RxJava 2.x使用详解

版本号:2019-03-15(18:00)

@[toc]

基本概念

1、什么是响应式编程?

> 1. 响应式编程是一种基于异步数据流概念的编程模式。
> 2. 只要接收到事件，就进行响应并执行一定操作。

2、什么是函数式编程?

新特性

3、RxJava 2.x支持新特性，依赖于四个基础接口

> 1. Publisher
> 2. Subscriber
> 3. Subscription
> 4. Processor

4、Publisher的作用

> 1. Publisher 可以发出一系列的事件

5、Subscriber的作用

> 1. Subscriber 负责和处理这些事件。

6、Publisher的Flowable：支持背压

> 背压是指在异步场景中，被观察者发送事件速度远快于观察者的处理事件速度的情况下，告诉被观察者降低发送速度的策略。

两种观察者模式

7、RxJava 2.x有两种观察者模式

> 1. Observable/Observer
> 2. Flowable/Subscriber

8、Observable/Observer不支持背压

9、Flowable/Subscriber支持背压

Observable

doOnNext

1、doOnNext是用于在订阅者接收到数据前进行一些操作

> 在下游的onNext()回调前调用

```
Observable.just(1, 2, 3, 4)
.doOnNext(new Consumer<Integer>() {
    @Override
    public void accept(@NonNull Integer integer) throws Exception {
        // 1、发送给观察者前进行额外操作：保存数据等等。
    }
}).subscribe(new Consumer<Integer>() {
    @Override
    public void accept(@NonNull Integer integer) throws Exception {

        // 2、观察者接收到数据
    }
});
```

ConnectableObservable

1、ConnectableObservable是什么?

> 1. 热被观察者
> 2. 能够持续的发出数据，不管有没有订阅者。

publish

2、如何创建一个ConnectableObservable?

> 1. 需要通过 普通`Observable` 进行 `publish` 转换为 `ConnectableObservable`

```
Observable.just(1)
        .publish();
```

connect

3、ConnectableObservable何时开始发出数据?

> 1. 和 `Cold Observable` 不同的地方在于，需要调用 `connect` 开始走 `subscribe()`内的方法
> 2. 会持续不断地发出数据。

```
mConnectableObservable.connect();
```

Observer

Consumer

1、简化的订阅方式

```
Observable.just("hello").subscribe(new Consumer<String>()
{
    @Override
    public void accept(String s) throws Exception
    {

    }
});
```

SingleObserver

2、SingleObserver的作用

> 1、SingleObserver只会调用 `onError()` 和 `onSuccess()`

```
Single.just(new Random().nextInt())
      .subscribe(new SingleObserver<Integer>() {
            @Override
            public void onSubscribe(@NonNull Disposable d) {

            }

            @Override
            public void onSuccess(@NonNull Integer integer) {
                // 成功接收到
            }

            @Override
            public void onError(@NonNull Throwable e) {
                // 出现异常
            }
      });
```

线程调度

subscribeOn

1、subscribeOn用于指定发射事件的线程

> 1. 指定 订阅时-subscribe() 在哪个线程中调用

```
Observable.just("hello")
    //在一个新线程中调用
        .subscribeOn(Schedulers.newThread())
        .subscribe(new Consumer<String>()
        {
            @Override
            public void accept(String s) throws Exception
            {
                Log.e("feather", s);
            }
        });
```

2、多次调用subscribeOn只有第一次有效

observeOn

2、observeOn用于指定订阅者接收事件的线程

> 1. 指定 观察者处理事件 是在哪个线程中---这就是处理事件的回调方法在哪个线程调用。

3、多次调用observeOn是可以的，每调用一次，下游的线程就会切换一次。

线程切换

```
Observable.create(new ObservableOnSubscribe<Integer>() {
    // 1、在IO线程中订阅
    public void subscribe(@NonNull ObservableEmitter<Integer> e) throws Exception {
        Log.e(TAG, "Observable thread is : " + Thread.currentThread().getName());
        e.onNext(1);
        e.onComplete();
    }
}).subscribeOn(Schedulers.newThread())
    // 1、在IO线程中订阅
        .subscribeOn(Schedulers.io())
          // 2、在主线程中处理
        .observeOn(AndroidSchedulers.mainThread())
        .doOnNext(new Consumer<Integer>() {
            // 2、在主线程中处理
            public void accept(@NonNull Integer integer) throws Exception {
                Log.e(TAG, "After observeOn(mainThread)，Current thread is " + Thread.currentTh
            }
        })
          // 3、在IO线程中处理
        .observeOn(Schedulers.io())
        .subscribe(new Consumer<Integer>() {
            // 3、在IO线程中处理
            public void accept(@NonNull Integer integer) throws Exception {
                Log.e(TAG, "After observeOn(io)，Current thread is " + Thread.currentThread().g
            }
        });
```

AndroidSchedulers.mainThread()

1、AndroidSchedulers.mainThread()就是安卓的主线程

Scheduler

1、Scheduler是什么?

> 1. 调度器
> 2. 用于RxJava进行线程切换

2、Schedulers的作用

> 用于生成用户所需要的 `Scheduler`

3、Schedulers的方法有哪些?

方法	作用
io()	子线程，使用线程池。io操作如网络请求、读写文件等。

方法	作用
newThred()	子线程，新创建一个线程
computation()	用于需要大量计算任务，不能用于IO操作，默认线程数 = 处理器的数量
from(executor)	使用指定的Executor作为调度器, 自定义线程池
single()	RxJava有一个单例线程，所有任务都在该线程中执行，后续任务按顺序排队
trampoline()	当前线程立即执行任务。当前线程正在执行的任务，会暂停， 在插入的任务执行完毕后，继续执行。
shutdown()/start()	所有调度器终止和开始

4、Schedulers.newThread() vs Schedulers.io()

> 1. newThread()会为每个实例创建一个新线程，而 io() 使用的是线程池来管理
> 2. 大量并发工作采用 `Schedulers.io()` ，可能会遇到IO线程，比如打开文件的最大数量，tcp连接的最大数量，也可能会用完RAM

Subject

AsyncSubject

1、AsyncSubject的作用

> 1. `Observer` 会接收到 `AsyncSubject.onComplete()` 之前的最后一个数据。
> 2. 下例中发送了 `1、2、3、onComplete、4` 只接收到 `3`

```
Subject<String> subject = AsyncSubject.create();
        subject.onNext("1");
        subject.onNext("2");
        subject.onNext("3");
        subject.onComplete();
        subject.onNext("4");

        subject.subscribe(new Observer<String>()
        {
            @Override
            public void onSubscribe(@NonNull Disposable disposable)
            {
                Log.d("feather", "onSubscribe");
            }

            @Override
            public void onNext(@NonNull String s)
            {
                Log.d("feather", "onNext: " + s);
            }

            @Override
            public void onError(@NonNull Throwable throwable)
            {
                Log.d("feather", "onError");
            }

            @Override
            public void onComplete()
            {
                Log.d("feather", "onComplete");
            }
        });
```

> 输出结果

```
onSubscribe
onNext: 3
onComplete
```

BehaviorSubject

1、BehaviorSubject的作用

> 1. 接收 订阅前的最后一个数据 ， 并且 继续接收后续的数据

```
Subject<String> subject = BehaviorSubject.create();
        subject.onNext("1");
        subject.onNext("2");

        subject.subscribe(new Observer<String>()
        {
            @Override
            public void onSubscribe(@NonNull Disposable disposable)
            {
                Log.d("feather", "onSubscribe");
            }

            @Override
            public void onNext(@NonNull String s)
            {
                Log.d("feather", "onNext: " + s);
            }

            @Override
            public void onError(@NonNull Throwable throwable)
            {
                Log.d("feather", "onError");
            }

            @Override
            public void onComplete()
            {
                Log.d("feather", "onComplete");
            }
        });

        subject.onNext("3");
        subject.onNext("4");
```

> 结果

```
onSubscribe
onNext: 2
onNext: 3
onNext: 4
```

PublishSubject

1、PublishSubject的作用

> 1. 只接收订阅之后发送的数据
> 2. 下例中只接收到 3、4

```
Subject<String> subject = PublishSubject.create();
subject.onNext("1");
subject.onNext("2");

subject.subscribe(new Observer<String>(){...});

subject.onNext("3");
subject.onNext("4");
```

ReplaySubject

1、ReplaySubject的作用和使用

> 1. 发射一切数据
> 2. 四种创建方式:
>   1. `ReplaySubject.create()` 创建默认初始缓存容量大小为16，当数据条目超过16会重新分配内存空间。
>   2. `ReplaySubject.create(100)` 创建指定初始缓存容量的ReplaySubject
>   3. `ReplaySubject.createWithSize(2)` 只缓存订阅前最后发送的2条数据
>   4. `ReplaySubject.createWithTime(xxx,TimeUnit.SECONDS,Schedulers.computation())` 被订阅前的前 xxx 秒内发送的数据才能被接收

```
Subject<String> subject = ReplaySubject.create();
subject.onNext("1");
subject.onNext("2");

subject.subscribe(new Observer<String>(){...});

subject.onNext("3");
subject.onNext("4");
```

数据的转发

1、Subject还可以进行数据的转发，作为中间桥梁，将A的数据转发给C

```
// A
Observable<String> observable = Observable.fromArray("123","456","789");
// B
ReplaySubject<String> replaySubject = ReplaySubject.create();

// A发数据给B
observable.subscribe(replaySubject);
// B转发数据给A
replaySubject.subscribe(new SubjectObserver<>("C"));
```

Processor

1、Processor和Subject使用方法类型，区别在于 `Processor`支持背压

PublishProcessor

AsyncProcessor

BehaviorProcessor

ReplayProcessor

Flowable

背压

1、背压是什么(Backpressure)?

> 1. RxJava中通过操作符调用链，数据从上游向下游传递
> 2. 当上游发送数据的速度大于下游处理数据的速度时，会抛出MissingBackpressureException异常。
> 3. 这种情况就需要 背压进行流量控制

Disposable

1、Disposable的作用

> 1. `dispose()` 进行反订阅

CompositeDisposable

2、CompositeDisposable的作用?

> 1. 对下游进行管理，在Activity销毁时，释放所有的异步工作
> 2. 避免内存泄露
> 3.

操作符

just

1、just的作用就是简单发射器，并依次调用 `onNext()` 方法

Map

1、Map的作用?

> 1. 将一个 Observable对象 转换为 另一个Observable对象

2、可以使用Map的场景

> 1- 传入本地图片路径 ， 根据路径获取到 图片的Bitmap

```
Observable.just(filepath)
.map(new Function<String, Bitmap>()
{
    @Override
    public Bitmap apply(@NonNull String path) throws Exception
    {
        // 1、path转换为Bitmap
        Bitmap bitmap = getBitmapByPath(path);
        return bitmap;
    }
}).subscribe(new Consumer<Bitmap>()
{
    @Override
    public void accept(Bitmap bitmap) throws Exception
    {
        // 2、获取到Bitmap进行展示
    }
});
```

3、利用map进行网络数据解析

```
Observable.create(new ObservableOnSubscribe<String>()
{
    @Override
    public void subscribe(@NonNull ObservableEmitter<String> observableEmitter) throws Exceptic
    {
        // 1、发起网络请求
        // Request
        // 2、获取返回的数据-JSon数据
        String json = xxx;
        // 3、将数据发送出去
        observableEmitter.onNext(json);
    }
}).map(new Function<String, UserDetailBean>()
{
    @Override
    public UserDetailBean apply(@NonNull String json) throws Exception
    {
        // 4、将网络数据转换为需要的实体
        return new Gson().fromJson(json, UserDetailBean.class);
    }
}).observeOn(AndroidSchedulers.mainThread()) // 5、在主线程处进行临时保存
.doOnNext(new Consumer<UserDetailBean>()
{
    @Override
    public void accept(UserDetailBean bean) throws Exception
    {
        // 5、主线程保存数据
    }
}).subscribeOn(Schedulers.io()) // 6、在io线程池中发送事件，处理数据的请求，返回和处理
.observeOn(AndroidSchedulers.mainThread()) // 7、主线程处理返回的结果
.subscribe(new Consumer<UserDetailBean>()
{
    // 7、成功接收到数据
    @Override
    public void accept(UserDetailBean bean) throws Exception
    {

    }
}, new Consumer<Throwable>()
{
    // 8、接收过程中出现了异常
    @Override
    public void accept(Throwable throwable) throws Exception
    {

    }
});
```

FlatMap

1、FlatMap是什么？和Map有什么区别？

1. 可以 将一个Observable对象转换为多个Observable对象
2. FlatMap 不会保证 发送事件的顺序

2、FlatMap获取到一个Student列表，差分为各个Student，依次打印其详细信息。

```
// 1-一个Observable，查询到多个学生的信息列表
Observable.create(new ObservableOnSubscribe<Student>()
{
    @Override
    public void subscribe(@NonNull ObservableEmitter<Student> observableEmitter) throws Excepti
    {

        // 1、网络请求到一个学生的列表
        ArrayList<Student> students = studentNames;
        // 2、遍历发出去
        for (Student student : students)
        {
            observableEmitter.onNext(student);
        }
    }
    // 2-拆分为多个Student的Observable，发出去，每个接受到后进行相应的处理
}).flatMap(new Function<Student, ObservableSource<String>>()
{
    @Override
    public ObservableSource<String> apply(@NonNull Student student) throws Exception
    {
        // 3、每个Student，都额外调用接口去查询数据
        return Observable.just(student.name);
    }
})
.subscribe(new Consumer<String>()
{
    @Override
    public void accept(String s) throws Exception
    {
        Log.d("feather", s);
    }
});
```

concatMap

1、concatMap属于有顺序的flatMap

1. concatMap使用方法和flatMap完全一致
2. 数据的发送符合顺序

switchMap

2、switchMap的作用?

1. 将一个 Observable对象 转换为 多个Observable对象
2. 每当 源Observable 发射一个 新的数据项（Observable）时 ， 它
   将 取消订阅并停止监视``之前的数据项产生的Observable ， 并开始监视当前发射的这一个。

3、switchMap的使用场景

1. 用户在搜索关键字时，迅速输入 old 和 new 来进行搜索。
2. 结果最后搜索的 new关键字的结果返回快 ， 后返回 old关键字的结果
3. 此时 关键字明明是new，且显示的是查询old关键字的内容

concat

1、concat的作用

1. 可以做到 多个Observable的订阅事件按顺序前后发生
2. 例如 ObservableA终止后(onComplete)，才会去 订阅第二个Observable

2、利用concat操作符先读取缓存再通过网络请求新数据

1. 对于 操作不敏感的数据时 ， 可以先读取缓存，再通过网络获取
2. 技巧在于 是否调用onComplete() 方法。

```
Observable<UserDetailBean> cache = Observable.create(new ObservableOnSubscribe<UserDetailBean>(
{
    @Override
    public void subscribe(@NonNull ObservableEmitter<UserDetailBean> observableEmitter) throws
    {
        // 1、读取缓存数据
        UserDetailBean cache = getCache();

        if(cache != null){
            // 2、具有缓存数据，直接返回
            observableEmitter.onNext(cache);
        }else{
            // 3、不具有返回数据，通过网络请求
            observableEmitter.onComplete();
        }

    }
});

Observable<UserDetailBean> network = Observable.create(new ObservableOnSubscribe<UserDetailBear
{
    @Override
    public void subscribe(@NonNull ObservableEmitter<UserDetailBean> observableEmitter) throws
    {
        // 1、读取网络数据
        UserDetailBean cache = fromInternet();

        if(cache != null){
            // 2、获取到数据，直接返回
            observableEmitter.onNext(cache);
        }else{
            // 3、不具有数据，出现网络错误。
            observableEmitter.onError(new NetworkErrorException());
        }
    }
});

// 1、合并缓存和网络请求
Observable.concat(cache, network)
        .subscribeOn(Schedulers.io()) // 2、IO读取缓存和网络数据
        .observeOn(AndroidSchedulers.mainThread()) // 3、主线程处理返回的结果
        .subscribe(new Consumer<UserDetailBean>()
        {
            // 4、获取到数据
            @Override
            public void accept(UserDetailBean bean) throws Exception
            {

            }
        }, new Consumer<Throwable>()
        {
            // 5、出现网络异常
            @Override
            public void accept(Throwable throwable) throws Exception
```

```
        {

        }
    });
```

concatEager

1、concatEager的作用

> 1. 多个Observable可以同时开始发射数据
> 2. 如果后一个Observable发射完成后，前一个Observable还有发射完数据，那么它会将后一个Observable的数据先缓存起来，等到前一个Observable发射完毕后，才将缓存的数据发射出去。

Zip

1、Zip的作用和使用场景

> 1. Zip用于 将多个`Observable`结合成一个数据发送出去
> 2. 适用于 将多个接口的数据共同返回后一起发送出去

```
Observable<String> observable1 = Observable.create(new ObservableOnSubscribe<String>()
{
    @Override
    public void subscribe(@NonNull ObservableEmitter<String> observableEmitter) throws Exceptic
    {
        Log.d("feather", "第一个接口正在请求数据");
        Thread.sleep(5000);
        observableEmitter.onNext("1");
        Log.d("feather", "第一个接口请求到数据");
    }
});

Observable<String> observable2 = Observable.create(new ObservableOnSubscribe<String>()
{
    @Override
    public void subscribe(@NonNull ObservableEmitter<String> observableEmitter) throws Exceptic
    {
        Log.d("feather", "第二个接口正在请求数据");
        observableEmitter.onNext("2");
        Log.d("feather", "第二个接口请求到数据");
    }
});

Observable.zip(observable1, observable2, new BiFunction<String, String, String>()
{
    @Override
    public String apply(@NonNull String s, @NonNull String s2) throws Exception
    {
        // 一起返回
        Log.d("feather", "两个接口的数据都获取到");
        return s + " " + s2;
    }
}).subscribeOn(Schedulers.io())
.subscribe(new Consumer<String>()
{
    @Override
    public void accept(String result) throws Exception
    {
        Log.d("feather", result);
    }
});
```

combineLatest

1、combineLatest的作用

> 1. 接收 多个Observable
> 2. 当任意一个Observable发射数据之后，会去取其它Observable最近一次发射的数据
> 3. 但是必须是 所有Observable至少发射过一次数据 ， 否则不进行回调处理
> 4. 例如: 下面SubjectA和SubjectB都发射过数据时，才会回调 apply() 并调用 观察者的onNext()进行后续处理

```
Observable.combineLatest(subjectA,subjectB,
        new BiFunction<String, String, Boolean>() {
            @Override
            public Boolean apply(String account, String password) throws Exception {
                // 账号需要长度大于5
                // 密码需要长度大于8
                return account.length() > 5 && password.length() > 8;
            }
 });
```

zip和combineLatest的区别

3、zip和combineLatest的区别?

> 1. zip:
>   1. 在一个Observable发射数据后，去组合所有Observable 最早一个未被组合的数据项
>   2. 也就是，组合后所发射的 第n个数据项 ， 必然是
>   由 每个Observable各自所发射的第n个数据项 所组成
> 2. combineLatest:
>   1. 在一个Observable发射数据后，组合所有Observable 最后一个发射的数据项
>   2. 前提是所有Observable都至少发射过一个数据

数据操作

skip

1、skip的作用

> 1. 跳过count数目的数据才开始接收

buffer

1、buffer的作用

> 1. 对数据进行 缓存 ， 可以设置缓存大小。 缓存满之后以List的形式发送出去

```
final int count = 0;

// 1、每两个数据打包成一组，最后多余出来的数据也会发送出来。
Observable.just(1, 2, 3, 4, 5, 6, 7).buffer(2).subscribe(new Consumer<List<Integer>>()
{
    // 2、分组为 (1, 2)(3, 4)(5, 6)(7)
    @Override
    public void accept(List<Integer> integers) throws Exception
    {
        Log.d("feather", count + " : " + integers.size());
    }
});
```

2、利用buffer实现网络数据获取后，对列表中数据进行预处理，然后打包成List发送出去。

buffer(count, skip)

3、buffer具有两个参数count和skip

> 1. 作用是将 `Observable` 中的数据按 `skip(步长)` 分成最大不超过 `count` 的buffer，然后生成一个 `Observable`
> 2. 实例: 当前有数据 `1、2、3、4、5、6`
> 3. `buffer(3, 1)`

```
1,2,3
2,3,4
3,4,5
4,5,6
```

> 4. `buffer(3, 2)`

```
1,2,3
3,4,5
5,6
```

> 5. `buffer(3, 3)`

```
1,2,3
4,5,6
```

> 5. `buffer(3, 4)`

```
1,2,3
5,6
```

take

2、take的作用是什么?

> 1. 发射前n项数据
> 2. 下例中 会只保留前3个数据 ， 然后走 `buffer`的流程

```
final int count = 0;

// 1、每两个数据打包成一组，最后多余出来的数据也会发送出来。
Observable.just(1, 2, 3, 4, 5, 6, 7).take(3).buffer(2).subscribe(new Consumer<List<Integer>>()
{
    // 2、分组为 (1, 2)(3, 4)(5, 6)(7)
    @Override
    public void accept(List<Integer> integers) throws Exception
    {
        Log.d("feather", count + " : " + integers.size());
    }
});
```

> 结果

```
D/feather: 0 : 2
D/feather: 0 : 1
```

takeUntil

1、takeUntil用于满足条件后，自动取消订阅。

takeWhile

Distinct

1、Distinct的作用是去除重复的项

> 基本数据类型、String常量都是能过滤的

2、Distinct如何处理对象的?

> 1. 例如：StudentA, name = "wang", StudentB, name = "wang"
> 2. 根据测试结果，是无法过滤的

```
Observable.just(new Student("wang"), new Student("chen"), new Student("hao"), new Student("wang
.distinct().subscribe(new Consumer<Student>()
{
    @Override
    public void accept(Student student) throws Exception
    {
        Log.d("feather", "" + student.name);
    }
});
```

> 结果

```
D/feather: wang
D/feather: chen
D/feather: hao
D/feather: wang
D/feather: wang
```

Filter

1、Filter的作用是过滤，通过判断的才发射。

```
Observable.just(new Student("wang"), new Student("chen"), new Student("hao"))
        // 过滤数据，自定义过滤的条件，返回true表示发射，false表示不会发射
.filter(new Predicate<Student>()
{
    @Override
    public boolean test(@NonNull Student student) throws Exception
    {
        return "wang".equals(student.name);
    }
})
.subscribe(new Consumer<Student>()
{
    @Override
    public void accept(Student student) throws Exception
    {
        Log.d("feather", "" + student.name);
    }
});
```

debounce

1、debounce的作用和使用方法

> 1. 去除掉发送频率过快的数据
> 2. 下例中移除掉发送间隔时间，低于500ms的数据。

```
// 1、间隔发送数据
Observable.create(new ObservableOnSubscribe<Integer>() {
    @Override
    public void subscribe(@NonNull ObservableEmitter<Integer> emitter) throws Exception {
        // send events with simulated time wait
        emitter.onNext(1); // skip
        Thread.sleep(400);
        emitter.onNext(2); // deliver
        Thread.sleep(505);
        emitter.onNext(3); // skip
        Thread.sleep(100);
        emitter.onNext(4); // deliver
        Thread.sleep(605);
        emitter.onNext(5); // deliver
        Thread.sleep(510);
        emitter.onComplete();
    }
    // 2. 去除掉发送间隔时间 <= 500ms都剔除掉，只发送了 2、4、5
}).debounce(500, TimeUnit.MILLISECONDS)
        .subscribeOn(Schedulers.io())
        .observeOn(AndroidSchedulers.mainThread())
        .subscribe(new Consumer<Integer>() {
            @Override
            public void accept(@NonNull Integer integer) throws Exception {
                    // 接收到数据
            }
        });
```

创建型

时间/周期

interval

1、interval的作用和使用场景

> 1. 间隔一定时间就发送一次数据
> 2. 默认在新线程
> 3. 适用场景: 定时轮询

```
    private Disposable mDisposable;

    @Override
    protected void doSomething() {
    // 1、1秒发送一次事件
        mDisposable = Flowable.interval(1, TimeUnit.SECONDS)
                .observeOn(AndroidSchedulers.mainThread())
                .subscribe(new Consumer<Long>() {
                    @Override
                    public void accept(@NonNull Long aLong) throws Exception {
                        // 2、定时周期性接收到事件，进行操作。
                    }
                });
    }

    /**
     * 2、销毁时停止心跳
     */
    @Override
    protected void onDestroy() {
        super.onDestroy();
        if (mDisposable != null){
            mDisposable.dispose();
        }
    }
```

intervalRange

1、intervalRange的作用

timer

1、timer的作用和使用

> 1. timer具有定时器的功能

```
Observable.timer(2, TimeUnit.SECONDS)
.subscribeOn(Schedulers.io())
// 1. timer 默认在新线程，所以需要切换回主线程
.observeOn(AndroidSchedulers.mainThread())
.subscribe(new Consumer<Long>() {
            @Override
            public void accept(@NonNull Long aLong) throws Exception {
                // 2. 接收到数据
            }
        });
```

辅助型操作符

delay

1、delay的作用

> 1. 用于延时发送数据

repeatWhen

1、repeatWhen提供重新订阅的功能

> 重订阅需要满足两个条件：
>
> 1. 需要上游回调 `onComplete()`
> 2. 告诉上游是否需要重订阅，通过 `repeatWhen的Function` 函数所返回的Observable确定
>   - 如果该Observable发送了 `onComplete/onError` 则表示 不需要重订阅 。 否则触发重订阅 。

2、repeatWhen使用时，发送onComplete，无法触发下游的onComplete回调。发送onError消息，可以触发下游的onError回调。为什么？

> 1. repeatWhen使用的是flatMap操作符
> 2. 下游的onComplete无法触发的原因:
>   1. flatMap变换后得到的每个子数据流中的completed事件并不会加入到合并后的事件流中
>   2. 只有flatMap的源事件流中的completed事件会加入到合并后的事件流中。
>   3. 这里如果返回Observable.Empty()，相当于是子事件流中的completed事件，所以不会触发最终的下游的onComplete回调。
> 3. 下游的onError可以触发的原因:
>   1. 但是error事件不同，任何一个flatMap的子数据流中的error都会中断最终的合并事件流并且被下游接收到。

retryWhen

1、retryWhen是收到onError()后触发是否要重订阅的逻辑判断

2、repeatWhen是收到上游的onComplete()后触发是否要重订阅的逻辑判断

defer

1、defer的作用

> 1. 每次 `Subscribe订阅时` 会创建一个新 `被观察者Observable`

```
// 1、def
Observable<Integer> observable = Observable.defer(new Callable<ObservableSource<? extends Integ
{
    @Override
    public ObservableSource<? extends Integer> call() throws Exception
    {
        // 2、每订阅一次，创建一个被观察者
        return Observable.just(1, 2, 3);
    }
});

// 订阅一次
observable.subscribe(xxx);
// 订阅一次
observable.subscribe(xxx);
```

last

1、last是取出可观察到的最后一个值

```
Observable.just(1, 2, 3)
          .last(4)
          .subscribe(new Consumer<Integer>() {
                    @Override
                    public void accept(@NonNull Integer integer) throws Exception {
                        mRxOperatorsText.append("last : " + integer + "\n");
                        Log.e(TAG, "last : " + integer + "\n");
                    }
          });
```

merge

2、merge的作用?

> 1. 能将多个 `Observable` 结合起来
> 2. `merge` 和 `concat` 区别在于:
>   - 不需要等待 `Observable 1` 事件全部发送完成，就可以发送 发射器2的事件

3、merge的使用

> 接收类型相同的数据

```
Observable.merge(Observable.just(1, 2), Observable.just(3, 4, 5))
                .subscribe(xxx);
                // 接收到1、2、3、4、5
```

接收类型不同的数据

4、如果merge的数据是不同的类型怎么办?

> 1. 用一个包装类进行包装，然后获取到之后通过 `instanceof` 进行判断

reduce

1、reduce的作用

> 1. 被观察者发出的每一个item都调用function进行处理，然后得到一个最终值，并且发射出去。
> 2. 例如1、2、3、4、5，funvtion是相加，结果=1+2+3+4+5

```
Observable.just(1, 2, 3)
.reduce(new BiFunction<Integer, Integer, Integer>() {
          // 1、两个item相加
          public Integer apply(@NonNull Integer integer, @NonNull Integer integer2) throws Exce
              return integer + integer2;
          }
}).subscribe(new Consumer<Integer>() {
           @Override
           public void accept(@NonNull Integer integer) throws Exception {
               // 输出结果 = 6
           }
});
```

scan

1、scan的作用？和reduce的区别?

> 1. 功能和reduce类似
> 2. 区别在于 每一个步骤的结果都会发射出去

```
Observable.just(1, 2, 3)
.scan(new BiFunction<Integer, Integer, Integer>() {
      @Override
      public Integer apply(@NonNull Integer integer, @NonNull Integer integer2) throws Exceptio
            return integer + integer2;
}})
.subscribe(xxx);
```

window

1、window的作用

> 1. 和Buffer类似，但是是发送
>
> 出 每组`item`的`Observable`， 每个`Observable`会依次发出这组数据中的数据

```
Observable.just(1, 2, 3)
.window(3, TimeUnit.SECONDS)
.subscribe(new Consumer<Observable<Integer>>() {
        @Override
        public void accept(@NonNull Observable<Integer> observable) throws Exception {
                // 1、接收到一组Item的Observable
                observable.subscribe(new Consumer<Integer>() {
                      @Override
                      public void accept(@NonNull Integer aLong) throws Exception {
                              // 2、依次接收到一组数据中的每个数据
                      }
                });
      }
});
```

Function

BiFunction

问题汇总

参考资料

1. 操作符-官方文档
2. RxJava 2.x 入门教程（五）
3. RxJava实战技巧大全
4. 放弃RxBus，拥抱RxJava（一）：为什么避免使用EventBus/RxBus
5. RxJava 教程第三部分：驯服数据流之 hot & cold Observable
6. Rxjava中的ConnectableObservable
7. RxJava2.0 操作符（9）—— Connectable Observable 连接操作符