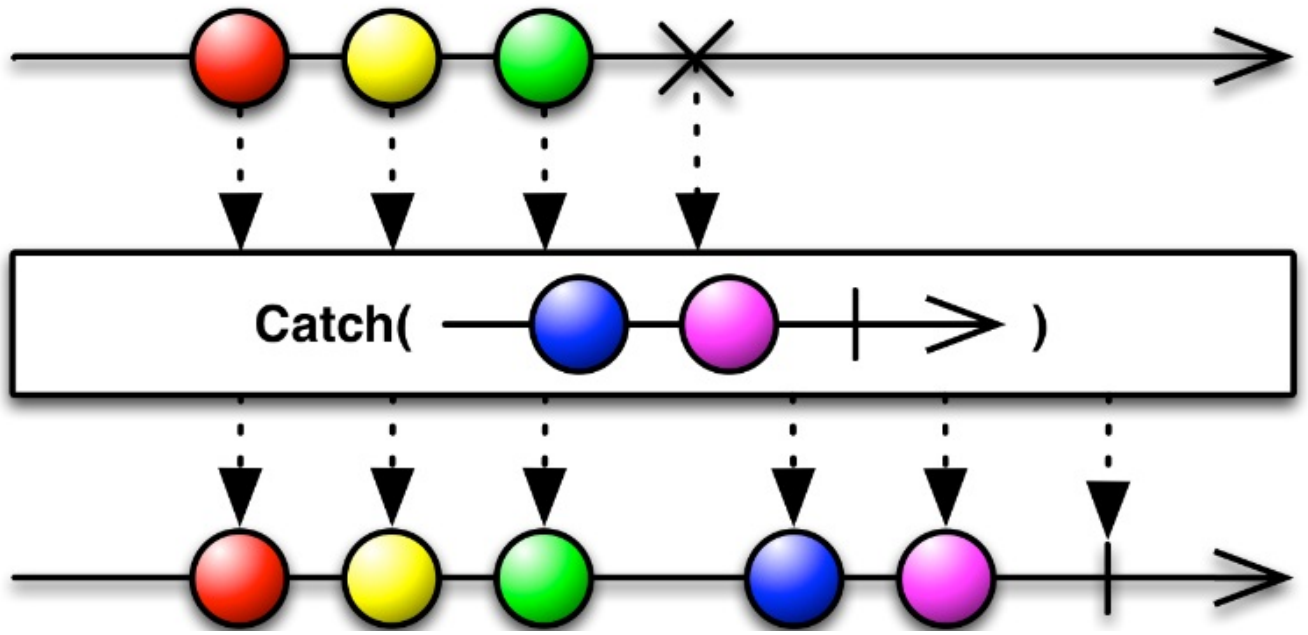
# Rxjava_操作符5_错误处理操作符

## Catch

从 `onError` 通知中恢复发射数据




Catch操作符拦截原始Observable的onError通知，将它替换为其它的数据项或数据序列，让产生的Observable能够正常终止或者根本不终止。

RxJava将Catch实现为三个不同的操作符：

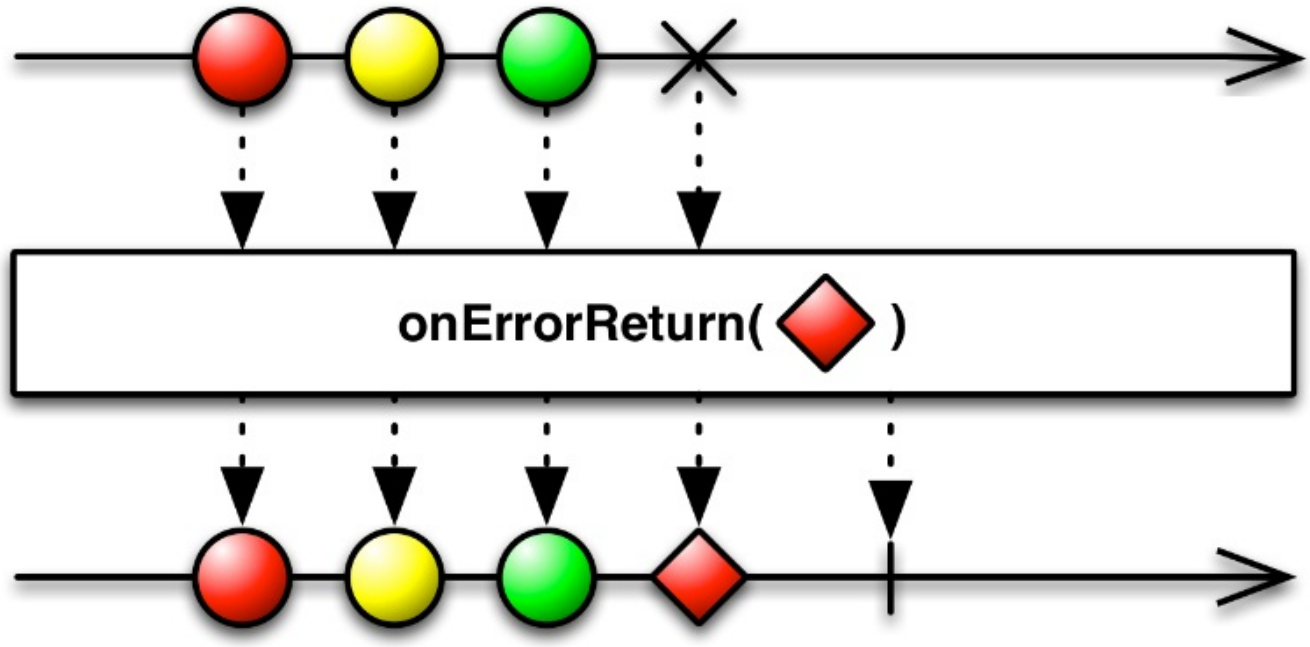
`onErrorReturn` 让Observable遇到错误时发射一个特殊的项并且正常终止。

`onErrorResumeNext` 让Observable在遇到错误时开始发射第二个Observable的数据序列。

`onExceptionResumeNext` 让Observable在遇到错误时继续发射后面的数据项。

## onErrorReturn

onErrorReturn操作符是在Observable发生错误或异常的时候（即将回调oError方法时），拦截错误并执行指定的逻辑，返回一个跟源Observable相同类型的结果，最后回调订阅者的onComplete方法，其流程图如下：

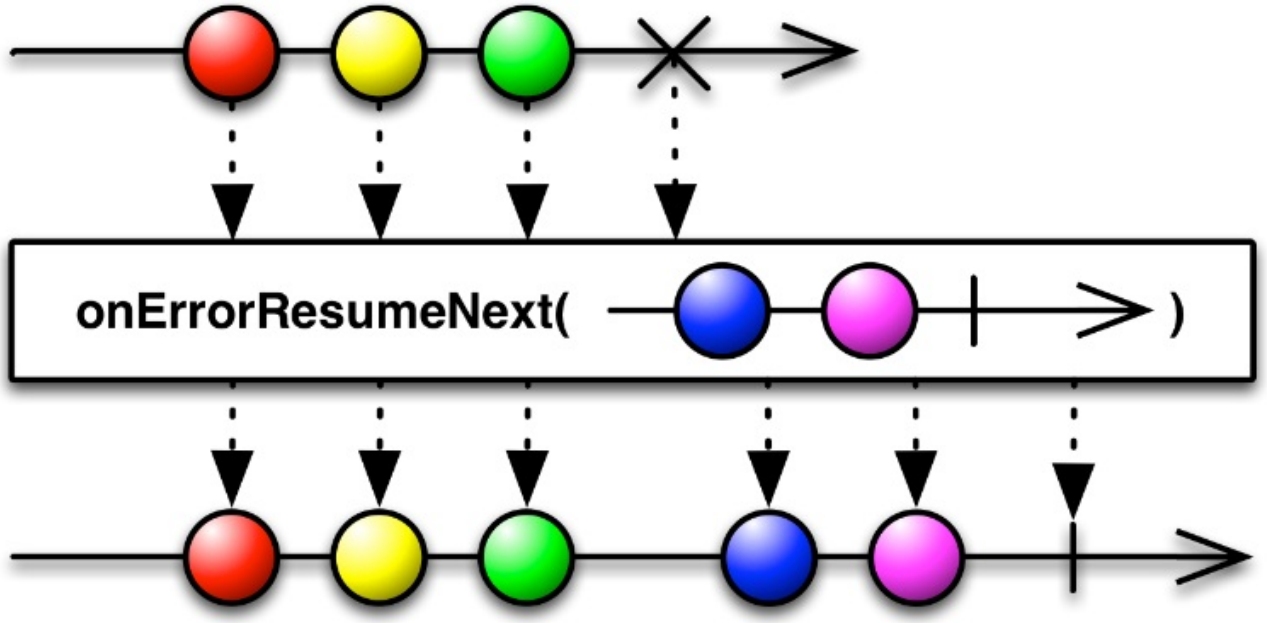



```
Observable<Integer> observable = Observable.create(new Observable.OnSubscribe<I
nteger>() {
            @Override
            public void call(Subscriber<? super Integer> subscriber) {
                if (subscriber.isUnsubscribed()) return;
                //循环输出数字
                try {
                    for (int i = 0; i < 10; i++) {
                        if (i == 4) {
                            throw new Exception("this is number 4 error！");
                        }
                        subscriber.onNext(i);
                    }
                    subscriber.onCompleted();
                } catch (Exception e) {
                    subscriber.onError(e);
                }
            }
        });

        observable.onErrorReturn(new Func1<Throwable, Integer>() {
            @Override
            public Integer call(Throwable throwable) {
                return 1004;
            }
        }).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:0
Next:1
Next:2
Next:3
Next:1004
Sequence complete.
```

## onErrorResumeNext




onErrorResumeNext方法返回一个镜像原有Observable行为的新Observable，后者会忽略前者的onError调用，不会将错误传递给观察者，作为替代，它会开始镜像另一个，备用的Observable。

```
Javadoc: onErrorResumeNext(Func1))

Javadoc: onErrorResumeNext(Observable))
```

```
Observable<Integer> observable = Observable.create(new Observable.OnSubscribe<I
nteger>() {
            @Override
```

```
            public void call(Subscriber<? super Integer> subscriber) {
                if (subscriber.isUnsubscribed()) return;
                //循环输出数字
                try {
                    for (int i = 0; i < 10; i++) {
                        if (i == 4) {
                            throw new Exception("this is number 4 error！");
                        }
                        subscriber.onNext(i);
                    }
                    subscriber.onCompleted();
                } catch (Exception e) {
                    subscriber.onError(e);
                }
            }
        });
        //Observable.just(100,101, 102) onErrorResumeNext的参数的另一种
        observable.onErrorResumeNext(new Func1<Throwable, Observable<? extends
Integer>>() {
            @Override
            public Observable<? extends Integer> call(Throwable throwable) {
                return Observable.just(100,101, 102);
            }
        }).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：

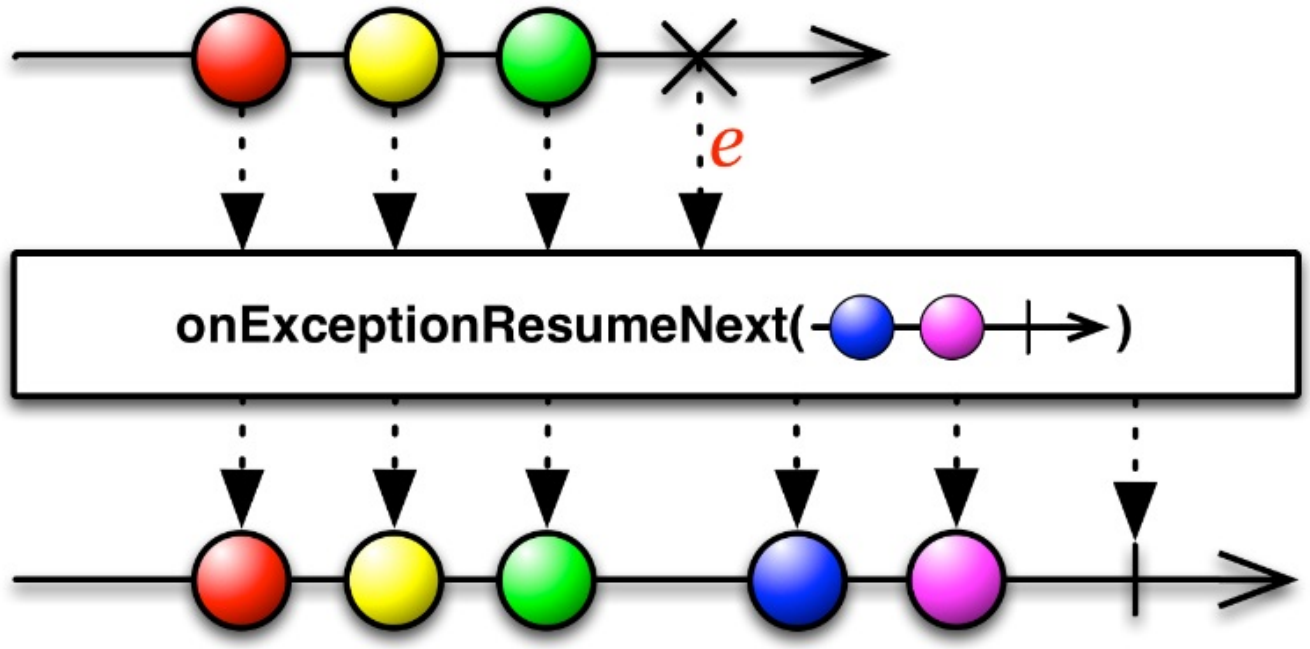
```
Next:0
Next:1
Next:2
Next:3
Next:100
Next:101
Next:102
```

```
Sequence complete
```

## onExceptionResumeNext

和onErrorResumeNext类似，onExceptionResumeNext方法返回一个镜像原有Observable行为的新Observable，也使用一个备用的Observable，不同的是，如果onError收到的Throwable不是一个Exception，它会将错误传递给观察者的onError方法，不会使用备用的Observable。

```
Javadoc: onExceptionResumeNext(Observable))
```



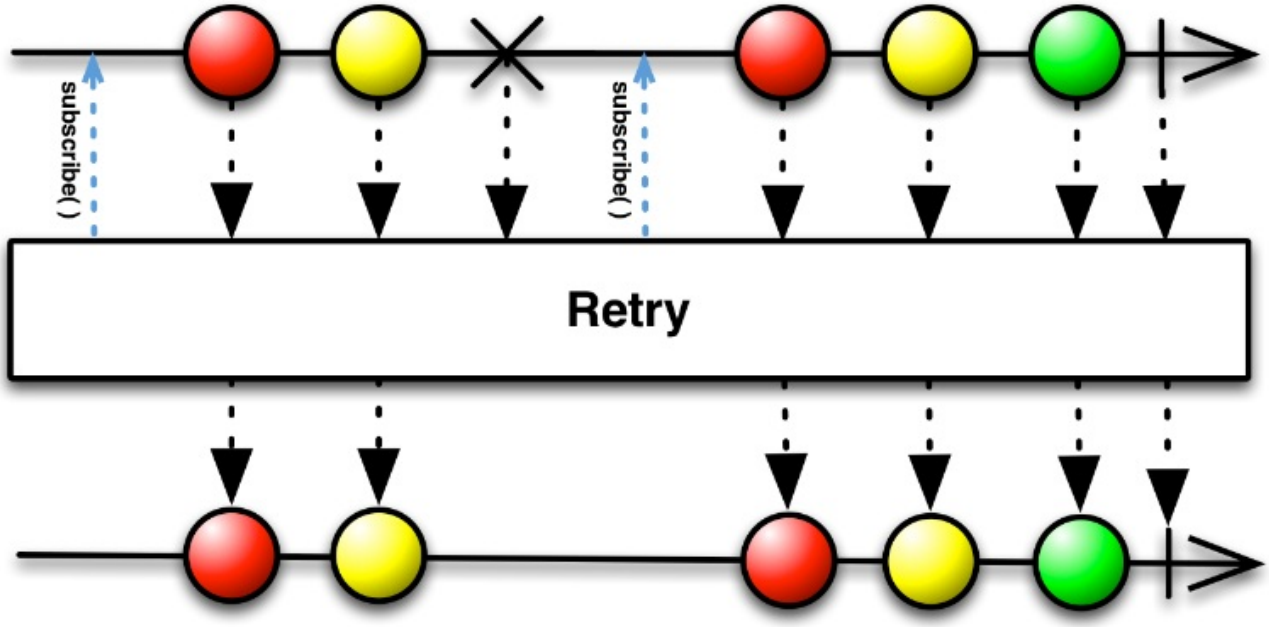

和onErrorResumeNext不同点：`onExceptionResumeNext`如果onError收到的Throwable不是一个Exception，它会将错误传递给观察者的onError方法，而不会调用备用的方法。

```
Observable<Integer> observable = Observable.create(new Observable.OnSubscribe<I
nteger>() {
            @Override
            public void call(Subscriber<? super Integer> subscriber) {
                if (subscriber.isUnsubscribed()) return;
                //循环输出数字
                try {
                    for (int i = 0; i < 10; i++) {
                        if (i == 4) {
                            throw new Exception("onErrorResumeNext,onExceptionR
esumeNext！");

                            throw new Error("onErrorResumeNext！");
                        }
                        subscriber.onNext(i);
                    }
                    subscriber.onCompleted();
```

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

# Retry

如果原始Observable遇到错误，重新订阅它，期望它能正常终止




Retry操作符不会将原始Observable的onError通知传递给观察者，它会订阅这个Observable，再给它一次机会无错误地完成它的数据序列。Retry总是传递onNext通知给观察者，由于重新订阅，可能会造成数据项重复，如上图所示。

`retry()` 无论收到多少次onError通知，无参数版本的retry都会继续订阅并发射原始Observable。

`retry(long)` 接受单个count参数的retry会最多重新订阅指定的次数，如果次数超了，它不会尝试再次订阅，它会把最新的一个onError通知传递给它的观察者。

`retry(Func2)` 还有一个版本的retry接受一个谓词函数作为参数，这个函数的两个参数是：重试次数和导致发射onError通知的Throwable。这个函数返回一个布尔值，如果返回true，retry应该再次订阅和镜像原始的Observable，如果返回false，retry会将最新的一个onError通知传递给它的观察者。

```
        Observable<Integer> observable = Observable.create(new Observable.OnSub
scribe<Integer>() {
            @Override
            public void call(Subscriber<? super Integer> subscriber) {
                if (subscriber.isUnsubscribed()) return;
                //循环输出数字
                try {
                    for (int i = 0; i < 10; i++) {
```

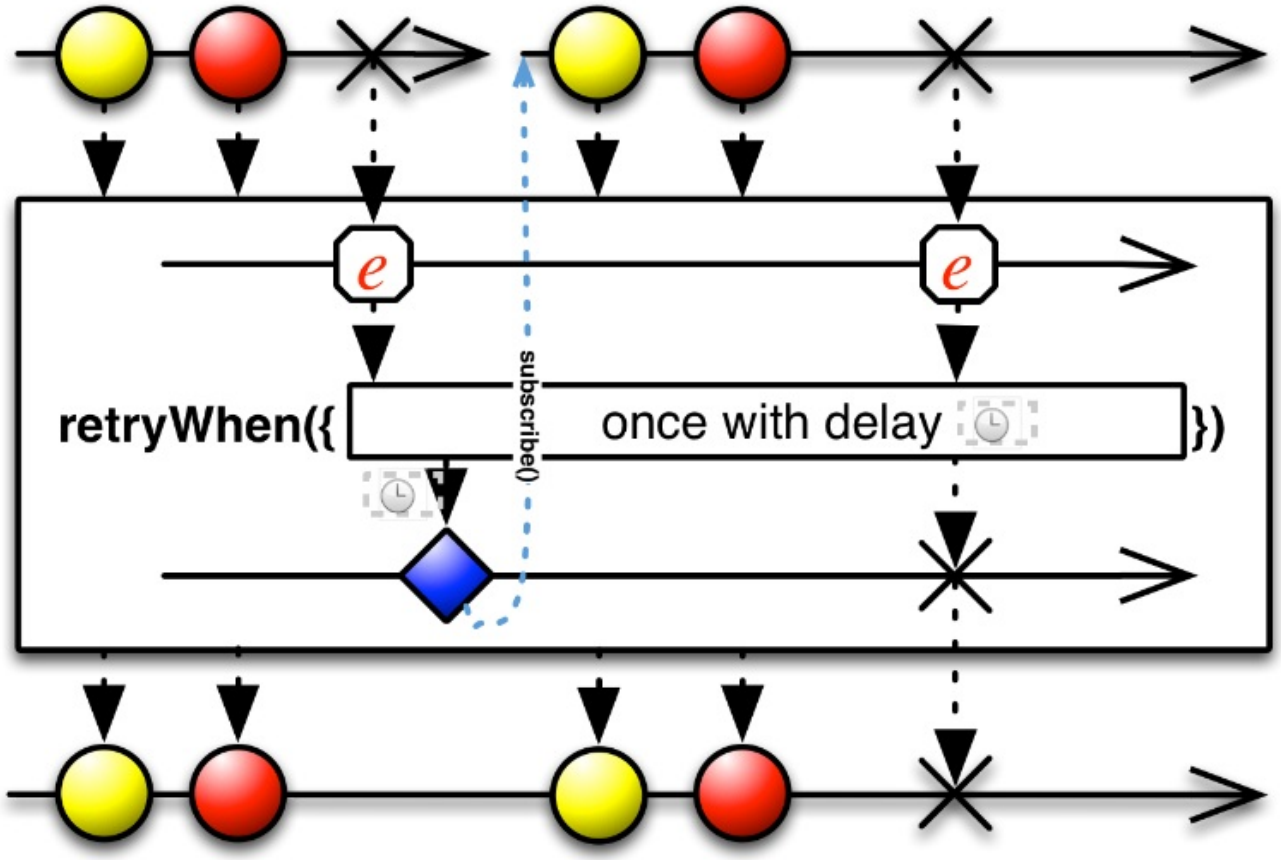
```
                    if (i == 4) {
                        throw new Exception("this is number 4 error！");
                    }
                    subscriber.onNext(i);
                }
                subscriber.onCompleted();
            } catch (Throwable e) {
                subscriber.onError(e);
            }
        }
    });

    observable.retry(new Func2<Integer, Throwable, Boolean>() {
        @Override
        public Boolean call(Integer integer, Throwable throwable) {
            throwable.printStackTrace();
            if(integer>4){
                return false;
            }else{
                return true;
            }
        }
    }).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:0
Next:1
Next:2
Next:3

Next:0
Next:1
Next:2
```

```
Next:3

Next:0
Next:1
Next:2
Next:3
Error: this is number 4 error !
```

## retryWhen




`retryWhen`和`retry`类似，区别是，retryWhen将onError中的Throwable传递给一个函数，这个函数产生另一个Observable，retryWhen观察它的结果再决定是不是要重新订阅原始的Observable。如果这个Observable发射了一项数据，它就重新订阅，如果这个Observable发射的是onError通知，它就将这个通知传递给观察者然后终止。

换个说法：
retryWhen操作符类似于retry操作符，都是在源observable出现错误或者异常时，重新尝试执行源observable的逻辑，不同在于retryWhen操作符是在源Observable出现错误或者异常时，通过回调第二个Observable来判断是否重新尝试执行源Observable的逻辑，如果第二个Observable没有错误或者异常出现，则就会重新尝试执行源Observable的逻辑，否则就会直接回调执行订阅者的onError方法。

```
第一个不行第二个不行，整个就死了，第二个行的话就救活了第一个
```

retryWhen默认在trampoline调度器上执行，你可以通过参数指定其它的调度器。

```
Observable<Integer> observable = Observable.create(new Observable.OnSubscribe<I
nteger>() {
            @Override
            public void call(Subscriber<? super Integer> subscriber) {
                System.out.println("subscribing");
                subscriber.onError(new RuntimeException("always fails"));
            }
        });
        //第一个不行第二个不行，整个就死了，第二个行的话就救活了第一个
        observable.retryWhen(new Func1<Observable<? extends Throwable>, Observa
ble<?>>() {
            @Override
            public Observable<?> call(Observable<? extends Throwable> observabl
e) {
                return observable.zipWith(Observable.range(1, 3), new Func2<Thr
owable, Integer, Integer>() {
                    @Override
                    public Integer call(Throwable throwable, Integer integer) {
                        return integer;
                    }
                }).flatMap(new Func1<Integer, Observable<?>>() {
                    @Override
                    public Observable<?> call(Integer integer) {
                        System.out.println("delay retry by " + integer + " seco
nd(s)");
                        //每一秒中执行一次
                        return Observable.timer(integer, TimeUnit.SECONDS);
                    }
                });
            }
        }).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);
            }
        });
```

```
subscribing
```

```
delay retry by 1 second(s)
subscribing
delay retry by 2 second(s)
subscribing
delay retry by 3 second(s)
subscribing
```