# RxJava2 使用

# 引入RxJava2

## 配置

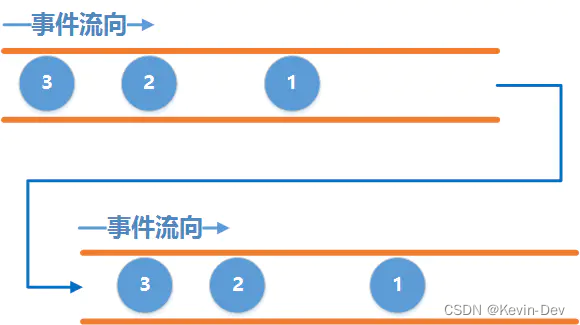
dependencies **{**

…

//RxJava  
 implementation 'io.reactivex.rxjava2:rxjava:2.2.10'  
 implementation 'io.reactivex.rxjava2:rxandroid:2.1.1'  
  
**}**

## 基本原理

先假设有两根水管：



上面一根水管为事件产生的水管，叫它上游，下面一根水管为事件接收的水管叫它下游。

两根水管通过一定的方式连接起来，使得上游每产生一个事件，下游就能收到该事件。注意这里和官网的事件图是反过来的, 这里的事件发送的顺序是先 1,后 2,后 3 这样的顺序, 事件接收的顺序也是先 1,后 2,后 3 的顺序, 我觉得这样更符合我们普通人的思维, 简单明了.

这里的上游和下游就分别对应着 RxJava 中的 Observable 和 Observer，它们之间的连接就对应着 subscribe()，因此这个关系用 RxJava 来表示就是：

//创建一个上游 Observable：  
Observable<Integer> observable = Observable.*create*(new ObservableOnSubscribe<Integer>() {  
 @Override  
 public void subscribe(ObservableEmitter<Integer> emitter) throws Exception {  
 emitter.onNext(1);  
 emitter.onNext(2);  
 emitter.onNext(3);  
 emitter.onComplete();  
 }  
});  
//创建一个下游 Observer  
Observer<Integer> observer = new Observer<Integer>() {  
 @Override  
 public void onSubscribe(Disposable d) {  
 Utils.*log*("subscribe");  
 }  
  
 @Override  
 public void onNext(Integer value) {  
 Utils.*log*("" + value);  
 }  
  
 @Override  
 public void onError(Throwable e) {  
 Utils.*log*("error");  
 }  
  
 @Override  
 public void onComplete() {  
 Utils.*log*("complete");  
 }  
};  
//建立连接  
observable.subscribe(observer);

注意: 只有当上游和下游建立连接之后, 上游才会开始发送事件. 也就是调用了subscribe() 方法之后才开始发送事件.

下面把这段代码连起来写就成了 RxJava 引以为傲的链式操作：

Observable.*create*(new ObservableOnSubscribe<Integer>() {  
 @Override  
 public void subscribe(ObservableEmitter<Integer> emitter) throws Exception {  
 emitter.onNext(1);  
 emitter.onNext(2);  
 emitter.onNext(3);  
 emitter.onComplete();  
 }  
}).subscribe(new Observer<Integer>() {  
 @Override  
 public void onSubscribe(Disposable d) {  
 Utils.*log*("subscribe");  
 }  
  
 @Override  
 public void onNext(Integer value) {  
 Utils.*log*("" + value);  
 }  
  
 @Override  
 public void onError(Throwable e) {  
 Utils.*log*("error");  
 }  
  
 @Override  
 public void onComplete() {  
 Utils.*log*("complete");  
 }  
});

解释说明：

1. ObservableEmitter

Emitter 是发射器的意思，那就很好猜了，这个就是用来发出事件的，它可以发出三种类型的事件，通过调用 emitter 的 onNext(T value) 、onComplete() 和 onError(Throwable error) 就可以分别发出 next 事件、complete 事件和 error 事件。

请注意，需要满足一定的规则：

上游可以发送无限个 onNext , 下游也可以接收无限个 onNext 。

当上游发送了一个 onComplete 后, 上游 onComplete 之后的事件将会继续发送, 而下游收到 onComplete 事件之后将不再继续接收事件.

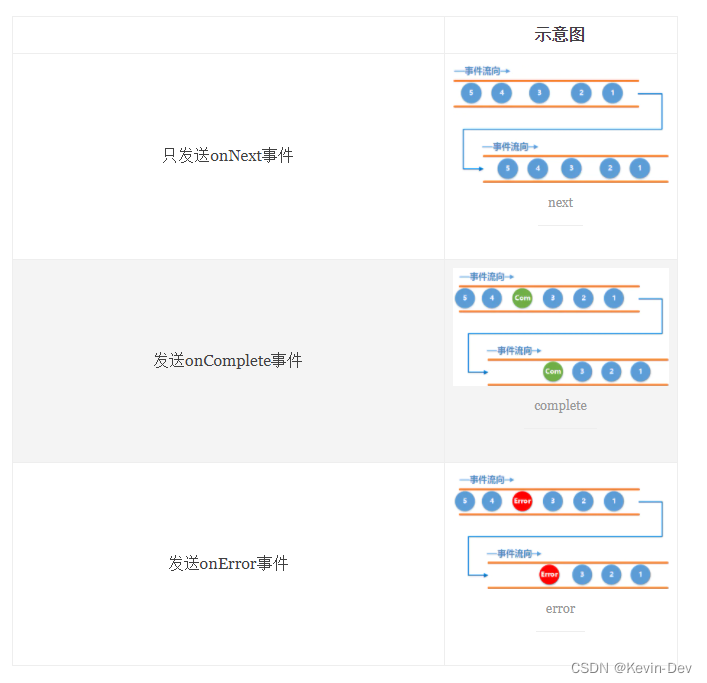
当上游发送了一个 onError 后, 上游 onError 之后的事件将继续发送, 而下游收到 onError 事件之后将不再继续接收事件.

上游可以不发送 onComplete 或 onError 。

最为关键的是 onComplete 和 onError 必须唯一并且互斥, 即不能发多个 onComplete , 也不能发多个 onError , 也不能先发一个 onComplete , 然后再发一个 onError , 反之亦然

注: 关于 onComplete 和 onError 唯一并且互斥这一点, 是需要自行在代码中进行控制, 如果你的代码逻辑中违背了这个规则,并不一定会导致程序崩溃. 比如发送多个 onComplete 是可以正常运行的, 依然是收到第一个 onComplete 就不再接收了, 但若是发送多个 onError, 则收到第二个 onError 事件会导致程序会崩溃.

以上几个规则用示意图表示如下:



1. Disposable

这个单词的字面意思是一次性用品,用完即可丢弃的. 那么在 RxJava 中怎么去理解它呢, 对应于上面的水管的例子, 我们可以把它理解成两根管道之间的一个机关, 当调用它的 dispose() 方法时, 它就会将两根管道切断, 从而导致下游收不到事件。

注意: 调用 dispose() 并不会导致上游不再继续发送事件, 上游会继续发送剩余的事件.

来看个例子, 我们让上游依次发送1,2,3,complete,4,在下游收到第二个事件之后, 切断水管, 看看运行结果:

Observable.*create*(new ObservableOnSubscribe<Integer>() {  
 @Override  
 public void subscribe(ObservableEmitter<Integer> emitter) throws Exception {  
 Utils.*log*("emit 1");  
 emitter.onNext(1);  
 Utils.*log*("emit 2");  
 emitter.onNext(2);  
 Utils.*log*("emit 3");  
 emitter.onNext(3);  
 Utils.*log*("emit complete");  
 emitter.onComplete();  
 Utils.*log*("emit 4");  
 emitter.onNext(4);  
 }  
}).subscribe(new Observer<Integer>() {  
 private Disposable mDisposable;  
 private int i;  
  
 @Override  
 public void onSubscribe(Disposable d) {  
 Utils.*log*("subscribe");  
 mDisposable = d;  
 }  
  
 @Override  
 public void onNext(Integer value) {  
 Utils.*log*("onNext: " + value);  
 i++;  
 if (i == 2) {  
 Utils.*log*("dispose");  
 mDisposable.dispose();  
 Utils.*log*("isDisposed : " + mDisposable.isDisposed());  
 }  
 }  
  
 @Override  
 public void onError(Throwable e) {  
 Utils.*log*("error");  
 }  
  
 @Override  
 public void onComplete() {  
 Utils.*log*("complete");  
 }  
});

在收到 onNext 2这个事件后, 切断了水管, 但是上游仍然发送了3, complete, 4这几个事件, 而且上游并没有因为发送了onComplete而停止. 同时可以看到下游的onSubscribe()方法是最先调用的。

subscribe()有多个重载的方法:

public final Disposable subscribe() {}  
public final Disposable subscribe(Consumer<? super T> onNext) {}  
public final Disposable subscribe(Consumer<? super T> onNext, Consumer<? super Throwable> onError) {}  
public final Disposable subscribe(Consumer<? super T> onNext, Consumer<? super Throwable> onError, Action onComplete) {}  
public final Disposable subscribe(Consumer<? super T> onNext, Consumer<? super Throwable> onError, Action onComplete, Consumer<? super Disposable> onSubscribe) {}  
public final void subscribe(Observer<? super T> observer) {}

最后一个带有 Observer 参数的我们已经使用过了，这里对其他几个方法进行说明：

不带任何参数的 subscribe() 表示下游不关心任何事件,你上游尽管发你的数据去吧, 老子可不管你发什么.

带有一个 Consumer 参数的方法表示下游只关心 onNext 事件, 其他的事件我假装没看见, 因此我们如果只需要 onNext 事件可以这么写:

Observable.*create*(new ObservableOnSubscribe<Integer>() {  
 @Override  
 public void subscribe(ObservableEmitter<Integer> emitter) throws Exception {  
 Utils.*log*("emit 1");  
 emitter.onNext(1);  
 Utils.*log*("emit 2");  
 emitter.onNext(2);  
 Utils.*log*("emit 3");  
 emitter.onNext(3);  
 Utils.*log*("emit complete");  
 emitter.onComplete();  
 Utils.*log*("emit 4");  
 emitter.onNext(4);  
 }  
}).subscribe(new Consumer<Integer>() {  
 @Override  
 public void accept(Integer integer) throws Exception {  
 Utils.*log*("onNext: " + integer);  
 }  
});

式样

微软雅黑 Light 5号

段落 固定值 23磅

# 目录1

## 目录2

### 目录3

#### 目录4

##### 目录5