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

概述

最近事情太多了，现在公司内部的变动，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(ObservableEmitter<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            }
```

```
        });
```

拿 create来说，

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

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

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

`ObservableOnSubscribe`是一个接口，里面就一个方法,也是我们实现的那个方法：
该方法的参数是 `ObservableEmitter`，我认为它是关联起 `Disposable`概念的一层：

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

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

```
public interface Emitter<T> {
    void onNext(T value);
    void onError(Throwable error);
    void onComplete();
}
```

`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()`:

```
    /**
     * Calls the associated hook function.
     * @param <T> the value type
     * @param source the hook's input value
     * @return the value returned by the hook
     */
    @SuppressWarnings({ "rawtypes", "unchecked" })
    public static <T> Observable<T> onAssembly(Observable<T> source) {
        Function<Observable, Observable> f = onObservableAssembly;
        if (f != null) {
            return apply(f, source);
        }
        return source;
    }
```

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:

```
public final class ObservableCreate<T> extends Observable<T> {
    final ObservableOnSubscribe<T> source;
    public ObservableCreate(ObservableOnSubscribe<T> source) {
        this.source = source;
    }
    @Override
```

```
    protected void s b cribeActual(Observer<? super T> observer) {
        CreateEmitter<T> parent = new CreateEmitter<T>(observer);
        observer.onSubscribe(parent);
        try {
            source.subscribe(parent);
        } catch (Throwable ex) {
            Exceptions.throwIfFatal(ex);
            parent.onError(ex);
        }
    }
```

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()`：

```
    public final void subscribe(Observer<? super T> observer) {
        ...
        try {
            //1 hook相关，略过
            observer = RxJavaPlugins.onSubscribe(this, observer);
            ...
            //2 真正的订阅处
            subscribeActual(observer);
        } catch (NullPointerException e) { // NOPMD
            throw e;
        } catch (Throwable e) {
            //3 错误处理，
            Exceptions.throwIfFatal(e);
            // can't call onError because no way      ıow if a Disposable has been set or
not
            // can't call onSubscribe because the call might have set a Subscription alr
eady
            //4 hook错误相关，略过
            RxJavaPlugins.onError(e);

            NullPointerException npe = new NullPointerException("Actually not, but can't
throw other exceptions due to RS");
            npe.initCause(e);
            throw npe;
        }
    }
```

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。**

```
    /**
     * Calls the associated hook function.
     * @param <T> the value type
     * @param source the hook's input value
     * @param observer the observer
     * @return the value returned by the hook
     */
    @SuppressWarnings({ "rawtypes", "unchecked" })
    public static <T> Observer<? super T> onSubscribe(Observable<T> source, Observer<? s
uper T> observer) {
        //1 默认onObservableSubscribe（可理解为一个flatmap的操作）是null
        BiFunction<Observable, Observer, Observer> f = onObservableSubscribe;
        //2 所以这句跳过，不会对其进行apply
        if (f != null) {
            return apply(f, source, observer);
        }
        //3 返回参数2
        return observer;
    }
```

我也是验证了一下 三个Hook相关的变量，确实是null：

```
        Consumer<Throwable> errorHandler = RxJavaPlugins.getErrorHandler();
        BiFunction<Observable, Observer, Observer> onObservableSubscribe = RxJavaPlugin
s.getOnObservableSubscribe();
        Function<Observable, Observable> onObservableAssembly = RxJavaPlugins.getOnObser
vableAssembly();

        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：

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

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

```
    @Override
    protected void subscribeActual(Observer<? super T> observer) {
        //1 创建CreateEmitter，也是一个适配器
        CreateEmitter<T> parent = new CreateEmitter<T>(observer);
        //2 onSubscribe（）参数是Disposable ，所以CreateEmitter可以将Observer->Disposable
。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
程调度影响。
        observer.onSubscribe(parent);
        try {
            //3 将ObservableOnSubscribe（源头）与CreateEmitter（Observer，终点）联系起来
            source.subscribe(parent);
        } catch (Throwable ex) {
            Exceptions.throwIfFatal(ex);
            //4 错误回调
            parent.onError(ex);
        }
    }
```

`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）。

```
public interface Observer<T> {
    void onSubscribe(Disposable d);
    void onNext(T value);
    void onError(Throwable e);
    void onComplete();
}
```

重点在这一句：

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

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

```
        new ObservableOnSubscribe<String>() {
            @Override
            public void subscribe(ObservableEmitter<String> e) throws Exception {
                e.onNext("1");
                e.onComplete();
            }
        }
```

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

```
 static final class CreateEmitter<T>
    extends AtomicReference<Disposable>
    implements ObservableEmitter<T>, Disposable {
        final Observer<? super T> observer;
        CreateEmitter(Observer<? super T> observer) {
            this.observer = observer;
```

```
        }

        @Override
        public void onNext(T t) {
            ...
            //如果没有被dispose，会调用Observer的onNext()方法
            if (!isDisposed()) {
                observer.onNext(t);
            }
        }

        @Override
        public void onError(Throwable t) {
            ...
            //1 如果没有被dispose，会调用Observer的onError()方法
            if (!isDisposed()) {
                try {
                    observer.onError(t);
                } finally {
                //2 一定会自动dispose()
                    dispose();
                }
            } else {
            //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
一次
                RxJavaPlugins.onError(t);
            }
        }

        @Override
        public void onComplete() {
         //1 如果没有被dispose，会调用Observer的onComplete()方法
            if (!isDisposed()) {
                try {
                    observer.onComplete();
                } finally {
                 //2 一定会自动dispose()
                    dispose();
                }
            }
        }

        @Override
        public void dispose() {
            DisposableHelper.dispose(this);
        }

        @Override
        public boolean isDisposed() {
            return DisposableHelper.isDisposed(get());
        }
    }
```

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。

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

```
        e.onComplete();
        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dispose d**，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：

```
//返回当前的引用。
V get()
//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值。
boolean compareAndSet(V expect, V update)
//原子地设为给定值并返回旧值。
V getAndSet(V newValue)
```

OK,铺垫完了我们看看源码吧：

```
public enum DisposableHelper implements Disposable {
    /**
     * The singleton instance representing a terminal, disposed state, don't leak it.
     */
    DISPOSED
    ;

    public static boolean isDisposed(Disposable d) {
        return d == DISPOSED;
    }

    public static boolean dispose(AtomicReference<Disposable> field) {
        //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
        //但是current是null引用
        Disposable current = field.get();
        Disposable d = DISPOSED;
        //2 null不等于DISPOSED
        if (current != d) {
            //3 field是DISPOSED了，current还是null
            current = field.getAndSet(d);
            if (current != d) {
            //4 默认情况下 走不到这里，这里是在设置了setCancellable()后会走到。
                if (current != null) {
                    current.dispose();
                }
                return true;
            }
        }
        return false;
    }
```

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`onXXXX()`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**

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

概述

最近事情太多了，现在公司内部的变动，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(ObservableEmitter<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            }
```

```
        });
```

拿 create来说，

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

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

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

`ObservableOnSubscribe`是一个接口，里面就一个方法,也是我们实现的那个方法：
该方法的参数是 `ObservableEmitter`，我认为它是关联起 `Disposable`概念的一层：

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

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

```
public interface Emitter<T> {
    void onNext(T value);
    void onError(Throwable error);
    void onComplete();
}
```

`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()`:

```
    /**
     * Calls the associated hook function.
     * @param <T> the value type
     * @param source the hook's input value
     * @return the value returned by the hook
     */
    @SuppressWarnings({ "rawtypes", "unchecked" })
    public static <T> Observable<T> onAssembly(Observable<T> source) {
        Function<Observable, Observable> f = onObservableAssembly;
        if (f != null) {
            return apply(f, source);
        }
        return source;
    }
```

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:

```
public final class ObservableCreate<T> extends Observable<T> {
    final ObservableOnSubscribe<T> source;
    public ObservableCreate(ObservableOnSubscribe<T> source) {
        this.source = source;
    }
    @Override
```

```
    protected void s b cribeActual(Observer<? super T> observer) {
        CreateEmitter<T> parent = new CreateEmitter<T>(observer);
        observer.onSubscribe(parent);
        try {
            source.subscribe(parent);
        } catch (Throwable ex) {
            Exceptions.throwIfFatal(ex);
            parent.onError(ex);
        }
    }
```

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()`：

```
    public final void subscribe(Observer<? super T> observer) {
        ...
        try {
            //1 hook相关，略过
            observer = RxJavaPlugins.onSubscribe(this, observer);
            ...
            //2 真正的订阅处
            subscribeActual(observer);
        } catch (NullPointerException e) { // NOPMD
            throw e;
        } catch (Throwable e) {
            //3 错误处理，
            Exceptions.throwIfFatal(e);
            // can't call onError because no way      ıow if a Disposable has been set or
not
            // can't call onSubscribe because the call might have set a Subscription alr
eady
            //4 hook错误相关，略过
            RxJavaPlugins.onError(e);

            NullPointerException npe = new NullPointerException("Actually not, but can't
throw other exceptions due to RS");
            npe.initCause(e);
            throw npe;
        }
    }
```

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。**

```
    /**
     * Calls the associated hook function.
     * @param <T> the value type
     * @param source the hook's input value
     * @param observer the observer
     * @return the value returned by the hook
     */
    @SuppressWarnings({ "rawtypes", "unchecked" })
    public static <T> Observer<? super T> onSubscribe(Observable<T> source, Observer<? s
uper T> observer) {
        //1 默认onObservableSubscribe（可理解为一个flatmap的操作）是null
        BiFunction<Observable, Observer, Observer> f = onObservableSubscribe;
        //2 所以这句跳过，不会对其进行apply
        if (f != null) {
            return apply(f, source, observer);
        }
        //3 返回参数2
        return observer;
    }
```

我也是验证了一下 三个Hook相关的变量，确实是null：

```
        Consumer<Throwable> errorHandler = RxJavaPlugins.getErrorHandler();
        BiFunction<Observable, Observer, Observer> onObservableSubscribe = RxJavaPlugin
s.getOnObservableSubscribe();
        Function<Observable, Observable> onObservableAssembly = RxJavaPlugins.getOnObser
vableAssembly();

        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：

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

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

```
    @Override
    protected void subscribeActual(Observer<? super T> observer) {
        //1 创建CreateEmitter，也是一个适配器
        CreateEmitter<T> parent = new CreateEmitter<T>(observer);
        //2 onSubscribe（）参数是Disposable ，所以CreateEmitter可以将Observer->Disposable
。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
程调度影响。
        observer.onSubscribe(parent);
        try {
            //3 将ObservableOnSubscribe（源头）与CreateEmitter（Observer，终点）联系起来
            source.subscribe(parent);
        } catch (Throwable ex) {
            Exceptions.throwIfFatal(ex);
            //4 错误回调
            parent.onError(ex);
        }
    }
```

`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）。

```
public interface Observer<T> {
    void onSubscribe(Disposable d);
    void onNext(T value);
    void onError(Throwable e);
    void onComplete();
}
```

重点在这一句：

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

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

```
        new ObservableOnSubscribe<String>() {
            @Override
            public void subscribe(ObservableEmitter<String> e) throws Exception {
                e.onNext("1");
                e.onComplete();
            }
        }
```

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

```
 static final class CreateEmitter<T>
    extends AtomicReference<Disposable>
    implements ObservableEmitter<T>, Disposable {
        final Observer<? super T> observer;
        CreateEmitter(Observer<? super T> observer) {
            this.observer = observer;
```

```
        }

        @Override
        public void onNext(T t) {
            ...
            //如果没有被dispose，会调用Observer的onNext()方法
            if (!isDisposed()) {
                observer.onNext(t);
            }
        }

        @Override
        public void onError(Throwable t) {
            ...
            //1 如果没有被dispose，会调用Observer的onError()方法
            if (!isDisposed()) {
                try {
                    observer.onError(t);
                } finally {
                //2 一定会自动dispose()
                    dispose();
                }
            } else {
            //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
一次
                RxJavaPlugins.onError(t);
            }
        }

        @Override
        public void onComplete() {
         //1 如果没有被dispose，会调用Observer的onComplete()方法
            if (!isDisposed()) {
                try {
                    observer.onComplete();
                } finally {
                 //2 一定会自动dispose()
                    dispose();
                }
            }
        }

        @Override
        public void dispose() {
            DisposableHelper.dispose(this);
        }

        @Override
        public boolean isDisposed() {
            return DisposableHelper.isDisposed(get());
        }
    }
```

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。

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

```
        e.onComplete();
        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dispose d**，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：

```
//返回当前的引用。
V get()
//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值。
boolean compareAndSet(V expect, V update)
//原子地设为给定值并返回旧值。
V getAndSet(V newValue)
```

OK,铺垫完了我们看看源码吧：

```
public enum DisposableHelper implements Disposable {
    /**
     * The singleton instance representing a terminal, disposed state, don't leak it.
     */
    DISPOSED
    ;

    public static boolean isDisposed(Disposable d) {
        return d == DISPOSED;
    }

    public static boolean dispose(AtomicReference<Disposable> field) {
        //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
        //但是current是null引用
        Disposable current = field.get();
        Disposable d = DISPOSED;
        //2 null不等于DISPOSED
        if (current != d) {
            //3 field是DISPOSED了，current还是null
            current = field.getAndSet(d);
            if (current != d) {
            //4 默认情况下 走不到这里，这里是在设置了setCancellable()后会走到。
                if (current != null) {
                    current.dispose();
                }
                return true;
            }
        }
        return false;
    }
```

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`onXXXX()`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**