课上所有会用到的demo：

OkAndGreat/RxJavaTeachDemo (github.com)

# 基础知识(课上不讲，大家自己看)

# Rx介绍

## ReactiveX的历史

ReactiveX是Reactive Extensions的缩写，一般简写为Rx，最初是LINQ的一个扩展，由微软的架构师Erik Meijer领导的团队开发，在2012年11月开源，Rx是一个编程模型，目标是提供一致的编程接口，帮助开发者更方便的处理异步数据流，Rx库支持.NET、JavaScript和C++，Rx近几年越来越流行了，现在已经支持几乎全部的流行编程语言了，Rx的大部分语言库由ReactiveX这个组织负责维护，比较流行的有RxJava/RxJS/Rx.NET，社区网站是 reactivex.io。

## 什么是ReactiveX

微软给的定义是，Rx是一个函数库，让开发者可以利用可观察序列和LINQ风格查询操作符来编写异步和基于事件的程序，使用Rx，开发者可以用Observables表示异步数据流，用LINQ操作符查询异步数据流， 用Schedulers参数化异步数据流的并发处理，Rx可以这样定义：Rx = Observables + LINQ + Schedulers。

ReactiveX.io给的定义是，Rx是一个使用可观察数据流进行异步编程的编程接口，ReactiveX结合了观察者模式、迭代器模式和函数式编程的精华。

## ReactiveX的应用

很多公司都在使用ReactiveX，例如Microsoft、Netflix、Github、Trello、SoundCloud。

## ReactiveX宣言

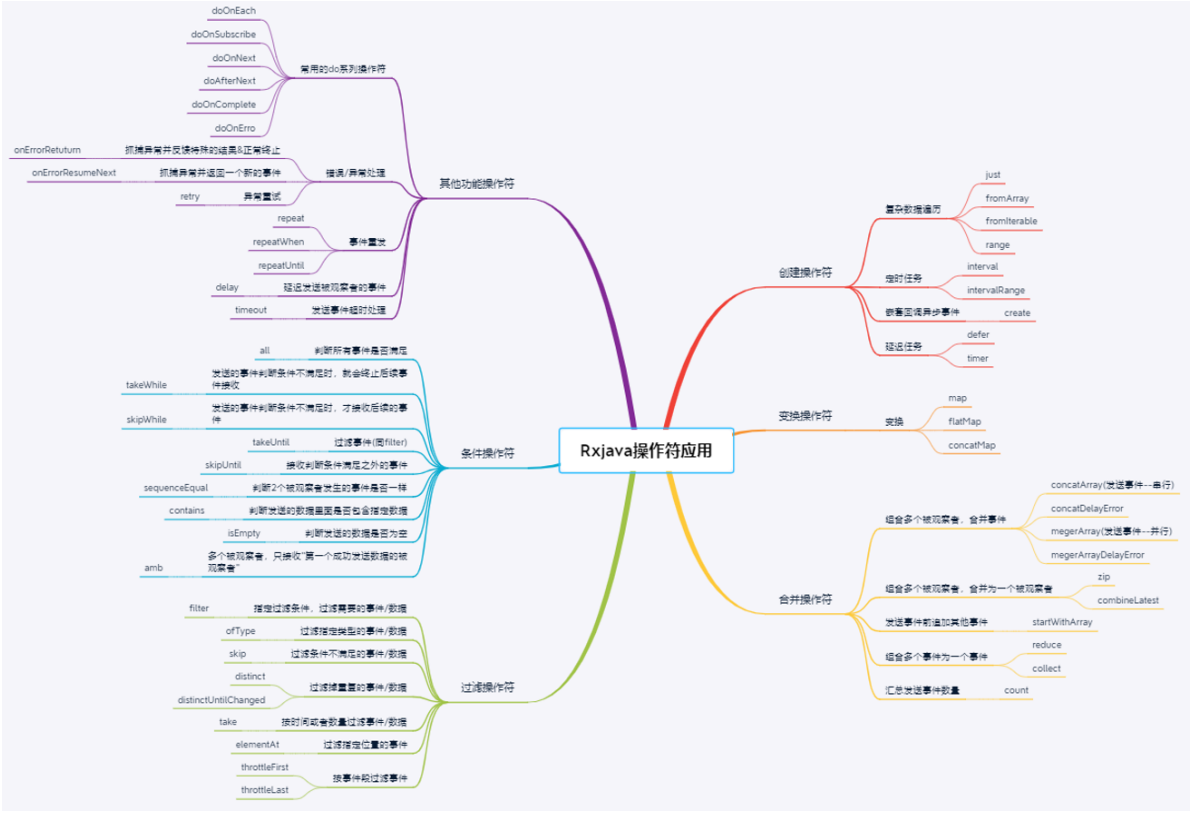
ReactiveX不仅仅是一个编程接口，它是一种编程思想的突破，它影响了许多其它的程序库和框架以及编程语言。

# 部分Rx操作符

看一遍有印象即可，把RxJava的操作符当作Api看待，要用的时候查阅相关资料即可

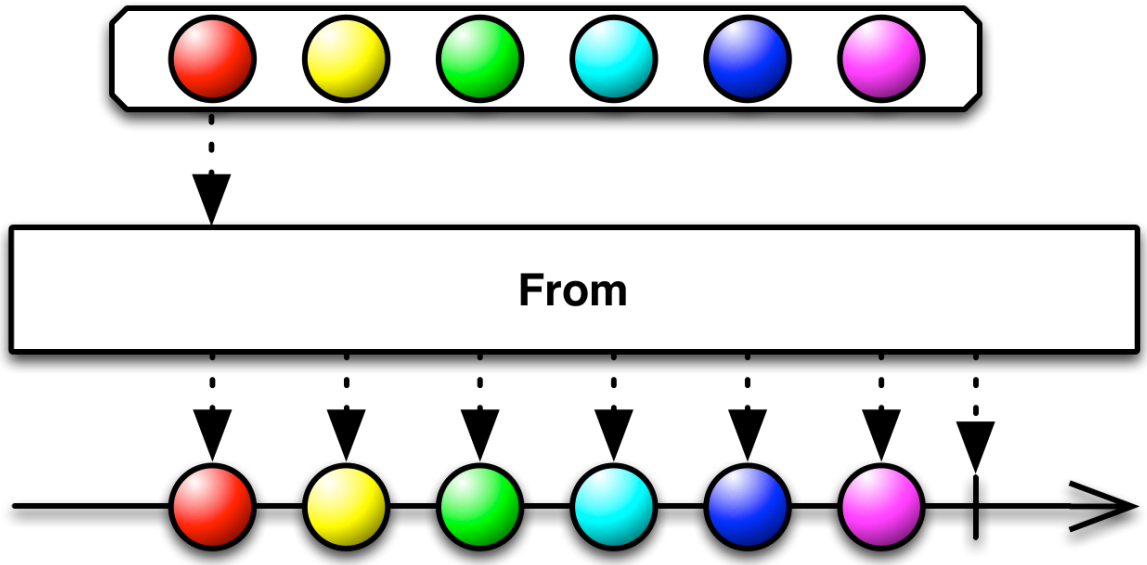
主要记住RxJava有哪几类操作符

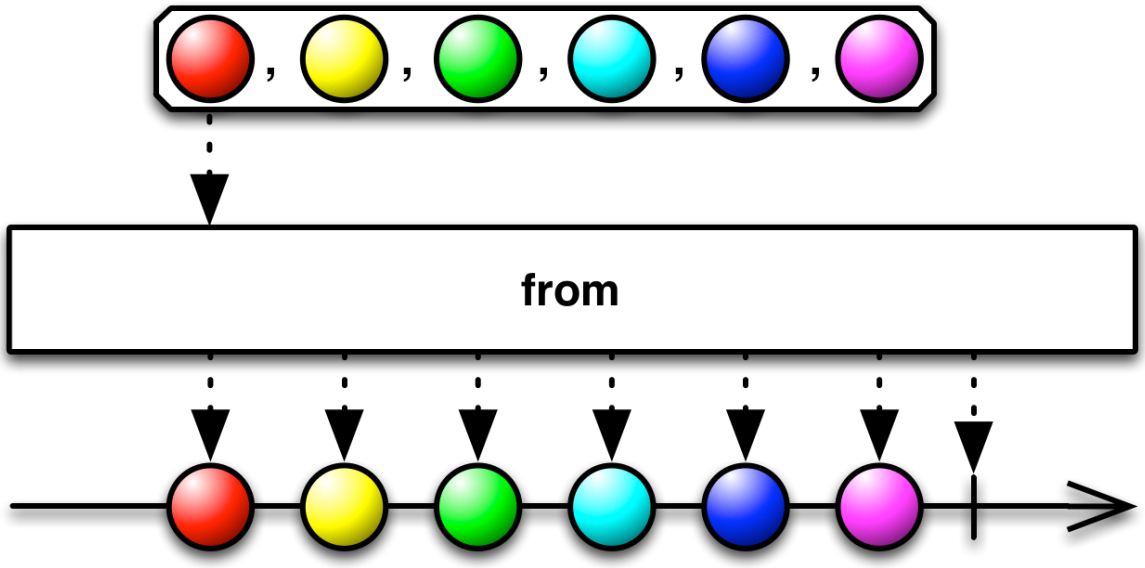
总表：




## From

将其它种类的对象和数据类型转换为Observable






示例代码

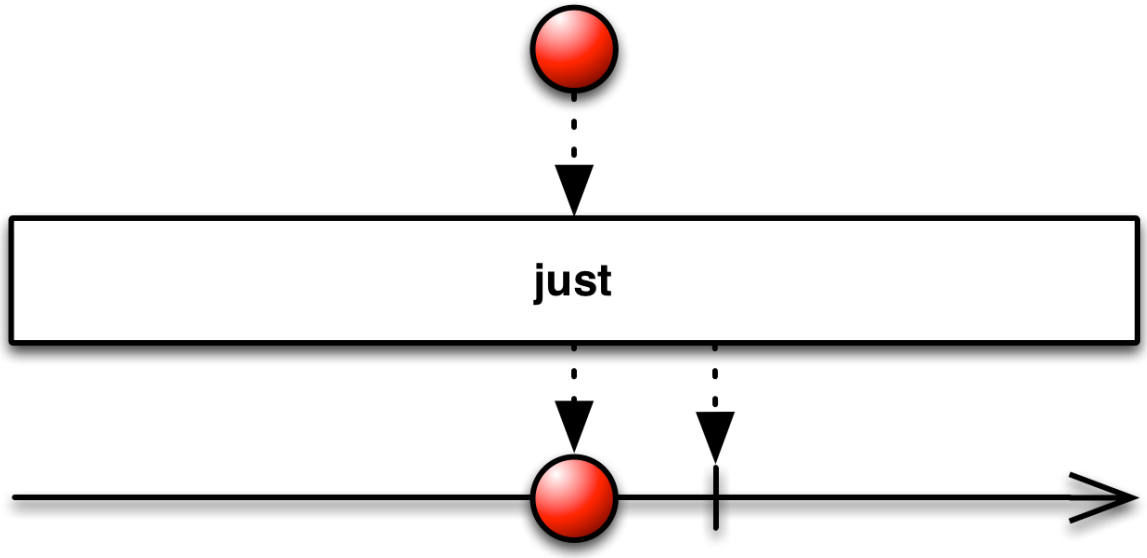
```
Integer[] items = { 0, 1, 2, 3, 4, 5 };
Observable myObservable = Observable.from(items);

myObservable.subscribe(
    new Action1<Integer>() {
        @Override
        public void call(Integer item) {
            System.out.println(item);
        }
    },
    new Action1<Throwable>() {
        @Override
        public void call(Throwable error) {
            System.out.println("Error encountered: " + error.getMessage());
        }
    },
    new Action0() {
        @Override
        public void call() {
            System.out.println("Sequence complete");
        }
    }
);
```

输出

```
0
1
2
3
4
5
Sequence complete
```

## Just

创建一个发射指定值的Observable




Just将单个数据转换为发射那个数据的Observable。

Just类似于From，但是From会将数组或Iterable的数据取出然后逐个发射，而Just只是简单的原样发射，将数组或Iterable当做单个数据。

注意：如果你传递`null`给Just，它会返回一个发射`null`值的Observable。不要误认为它会返回一个空Observable（完全不发射任何数据的Observable），如果需要空Observable你应该使用Empty操作符。

RxJava将这个操作符实现为`just`函数，它接受一至九个参数，返回一个按参数列表顺序发射这些数据的Observable。

示例代码：

```
Observable.just(1, 2, 3)
          .subscribe(new Subscriber<Integer>() {
        @Override
        public void onNext(Integer item) {
            System.out.println("Next: " + item);
        }

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

        @Override
        public void onCompleted() {
            System.out.println("Sequence complete.");
        }
    });
```

输出

```
Next: 1
Next: 2
Next: 3
Sequence complete.
```

## Defer

顾名思义，延迟创建

示例代码：

```
private String[] strings1 = {"Hello", "World"};
    private String[] strings2 = {"Hello", "RxJava"};

    private void test() {
        Observable<String> observable = Observable.defer(new
Func0<Observable<String>>() {
            @Override
            public Observable<String> call() {
                return Observable.from(strings1);
            }
        });

        strings1 = strings2; //订阅前把strings给改了
        observable.subscribe(new Action1<String>() {
            @Override
            public void call(String s) {
                System.out.println("onNext--> " + s);
            }
        }, new Action1<Throwable>() {
            @Override
            public void call(Throwable throwable) {
                System.out.println("onError--> " + throwable.getMessage());
            }
        }, new Action0() {
            @Override
            public void call() {
                System.out.println("onComplete");
            }
        });
    }
```

结果：

```
onNext--> Hello
onNext--> RxJava
onComplete
```

从结果可以知道defer操作符起到的是一个“预创建”的作用，真正创建是发生在订阅的时候

## Range

创建一个发射一组整数序列的Observable

示例代码：

```
Observable.range(3, 8)
        .subscribe(new Action1<Object>() {
            @Override
            public void call(Object o) {
                System.out.println("onNext--> " + o);
            }
        }, new Action1<Throwable>() {
            @Override
            public void call(Throwable throwable) {
                System.out.println("onError--> " + throwable.getMessage());
            }
        }, new Action0() {
            @Override
            public void call() {
                System.out.println("onComplete");
            }
        });
```

结果：

```
onNext--> 3
onNext--> 4
onNext--> 5
onNext--> 6
onNext--> 7
onNext--> 8
onNext--> 9
onNext--> 10
onComplete
```

## Empty

创建一个空的，不会发射任何事件（数据）的Observable

示例代码：

```
Observable.empty()
        .subscribe(new Action1<Object>() {
            @Override
            public void call(Object o) {
                System.out.println("onNext--> " + o);
            }
        }, new Action1<Throwable>() {
            @Override
            public void call(Throwable throwable) {
                System.out.println("onError--> " + throwable.getMessage());
            }
        }, new Action0() {
            @Override
            public void call() {
                System.out.println("onComplete");
```

```
            }
        });
```

结果：

```
onComplete
```

## Interval

创建一个无限的计时序列，每隔一段时间发射一个数字（从0开始）的Observable

示例代码：

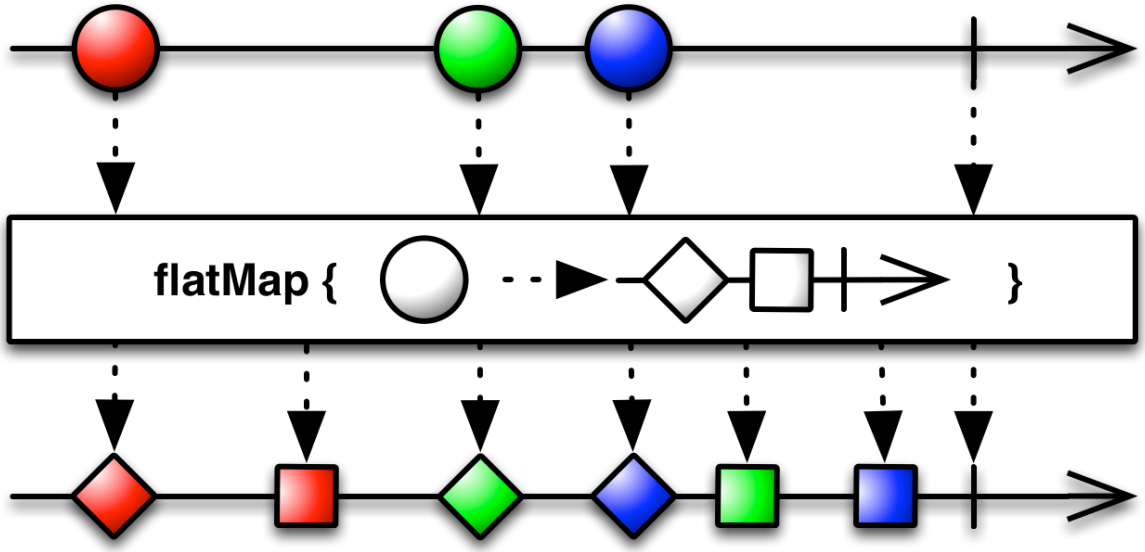
```
    Observable.interval(1, TimeUnit.SECONDS)
         .subscribe(new Action1<Object>() {
             @Override
             public void call(Object o) {
                 System.out.println("onNext--> " + o);
             }
         }, new Action1<Throwable>() {
             @Override
             public void call(Throwable throwable) {
                 System.out.println("onError--> " + throwable.getMessage());
             }
         }, new Action0() {
             @Override
             public void call() {
                 System.out.println("onComplete");
             }
         });

    System.in.read();//阻塞当前线程，防止JVM结束程序
```

结果：

```
onNext--> 0
onNext--> 1
onNext--> 2
onNext--> 3
onNext--> 4
onNext--> 5
onNext--> 6
...
```

## FlatMap

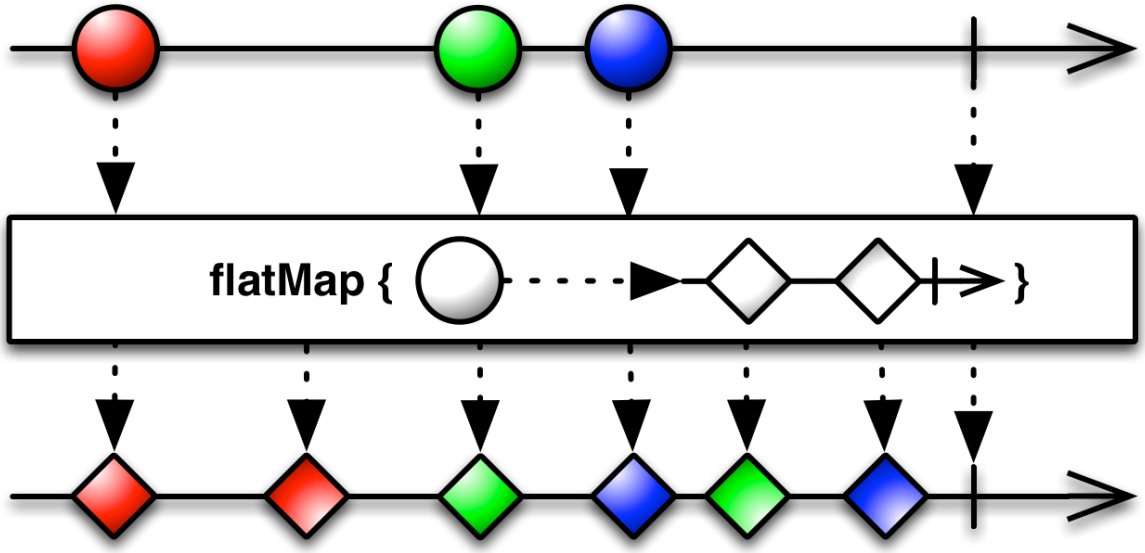
`FlatMap`将一个发射数据的Observable变换为多个Observables，然后将它们发射的数据合并后放进一个单独的Observable




`FlatMap`操作符使用一个指定的函数对原始Observable发射的每一项数据执行变换操作，这个函数返回一个本身也发射数据的Observable，然后`FlatMap`合并这些Observables发射的数据，最后将合并后的结果当做它自己的数据序列发射。

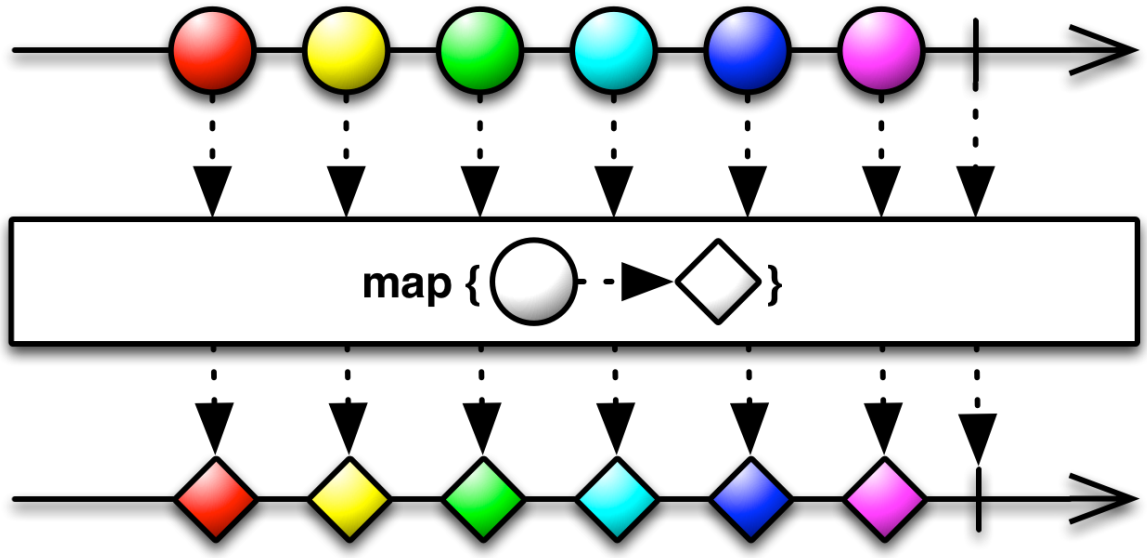
这个方法是很有用的，例如，当你有一个这样的Observable：它发射一个数据序列，这些数据本身包含Observable成员或者可以变换为Observable，因此你可以创建一个新的Observable发射这些次级Observable发射的数据的完整集合。

注意：`FlatMap`对这些Observables发射的数据做的是合并(`merge`)操作，因此它们可能是交错的。




## Map

对Observable发射的每一项数据应用一个函数，执行变换操作




`Map` 操作符对原始Observable发射的每一项数据应用一个我们写的变换函数，然后返回一个发射这些结果的Observable。

示例代码：

```
Observable.create(new ObservableOnSubscribe<Integer>() {
    @Override
    public void subscribe(@NonNull ObservableEmitter<Integer> emitter) {
        emitter.onNext(1);
    }
})
        .map(new Function<Integer, String>() {
            @Override
            public String apply(Integer integer) {
                return integer.toString();
            }
        })
        .subscribe(new Observer<String>() {
            @Override
            public void onSubscribe(@NonNull Disposable d) {

            }

            @Override
            public void onNext(@NonNull String s) {

            }


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

            }

            @Override
            public void onComplete() {

            }
        });
```

## Buffer

将原发射出来的数据已count为单元打包之后在分别发射出来

示例代码

```
Observable.just(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
        .buffer(3)
        .subscribe(new Action1<Object>() {
            @Override
            public void call(Object o) {
                System.out.println("onNext--> " + o);
            }
        }, new Action1<Throwable>() {
            @Override
            public void call(Throwable throwable) {
                System.out.println("onError--> " + throwable.getMessage());
            }
        }, new Action0() {
            @Override
            public void call() {
                System.out.println("onComplete");
            }
        });
```

结果

```
onNext--> [1, 2, 3]
onNext--> [4, 5, 6]
onNext--> [7, 8, 9]
onNext--> [10]
onComplete
```

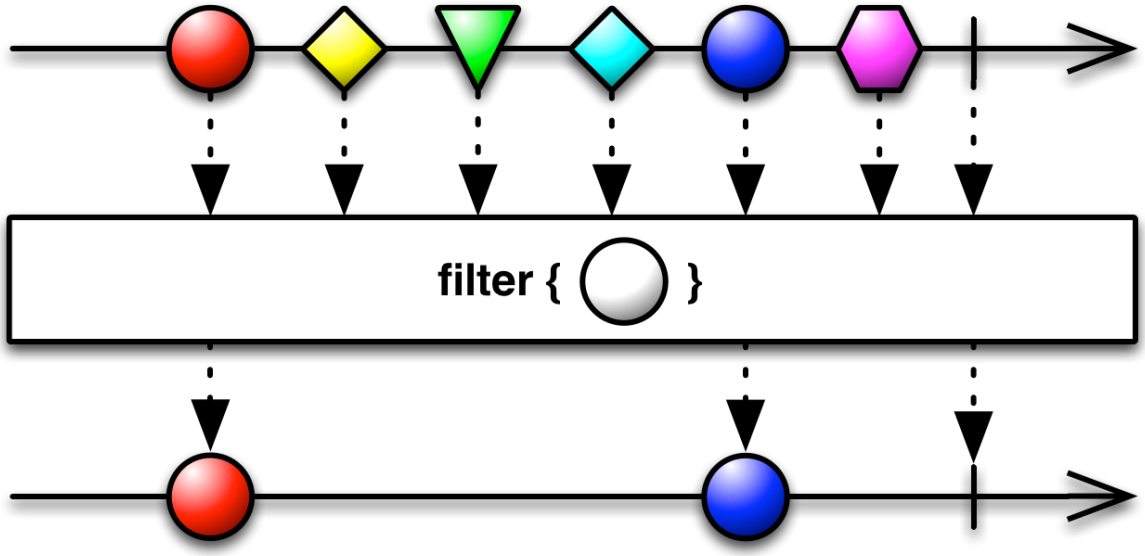
## Filter

顾名思义 是一个过滤数据的操作符

只发射通过了谓词测试的数据项


```
filter(x => x > 10)
```


`Filter`操作符使用你指定的一个谓词函数检测数据项，只有通过检测的数据才会被发射。




RxJava将这个操作符实现为`filter`函数。

示例代码1:

```
Observable.just(1, 2, 3, 4, 5)
          .filter(new Func1<Integer, Boolean>() {
              @Override
              public Boolean call(Integer item) {
                return( item < 4 );
              }
          }).subscribe(new Subscriber<Integer>() {
        @Override
        public void onNext(Integer item) {
            System.out.println("Next: " + item);
        }

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

        @Override
        public void onCompleted() {
            System.out.println("Sequence complete.");
        }
    });
```

输出

```
Next: 1
Next: 2
Next: 3
Sequence complete.
```

示例代码2:

```
Observable.just("Hello", "RxJava", "Nice to meet you")
        .filter(new Func1<String, Boolean>() {
            @Override
            public Boolean call(String s) {
```

```
                //这里的显示条件是s的长度大于5，而Hello的长度刚好是5
                //所以不能满足条件
                return s.length() > 5;
            }
        })
        .subscribe(new Action1<String>() {
            @Override
            public void call(String s) {
                System.out.println("onNext--> " + s);
            }
        }, new Action1<Throwable>() {
            @Override
            public void call(Throwable throwable) {
                System.out.println("onError--> " + throwable.getMessage());
            }
        }, new Action0() {
            @Override
            public void call() {
                System.out.println("onComplete");
            }
        });
```

输入2：

```
onNext--> RxJava
onNext--> Nice to meet you
onComplete
```

## Take

只发射前N项的数据（***takeLast与take想反，只取最后N项数据***）

```
Observable.just("Hello", "RxJava", "Nice to meet you")
        .take(2)
        //.taktLast(2)
        .subscribe(new Action1<String>() {
            @Override
            public void call(String s) {
                System.out.println("onNext--> " + s);
            }
        }, new Action1<Throwable>() {
            @Override
            public void call(Throwable throwable) {
                System.out.println("onError--> " + throwable.getMessage());
            }
        }, new Action0() {
            @Override
            public void call() {
                System.out.println("onComplete");
            }
        });
    }
```

```
onNext--> Hello
onNext--> RxJava
onComplete
```

## Skip

发射数据时忽略前N项数据（***skpiLast忽略后N项数据***）

```
Observable.just("Hello", "RxJava", "Nice to meet you")
        .skip(2)
        //.skipLast(2)
        .subscribe(new Action1<String>() {
            @Override
            public void call(String s) {
                System.out.println("onNext--> " + s);
            }
       }, new Action1<Throwable>() {
            @Override
            public void call(Throwable throwable) {
                System.out.println("onError--> " + throwable.getMessage());
            }
        }, new Action0() {
            @Override
            public void call() {
                System.out.println("onComplete");
            }
        });
    }
```

```
onNext--> Nice to meet you
onComplete
```

## ElementAt

获取原数据的第N项数据作为唯一的数据发射出去（***elementAtOrDefault会在index超出范围时，给出一个默认值发射出来***）

```
Observable.just("Hello", "RxJava", "Nice to meet you")
        .elementAtOrDefault(1, "Great")
        .subscribe(new Action1<String>() {
            @Override
            public void call(String s) {
                System.out.println("onNext--> " + s);
            }
        }, new Action1<Throwable>() {
            @Override
            public void call(Throwable throwable) {
                System.out.println("onError--> " + throwable.getMessage());
            }
        }, new Action0() {
            @Override
            public void call() {
                System.out.println("onComplete");
            }
        });
```

```
onNext--> RxJava
onComplete
```

## Distinct

过滤掉重复项

```
Observable.just("Hello", "Hello", "Hello", "RxJava", "Nice to meet you")
        .distinct()
        .subscribe(new Action1<String>() {
            @Override
            public void call(String s) {
                System.out.println("onNext--> " + s);
            }
        }, new Action1<Throwable>() {
            @Override
            public void call(Throwable throwable) {
                System.out.println("onError--> " + throwable.getMessage());
            }
        }, new Action0() {
            @Override
            public void call() {
                System.out.println("onComplete");
            }
        });
```

```
onNext--> Hello
onNext--> RxJava
onNext--> Nice to meet you
onComplete
```

## Merge

将多个Observables发射的数据合并后在发射

```
Observable.merge(Observable.just(1, 2, 3), Observable.just(4, 5),
        Observable.just(6, 7), Observable.just(8, 9, 10))
        .subscribe(new Action1<Integer>() {
            @Override
            public void call(Integer s) {
                System.out.println("onNext--> " + s);
            }
        }, new Action1<Throwable>() {
            @Override
            public void call(Throwable throwable) {
                System.out.println("onError--> " + throwable.getMessage());
            }
        }, new Action0() {
            @Override
            public void call() {
                System.out.println("onComplete");
            }
        });
```

```
onNext--> 1
onNext--> 2
onNext--> 3
onNext--> 4
onNext--> 5
onNext--> 6
onNext--> 7
onNext--> 8
onNext--> 9
onNext--> 10
onComplete
```

# RxJava与Retrofit结合

常规写法:

```
public interface Api {

    @GET("/article/top/json")
    Call<bean> getData();

}
```

```
retrofit = new Retrofit.Builder()
        .baseUrl(BASEURL)
        .addConverterFactory(GsonConverterFactory.create())
        .build();

retrofit.create(Api.class).getData().enqueue(new Callback<bean>() {
    @Override
    public void onResponse(Call<bean> call, Response<bean> response) {
        Log.d(TAG, "onResponse: -->" +
response.body().getData().get(0).toString());
    }

    @Override
    public void onFailure(Call<bean> call, Throwable t) {

    }
});
```

与RxJava结合:注意四点

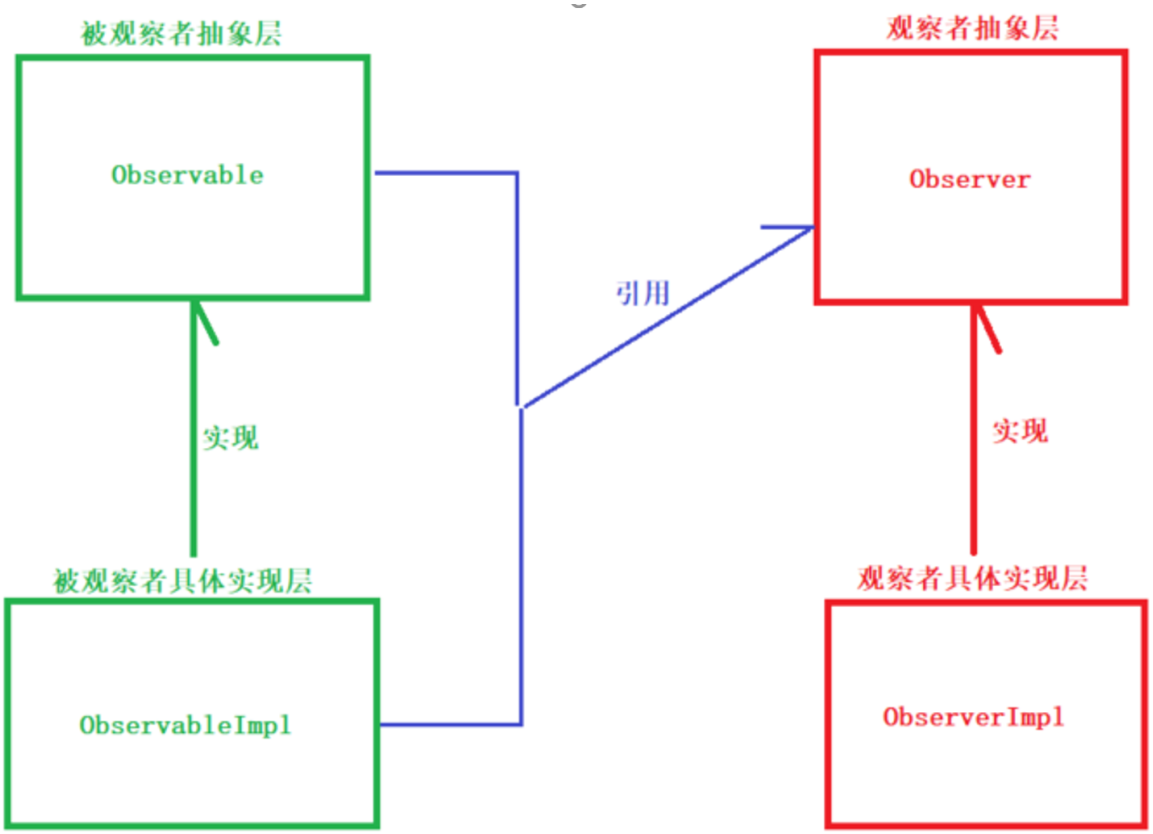
```
1.改变返回类型为Observable
public interface Api {

    @GET("/article/top/json")
    Observable<bean> getData();

}
2. addCallAdapterFactory(RxJava3CallAdapterFactory.create())
        retrofit = new Retrofit.Builder()
                .baseUrl(BASEURL)
                .addConverterFactory(GsonConverterFactory.create())
                .addCallAdapterFactory(RxJava3CallAdapterFactory.create())
```

```
                .build();

3.
        retrofit.create(Api.class)
                .getData()
                .subscribeOn(Schedulers.io())
                .observeOn(AndroidSchedulers.mainThread())
                .subscribe(new Observer<bean>() {
                    @Override
                    public void onSubscribe(@NonNull Disposable d) {
                        disposable = d;
                    }

                    @Override
                    public void onNext(@NonNull bean bean) {
                        Log.d(TAG, "onNext: -->" +
bean.getData().get(0).toString());
                    }

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

                    }

                    @Override
                    public void onComplete() {

                    }
                });

4.如果activity销毁时要让RxJava停止工作
        @Override
    protected void onDestroy() {
        if (disposable != null)
            if (!disposable.isDisposed())
                disposable.dispose();
        super.onDestroy();
    }
```

# 观察者设计模式

## 观察者模式的定义

在对象之间定义了一对多的依赖，当一个对象改变状态，依赖它的对象会收到通知并自动更新。




## 使用场景例子

有一个微信公众号服务，不定时发布一些消息，关注公众号就可以收到推送消息，取消关注就收不到推送消息。

## Android中的观察者模式

android源码中也有很多使用了观察者模式，比如OnClickListener、ContentObserver、android.database.Observable等

以OnClickListener为例

OnClickListener为观察者，View为被观察者，通过setOnClickListener俩者完成订阅关系，View在被点击的时候通知观察者，也就是OnClickListener.

# RxJava源码

## 1.观察者和被观察者是怎么完成订阅关系并发送信息的

```
//demo1
        //观察者和被观察者怎么完成订阅过程并传递消息的
        Observable.create(new ObservableOnSubscribe<Integer>() {
            @Override
            public void subscribe(@NonNull ObservableEmitter<Integer> emitter) {
                emitter.onNext(1);
            }
        })

                //导火索，将被观察者与观察者连接在一起
                .subscribe(new Observer<Integer>() {
                    @Override
```

```
                    public void onSubscribe(@NonNull Disposable d) {

                    }

                    @Override
                    public void onNext(@NonNull Integer integer) {
                        Log.d(TAG, "onNext: 我是自定义Observer，我收到被观察者的信息
啦！数字为-->" + integer);
                    }

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

                    }

                    @Override
                    public void onComplete() {

                    }
                });
```

首先看到Observable.create

```
public static <T> Observable<T> create(@NonNull ObservableOnSubscribe<T> source)
{
    Objects.requireNonNull(source, "source is null");
    return RxJavaPlugins.onAssembly(new ObservableCreate<>(source));
}
```

首先进行了判空，然后返回RxJavaPlugins.onAssembly(new ObservableCreate<>(source))

对于RxJavaPlugins.onAssembly在分析RxJava源码时经常能看见，它是一个Hook方法，可以用来对RxJava的运行过程进行全局监听

```
public static <T> Observable<T> onAssembly(@NonNull Observable<T> source) {
        Function<? super Observable, ? extends Observable> f =
onObservableAssembly;
        if (f != null) {
            return apply(f, source);
        }
        return source;
    }
```

Function<? super Observable, ? extends Observable> f我们没有去设置的话一般为null，所以会直接返回source，也就是说如果没有特意设置我们传递什么参数给RxJavaPlugins.onAssembly它就会原封不动的返回，因此我们下面分析RxJava源码时再碰到这个Hook方法可以忽略。

因此Observable.create的结果就是创建了ObservableCreate这个类并把我们创建的匿名内部类当作参数传递了进去

```
final ObservableOnSubscribe<T> source;

public ObservableCreate(ObservableOnSubscribe<T> source) {
    this.source = source;
}
```

再看subscribe

```
@SchedulerSupport(SchedulerSupport.NONE)
    @Override
    public final void subscribe(@NonNull Observer<? super T> observer) {
        Objects.requireNonNull(observer, "observer is null");
        try {
            observer = RxJavaPlugins.onSubscribe(this, observer);

            Objects.requireNonNull(observer, "The RxJavaPlugins.onSubscribe hook
returned a null Observer. Please change the handler provided to
RxJavaPlugins.setOnObservableSubscribe for invalid null returns. Further reading:
https://github.com/ReactiveX/RxJava/wiki/Plugins");

            subscribeActual(observer);
        } catch (NullPointerException e) { // NOPMD
            throw e;
        } catch (Throwable e) {
            Exceptions.throwIfFatal(e);
            // can't call onError because no way to know if a Disposable has
been set or not
            // can't call onSubscribe because the call might have set a
Subscription already
            RxJavaPlugins.onError(e);

            NullPointerException npe = new NullPointerException("Actually not,
but can't throw other exceptions due to RS");
            npe.initCause(e);
            throw npe;
        }
    }
```

核心代码只有一句

subscribeActual(observer)

直接看到ObservableCreate的subscribeActual

```
@Override
protected void subscribeActual(Observer<? super T> observer) {
    CreateEmitter<T> parent = new CreateEmitter<>(observer);
    observer.onSubscribe(parent);

    try {
        source.subscribe(parent);
    } catch (Throwable ex) {
        Exceptions.throwIfFatal(ex);
        parent.onError(ex);
    }
}
```

可以看到这里new了一个CreateEmitter并将观察者当作参数传给了它，后面再通过source.subscribe(parent);将CreateEmitter与被观察者进行订阅，看到这里我们的第一个目的就达到了.

接下来看被观察者是怎么发送消息给观察者的

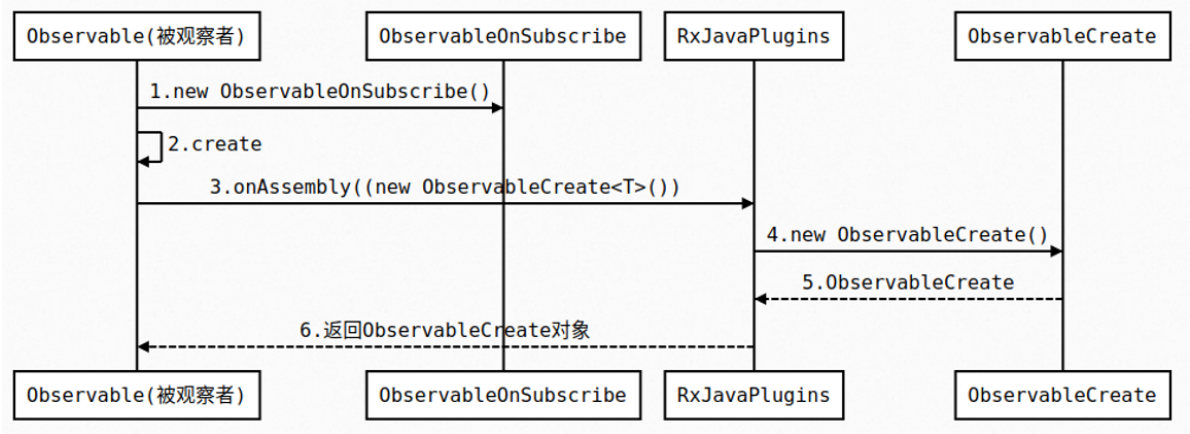
```
Observable.create(new ObservableOnSubscribe<Integer>() {
    @Override
    public void subscribe(@NonNull ObservableEmitter<Integer> emitter) {
        emitter.onNext(1);
    }
})
```

source就是这里我们创建的匿名内部类，通过emitter.onNext(1);消息就从被观察者传到了观察者

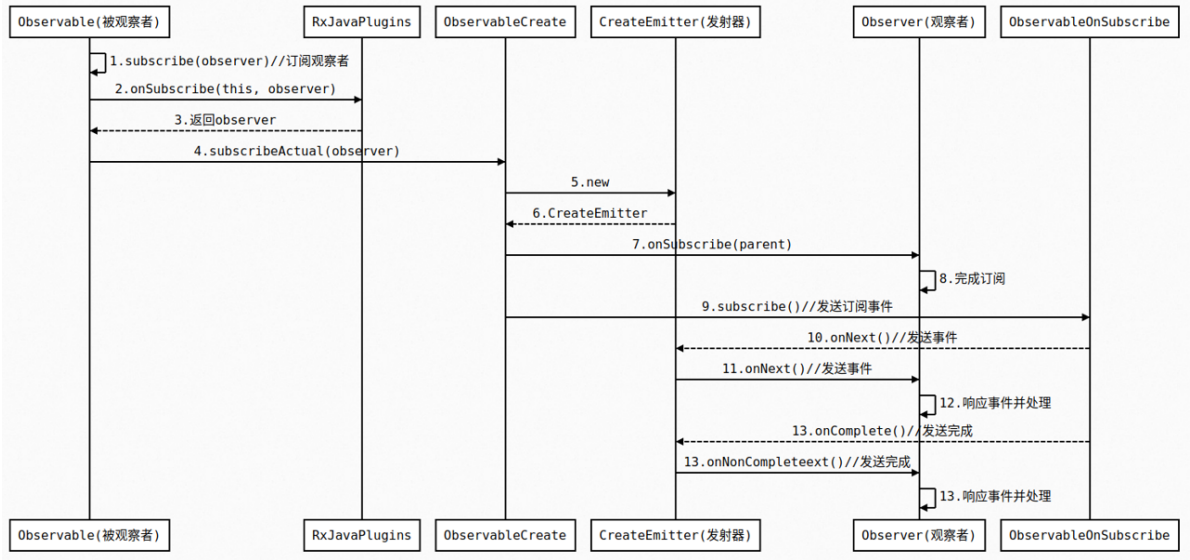
```
@Override
public void onNext(T t) {
    if (t == null) {
        onError(ExceptionHelper.createNullPointerException("onNext called with a
null value."));
        return;
    }
    if (!isDisposed()) {
        observer.onNext(t);
    }
}
```

分析完上述源码后发现RxJava用的并不是纯粹的的观察者模式，而是扩展过的。

Observable创建过程时序图：




Observable 与 Observer 订阅的过程时序图：



## 2.操作符的原理

```
Observable.create(new ObservableOnSubscribe<Integer>() {
    @Override
    public void subscribe(@NonNull ObservableEmitter<Integer> emitter) {
        emitter.onNext(1);
    }
})
        .map(new Function<Integer, String>() {
            @Override
            public String apply(Integer integer) {
                return integer.toString();
            }
        })
        //导火索，将被观察者与观察者连接在一起
        .subscribe(new Observer<String>() {
            @Override
            public void onSubscribe(@NonNull Disposable d) {

            }

            @Override
            public void onNext(@NonNull String s) {
                Log.d(TAG, "onNext: 我是自定义Observer，我收到被观察者的信息啦！信息的类
型为-->"
                        + s.getClass().getName());
            }


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

            }

            @Override
            public void onComplete() {

            }
        });
```

和上面的代码相比只加了

```
.map(new Function<Integer, String>() {
            @Override
            public String apply(Integer integer) {
                return integer.toString();
            }
        })
```

由上面的分析可知

```
Observable.create(new ObservableOnSubscribe<Integer>() {
            @Override
            public void subscribe(@NonNull ObservableEmitter<Integer> emitter) {
                emitter.onNext(1);
            }
        })
```

做的就是new 了一个ObservableCreate并将传入的匿名内部类存储下来（这个匿名内部类ObservableOnSubscribe是真正干活的类，也即发送消息），然后返回ObservableCreate

因此要分析map操作符，我们看ObservableCreate的map方法，而ObservableCreate是继承自Observable，它也没有重写这个方法，所以看到Observable的map方法

```
public final <R> Observable<R> map(@NonNull Function<? super T, ? extends R>
mapper) {
        Objects.requireNonNull(mapper, "mapper is null");
        return RxJavaPlugins.onAssembly(new ObservableMap<>(this, mapper));
    }
```

这里又有一个钩子方法，通过上面的分析可知可以忽略，直接看new ObservableMap<>(this, mapper)做了什么事即可

```
public ObservableMap(ObservableSource<T> source, Function<? super T, ? extends
U> function) {
    super(source);
    this.function = function;
}
```

这里其实和ObservableCreate做的事情差不多，将ObservableCreate存储下来并且将转换函数存储了下来

接着看

```
.subscribe(new Observer<String>() {
    @Override
    public void onSubscribe(@NonNull Disposable d) {

    }

    @Override
    public void onNext(@NonNull String s) {
        Log.d(TAG, "onNext: 我是自定义Observer，我收到被观察者的信息啦！信息的类型为-->"
                + s.getClass().getName());
    }


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

    }

    @Override
    public void onComplete() {

    }
```

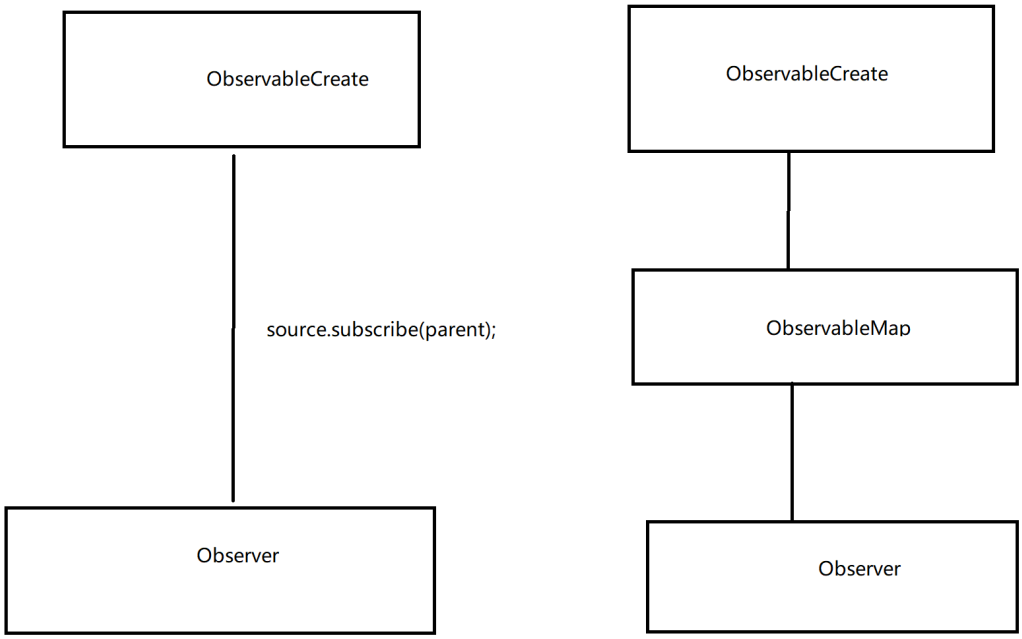
```
});
```

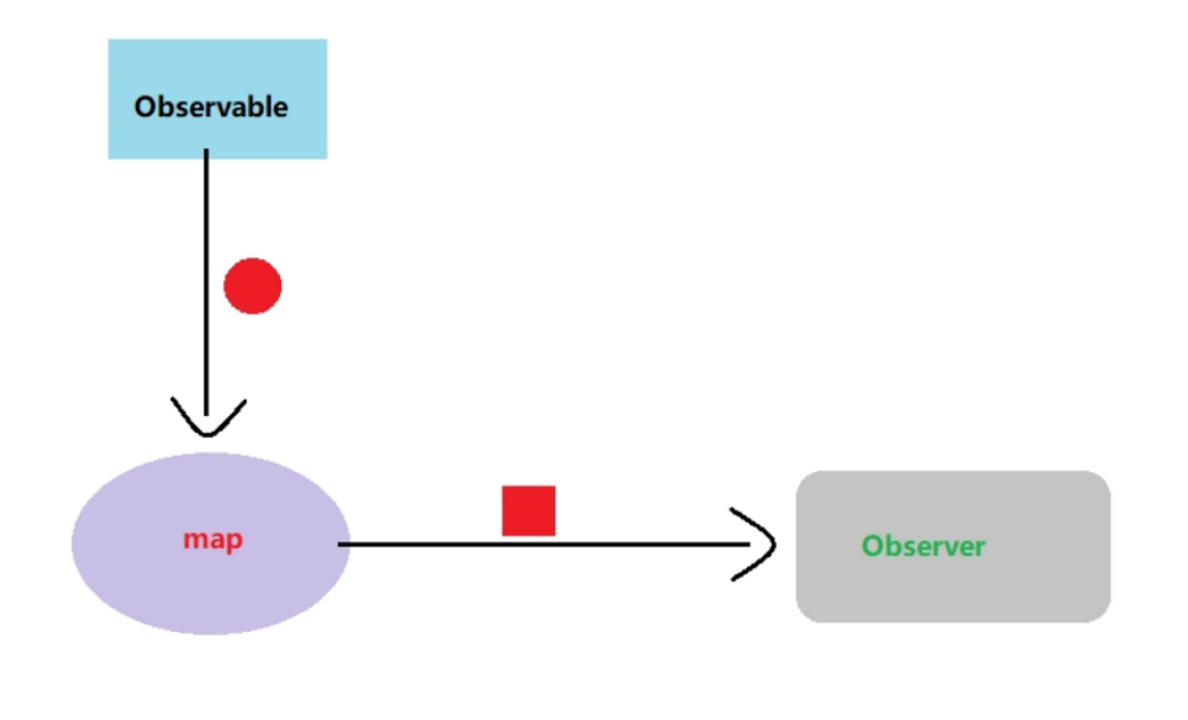
直接看到ObservableMap的subscribeActual方法

```
@Override
public void subscribeActual(Observer<? super U> t) {
    source.subscribe(new MapObserver<T, U>(t, function));
}
```

这里我们来对比下ObservableCreate的subscribeActual方法

```
@Override
protected void subscribeActual(Observer<? super T> observer) {
    CreateEmitter<T> parent = new CreateEmitter<>(observer);
    observer.onSubscribe(parent);

    try {
        source.subscribe(parent);
    } catch (Throwable ex) {
        Exceptions.throwIfFatal(ex);
        parent.onError(ex);
    }
}
```






其实之前就说过，ObservableCreate才是真正发送消息的那个类，在我们上面的demo中，直接由ObservableCreate发送消息给观察者，而在这里ObservableCreate先将消息传给ObservableMap，然后ObservableMap收到消息后可以把消息更改后再继续传给下游。

来分析源码

source.subscribe(new MapObserver<T, U>(t, function));

其实就是

```
@Override
protected void subscribeActual(Observer<? super T> observer) {
    CreateEmitter<T> parent = new CreateEmitter<>(observer);
    observer.onSubscribe(parent);

    try {
        source.subscribe(parent);
    } catch (Throwable ex) {
        Exceptions.throwIfFatal(ex);
        parent.onError(ex);
    }
}
```

在demo1中ObservableCreate subscribeActual方法传入的真正的observer也即是

```
new Observer<String>() {
                    @Override
                    public void onSubscribe(@NonNull Disposable d) {

                    }

                    @Override
                    public void onNext(@NonNull String s) {
                        Log.d(TAG, "onNext: 我是自定义Observer，我收到被观察者的信息
啦！信息的类型为-->"
                                + s.getClass().getName());
                    }
```

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

                }

                @Override
                public void onComplete() {

                }
            }
```

，而在这里我们传入的是

MapObserver，MapObserver储存了原来本来应该传给ObservableCreate subscribeActual方法的那个Observer和转换函数

当ObservableCreate的subscribeActual方法执行到source.subscribe(parent);

时，emitter.onNext(1);也即是执行observer.onNext(t);

在demo1中此时消息应该就传给我们创建的那个Observer了，但是现在消息先传给了MapObserver，

其实这里可以猜到其实就是MapObserver拦截了消息并利用存储下来的转换函数将消息改变后再发给我们创建的的那个Observer

我们来看看源码

```
@Override
        public void onNext(T t) {
            if (done) {
                return;
            }

            if (sourceMode != NONE) {
                downstream.onNext(null);
                return;
            }

            U v;

            try {
                v = Objects.requireNonNull(mapper.apply(t), "The mapper function
returned a null value.");
            } catch (Throwable ex) {
                fail(ex);
                return;
            }
            downstream.onNext(v);
        }
```

关键是这俩行

v = Objects.requireNonNull(mapper.apply(t), "The mapper function returned a null value.");

downstream.onNext(v);

## 3.线程调度的原理

如果理解了上面讲的操作符的原理，那么理解RxJava是怎么进行线程调度就比较简单了，来看

subscribeOn操作符的原理

```
Observable.create(new ObservableOnSubscribe<String>() {
    @Override
    public void subscribe(ObservableEmitter<String> e) {
        e.onNext("Hello World");

        Log.d(TAG, "subscribe" + Thread.currentThread().getName());
    }
})
        .subscribeOn(Schedulers.io())
        .subscribe(new Observer<String>() {
            @Override
            public void onSubscribe(Disposable d) {

                Disposable disposable = d;
                Log.d(TAG, "onSubscribe: " + Thread.currentThread().getName());
            }

            @Override
            public void onNext(String s) {
                Log.d(TAG, "onNext: " + Thread.currentThread().getName());
            }

            @Override
            public void onError(Throwable e) {
            }

            @Override
            public void onComplete() {
            }
        });
```

Schedulers.io()封装了线程池，看到subscribeOn

```
public final Observable<T> subscribeOn(@NonNull Scheduler scheduler) {
    Objects.requireNonNull(scheduler, "scheduler is null");
    return RxJavaPlugins.onAssembly(new ObservableSubscribeOn<>(this,
scheduler));
}
```

是不是和分析map操作符很类似?

接着看ObservableSubscribeOn

```
public ObservableSubscribeOn(ObservableSource<T> source, Scheduler scheduler) {
    super(source);
    this.scheduler = scheduler;
}
```

可以看到将那个包装了线程池的对象进行了储存

接下来看ObservableSubscribeOn的subscribeActual

```
@Override
public void subscribeActual(final Observer<? super T> observer) {
    final SubscribeOnObserver<T> parent = new SubscribeOnObserver<>(observer);

    observer.onSubscribe(parent);

    parent.setDisposable(scheduler.scheduleDirect(new SubscribeTask(parent)));
}
```

看到new SubscribeTask(parent)

```
final class SubscribeTask implements Runnable {
    private final SubscribeOnObserver<T> parent;

    SubscribeTask(SubscribeOnObserver<T> parent) {
        this.parent = parent;
    }

    @Override
    public void run() {
        source.subscribe(parent);
    }
}
```

将观察者包装进了一个Runnable对象里

scheduler.scheduleDirect(new SubscribeTask(parent))将这个任务放进了线程池中执行。

看看scheduler.scheduleDirect

```
@NonNull
public Disposable scheduleDirect(@NonNull Runnable run) {
    return scheduleDirect(run, 0L, TimeUnit.NANOSECONDS);
}

@NonNull
    public Disposable scheduleDirect(@NonNull Runnable run, long delay, @NonNull
TimeUnit unit) {
        final Worker w = createWorker();

        final Runnable decoratedRun = RxJavaPlugins.onSchedule(run);

        DisposeTask task = new DisposeTask(decoratedRun, w);

        w.schedule(task, delay, unit);

        return task;
    }

@NonNull
        @Override
        public Disposable schedule(@NonNull Runnable action, long delayTime,
@NonNull TimeUnit unit) {
            if (tasks.isDisposed()) {
```

```
                // don't schedule, we are unsubscribed
                return EmptyDisposable.INSTANCE;
            }

            return threadWorker.scheduleActual(action, delayTime, unit, tasks);
        }

@NonNull
    public ScheduledRunnable scheduleActual(final Runnable run, long delayTime,
@NonNull TimeUnit unit, @Nullable DisposableContainer parent) {
        Runnable decoratedRun = RxJavaPlugins.onSchedule(run);

        ScheduledRunnable sr = new ScheduledRunnable(decoratedRun, parent);

        if (parent != null) {
            if (!parent.add(sr)) {
                return sr;
            }
        }

        Future<?> f;
        try {
            if (delayTime <= 0) {
                f = executor.submit((Callable<Object>)sr);
            } else {
                f = executor.schedule((Callable<Object>)sr, delayTime, unit);
            }
            sr.setFuture(f);
        } catch (RejectedExecutionException ex) {
            if (parent != null) {
                parent.remove(sr);
            }
            RxJavaPlugins.onError(ex);
        }

        return sr;
    }
```

接着看ObserverOn的源码

为了方便讲解将RxJava运行在了子线程中

```
new Thread() {
    @Override
    public void run() {
        super.run();

        ThreadDemo();
    }
}.start();

    void ThreadDemo() {
        Observable.create(new ObservableOnSubscribe<String>() {
            @Override
            public void subscribe(ObservableEmitter<String> e) {
                e.onNext("qwerty");
```

```
                Log.d(TAG, "subscribe " + Thread.currentThread().getName());
            }
        })
                .observeOn(AndroidSchedulers.mainThread())
                .subscribe(new Observer<String>() {
                    @Override
                    public void onSubscribe(Disposable d) {
                        Log.d(TAG, "onSubscribe: " +
Thread.currentThread().getName());
                    }

                    @Override
                    public void onNext(String s) {
                        Log.d(TAG, "onNext: " +
Thread.currentThread().getName());
                    }

                    @Override
                    public void onError(Throwable e) {

                    }

                    @Override
                    public void onComplete() {

                    }
                });
    }
```

AndroidSchedulers.mainThread()和Schedulers.io()类似，Schedulers.io()里面封装了线程池以让任务可以执行在子线程中，而AndroidSchedulers.mainThread()封装了handler以便任务可以执行在主线程中

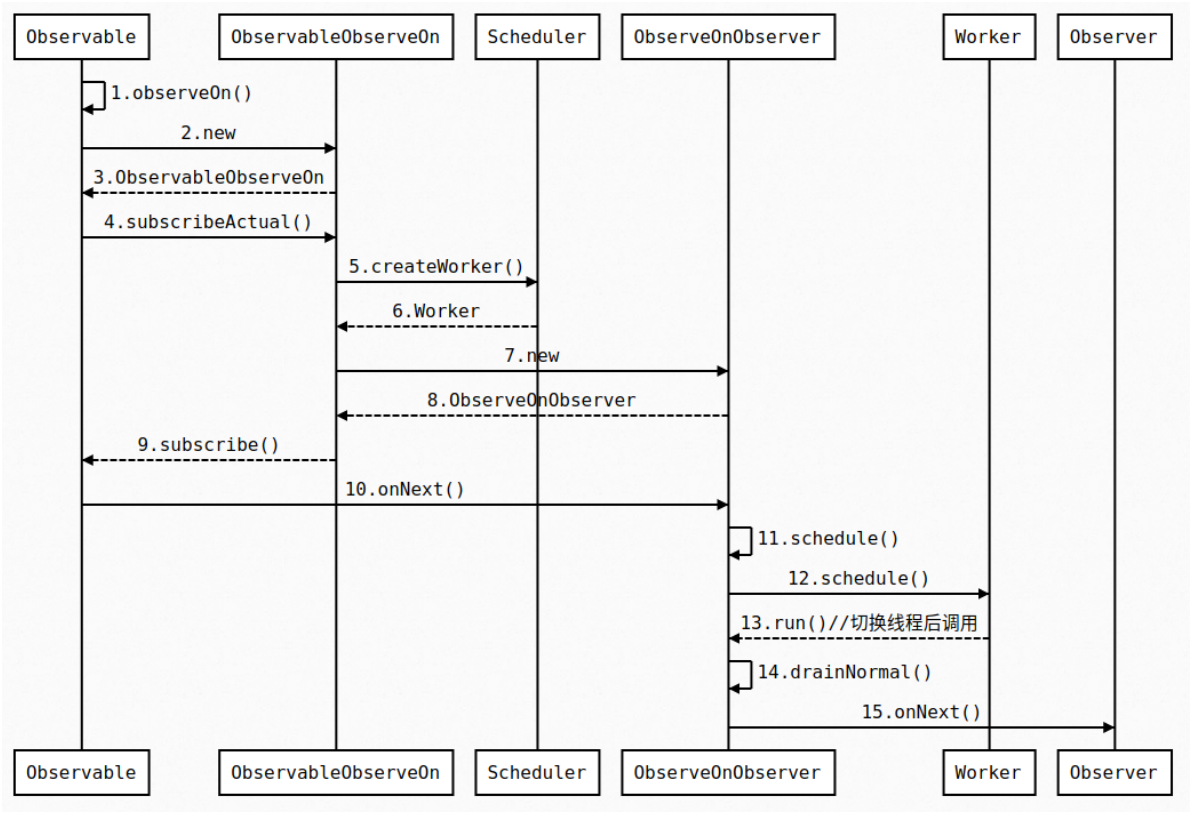
分析observeOn

```
public final Observable<T> observeOn(@NonNull Scheduler scheduler) {
    return observeOn(scheduler, false, bufferSize());
}

public final Observable<T> observeOn(@NonNull Scheduler scheduler, boolean
delayError, int bufferSize) {
        Objects.requireNonNull(scheduler, "scheduler is null");
        ObjectHelper.verifyPositive(bufferSize, "bufferSize");
        return RxJavaPlugins.onAssembly(new ObservableObserveOn<>(this,
scheduler, delayError, bufferSize));
    }

public ObservableObserveOn(ObservableSource<T> source, Scheduler scheduler,
boolean delayError, int bufferSize) {
        super(source);
        this.scheduler = scheduler;
        this.delayError = delayError;
        this.bufferSize = bufferSize;
    }
```

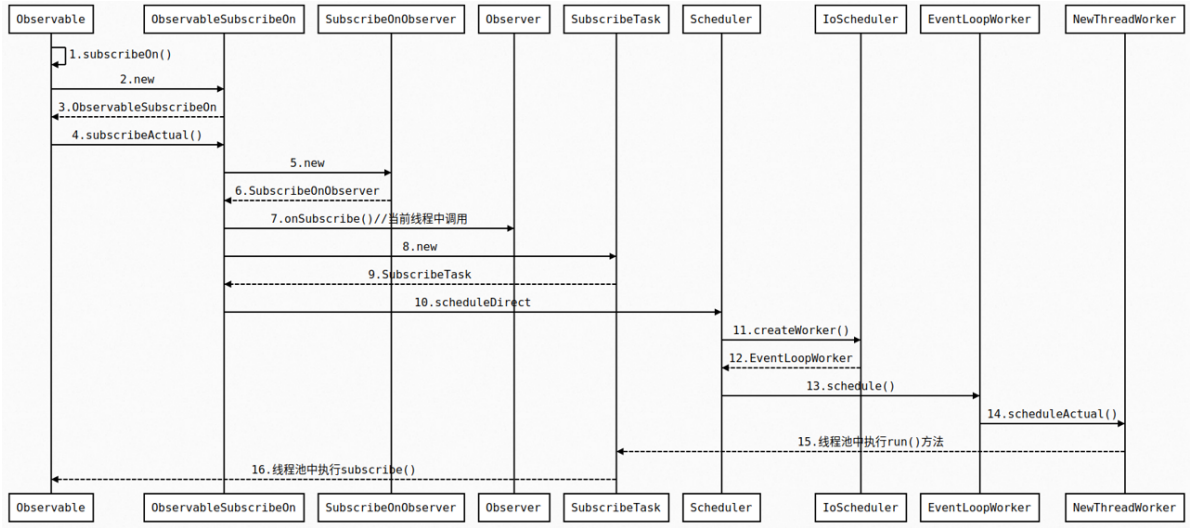
可知将scheduler进行了存储

有了map操作符和SubscribeOn操作符的基础，分析ObserverOn原理其实是一样的。

observeOn()切换线程时序图：




subscribeOn()切换线程时序图：




对observeOn()与subscribeOn()的使用做一个总结：

1.只有第一个subscribeOn() 起作用（所以多个 subscribeOn() 无意义）；
2.这个 subscribeOn() 控制从流程开始的第一个操作，直到遇到第一个 observeOn()；
3.observeOn() 可以使用多次，每个 observeOn() 将导致一次线程切换()，这次切换开始于这次 observeOn() 的下一个操作；
4.不论是 subscribeOn() 还是 observeOn()，每次线程切换如果不受到下一个 observeOn() 的干预，线程将不再改变，不会自动切换到其他线程。

# RxJava核心思想

用RxJava写出来的代码像一条有始有终的河流，我们在起点放入事件，然后事件从起点流向终点的过程中我们可以不断对事件进行拦截并更改，最终到终点时是我们想要的事件。

什么意思?

比如说我们要下载一张图片并展示，起点事件就是图片String类型的URL，然后我们可以拦截这个事件，然后将URL转变为Bitmap再将事件流向下游，这样事件就发生了更改，我们也可以再次拦截事件然后打一个log然后再次将事件传向下游，这样到终点时事件就是bitmap信息而不是String了，我们此时可以拿得到的bitmap信息做想做的事情，比如赋值给ImageView

```
public void rxJavaDownloadImageAction(View view) {

        // 起点
        Observable.just(PATH)
                .map(new Function<String, Bitmap>() {
                    @Override
                    public Bitmap apply(String s) throws Exception {
                        URL url = new URL(PATH);
                        HttpURLConnection httpURLConnection =
(HttpURLConnection) url.openConnection();
                        httpURLConnection.setConnectTimeout(5000);
                        int responseCode = httpURLConnection.getResponseCode();
// 才开始 request
                        if (responseCode == HttpURLConnection.HTTP_OK) {
                            InputStream inputStream =
httpURLConnection.getInputStream();
                            Bitmap bitmap =
BitmapFactory.decodeStream(inputStream);
                            return bitmap;
                        }
                        return null;
                    }
                })
                // 日志记录
                .map(new Function<Bitmap, Bitmap>() {
                    @Override
                    public Bitmap apply(Bitmap bitmap) {
                        Log.d(TAG, "apply: 是这个时候下载了图片啊:" +
System.currentTimeMillis());
                        return bitmap;
                    }
                })
                .map(new Function<Bitmap, Bitmap>() {
                    @Override
                    public Bitmap apply(Bitmap bitmap) {
                        Paint paint = new Paint();
                        paint.setTextSize(88);
                        paint.setColor(Color.RED);
                        return drawTextToBitmap(bitmap, "拦截图片并加水印", paint,
88, 88);
                    }
                })
                .subscribeOn(Schedulers.io())
                .observeOn(AndroidSchedulers.mainThread())
                .subscribe(new Observer<Bitmap>() {
                    @Override
                    public void onSubscribe(Disposable d) {
```

```
                        progressDialog = new
ProgressDialog(DownloadActivity.this);
                        progressDialog.setTitle("download run");
                        progressDialog.show();
                    }

                    @Override
                    public void onNext(Bitmap bitmap) {
                        image.setImageBitmap(bitmap);
                    }

                    @Override
                    public void onError(Throwable e) {

                    }

                    @Override
                    public void onComplete() {
                        if (progressDialog != null)
                            progressDialog.dismiss();
                    }
                });
    }
```

这样子写出来的代码思维也像河流一样不会中断，写出来的代码更好维护。

来看看传统方式实现上面的代码：

```
public void downloadImageAction(View view) {
    progressDialog = new ProgressDialog(this);
    progressDialog.setTitle("下载图片中...");
    progressDialog.show();

    new Thread(new Runnable() {
        @Override
        public void run() {
            try {
                URL url = new URL(PATH);
                HttpURLConnection httpURLConnection = (HttpURLConnection)
url.openConnection();
                httpURLConnection.setConnectTimeout(5000);
                int responseCode = httpURLConnection.getResponseCode();
                if (responseCode == HttpURLConnection.HTTP_OK) {
                    InputStream inputStream =
httpURLConnection.getInputStream();
                    Bitmap bitmap = BitmapFactory.decodeStream(inputStream);
                    Message message = handler.obtainMessage();
                    message.obj = bitmap;
                    handler.sendMessage(message);
                }
            } catch (Exception e) {
                e.printStackTrace();
            }
        }
    }).start();
}
```

```
private final Handler handler = new Handler(new Handler.Callback() {

    @Override
    public boolean handleMessage(@NonNull Message msg) {
        Bitmap bitmap = (Bitmap) msg.obj;
        image.setImageBitmap(bitmap);

        if (progressDialog != null) progressDialog.dismiss();
        return false;
    }
});
```

相对于RxJava写出来的流式的代码，无疑这些代码“丑陋了很多”。

# 自定义RxJava操作符

仿照其它操作符并借助throttleFirst操作符完成View的防抖操作符

```
public static <T> Observable<T> create(@NonNull ObservableOnSubscribe<T> source)
{
    Objects.requireNonNull(source, "source is null");
    return RxJavaPlugins.onAssembly(new ObservableCreate<>(source));
}

-->
    public class RxView {

    public static Observable<Object> clicks(View view) {
        Objects.requireNonNull(view, "source is null");
        return RxJavaPlugins.onAssembly(new RxViewObservable(view));
    }
}
```

```
public final class ObservableCreate<T> extends Observable<T> {
    final ObservableOnSubscribe<T> source;

    public ObservableCreate(ObservableOnSubscribe<T> source) {
        this.source = source;
    }

    @Override
    protected void subscribeActual(Observer<? super T> observer) {
        CreateEmitter<T> parent = new CreateEmitter<>(observer);
        observer.onSubscribe(parent);

        try {
            source.subscribe(parent);
        } catch (Throwable ex) {
            Exceptions.throwIfFatal(ex);
            parent.onError(ex);
        }
    }
```

```
    ...
}

-->

public class RxViewObservable extends Observable<Object> {
    View view;
    // 用来给onNext传参的,实际无用
    private static final Object EVENT = new Object();

    public RxViewObservable(View view) {
        this.view = view;
    }

    @Override
    protected void subscribeActual(@NonNull Observer<? super Object> observer) {
        view.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                observer.onNext(EVENT);
            }
        });
    }
}
```

上面那个操作符没有考虑流的中断，有余力的可以看下完整实现代码：

```
//可以用disposable来中断流
public class RxViewObservableX extends Observable<Object> {
    View view;
    // 用来给onNext传参的,实际无用
    private static final Object EVENT = new Object();

    public RxViewObservableX(View view) {
        this.view = view;
    }

    @Override
    protected void subscribeActual(@NonNull Observer<? super Object> observer) {
        MyListener myListener = new MyListener(view, observer);
        observer.onSubscribe(myListener);
        view.setOnClickListener(myListener);
    }

    static final class MyListener implements View.OnClickListener, Disposable {

        private final View view;
        private Observer<Object> observer;
        //线程安全
        private final AtomicBoolean isDisposable = new AtomicBoolean();

        public MyListener(View view, Observer<Object> observer) {
            this.view = view;
            this.observer = observer;
        }
```

```
        @Override
        public void onClick(View v) {
            //流没有被中断才能继续传递
            if (isDisposed() == false) {
                observer.onNext(EVENT);
            }
        }

        // 如果调用了中断
        @Override
        public void dispose() {
            // 如果没有中断过,才能取消view.setOnClickListener(null);
            if (isDisposable.compareAndSet(false, true)) {
                if (Looper.myLooper() == Looper.getMainLooper()) {
                    view.setOnClickListener(null);

                } else {
                    AndroidSchedulers.mainThread().scheduleDirect(new Runnable()
{
                        @Override
                        public void run() {
                            view.setOnClickListener(null);
                        }
                    });
                }
            }
        }

        @Override
        public boolean isDisposed() {
            return isDisposable.get();
        }
    }
}
```

# RxHook

前面的东西都不太好理解，最后讲个好玩的

还记得之前讲的RxJavaPlugins.onAssembly这个钩子方法吗?

上面我们只是简单说了下这个东西，我们来仔细看看这个东西能用了干嘛

```
public static <T> Observable<T> onAssembly(@NonNull Observable<T> source) {
    Function<? super Observable, ? extends Observable> f = onObservableAssembly;
    if (f != null) {
        return apply(f, source);
    }
    return source;
}
```

之前我们说我们没有设置Function<? super Observable, ? extends Observable> f的话f为null所以这个方法就会什么也没干，传入什么就返回了什么，现在我们来看看怎么设置

一番探寻后可知是

RxJavaPlugins.setOnObservableAssembly

这个方法

```
public static void setOnObservableAssembly(@Nullable Function<? super
Observable, ? extends Observable> onObservableAssembly) {
    if (lockdown) {
        throw new IllegalStateException("Plugins can't be changed anymore");
    }
    RxJavaPlugins.onObservableAssembly = onObservableAssembly;
}
```

假如f不会null的话就会走return apply(f, source);而不是return source; 来看看apply(f, source)

```
static <T, R> R apply(@NonNull Function<T, R> f, @NonNull T t) {
    try {
        return f.apply(t);
    } catch (Throwable ex) {
        throw ExceptionHelper.wrapOrThrow(ex);
    }
}
```

依据这个可以编写以下代码全局监听RxJava

```
RxJavaPlugins.setOnObservableAssembly(new Function<Observable, Observable>() {
    @Override
    public Observable apply(Observable observable) {
        Log.d(TAG, "apply: 整个项目 全局 监听 到底有多少地方使用 RxJava:" +
observable);
        return observable; // 不破坏人家的功能
    }
});
```