

RxJava2 源码解析（一）

原创

2017年03月12日 19:42:21

标签：源码解析 / RxJava / Android / Observable / Observer

5651

转载请标明出处：

<http://blog.csdn.net/zxt0601/article/details/61614799>本文出自：【张旭童的博客】(<http://blog.csdn.net/zxt0601>)

概述

最近事情太多了，现在公司内部变动，自己岗位的变化，以及最近决定找工作。所以博客耽误了，准备面试中，打算看一看RxJava2的源码，遂有了这篇文章。

不会对RxJava2的源码逐字逐句的阅读，只寻找关键处，我们平时接触得到的那些代码。

背压实际中接触较少，故只分析了 `Observable`。

分析的源码版本为：**2.0.1**

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

本文先达到目的1，2，3。

我个人认为主要还是适配器模式的体现，我们接触的就只有 `Observable` 和 `Observer`，其实内部有大量的中间对象在适配：将它们两联系起来，加入一些额外功能，例如考虑dispose和hook等。

从create开始。

这是一段不涉及操作符和线程切换的简单例子：

```
1      Observable.create(new ObservableOnSubscribe<String>() {
2          @Override
3          public void subscribe(Observer<String> e) throws Exception {
4              e.onNext("1");
5              e.onComplete();
6          }
7      }).subscribe(new Observer<String>() {
8          @Override
9          public void onSubscribe(Disposable d) {
10             Log.d(TAG, "onSubscribe() called with: d = [" + d + "]);
11         }
12
13         @Override
14         public void onNext(String value) {
15             Log.d(TAG, "onNext() called with: value = [" + value + "]);
16         }
17
18         @Override
19         public void onError(Throwable e) {
20             Log.d(TAG, "onError() called with: e = [" + e + "]);
21         }
22
23         @Override
24         public void onComplete() {
25             Log.d(TAG, "onComplete() called");
26         }
27     });
```

```
27         });
```

拿 create来说,

```
1 public static <T> Observable<T> create(ObservableOnSubscribe<T> source) {
2     //.....
3     return RxJavaPlugins.onAssembly(new ObservableCreate<T>(source));
4 }
```

返回值是 `Observable`, 参数是 `ObservableOnSubscribe`, 定义如下:

```
1 public interface ObservableOnSubscribe<T> {
2     void subscribe(ObservableEmitter<T> e) throws Exception;
3 }
```

`ObservableOnSubscribe` 是一个接口, 里面就一个方法, 也是我们实现的那个方法:

该方法参数是 `ObservableEmitter`, 我认为它是关联起 `Disposable` 概念的一层:

```
1 public interface ObservableEmitter<T> extends Emitter<T> {
2     void setDisposable(Disposable d);
3     void setCancellable(Cancellable c);
4     boolean isDisposed();
5     ObservableEmitter<T> serialize();
6 }
```

`ObservableEmitter` 也是一个接口。里面方法很多, 它也继承了 `Emitter<T>` 接口。

```
1 public interface Emitter<T> {
2     void onNext(T value);
3     void onError(Throwable error);
4     void onComplete();
5 }
```

`Emitter<T>` 定义了我们在 `ObservableOnSubscribe` 中实现 `subscribe()` 方法里最常用的三个方法。

好, 我们回到原点, `create()` 方法里就一句话 `return RxJavaPlugins.onAssembly(new ObservableCreate<T>(source));`, 其中提到 `RxJavaPlugins.onAssembly()`:

```
1 /**
2  * Calls the associated hook function.
3  * @param <T> the value type
4  * @param source the hook's input value
5  * @return the value returned by the hook
6  */
7 @SuppressWarnings({ "rawtypes", "unchecked" })
8 public static <T> Observable<T> onAssembly(Observable<T> source) {
9     Function<Observable, Observable> f = onObservableAssembly;
10     if (f != null) {
11         return apply(f, source);
12     }
13     return source;
14 }
```

可以看到这是一个关于hook的方法, 关于hook我们暂且不表, 不影响主流程, 我们默认使用中都没有hook, 所以这里就是直接返回 `source`, 即传入的对象, 也就是 `new ObservableCreate<T>(source)`。

`ObservableCreate` 我认为算是一种适配器的体现, `create()` 需要返回的是 `Observable`, 而我现在有的是 (方法传入的是) `ObservableOnSubscribe` 对象, `ObservableCreate` 将 `ObservableOnSubscribe` 适配成 `Observable`。

其中 `subscribeActual()` 方法表示的是被订阅时真正被执行的方法, 放后面解析:

```
1 public final class ObservableCreate<T> extends Observable<T> {
2     final ObservableOnSubscribe<T> source;
3     public ObservableCreate(ObservableOnSubscribe<T> source) {
4         this.source = source;
5     }
6     @Override
```

```

7     protected void subscribeActual(Observer<? super T> observer) {
8         CreateEmitter<T> parent = new CreateEmitter<T>(observer);
9         observer.onSubscribe(parent);
10        try {
11            source.subscribe(parent);
12        } catch (Throwable ex) {
13            Exceptions.throwIfFatal(ex);
14            parent.onError(ex);
15        }
16    }

```

OK,至此，创建流程结束，我们得到了`Observable<T>`对象，其实就是`ObservableCreate<T>`。

到订阅subscribe 结束

`subscribe()`:

```

1     public final void subscribe(Observer<? super T> observer) {
2         ...
3         try {
4             //1 hook相关，略过
5             observer = RxJavaPlugins.onSubscribe(this, observer);
6             ...
7             //2 真正的订阅处
8             subscribeActual(observer);
9         } catch (NullPointerException e) { // NOPMD
10            throw e;
11        } catch (Throwable e) {
12            //3 错误处理，
13            Exceptions.throwIfFatal(e);
14            // can't call onError because no way to know if a Disposable has been set or
15            not
16            // can't call onSubscribe because the call might have set a Subscription already
17            ready
18            //4 hook错误相关，略过
19            RxJavaPlugins.onError(e);
20
21            NullPointerException npe = new NullPointerException("Actually not, but can't
22            throw other exceptions due to RS");
23            npe.initCause(e);
24            throw npe;
25        }
26    }

```

关于hook的代码：

可以看到如果没有hook，即相应的对象是null，则是传入什么返回什么的。

```

1     /**
2      * Calls the associated hook function.
3      * @param <T> the value type
4      * @param source the hook's input value
5      * @param observer the observer
6      * @return the value returned by the hook
7      */
8     @SuppressWarnings({ "rawtypes", "unchecked" })
9     public static <T> Observer<? super T> onSubscribe(Observable<T> source, Observer<? super
10     T> observer) {
11         //1 默认onObservableSubscribe（可理解为一个flatMap的操作）是null
12         BiFunction<Observable, Observer, Observer> f = onObservableSubscribe;
13         //2 所以这句跳过，不会对其进行apply
14         if (f != null) {
15             return apply(f, source, observer);
16         }
17         //3 返回参数2
18         return observer;
19     }

```

我也是验证了一下 三个Hook相关的变量，确实是null：

```

1      Consumer<Throwable> errorHandler = RxJavaPlugins.getErrorHandler();
2      BiFunction<Observable, Observer, Observer> onObservableSubscribe = RxJavaPlugin
3      s.getOnObservableSubscribe();
4      Function<Observable, Observable> onObservableAssembly = RxJavaPlugins.getOnObser
5      vableAssembly();
6
7      Log.e(TAG, "errorHandler = [" + errorHandler + "]");
      Log.e(TAG, "onObservableSubscribe = [" + onObservableSubscribe + "]");
      Log.e(TAG, "onObservableAssembly = [" + onObservableAssembly + "]");

```

所以订阅时的重点就是：

```

1      //2 真正的订阅处
2      subscribeActual(observer);

```

我们将第一节提到的 `ObservableCreate` 里的 `subscribeActual()` 方法拿出来看看：

```

1      @Override
2      protected void subscribeActual(Observer<? super T> observer) {
3          //1 创建CreateEmitter，也是一个适配器
4          CreateEmitter<T> parent = new CreateEmitter<T>(observer);
5          //2 onSubscribe () 参数是Disposable，所以CreateEmitter可以将Observer->Disposable
6          。还有一点要注意的是`onSubscribe()`是在我们执行`subscribe()`这句代码的那个线程回调的，并不受线
7          程调度影响。
8          observer.onSubscribe(parent);
9          try {
10             //3 将ObservableOnSubscribe（源头）与CreateEmitter（Observer，终点）联系起来
11             source.subscribe(parent);
12         } catch (Throwable ex) {
13             Exceptions.throwIfFatal(ex);
14             //4 错误回调
15             parent.onError(ex);
16         }
17     }

```

`Observer` 是一个接口，里面就四个方法，我们在开头的例子中已经全部实现（打印Log）。

```

1      public interface Observer<T> {
2          void onSubscribe(Disposable d);
3          void onNext(T value);
4          void onError(Throwable e);
5          void onComplete();
6      }

```

重点在这一句：

```

1      //3 将ObservableOnSubscribe（源头）与CreateEmitter（Observer，终点）联系起来
2      source.subscribe(parent);

```

`source` 即 `ObservableOnSubscribe` 对象，在本文中是：

```

1      new ObservableOnSubscribe<String>() {
2          @Override
3          public void subscribe(ObserverEmitter<String> e) throws Exception {
4              e.onNext("1");
5              e.onComplete();
6          }
7      }

```

则会调用 `parent.onNext()` 和 `parent.onComplete()`，`parent` 是 `CreateEmitter` 对象，如下：

```

1      static final class CreateEmitter<T>
2          extends AtomicReference<Disposable>
3          implements ObservableEmitter<T>, Disposable {
4          final Observer<? super T> observer;
5          CreateEmitter(Observer<? super T> observer) {
6              this.observer = observer;

```

```

7         }
8
9         @Override
10        public void onNext(T t) {
11            ...
12            //如果没有被dispose, 会调用Observer的onNext()方法
13            if (!isDisposed()) {
14                observer.onNext(t);
15            }
16        }
17
18        @Override
19        public void onError(Throwable t) {
20            ...
21            //1 如果没有被dispose, 会调用Observer的onError()方法
22            if (!isDisposed()) {
23                try {
24                    observer.onError(t);
25                } finally {
26                    //2 一定会自动dispose()
27                    dispose();
28                }
29            } else {
30                //3 如果已经被dispose了, 会抛出异常。所以onError、onComplete彼此互斥, 只能被调用
31                一次
32                RxJavaPlugins.onError(t);
33            }
34        }
35
36        @Override
37        public void onComplete() {
38            //1 如果没有被dispose, 会调用Observer的onComplete()方法
39            if (!isDisposed()) {
40                try {
41                    observer.onComplete();
42                } finally {
43                    //2 一定会自动dispose()
44                    dispose();
45                }
46            }
47        }
48
49        @Override
50        public void dispose() {
51            DisposableHelper.dispose(this);
52        }
53
54        @Override
55        public boolean isDisposed() {
56            return DisposableHelper.isDisposed(get());
57        }
58    }

```

总结重点：

1. `Observable`和`Observer`的关系没有被`dispose`，才会回调`Observer`的`onXXXX()`方法
2. `Observer`的`onComplete()`和`onError()`互斥只能执行一次，因为`CreateEmitter`在回调他们两中任意一个后，都会自动`dispose()`。根据第一点，验证此结论。
3. `Observable`和`Observer`关联时（订阅时），`Observable`才会开始发送数据。
4. `ObservableCreate`将`ObservableOnSubscribe`(真正的源)->`Observable`。
5. `ObservableOnSubscribe`(真正的源)需要的是发射器`ObservableEmitter`。
6. `CreateEmitter`将`Observer`->`ObservableEmitter`,同时它也是`Disposable`。
7. 先`error`后`complete`，`complete`不显示。反之会`crash`，感兴趣的可以写如下代码验证。

```

1    e.onNext("1");
2    //先error后complete, complete不显示。反之 会crash
3    //e.onError(new IOException("sb error"));

```

```

4         e.onComplete();
5         e.onError(new IOException("sb error"));

```

一个好玩的地方 DisposableHelper

原本到这里，最简单的一个流程我们算是搞清了。

还值得一提的是，DisposableHelper.dispose(this);

DisposableHelper 很有趣，它是一个枚举，这是利用枚举实现了一个单例 disposed state, 即是否 disposed, 如果 Disposable 类型的变量的引用等于 DISPOSED, 则起点和终点已经断开联系。

其中大多数方法都是静态方法，所以 isDisposed() 方法的实现就很简单，直接比较引用即可。

其他的几个方法，和 AtomicReference 类搅基在了一起。

这是一个实现引用原子操作的类，对象引用的原子更新，常用方法如下：

```

1 //返回当前的引用。
2 V get()
3 //如果当前值与给定的expect引用相等，（注意是引用相等而不是equals()相等），更新为指定的update
4 值。
5 boolean compareAndSet(V expect, V update)
6 //原子地设为给定值并返回旧值。
7 V getAndSet(V newValue)

```

OK,铺垫完了我们看看源码吧：

```

1 public enum DisposableHelper implements Disposable {
2     /**
3      * The singleton instance representing a terminal, disposed state, don't leak it.
4      */
5     DISPOSED
6     ;
7
8     public static boolean isDisposed(Disposable d) {
9         return d == DISPOSED;
10    }
11
12    public static boolean dispose(AtomicReference<Disposable> field) {
13        //1 通过断点查看，默认情况下,field的值是"null"，并非引用是null哦！大坑大坑大坑
14        //但是current是null引用
15        Disposable current = field.get();
16        Disposable d = DISPOSED;
17        //2 null不等于DISPOSED
18        if (current != d) {
19            //3 field是DISPOSED了，current还是null
20            current = field.getAndSet(d);
21            if (current != d) {
22                //4 默认情况下 走不到这里，这里是在设置了setCancellable()后会走到。
23                if (current != null) {
24                    current.dispose();
25                }
26                return true;
27            }
28        }
29        return false;
30    }

```

总结

1. 在 subscribeActual() 方法中，源头和终点关联起来。
2. source.subscribe(parent); 这句代码执行时，才开始从发送 ObservableOnSubscribe 中利用 ObservableEmitter 发送数据给 Observer。即数据是从源头 push 给终点的。
3. CreateEmitter 中，只有 Observable 和 Observer 的关系没有被 dispose，才会回调 Observer 的 onXXX x() 方法
4. Observer 的 onComplete() 和 onError() 互斥只能执行一次，因为 CreateEmitter 在回调他们两中任意一个后，都会自动 dispose()。根据上一点，验证此结论。
5. 先 error 后 complete，complete 不显示。反之会 crash
6. 还有一点要注意的是 onSubscribe() 是在我们执行 subscribe() 这句代码的那个线程回调的，并不受线程