RxJava2.X 源码分析（六）：变换操作符的实现原理（上）

作者：angels
原文链接：https://ld246.com/article/1500265551945
来源网站：链滴

- 更多分享： http://www.cherylgood.cn

一、前言

- Ok，RxJava用的最爽的当然还有我们的变换操作符了
- 看到这么好用的东西，估计你也有过想一探究竟的冲动，想看下内部是如何实现的
- 操作符的分析我打算是分成两篇讲解，先从简单的 map入手，当了解其本质后再分析强大的flatMap操作符

二、从Demo到源码

- 我们依然是以前的套路，先看一个demo

```
Observable observable = Observable.create(new ObservableOnSubscribe() {
    @Override
  public void subscribe(@NonNull ObservableEmitter emitter) throws Exception {
        emitter.onNext(1);
        emitter.onNext(2);
        emitter.onNext(3);

    }

});

observable.map(new Function() {
            @Override
  public Integer apply(@NonNull Integer integer) throws Exception {
                return integer*integer;
            }
        }).subscribe(new Consumer() {
    @Override
  public void accept(@NonNull Integer integer) throws Exception {
        Log.i(TAG, ">>>data is : " + integer);
    }
});
```

- 输出结果

```
07-17 09:34:52.123 3710-3729/? I/RxJavaDemo2: >>>data is : 1
07-17 09:34:52.123 3710-3729/? I/RxJavaDemo2: >>>data is : 4
07-17 09:34:52.123 3710-3729/? I/RxJavaDemo2: >>>data is : 9
```

- 当然，操作符map提供的能力肯定不止这样，你可以的 apply回调里面编写需要的逻辑代码。

三、源码分析

- OK，从demo中我们看到，经过map后，我们的结果跟我们的预期一样。

● 我们以map为切入点，看下内部都做了些什么呢

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

● 果然，还是熟悉的代码，变得只有 onAssembly参数里面的东西，这里可以注意一下T上游Observble下发的数据类型，R为下游Observer将要接收的数据类型，也就是说，暂时我们可以这样理解，T换为R，为什么这样说呢，因为到flatMap时，就不能这样简单的理解了

● Ok， RxJavaPlugins.onAssembly我们都知道啦，有关hook的，我们继续往下看

```
public final class ObservableMap<T, U> extends AbstractObservableWithUpstream<T, U> {

    final Function<? super T, ? extends U> function;

    public ObservableMap(ObservableSource<T> source, Function<? super T, ? extends U> f
nction) {
        //1、source 为上游的Observable
        super(source);
         //2、 function 为我们传入的funcation对象
        this.function = function;
    }

    @Override
   public void subscribeActual(Observer<? super U> t) {
        //3、t为下游的Observer对象
        source.subscribe(new MapObserver<T, U>(t, function));
    }

    static final class MapObserver<T, U> extends BasicFuseableObserver<T, U> {
        final Functionsuper T, ? extends U> mapper;

        MapObserver(Observer<? super U> actual, Functionsuper T, ? extends U> mapper) {
            //4、actual 为下游的Observer
            super(actual);
            //5、mapper为我们传入的function函数对象
            this.mapper = mapper;
        }

        @Override
   public void onNext(T t) {
            if (done) {
                return;
            }
            ...
            U v;

            try {
                //6、调用mapper的apply方法，或者apply回调的返回值
                v = ObjectHelper.requireNonNull(mapper.apply(t), "The mapper function returned
a null value.");
            } catch (Throwable ex) {
```

```
                fail(ex);
                return;
            }
            //7、回调下游Obsever的onNext方法
            actual.onNext(v);
        }

    .....
    }
}
```

- 按照流程应该是这样的:

下游Obsever.subscribe->触发ObservableMap.subscribeActual->在subscribeActual中通过中间bserver订阅上游Observable->1、上游Observable执行subscribeActual、2、执行中间Observer的nSubscribe ; 3、执行中间Obsever的onXXX方法下发数据。->中间Observer调用：1、下游的Obsever的onSubscribe 以及执行mapper回到后将apply的返回值传递给onXXX回调完成数据的转换级数的下发传递。

- 但是，我们目前没发现 MapObserver里面的onSubscribe方法，估计是在父类了

```
public abstract class BasicFuseableObserver<T, R> implements Observer<T>, QueueDisposable<R> {
...
public BasicFuseableObserver(Observersuper R> actual) {
        this.actual = actual;
    }


  @SuppressWarnings("unchecked")
   @Override
  public final void onSubscribe(Disposable s) {
        if (DisposableHelper.validate(this.s, s)) {
            //1、接收下游的Disposable加入管理队列
            this.s = s;
            if (s instanceof QueueDisposable) {
                this.qs = (QueueDisposable<T>)s;
            }
            //2、可以重写弘治onSubscribe()的回调
            if (beforeDownstream()) {

                actual.onSubscribe(this);
              //3、可重写在onSubscribe调用后做一些操作
                afterDownstream();
            }

        }
    }


  @Override
  public void onError(Throwable t) {
        if (done) {
            RxJavaPlugins.onError(t);
```

```
            return;
        }
        done = true;
        actual.onError(t);
    }

    @Override
  public void onComplete() {
        if (done) {
            return;
        }
        done = true;
        actual.onComplete();
    }
...
}
```

- Ok，果然在父类中抽取了一些公共的操作减少子类的代码量。

四、总结：

- Ok，根据上面的分析，其实对于map的操作过程我们已经很清楚了，其跟之前的线程切换的实现理基本一样，通过在中间使用装饰者模式插入一个中间的Observable和Observe，你可以想象为代理。
- 代理Observable做的事就是接收下游Obsever的订阅事件，然后通过代理Obsever订阅上游Observr，然后在上游Observer下发数据給代理Observer时，通过先调用mapper.apply转换回调函数获得换后的数据，然后下发给下游Obsever。
- Ok,其实就是这样，在RxJava2中大量运用装饰者模式来实现扩展功能。
- RxJava2的 flatMap高级转换函数我们将再下篇进行分析。
- 喜欢就给我留言哦，有好的建议也可以在下方留言。

五、相关文章

- RxJava2.X 源码解析（一）： 探索RxJava2分发订阅流程
- RxJava2.X 源码解析（二）：探索RxJava2神秘的随意取消订阅流程的原理
- RxJava2.X 源码分析（三）：探索RxJava2之订阅线程切换原理
- RxJava2.X 源码分析（四）探索RxJava2之观察者线程切换原理
- RxJava2.X 源码分析（五）：论RxJava2.X切换线程次数的有效性