# 01.RxJava概念与观察者设计模式。

1、基于事件流编程 -----> 事件流 这三个字要理解

起点（事件源）--> 终点（接收事件）

观察者设计模式的一种演变

依赖包

rxjava

rxandroid

2、

Observable 被观察者 发送改变

|| 订阅

Observer 观察者 观察 观察

## 链接：标准的观察者模式

1.起点 和 终点，一旦满足 起点 和 终点 这样的需求，都可以使用RxJava来实现。

2.标准中的观察者设计模式，一个被观察者 ---- 多个观察者 多次注册。

3.RxJava是改装的观察者设计模式，一个订阅(注册) 一个观察者。

////////////////////////////////////////////////////////////////////////////////////////////////////

# 02.RxJava上游和下游。

1.上游 Observable 被观察者， 下游 Observer 观察者。

2.ObservableEmitter<Integer> emitter 发射器 发射事件。

3.拆分来写的，链式调用。

4.RxJava流程1，大致流程，同学们去看代码。

5.RxJava流程2，3个结论。

6.RxJava切断下游，让下游不再接收上游发射的事件。

////////////////////////////////////////////////////////////////////////////////////////////////////

# 03.RxJava创建型操作符 -- 专门创建 被观察者/上游/Observable。

观察者：下游，接收事件 完整版本Observer 简化版Consumer

（1）create：使用者自己发射事件

Observable.create(new ObservableOnSubscribe<String>() {......})

.subscribe......

（2）just 内部自己发射的，单一对象

Observable.just("A", "B") // 内部会去发射 A B

.subscribe......

（3）fromArray 内部自己发射的，数集对象

String[] strings = {"1", "2", "3"}; // 内部会去发射 1 2 3

// 上游 被观察者

Observable.fromArray(strings)

.subscribe......

1. empty：内部自己发射的 ，下游默认是Object，无法发出有值事件，只会发射 onComplete

Observable.empty() // 内部一定会只调用 发射 onComplete 完毕事件

.subscribe......

1. range：内部自己发射的，start 1 累加 count 5 最后结果：1 2 3 4 5

// Observable.range(1, 8) // 1 2 3 4 5 6 7 8 从1开始加 数量共8个

Observable.range(80, 5) // 80开始 80 81 82 83 84 加 数量共5个

.subscribe......

|  |
| --- |
| /\*\*  \* create 操作符 创建 Observable  \*  \* @param view  \*/  public void r01(TextView view) {  // 上游  Observable.create(new ObservableOnSubscribe<String>() {  @Override  public void subscribe(ObservableEmitter<String> e) throws Exception {  // 上游发射的  e.onNext("A"); // 使用者自己发射  }  })  .delay(10, TimeUnit.SECONDS)  // 订阅  .subscribe(  // 下游  new Observer<String>() {  @Override  public void onSubscribe(Disposable d) {  //将这个d拿出来用，d.dispose();  }  @Override  public void onNext(String s) {  Log.d(TAG, "下游接收 onNext: " + s);  view.setText("我改变了" + s);  }  @Override  public void onError(Throwable e) {  }  @Override  public void onComplete() {  }  });  }  /\*\*  \* just 操作符 创建 Observable  \*  \* @param view  \*/  public void r02(View view) {  // 上游  Observable.just("A", "B") // 内部会去发射 A B  // 订阅  .subscribe(  // 下游  new Observer<String>() {  @Override  public void onSubscribe(Disposable d) {  }  @Override  public void onNext(String s) {  Log.d(TAG, "onNext: " + s);  ((TextView)view).setText("我改变了" + s);  }  @Override  public void onError(Throwable e) {  }  @Override  public void onComplete() {  }  }  );  }  /\*\*  \* fromArray 操作符 创建 Observable  \*  \* @param view  \*/  public void r03(View view) {  /\*  String[] strings = {"1", "2", "3"}; // 内部会去发射 1 2 3  // 上游 被观察者  Observable.fromArray(strings)  // 订阅  .subscribe(  // 下游  new Observer<String>() {  @Override  public void onSubscribe(Disposable d) {  }  @Override  public void onNext(String s) {  Log.d(TAG, "onNext: " + s);  }  @Override  public void onError(Throwable e) {  }  @Override  public void onComplete() {  }  }  );  \*/  String[] strings = {"张三", "李四", "王五"};  // for  for (String string : strings) {  Log.d(TAG, "r03: " + string);  }  Log.d(TAG, "r03: ----- ");  // RxJava  Observable.fromArray(strings)  .subscribe(new Consumer<String>() {  @Override  public void accept(String s) throws Exception {  Log.d(TAG, "accept: " + s);  }  });  }  /\*\*  \* 为什么只支持Object ？  \* 上游没有发射有值得事件，下游无法确定类型，默认Object，RxJava泛型 泛型默认类型==Object  \* <p>  \* 做一个耗时操作，不需要任何数据来刷新UI， empty的使用场景之一  \*  \* @param view  \*/  public void r04(View view) {  // 上游无法指定 事件类型  Observable.empty() // 内部一定会只调用 发射 onComplete 完毕事件  // 订阅  .subscribe(  // 下游 观察者  new Observer<Object>() {  @Override  public void onSubscribe(Disposable d) {  }  @Override  public void onNext(Object integer) {  // 没有事件可以接受  Log.d(TAG, "onNext: " + integer);  }  @Override  public void onError(Throwable e) {  }  @Override  public void onComplete() {  Log.d(TAG, "onComplete: ");  // 隐藏 加载框...  }  }  /\*// 简化版 观察者  new Consumer<Object>() {  @Override  public void accept(Object o) throws Exception {  // 接受不到  // 没有事件可以接受  Log.d(TAG, "accept: " + o);  }  }\*/  );  }  public void r05(View view) {  // 上游 range内部会去发射  // Observable.range(1, 8) // 1 2 3 4 5 6 7 8 从1开始加 数量共8个  Observable.range(80, 5) // 80开始 80 81 82 83 84 加 数量共5个  // 订阅  .subscribe(  // 下游  new Consumer<Integer>() {  @Override  public void accept(Integer integer) throws Exception {  Log.d(TAG, "accept: " + integer);  }  });  }  } |

////////////////////////////////////////////////////////////////////////////////////////////////////

# 04.RxJava变换型操作符。

上游 -------> 变换操作(往右边流向的时候，进行变换) ----------> 下游

1. map 把上一层Int Int变换String 观察者String类型。

// .map(new Function<Integer, String>() {...})

//通过map将int转变为string

1. flatMap 把上一层Int Int变换ObservableSource<String>{还可以再次发射多次事件} 观察者String类型。 不排序的

//将上一层的事件流事件 转化为 被观察者， 不排序的

1. concatMap 把上一层Int Int变换ObservableSource<Bitmap>{还可以再次发射多次事件} 观察者Bitmap类型。 排序的

//将上一层的事件流事件 转化为 被观察者， 排序的

1. groupBy 把上一层Int Int变换String(高端配置电脑) 观察者GroupedObservable类型 {key="高端", 细节再包裹一层}

//分组 根据不同的输入return不同的组名。 有点像switch case

5.buffer 100个事件 Integer .buffer(20) 观察者List<Integer>==五个集合

//分批次发送

map:获取一个新元素（原本几个元素还是几个元素） flatmap 获取一个或者多个新元素（比原来的元素多）。

map：map方法返回的是一个object，map将流中的当前元素替换为此返回值；

flatMap：flatMap方法返回的是一个stream，flatMap将流中的当前元素替换为此返回流拆解的流元素；

|  |
| --- |
| /\*\*  \* map 变换 操作符  \* @param view  \*/  public void r01(View view) {  // 上游  Observable.just(1) // 发射 1  // 在上游和下游之间 变换  .map(new Function<Integer, String>() {  @Override  public String apply(Integer integer) throws Exception {  // 1  Log.d(TAG, "map1 apply: " + integer);  return "【" + integer + "】";  }  })  .map(new Function<String, Bitmap>() {  @Override  public Bitmap apply(String s) throws Exception {  // s == 【" + integer + "】  Log.d(TAG, "map2 apply: " + s);  return Bitmap.createBitmap(1920, 1280, Bitmap.Config.ARGB\_8888);  // return null; // 如果返回null，下游无法接收  }  })  // 订阅  .subscribe(  // 下游  new Observer<Bitmap>() {  @Override  public void onSubscribe(Disposable d) {  }  @Override  public void onNext(Bitmap string) {  Log.d(TAG, "下游 onNext: " + string);  }  @Override  public void onError(Throwable e) {  }  @Override  public void onComplete() {  }  }  );  }  /\*\*  \* flatMap 变换 操作符  \* @param view  \*/  public void r02(View view) {  // 上游  Observable.just(1111)  // 变换操作符  .flatMap(new Function<Integer, ObservableSource<String>>() {  @Override  public ObservableSource<String> apply(final Integer integer) throws Exception {  // integer == 111  // ObservableSource == 可以再次手动发送事件  return Observable.create(new ObservableOnSubscribe<String>() {  @Override  public void subscribe(ObservableEmitter<String> e) throws Exception {  e.onNext(integer + "1flatMap变换操作符");  e.onNext(integer + "2flatMap变换操作符");  e.onNext(integer + "3flatMap变换操作符");  }  });  }  })  // 订阅  .subscribe(  // 下游  new Consumer<String>() {  @Override  public void accept(String string) throws Exception {  Log.d(TAG, "下游接收 变换操作符 发射的事件 accept: " + string);  }  });  }  /\*\*  \* 体现 flatMap 变换 操作符 是不排序的  \* @param view  \*/  public void r03(View view) {  // 上游  Observable.create(new ObservableOnSubscribe<String>() {  @Override  public void subscribe(ObservableEmitter<String> e) throws Exception {  e.onNext("步惊云"); // String  e.onNext("雄霸");  e.onNext("李四");  e.onComplete();  }  })  .flatMap(new Function<String, ObservableSource<?>>() { // ? 通配符 默认Object  @Override  public ObservableSource<?> apply(String s) throws Exception {  List<String> list = new ArrayList<>();  for (int i = 0; i < 3; i++) {  list.add(s + " 下标：" + (1 + i));  }  return Observable.fromIterable(list).delay(5, TimeUnit.SECONDS); // 创建型操作符，创建被观察者  }  })  // 订阅  .subscribe(/\*new Consumer<Object>() { // 下游  @Override  public void accept(Object s) throws Exception {  Log.d(TAG, "下游 accept: " + s);  }  }\*/  new Observer<Object>() {  @Override  public void onSubscribe(Disposable d) {  }  @Override  public void onNext(Object o) {  Log.d(TAG, "下游 onNext: " + o);  }  @Override  public void onError(Throwable e) {  }  @Override  public void onComplete() {  }  }  );  }  /\*\*  \* 体现 concatMap 变换操作符 排序的  \* @param view  \*/  public void r04(View view) {  // 上游  Observable.just("A", "B", "C")  // 变换操作符  .concatMap(new Function<String, ObservableSource<?>>() {  @Override  public ObservableSource<?> apply(String s) throws Exception {  List<String> list = new ArrayList<>();  for (int i = 0; i < 3; i++) {  list.add(s + " 下标：" + (1 + i));  }  return Observable.fromIterable(list).delay(5, TimeUnit.SECONDS); // 创建型操作符，创建被观察者  }  })  .subscribe(new Consumer<Object>() { // 下游  @Override  public void accept(Object s) throws Exception {  Log.d(TAG, "accept: " + s);  }  });  }  /\*\*  \* 分组变换 groupBy  \* @param view  \*/  public void r05(View view) {  // 上游  Observable.just(6000, 7000, 8000, 9000, 10000, 14000)  // 变换  .groupBy(new Function<Integer, String>() {  @Override  public String apply(Integer integer) throws Exception {  return integer > 8000 ? "高端配置电脑" : "中端配置电脑"; // 分组  }  })  // 订阅  /\*.subscribe(new Consumer<String>() { // 下游 使用 groupBy  @Override  public void accept(String string) throws Exception {  }  });\*/  // 使用groupBy下游是 有标准的  .subscribe(new Consumer<GroupedObservable<String, Integer>>() {  @Override  public void accept(final GroupedObservable<String, Integer> groupedObservable) throws Exception {  Log.d(TAG, "accept: " + groupedObservable.getKey());  // 以上还不能把信息给打印全面，只是拿到了，分组的key  // 输出细节，还需要再包裹一层  // 细节 GroupedObservable 被观察者  groupedObservable.subscribe(new Consumer<Integer>() {  @Override  public void accept(Integer integer) throws Exception {  Log.d(TAG, "accept: 类别：" + groupedObservable.getKey() + " 价格：" + integer);  }  });  }  });  }  /\*\*  \* 很多的数据，不想全部一起发射出去，分批次，先缓存到Buffer  \* @param view  \*/  public void r06(View view) {  // 上游  Observable.create(new ObservableOnSubscribe<Integer>() {  @Override  public void subscribe(ObservableEmitter<Integer> e) throws Exception {  for (int i = 0; i < 100; i++) {  e.onNext(i);  }  e.onComplete();  }  })  // 变换 buffer  .buffer(20)  .subscribe(new Consumer<List<Integer>>() {  @Override  public void accept(List<Integer> integer) throws Exception {  Log.d(TAG, "accept: " + integer);  }  });  } |

////////////////////////////////////////////////////////////////////////////////////////////////////

# 05.RxJava过滤型操作符。

上游 -------> 过滤操作(往右边流向的时候，进行过滤) ----------> 下游

1. filter 如果是false全部都发射给下游，如果是true，全部都不发射给下游。

设置new Predicate(); 根据不同回调的返回值，判断是否过滤。

1. take ：只有再定时器运行基础上 加入take过滤操作符，才有take过滤操作符的价值。

take执行次数过滤

1. distinct过滤重复事件。

//去重

1. elementAl 指定发射事件内容，如果没有指定，有默认的事件。

//.elementAt(2, "默认经") // 指定下标输出 事件

|  |
| --- |
| /\*\*  \* filter 过滤  \* 需求：过滤掉 哪些不合格的奶粉，输出哪些合格的奶粉  \* @param view  \*/  public void r01(View view) {  // 上游  Observable.just("三鹿", "合生元", "飞鹤")  .filter(new Predicate<String>() {  @Override  public boolean test(String s) throws Exception {  // return true; // 不去过滤，默认全部都会打印  // return false; // 如果是false 就全部都不会打印  if ("三鹿".equals(s)) {  return false; // 不合格  }  return true;  }  })  // 订阅  .subscribe(new Consumer<String>() { // 下游  @Override  public void accept(String s) throws Exception {  Log.d(TAG, "accept: " + s);  }  });  }  /\*\*  \* take过滤操作符  \* @param view  \*/  public void r02(View view) {  // 定时器 运行 只有在定时器运行基础上 加入take过滤操作符，才有take过滤操作符的价值  // 上游  Observable.interval(2, TimeUnit.SECONDS)  // 增加过滤操作符，停止定时器  .take(8) // 执行次数达到8 停止下来  .subscribe(new Consumer<Long>() { // 下游  @Override  public void accept(Long aLong) throws Exception {  Log.d(TAG, "accept: " + aLong);  }  });  }  /\*\*  \* distinct过滤重复事件  \* @param view  \*/  public void r03(View view) {  // 上游  Observable.create(new ObservableOnSubscribe<Integer>() {  @Override  public void subscribe(ObservableEmitter<Integer> e) throws Exception {  e.onNext(1);  e.onNext(1);  e.onNext(2);  e.onNext(3);  e.onNext(4);  e.onNext(4);  e.onComplete();  }  })  .distinct() // 过滤重复 发射的事件  .subscribe(new Consumer<Integer>() { // 下游 观察者  @Override  public void accept(Integer integer) throws Exception {  Log.d(TAG, "accept: " + integer); // 事件不重复  }  });  }  /\*\*  \* elementAl 指定过滤的内容  \* @param view  \*/  public void r04(View view) {  // 上游  Observable.create(new ObservableOnSubscribe<String>() {  @Override  public void subscribe(ObservableEmitter<String> e) throws Exception {  e.onNext("九阴真经");  e.onNext("九阳真经");  e.onNext("易筋经");  e.onNext("神照经");  e.onComplete();  }  })  // 过滤操作符  .elementAt(2, "默认经") // 指定下标输出 事件  // 订阅  .subscribe(new Consumer<String>() { // 下游  @Override  public void accept(String s) throws Exception {  Log.d(TAG, "accept: " + s);  }  });  }  } |

////////////////////////////////////////////////////////////////////////////////////////////////////

# 06.RxJava条件型操作符。

上游 -------> 条件操作(往右边流向的时候，条件判断) ----------> 下游

操作符：RxJava说的很神奇，API的调用， RxJava改变开发者的思维

RxJava == Java编程

语法 == 操作符API

所有的操作符都学会了，才能真正的证明把RxJava的使用学会了 == Java所有的语法学会，Java入门

RxJava作为: Android之神 2010 开源过Android开源的框架库， RxJava巅峰之作

1. All: 如同 if 那样的功能 ：全部为true，才是true，只要有一个为false，就是false.

//里面对事件进行判断，return boolean.

1. contains 是否包含

//判断事件流中是否包含某一个元素，最后只发送一个事件。

（3）any 全部为 false，才是false， 只要有一个为true，就是true

注：如果使用了条件操作符，下一层，接收的类型 就是条件类型(Boolean)

|  |
| --- |
| /\*\*  \* all，如同 if 那样的功能 ：全部为true，才是true，只要有一个为false，就是false  \* @param view  \*/  public void r01(View view) {  String v1 = "1";  String v2 = "2";  String v3 = "3";  String v4 = "cc";  // 需求：只要有一个为 cc的，就是false  // 平常的写法  // if (v1.equals("cc") || v2.equals("cc") || v3.equals("cc") || v4.equals("cc")) {  if (v1 == null) {  Log.d(TAG, "r01: " + false);  } else {  Log.d(TAG, "r01: " + true);  }  // RxJava的写法  // 上游  Observable.just(v1, v2, v3, v4) // RxJava 2.X 之后 不能传递null，否则会报错  .all(new Predicate<String>() {  @Override  public boolean test(String s) throws Exception {  return !s.equals("cc"); // 如果s不等于cc，就是true  }  })  .subscribe(new Consumer<Boolean>() { // 下游  @Override  public void accept(Boolean s) throws Exception {  Log.d(TAG, "accept: " + s);  }  });  }  /\*\*  \* contains 是否包含  \* @param view  \*/  public void r02(View view) {  Observable.just("JavaSE", "JavaEE", "JavaME", "Android", "iOS", "Rect.js", "NDK")  .contains("C") // 是否包含了 Android，条件是否满足  .subscribe(new Consumer<Boolean>() {  @Override  public void accept(Boolean s) throws Exception {  Log.d(TAG, "accept: " + s);  }  });  }  /\*\*  \* Any 和 All相反的  \* All全部为true，才是true，只要有一个为false，就是false  \* any 全部为 false，才是false， 只要有一个为true，就是true  \* @param view  \*/  public void r03(View view) {  Observable.just("JavaSE", "JavaEE", "JavaME", "Android", "iOS", "Rect.js", "NDK")  .any(new Predicate<String>() {  @Override  public boolean test(String s) throws Exception {  return s.equals("Android");  }  }) // 是否包含了 Android，条件是否满足  .subscribe(new Consumer<Boolean>() {  @Override  public void accept(Boolean s) throws Exception {  Log.d(TAG, "accept: " + s);  }  });  } |

////////////////////////////////////////////////////////////////////////////////////////////////////

# 07.RxJava合并型操作符。

两个或者多个 被观察者 合并。（操作对象都是被观察者）

1. startWith，concatWith ：先创建被观察者，然后再组合其他的被观察者，然后再订阅

//startWith 合并操作符, 被观察者1.startWith(被观察者2) 先执行 被观察者2 里面发射的事件，然后再执行 被观察者1 发射的事件。

//concatWith 和 startWith 的区别，是相反的

concatWith 合并操作符, 被观察者1.concatWith(被观察者2) 先执行 被观察者1 里面发射的事件，然后再执行 被观察者2 发射的事件

2.concat/merge/zip：直接合并多个被观察者，然后订阅

细节：

a:startWait 先执行 startWait括号里面的被观察者

b:concatWait 后执行 concatWait括号里面的被观察者

c:concat 是按照顺序依次执行 最多四个被观察者进行合并。

//concat 合并操作符 的特性：最多能够合并四个，按照我们存入的顺序执行.

d:merge 并列执行的（并发），（演示并列的执行，所以学了intervalRange） 最多四个被观察者进行合并

e:zip 需要对应关系 需要对应，如果不对应，会被忽略的， 最多9个被观察者 进行合并

//需求：考试 课程 == 分数

//一个是发送课程的被观察者，一个是发送发送的被观察者，他们的个数相等，要一一对应。

一个请求依赖另一个请求，使用flatMap

同时发起多个请求，并结合他们的结果。我有三个网络请求，它们互不依赖，我想同时执行以提高响应时间。等到三个请求都结束后才发射值，使用zip。

////////////////////////////////////////////////////////////////////////////////////////////////////

# 08.RxJava异常处理操作符。

1.RxJava中是不标准的用法: throw new IllegalAccessError("我要报错了");

2.应该用： RxJava标准的e.onError(XXX);

e.onError(new IllegalAccessError("我要报错了")); // 发射异常事件

3.onErrorReturn最先拦截到e.onError并且可以给下游返回一个 标识400,

throw new XXX 拦截不到，整个程序奔溃。

onErrorReturn异常操作符：1.能够接收e.onError， 2.如果接收到异常，会中断上游后续发射的所有事件

1. onErrorResumeNext最先拦截到e.onError并且可以给下游返回一个 被观察者（还可以再次发送）, throw new XXX 拦截不到，整个程序奔溃
2. onExceptionResumeNext 能在发生异常的时候，扭转乾坤，能够处理 throw new XXX，可以真正的让App不奔溃

//onExceptionResumeNext 操作符，能在发生异常的时候，扭转乾坤，（这种错误一定是可以接受的，才这样使用）。

慎用：自己去考虑，是否该使用。

7.retry重试操作符 ： return false; 代表不去重试 return true; 不停的重试， 演示二 重试次数， 演示三 打印重试了多少次，计数

////////////////////////////////////////////////////////////////////////////////////////////////////

# 09.RxJava线程切换。

1.异步线程区域

Schedulers.io() ：代表io流操作，网络操作，文件流，耗时操作

Schedulers.newThread() ： 比较常规的，普普通通

Schedulers.computation() ： 代表CPU 大量计算 所需要的线程

2.AndroidSchedulers.mainThread() ： 专门为Android main线程量身定做的

3.给上游分配多次，只会在第一次切换，后面的不切换了（忽略）

给下游分配多次，每次都会去切换

5.如果不配置异步线程，上游发一次，下游接收一次，上游发一次，下游接收一次，上游发一次，下游接收一次

6.配置好异步线程，就是异步的表现

7.传统下载图片的写法，容易四分五裂

8.RxJava下载图片，基于事件流编程，一条链子，起点和终点

|  |
| --- |
| /\*\*  \* todo 使用RxJava去下载图片  \* @param view  \*/  public void r04(View view) {  // 起点  // 上游 被观察者 Observable  Observable.just(PATH) // 内部发射  // String Path 变换 Bitmap  .map(new Function<String, Bitmap>() {  @Override  public Bitmap apply(String s) throws Exception {  try {  Thread.sleep(2000);  URL url = new URL(PATH);  URLConnection urlConnection = url.openConnection();  HttpURLConnection httpURLConnection = (HttpURLConnection) urlConnection;  httpURLConnection.setConnectTimeout(5000);  int responseCode = httpURLConnection.getResponseCode();  if (HttpURLConnection.HTTP\_OK == responseCode) {  Bitmap bitmap = BitmapFactory.decodeStream(httpURLConnection.getInputStream());  return bitmap;  }  } catch (Exception e) {  e.printStackTrace();  }  return null;  }  })  .map(new Function<Bitmap, Bitmap>() {  @Override  public Bitmap apply(Bitmap bitmap) throws Exception {  // 给图片加水印  Paint paint = new Paint();  paint.setColor(Color.RED);  paint.setTextSize(30);  Bitmap bitmapSuccess = drawTextToBitmap(bitmap, "同学们大家好", paint, 60, 60);  return bitmapSuccess;  }  })  // 比如：增加一个 日志纪录功能，只需要添加要给 变换操作符  .map(new Function<Bitmap, Bitmap>() {  @Override  public Bitmap apply(Bitmap bitmap) throws Exception {  Log.d(TAG, "apply: 下载的Bitmap 是这个样子的" + bitmap);  return bitmap;  }  })  .subscribeOn(Schedulers.io()) // todo 给上游分配 异步线程  .observeOn(AndroidSchedulers.mainThread()) // todo 给下游分配 主线程  .subscribe(new Observer<Bitmap>() { // 下游  @Override  public void onSubscribe(Disposable d) {  progressDialog = new ProgressDialog(MainActivity8.this);  progressDialog.setMessage("RxJava下载图片中..");  progressDialog.show();  }  @Override  public void onNext(Bitmap bitmap) {  if (imageView != null)  imageView.setImageBitmap(bitmap);  }  @Override  public void onError(Throwable e) { // 发生异常  // if (imageView != null)  // imageView.setImageResource(R.mipmap.ic\_launcher); // 下载错误的图片  }  @Override  public void onComplete() { // 终点  if (progressDialog != null)  progressDialog.dismiss();  }  });  } |

////////////////////////////////////////////////////////////////////////////////////////////////////

# 10.RxJava背压模式。

背压模式的由来：

RxJava1.X的时候，还没有背压模式， 我们的上游不停的发射，我们的下游处理不过来，就会照成内存泄漏

RxJava2.X之后，增加背压模式，Observable Flowable（解决背压）

Observable --- > Flowable（解决背压）

什么时候用Observable<--->Observer， 什么时候使用Flowable<--->Subscriber ？

答：发射的事件，大量的事件(1000个)，并且考虑到下游处理不过来，就需要使用Flowable

1.// ERROR 放入缓存池，如果池子满了 水缸 max 128

BackpressureStrategy.ERROR // todo 上游不停的发射大量事件，下游阻塞了 处理不过来，放入缓存池，如果池子满了，就会抛出异常

2.BackpressureStrategy.BUFFER // todo 上游不停的发射大量事件，下游阻塞了 处理不过来，放入缓存池，”等待“下游来接收事件处理

3.同步的，没有执行Subscription s.request(), 当上游发射1，下游无法处理（没有执行s.request()），会抛出异常

4.异步的，上游不停的发射，可以在r02方法中，s.request(10) 可以取出来给 下游接收事件处理的

5.一旦下游处理了一次上游的事件，缓存池 - 1

6.Observable 它的升级版 Flowable：功能强大，还有背压模式。

////////////////////////////////////////////////////////////////////////////////////////////////////

# 11.Flowable --- Observable

如果我们会使用Observable， 那么一定会使用Flowable

Flowable的设计，是按照Observable 依葫芦画瓢来设计Flowable，所以使用才一摸一样，只不过类名不同而已， Flowable还增加了背压模式

1.Observable<--->Observer， Flowable<--->Subscriber 对应关系 ？

2.Observable的设计和 Flowable一致的，在Observable的基础上 增加了一套Flowable的代码，而且增加的时候 依葫芦画瓢的，Flowable增加了背压模式

3.Observable--Observer下游 -- onSubscribe(Disposable d)切断下游（水管）

4.Flowable---Subscriber下游 -- onSubscribe(Subscription s) 取出（s.request(5)）事件 给下游接收使用

////////////////////////////////////////////////////////////////////////////////////////////////////

# 12.RxJava配合Retrofit。

RxJava + Retrofit （请求网络OkHttp ---- Retorfit --- Observable）

1.OkHttp 请求网络 （Retorfit）

2.Retorfit 返回一个结果 （Retorfit） --- Observable

3.最终的结果 是RxJava中的 被观察者 上游 Observable

4.一行代码写完需求流程： 从上往下

1.请求服务器，执行注册操作（耗时）切换异步线程

2.更新注册后的所有 注册相关UI - main 切换主线程

3.请求服务器，执行登录操作（耗时）切换异步线程

4.更新登录后的所有 登录相关UI - main 切换主线程

5.看RxJava另外一种的执行流程

初始点 开始点 订阅

1.onSubscribe

2.registerAction(new RegisterRequest())

3..doOnNext 更新注册后的 所有UI

4.flatMap执行登录的耗时操作

5.订阅的观察者 下游 onNext 方法，更新所有登录后的UI

6.progressDialog.dismiss()

////////////////////////////////////////////////////////////////////////////////////////////////////

# 13.泛型 Java

由于RxJava大量的使用到了泛型，所有学习

如果我们不指定泛型类型，默认就是Object，Object的扩展集

1.? super F 下限：F or F 所有父类 都可以，所以属于把下面的类型限限制了，下面的类型不能低于F，不能是F的子类，否则编译不通过。

2.? extends F 上限：F or F 所有子类 都可以，所有属于把上面的类型给限制了，上面的类型不能高于F，不能是F的父类，否则编译不通过。

一句话记住（? super F: F 或者 F的所有父类 都可以， ? extends F: F 或者 F的所有子类都可以）

3.可写模式<？super F> 可写，不完全可读

4.可读模式<? extends F> 可读，不可写

# 14.RxJava无死角回顾。

1.RxJava的概念 和 标准观察者设计模式（多个观察者，注册多次） -- RxJava中的观察者设计模式(不标准 变异)（一个观察者，订阅一次）。

2.RxJava上游和下游，

3.RxJava创建型操作符。 创建（被观察者） create just fromArray ...

4.RxJava变换型操作符。 上一层的事件类型 变换 下一层需要的事件类型

5.RxJava过滤型操作符。 上一层的事件，过滤筛选 给下一层

6.RxJava条件型操作符。 把上一层的事件 ，是否满足条件 给下一层 Boolean

7.RxJava合并型操作符。 把多个被观察者 合并成一个被观察者 --- 》 发射事件

8.RxJava异常型操作符。 上一层发射事件 异常的处理 异常操作符 -- try ----》 下一层。

9.RxJava线程的切换。

默认情况下main线程， 异步线程io， 主线程AndroidSchedulers.mainThread()

给上游分配多次，只会在第一次切换，后面的不切换了 （只在第一次有效）

给下游分配多次，每次都会去切换 （每次都会有效）

10.背压 上游大量的发射事件，下游处理（阻塞）不过来，就会考虑到我们背压模式(ERROR, BUFFER)

11.RxJava Flowable --- Observable

12.RxJava配合Retrofit+OkHttp

13.RxJava泛型 上限和下限，读写模式

14.RxJava手写create操作符, 就是一个接口回调。

15.RxJava手写create操作符的读写模式。(在方法参数中：一定是上限和下限, 真正的使用到了泛型操作：读写模式)

16.RxJava手写jsut操作符，就是一个接口回调，在just方法里面调用的。

17.RxJava手写map操作符。变换操作符

18.RxJava手写map+create+observer操作符流程讲解，从订阅到上游，然后再从上游到下游