RxJava2 浅析 - 泡在网上的日子

泡在网上的日子 / 文 发表于2016-09-07 14:21 第3578次阅读 RxJava
4
编辑推荐：稀土掘金，这是一个针对技术开发者的一个应用，你可以在掘金上获取最新最优质的技术干货，不仅仅是Android知识、前端、后端以至于产品和设计都有涉猎，想成为全栈工程师的朋友不要错过！

原文地址：http://blog.csdn.net/maplejaw_/article/details/52442065

Observable

在RxJava1.x中，最熟悉的莫过于Observable这个类了，笔者刚使用RxJava2.x时，创建一个Observable后，顿时是懵逼的。因为我们熟悉的Subscriber居然没影了，取而代之的是ObservableEmitter,俗称发射器。此外，由于没有了Subscriber的踪影，我们创建观察者时需使用Observer。而Observer也不是我们熟悉的那个Observer,其回调的Disposable参数更是让人摸不到头脑。

废话不多说，从会用开始，还记得使用RxJava的三部曲吗？
第一步：初始化一个Observable

```
1.              Observable<Integer>  observable=Observable.create(new  ObservableOnSubsc
   ribe<Integer>()  {
2.                      @Override
3.                      public  void  subscribe(ObservableEmitter<Integer>  e)  throws
     Exception  {
4.                              e.onNext(1);
5.                              e.onNext(2);
6.                              e.onComplete();
7.                      }
8.              });
```

第二步：初始化一个Observer

```
1.              Observer<Integer>  observer=  new  Observer<Integer>()  {
2.                      @Override
3.                      public  void  onSubscribe(Disposable  d)  {
4.                      }
5.                      @Override
6.                      public  void  onNext(Integer  value)  {
7.                      }
8.                      @Override
9.                      public  void  onError(Throwable  e)  {
```

```
10.                         }
11.                         @Override
12.                         public  void  onComplete()  {
13.                         }
14.                 }
```

第三部：建立订阅关系

```
1.          observable.subscribe(observer);  //建立订阅关系
```

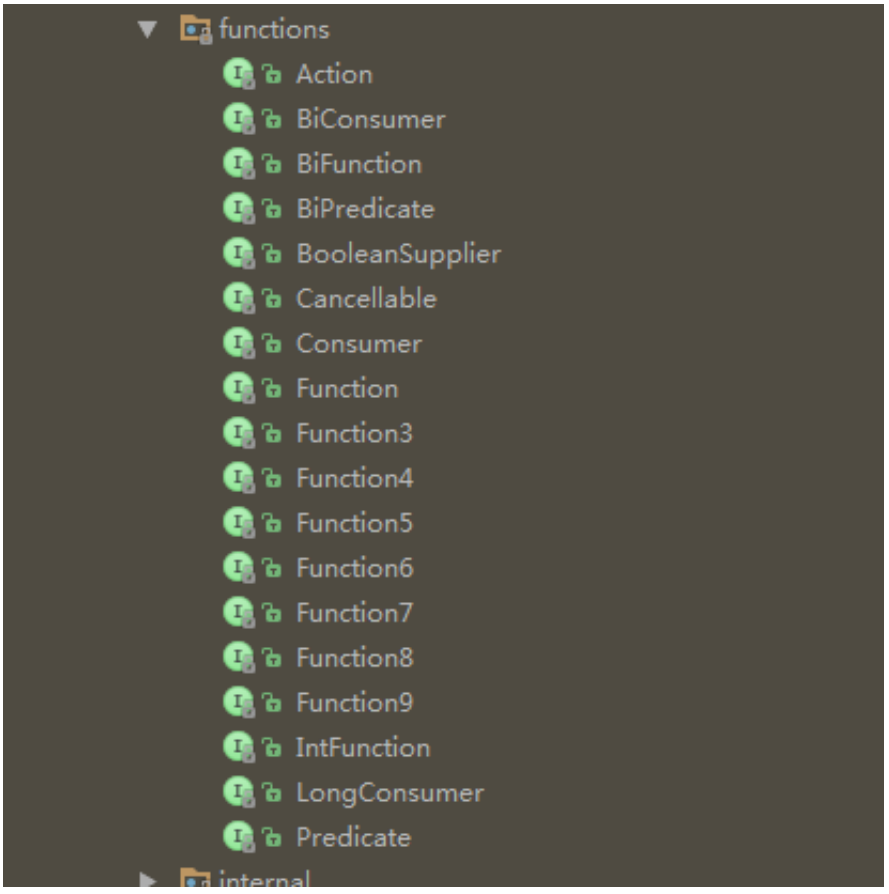
不难看出，与RxJava1.x还是存在着一些区别的。首先，创建Observable时，回调的是ObservableEmitter,字面意思即发射器，用于发射数据（onNext）和通知（onError/onComplete）。其次，创建的Observer中多了一个回调方法onSubscribe，传递参数为Disposable ，Disposable相当于RxJava1.x中的Subscription,用于解除订阅。你可能纳闷为什么不像RxJava1.x中订阅时返回Disposable，而是选择回调出来呢。官方说是为了设计成Reactive-Streams架构。不过仔细想想这么一个场景还是很有用的，假设Observer需要在接收到异常数据项时解除订阅，在RxJava2.x中则非常简便，如下操作即可。

```
1.     Observer<Integer>  observer  =  new  Observer<Integer>()  {
2.                         private  Disposable  disposable;
3.                         @Override
4.                         public  void  onSubscribe(Disposable  d)  {
5.                                 disposable  =  d;
6.                         }
7.                         @Override
8.                         public  void  onNext(Integer  value)  {
9.                                 Log.d("JG",  value.toString());
10.                                if  (value  >  3)  {      //  >3  时为异常数据，解除订
    阅
11.                                        disposable.dispose();
12.                                }
13.                        }
14.                        @Override
15.                        public  void  onError(Throwable  e)  {
16.                        }
17.                        @Override
18.                        public  void  onComplete()  {
19.                        }
20.                };
```

此外，RxJava2.x中仍然保留了其他简化订阅方法，我们可以根据需求，选择相应的简化订阅。只不过传入的对象改为了Consumer。`

```
1.        Disposable  disposable  =  observable.subscribe(new  Consumer<Integer>()  {
2.                            @Override
3.                            public  void  accept(Integer  integer)  throws  Exception  {
4.                                    //这里接收数据项
5.                            }
6.                    },  new  Consumer<Throwable>()  {
7.                            @Override
8.                            public  void  accept(Throwable  throwable)  throws  Exception
      {
9.                                //这里接收onError
10.                           }
11.                   },  new  Action()  {
12.                           @Override
13.                           public  void  run()  throws  Exception  {
14.                               //这里接收onComplete。
15.                           }
16.                   });
```

不同于RxJava1.x，RxJava2.x中没有了一系列的Action/Func接口，取而代之的是与Java8命名类似的函数式接口，如下图：



其中Action类似于RxJava1.x中的Action0,区别在于Action允许抛出异常。

```
1. public  interface  Action  {
```

```
2.          /**
3.            * Runs the action and optionally throws a checked exception
4.            * @throws Exception if the implementation wishes to throw a checke
   d exception
5.            */
6.          void run() throws Exception;
7. }
```

而Consumer即消费者，用于接收单个值，BiConsumer则是接收两个值，Function用于变换对象，Predicate用于判断。这些接口命名大多参照了Java8，熟悉Java8新特性的应该都知道意思，这里也就不再赘述了。

线程调度

关于线程切换这点，RxJava1.x和RxJava2.x的实现思路是一样的。这里就简单看下相关源码。

subscribeOn

同RxJava1.x一样，subscribeOn用于指定subscribe()时所发生的线程,从源码角度可以看出，内部线程调度是通过ObservableSubscribeOn来实现的。

```
1.        public final Observable<T> subscribeOn(Scheduler scheduler) {
2.                ObjectHelper.requireNonNull(scheduler, "scheduler is null");
3.                return RxJavaPlugins.onAssembly(new ObservableSubscribeOn<T>
   (this, scheduler));
4.          }
```

ObservableSubscribeOn的核心源码在subscribeActual方法中，通过代理的方式使用SubscribeOnObserver包装Observer后，设置Disposable来将subscribe切换到Scheduler线程中

```
1.          @Override
2.          public void subscribeActual(final Observer<? super T> s) {
3.                  final SubscribeOnObserver<T> parent = new SubscribeOnObserver<T>
   (s);
4.                  s.onSubscribe(parent);  //回调Disposable
5.                  parent.setDisposable(scheduler.scheduleDirect(new Runnable() { //设
   置`Disposable`
6.                          @Override
7.                          public void run() {
8.                                  source.subscribe(parent);  //使Observable的subscribe发
   生在Scheduler线程中
9.                          }
```

```
10.                 }));
11.         }
```

observeOn

observeOn方法用于指定下游Observer回调发生的线程。

```
1.         public final Observable<T> observeOn(Scheduler scheduler, boolean delayE
   rror, int bufferSize) {
2.                 //..
3.                 //验证安全
4.                 return RxJavaPlugins.onAssembly(new ObservableObserveOn<T>
   (this, scheduler, delayError, bufferSize));
5.         }
```

主要实现在ObservableObserveOn中的subscribeActual,可以看出，不同于subscribeOn,没有将suncribe操作全部切换到Scheduler中，而是通过ObserveOnSubscriber与Scheduler配合，通过schedule()达到切换下游Observer回调发生的线程，这一点与RxJava1.x实现几乎相同。关于ObserveOnSubscriber的源码这里不再重复描述了，有兴趣的可以查看本人RxJava源码解读这篇文章

```
1.         @Override
2.         protected void subscribeActual(Observer<? super T> observer) {
3.                 if (scheduler instanceof TrampolineScheduler) {
4.                         source.subscribe(observer);
5.                 } else {
6.                         Scheduler.Worker w = scheduler.createWorker();
7.                         source.subscribe(new ObserveOnSubscriber<T>
   (observer, w, delayError, bufferSize));
8.                 }
9.         }
```

Flowable

Flowable是RxJava2.x中新增的类，专门用于应对背压（Backpressure）问题，但这并不是RxJava2.x中新引入的概念。所谓背压，即生产者的速度大于消费者的速度带来的问题，比如在Android中常见的点击事件，点击过快则会造成点击两次的效果。
我们知道，在RxJava1.x中背压控制是由Observable完成的，使用如下：

```
1.     Observable.range(1,10000)
2.                         .onBackpressureDrop()
3.                         .subscribe(integer -> Log.d("JG",integer.toString()));
```

而在RxJava2.x中将其独立了出来，取名为Flowable。因此，原先的Observable已经不具备背压处理能力。
通过Flowable我们可以自定义背压处理策略。

```
enum BackpressureMode {
    /**
     * OnNext events are written without any buffering or dropping.
     * Downstream has to deal with any overflow.
     * <p>Useful when one applies one of the custom-parameter onBackpressureXXX operators.
     */
    NONE,
    /**
     * Signals a MissingBackpressureException in case the downstream can't keep up.
     */
    ERROR,
    /**
     * Buffers <em>all</em> onNext values until the downstream consumes it.
     */
    BUFFER,
    /**
     * Drops the most recent onNext value if the downstream can't keep up.
     */
    DROP,
    /**
     * Keeps only the latest onNext value, overwriting any previous value if the
     * downstream can't keep up.
     */
    LATEST
}
```

测试Flowable例子如下：

```
1.      Flowable.create(new  FlowableOnSubscribe<Integer>()  {
2.                          @Override
3.                          public  void  subscribe(FlowableEmitter<Integer>  e)  throws
    Exception  {
4.                                  for(int  i=0;i<10000;i++){
5.                                          e.onNext(i);
6.                                  }
7.                                  e.onComplete();
8.                          }
9.                  },  FlowableEmitter.BackpressureMode.ERROR)  //指定背压处理策略，抛出
    异常
10.                                 .subscribeOn(Schedulers.computation())
11.                                 .observeOn(Schedulers.newThread())
12.                                 .subscribe(new  Consumer<Integer>()  {
13.                                         @Override
14.                                         public  void  accept(Integer  integer)  throws
      Exception  {
15.                                                 Log.d("JG",  integer.toString());
16.                                                 Thread.sleep(1000);
17.                                         }
18.                                 },  new  Consumer<Throwable>()  {
```

```
                                        @Override
                                        public void accept(Throwable throwable) th
    rows Exception {
                                                Log.d("JG",throwable.toString());
                                        }
                                });
```

或者可以使用类似RxJava1.x的方式来控制。

```
    Flowable.range(1,10000)
                                .onBackpressureDrop()
                                .subscribe(integer -
    > Log.d("JG",integer.toString()));
```

其中还需要注意的一点在于，Flowable并不是订阅就开始发送数据，而是需等到执行Subscription#request才能开始发送数据。当然，使用简化subscribe订阅方法会默认指定Long.MAX_VALUE。手动指定的例子如下：

```
                Flowable.range(1,10).subscribe(new Subscriber<Integer>() {
                        @Override
                        public void onSubscribe(Subscription s) {
                                s.request(Long.MAX_VALUE);//设置请求数
                        }
                        @Override
                        public void onNext(Integer integer) {
                        }
                        @Override
                        public void onError(Throwable t) {
                        }
                        @Override
                        public void onComplete() {
                        }
                });
```

Single

不同于RxJava1.x中的SingleSubscriber,RxJava2中的SingleObserver多了一个回调方法onSubscribe。

```
interface SingleObserver<T> {
        void onSubscribe(Disposable d);
        void onSuccess(T value);
```

```
4.          void  onError(Throwable  error);
5. }
```

Completable

同Single，Completable也被重新设计为Reactive-Streams架构，RxJava1.x的CompletableSubscriber改为CompletableObserver，源码如下：

```
1. interface  CompletableObserver<T>  {
2.          void  onSubscribe(Disposable  d);
3.          void  onComplete();
4.          void  onError(Throwable  error);
5. }
```

Subject/Processor

Processor和Subject的作用是相同的。关于Subject部分，RxJava1.x与RxJava2.x在用法上没有显著区别，这里就不介绍了。其中Processor是RxJava2.x新增的，继承自Flowable,所以支持背压控制。而Subject则不支持背压控制。使用如下：

```
1.                  //Subject
2.                  AsyncSubject<String>  subject  =  AsyncSubject.create();
3.                  subject.subscribe(o  ->  Log.d("JG",o));//three
4.                  subject.onNext("one");
5.                  subject.onNext("two");
6.                  subject.onNext("three");
7.                  subject.onComplete();
8.                 //Processor
9.                  AsyncProcessor<String>  processor  =  AsyncProcessor.create();
10.                 processor.subscribe(o  ->  Log.d("JG",o));  //three
11.                 processor.onNext("one");
12.                 processor.onNext("two");
13.                 processor.onNext("three");
14.                 processor.onComplete();
```

操作符

关于操作符，RxJava1.x与RxJava2.x在命名和行为上大多数保持了一致，部分操作符请查阅文档。

最后

RxJava1.x 如何平滑升级到RxJava2.x？
由于RxJava2.x变化较大无法直接升级，幸运的是，官方提供了RxJava2Interop这个库，可以方便地将RxJava1.x升级到RxJava2.x，或者将RxJava2.x转回RxJava1.x。地址：https://github.com/akarnokd/RxJava2Interop