RxJava2 源码解析（二）

原创 2017年03月20日 10:06:29 标签：Rxjava / Rxjava2 / 源码解析 / 线程调度 / 操作符 3016

概述

承接上一篇RxJava2 源码解析（一），
本系列我们的目的：

1. 知道源头(`Observable`)是如何将数据发送出去的。
2. 知道终点（`Observer`）是如何接收到数据的。
3. **何时**将源头和终点**关联**起来的
4. **知道线程调度是怎么实现的**
5. **知道操作符是怎么实现的**

本篇计划讲解一下4,5.

RxJava最强大的莫过于它的**线程调度** 和 **花式操作符**。

map操作符

map是一个高频的操作符，我们首先拿他开刀。
例子如下，源头`Observable`发送的是String类型的数字，利用map转换成int型，最终在终点`Observer`接受到的也是int类型数据。：

```
        final Observable<String> testCreateObservable = Observable.create(new Observable
OnSubscribe<String>() {
            @Override
            public void subscribe(ObservableEmitter<String> e) throws Exception {
                e.onNext("1");
                e.onComplete()
            }
        });
```

```
                Observable<Integer> map = testCreateObservable.map(new Function<String,
Integer>() {
                    @Override
                    public Integer apply(String s) throws Exception {
                        return Integer.parseInt(s);
                    }
                });
                map.subscribe(new Observer<Integer>() {
                    @Override
                    public void onSubscribe(Disposable d) {
                        Log.d(TAG, "onSubscribe() called with: d = [" + d + "]");
                    }

                    @Override
                    public void onNext(Integer value) {
                        Log.d(TAG, "onNext() called with: value = [" + value + "]");
                    }

                    @Override
                    public void onError(Throwable e) {
                        Log.d(TAG, "onError() called with: e = [" + e + "]");
                    }

```

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

我们看一下`map`函数的源码：

```
    public final <R> Observable<R> map(Function<? super T, ? extends R> mapper) {
        //判空略过
        ObjectHelper.requireNonNull(mapper, "mapper is null");
        //RxJavaPlugins.onAssembly()是hook 上文提到过
        return RxJavaPlugins.onAssembly(new ObservableMap<T, R>(this, mapper));
    }
```

`RxJavaPlugins.onAssembly()`是hook 上文提到过,所以我们只要看`ObservableMap`，它就是**返回到我们手里的`Observable`**：

```
public final class ObservableMap<T, U> extends AbstractObservableWithUpstream<T, U> {
    //将function变换函数类保存起来
    final Function<? super T, ? extends U> function;

    public ObservableMap(ObservableSource<T> source, Function<? super T, ? extends U> fu
nction) {
        //super()将上游的Observable保存起来 ， 用于subscribeActual()中用。
        super(source);
        this.function = function;
    }

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

它继承自`AbstractObservableWithUpstream`,该类继承自`Observable`，很简单，就是**将上游的`ObservableSource`保存起来**，做一次wrapper，所以它也算是装饰者模式的提现，如下:

```
abstract class AbstractObservableWithUpstream<T, U> extends Observable<U> implements Has
UpstreamObservableSource<T> {
    //将上游的`ObservableSource`保存起来
    protected final ObservableSource<T> source;
    AbstractObservableWithUpstream(ObservableSource<T> source) {
        this.source = source;
    }
    @Override
    public final ObservableSource<T> source() {
        return source;
    }
}
```

关于`ObservableSource`，代表了一个标准的无背压的 源数据接口，可以被`Observer`消费（订阅），如下：

```
public interface ObservableSource<T> {
    void subscribe(Observer<? super T> observer);
}
```

所有的`Observable`都已经实现了它,所以我们可以**认为`Observable`和`ObservableSource`在本文中是相等的**：

```
public abstract class Observable<T> implements ObservableSource<T> {
```

所以我们得到的`ObservableMap`对象也很简单，就是**将上游的`Observable`和变换函数类`Function`保存起来。**
`Function`的定义超级简单，就是一个接口，给我一个T，还你一个R.

```
public interface Function<T, R> {
    R apply(T t) throws Exception;
}
```

本例写的是将String->int.

重头戏，`subscribeActual()`是订阅真正发生的地方，`ObservableMap`如下编写，就一句话，**用MapObserver订阅上游Observable。**：

```
    @Override
    public void subscribeActual(Observer<? super U> t) {
    //用MapObserver订阅上游Observable。
        source.subscribe(new MapObserver<T, U>(t, function));
    }
```

`MapObserver`也是装饰者模式，对终点（下游）`Observer`修饰。

```
    static final class MapObserver<T, U> extends BasicFuseableObserver<T, U> {
        final Function<? super T, ? extends U> mapper;
        MapObserver(Observer<? super U> actual, Function<? super T, ? extends U> mapper)
{
            //super()将actual保存起来
            super(actual);
            //保存Function变量
            this.mapper = mapper;
        }
        @Override
        public void onNext(T t) {
            //done在onError 和 onComplete以后才会是true，默认这里是false，所以跳过
            if (done) {
                return;
            }
            //默认sourceMode是0，所以跳过
            if (sourceMode != NONE) {
                actual.onNext(null);
                return;
            }
            //下游Observer接受的值
            U v;
            //这一步执行变换,将上游传过来的T，利用Function转换成下游需要的U。
            try {
                v = ObjectHelper.requireNonNull(mapper.apply(t), "The mapper function re
turned a null value.");
            } catch (Throwable ex) {
                fail(ex);
                return;
            }
            //变换后传递给下游Observer
            actual.onNext(v);
        }
```

到此我们梳理一下流程：
订阅的过程，是从**下游到上游依次订阅**的。

1. 即**终点** `Observer` **订阅了** `map` 返回的`ObservableMap`。
2. 然后`map`的`Observable`(`ObservableMap`)在被订阅时，会**订阅其内部保存上游**`Observable`，用于订阅上游的`Observer`是一个装饰者(`MapObserver`)，**内部保存了下游（本例是终点）**`Observer`，**以便上游发送数据过来时，能传递给下游。**
3. 以此类推，**直到源头**`Observable`**被订阅**，根据上节课内容，它**开始向Observer发送数据。**

数据传递的过程，当然是**从上游push到下游**的，

1. **源头**`Observable`**传递数据给下游**`Observer`（本例就是`MapObserver`）
2. 然后`MapObserver`接收到数据，**对其变换操作后**(实际的function在这一步执行)，再调用内部保存的下游`Observer`的`onNext()`**发送数据给下游**
3. 以此类推，直到**终点**`Observer`。

线程调度subscribeOn

简化问题，代码如下：

```
                Observable.create(new ObservableOnSubscribe<String>() {
                    @Override
                    public void subscribe(ObservableEmitter<String> e) throws Exception
{
                        Log.d(TAG, "subscribe() called with: e = [" + e + "]" + Thread.c
urrentThread());
                        e.onNext("1");
                        e.onComplete();
                    }
                    //只是在Observable和Observer之间增加了一句线程调度代码
                }).subscribeOn(Schedulers.io())
                        .subscribe(new Observer<String>() {
                            @Override
                            public void onSubscribe(Disposable d) {
                                Log.d(TAG, "onSubscribe() called with: d = [" + d +
"]");
                            }
                            @Override
                            public void onNext(String value) {
                                Log.d(TAG, "onNext() called with: value = [" + value +
"]");
                            }
                            @Override
                            public void onError(Throwable e) {
                                Log.d(TAG, "onError() called with: e = [" + e + "]");
                            }
                            @Override
                            public void onComplete() {
                                Log.d(TAG, "onComplete() called");
                            }
                        });
```

只是在`Observable`和`Observer`之间增加了一句线程调度代码: `.subscribeOn(Schedulers.io())`.
查看`subscribeOn()`源码：

```
    public final Observable<T> subscribeOn(Scheduler scheduler) {
    //判空略过
        ObjectHelper.requireNonNull(scheduler, "scheduler is null");
        //抛开Hook，重点还是ObservableSubscribeOn
        return RxJavaPlugins.onAssembly(new ObservableSubscribeOn<T>(this, scheduler));
    }
```

等等，怎么有种似曾相识的感觉，大家可以把文章向上翻，看看`map()`的源码。
和`subscribeOn()`的套路如出一辙，那么我们根据上面的结论，
先猜测**`ObservableSubscribeOn`类也是一个包装类（装饰者）**，点进去查看：

```
public final class ObservableSubscribeOn<T> extends AbstractObservableWithUpstream<T, T>
{
    //保存线程调度器
    final Scheduler scheduler;
    public ObservableSubscribeOn(ObservableSource<T> source, Scheduler scheduler) {
        //map的源码中我们分析过，super()只是简单的保存ObservableSource
        super(source);
        this.scheduler = scheduler;
    }
    @Override
    public void subscribeActual(final Observer<? super T> s) {
        //1  创建一个包装Observer
        final SubscribeOnObserver<T> parent = new SubscribeOnObserver<T>(s);
        //2  手动调用 下游（终点）Observer.onSubscribe()方法,所以onSubscribe()方法执行在 订
阅处所在的线程
        s.onSubscribe(parent);
        //3 setDisposable()是为了将子线程的操作加入Disposable管理中
        parent.setDisposable(scheduler.scheduleDirect(new Runnable() {
            @Override
            public void run() {
            //4 此时已经运行在相应的Scheduler 的线程中
                source.subscribe(parent);
            }
```

```
            }));
        }
```

和map套路大体一致，`ObservableSubscribeOn`自身同样是个包装类，同**样继承**`AbstractObservableWithUpstream`。
创建了一个`SubscribeOnObserver`类，该类按照套路，应该也是**实现了**`Observer`、`Disposable`**接口**的包装类，让我们看一下：

```
    static final class SubscribeOnObserver<T> extends AtomicReference<Disposable> implem
ents Observer<T>, Disposable {
        //真正的下游（终点）观察者
        final Observer<? super T> actual;
        //用于保存上游的Disposable，以便在自身dispose时，连同上游一起dispose
        final AtomicReference<Disposable> s;

        SubscribeOnObserver(Observer<? super T> actual) {
            this.actual = actual;
            this.s = new AtomicReference<Disposable>();
        }

        @Override
        public void onSubscribe(Disposable s) {
            //onSubscribe()方法由上游调用，传入Disposable。在本类中赋值给this.s，加入管理。
            DisposableHelper.setOnce(this.s, s);
        }

        //直接调用下游观察者的对应方法
        @Override
        public void onNext(T t) {
            actual.onNext(t);
        }
        @Override
        public void onError(Throwable t) {
            actual.onError(t);
        }
        @Override
        public void onComplete() {
            actual.onComplete();
        }

        //取消订阅时，连同上游Disposable一起取消
        @Override
        public void dispose() {
            DisposableHelper.dispose(s);
            DisposableHelper.dispose(this);
        }

        @Override
        public boolean isDisposed() {
            return DisposableHelper.isDisposed(get());
        }
        //这个方法在subscribeActual()中被手动调用，为了将Schedulers返回的Worker加入管理
        void setDisposable(Disposable d) {
            DisposableHelper.setOnce(this, d);
        }
    }
```

这两个类根据上一节的铺垫加上注释，其他都好理解，稍微不好理解的应该是下面两句代码：

```
        //ObservableSubscribeOn类
        //3 setDisposable()是为了将子线程的操作加入Disposable管理中
        parent.setDisposable(scheduler.scheduleDirect(new Runnable() {
            @Override
            public void run() {
            //4 此时已经运行在相应的Scheduler 的线程中
                source.subscribe(parent);
            }
        }));

```

```
        //SubscribeOnObserver类
        //这个方法在subscribeActual()中被手动调用，为了将Schedulers返回的Worker加入管理
        void setDisposable(Disposable d) {
            DisposableHelper.setOnce(this, d);
        }
```

其中`scheduler.scheduleDirect(new Runnable()..)`方法源码如下：

```
    /**
     * Schedules the given task on this scheduler non-delayed execution.
     * .....
     */
    public Disposable scheduleDirect(Runnable run) {
        return scheduleDirect(run, 0L, TimeUnit.NANOSECONDS);
    }
```

从注释和方法名我们可以看出，这个**传入的`Runnable`会立刻执行**。
再继续往里面看：

```
    public Disposable scheduleDirect(Runnable run, long delay, TimeUnit unit) {
        //class Worker implements Disposable ，Worker本身是实现了Disposable
        final Worker w = createWorker();
        //hook略过
        final Runnable decoratedRun = RxJavaPlugins.onSchedule(run);
        //开始在Worker的线程执行任务，
        w.schedule(new Runnable() {
            @Override
            public void run() {
                try {
                //调用的是 run()不是 start()方法执行的线程的方法。
                    decoratedRun.run();
                } finally {
                //执行完毕会 dispose()
                    w.dispose();
                }
            }
        }, delay, unit);
        //返回Worker对象
        return w;
    }
```

`createWorker()`是一个抽象方法，由具体的`Scheduler`类实现，例如`IoScheduler`对应的`Schedulers.io()`.

```
    public abstract Worker createWorker();
```

初看源码，为了了解大致流程，不宜过入深入，先点到为止。
OK，现在我们总结一下`scheduler.scheduleDirect(new Runnable()..)`的重点：

1. 传入的`Runnable`是**立刻执行**的。
2. 返回的**`Worker`对象**就是一个**`Disposable`对象**，
3. `Runnable`执行时，是直接手动**调用的 `run()`**，而不是 `start()`方法.
4. 上一点应该是为了，能控制在`run()`结束后(包括异常终止)，都会**自动执行`Worker.dispose()`**.

而返回的**`Worker`对象**也会被`parent.setDisposable(...)`加入管理中，以便在手动`dispose()`时能取消线程里的工作。

我们**总结**一下`subscribeOn(Schedulers.xxx())`的**过程**：

1. 返回一个`ObservableSubscribeOn`包装类对象
2. 上一步返回的对象**被订阅时**，回调该类中的`subscribeActual()`方法，在其中会**立刻将线程切换**到对应的`Schedulers.xxx()`线程。
3. **在切换后的线程中**，执行`source.subscribe(parent);`，**对上游(终点)`Observable`订阅**
4. **上游(终点)`Observable`开始发送数据**，根据RxJava2 源码解析（一），上游发送数据**仅仅是调用下游观察者对应的`onXXX()`方法而已**，所以此时**操作是在切换后的线程中进行**。

一点扩展，
大家可能看过一个结论：
`subscribeOn(Schedulers.xxx())`切换线程N次，总是**以第一次为准**，或者说**离源Observable最近的那次为准**，并且对其**上面的代码生效**（这一点对比的`ObserveOn()`）。

为什么？
- 因为根据RxJava2 源码解析（一）中提到，订阅流程**从下游往上游传递**
- 在`subscribeActual()`里开启了Scheduler的工作，`source.subscribe(parent);`,**从这一句开始切换了线程**，所以在**这之上的代码**都是在**切换后**的线程里的了。
- 但如果**连续切换**，**最上面的切换最晚执行**,此时线程变成了最上面的`subscribeOn(xxxx)`指定的线程，
- 而**数据push时，是从上游到下游的**，所以会在离源头最近的那次`subscribeOn(xxxx)`的线程里push数据（`onXXX()`）给下游。

可写如下代码验证：

```
Observable.create(new ObservableOnSubscribe<String>() {
                    @Override
                    public void subscribe(ObservableEmitter<String> e) throws Exception
{
                        Log.d(TAG, "subscribe() called with: e = [" + e + "]" + Thread.c
urrentThread());
                        e.onNext("1");
                        e.onComplete();
                    }
                }).subscribeOn(Schedulers.io())
                        .map(new Function<String, String>() {
                            @Override
                            public String apply(String s) throws Exception {
                                //依然是io线程
                                Log.d(TAG, "apply() called with: s = [" + s + "]" + Thre
ad.currentThread());
                                return s;
                            }
                        })
                        .subscribeOn(Schedulers.computation())
                        .subscribe(new Observer<String>() {
                            @Override
                            public void onSubscribe(Disposable d) {
                                Log.d(TAG, "onSubscribe() called with: d = [" + d +
"]");
                            }
                            @Override
                            public void onNext(String value) {
                                Log.d(TAG, "onNext() called with: value = [" + value +
"]");
                            }
                            @Override
                            public void onError(Throwable e) {
                                Log.d(TAG, "onError() called with: e = [" + e + "]");
                            }
                            @Override
                            public void onComplete() {
                                Log.d(TAG, "onComplete() called");
                            }
                        });
```

线程调度observeOn

在上一节的基础上，增加一个`observeOn(AndroidSchedulers.mainThread())`,就完成了观察者线程的切换。

```
                        .subscribeOn(Schedulers.computation())
                        //在上一节的基础上，增加一个ObserveOn
                        .observeOn(AndroidSchedulers.mainThread())
                        .subscribe(new Observer<String>() {
```

继续看源码吧，我已经能猜出来了，`hook+new XXXObservable();`

```
    public final Observable<T> observeOn(Scheduler scheduler) {
```

```
        return observeOn(scheduler, false, bufferSize());
    }

    public final Observable<T> observeOn(Scheduler scheduler, boolean delayError, int bu
fferSize) {
        ....
        return RxJavaPlugins.onAssembly(new ObservableObserveOn<T>(this, scheduler, dela
yError, bufferSize));
    }
```

果然，查看`ObservableObserveOn`,:
高能预警，这部分的代码 有些略多，建议读者打开源码边看边读。

```
public final class ObservableObserveOn<T> extends AbstractObservableWithUpstream<T, T> {
    //本例是 AndroidSchedulers.mainThread()
    final Scheduler scheduler;
    //默认false
    final boolean delayError;
    //默认128
    final int bufferSize;
    public ObservableObserveOn(ObservableSource<T> source, Scheduler scheduler, boolean
delayError, int bufferSize) {
        super(source);
        this.scheduler = scheduler;
        this.delayError = delayError;
        this.bufferSize = bufferSize;
    }

    @Override
    protected void subscribeActual(Observer<? super T> observer) {
        // false
        if (scheduler instanceof TrampolineScheduler) {
            source.subscribe(observer);
        } else {
            //1 创建出一个 主线程的Worker
            Scheduler.Worker w = scheduler.createWorker();
            //2 订阅上游数据源，
            source.subscribe(new ObserveOnObserver<T>(observer, w, delayError, bufferSiz
e));
        }
    }
```

本例中，就是两步：

1. 创建一个`AndroidSchedulers.mainThread()`对应的`Worker`
2. 用`ObserveOnObserver`订阅上游数据源。这样当数据从上游push下来，会由`ObserveOnObserver`对应的`onXXX()`处理。

```
static final class ObserveOnObserver<T> extends BasicIntQueueDisposable<T>
    implements Observer<T>, Runnable {
        //下游的观察者
        final Observer<? super T> actual;
        //对应Scheduler里的Worker
        final Scheduler.Worker worker;
        //上游被观察者 push 过来的数据都存在这里
        SimpleQueue<T> queue;
        Disposable s;
        //如果onError了，保存对应的异常
        Throwable error;
        //是否完成
        volatile boolean done;
        //是否取消
        volatile boolean cancelled;
        // 代表同步发送 异步发送
        int sourceMode;
        ....
        @Override
        public void onSubscribe(Disposable s) {
            if (DisposableHelper.validate(this.s, s)) {
```

```
                this.s = s;
                //省略大量无关代码
                //创建一个queue 用于保存上游 onNext() push的数据
                queue = new SpscLinkedArrayQueue<T>(bufferSize);
                //回调下游观察者onSubscribe方法
                actual.onSubscribe(this);
            }
        }

        @Override
        public void onNext(T t) {
            //1 执行过error / complete 会是true
            if (done) {
                return;
            }
            //2 如果数据源类型不是异步的， 默认不是
            if (sourceMode != QueueDisposable.ASYNC) {
                //3 将上游push过来的数据 加入 queue里
                queue.offer(t);
            }
            //4 开始进入对应Workder线程，在线程里 将queue里的t 取出 发送给下游Observer
            schedule();
        }

        @Override
        public void onError(Throwable t) {
            //已经done 会 抛异常 和 上一篇文章里提到的一样
            if (done) {
                RxJavaPlugins.onError(t);
                return;
            }
            //给error存个值
            error = t;
            done = true;
            //开始调度
            schedule();
        }

        @Override
        public void onComplete() {
        //已经done 会 返回  不会crash 和上一篇文章里提到的一样
            if (done) {
                return;
            }
            done = true;
            //开始调度
            schedule();
        }

        void schedule() {
            if (getAndIncrement() == 0) {
                //该方法需要传入一个线程， 注意看本类实现了Runnable的接口，所以查看对应的ru
n()方法
                worker.schedule(this);
            }
        }
        //从这里开始，这个方法已经是在Workder对应的线程里执行的了
        @Override
        public void run() {
            //默认是false
            if (outputFused) {
                drainFused();
            } else {
                //取出queue里的数据 发送
                drainNormal();
            }
        }


        void drainNormal() {
            int missed = 1;
```

```
93
94              final SimpleQueue<T> q = queue;
95              final Observer<? super T> a = actual;
96
97              for (;;) {
98                  // 1 如果已经 终止 或者queue空，则跳出函数，
99                  if (checkTerminated(done, q.isEmpty(), a)) {
100                     return;
101                 }
102
103                 for (;;) {
104                     boolean d = done;
105                     T v;
106
107                     try {
108                         //2 从queue里取出一个值
109                         v = q.poll();
110                     } catch (Throwable ex) {
111                         //3 异常处理 并跳出函数
112                         Exceptions.throwIfFatal(ex);
113                         s.dispose();
114                         q.clear();
115                         a.onError(ex);
116                         return;
117                     }
118                     boolean empty = v == null;
119                     //4 再次检查 是否 终止  如果满足条件 跳出函数
120                     if (checkTerminated(d, empty, a)) {
121                         return;
122                     }
123                     //5 上游还没结束数据发送，但是这边处理的队列已经是空的，不会push给下游 Ob
124 server
125                     if (empty) {
126                         //仅仅是结束这次循环，不发送这个数据而已，并不会跳出函数
127                         break;
128                     }
129                     //6 发送给下游了
130                     a.onNext(v);
131                 }
132
133                 //7 对不起这里我也不是很明白，大致猜测是用于 同步原子操作 如有人知道 烦请告知
134                 missed = addAndGet(-missed);
135                 if (missed == 0) {
136                     break;
137                 }
138             }
139         }
140
141         //检查 是否 已经 结束（error complete），  是否没数据要发送了(empty 空)，
142         boolean checkTerminated(boolean d, boolean empty, Observer<? super T> a) {
143             //如果已经disposed
144             if (cancelled) {
145                 queue.clear();
146                 return true;
147             }
148             // 如果已经结束
149             if (d) {
150                 Throwable e = error;
151                 //如果是延迟发送错误
152                 if (delayError) {
153                     //如果空
154                     if (empty) {
155                         if (e != null) {
156                             a.onError(e);
157                         } else {
158                             a.onComplete();
159                         }
160                         //停止worker（线程）
161                         worker.dispose();
162                         return true;
163                     }
```

```
                } else {
                    //发送错误
                    if (e != null) {
                        queue.clear();
                        a.onError(e);
                        worker.dispose();
                        return true;
                    } else
                    //发送complete
                    if (empty) {
                        a.onComplete();
                        worker.dispose();
                        return true;
                    }
                }
            }
            return false;
        }
    }
```

核心处都加了注释，总结起来就是，

1. `ObserveOnObserver`实现了`Observer`和`Runnable`接口。
2. 在`onNext()`里，先不切换线程，**将数据加入队列`queue`。然后**开始**切换线程**，在另一线程中，**从`queue`里取出数据，`push`给下游`Observer`**
3. `onError()` `onComplete()`除了和RxJava2 源码解析（一）提到的一样特性之外，也是将**错误/完成信息先保存，切换线程后再发送。**
4. 所以`observeOn()`影响的是其下游的代码，且多次调用仍然生效。
5. 因为其切**换线程代码**是在`Observer`里`onXXX()`做的，这是一个主动的push行为（**影响下游**）。
6. 关于**多次调用生效问题**。对比`subscribeOn()`切换线程是在`subscribeActual()`里做的，只是主动切换了上游的订阅线程，从而影响其发射数据时所在的线程。而**直到真正发射数据之前**，任何改变线程的行为，都会生效（影响发射数据的线程）。所以`subscribeOn()`只生效一次。`observeOn()`是一个主动的行为，并且切换线程后会立刻发送数据，所以会生效多次.

总结

本文带大家走读分析了三个东西:

map操作符原理：

- 内部**对上游`Observable`进行订阅**
- 内部订阅者**接收到数据后**，将**数据转换**，**发送给下游**`Observer`.
- **操作符返回的`Observable`和其内部订阅者**、是装饰者模式的体现。
- **操作符**数据变换的操作，也是**发生在订阅后**。

线程调度`subscribeOn()`：

- 内部**先切换线程**，在切换后的线程中**对上游**`Observable`进行**订阅**，这样上游发送数据时就是处于被切换后的线程里了。
- 也因此**多次切换线程**，**最后一次切换（离源数据最近）的生效**。
- 内部订阅者接收到数据后，**直接**发送给下游`Observer`.
- 引入内部订阅者是为了**控制线程**（dispose）
- **线程切换**发生在`Observable`中。

线程调度`observeOn()`:

- 使用装饰的`Observer`**对上游**`Observable`进行**订阅**
- 在`Observer`中**`onXXX()`方法里**，将**待发送数据存入队列**，同时**请求切换线程**处理**真正push数据**给下

游。

- **多次切换线程**，都会**对下游生效**。

源码里那些实现了`Runnable`的类或者匿名内部类，最终并没有像往常那样被丢给`Thread`类执行。
而是先切换线程，再直接执行`Runnable`的`run()`方法。
这也加深了我对面向对象，对抽象、 `Runnable`的理解，它就是一个简简单单的接口，里面就一个简简单单的`run()`，
我认为，之所以有`Runnable`，只是抽象出 一个可运行的任务的概念。
也许这句话很平淡，书上也会提到，各位大佬早就知道，但是如今我顺着RxJava2的源码这么走读了一遍，确真真切切的感受到了这些设计思想的美妙。
