# [谜之RxJava （三）update —— 线程切换](https://segmentfault.com/a/1190000004855661)

在RxJava中，我们只要调用调用  
subscribOn()和observeOn()就能切换我们的工作线程，是不是让小伙伴都惊呆了？

然后结合RxJava的Operator，写异步的时候，想切换线程就是一行代码的事情，整个workflow还非常清晰：

Observable.create()

// do something on io thread  
 .work() // work.. work..  
 .subscribeOn(Schedulers.io())

// observeOn android main thread  
 .observeOn(AndroidSchedulers.mainThread())  
 .subscribe();

我们再也不用去写什么见鬼的new Thread和Handler了，在这么几行代码里，我们实现了在io线程上做我们的工作(work)，在main线程上，更新UI

## Subscribe On

先看下subscribeOn干了什么

public final Observable<T> subscribeOn(Scheduler scheduler) {  
 if (this instanceof ScalarSynchronousObservable) {  
 return ((ScalarSynchronousObservable<T>)this).scalarScheduleOn(scheduler);  
 }  
 return nest().lift(new OperatorSubscribeOn<T>(scheduler));  
}

啊，原来也是个lift，就是从一个Observable生成另外一个Observable咯，这个nest是干嘛用？

public final Observable<Observable<T>> nest() {  
 return just(this);  
}

这里返回类型告诉我们，它产生一个Observable<Observable<T>>  
讲到这里，会有点晕，先记着这个，然后我们看OperatorSubscribeOn这个操作符,

构造函数是

public OperatorSubscribeOn(Scheduler scheduler) {  
 this.scheduler = scheduler;  
}

OK，这里保存了scheduler对象，然后就是我们前一章说过的转换方法。

@Override  
public Subscriber<? super Observable<T>> call(final Subscriber<? super T> subscriber) {  
 final Worker inner = scheduler.createWorker();  
 subscriber.add(inner);  
 return new Subscriber<Observable<T>>(subscriber) {  
  
 @Override  
 public void onCompleted() {  
 // ignore because this is a nested Observable and we expect only 1 Observable<T> emitted to onNext  
 }  
  
 @Override  
 public void onError(Throwable e) {  
 subscriber.onError(e);  
 }  
  
 @Override  
 public void onNext(final Observable<T> o) {  
 inner.schedule(new Action0() {  
  
 @Override  
 public void call() {  
 final Thread t = Thread.*currentThread*();  
 o.unsafeSubscribe(new Subscriber<T>(subscriber) {  
  
 @Override  
 public void onCompleted() {  
 subscriber.onCompleted();  
 }  
  
 @Override  
 public void onError(Throwable e) {  
 subscriber.onError(e);  
 }  
  
 @Override  
 public void onNext(T t) {  
 subscriber.onNext(t);  
 }  
  
 @Override  
 public void setProducer(final Producer producer) {  
 subscriber.setProducer(new Producer() {  
  
 @Override  
 public void request(final long n) {  
 if (Thread.*currentThread*() == t) {  
 // don't schedule if we're already on the thread

//(primarily for first setProducer call)  
 // see unit test 'testSetProducerSynchronousRequest'

//for more context on this  
 producer.request(n);  
 } else {  
 inner.schedule(new Action0() {  
  
 @Override  
 public void call() {  
 producer.request(n);  
 }  
 });  
 }  
 }  
  
 });  
 }  
  
 });  
 }  
 });  
 }  
  
 };  
}

## 让人纠结的类模板

看完这段又臭又长的，先深呼吸一口气，我们慢慢分析下。  
首先要注意RxJava里面最让人头疼的模板问题，那么OperatorMap这个类的声明是

public final class OperatorMap<T, R> implements Operator<R, T>

而Operator这个接口继承Func1

public interface Func1<T, R> extends Function {  
 R call(T t);  
}

我们这里不要记T和R，记住传入左边的模板是形参，传入右边的模板是返回值。

好了，那么这里的call就是从一个T转换成一个Observable<T>的过程了。

总结一下，我们这一次调用subscribeOn，做了两件事

1、nest() 为Observable<T>生成了一个Observable<Observable<T>>   
2、lift() 对Observalbe<Observalbe<T>>进行一个变化，变回Observable<T>

因为lift是一个模板函数，它的返回值的类型是参照它的形参来，而他的形参是Operator<T, Observable<T>> 这个结论非常重要！！  
OK，到这里我们已经存储了所有的序列，等着我们调用了。

## 调用链

首先，记录我们在调用这条指令之前的Observable<T>，记为Observable$1  
然后，经过lift生成的Observable<T>记为Observable$2

好了，现在我们拿到的依然是Observable<T>这个对象，但是它****不是****原始的Observable$1，要深深记住这一点，它是由lift生成的Observable$2，这时候进行subscribe，那看到首先调用的就是OnSubscribe.call方法，好，直接进入lift当中生成的那个地方。

我们知道这一层lift的operator就是刚刚的OperatorSubscribOn，那么调用它的call方法，生成的是一个Subscriber<Observable<T>>

Subscriber<? super T> st = hook.onLift(operator).call(o);try {  
 // new Subscriber created and being subscribed with so 'onStart' it  
 st.onStart();  
 onSubscribe.call(st);  
} catch (Throwable e) {  
 ...  
}

好，还记得我们调用过nest么？，这里的onSubscribe可是nest上下文中的噢，每一次，到这个地方，这个onSubscribe就是上一层Observable的onSubscribe，即Observable<Observable<T>>的onSubscribe，相当于栈弹出了一层。它的call直接在Subscriber的onNext中给出了最开始的Observable<T>，我们这里就要看下刚刚在OperatorSubscribeOn中生成的Subscriber

new Subscriber<Observable<T>>(subscriber) {  
  
 @Override  
 public void onCompleted() {  
 // ignore because this is a nested Observable and we expect only 1 Observable<T> emitted to onNext  
 }  
  
 @Override  
 public void onError(Throwable e) {  
 subscriber.onError(e);  
 }  
  
 @Override  
 public void onNext(final Observable<T> o) {  
 inner.schedule(new Action0() {  
  
 @Override  
 public void call() {  
 final Thread t = Thread.*currentThread*();  
 o.unsafeSubscribe(new Subscriber<T>(subscriber) {  
  
 @Override  
 public void onCompleted() {  
 subscriber.onCompleted();  
 }  
  
 @Override  
 public void onError(Throwable e) {  
 subscriber.onError(e);  
 }  
  
 @Override  
 public void onNext(T t) {  
 subscriber.onNext(t);  
 }  
 });  
 }  
 });  
 }  
}

对，就是它，这里要注意，这里的subscriber就是我们在lift中，传入的o

Subscriber<? super T> st = hook.onLift(operator).call(o);

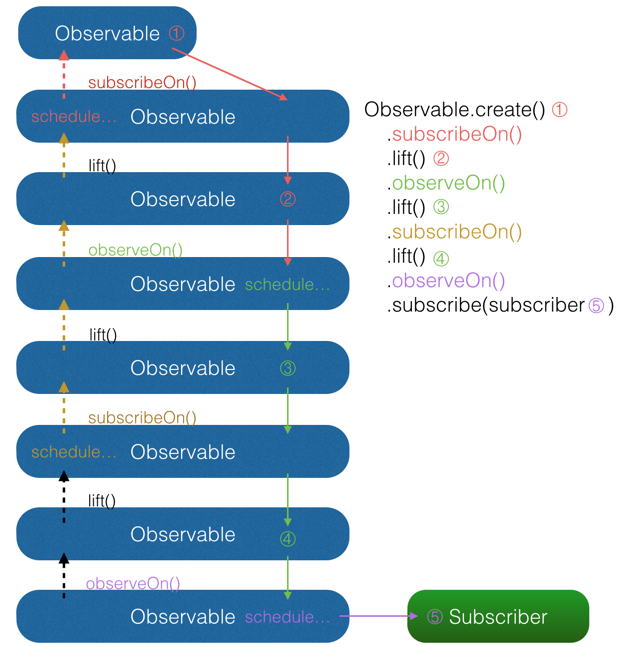
对，就是它，其实它就是SafeSubscriber。

回过头，看看刚刚的onNext()方法，inner.schedule() 这个函数，我们可以认为就是postRun()类似的方法，而onNext()中传入的o是我们之前生成的Observable$1，是从Observable.just封装出来的Observable<Observable<T>>中产生的，这里调用了Observable$1.unsafeSubscribe方法，我们暂时不关心它和subscribe有什么不同，但是我们知道最终功能是一样的就好了。

注意它运行时的线程！！在inner这个Worker上！于是它的运行线程已经被改了！！

好，这里的unsafeSubscribe调用的方法就是调用原先Observable$1.onSubscribe中的call方法：  
这个Observable$1就是我们之前自己定义的Observable了。

综上所述，如果我们需要我们的Observable$1在一个别的线程上运行的时候，只需要在后面跟一个subscribeOn即可。

结合扔物线大大的图如下：  


在RxJava更新版本后，OperatorSubscribeOn这个接口进行了一个重构，变换方式从一个比较难理解的递归嵌套的Observable<Observable<T>>上做一个Operator改成了从OnSubscribe角度上进行了一层封装。

从类型来说，OperatorSubscribeOn脱离了Operator的概念，变身成了OnSubscribe。

我们来比对下吧~

老版本的核心实现:

@Override  
public Subscriber<? super Observable<T>> call(final Subscriber<? super T> subscriber) {  
 final Worker inner = scheduler.createWorker();  
 subscriber.add(inner);  
 return new Subscriber<Observable<T>>(subscriber) {  
  
 @Override  
 public void onCompleted() {  
 // ignore because this is a nested Observable   
 // and we expect only 1 Observable<T> emitted to onNext  
 }  
  
 @Override  
 public void onError(Throwable e) {  
 subscriber.onError(e);  
 }  
  
 @Override  
 public void onNext(final Observable<T> o) {  
 inner.schedule(new Action0() {  
  
 @Override  
 public void call() {  
 final Thread t = Thread.*currentThread*();  
 o.unsafeSubscribe(new Subscriber<T>(subscriber) {  
  
 @Override  
 public void onCompleted() {  
 subscriber.onCompleted();  
 }  
  
 @Override  
 public void onError(Throwable e) {  
 subscriber.onError(e);  
 }  
  
 @Override  
 public void onNext(T t) {  
 subscriber.onNext(t);  
 }  
  
 @Override  
 public void setProducer(final Producer producer) {  
 subscriber.setProducer(new Producer() {  
  
 @Override  
 public void request(final long n) {  
 if (Thread.*currentThread*() == t) {  
 // don't schedule if we're already on the thread   
 // (primarily for first setProducer call)  
 // see unit test 'testSetProducerSynchronousRequest'   
 // for more context on this  
 producer.request(n);  
 } else {  
 inner.schedule(new Action0() {  
  
 @Override  
 public void call() {  
 producer.request(n);  
 }  
 });  
 }  
 }  
  
 });  
 }  
  
 });  
 }  
 });  
 }  
  
 };  
}

操作符，核心是把一个Subscriber转换成另外一个Subscriber

再看看新版实现

@Override  
public void call(final Subscriber<? super T> subscriber) {  
 final Worker inner = scheduler.createWorker();  
 subscriber.add(inner);  
  
 inner.schedule(new Action0() {  
 @Override  
 public void call() {  
 final Thread t = Thread.*currentThread*();  
  
 Subscriber<T> s = new Subscriber<T>(subscriber) {  
 @Override  
 public void onNext(T t) {  
 subscriber.onNext(t);  
 }  
  
 @Override  
 public void onError(Throwable e) {  
 try {  
 subscriber.onError(e);  
 } finally {  
 inner.unsubscribe();  
 }  
 }  
  
 @Override  
 public void onCompleted() {  
 try {  
 subscriber.onCompleted();  
 } finally {  
 inner.unsubscribe();  
 }  
 }  
  
 @Override  
 public void setProducer(final Producer p) {  
 subscriber.setProducer(new Producer() {  
 @Override  
 public void request(final long n) {  
 if (t == Thread.*currentThread*()) {  
 p.request(n);  
 } else {  
 inner.schedule(new Action0() {  
 @Override  
 public void call() {  
 p.request(n);  
 }  
 });  
 }  
 }  
 });  
 }  
 };  
  
 source.unsafeSubscribe(s);  
 }  
 });  
}

这里实现的是OnSubscribe接口，我们知道，OnSubscribe是Observable真正执行的代码段。

在新的接口重构后，唯一的不同是，在它里面需要存一个指向原始Observable的source变量。 而在老接口中，变换前的Observable是通过Observable<Observable>传进来的。