一 RxJava 的原理

RxJava 有四个基本概念：`Observable` (可观察者，即被观察者)、 `Observer` (观察者)、 `subscribe` (订阅)、事件。`Observable` 和 `Observer` 通过 `subscribe()` 方法实现订阅关系，从而 `Observable` 可以在需要的时候发出事件来通知 `Observer`。

1.创建一个简单的Observable

```
public class Observable<T> {
final OnSubscribe<T> onSubscribe;

private Observable(OnSubscribe<T> onSubscribe) {
this.onSubscribe = onSubscribe;
}

public static <T> Observable<T> create(OnSubscribe<T> onSubscribe) {
return new Observable<T>(onSubscribe);
}

public Subscriber subscribe(Subscriber<? super T> subscriber) {
// 是 Subscriber 增加的方法 事件还未发送之前执行，用于数据清零或重置
// 如果对线程有要求，要求在主线程上就只能用doOnSubscribe
subscriber.onStart();
// 按照设置的执行计划开始执行
onSubscribe.call(subscriber);
return subscriber;
}

// 相当于一个执行计划
public interface OnSubscribe<T> {
void call(Subscriber<? super T> subscriber);
}
}
```

2.创建观察者Observer ， 创建它的实现抽象**Subscriber**

```
1 public interface Observer<T> {
2  void onCompleted();
3  void onError(Throwable t);
4  void onNext(T var1);
5 }
6
7 public abstract class Subscriber<T> implements Observer<T> {
```

```
 public void onStart() {
 //可以用于做一些准备工作，例如数据的清零或重置
  //但如果必须在主线程就不能用了，只能适用 doOnSubscribe
  System.out.println("准备工作");
  }
}
```

测试下

```
// 创建Observable 将Subscriber 订阅上去 执行内部的执行计划
Observable.create(new Observable.OnSubscribe<Integer>(){
 @Override
 public void call(Subscriber<? super Integer> subscriber) {
 for (int i = 0; i <10 ; i++) {
 subscriber.onNext(i);
 }
 }
}).subscribe(new Subscriber<Integer>() {
  ... ...
 @Override
 public void onNext(Integer var1) {
 System.out.println(var1);
 }
});
```

二 map 操作符

现在想把Integer转换成 字符怎么在中间加步操作呢?

```
1. 创建一个接口
/**
 * 转换类型
 * @param <T> 转入类型
 * @param <R> 转出类型
 */
public interface Transformer<T, R> {
 R call(T from);
}
 //map操作符的作用是将T类型的Event转化成R类型，转化策略抽象成Transformer<T, R>
 public <R> Observable<R> map(Transformer<? super T, ? extends R> transfo
rmer) {
  return create(new OnSubscribe<R>() { // 生成一个桥接的Observable和 OnSubs
cribe
 @Override
```

```
 public void call(Subscriber<? super R> subscriber) {
 // Observable.this 这个是闭包中上层的Observable对象
 Observable.this.subscribe(new Subscriber<T>() { // 订阅上层的Observable
 // .... ...................省略complete 和error
 @Override
 public void onNext(T var1) {
 // 将上层的onSubscribe发送过来的Event，通过转换和处理，转发给目标的subscriber
 subscriber.onNext(transformer.call(var1));
 }
 });
 }
 });
}
```

测试方法

```
// 创建第一个Observable
Observable.create(new OnSubscribe<Integer>() {
 @Override
 public void call(Subscriber<? super Integer> subscriber) {
 for (int i = 0; i < 10; i++) {
 subscriber.onNext(i);
 }
 }
}).map(new Transformer<Integer, String>() {
 @Override
 public String call(Integer from) {
 return "maping " + from;
 } //创建的第二个Observable2
}).subscribe(new Subscriber<String>() {
 @Override
 public void onNext(String var1) {
 System.out.println(var1);
 }
});
//此时被订阅的是Observable2，内部调用onSubscribe.call(s1)，此时的执行计划是Observable2通过map创建的
// onSubscribe.call 里面 根据闭包，创建时Observable2 传入的是Observable1，所以此时Observable1 开始
// 被订阅， 传入新的Subscriber(内部的onNext 调用的是Observable2的s1.onNext(transForm.call(初始值)))) )
```

可以吧map 中的逻辑抽取出来

```
public <R> Observable<R> map2(Transformer<? super T, ? extends R> transformer) {
 return create(new MapOnSubscribe<T, R>(this, transformer));
}

public class MapOnSubscribe<T, R> implements Observable.OnSubscribe<R> {
 final Observable<T> source;
 final Observable.Transformer<? super T, ? extends R> transformer;
 public MapOnSubscribe(Observable<T> source, Observable.Transformer<? super T, ? extends R> transformer) {
 this.source = source;
  this.transformer = transformer;
  }
  @Override
  public void call(Subscriber<? super R> subscriber) {
  source.subscribe(new MapSubscriber<R, T>(subscriber, transformer));
  }
 }

 public class MapSubscriber<T, R> extends Subscriber<R> {
  final Subscriber<? super T> actual;
  final Observable.Transformer<? super R, ? extends T> transformer;
  public MapSubscriber(Subscriber<? super T> actual, Observable.Transformer<? super R, ? extends T> transformer) {
  this.actual = actual;
  this.transformer = transformer;
  }
 //...........省略
  @Override
  public void onNext(R var1) {
  actual.onNext(transformer.call(var1));
  }
 }

 Observable.create(new Observable.OnSubscribe<Integer>() {
  @Override
  public void call(Subscriber<? super Integer> subscriber) {
  for (int i = 0; i < 10; i++) {
  subscriber.onNext(i);
  }
```

```
 subscriber.onCompleted();
 }
}).map2(new Observable.Transformer<Integer, String>() {
 @Override
 public String call(Integer var1) {
 return var1+"demo";
 }
}).subscribe(new Subscriber<Object>() {
 //.........
 @Override
 public void onNext(Object var1) {
 System.out.println(var1);
 }
}

```

三 flatMap()

flatMap 返回的是一个Observable对象，而 map 返回的是一个普通转换后的对象;

flatMap 返回的Observable对象并不是直接发送到Subscriber的回调中，而是重新创建一个Observable对象，并激活这个Observable对象，使之开始发送事件；而 map 变换后返回的对象直接发到Subscriber回调中；

flatMap 变换后产生的每一个Observable对象发送的事件，最后都汇入同一个Observable，进而发送给Subscriber回调；

map返回类型 与 flatMap 返回的Observable事件类型，可以与原来的事件类型一样；

可以对一个Observable多次使用 map 和 flatMap；

鉴于 flatMap 自身强大的功能，这常常被用于 嵌套的异步操作，例如嵌套网络请求。传统的嵌套请求，一般都是在前一个请求的 onSuccess() 回调里面发起新的请求，这样一旦嵌套多个的话，缩进就是大问题了，而且严重的影响代码的可读性。而RxJava嵌套网络请求仍然通过链式结构，保持代码逻辑的清晰

四 lift()

```
// 更高级的map 将转换逻辑 提取出来
public <R> Observable<R> lift(Operator<? extends R,? super T> operator) {
 return Observable.create(new OnSubscribe<R>() {
 @Override
 public void call(Subscriber<? super R> subscriber) {
 Subscriber newSubscriber = operator.call(subscriber);
 newSubscriber.onStart();
```

```
 // 最原始的Onsubscribe 闭包原理
 onSubscribe.call(newSubscriber);
  }
  });
}
```

```
public interface Operator<R, T> {
 public Subscriber<? super T> call(Subscriber<? super R> subscriber);
}
```

```
.lift(new Operator<String, Integer>() {
 @Override
 public Subscriber<? super Integer> call(final Subscriber<? super String>
subscriber) {
 return new Subscriber<Integer>() {
 @Override
 public void onCompleted() {
 subscriber.onCompleted();
 }

  @Override
  public void onError(Throwable t) {
  subscriber.onError(t);
  }

  @Override
  public void onNext(Integer var1) {
  subscriber.onNext(var1 + "toString");
  }
  };
  }
});
```

RxJava 都不建议开发者自定义 `Operator` 来直接使用 `lift()`，而是建议尽量使用已有的 `lift()` 包装方法（如 `map()` `flatMap()` 等）进行组合来实现需求，因为直接使用 lift() 非常容易发生一些难以发现的错误。

五 Compose

传入一个 可回调的Transformer

六 线程控制

```
public class Scheduler {
 // 创建线程池
 final Executor executor;
 public Scheduler(Executor executor) {
 this.executor = executor;
 }

 public Worker createWorker() {
 return new Worker(executor);
  }

 //具体任务的执行者
  public static class Worker {
  final Executor executor;
```

```
15   public Worker(Executor executor) {
16   this.executor = executor;
17   }
18
19   public void schedule(Runnable runnable) {
20   executor.execute(runnable);
21   }
22   }
23 }
```

创建工厂方法

```
1 public class Schedulers {
2   private static final Scheduler ioScheduler = new Scheduler(Executors.new
CachedThreadPool());
3   public static Scheduler io() {
4   return ioScheduler;
5   }
6 }
```

实现 subscribeOn

```
1 public Observable<T> subscribeOn(Scheduler scheduler) {
2   return Observable.create(new OnSubscribe<T>() {
3   @Override
4   public void call(Subscriber<? super T> subscriber) {
5   subscriber.onStart();
6   // 将事件的生产切换到新的线程。
7   scheduler.createWorker().schedule(new Runnable() {
8   @Override
9   public void run() {
10   Observable.this.onSubscribe.call(subscriber);
11   }
12   });
13   }
14   });
15 }
16 //subscribeOn是只作用于上层OnSubscribe的，可以让OnSubscribe的call方法在新线
程中执行。
17 Observable.create(new Observable.OnSubscribe<Integer>() {...})
18   // subscribeOn 是作用在上层的线程
19 .subscribeOn(Schedulers.io()).lift(new Operator<Integer, Integer>()
{....})
20 .subscribe(new Subscriber<Integer>() {})
```

```
21
22 mainlift
23 注册成功,开始执行
24 mainsubscribeOn
25 注册成功,开始执行
26 pool-1-thread-1OnSubscribe
27
28 //所以最后的subscriber 都是在 pool-1-thread-1OnSubscribe 这个线程上执行回调的
```

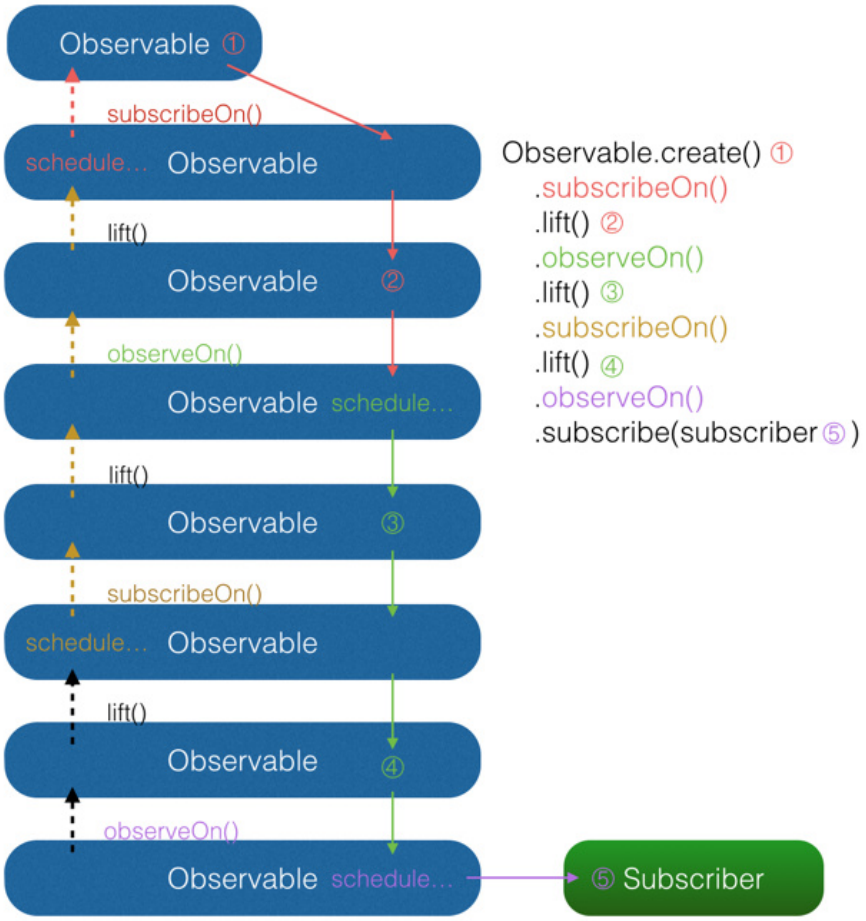
实现 observerOn

```
1  public Observable<T> observeOn(Scheduler scheduler) {
2   return Observable.create(new OnSubscribe<T>() {
3   @Override
4   public void call(Subscriber<? super T> subscriber) {
5   subscriber.onStart();
6   Scheduler.Worker worker = scheduler.createWorker();
7   // 通过创建一个新的跑在新线程上的Subscriber
8   Observable.this.onSubscribe.call(new Subscriber<T>() {
9   ...........
10   @Override
11   public void onNext(T var1) {
12   worker.schedule(new Runnable() {
13   @Override
14   public void run() {
15   subscriber.onNext(var1);
16   }
17   });
18   }
19   });
20   }
21   });
22  }
23
```

`subscribeOn()`的线程切换发生在 `OnSubscribe` 中，即在它通知上一级 `OnSubscribe` 时，这时事件还没有开始发送，因此 `subscribeOn()` 的线程控制可以从事件发出的开端就造成影响；即所有的 subscriber 都会在该线程上

而 `observeOn()` 的线程切换则发生在它内建的 `Subscriber` 中，即发生在它即将给下一级 `Subscriber` 发送事件时，因此 `observeOn()` 控制的是它后面的线程

①和②两处受第一个 `subscribeOn()` 影响，运行在红色线程；



③和④处受第一个 `observeOn()` 的影响，运行在绿色线程；

⑤处受第二个 `onserveOn()` 影响，运行在紫色线程；

而第二个 `subscribeOn()` ，由于在通知过程中线程就被第一个 `subscribeOn()` 截断，因此对整个流程并没有任何影响。这里也就回答了前面的问题：当使用了多个 `subscribeOn()` 的时候，只有第一个 `subscribeOn()` 起作用

Schedulers.io(): I/O 操作（读写文件、数据库、网络请求等），与newThread()差不多，区别在于io() 的内部实现是是用一个无数量上限的线程池，可以重用空闲的线程，因此多数情况下 io() 效率比 newThread() 更高。值得注意的是，在 io() 下，不要进行大量的计算，以免产生不必要的线程；

Schedulers.newThread(): 开启新线程操作；

//Schedulers.immediate(): 默认指定的线程，也就是当前线程； Rx 2 已经废弃

Schedulers.computation():计算所使用的调度器。这个计算指的是 CPU 密集型计算，即不会被 I/O等操作限制性能的操作，例如图形的计算。这个 Scheduler 使用的固定的线程池，

大小为 CPU 核数。值得注意的是，不要把 I/O 操作放在 computation() 中，否则 I/O 操作的等待时间会浪费 CPU；

AndroidSchedulers.mainThread(): RxJava 扩展的 Android 主线程；

六 **doOnSubscribe()**

之前说过 onStart 是 subScriber比Observer多的一个方法 可以用来初始化数据等

但是由于它是在subscribe()发送 就被调用了，不能指定线程，而是只能执行

在 subscribe() 被调用时的线程

`doOnSubscribe()` 它和 `Subscriber.onStart()` 同样是在 `subscribe()` 调用后而且在事件发送前执行，但区别在于它可以指定线程。默认情况下， `doOnSubscribe()` 执行在 `subscribe()` 发生的线程；而如果在 `doOnSubscribe()` 之后有 `subscribeOn()` 的话，它将执行在离它最近的 `subscribeOn()` 所指定的线程。

`doOnSubscribe()`的后面跟一个 `subscribeOn()` ，就能指定准备工作的线程了

```
1 //doOnSubscribe() 内部就是调用lift 返回一个新的Observable
2 // 是在调用onSubscribe 的call 方法里 调用operator ， 所以 由于subscribeOn() 的线程切换发生在 OnSubscribe 中
3 //所以它将执行在离它最近的 subscribeOn() 所指定的线程。
4 public final Observable<T> doOnSubscribe(final Action0 subscribe) {
5  return lift(new OperatorDoOnSubscribe<T>(subscribe));
6 }
```

RxJava 的使用

1. from　接收一个集合作为输入，然后每次输出一个元素给subscriber。
2. just　接收一个可变参数作为输入，最终也是生成数组，调用from()，然后每次输出一个元素给subscriber。
3. take 最多保留的事件数。
4. doOnNext 在处理下一个事件之前要做的事。
5. *// 间隔400ms以内的事件将被丢弃* .debounce(400, TimeUnit.MILLISECONDS)
6. merge 用于合并两个Observable为一个Observable。较为简单。ZIP是将两个Observable 或多个根据传入函数处理合并成一个Observable

7. concat 顺序执行多个Observable，个数为1 ~ 9。例子稍后与first操作符一起~~

8. compose 与 flatMap 类似，都是进行变换，返回Observable对象，激活并发送事件。

compose 是唯一一个能够从数据流中得到原始Observable的操作符，所以，那些需要对整个数据流产生作用的操作（比如，subscribeOn()和observeOn()）需要使用 compose 来实现。相较而言，如果在flatMap()中使用subscribeOn()或者observeOn()，那么它仅仅对在 flatMap 中创建的Observable起作用，而不会对剩下的流产生影响。这样就可以简化subscribeOn()以及observeOn()的调用次数了。

compose 是对 Observable 整体的变换，换句话说， flatMap 转换Observable里的每一个事件，而 compose 转换的是整个Observable数据流。flatMap 每发送一个事件都创建一个 Observable，所以效率较低。而compose 操作符只在主干数据流上执行操作。

建议使用 compose 代替 flatMap

9.timer 可以做定时操作，换句话讲，就是延迟执行。事件间隔由timer控制

10.interval 定时的周期性操作，与timer的区别就在于它可以重复操作。事件间隔由interval控制。

11.throttleFirst 与debounce类似，也是时间间隔太短，就丢弃事件。可以用于防抖操作，比如防止双击。

```
.throttleFirst(1, TimeUnit.SECONDS)
```

12. **SingleObserver**，相当于是精简版。订阅者回调的不是OnNext/OnError/onCompleted，而是回调OnSuccess/OnError。

Subject

Subject这个类，既是Observable又是Observer，啥意思呢？就是它自身既是事件的生产者，又是事件的消费者，相当于自身是一条管道，从一端进，又从另一端出。举个栗子：PublishSubject

`OnSubscribe` 对象作为参数。`OnSubscribe` 会被存储在返回的 `Observable` 对象中，**它的作用相当于一个计划表，当 `Observable` 被订阅的时候，**

`OnSubscribe` 的 `call()` 方法会自动被调用，事件序列就会依照设定依次触发（对于上面的代码，就是观察者`Subscriber` 将会被调用三次 `onNext()` 和一次 `onCompleted()`）。这样，由被观察者调用了观察者的回调方法，就实现了由被观察者向观察者的事件传递，即观察者模式。

最后再说下节流的问题

Flow Control（流控）有哪些思路呢？大概是有四种:

1. 背压（Backpressure）；
2. 节流（Throttling）；
3. 打包处理；
4. 调用栈阻塞（Callstack blocking）

Throttling

sample 操作函数可以指定生产者发射数据的最大速度，多余的数据被丢弃了，在当前时间段内选最后那个值。所以它也叫throttleLast。与之对应的是throttleFirst

打包处理 Collect

如果你不想丢弃数据，则当消费者忙的时候可以使用 buffer 和 window 操作函数来收集数据

buffer和window

最后再来说下背压问题(真形象)

Backpressure

pull 模型， 消费者等待生产者， 而Rx为push模型，只要生产好就发射出去就会出现背压问题

1.如果没有使用多线程，同步调用时，调用onNext 会阻塞到消费者消费完才执行第二个onNext

2.使用observeOn， 因为是切换了消费者的线程，因此内部实现用队列存储事件。在Android 中默认的 buffersize 大小是16，因此当消费比生产慢时， 队列中的数目积累到超过16个，就会抛出MissingBackpressureException。

Observable不再支持Backpressure，而是改用Flowable来专门支持Backpressure

- BackpressureStrategy.BUFFER 不丢弃数据的处理方式
- BackpressureStrategy.DROP
- BackpressureStrategy.LATEST

DROP 和LATEST都会丢弃数据。这两种策略相当于一种令牌机制（或者配额机制），下游通过request请求产生令牌（配额）给上游，上游接到多少令牌，就给下游发送多少数据。当令牌数消耗到0的时候，上游开始丢弃数据，而LATEST 是发送上一个接到令牌缓存的最新值。

新版本 Flowable 代替了Observable 强制设置背压策略

```
Flowable.create(new FlowableOnSubscribe<Integer>() {
 @Override
 public void subscribe(FlowableEmitter<Integer> emitter) throws Exception
{
 emitter.onNext(1);
 }
}, BackpressureStrategy.BUFFER);
```

而Consumer即消费者，用于接收单个值，BiConsumer则是接收两个值，Function用于变换对象，Predicate用于判断。这些接口命名大多参照了Java8

在RxJava2.0中，有五种观察者模式：

1. Observable/Observer
2. Flowable/Subscriber
3. Single/SingleObserver
4. Completable/CompletableObserver
5. Maybe/MaybeObserver

http://gank.io/post/560e15be2dca930e00da1083

https://blog.csdn.net/TellH/article/details/71534704

https://tech.youzan.com/xiang-ying-shi-jia-gou-yu-rxjavazai-you-zan-ling-shou-de-shi-jian/

https://blog.csdn.net/jdsjlzx/article/details/51493772