# RxJava2 使用

概念

Observable [əbˈzɜːvəbl] 能看得到的;能察觉到的;

observer [əbˈzɜːvə(r)] 观察者

subscriber [səbˈskraɪbə(r)] 订阅人

# 引入RxJava2

## 配置

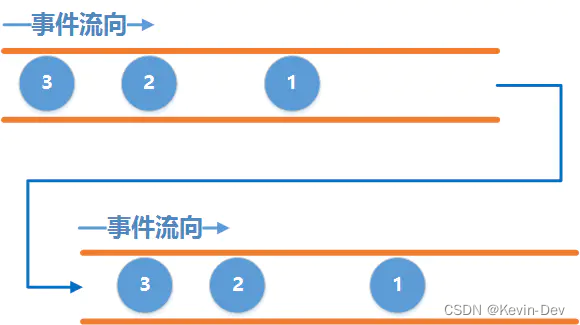
dependencies **{**

…

//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.

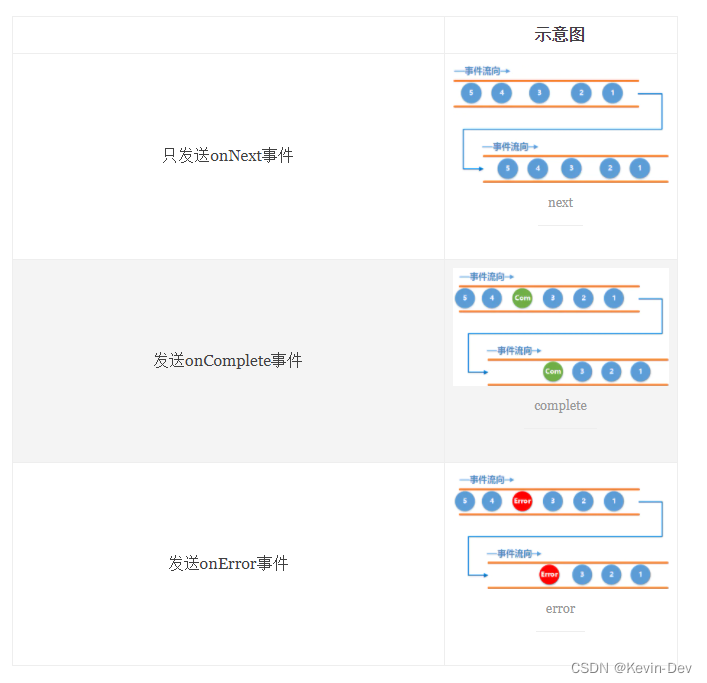
当上游发送了一个 onError 后, 上游 onError 之后的事件将继续发送, 而下游收到 onError 事件之后将不再继续接收事件.

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.

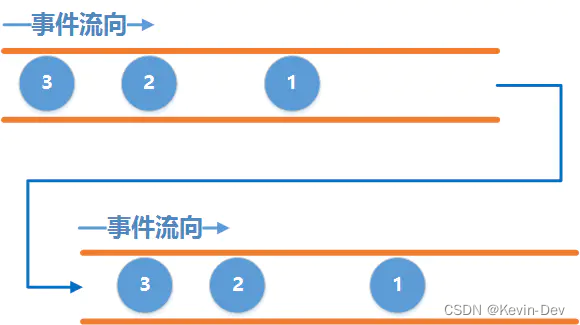
带有一个 Consumer 参数的方法表示下游只关心 onNext 事件, 其他的事件我假装没看见, 因此我们如果只需要 onNext 事件可以这么写:

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);  
 }  
});

# 强大的线程控制

在本节中, 我们将学习 RxJava 强大的线程控制。

开始



正常情况下, 上游和下游是工作在同一个线程中的, 也就是说上游在哪个线程发事件, 下游就在哪个线程接收事件.

怎么去理解呢, 以 Android 为例, 一个 Activity 的所有动作默认都是在主线程中运行的, 比如我们在 onCreate 中打出当前线程的名字:

@Override  
protected void onCreate(Bundle savedInstanceState) {  
 super.onCreate(savedInstanceState);  
 setContentView(R.layout.*activity\_thread*);  
 Log.*d*(*TAG*, Thread.*currentThread*().getName());  
}

输出结果：

D/ThreadActivity: main

回到 RxJava 中, 当我们在主线程中去创建一个上游 Observable 来发送事件, 则这个上游默认就在主线程发送事件.当我们在主线程去创建一个下游 Observer 来接收事件, 则这个下游默认就在主线程中接收事件, 来看段代码:

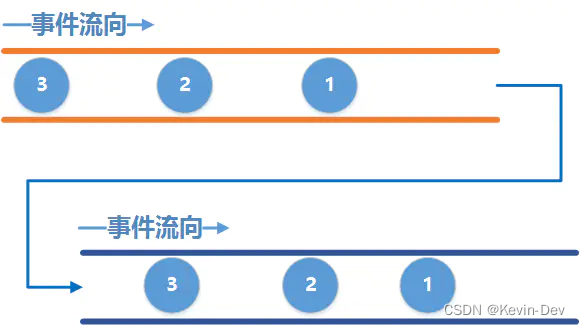
public void onTest1(View v) {  
 Observable<Integer> observable = Observable.*create*(new ObservableOnSubscribe<Integer>() {  
 @Override  
 public void subscribe(ObservableEmitter<Integer> emitter) throws Exception {  
 Log.*d*(*TAG*, "Observable thread is : " + Thread.*currentThread*().getName());  
 Log.*d*(*TAG*, "emit 1");  
 emitter.onNext(1);  
 }  
 });  
  
 Consumer<Integer> consumer = new Consumer<Integer>() {  
 @Override  
 public void accept(Integer integer) throws Exception {  
 Log.*d*(*TAG*, "Observer thread is :" + Thread.*currentThread*().getName());  
 Log.*d*(*TAG*, "onNext: " + integer);  
 }  
 };  
  
 observable.subscribe(consumer);  
}

输出结果：

D/MainActivity: Observable thread is : main  
D/MainActivity: emit 1  
D/MainActivity: Observer thread is :main  
D/MainActivity: onNext: 1

这就验证了刚才所说, 上下游默认是在同一个线程工作。

这样肯定是满足不了我们的需求的, 我们更多想要的是这么一种情况, 在子线程中做耗时的操作, 然后回到主线程中来操作UI, 用图片来描述就是下面这个图片:



在这个图中, 我们用黄色水管表示子线程, 深蓝色水管表示主线程。

要达到这个目的, 我们需要先改变上游发送事件的线程, 让它去子线程中发送事件, 然后再改变下游的线程, 让它去主线程接收事件. 通过RxJava内置的线程调度器可以很轻松的做到这一点. 接下来看一段代码:

public void onTest2(View v) {  
 Observable<Integer> observable = Observable.*create*(new ObservableOnSubscribe<Integer>() {  
 @Override  
 public void subscribe(ObservableEmitter<Integer> emitter) throws Exception {  
 Log.*d*(*TAG*, "Observable thread is : " + Thread.*currentThread*().getName());  
 Log.*d*(*TAG*, "emit 1");  
 emitter.onNext(1);  
 }  
 });  
  
 Consumer<Integer> consumer = new Consumer<Integer>() {  
 @Override  
 public void accept(Integer integer) throws Exception {  
 Log.*d*(*TAG*, "Observer thread is :" + Thread.*currentThread*().getName());  
 Log.*d*(*TAG*, "onNext: " + integer);  
 }  
 };  
  
 observable.subscribeOn(Schedulers.*newThread*())  
 .observeOn(AndroidSchedulers.*mainThread*())  
 .subscribe(consumer);  
}

输出结果：

D/MainActivity: Observable thread is : RxNewThreadScheduler-1  
D/MainActivity: emit 1  
D/MainActivity: Observer thread is :main  
D/MainActivity: onNext: 1

可以看到, 上游发送事件的线程的确改变了, 是在一个叫 RxNewThreadScheduler-1 的线程中发送的事件, 而下游仍然在主线程中接收事件, 这说明我们的目的达成了, 接下来看看是如何做到的。

和上一段代码相比,这段代码只不过是增加了两行代码:

.subscribeOn(Schedulers.*newThread*())

.observeOn(AndroidSchedulers.*mainThread*())

简单的来说, subscribeOn() 指定的是上游发送事件的线程, observeOn() 指定的是下游接收事件的线程.

多次指定上游的线程只有第一次指定的有效, 也就是说多次调用subscribeOn() 只有第一次的有效, 其余的会被忽略.

多次指定下游的线程是可以的, 也就是说每调用一次observeOn() , 下游的线程就会切换一次.

举个例子:

observable.subscribeOn(Schedulers.*newThread*())  
 .subscribeOn(Schedulers.*io*())  
 .observeOn(AndroidSchedulers.*mainThread*())  
 .observeOn(Schedulers.*io*())  
 .subscribe(consumer);

这段代码中指定了两次上游发送事件的线程, 分别是 newThread 和 IO 线程, 下游也指定了两次线程,分别是 main 和 IO 线程. 运行结果为:

D/MainActivity: Observable thread is : RxNewThreadScheduler-3  
D/MainActivity: emit 1  
D/MainActivity: Observer thread is :RxCachedThreadScheduler-1  
D/MainActivity: onNext: 1

可以看到, 上游虽然指定了两次线程, 但只有第一次指定的有效, 依然是在 RxNewThreadScheduler 线程中, 而下游则跑到了RxCachedThreadScheduler 中, 这个CacheThread其实就是 IO 线程池中的一个.

为了更清晰的看到下游的线程切换过程, 我们加点 log :

observable.subscribeOn(Schedulers.*newThread*())  
 .subscribeOn(Schedulers.*io*())  
 .observeOn(AndroidSchedulers.*mainThread*())  
 .doOnNext(new Consumer<Integer>() {  
 @Override  
 public void accept(Integer integer) throws Exception {  
 Log.*d*(*TAG*, "After observeOn(mainThread), current thread is: " + Thread.*currentThread*().getName());  
 }  
 })  
 .observeOn(Schedulers.*io*())  
 .doOnNext(new Consumer<Integer>() {  
 @Override  
 public void accept(Integer integer) throws Exception {  
 Log.*d*(*TAG*, "After observeOn(io), current thread is : " + Thread.*currentThread*().getName());  
 }  
 })  
 .subscribe(consumer);

运行结果:

D/MainActivity: Observable thread is : RxNewThreadScheduler-1  
D/MainActivity: emit 1  
D/MainActivity: After observeOn(mainThread), current thread is: main  
D/MainActivity: After observeOn(io), current thread is : RxCachedThreadScheduler-2  
D/MainActivity: Observer thread is :RxCachedThreadScheduler-2  
D/MainActivity: onNext: 1

在RxJava中, 已经内置了很多线程选项供我们选择, 例如：

Schedulers.io() 代表io操作的线程, 通常用于网络,读写文件等io密集型的操作

Schedulers.computation() 代表CPU计算密集型的操作, 例如需要大量计算的操作

Schedulers.newThread() 代表一个常规的新线程

AndroidSchedulers.mainThread() 代表Android的主线程

这些内置的Scheduler已经足够满足我们开发的需求, 因此我们应该使用内置的这些选项, 在RxJava内部使用的是线程池来维护这些线程, 所有效率也比较高.

实践

对于我们 Android 开发人员来说, 经常会将一些耗时的操作放在后台, 比如网络请求或者读写文件,操作数据库等等,等到操作完成之后回到主线程去更新 UI, 有了上面的这些基础, 那么现在我们就可以轻松的去做到这样一些操作。

网络请求

1. 使用 Retrofit,先添加 Gradle 配置:

//retrofit

implementation 'com.squareup.retrofit2:retrofit:2.9.0'

//Gson converter

implementation 'com.squareup.retrofit2:converter-gson:2.6.1'

//RxJava2 Adapter

implementation 'com.jakewharton.retrofit:retrofit2-rxjava2-adapter:1.0.0'

//okhttp

implementation 'com.squareup.okhttp3:okhttp:3.14.9'

implementation 'com.squareup.okhttp3:logging-interceptor:3.10.0'

1

2

3

4

5

6

7

8

9

2. 定义 Api 接口:

public interface Api {

@GET

Observable<LoginResponse> login(@Body LoginRequest request);

@GET

Observable<RegisterResponse> register(@Body RegisterRequest request);

}

1

2

3

4

5

6

7

3. 创建一个 Retrofit 客户端:

private static Retrofit create() {

OkHttpClient.Builder builder = new OkHttpClient().newBuilder();

builder.readTimeout(10, TimeUnit.SECONDS);

builder.connectTimeout(9, TimeUnit.SECONDS);

if (BuildConfig.DEBUG) {

HttpLoggingInterceptor interceptor = new HttpLoggingInterceptor();

interceptor.setLevel(HttpLoggingInterceptor.Level.BODY);

builder.addInterceptor(interceptor);

}

return new Retrofit.Builder().baseUrl(ENDPOINT)

.client(builder.build())

.addConverterFactory(GsonConverterFactory.create())

.addCallAdapterFactory(RxJava2CallAdapterFactory.create())

.build();

}

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

4. 发起请求就很简单了:

Api api = retrofit.create(Api.class);

api.login(request)

.subscribeOn(Schedulers.io()) //在IO线程进行网络请求

.observeOn(AndroidSchedulers.mainThread()) //回到主线程去处理请求结果

.subscribe(new Observer<LoginResponse>() {

@Override

public void onSubscribe(Disposable d) {}

@Override

public void onNext(LoginResponse value) {}

@Override

public void onError(Throwable e) {

Toast.makeText(mContext, "登录失败", Toast.LENGTH\_SHORT).show();

}

@Override

public void onComplete() {

Toast.makeText(mContext, "登录成功", Toast.LENGTH\_SHORT).show();

}

});

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

看似很完美, 但我们忽略了一点, 如果在请求的过程中Activity已经退出了, 这个时候如果回到主线程去更新UI, 那么APP肯定就崩溃了, 怎么办呢, 上一节我们说到了Disposable , 说它是个开关, 调用它的dispose()方法时就会切断水管, 使得下游收不到事件, 既然收不到事件, 那么也就不会再去更新UI了. 因此我们可以在Activity中将这个Disposable 保存起来, 当Activity退出时, 切断它即可.

那如果有多个Disposable 该怎么办呢, RxJava中已经内置了一个容器CompositeDisposable, 每当我们得到一个Disposable时就调用CompositeDisposable.add()将它添加到容器中, 在退出的时候, 调用CompositeDisposable.clear() 即可切断所有的水管.

读写数据库

读写数据库也算一个耗时的操作, 因此我们也最好放在IO线程里去进行, 这个例子就比较简单, 直接上代码:

public Observable<List<Record>> readAllRecords() {

return Observable.create(new ObservableOnSubscribe<List<Record>>() {

@Override

public void subscribe(ObservableEmitter<List<Record>> emitter) throws Exception {

Cursor cursor = null;

try {

cursor = getReadableDatabase().rawQuery("select \* from " + TABLE\_NAME, new String[]{});

List<Record> result = new ArrayList<>();

while (cursor.moveToNext()) {

result.add(Db.Record.read(cursor));

}

emitter.onNext(result);

emitter.onComplete();

} finally {

if (cursor != null) {

cursor.close();

}

}

}

}).subscribeOn(Schedulers.io()).observeOn(AndroidSchedulers.mainThread());

}

# 强大的操作符

## 前言

上一节讲解了线程调度, 并且举了两个实际中的例子, 其中有一个登录的例子, 不知大家有没有想过这么一个问题, 如果是一个新用户, 必须先注册, 等注册成功之后再自动登录该怎么做呢？

很明显, 这是一个嵌套的网络请求, 首先需要去请求注册, 待注册成功回调了再去请求登录的接口。

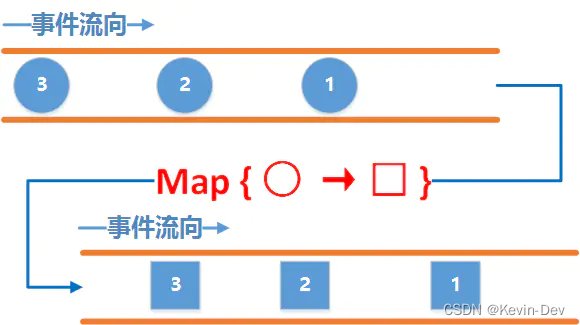
我们当然可以想当然的写成这样：

private void login() {  
 api.login(new LoginRequest())  
 .subscribeOn(Schedulers.io()) //在IO线程进行网络请求  
 .observeOn(AndroidSchedulers.mainThread()) //回到主线程去处理请求结果  
 .subscribe(new Consumer<LoginResponse>() {  
 @Override  
 public void accept(LoginResponse loginResponse) throws Exception {  
 Toast.makeText(MainActivity.this, "登录成功", Toast.LENGTH\_SHORT).show();  
 }  
 }, new Consumer<Throwable>() {  
 @Override  
 public void accept(Throwable throwable) throws Exception {  
 Toast.makeText(MainActivity.this, "登录失败", Toast.LENGTH\_SHORT).show();  
 }  
 });  
}  
  
private void register() {  
 api.register(new RegisterRequest())  
 .subscribeOn(Schedulers.io()) //在IO线程进行网络请求  
 .observeOn(AndroidSchedulers.mainThread()) //回到主线程去处理请求结果  
 .subscribe(new Consumer<RegisterResponse>() {  
 @Override  
 public void accept(RegisterResponse registerResponse) throws Exception {  
 Toast.makeText(MainActivity.this, "注册成功", Toast.LENGTH\_SHORT).show();  
 login(); //注册成功, 调用登录的方法  
 }  
 }, new Consumer<Throwable>() {  
 @Override  
 public void accept(Throwable throwable) throws Exception {  
 Toast.makeText(MainActivity.this, "注册失败", Toast.LENGTH\_SHORT).show();  
 }  
 });  
}

这样的代码能够工作, 但不够优雅, 通过本节的学习, 可以让我们用一种更优雅的方式来解决这个问题。

## map

map 是 RxJava 中最简单的一个变换操作符了, 它的作用就是对上游发送的每一个事件应用一个函数, 使得每一个事件都按照指定的函数去变化. 用事件图表示如下:



图中 map 中的函数作用是将圆形事件转换为矩形事件, 从而导致下游接收到的事件就变为了矩形.用代码来表示这个例子就是:

Observable.*create*(new ObservableOnSubscribe<Integer>() {  
 @Override  
 public void subscribe(ObservableEmitter<Integer> emitter) throws Exception {  
 emitter.onNext(1);  
 emitter.onNext(2);  
 emitter.onNext(3);  
 }  
}).map(new Function<Integer, String>() {  
 @Override  
 public String apply(Integer integer) throws Exception {  
 return "This is result " + integer;  
 }  
}).subscribe(new Consumer<String>() {  
 @Override  
 public void accept(String s) throws Exception {  
 Log.*d*(*TAG*, s);  
 }  
});

运行结果：

D/MainActivity: This is result 1

D/MainActivity: This is result 2

D/MainActivity: This is result 3

## flatMap

flatMap 是一个非常强大的操作符, 先用一个比较难懂的概念说明一下:

flatMap 将一个发送事件的上游 Observable 变换为多个发送事件的 Observables

，然后将它们发射的事件合并后放进一个单独的 Observable 里。

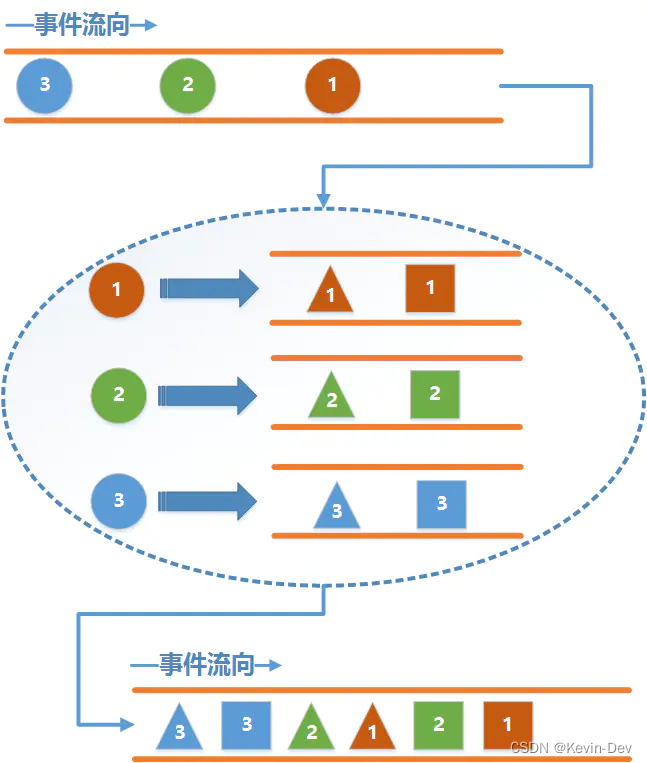
这句话比较难以理解, 我们先通俗易懂的图片来详细的讲解一下, 先来看看整体的一个图片:



先看看上游, 上游发送了三个事件, 分别是1,2,3, 注意它们的颜色.

中间 flatMap 的作用是将圆形的事件转换为一个发送矩形事件和三角形事件的新的上游 Observable .

还是不能理解? 别急, 再来看看分解动作:



这样就很好理解了吧 !!!

上游每发送一个事件, flatMap 都将创建一个新的水管, 然后发送转换之后的新的事件, 下游接收到的就是这些新的水管发送的数据. 这里需要注意的是, flatMap 并不保证事件的顺序, 也就是图中所看到的, 并不是事件1就在事件2的前面. 如果需要保证顺序则需要使用 concatMap.

说了原理, 我们还是来看看实际中的代码如何写吧:

Observable.*create*(new ObservableOnSubscribe<Integer>() {  
 @Override  
 public void subscribe(ObservableEmitter<Integer> emitter) throws Exception {  
 emitter.onNext(1);  
 emitter.onNext(2);  
 emitter.onNext(3);  
 }  
}).flatMap(new Function<Integer, ObservableSource<String>>() {  
 @Override  
 public ObservableSource<String> apply(Integer integer) throws Exception {  
 final List<String> list = new ArrayList<>();  
 for (int i = 0; i < 3; i++) {  
 list.add("I am value " + integer);  
 }  
 return Observable.*fromIterable*(list).delay(10, TimeUnit.*MILLISECONDS*);  
 }  
}).subscribe(new Consumer<String>() {  
 @Override  
 public void accept(String s) throws Exception {  
 Log.*d*(*TAG*, s);  
 }  
});

运行结果：

D/MainActivity: I am value 1

D/MainActivity: I am value 1

D/MainActivity: I am value 1

D/MainActivity: I am value 3

D/MainActivity: I am value 3

D/MainActivity: I am value 3

D/MainActivity: I am value 2

D/MainActivity: I am value 2

D/MainActivity: I am value 2

这里也简单说一下 concatMap 吧, 它和 flatMap 的作用几乎一模一样, 只是它的结果是严格按照上游发送的顺序来发送的, 只是将之前的 flatMap 改为了concatMap, 其余原封不动, 运行结果如下:

D/MainActivity: I am value 1

D/MainActivity: I am value 1

D/MainActivity: I am value 1

D/MainActivity: I am value 2

D/MainActivity: I am value 2

D/MainActivity: I am value 2

D/MainActivity: I am value 3

D/MainActivity: I am value 3

D/MainActivity: I am value 3

## 实践

如何优雅的解决嵌套请求, 只需要用 flatMap 转换一下就行了?

1. 请求接口:

public interface Api {  
 @GET  
 Observable<LoginResponse> login(@Body LoginRequest request);  
  
 @GET  
 Observable<RegisterResponse> register(@Body RegisterRequest request);  
}

可以看到登录和注册返回的都是一个上游Observable, 而我们的flatMap操作符的作用就是把一个Observable转换为另一个Observable, 因此结果就很显而易见了:

api.register(new RegisterRequest()) //发起注册请求  
 .subscribeOn(Schedulers.io()) //在IO线程进行网络请求  
 .observeOn(AndroidSchedulers.mainThread()) //回到主线程去处理请求注册结果  
 .doOnNext(new Consumer<RegisterResponse>() {  
@Override  
public void accept(RegisterResponse registerResponse) throws Exception {  
 //先根据注册的响应结果去做一些操作  
 }  
 })  
 .observeOn(Schedulers.io()) //回到IO线程去发起登录请求  
 .flatMap(new Function<RegisterResponse, ObservableSource<LoginResponse>>() {  
@Override  
public ObservableSource<LoginResponse> apply(RegisterResponse registerResponse) throws Exception {  
 return api.login(new LoginRequest());  
 }  
 })  
 .observeOn(AndroidSchedulers.mainThread()) //回到主线程去处理请求登录的结果  
 .subscribe(new Consumer<LoginResponse>() {  
@Override  
public void accept(LoginResponse loginResponse) throws Exception {  
 Toast.makeText(MainActivity.this, "登录成功", Toast.LENGTH\_SHORT).show();  
 }  
 }, new Consumer<Throwable>() {  
@Override  
public void accept(Throwable throwable) throws Exception {  
 Toast.makeText(MainActivity.this, "登录失败", Toast.LENGTH\_SHORT).show();  
 }  
 });

# 强大的 zip 操作符

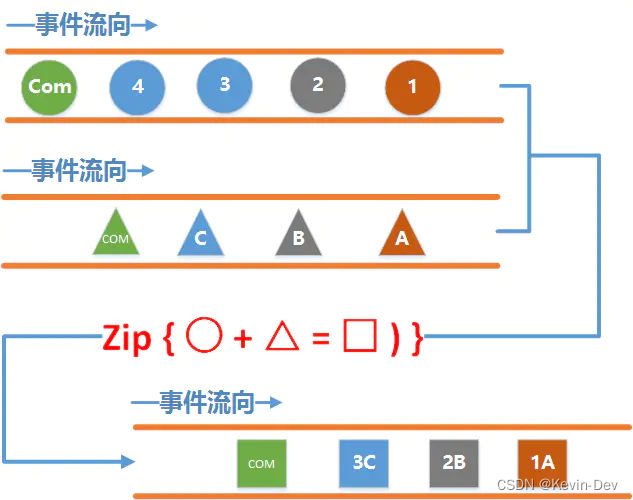
前言

本节学习 zip 这个操作符, 这个操作符也是比较牛逼的东西了， 涉及到的东西也比较多， 主要是一些细节上的东西太多， 通过学习这个操作符，可以为我们下一节的 Backpressure 做个铺垫。

开始

zip 通过一个函数将多个 Observable 发送的事件结合到一起，然后发送这些组合到一起的事件. 它按照严格的顺序应用这个函数。它只发射与发射数据项最少的那个 Observable 一样多的数据。

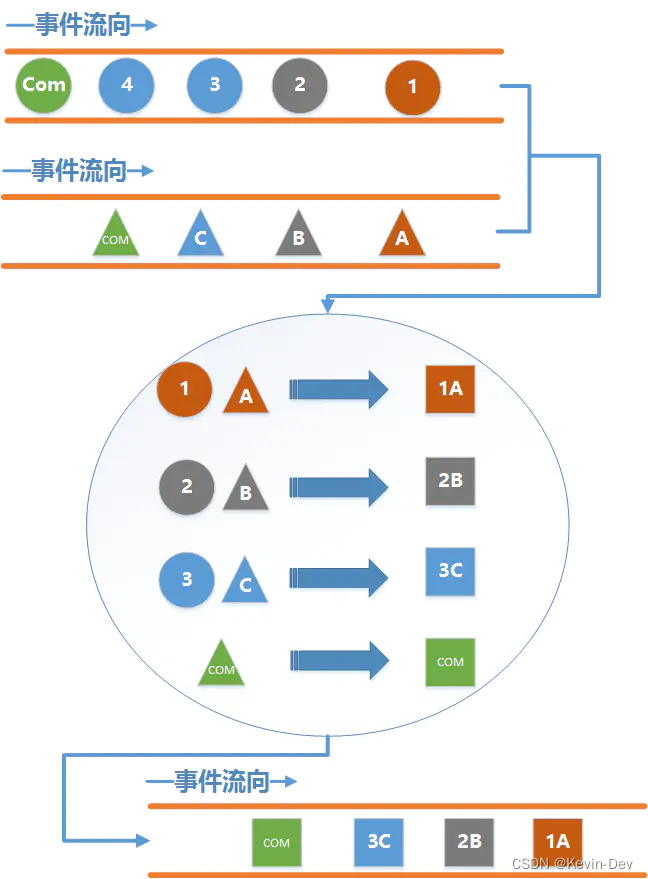
我们再用通俗易懂的图片来解释一下:



从这个图中可以看见, 这次上游和以往不同的是, 我们有两根水管了.

其中一根水管负责发送圆形事件 , 另外一根水管负责发送三角形事件 , 通过Zip操作符, 使得圆形事件 和三角形事件 合并为了一个矩形事件 .

下面我们再来看看分解动作：



通过分解动作我们可以看出:

1. 组合的过程是分别从 两根水管里各取出一个事件 来进行组合, 并且一个事件只能被使用一次, 组合的顺序是严格按照事件发送的顺利 来进行的, 也就是说不会出现圆形1 事件和三角形B 事件进行合并, 也不可能出现圆形2 和三角形A 进行合并的情况.

2. 最终下游收到的事件数量 是和上游中发送事件最少的那一根水管的事件数量 相同. 这个也很好理解, 因为是从每一根水管 里取一个事件来进行合并, 最少的 那个肯定就最先取完 , 这个时候其他的水管尽管还有事件 , 但是已经没有足够的事件来组合了, 因此下游就不会收到剩余的事件了.

分析了大概的原理, 我们还是劳逸结合, 先来看看实际中的代码怎么写吧:

Observable<Integer> observable1 = Observable.*create*(new ObservableOnSubscribe<Integer>() {  
 @Override  
 public void subscribe(ObservableEmitter<Integer> emitter) throws Exception {  
 Log.*d*(*TAG*, "emit 1");  
 emitter.onNext(1);  
 Log.*d*(*TAG*, "emit 2");  
 emitter.onNext(2);  
 Log.*d*(*TAG*, "emit 3");  
 emitter.onNext(3);  
 Log.*d*(*TAG*, "emit 4");  
 emitter.onNext(4);  
 Log.*d*(*TAG*, "emit complete1");  
 emitter.onComplete();  
 }  
});  
  
Observable<String> observable2 = Observable.*create*(new ObservableOnSubscribe<String>() {  
 @Override  
 public void subscribe(ObservableEmitter<String> emitter) throws Exception {  
 Log.*d*(*TAG*, "emit A");  
 emitter.onNext("A");  
 Log.*d*(*TAG*, "emit B");  
 emitter.onNext("B");  
 Log.*d*(*TAG*, "emit C");  
 emitter.onNext("C");  
 Log.*d*(*TAG*, "emit complete2");  
 emitter.onComplete();  
 }  
});   
   
Observable.zip(observable1, observable2, new BiFunction<Integer, String, String>() {  
@Override  
public String apply(Integer integer, String s) throws Exception {  
 return integer + s;  
 }  
 }).subscribe(new Observer<String>() {  
@Override  
public void onSubscribe(Disposable d) {  
 Log.d(TAG, "onSubscribe");  
 }  
  
@Override  
public void onNext(String value) {  
 Log.d(TAG, "onNext: " + value);  
 }  
  
@Override  
public void onError(Throwable e) {  
 Log.d(TAG, "onError");  
 }  
  
@Override  
public void onComplete() {  
 Log.d(TAG, "onComplete");  
 }  
 });

运行结果：

D/MainActivity: onSubscribe

D/MainActivity: emit 1

D/MainActivity: emit 2

D/MainActivity: emit 3

D/MainActivity: emit 4

D/MainActivity: emit complete1

D/MainActivity: emit A

D/MainActivity: onNext: 1A

D/MainActivity: emit B

D/MainActivity: onNext: 2B

D/MainActivity: emit C

D/MainActivity: onNext: 3C

D/MainActivity: emit complete2

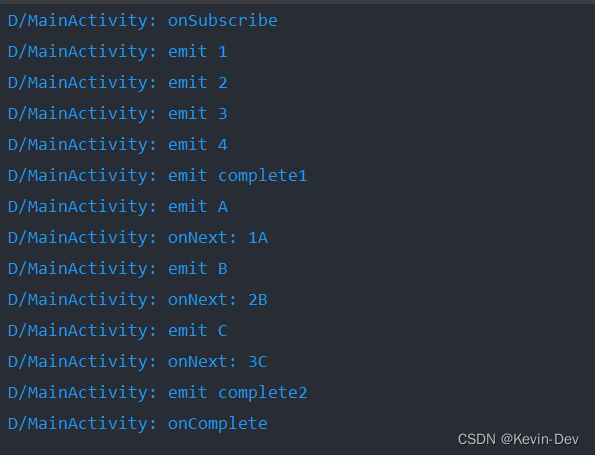
D/MainActivity: onComplete

结果似乎是对的… 但是总感觉什么地方不对劲…

哪儿不对劲呢, 为什么感觉是水管一发送完了之后, 水管二才开始发送啊? 到底是不是呢, 我们来验证一下:

Observable<Integer> observable1 = Observable.*create*(new ObservableOnSubscribe<Integer>() {  
 @Override  
 public void subscribe(ObservableEmitter<Integer> emitter) throws Exception {  
 Log.*d*(*TAG*, "emit 1");  
 emitter.onNext(1);  
 Thread.*sleep*(1000);  
  
 Log.*d*(*TAG*, "emit 2");  
 emitter.onNext(2);  
 Thread.*sleep*(1000);  
  
 Log.*d*(*TAG*, "emit 3");  
 emitter.onNext(3);  
 Thread.*sleep*(1000);  
  
 Log.*d*(*TAG*, "emit 4");  
 emitter.onNext(4);  
 Thread.*sleep*(1000);  
  
 Log.*d*(*TAG*, "emit complete1");  
 emitter.onComplete();  
 }  
});  
  
Observable<String> observable2 = Observable.*create*(new ObservableOnSubscribe<String>() {  
 @Override  
 public void subscribe(ObservableEmitter<String> emitter) throws Exception {  
 Log.*d*(*TAG*, "emit A");  
 emitter.onNext("A");  
 Thread.*sleep*(1000);  
  
 Log.*d*(*TAG*, "emit B");  
 emitter.onNext("B");  
 Thread.*sleep*(1000);  
  
 Log.*d*(*TAG*, "emit C");  
 emitter.onNext("C");  
 Thread.*sleep*(1000);  
  
 Log.*d*(*TAG*, "emit complete2");  
 emitter.onComplete();  
 }  
});   
  
Observable.zip(observable1, observable2, new BiFunction<Integer, String, String>() {  
@Override  
public String apply(Integer integer, String s) throws Exception {  
 return integer + s;  
 }  
 }).subscribe(new Observer<String>() {  
@Override  
public void onSubscribe(Disposable d) {  
 Log.d(TAG, "onSubscribe");  
 }  
  
@Override  
public void onNext(String value) {  
 Log.d(TAG, "onNext: " + value);  
 }  
  
@Override  
public void onError(Throwable e) {  
 Log.d(TAG, "onError");  
 }  
  
@Override  
public void onComplete() {  
 Log.d(TAG, "onComplete");  
 }  
 });

运行结果:

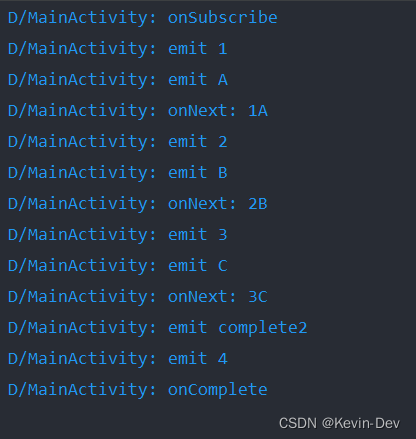


好像真的是先发送的水管一再发送的水管二呢, 为什么会有这种情况呢? 因为我们两根水管都是运行在同一个线程里, 同一个线程里执行代码肯定有先后顺序呀.

因此我们来稍微改一下, 不让他们在同一个线程, 不知道怎么切换线程的, 请掉头看前面几节.

Observable<Integer> observable1 = Observable.*create*(new ObservableOnSubscribe<Integer>() {  
 @Override  
 public void subscribe(ObservableEmitter<Integer> emitter) throws Exception {  
 Log.*d*(*TAG*, "emit 1");  
 emitter.onNext(1);  
 Thread.*sleep*(1000);  
  
 Log.*d*(*TAG*, "emit 2");  
 emitter.onNext(2);  
 Thread.*sleep*(1000);  
  
 Log.*d*(*TAG*, "emit 3");  
 emitter.onNext(3);  
 Thread.*sleep*(1000);  
  
 Log.*d*(*TAG*, "emit 4");  
 emitter.onNext(4);  
 Thread.*sleep*(1000);  
  
 Log.*d*(*TAG*, "emit complete1");  
 emitter.onComplete();  
 }  
}).subscribeOn(Schedulers.*io*());  
  
Observable<String> observable2 = Observable.*create*(new ObservableOnSubscribe<String>() {  
 @Override  
 public void subscribe(ObservableEmitter<String> emitter) throws Exception {  
 Log.*d*(*TAG*, "emit A");  
 emitter.onNext("A");  
 Thread.*sleep*(1000);  
  
 Log.*d*(*TAG*, "emit B");  
 emitter.onNext("B");  
 Thread.*sleep*(1000);  
  
 Log.*d*(*TAG*, "emit C");  
 emitter.onNext("C");  
 Thread.*sleep*(1000);  
  
 Log.*d*(*TAG*, "emit complete2");  
 emitter.onComplete();  
 }  
}).subscribeOn(Schedulers.*io*());  
  
Observable.*zip*(observable1, observable2, new BiFunction<Integer, String, String>() {  
 @Override  
 public String apply(Integer integer, String s) throws Exception {  
 return integer + s;  
 }  
}).subscribe(new Observer<String>() {  
 @Override  
 public void onSubscribe(Disposable d) {  
 Log.*d*(*TAG*, "onSubscribe");  
 }  
  
 @Override  
 public void onNext(String value) {  
 Log.*d*(*TAG*, "onNext: " + value);  
 }  
  
 @Override  
 public void onError(Throwable e) {  
 Log.*d*(*TAG*, "onError");  
 }  
  
 @Override  
 public void onComplete() {  
 Log.*d*(*TAG*, "onComplete");  
 }  
});

输出结果：



实践

比如一个界面需要展示用户的一些信息, 而这些信息分别要从两个服务器接口中获取, 而只有当两个都获取到了之后才能进行展示, 这个时候就可以用 zip了:

首先分别定义这两个请求接口:

public interface Api {  
 @GET  
 Observable<UserBaseInfoResponse> getUserBaseInfo(@Body UserBaseInfoRequest request);  
  
 @GET  
 Observable<UserExtraInfoResponse> getUserExtraInfo(@Body UserExtraInfoRequest request);  
}

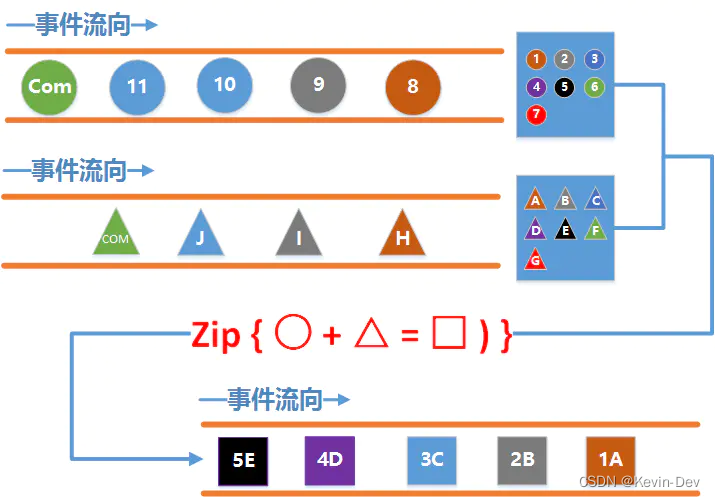
接着用 zip 来打包请求:

Observable<UserBaseInfoResponse> observable1 =api.getUserBaseInfo(new UserBaseInfoRequest()).subscribeOn(Schedulers.*io*());  
Observable<UserExtraInfoResponse> observable2 =api.getUserExtraInfo(new UserExtraInfoRequest()).subscribeOn(Schedulers.*io*());   
   
Observable.zip(observable1, observable2,  
 new BiFunction<UserBaseInfoResponse, UserExtraInfoResponse, UserInfo>() {  
@Override  
public UserInfo apply(UserBaseInfoResponse baseInfo,  
 UserExtraInfoResponse extraInfo) throws Exception {  
 return new UserInfo(baseInfo, extraInfo);  
 }  
 }).observeOn(AndroidSchedulers.mainThread())  
 .subscribe(new Consumer<UserInfo>() {  
@Override  
public void accept(UserInfo userInfo) throws Exception {  
 //do something;   
 }  
 });

# Backpressure

## 开始

上一节中我们说到 zip 可以将多个上游发送的事件组合起来发送给下游, 那大家有没有想过一个问题, 如果其中一个水管 A 发送事件特别快, 而另一个水管 B 发送事件特别慢, 那就可能出现这种情况, 发得快的水管 A 已经发送了 1000 个事件了, 而发的慢的水管 B 才发一个出来, 组合了一个之后水管 A 还剩 999 个事件, 这些事件需要继续等待水管 B 发送事件出来组合, 那么这么多的事件是放在哪里的呢? 总有一个地方保存吧? 没错, zip 给我们的每一根水管都弄了一个水缸 , 用来保存这些事件, 用通俗易懂的图片来表示就是:

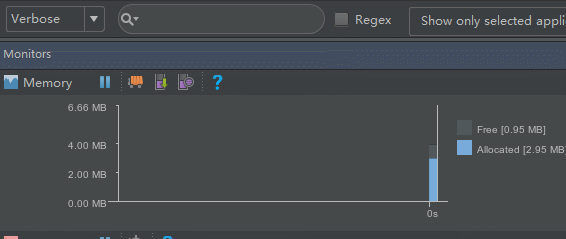


如图中所示, 其中蓝色的框框就是zip给我们的水缸! 它将每根水管发出的事件保存起来, 等两个水缸都有事件了之后就分别从水缸中取出一个事件来组合, 当其中一个水缸是空的时候就处于等待的状态.

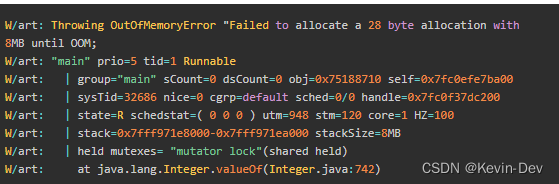
好了回到正题上来, 这个水缸有大小限制吗? 要是一直往里存会怎样? 我们来看个例子:

Observable<Integer> observable1 = Observable.*create*(new ObservableOnSubscribe<Integer>() {  
 @Override  
 public void subscribe(ObservableEmitter<Integer> emitter) throws Exception {  
 for (int i = 0; ; i++) { //无限循环发事件   
 emitter.onNext(i);  
 }  
 }  
}).subscribeOn(Schedulers.*io*());  
  
Observable<String> observable2 = Observable.*create*(new ObservableOnSubscribe<String>() {  
 @Override  
 public void subscribe(ObservableEmitter<String> emitter) throws Exception {  
 emitter.onNext("A");  
 }  
}).subscribeOn(Schedulers.*io*());  
  
Observable.*zip*(observable1, observable2, new BiFunction<Integer, String, String>() {  
 @Override  
 public String apply(Integer integer, String s) throws Exception {  
 return integer + s;  
 }  
}).observeOn(AndroidSchedulers.mainThread()).subscribe(new Consumer<String>() {  
 @Override  
 public void accept(String s) throws Exception {  
 Log.*d*(*TAG*, s);  
 }  
}, new Consumer<Throwable>() {  
 @Override  
 public void accept(Throwable throwable) throws Exception {  
 Log.*w*(*TAG*, throwable);  
 }  
});

在这个例子中, 我们分别创建了两根水管, 第一根水管用机器指令的执行速度来无限循环发送事件, 第二根水管随便发送点什么, 由于我们没有发送Complete事件, 因此第一根水管会一直发事件到它对应的水缸里去, 我们来看看运行结果是什么样.



内存占用以斜率为1的直线迅速上涨, 几秒钟就 300 多 M , 最终报出了 OOM :



出现这种情况肯定是我们不想看见的, 这里就可以引出我们的 Backpressure了, 所谓的 Backpressure 其实就是为了控制流量, 水缸存储的能力毕竟有限, 因此我们还得从源头去解决问题, 既然你发那么快, 数据量那么大, 那我就想办法不让你发那么快呗.

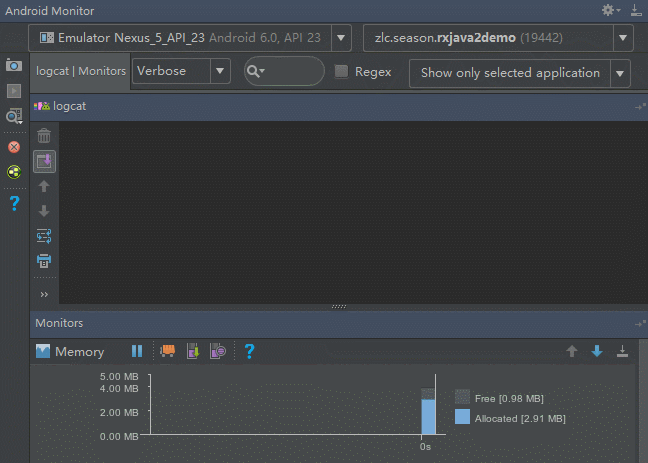
那么这个源头到底在哪里, 究竟什么时候会出现这种情况, 这里只是说的Zip这一个例子, 其他的地方会出现吗? 带着这个问题我们来探究一下.

我们让事情变得简单一点, 从一个单一的 Observable 说起.

来看段代码:

Observable.*create*(new ObservableOnSubscribe<Integer>() {  
 @Override  
 public void subscribe(ObservableEmitter<Integer> emitter) throws Exception {  
 for (int i = 0; ; i++) { //无限循环发事件   
 emitter.onNext(i);  
 }  
 }  
}).subscribe(new Consumer<Integer>() {  
 @Override  
 public void accept(Integer integer) throws Exception {  
 Thread.*sleep*(2000);  
 Log.*d*(*TAG*, "" + integer);  
 }  
});

这段代码很简单, 上游同样无限循环的发送事件, 在下游每次接收事件前延时2秒. 上下游工作在同一个线程里, 来看下运行结果:



为什么呢, 因为上下游工作在同一个线程呀骚年们! 这个时候上游每次调用 emitter.onNext(i)其实就相当于直接调用了Consumer中的:

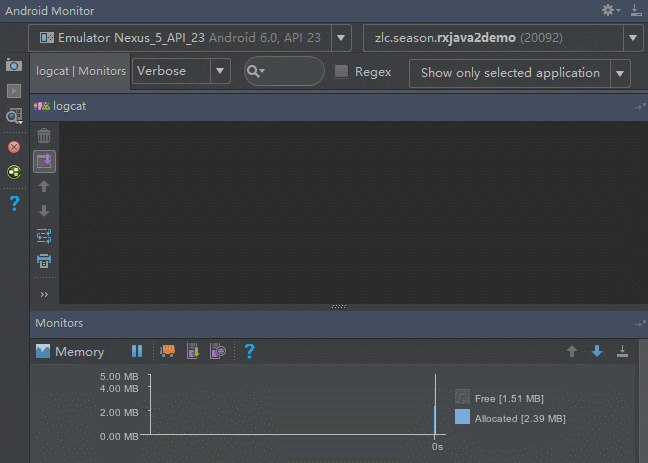
public void accept(Integer integer) throws Exception {  
 Thread.*sleep*(2000);  
 Log.*d*(*TAG*, "" + integer);  
}

所以这个时候其实就是上游每延时2秒发送一次. 最终的结果也说明了这一切.

那我们加个线程呢, 改成这样:

Observable.*create*(new ObservableOnSubscribe<Integer>() {  
 @Override  
 public void subscribe(ObservableEmitter<Integer> emitter) throws Exception {  
 for (int i = 0; ; i++) { //无限循环发事件   
 emitter.onNext(i);  
 }  
 }  
 }).subscribeOn(Schedulers.*io*())  
 .observeOn(AndroidSchedulers.mainThread())  
 .subscribe(new Consumer<Integer>() {  
 @Override  
 public void accept(Integer integer) throws Exception {  
 Thread.*sleep*(2000);  
 Log.*d*(*TAG*, "" + integer);  
 }  
 });

这个时候把上游切换到了IO线程中去, 下游到主线程去接收, 来看看运行结果:



可以看到, 给上游加了个线程之后, 它就像脱缰的野马一样, 内存又爆掉了.

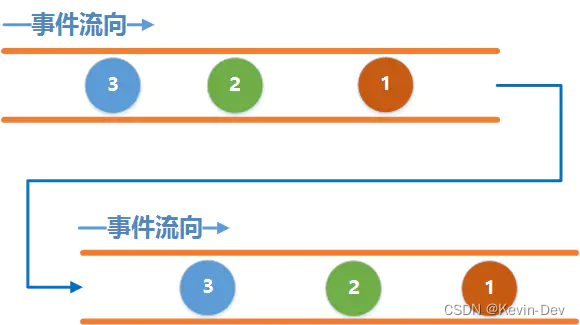
为什么不加线程和加上线程区别这么大呢, 这就涉及了 同步和异步 的知识了.

当上下游工作在同一个线程中时, 这时候是一个同步的订阅关系, 也就是说上游每发送一个事件必须等到下游接收处理完了以后才能接着发送下一个事件.

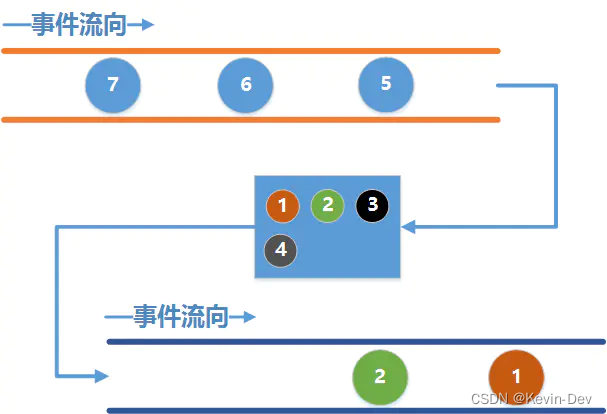
当上下游工作在不同的线程中时, 这时候是一个异步的订阅关系, 这个时候上游发送数据不需要等待下游接收, 为什么呢, 因为两个线程并不能直接进行通信, 因此上游发送的事件并不能直接到下游里去, 这个时候就需要一个田螺姑娘来帮助它们俩, 这个田螺姑娘就是我们刚才说的水缸 ! 上游把事件发送到水缸里去, 下游从水缸里取出事件来处理, 因此, 当上游发事件的速度太快, 下游取事件的速度太慢, 水缸就会迅速装满, 然后溢出来, 最后就OOM了.

这两种情况用图片来表示如下:

同步:



异步:



从图中我们可以看出, 同步和异步的区别仅仅在于是否有水缸.

相信通过这个例子大家对线程之间的通信也有了比较清楚的认知和理解.

源头找到了, 只要有水缸, 就会出现上下游发送事件速度不平衡的情况, 因此当我们以后遇到这种情况时, 仔细思考一下水缸在哪里, 找到水缸, 你就找到了解决问题的办法.

# 如何去解决 OOM

在上一节中, 我们找到了上下游流速不均衡的源头 , 在这一节里我们将学习如何去治理它 . 可能很多看过其他人写的文章的朋友都会觉得只有Flowable才能解决 , 所以大家对这个Flowable都抱有很大的期许 , 其实呐 , 你们毕竟图样图森破 , 今天我们先抛开Flowable, 仅仅依靠我们自己的双手和智慧 , 来看看我们如何去治理 , 通过本节的学习之后我们再来看Flowable, 你会发现它其实并没有想象中那么牛叉, 它只是被其他人过度神化了.

## 开始

我们接着来看上一节的这个例子:

Observable.*create*(new ObservableOnSubscribe<Integer>() {  
 @Override  
 public void subscribe(ObservableEmitter<Integer> emitter) throws Exception {  
 for (int i = 0; ; i++) { //无限循环发送事件  
 emitter.onNext(i);  
 }  
 }  
 }).subscribeOn(Schedulers.*io*())  
 .observeOn(AndroidSchedulers.mainThread())  
 .subscribe(new Consumer<Integer>() {  
 @Override  
 public void accept(Integer integer) throws Exception {  
 Log.*d*(*TAG*, "" + integer);  
 }  
 });

上一节中我们看到了它的运行结果是直接爆掉了内存, 也明白它为什么就爆掉了内存, 那么我们能做些什么, 才能不让这种情况发生呢.

之前我们说了, 上游发送的所有事件都放到水缸里了, 所以瞬间水缸就满了, 那我们可以只放我们需要的事件到水缸里呀, 只放一部分数据到水缸里, 这样不就不会溢出来了吗, 因此, 我们把上面的代码修改一下:

Observable.*create*(new ObservableOnSubscribe<Integer>() {  
 @Override  
 public void subscribe(ObservableEmitter<Integer> emitter) throws Exception {  
 for (int i = 0; ; i++) {  
 emitter.onNext(i);  
 }  
 }  
 }).subscribeOn(Schedulers.*io*())  
 .filter(new Predicate<Integer>() {  
 @Override  
 public boolean test(Integer integer) throws Exception {  
 return integer % 10 == 0;  
 }  
 })  
 .observeOn(AndroidSchedulers.mainThread())  
 .subscribe(new Consumer<Integer>() {  
 @Override  
 public void accept(Integer integer) throws Exception {  
 Log.*d*(*TAG*, "" + integer);  
 }  
 });

虽然内存依然在增长, 但是增长速度相比之前, 已经减少了太多了, 至少在我录完GIF之前还没有爆掉内存, 大家可以试着改成能被100整除试试.

可以看到, 通过减少进入水缸的事件数量的确可以缓解上下游流速不均衡的问题, 但是力度还不够, 我们再来看一段代码:

Observable.*create*(new ObservableOnSubscribe<Integer>() {  
 @Override  
 public void subscribe(ObservableEmitter<Integer> emitter) throws Exception {  
 for (int i = 0; ; i++) {  
 emitter.onNext(i);  
 }  
 }  
 }).subscribeOn(Schedulers.*io*())  
 .sample(2, TimeUnit.*SECONDS*) //sample取样  
 .observeOn(AndroidSchedulers.mainThread())  
 .subscribe(new Consumer<Integer>() {  
 @Override  
 public void accept(Integer integer) throws Exception {  
 Log.*d*(*TAG*, "" + integer);  
 }  
 });

这里用了一个sample操作符, 简单做个介绍, 这个操作符每隔指定的时间就从上游中取出一个事件发送给下游. 这里我们让它每隔2秒取一个事件给下游, 来看看这次的运行结果吧:

这次我们可以看到, 虽然上游仍然一直在不停的发事件, 但是我们只是每隔一定时间取一个放进水缸里, 并没有全部放进水缸里, 因此这次内存仅仅只占用了5M.

前面这两种方法归根到底其实就是减少放进水缸的事件的数量, 是以数量取胜, 但是这个方法有个缺点, 就是丢失了大部分的事件.

那么我们换一个角度来思考, 既然上游发送事件的速度太快, 那我们就适当减慢发送事件的速度, 从速度上取胜, 听上去不错, 我们来试试:

Observable.*create*(new ObservableOnSubscribe<Integer>() {  
 @Override  
 public void subscribe(ObservableEmitter<Integer> emitter) throws Exception {  
 for (int i = 0; ; i++) {  
 emitter.onNext(i);  
 Thread.*sleep*(2000); //每次发送完事件延时2秒  
 }  
 }  
 }).subscribeOn(Schedulers.*io*())  
 .observeOn(AndroidSchedulers.mainThread())  
 .subscribe(new Consumer<Integer>() {  
 @Override  
 public void accept(Integer integer) throws Exception {  
 Log.*d*(*TAG*, "" + integer);  
 }  
 });

可以看到, 我们给上游加上延时了之后, 瞬间一头发情的公牛就变得跟只小绵羊一样, 如此温顺, 如此平静, 如此平稳的内存线, 美妙极了. 而且事件也没有丢失, 上游通过适当的延时, 不但减缓了事件进入水缸的速度, 也可以让下游有充足的时间从水缸里取出事件来处理 , 这样一来, 就不至于导致大量的事件涌进水缸, 也就不会OOM啦.

到目前为止, 我们没有依靠任何其他的工具, 就轻易解决了上下游流速不均衡的问题.

因此我们总结一下, 本节中的治理的办法就两种:

一是从数量上进行治理, 减少发送进水缸里的事件

二是从速度上进行治理, 减缓事件发送进水缸的速度

大家一定没忘记, 在上一节还有个Zip的例子, 这个例子也爆了我们的内存, 现学现用, 我们用刚学到的办法来试试能不能惩奸除恶, 先来看看第一种办法.

先来减少进入水缸的事件的数量:

Observable<Integer> observable1 = Observable.*create*(new ObservableOnSubscribe<Integer>() {  
 @Override  
 public void subscribe(ObservableEmitter<Integer> emitter) throws Exception {  
 for (int i = 0; ; i++) {  
 emitter.onNext(i);  
 }  
 }  
}).subscribeOn(Schedulers.*io*()).sample(2, TimeUnit.*SECONDS*); //进行sample采样  
  
Observable<String> observable2 = Observable.*create*(new ObservableOnSubscribe<String>() {  
 @Override  
 public void subscribe(ObservableEmitter<String> emitter) throws Exception {  
 emitter.onNext("A");  
 }  
}).subscribeOn(Schedulers.*io*());  
  
Observable.*zip*(observable1, observable2, new BiFunction<Integer, String, String>() {  
 @Override  
 public String apply(Integer integer, String s) throws Exception {  
 return integer + s;  
 }  
}).observeOn(AndroidSchedulers.mainThread()).subscribe(new Consumer<String>() {  
 @Override  
 public void accept(String s) throws Exception {  
 Log.*d*(*TAG*, s);  
 }  
}, new Consumer<Throwable>() {  
 @Override  
 public void accept(Throwable throwable) throws Exception {  
 Log.*w*(*TAG*, throwable);  
 }  
});

这次我们来减缓速度:

Observable<Integer> observable1 = Observable.*create*(new ObservableOnSubscribe<Integer>() {  
 @Override  
 public void subscribe(ObservableEmitter<Integer> emitter) throws Exception {  
 for (int i = 0; ; i++) {  
 emitter.onNext(i);  
 Thread.*sleep*(2000); //发送事件之后延时2秒  
 }  
 }  
}).subscribeOn(Schedulers.*io*());  
  
Observable<String> observable2 = Observable.*create*(new ObservableOnSubscribe<String>() {  
 @Override  
 public void subscribe(ObservableEmitter<String> emitter) throws Exception {  
 emitter.onNext("A");  
 }  
}).subscribeOn(Schedulers.*io*());  
  
Observable.*zip*(observable1, observable2, new BiFunction<Integer, String, String>() {  
 @Override  
 public String apply(Integer integer, String s) throws Exception {  
 return integer + s;  
 }  
}).observeOn(AndroidSchedulers.mainThread()).subscribe(new Consumer<String>() {  
 @Override  
 public void accept(String s) throws Exception {  
 Log.*d*(*TAG*, s);  
 }  
}, new Consumer<Throwable>() {  
 @Override  
 public void accept(Throwable throwable) throws Exception {  
 Log.*w*(*TAG*, throwable);  
 }  
});

通过本节的学习, 大家应该对如何处理上下游流速不均衡已经有了基本的认识了, 大家也可以看到, 我们并没有使用 Flowable, 所以很多时候仔细去分析问题, 找到问题的原因, 从源头去解决才是最根本的办法. 后面我们讲到 Flowable 的时候, 大家就会发现它其实没什么神秘的, 它用到的办法和我们本节所讲的基本上是一样的, 只是它稍微做了点封装.

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