

# 给初学者的RxJava2.0教程(九)



Season\_zlc (/u/c50b715ccaeb) [+ 关注](#)

2017.01.19 16:54\* 字数 2840 阅读 23552 评论 194 喜欢 634 赞赏 55

(/u/c50b715ccaeb)

Outline

[TOC]

## 前言

好久不见朋友们，最近一段时间在忙工作上的事情，没来得及写文章，这两天正好有点时间，赶紧写下了这篇教程，免得大家说我太监了。

## 正题

先来回顾一下上上节，我们讲Flowable的时候，说它采用了 响应式拉 的方式，我们还举了个 叶问打小日本 的例子，再来回顾一下吧，我们说把 上游 看成 小日本，把 下游 当作 叶问，当调用 Subscription.request(1) 时，叶问 就说 我要打一个！然后 小日本 就拿出 一个鬼子 给叶问，让他打，等叶问打死这个鬼子之后，再次调用 request(10)，叶问就又说 我要打十个！然后小日本又派出 十个鬼子 给叶问，然后就在边上看热闹，看叶问能不能打死十个鬼子，等叶问打死十个鬼子后再继续要鬼子接着打。

但是不知道大家有没有发现，在我们前两节中的例子中，我们口中声称的 响应式拉 并没有完全体现出来，比如这个例子：

```
Flowable.create(new FlowableOnSubscribe<Integer>() {
    @Override
    public void subscribe(FlowableEmitter<Integer> emitter) throws Exception {
        Log.d(TAG, "emit 1");
        emitter.onNext(1);
        Log.d(TAG, "emit 2");
        emitter.onNext(2);
        Log.d(TAG, "emit 3");
        emitter.onNext(3);
        Log.d(TAG, "emit complete");
        emitter.onComplete();
    }
}, BackpressureStrategy.ERROR).subscribeOn(Schedulers.io())
    .observeOn(AndroidSchedulers.mainThread())
    .subscribe(new Subscriber<Integer>() {

        @Override
        public void onSubscribe(Subscription s) {
            Log.d(TAG, "onSubscribe");
            mSubscription = s;
            s.request(1);
        }

        @Override
        public void onNext(Integer integer) {
            Log.d(TAG, "onNext: " + integer);
            mSubscription.request(1);
        }

        @Override
        public void onError(Throwable t) {
            Log.w(TAG, "onError: ", t);
        }

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

虽然我们在下游中是每次处理掉了一个事件之后才调用request(1)去请求下一个事件，也就是说叶问的确是在打死了一个鬼子之后才继续打下一个鬼子，可是上游呢？上游真的是每次当下游请求一个才拿出一个吗？从上一篇文章中我们知道并不是这样的，上游仍然是一开始就发送了所有的事件，也就是说小日本并没有等叶问打死一个才拿出一个，而是一开始就拿出了所有的鬼子，这些鬼子从一开始就在这儿排队等着被打死。

有个故事是这么说的：

楚人有卖盾与矛者，先誉其盾之坚，曰：“吾盾之坚，物莫能陷也。”俄而又誉其矛之利，曰：“吾矛之利，万物莫不陷也。”市人诘之曰：“以子之矛陷子之盾，何如？”其人弗能应也。众皆笑之。

没错，我们前后所说的就是自相矛盾了，这说明了什么呢，说明我们的实现并不是一个完整的实现，那么，究竟怎样的实现才是完整的呢？

我们先自己来想一想，在下游中调用Subscription.request(n)就可以告诉上游，下游能够处理多少个事件，那么上游要根据下游的处理能力正确的去发送事件，那么上游是不是应该知道下游的处理能力是多少啊，对吧，不然，一个巴掌拍不响啊，这种事情得你情我愿才行。

那么上游从哪里得知下游的处理能力呢？我们来看看上游最重要的部分，肯定就是FlowableEmitter了啊，我们就是通过它来发送事件的啊，来看看它的源码吧(别紧张，它的代码灰常简单)：

```
public interface FlowableEmitter<T> extends Emitter<T> {
    void setDisposable(Disposable s);
    void setCancellable(Cancellable c);

    /**
     * The current outstanding request amount.
     * <p>This method is thread-safe.
     * @return the current outstanding request amount
     */
    long requested();

    boolean isCancelled();
    FlowableEmitter<T> serialize();
}
```

FlowableEmitter是个接口，继承Emitter，Emitter里面就是我们的onNext(),onComplete()和onError()三个方法。我们看到FlowableEmitter中有这么一个方法：

```
long requested();
```

方法注释的意思就是 当前外部请求的数量，哇哦，这好像就是我们要找的答案呢. 我们还是实际验证一下吧.

先来看 同步 的情况吧：

```

public static void demo1() {
    Flowable
        .create(new FlowableOnSubscribe<Integer>() {
            @Override
            public void subscribe(FlowableEmitter<Integer> emitter) throws Except
ion {
                Log.d(TAG, "current requested: " + emitter.requested());
            }
        }, BackpressureStrategy.ERROR)
        .subscribe(new Subscriber<Integer>() {

            @Override
            public void onSubscribe(Subscription s) {
                Log.d(TAG, "onSubscribe");
                mSubscription = s;
            }

            @Override
            public void onNext(Integer integer) {
                Log.d(TAG, "onNext: " + integer);
            }

            @Override
            public void onError(Throwable t) {
                Log.w(TAG, "onError: ", t);
            }

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

        });
}

```

这个例子中，我们在上游中打印出当前的request数量，下游什么也不做。

我们先猜测一下结果，下游没有调用request()，说明当前下游的处理能力为0，那么上游得到的requested也应该是0，是不是呢？

来看看运行结果：

```

D/TAG: onSubscribe
D/TAG: current requested: 0

```

哈哈，结果果然是0，说明我们的结论基本上是对的。

那下游要是调用了request()呢，来看看：

```

public static void demo1() {
    Flowable
        .create(new FlowableOnSubscribe<Integer>() {
            @Override
            public void subscribe(FlowableEmitter<Integer> emitter) throws Except
ion {
                Log.d(TAG, "current requested: " + emitter.requested());
            }
        }, BackpressureStrategy.ERROR)
        .subscribe(new Subscriber<Integer>() {

            @Override
            public void onSubscribe(Subscription s) {
                Log.d(TAG, "onSubscribe");
                mSubscription = s;
                s.request(10); //我要打十个!
            }

            @Override
            public void onNext(Integer integer) {
                Log.d(TAG, "onNext: " + integer);
            }

            @Override
            public void onError(Throwable t) {
                Log.w(TAG, "onError: ", t);
            }

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

```

这次在下游中调用了request(10)，告诉上游我要打十个，看看运行结果：

```

D/TAG: onSubscribe
D/TAG: current requested: 10

```

果然！上游的requested的确是根据下游的请求来决定的，那要是下游多次请求呢？比如这样：

```

public static void demo1() {
    Flowable
        .create(new FlowableOnSubscribe<Integer>() {
            @Override
            public void subscribe(FlowableEmitter<Integer> emitter) throws Except
ion {
                Log.d(TAG, "current requested: " + emitter.requested());
            }
        }, BackpressureStrategy.ERROR)
        .subscribe(new Subscriber<Integer>() {

            @Override
            public void onSubscribe(Subscription s) {
                Log.d(TAG, "onSubscribe");
                mSubscription = s;
                s.request(10); //我要打十个!
                s.request(100); //再给我一百个!
            }

            @Override
            public void onNext(Integer integer) {
                Log.d(TAG, "onNext: " + integer);
            }

            @Override
            public void onError(Throwable t) {
                Log.w(TAG, "onError: ", t);
            }

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

```

下游先调用了request(10), 然后又调用了request(100), 来看看运行结果：

```
D/TAG: onSubscribe
D/TAG: current requested: 110
```

看来多次调用也没问题，做了 加法。

诶加法？对哦，只是做加法，那什么时候做 减法 呢？

当然是发送事件啦！

来看个例子吧：

```
public static void demo2() {
    Flowable
        .create(new FlowableOnSubscribe<Integer>() {
            @Override
            public void subscribe(final FlowableEmitter<Integer> emitter) throws
Exception {
                Log.d(TAG, "before emit, requested = " + emitter.requested());

                Log.d(TAG, "emit 1");
                emitter.onNext(1);
                Log.d(TAG, "after emit 1, requested = " + emitter.requested());

                Log.d(TAG, "emit 2");
                emitter.onNext(2);
                Log.d(TAG, "after emit 2, requested = " + emitter.requested());

                Log.d(TAG, "emit 3");
                emitter.onNext(3);
                Log.d(TAG, "after emit 3, requested = " + emitter.requested());

                Log.d(TAG, "emit complete");
                emitter.onComplete();

                Log.d(TAG, "after emit complete, requested = " + emitter.requeste
d());
            }
        }, BackpressureStrategy.ERROR)
        .subscribe(new Subscriber<Integer>() {
            @Override
            public void onSubscribe(Subscription s) {
                Log.d(TAG, "onSubscribe");
                mSubscription = s;
                s.request(10); //request 10
            }

            @Override
            public void onNext(Integer integer) {
                Log.d(TAG, "onNext: " + integer);
            }

            @Override
            public void onError(Throwable t) {
                Log.w(TAG, "onError: ", t);
            }

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

代码很简单，来看看运行结果：

```

D/TAG: onSubscribe
D/TAG: before emit, requested = 10
D/TAG: emit 1
D/TAG: onNext: 1
D/TAG: after emit 1, requested = 9
D/TAG: emit 2
D/TAG: onNext: 2
D/TAG: after emit 2, requested = 8
D/TAG: emit 3
D/TAG: onNext: 3
D/TAG: after emit 3, requested = 7
D/TAG: emit complete
D/TAG: onComplete
D/TAG: after emit complete, requested = 7

```

大家应该能看出端倪了吧，下游调用request(n) 告诉上游它的处理能力，上游每发送一个 next事件 之后，requested就减一， 注意是next事件，complete和error事件不会消耗requested，当减到0时，则代表下游没有处理能力了，这个时候你如果继续发送事件，会发生什么后果呢？当然是 MissingBackpressureException 啦，试一试：

```

public static void demo2() {
    Flowable
        .create(new FlowableOnSubscribe<Integer>() {
            @Override
            public void subscribe(final FlowableEmitter<Integer> emitter) throws
Exception {
                Log.d(TAG, "before emit, requested = " + emitter.requested());

                Log.d(TAG, "emit 1");
                emitter.onNext(1);
                Log.d(TAG, "after emit 1, requested = " + emitter.requested());

                Log.d(TAG, "emit 2");
                emitter.onNext(2);
                Log.d(TAG, "after emit 2, requested = " + emitter.requested());

                Log.d(TAG, "emit 3");
                emitter.onNext(3);
                Log.d(TAG, "after emit 3, requested = " + emitter.requested());

                Log.d(TAG, "emit complete");
                emitter.onComplete();

                Log.d(TAG, "after emit complete, requested = " + emitter.requeste
d());
            }
        }, BackpressureStrategy.ERROR)
        .subscribe(new Subscriber<Integer>() {

            @Override
            public void onSubscribe(Subscription s) {
                Log.d(TAG, "onSubscribe");
                mSubscription = s;
                s.request(2); //request 2
            }

            @Override
            public void onNext(Integer integer) {
                Log.d(TAG, "onNext: " + integer);
            }

            @Override
            public void onError(Throwable t) {
                Log.w(TAG, "onError: ", t);
            }

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

        });
}

```

还是这个例子，只不过这次只request(2), 看看运行结果：

```

D/TAG: onSubscribe
D/TAG: before emit, requested = 2
D/TAG: emit 1
D/TAG: onNext: 1
D/TAG: after emit 1, requested = 1
D/TAG: emit 2
D/TAG: onNext: 2
D/TAG: after emit 2, requested = 0
D/TAG: emit 3
W/TAG: onError: io.reactivex.exceptions.MissingBackpressureException: create: could not
emit value due to lack of requests
    at io.reactivex.internal.operators.flowable.FlowableCreate$ErrorAsyncEmi
tter.onOverflow(FlowableCreate.java:411)
    at io.reactivex.internal.operators.flowable.FlowableCreate$NoOverflowBas
eAsyncEmitter.onNext(FlowableCreate.java:377)
    at zlc.season.rxjava2demo.demo.ChapterNine$4.subscribe(ChapterNine.java:
80)
    at io.reactivex.internal.operators.flowable.FlowableCreate.subscribeActu
al(FlowableCreate.java:72)
    at io.reactivex.Flowable.subscribe(Flowable.java:12218)
    at zlc.season.rxjava2demo.demo.ChapterNine.demo2(ChapterNine.java:89)
    at zlc.season.rxjava2demo.MainActivity$2.onClick(MainActivity.java:36)
    at android.view.View.performClick(View.java:4780)
    at android.view.View$PerformClick.run(View.java:19866)
    at android.os.Handler.handleCallback(Handler.java:739)
    at android.os.Handler.dispatchMessage(Handler.java:95)
    at android.os.Looper.loop(Looper.java:135)
    at android.app.ActivityThread.main(ActivityThread.java:5254)
    at java.lang.reflect.Method.invoke(Native Method)
    at java.lang.reflect.Method.invoke(Method.java:372)
    at com.android.internal.os.ZygoteInit$MethodAndArgsCaller.run(ZygoteInit
.java:903)
    at com.android.internal.os.ZygoteInit.main(ZygoteInit.java:698)
D/TAG: after emit 3, requested = 0
D/TAG: emit complete
D/TAG: after emit complete, requested = 0

```

到目前为止我们一直在说同步的订阅，现在同步说完了，我们先用一张图来总结一下同步的情况：

同步request.png

这张图的意思就是当上下游在同一个线程中的时候，在下游调用request(n)就会直接改变上游中的requested的值，多次调用便会叠加这个值，而上游每发送一个事件之后便会去减少这个值，当这个值减少至0的时候，继续发送事件便会抛异常了。

我们再来说说异步的情况，异步和同步会有区别吗？会有什么区别呢？带着这个疑问我们继续来探究。

同样的先来看一个基本的例子：

```

public static void demo3() {
    Flowable
        .create(new FlowableOnSubscribe<Integer>() {
            @Override
            public void subscribe(FlowableEmitter<Integer> emitter) throws Except
ion {
                Log.d(TAG, "current requested: " + emitter.requested());
            }
        }, BackpressureStrategy.ERROR)
        .subscribeOn(Schedulers.io())
        .observeOn(AndroidSchedulers.mainThread())
        .subscribe(new Subscriber<Integer>() {

            @Override
            public void onSubscribe(Subscription s) {
                Log.d(TAG, "onSubscribe");
                mSubscription = s;
            }

            @Override
            public void onNext(Integer integer) {
                Log.d(TAG, "onNext: " + integer);
            }

            @Override
            public void onError(Throwable t) {
                Log.w(TAG, "onError: ", t);
            }

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

```

这次是异步的情况，上游啥也不做，下游也啥也不做，来看看运行结果：

```

D/TAG: onSubscribe
D/TAG: current requested: 128

```

哈哈，又是128，看了我前几篇文章的朋友肯定很熟悉这个数字啊！这个数字为什么和我们之前所说的默认的水缸大小一样啊，莫非？

带着这个疑问我们继续来研究一下：



```

public static void demo3() {
    Flowable
        .create(new FlowableOnSubscribe<Integer>() {
            @Override
            public void subscribe(FlowableEmitter<Integer> emitter) throws Except
ion {
                Log.d(TAG, "current requested: " + emitter.requested());
            }
        }, BackpressureStrategy.ERROR)
        .subscribeOn(Schedulers.io())
        .observeOn(AndroidSchedulers.mainThread())
        .subscribe(new Subscriber<Integer>() {

            @Override
            public void onSubscribe(Subscription s) {
                Log.d(TAG, "onSubscribe");
                mSubscription = s;
                s.request(1000); //我要打1000个!!
            }

            @Override
            public void onNext(Integer integer) {
                Log.d(TAG, "onNext: " + integer);
            }

            @Override
            public void onError(Throwable t) {
                Log.w(TAG, "onError: ", t);
            }

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

        });
}
}

```

这次我们在下游调用了request ( 1000 ) 告诉上游我要打1000个，按照之前我们说的，这次的运行结果应该是1000，来看看运行结果：

```

D/TAG: onSubscribe
D/TAG: current requested: 128

```

卧槽，你确定你没贴错代码？

是的，真相就是这样，就是128，蜜汁128。。。

what happened?

---

I don't know !

---

为了答疑解惑，我就直接上图了：

异步request.png

---

可以看到，当上下游工作在不同的线程里时，每一个线程里都有一个requested，而我们调用request ( 1000 ) 时，实际上改变的是下游主线程中的requested，而上游中的requested的值是由RxJava内部调用request(n)去设置的，这个调用会在合适的时候自动触发。

现在我们就理解为什么没有调用request，上游中的值是128了，因为下游在 一开始就在 内部调用了 request(128)去设置了上游中的值，因此即使下游没有调用request()，上游也能发送128个事件，这也可以解释之前我们为什么说Flowable中默认的水缸大小是128，其实就是这里设置的。

刚才同步的时候我们说了，上游每发送一个事件，requested的值便会减一，对于异步来说同样如此，那有人肯定有疑问了，一开始上游的requested的值是128，那这128个事件发送完了不就不能继续发送了吗？

刚刚说了，设置上游requested的值的这个内部调用会在 合适的时候 自动触发，那到底什么时候是合适的时候呢？是发完128个事件才去调用吗？还是发送了一半才去调用呢？

带着这个疑问我们来看下一段代码：

```

public static void request() {
    mSubscription.request(96); //请求96个事件
}

public static void demo4() {
    Flowable
        .create(new FlowableOnSubscribe<Integer>() {
            @Override
            public void subscribe(FlowableEmitter<Integer> emitter) throws Except
ion {
                Log.d(TAG, "First requested = " + emitter.requested());
                boolean flag;
                for (int i = 0; ; i++) {
                    flag = false;
                    while (emitter.requested() == 0) {
                        if (!flag) {
                            Log.d(TAG, "Oh no! I can't emit value!");
                            flag = true;
                        }
                    }
                    emitter.onNext(i);
                    Log.d(TAG, "emit " + i + " , requested = " + emitter.requeste
d());
                }
            }
        }, BackpressureStrategy.ERROR)
        .subscribeOn(Schedulers.io())
        .observeOn(AndroidSchedulers.mainThread())
        .subscribe(new Subscriber<Integer>() {

            @Override
            public void onSubscribe(Subscription s) {
                Log.d(TAG, "onSubscribe");
                mSubscription = s;
            }

            @Override
            public void onNext(Integer integer) {
                Log.d(TAG, "onNext: " + integer);
            }

            @Override
            public void onError(Throwable t) {
                Log.w(TAG, "onError: ", t);
            }

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

        });
}

```

这次的上游稍微复杂了一点点，首先仍然是个无限循环发事件，但是是有条件的，只有当上游的requested != 0的时候才会发事件，然后我们调用request ( 96 ) 去消费96个事件（为什么是96而不是其他的数字先不要管），来看看运行结果吧：

```

D/TAG: onSubscribe
D/TAG: First requested = 128
D/TAG: emit 0 , requested = 127
D/TAG: emit 1 , requested = 126
D/TAG: emit 2 , requested = 125
...
D/TAG: emit 124 , requested = 3
D/TAG: emit 125 , requested = 2
D/TAG: emit 126 , requested = 1
D/TAG: emit 127 , requested = 0
D/TAG: Oh no! I can't emit value!

```

首先运行之后上游便会发送完128个事件，之后便不做任何事情，从打印的结果中我们也可以看出这一点。

然后我们调用request(96)，这会让下游去消费96个事件，来看看运行结果吧：

```
D/TAG: onNext: 0
D/TAG: onNext: 1
...
D/TAG: onNext: 92
D/TAG: onNext: 93
D/TAG: onNext: 94
D/TAG: onNext: 95
D/TAG: emit 128 , requested = 95
D/TAG: emit 129 , requested = 94
D/TAG: emit 130 , requested = 93
D/TAG: emit 131 , requested = 92
...
D/TAG: emit 219 , requested = 4
D/TAG: emit 220 , requested = 3
D/TAG: emit 221 , requested = 2
D/TAG: emit 222 , requested = 1
D/TAG: emit 223 , requested = 0
D/TAG: Oh no! I can't emit value!
```

可以看到，当下游消费掉第96个事件之后，上游又开始发事件了，而且可以看到当前上游的requested的值是96(打印出来的95是已经发送了一个事件减一之后的值)，最终发出了第223个事件之后又进入了等待区，而223-127 正好等于 96。

这不是说明当下游每消费96个事件便会自动触发内部的request()去设置上游的requested的值啊！没错，就是这样，而这个新的值就是96。

朋友们可以手动试试请求95个事件，上游是不会继续发送事件的。

至于这个96是怎么得出来的（肯定不是我猜的蒙的啊），感兴趣的朋友可以自行阅读源码寻找答案，对于初学者而言应该没什么必要，管它内部怎么实现的呢对吧。

好了今天的教程就到这里了！通过本节的学习，大家应该知道如何正确的去实现一个完整的响应式拉取了，在 某些场景 下，可以在发送事件前先判断当前的requested的值是否大于0，若等于0则说明下游处理不过来了，则需要等待，例如下面这个例子。

## 实践

这个例子是读取一个文本文件，需要一行一行读取，然后处理并输出，如果文本文件很大的时候，比如几十M的时候，全部先读入内存肯定不是明智的做法，因此我们可以一边读取一边处理，实现的代码如下：

```

public static void main(String[] args) {
    practice1();
    try {
        Thread.sleep(10000000);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}

public static void practice1() {
    Flowable
        .create(new FlowableOnSubscribe<String>() {
            @Override
            public void subscribe(FlowableEmitter<String> emitter) throws Excepti
on {
                try {
                    FileReader reader = new FileReader("test.txt");
                    BufferedReader br = new BufferedReader(reader);

                    String str;

                    while ((str = br.readLine()) != null && !emitter.isCancelled(
)) {
                        while (emitter.requested() == 0) {
                            if (emitter.isCancelled()) {
                                break;
                            }
                        }
                        emitter.onNext(str);
                    }

                    br.close();
                    reader.close();

                    emitter.onComplete();
                } catch (Exception e) {
                    emitter.onError(e);
                }
            }
        }, BackpressureStrategy.ERROR)
        .subscribeOn(Schedulers.io())
        .observeOn(Schedulers.newThread())
        .subscribe(new Subscriber<String>() {

            @Override
            public void onSubscribe(Subscription s) {
                mSubscription = s;
                s.request(1);
            }

            @Override
            public void onNext(String string) {
                System.out.println(string);
                try {
                    Thread.sleep(2000);
                    mSubscription.request(1);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }

            @Override
            public void onError(Throwable t) {
                System.out.println(t);
            }

            @Override
            public void onComplete() {
            }





        }));
}

```

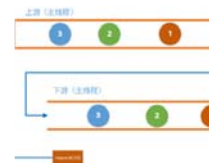
运行的结果便是：

(/apps/download?utm\_source=nbc)

被以下专题收入，发现更多相似内容


-  RxJava2.x (/c/299d0a51fdd4?utm\_source=desktop&utm\_medium=notes-included-collection)
-  RxJava (/c/a904f328163d?utm\_source=desktop&utm\_medium=notes-included-collection)
-  Android... (/c/58b4c20abf2f?utm\_source=desktop&utm\_medium=notes-included-collection)
-  Android知识 (/c/3fde3b545a35?utm\_source=desktop&utm\_medium=notes-included-collection)
-  首页投稿 (/c/bDHhpK?utm\_source=desktop&utm\_medium=notes-included-collection)
-  AndroidHot (/c/5e13337f1ef?utm\_source=desktop&utm\_medium=notes-included-collection)
-  手机移动程序开发 (/c/52b64f3155e1?utm\_source=desktop&utm\_medium=notes-included-collection)
- 展开更多 ▾

(/p/186589febcec?




utm\_campaign=maleskine&utm\_content=note&utm\_medium=seo\_notes&utm\_source=recommendation)  
**给初学者的RxJava2.0教程(九)** (/p/186589febcec?utm\_campaign=males...

特此声明：本文为转载文章！尊重原创的劳动果实，严禁剽窃本文转载于：  
http://www.jianshu.com/p/36e0f7f43a51出自于：【Season\_zlc】前言 好久不见朋友们，最近一段时间在...

 社会你鹏哥 (/u/d512e2781e94?  
utm\_campaign=maleskine&utm\_content=user&utm\_medium=seo\_notes&utm\_source=recommendation)

**RxJava 2.x 学习2 (转载)** (/p/f73193dccc69?utm\_campaign=maleskine...

怎么如此平静，感觉像是走错了片场。为什么呢，因为上下游工作在同一个线程呀骚年们！这个时候上游每次调用emitter.onNext(i)其实就相当于直接调用了Consumer中的: public void accept(Integer integer) throws E...


 Young1657 (/u/64cccdaa9204?  
utm\_campaign=maleskine&utm\_content=user&utm\_medium=seo\_notes&utm\_source=recommendation)

(/p/f4ed455de5f0?



utm\_campaign=maleskine&utm\_content=note&utm\_medium=seo\_notes&utm\_source=recommendation)  
**给初学者的RxJava2.0教程(七)** (/p/f4ed455de5f0?utm\_campaign=malesk...

特此声明：本文为转载文章！尊重原创的劳动果实，严禁剽窃本文转载于：  
http://www.jianshu.com/p/9b1304435564出自于：【Season\_zlc】前言 上一节里我们学习了只使用...

 社会你鹏哥 (/u/d512e2781e94?

utm\_campaign=maleskine&utm\_content=user&utm\_medium=seo\_notes&utm\_source=recommendation)

(/p/9b1304435564?



utm\_campaign=maleskine&utm\_content=note&utm\_medium=seo\_notes&utm\_source=recommendation)

## 给初学者的RxJava2.0教程(七) (/p/9b1304435564?utm\_campaign=males...

Outline [TOC] 前言 上一节里我们学习了只使用Observable如何去解决上下游流速不均衡的问题, 之所以学习这个是因为Observable还是有很多它使用的场景, 有些朋友自从听说了Flowable之后就觉得Flowable能解决...



Season\_zlc (/u/c50b715ccaeb?

utm\_campaign=maleskine&utm\_content=user&utm\_medium=seo\_notes&utm\_source=recommendation)

## 给初学者的RxJava2.0教程(八) (/p/a75ecf461e02?utm\_campaign=malesk...

Outline [TOC] 前言 在上一节中, 我们学习了Flowable的一些基本知识, 同时也挖了许多坑, 这一节就让我们来填坑吧. 正题 在上一节中最后我们有个例子, 当上游一次性发送128个事件的时候是没有任何问题的, 一...



Season\_zlc (/u/c50b715ccaeb?

utm\_campaign=maleskine&utm\_content=user&utm\_medium=seo\_notes&utm\_source=recommendation)

(/p/275fb2faebee?



utm\_campaign=maleskine&utm\_content=note&utm\_medium=seo\_notes&utm\_source=recommendation)

## 第一个月检视 (/p/275fb2faebee?utm\_campaign=maleskine&utm\_conte...

参加易效能90践行已经30天了, 从每一天早起打卡开始到排程确定当日青蛙, 每天像玩游戏一样一步一步闯关, 遇到受伤和难关到群里找宝物与充血继续战斗..... 第一周完成; 梦想版, 90天目标计划, 每日找青蛙...



夏峥嵘 (/u/e3fd7371464a?

utm\_campaign=maleskine&utm\_content=user&utm\_medium=seo\_notes&utm\_source=recommendation)

(/p/4e88e460bf41?



utm\_campaign=maleskine&utm\_content=note&utm\_medium=seo\_notes&utm\_source=recommendation)

## 优柔寡断这种病, 这本书也许可以治 (/p/4e88e460bf41?utm\_campaign=m...

简书第二篇文章, 也是与《思考的艺术》这本书交谈的第一天, 希望这会是一段波澜壮阔的旅程. 现在这本书就摆在我的左边, 办公室的同事已经走得一个不剩, 诺大的办公室变成了我的秘密基地, 噢, 不对...



魔镜神灯 (/u/b2ef12a2abf6?

utm\_campaign=maleskine&utm\_content=user&utm\_medium=seo\_notes&utm\_source=recommendation)

## 三思而后言 (/p/357376ab5837?utm\_campaign=maleskine&utm\_content...

三思而后行是一个汉语词汇, 读作是sān sī ér hòu xíng。三思而后行出于《论语》, 这句话的意思是: 凡事都要再三思考而后行。——摘自百度“三思而后言”, 故名思义, 凡事再三思考后再说出口。想要写这个是...



Lyumy (/u/c2c7e54bf786?

utm\_campaign=maleskine&utm\_content=user&utm\_medium=seo\_notes&utm\_source=recommendation)

(/p/2716a3341b0d?



utm\_campaign=maleskine&utm\_content=note&utm\_medium=seo\_notes&utm\_source=recommendation)

## 微情书，写给未来的你 (/p/2716a3341b0d?utm\_campaign=maleskine&ut...

一《爱》如果你愿意 我想把我的喜欢 来一个升华 二《珍惜》教会我珍惜的人 一定不是你 你是让我去珍惜的人 三《默契》我们相视一笑 像一场 温暖的话剧 四《灯塔》如果你迷路了 不要怕 我会是你的灯塔 五...



陌轩 (/u/2a3371829ba4?

utm\_campaign=maleskine&utm\_content=user&utm\_medium=seo\_notes&utm\_source=recommendation)

---

## 石墨烯第一龙头躁动，神秘巨资悄然抬轿，9月有望复制方大碳素 (/p/5a08f...

儿子把学习单拿回来，老妈一看，大喜道：“真不错，语文飚升，85分;数学小涨，92分;英语横盘;75分;政治止跌反弹，85分;;物理已经企稳，80分;只有化学微跌，74分，不过还在上升道。”老爸忙接过话：“你小子...



皓宇看股 (/u/ed111034a99c?

utm\_campaign=maleskine&utm\_content=user&utm\_medium=seo\_notes&utm\_source=recommendation)