EBook Gratis

APRENDIZAJE rx-java

Free unaffiliated eBook created from
Stack Overflow contributors.

#rx-java

Tabla de contenido

Acerca de

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: rx-java

It is an unofficial and free rx-java ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official rx-java.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Capítulo 1: Empezando con rx-java

Observaciones

Esta sección proporciona una descripción básica y una introducción superficial a rx-java.

RxJava es una implementación Java VM de Extensiones reactivas : una biblioteca para componer programas asíncronos y basados en eventos mediante el uso de secuencias observables.

Aprenda más sobre RxJava en la página principal de Wiki .

Versiones

Versión	Estado	Última versión estable	Fecha de lanzamiento
1.x	Estable	1.3.0	2017-05-05
2.x	Estable	2.1.1	2017-06-21

Examples

Instalación o configuración

configuración de rx-java

1. Gradle

```
compile 'io.reactivex:rxjava2:rxjava:2.1.1'
```

2. Maven

```
<dependency>
    <groupId>io.reactivex.rxjava2</groupId>
    <artifactId>rxjava</artifactId>
    <version>2.1.1</version>
</dependency>
```

3. Hiedra

```
<dependency org="io.reactivex.rxjava2" name="rxjava" rev="2.1.1" />
```

4. Instantáneas de JFrog

```
repositories {
maven { url 'https://oss.jfrog.org/libs-snapshot' }
```

```
}

dependencies {
    compile 'io.reactivex:rxjava:2.0.0-SNAPSHOT'
}
```

5. Si necesita descargar los archivos jar en lugar de usar un sistema de compilación, cree un archivo `pom` Maven como este con la versión deseada:

```
<?xml version="1.0"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <groupId>com.netflix.rxjava.download</groupId>
    <artifactId>rxjava-download</artifactId>
    <version>1.0-SNAPSHOT</version>
    <name>Simple POM to download rxjava and dependencies</name>
    <url>http://github.com/ReactiveX/RxJava</url>
    <dependencies>
        <dependency>
            <groupId>io.reactivex</groupId>
            <artifactId>rxjava</artifactId>
            <version>2.0.0</version>
            <scope/>
        </dependency>
    </dependencies>
</project>
```

Luego ejecuta:

```
$ mvn -f download-rxjava-pom.xml dependency:copy-dependencies
```

Ese comando descarga `rxjava-*.jar` y sus dependencias en `./target/dependency/`.

Necesitas Java 6 o posterior.

¡Hola Mundo!

A continuación se imprime el mensaje `Hello, World!` consolar

```
public void hello() {
  Observable.just("Hello, World!") // create new observable
    .subscribe(new Action1<String>() { // subscribe and perform action

       @Override
       public void call(String st) {
         System.out.println(st);
       }

    });
}
```

O usando la notación lambda de Java 8

```
public void hello() {
      Observable.just("Hello, World!") // create new observable
        .subscribe(onNext -> { // subscribe and perform action
             System.out.println(onNext);
        });
}
```

Una introducción a RxJava

Los conceptos centrales de RxJava son sus `Observables` y `Subscribers` . Un `Observable` emite objetos, mientras que un `Subscriber` consume.

Observable

`Observable` es una clase que implementa el patrón de diseño reactivo. Estos Observables proporcionan métodos que permiten a los consumidores suscribirse a cambios de eventos. Los cambios de evento son activados por lo observable. No hay restricción en el número de suscriptores que un `Observable` puede tener, o el número de objetos que un `Observable` puede emitir.

Tomar como ejemplo:

```
Observable<Integer> integerObservable = Observable.just(1, 2, 3); // Integer observable
Observable<String> stringObservable = Observable.just("Hello, ", "World", "!"); // String
observable
```

Aquí, un objeto observable llamado `integerObservable` y `stringObservable` se crean a partir del método de fábrica que `just` proporcionar la biblioteca Rx. Observe que `Observable` es genérico y, por lo tanto, puede emitir cualquier objeto.

Abonado

Un `Subscriber` es el consumidor. Un `Subscriber` puede suscribirse a ***un solo*** observable. El `Observable` llama a los `onNext()` , `onCompleted()` y `onError()` del `Subscriber` .

```
Subscriber<Integer> mSubscriber = new Subscriber<Integer>() {
        // NOTE THAT ALL THESE ARE CALLED BY THE OBSERVABLE
        @Override
        public void onCompleted() {
            // called when all objects are emitted
            System.out.println("onCompleted called!");
        }

        @Override
        public void onError(Throwable throwable) {
            // called when an error occurs during emitting objects
            System.out.println("onError called!");
        }

        @Override
        public void onNext(Integer integer) {
            // called for each object that is emitted
            System.out.println("onNext called with: " + integer);
```

```
        }
    };
```

Tenga en cuenta que el `Subscriber` también es genérico y puede admitir cualquier objeto. Un `Subscriber` debe suscribirse al observable llamando al método de `subscribe` en el observable.

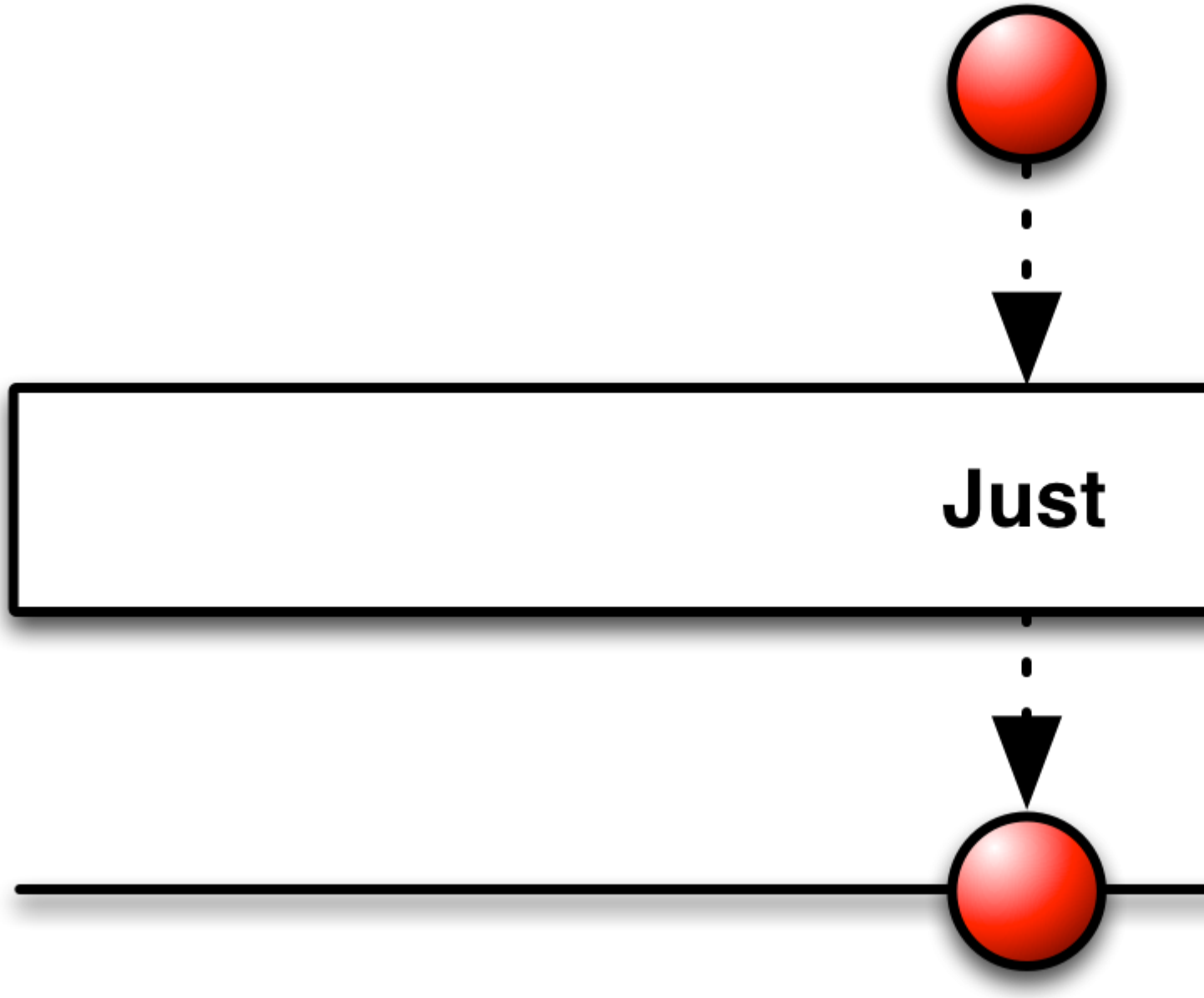
```
integerObservable.subscribe(mSubscriber);
```

Lo anterior, cuando se ejecuta, producirá el siguiente resultado:

```
onNext called with: 1
onNext called with: 2
onNext called with: 3
onCompleted called!
```

Entendiendo los diagramas de mármol

Un observable se puede considerar como un flujo de eventos. Cuando define un Observable, tiene tres escuchas: onNext, onComplete y onError. onNext se llamará cada vez que el observable adquiera un nuevo valor. Se llamará a onComplete si el Observable principal notifica que terminó de producir más valores. Se llama a onError si se lanza una excepción en cualquier momento durante la ejecución de la cadena Observable. Para mostrar los operadores en Rx, el diagrama de mármol se usa para mostrar lo que sucede con una operación en particular. A continuación se muestra un ejemplo de un simple operador observable "Just".



Los diagramas de mármol tienen un bloque horizontal que representa la operación que se está realizando, una barra vertical para representar el evento completado, una X para representar un error y cualquier otra forma representa un valor. Con eso en mente, podemos ver que "Just" solo tomará nuestro valor y hará un onNext y luego terminará con onComplete. Hay muchas más operaciones que simplemente "solo". Puede ver todas las operaciones que forman parte del proyecto ReativeX y sus implementaciones en RxJava en el sitio de ReativeX . También hay diagramas de mármol interactivos a través del sitio RxMarbles .

Lea Empezando con rx-java en línea: https://riptutorial.com/es/rx-java/topic/974/empezando-con-rx-java

Capítulo 2: Android con RxJava

Observaciones

RxAndroid solía ser una biblioteca con muchas características. Se ha dividido en muchas bibliotecas diferentes, pasando de la versión 0.25.0 a la 1.x.

Aquí se mantiene una lista de las bibliotecas que implementan las funciones disponibles antes de la 1.0.

Examples

RxAndroid - AndroidSchedulers

Esto es, literalmente, lo único que necesita para comenzar a usar RxJava en Android.

Incluya RxJava y RxAndroid en sus dependencias de gradle:

```
 // use the last version
compile 'io.reactivex.rxjava2:rxjava:2.1.1'
compile 'io.reactivex.rxjava2:rxandroid:2.0.1'
```

La adición principal de RxAndroid a RxJava es un programador para el subproceso principal de Android o subproceso de interfaz de usuario.

En su código:

```
Observable.just("one", "two", "three", "four", "five")
        .subscribeOn(Schedulers.newThread())
        .observeOn(AndroidSchedulers.mainThread())
        .subscribe(
            data -> doStuffOnMainThread(),
            error -> handleErrorOnMainThread()
        )
```

O puede crear un Programador para un `Looper` personalizado:

```
Looper backgroundLooper = // ...
Observable.just("one", "two", "three", "four", "five")
        .subscribeOn(Schedulers.newThread())
        .observeOn(AndroidSchedulers.from(backgroundLooper))
        .subscribe(
            data -> doStuffOnMainThread(),
            error -> handleErrorOnMainThread()
        )
```

Para casi todo lo demás, puede consultar la documentación estándar de RxJava.

Componentes de RxLifecycle

La biblioteca RxLifecycle facilita la vinculación de suscripciones observables a actividades de Android y fragmenta el ciclo de vida.

Tenga en cuenta que olvidarse de cancelar la suscripción de un Observable puede causar pérdidas de memoria y mantener activo su evento / fragmento de actividad después de que el sistema lo haya destruido.

Agregue la biblioteca a las dependencias:

```
// use the last version available
compile 'com.trello:rxlifecycle:0.6.1'
compile 'com.trello:rxlifecycle-components:0.6.1'
```

Luego extiende las clases de `Rx*` :

- `RxActivity` / `support.RxFragmentActivity` / `support.RxAppCompatActivity`
- `RxFragment` / `support.RxFragment`
- `RxDialogFragment` / `support.RxDialogFragment`
- `support.RxAppCompatDialogActivity`

Está todo listo, cuando se suscribe a un Observable ahora puede:

```
someObservable
    .compose(bindToLifecycle())
    .subscribe();
```

Si ejecuta esto en el método `onCreate()` de la actividad, se cancelará automáticamente la suscripción en `onDestroy()` .

Lo mismo sucede para:

- `onStart()` -> `onStop()`
- `onResume()` -> `onPause()`
- `onAttach()` -> `onDetach()` *(solo fragmento)*
- `onViewCreated()` -> `onDestroyView()` *(solo fragmento)*

Como alternativa, puede especificar el evento cuando desea que se produzca la baja de la suscripción:

De una actividad:

```
someObservable
    .compose(bindUntilEvent(ActivityEvent.DESTROY))
    .subscribe();
```

De un fragmento:

```
someObservable
```

```
    .compose(bindUntilEvent(FragmentEvent.DESTROY_VIEW))
    .subscribe();
```

También puede obtener el ciclo de vida observable usando el `lifecycle()` del método `lifecycle()` para escuchar los eventos del ciclo de vida directamente.

RxLifecycle también se puede utilizar directamente y le pasa el ciclo de vida observable:

```
.compose(RxLifecycleAndroid.bindActivity(lifecycle))
```

Si necesita manejar `Single` o `Completable` , puede hacerlo simplemente agregando respectivamente `forSingle()` o `forCompletable` después del método de enlace:

```
someSingle
    .compose(bindToLifecycle().forSingle())
    .subscribe();
```

También se puede utilizar con la biblioteca Navi .

Permisos Rx

Esta biblioteca permite el uso de RxJava con el nuevo modelo de permiso de Android M.

Agregue la biblioteca a las dependencias:

Rxjava

```
dependencies {
    compile 'com.tbruyelle.rxpermissions:rxpermissions:0.8.0@aar'
}
```

Rxjava2

```
dependencies {
    compile 'com.tbruyelle.rxpermissions2:rxpermissions:0.8.1@aar'
}
```

Uso

Ejemplo (con Retrolambda por brevedad, pero no es obligatorio):

```
// Must be done during an initialization phase like onCreate
RxPermissions.getInstance(this)
    .request(Manifest.permission.CAMERA)
    .subscribe(granted -> {
        if (granted) { // Always true pre-M
           // I can control the camera now
        } else {
           // Oups permission denied
        }
    });
```

Lea más: https://github.com/tbruyelle/RxPermissions .

Lea Android con RxJava en línea: https://riptutorial.com/es/rx-java/topic/7125/android-con-rxjava

Capítulo 3: Asignaturas

Sintaxis

- Asunto <T, R> subject = AsyncSubject.create (); // AsyncSubject predeterminado
- Asunto <T, R> subject = BehaviorSubject.create (); // Default BehaviorSubject
- Asunto <T, R> subject = PublishSubject.create (); // Predeterminado PublishSubject
- Asunto <T, R> subject = ReplaySubject.create (); // ReplaySubject predeterminado
- mySafeSubject = new SerializedSubject (unSafeSubject); // Convertir un unsafeSubject en un safeSubject - generalmente para temas de múltiples hilos

Parámetros

Parámetros	Detalles
T	Tipo de entrada
R	Tipo de salida

Observaciones

Esta documentación proporciona detalles y explicaciones sobre el `Subject` . Para más información y lecturas adicionales, por favor visite la documentación oficial .

Examples

Materias basicas

Un `Subject` en RxJava es una clase que es tanto un `Observable` como un `Observer` . Básicamente, esto significa que puede actuar como un `Observable` y pasar entradas a los suscriptores y como un `Observer` para obtener entradas de otro Observable.

```
Subject<String, String> subject = PublishSubject.create();
subject.subscribe(System.out::print);
subject.onNext("Hello, World!");
```

Las impresiones de arriba "¡Hola mundo!" consolar utilizando `Subjects` .

Explicación

1. La primera línea de código define un nuevo `Subject` de tipo `PublishSubject`

```
Subject<String, String> subject = PublishSubject.create();
    |      |       |       |                  |
```

```
subject<input, output>  name   = default publish subject
```

2. La segunda línea se suscribe al sujeto, mostrando el comportamiento del `Observer` .

```
subject.subscribe(System.out::print);
```

   Esto permite al `Subject` tomar entradas como un suscriptor regular

3. La tercera línea llama al método `onNext` del sujeto, mostrando el comportamiento `Observable` .

```
subject.onNext("Hello, World!");
```

   Esto permite al `Subject` dar entradas a todos los que se suscriban.

Los tipos

Un `Subject` (en RxJava) puede ser de cualquiera de estos cuatro tipos:

- AsyncSubject
- Comportamiento
- PublishSubject
- ReplaySubject

Además, un `Subject` puede ser de tipo `SerializedSubject` . Este tipo garantiza que el `Subject` no infringe el *contrato observable* (que especifica que todas las llamadas deben ser serializadas)

Otras lecturas:

- Para usar o no usar el tema del blog de Dave Sexton

PublishSubject

`PublishSubject` emite a un `Observer` solo aquellos elementos que emite la fuente `Observable` después de la suscripción.

Un ejemplo simple de `PublishSubject` :

```
Observable<Long> clock = Observable.interval(500, TimeUnit.MILLISECONDS);
Subject<Long, Long> subjectLong = PublishSubject.create();

clock.subscribe(subjectLong);

System.out.println("sub1 subscribing...");
subjectLong.subscribe(l -> System.out.println("sub1 -> " + l));
Thread.sleep(3000);
System.out.println("sub2 subscribing...");
subjectLong.subscribe(l -> System.out.println("sub2 -> " + l));
Thread.sleep(5000);
```

Salida:

```
sub1 subscribing...
sub1 -> 0
sub1 -> 1
sub2 subscribing...
sub1 -> 2
sub2 -> 2
sub1 -> 3
sub2 -> 3
```

En el ejemplo anterior, un `PublishSubject` suscribe a un `Observable` que actúa como un reloj y emite elementos (Largo) cada 500 mili segundos. Como se ve en la salida, `PublishSubject` pasa los `PublishSubject` que recibe de la fuente (`clock`) a sus suscriptores (`sub1` y `sub2`).

Un `PublishSubject` puede comenzar a emitir elementos tan pronto como se crea, sin ningún observador, lo que corre el riesgo de perder uno o más elementos hasta que un observador pueda suscribirse.

```
createClock(); // 3 lines moved for brevity. same as above example

Thread.sleep(5000); // introduces a delay before first subscribe

sub1andsub2(); // 6 lines moved for brevity. same as above example
```

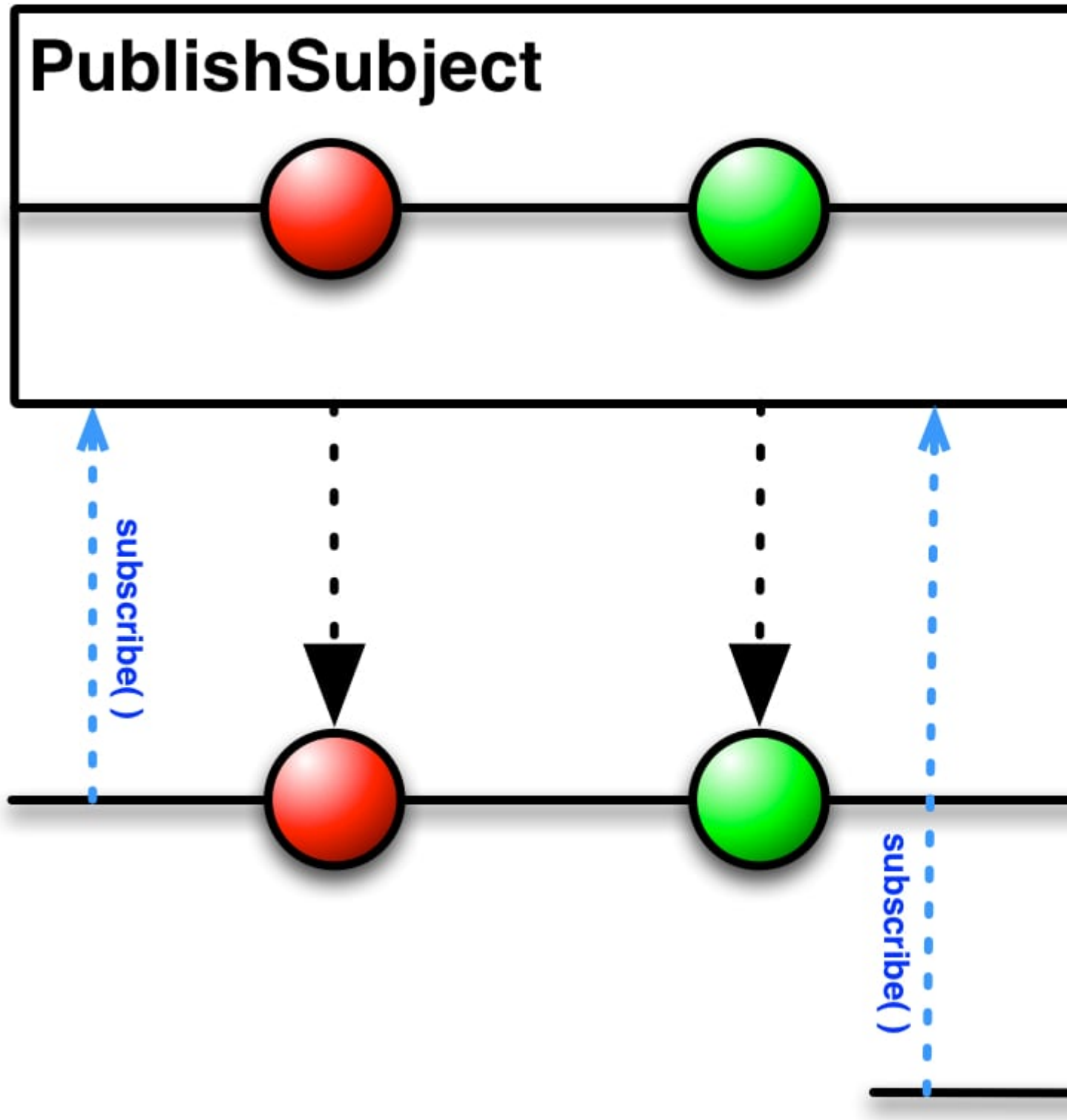
Salida:

```
sub1 subscribing...
sub1 -> 10
sub1 -> 11
sub2 subscribing...
sub1 -> 12
sub2 -> 12
sub1 -> 13
sub2 -> 13
```

Observe que `sub1` emite valores a partir de `10` . El retraso de 5 segundos introducido causó una *pérdida* de artículos. Estos no pueden ser reproducidos. Esto esencialmente hace de `PublishSubject` un `Hot Observable` .

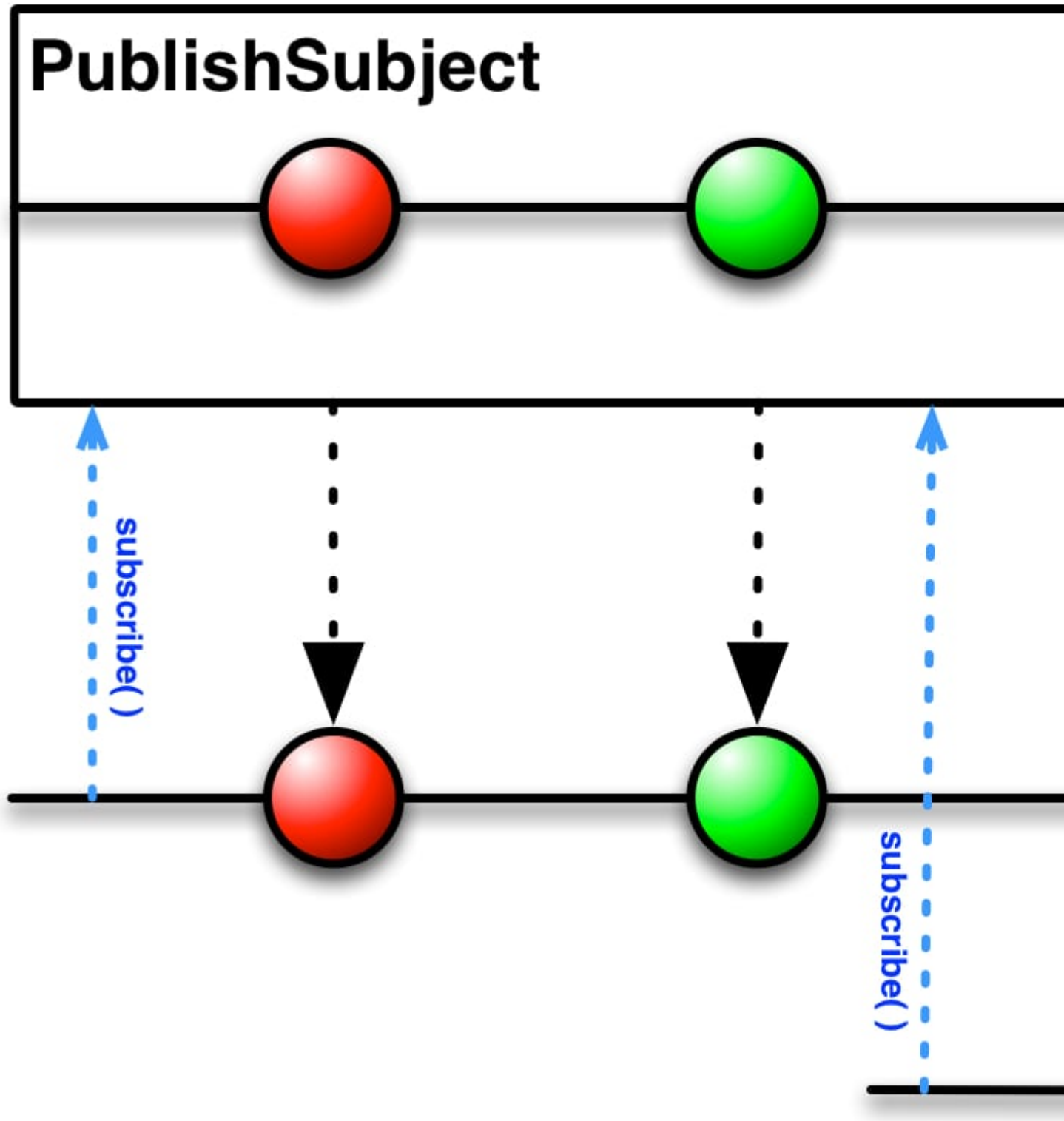
Además, tenga en cuenta que si un observador se suscribe a `PublishSubject` después de que haya emitido ***n*** elementos, estos ***n*** elementos *no se* pueden reproducir para este observador.

A continuación se muestra el diagrama de mármol de `PublishSubject`



`PublishSubject` emite elementos a todos los que se han suscrito, en cualquier momento antes de que se `onCompleted` al `onCompleted` de la fuente `Observable` .

Si la fuente `Observable` termina con un error, `PublishSubject` no emitirá ningún elemento a los observadores posteriores, sino que simplemente transmitirá la notificación de error desde la fuente observable.



Caso de uso

Supongamos que desea crear una aplicación que supervise los precios de las acciones de una determinada empresa y la envíe a todos los clientes que la soliciten.

```
/* Dummy stock prices */
Observable<Integer> prices = Observable.just(11, 12, 14, 11, 10, 12, 15, 11, 10);

/* Your server */
PublishSubject<Integer> watcher = PublishSubject.create();
/* subscribe to listen to stock price changes and push to observers/clients */
```

```
prices.subscribe(watcher);

/* Client application */
stockWatcher = getWatcherInstance(); // gets subject
Subscription steve = stockWatcher.subscribe(i -> System.out.println("steve watching " + i));
Thread.sleep(1000);
System.out.println("steve stops watching");
steve.unsubscribe();
```

En el caso de uso de ejemplo anterior, `PublishSubject` actúa como un puente para pasar los valores de su servidor a todos los clientes que se suscriben a su `watcher` .

Otras lecturas:

- PublishSubject javadocs
- Blog de Thomas Nield (Lectura avanzada)

Lea Asignaturas en línea: https://riptutorial.com/es/rx-java/topic/3287/asignaturas

Capítulo 4: Contrapresion

Examples

Introducción

La contrapresión es cuando en un proceso de procesamiento `Observable` , algunas etapas asíncronas no pueden procesar los valores con la rapidez suficiente y necesitan una forma de decirle al productor ascendente que disminuya la velocidad.

El caso clásico de la necesidad de contrapresión es cuando el productor es una fuente candente:

```
PublishSubject<Integer> source = PublishSubject.create();

source
.observeOn(Schedulers.computation())
.subscribe(v -> compute(v), Throwable::printStackTrace);

for (int i = 0; i < 1_000_000; i++) {
    source.onNext(i);
}

Thread.sleep(10_000);
```

En este ejemplo, el hilo principal producirá 1 millón de artículos para un consumidor final que lo procesa en un hilo de fondo. Es probable que el método de `compute(int)` tome algún tiempo, pero la sobrecarga de la cadena de operadores `Observable` también puede aumentar el tiempo que se tarda en procesar los elementos. Sin embargo, el subproceso que produce con el bucle for no puede saber esto y continúa en `onNext` .

Internamente, los operadores asíncronos tienen buffers para mantener dichos elementos hasta que puedan procesarse. En el Rx.NET clásico y el primer RxJava, estos buffers no tenían límites, lo que significa que probablemente obtendrían casi todos los 1 millón de elementos del ejemplo. El problema comienza cuando hay, por ejemplo, 1 billón de elementos o la misma secuencia de 1 millón aparece 1000 veces en un programa, lo que lleva a `OutOfMemoryError` y generalmente se ralentiza debido a una sobrecarga de GC excesiva.

De manera similar a cómo el manejo de errores se convirtió en un ciudadano de primera clase y en operadores recibidos para lidiar con él (a través de los operadores `onErrorXXX`), la contrapresión es otra propiedad de los flujos de datos que el programador debe considerar y manejar (a través de los operadores `onBackpressureXXX`).

Más allá del `PublishSubject` anterior, hay otros operadores que no admiten la contrapresión, principalmente debido a razones funcionales. Por ejemplo, el `interval` del operador emite valores periódicamente, lo que hace que la contrapresión lleve a cambios en el período en relación con un reloj de pared.

En RxJava moderno, la mayoría de los operadores asíncronos ahora tienen un búfer interno

acotado, como `observeOn` arriba y cualquier intento de desbordar este búfer terminará toda la secuencia con `MissingBackpressureException` . La documentación de cada operador tiene una descripción sobre su comportamiento de contrapresión.

Sin embargo, la contrapresión está presente más sutilmente en las secuencias frías regulares (que no lo hacen y no deberían producir la `MissingBackpressureException`). Si el primer ejemplo es reescrito:

```
Observable.range(1, 1_000_000)
.observeOn(Schedulers.computation())
.subscribe(v -> compute(v), Throwable::printStackTrace);

Thread.sleep(10_000);
```

No hay error y todo funciona sin problemas con el uso de memoria pequeña. La razón de esto es que muchos operadores de origen pueden "generar" valores a pedido y, por lo tanto, `observeOn` del operador puede decir que el `range` genera a lo sumo tantos valores que el búfer de `observeOn` puede contener de una vez sin desbordamiento.

Esta negociación se basa en el concepto de informática de co-rutinas (yo te llamo, tú me llamas). El operador `range` envía una devolución de llamada, en la forma de una implementación del `Producer` interfaz, a la `observeOn` llamando a su (interior `Subscriber` 's) `setProducer` . A cambio, `observeOn` llama a `Producer.request(n)` con un valor para indicar el `range` que tiene permitido producir (es decir, `onNext`) muchos elementos **adicionales** . Entonces, es responsabilidad de `observeOn` llamar al método de `request` en el momento adecuado y con el valor correcto para mantener los datos fluyendo pero sin desbordarse.

Rara vez es necesario expresar la contrapresión en los consumidores finales (porque son síncronos respecto a su flujo ascendente inmediato y la contrapresión ocurre naturalmente debido al bloqueo de la pila de llamadas), pero puede ser más fácil comprender su funcionamiento:

```
Observable.range(1, 1_000_000)
.subscribe(new Subscriber<Integer>() {
    @Override
    public void onStart() {
        request(1);
    }

    public void onNext(Integer v) {
        compute(v);

        request(1);
    }

    @Override
    public void onError(Throwable ex) {
        ex.printStackTrace();
    }

    @Override
    public void onCompleted() {
        System.out.println("Done!");
    }
```

```
});
```

Aquí la implementación de `onStart` indica el `range` para producir su primer valor, que luego se recibe en `onNext` . Una vez que finaliza la `compute(int)` , el otro valor se solicita desde el `range` . En una implementación ingenua de `range` , dicha llamada recurriría recursivamente `onNext` , lo que llevaría a `StackOverflowError` que por supuesto no es deseable.

Para evitar esto, los operadores utilizan la llamada lógica de trampolín que evita tales llamadas reentrantes. En términos de `range` , recordará que hubo una `request(1)` mientras que llamó `onNext()` y una vez que `onNext()` devuelve, realizará otra ronda y llamará a `onNext()` con el siguiente valor entero. Por lo tanto, si se intercambian los dos, el ejemplo sigue funcionando igual:

```
@Override
public void onNext(Integer v) {
    request(1);

    compute(v);
}
```

Sin embargo, esto no es cierto para `onStart` . Aunque la infraestructura `Observable` garantiza que se llamará como máximo una vez en cada `Subscriber` , la llamada a `request(1)` puede desencadenar la emisión de un elemento de inmediato. Si uno tiene lógica de inicialización después de la llamada a `request(1)` que necesita `onNext` , puede terminar con excepciones:

```
Observable.range(1, 1_000_000)
.subscribe(new Subscriber<Integer>() {

    String name;

    @Override
    public void onStart() {
        request(1);

        name = "RangeExample";
    }

    @Override
    public void onNext(Integer v) {
        compute(name.length + v);

        request(1);
    }

    // ... rest is the same
});
```

En este caso síncrono, se lanzará una `NullPointerException` inmediatamente mientras se sigue ejecutando `onStart` . Un error más sutil ocurre si la llamada a `request(1)` desencadena una llamada asíncrona a `onNext` en algún otro hilo y lee el `name` en `onNext` carreras `onNext` escriben en la `request` posterior al `onStart` .

Por lo tanto, uno debe hacer toda la inicialización de campo en `onStart` o incluso antes de eso y llamar a `request()` último lugar. Las implementaciones de `request()` en los operadores aseguran

una relación correcta antes del suceso (o en otros términos, liberación de memoria o valla completa) cuando sea necesario.

Los operadores onBackpressureXXX

La mayoría de los desarrolladores encuentran una contrapresión cuando su aplicación falla con `MissingBackpressureException` y la excepción generalmente apunta al operador de `observeOn` . La causa real suele ser el uso no `PublishSubject` de `PublishSubject` , `timer()` o `interval()` u operadores personalizados creados a través de `create()` .

Hay varias maneras de lidiar con tales situaciones.

Aumentar los tamaños de búfer.

A veces, tales desbordamientos ocurren debido a fuentes ráfagas. De repente, el usuario toca la pantalla con demasiada rapidez y `observeOn` el búfer interno de 16 elementos predeterminado de `observeOn` en los desbordamientos de Android.

La mayoría de los operadores sensibles a la contrapresión en las versiones recientes de RxJava ahora permiten a los programadores especificar el tamaño de sus búferes internos. Los parámetros relevantes generalmente se llaman `bufferSize` , `prefetch` o `capacityHint` . Dado el ejemplo desbordante en la introducción, solo podemos aumentar el tamaño del búfer de `observeOn` para tener suficiente espacio para todos los valores.

```
PublishSubject<Integer> source = PublishSubject.create();

source.observeOn(Schedulers.computation(), 1024 * 1024)
      .subscribe(e -> { }, Throwable::printStackTrace);

for (int i = 0; i < 1_000_000; i++) {
    source.onNext(i);
}
```

Sin embargo, tenga en cuenta que, en general, esto puede ser solo una solución temporal, ya que el desbordamiento todavía puede ocurrir si la fuente produce en exceso el tamaño del búfer previsto. En este caso, uno puede usar uno de los siguientes operadores.

Valores por lotes / saltos con operadores estándar

En caso de que los datos de origen puedan procesarse más eficientemente en lotes, se puede reducir la probabilidad de `MissingBackpressureException` mediante el uso de uno de los operadores de procesamiento por lotes estándar (por tamaño y / o por tiempo).

```
PublishSubject<Integer> source = PublishSubject.create();

source
      .buffer(1024)
      .observeOn(Schedulers.computation(), 1024)
      .subscribe(list -> {
```

```
          list.parallelStream().map(e -> e * e).first();
      }, Throwable::printStackTrace);

for (int i = 0; i < 1_000_000; i++) {
    source.onNext(i);
}
```

Si algunos de los valores se pueden ignorar de manera segura, uno puede usar el muestreo (con tiempo u otro Observable) y los operadores de regulación (`throttleFirst` , `throttleLast` , `throttleWithTimeout`).

```
PublishSubject<Integer> source = PublishSubject.create();

source
      .sample(1, TimeUnit.MILLISECONDS)
      .observeOn(Schedulers.computation(), 1024)
      .subscribe(v -> compute(v), Throwable::printStackTrace);

for (int i = 0; i < 1_000_000; i++) {
    source.onNext(i);
}
```

Tenga en cuenta, sin embargo, que estos operadores solo reducen la tasa de recepción de valor en sentido descendente y, por lo tanto, aún pueden llevar a la `MissingBackpressureException` .

onBackpressureBuffer ()

Este operador en su forma sin parámetros reintroduce un búfer ilimitado entre la fuente ascendente y el operador descendente. Ser ilimitado significa que mientras la JVM no se quede sin memoria, puede manejar casi cualquier cantidad que provenga de una fuente ráfaga.

```
 Observable.range(1, 1_000_000)
           .onBackpressureBuffer()
           .observeOn(Schedulers.computation(), 8)
           .subscribe(e -> { }, Throwable::printStackTrace);
```

En este ejemplo, `observeOn` va con un tamaño de búfer muy bajo, sin embargo, no hay una `MissingBackpressureException` ya que `onBackpressureBuffer` absorbe todos los valores de 1 millón y entrega lotes pequeños de este para `observeOn` .

Tenga en cuenta, sin embargo, que `onBackpressureBuffer` consume su fuente de manera ilimitada, es decir, sin aplicarle ninguna contrapresión. Esto tiene la consecuencia de que incluso una fuente de soporte de contrapresión como el `range` se realizará completamente.

Hay 4 sobrecargas adicionales de `onBackpressureBuffer`

onBackpressureBuffer (capacidad int)

Esta es una versión limitada que señala a `BufferOverflowError` en caso de que su buffer alcance la capacidad dada.

```
Observable.range(1, 1_000_000)
          .onBackpressureBuffer(16)
          .observeOn(Schedulers.computation())
          .subscribe(e -> { }, Throwable::printStackTrace);
```

La relevancia de este operador está disminuyendo a medida que más y más operadores ahora permiten configurar sus tamaños de búfer. Para el resto, esto brinda la oportunidad de "ampliar su búfer interno" al tener un número mayor con `onBackpressureBuffer` que su valor predeterminado.

onBackpressureBuffer (capacidad int, Action0 onOverflow)

Esta sobrecarga llama a una acción (compartida) en caso de que ocurra un desbordamiento. Su utilidad es bastante limitada, ya que no se proporciona otra información sobre el desbordamiento que la pila de llamadas actual.

onBackpressureBuffer (capacidad int, Action0 onOverflow, BackpressureOverflow.Strategy estrategia)

Esta sobrecarga es en realidad más útil, ya que permite definir qué hacer en caso de que se haya alcanzado la capacidad. `BackpressureOverflow.Strategy` es una interfaz en realidad, pero la clase `BackpressureOverflow` ofrece 4 campos estáticos con implementaciones que representan acciones típicas:

- `ON_OVERFLOW_ERROR` : este es el comportamiento predeterminado de las dos sobrecargas anteriores, que señala una `BufferOverflowException`
- `ON_OVERFLOW_DEFAULT` : actualmente es lo mismo que `ON_OVERFLOW_ERROR`
- `ON_OVERFLOW_DROP_LATEST` : si ocurriera un desbordamiento, el valor actual simplemente se ignorará y solo se entregarán los valores antiguos una vez que las solicitudes posteriores.
- `ON_OVERFLOW_DROP_OLDEST` : elimina el elemento más antiguo del búfer y le agrega el valor actual.

```
Observable.range(1, 1_000_000)
          .onBackpressureBuffer(16, () -> { },
              BufferOverflowStrategy.ON_OVERFLOW_DROP_OLDEST)
          .observeOn(Schedulers.computation())
          .subscribe(e -> { }, Throwable::printStackTrace);
```

Tenga en cuenta que las dos últimas estrategias causan discontinuidad en el flujo a medida que abandonan los elementos. Además, no señalarán la `BufferOverflowException` .

onBackpressureDrop ()

Cuando el flujo descendente no esté listo para recibir valores, este operador eliminará ese elemento de la secuencia. Uno puede pensar en ello como una capacidad de 0 en el `onBackpressureBuffer` con la estrategia `ON_OVERFLOW_DROP_LATEST` .

Este operador es útil cuando uno puede ignorar de forma segura los valores de una fuente (como los movimientos del mouse o las señales de ubicación de GPS actuales) ya que habrá valores

más actualizados más adelante.

```
component.mouseMoves()
.onBackpressureDrop()
.observeOn(Schedulers.computation(), 1)
.subscribe(event -> compute(event.x, event.y));
```

Puede ser útil en conjunto con el `interval()` operador de origen `interval()` . Por ejemplo, si uno desea realizar alguna tarea periódica en segundo plano, pero cada iteración puede durar más que el período, es seguro eliminar la notificación de intervalo de exceso, ya que habrá más adelante:

```
Observable.interval(1, TimeUnit.MINUTES)
.onBackpressureDrop()
.observeOn(Schedulers.io())
.doOnNext(e -> networkCall.doStuff())
.subscribe(v -> { }, Throwable::printStackTrace);
```

Existe una sobrecarga de este operador: `onBackpressureDrop(Action1<? super T> onDrop)` donde se llama a la acción (compartida) y se descarta el valor. Esta variante permite limpiar los propios valores (por ejemplo, liberar recursos asociados).

onBackpressureLatest ()

El operador final conserva solo el último valor y prácticamente sobrescribe los valores antiguos y no entregados. Uno puede pensar en esto como una variante del `onBackpressureBuffer` con una capacidad de 1 y la estrategia de `ON_OVERFLOW_DROP_OLDEST` .

A diferencia de `onBackpressureDrop` siempre hay un valor disponible para el consumo si el flujo descendente se está quedando atrás. Esto puede ser útil en algunas situaciones similares a la telemetría en las que los datos pueden venir en un patrón de ráfagas, pero solo el último es interesante para el procesamiento.

Por ejemplo, si el usuario hace mucho clic en la pantalla, todavía queremos reaccionar a su última entrada.

```
component.mouseClicks()
.onBackpressureLatest()
.observeOn(Schedulers.computation())
.subscribe(event -> compute(event.x, event.y), Throwable::printStackTrace);
```

El uso de `onBackpressureDrop` en este caso llevaría a una situación en la que el último clic se cae y deja al usuario preguntándose por qué no se ejecutó la lógica empresarial.

Creando fuentes de datos con presión

Crear fuentes de datos con presión es una tarea relativamente más fácil cuando se trata de la presión en general, ya que la biblioteca ya ofrece métodos estáticos en `Observable` que manejan la presión para el desarrollador. Podemos distinguir dos tipos de métodos de fábrica: los "generadores" fríos que devuelven y generan elementos basados en la demanda descendente y

los "impulsores" en caliente que generalmente conectan las fuentes de datos no reactivos y / o no recuperables y superponen el manejo de la contrapresión. ellos.

sólo

La fuente de conciencia de contrapresión más básica se crea a través de `just` :

```
Observable.just(1).subscribe(new Subscriber<Integer>() {
    @Override
    public void onStart() {
        request(0);
    }

    @Override
    public void onNext(Integer v) {
        System.out.println(v);
    }

    // the rest is omitted for brevity
}
```

Como no solicitamos explícitamente en `onStart` , esto no imprimirá nada. `just` es genial cuando hay un valor constante que nos gustaría iniciar una secuencia.

Desafortunadamente, `just` se confunde a menudo con una forma de calcular algo de forma dinámica para que lo consuman los `Subscriber` :

```
int counter;

int computeValue() {
   return ++counter;
}

Observable<Integer> o = Observable.just(computeValue());

o.subscribe(System.out:println);
o.subscribe(System.out:println);
```

Sorprendente para algunos, esto imprime 1 dos veces en lugar de imprimir 1 y 2 respectivamente. Si la llamada se reescribe, se vuelve obvio por qué funciona así:

```
int temp = computeValue();

Observable<Integer> o = Observable.just(temp);
```

El `computeValue` se llama como parte de la rutina principal y no en respuesta a la suscripción de los suscriptores.

de Callable

Lo que la gente realmente necesita es el método de `fromCallable` :

```
Observable<Integer> o = Observable.fromCallable(() -> computeValue());
```

Aquí, `computeValue` se ejecuta solo cuando un suscriptor se suscribe y para cada uno de ellos, imprime el 1 y el 2. `fromCallable` . Desde `fromCallable` también es compatible con la contrapresión y no emitirá el valor calculado a menos que se solicite. Tenga en cuenta, sin embargo, que el cálculo sucede de todos modos. En caso de que el cálculo en sí se retrase hasta que el flujo descendente solicite, podemos usar `just` con `map` :

```
Observable.just("This doesn't matter").map(ignored -> computeValue())...
```

`just` no emitirá su valor constante hasta que se solicite cuando se asigna al resultado de `computeValue` , que aún se llama para cada suscriptor individualmente.

desde

Si los datos ya está disponible como un conjunto de objetos, una lista de objetos o cualquier `Iterable` fuente, el respectivo `from` sobrecargas se encargará de la contrapresión y la emisión de dichas fuentes:

```
 Observable.from(Arrays.asList(1, 2, 3, 4, 5)).subscribe(System.out::println);
```

Para mayor comodidad (y evitar las advertencias sobre creación de la matriz genérica) existen 2 a 10 sobrecargas argumento para `just` que el delegado internamente a `from` .

El `from(Iterable)` también da una oportunidad interesante. La generación de muchos valores puede expresarse en forma de una máquina de estado. Cada elemento solicitado desencadena una transición de estado y el cálculo del valor devuelto.

Escribir máquinas de estado como `Iterable` s es un tanto complicado (pero aún más fácil que escribir un `Observable` para consumirlo) y, a diferencia de C #, Java no tiene ningún soporte del compilador para construir dichas máquinas de estado simplemente escribiendo código de aspecto clásico (con `yield return` y `yield break`). Algunas bibliotecas ofrecen ayuda, como `AbstractIterable` Google Guava e `Ix.generate()` `Ix.forloop()` e `Ix.forloop()` . Estos son por sí mismos dignos de una serie completa, así que veamos una fuente de `Iterable` muy básica que repite un valor constante de forma indefinida:

```
Iterable<Integer> iterable = () -> new Iterator<Integer>() {
    @Override
    public boolean hasNext() {
        return true;
    }

    @Override
    public Integer next() {
        return 1;
    }
};

Observable.from(iterable).take(5).subscribe(System.out::println);
```

Si consumiéramos el `iterator` través del bucle for clásico, daríamos como resultado un bucle infinito. Ya que construimos un `Observable` partir de él, podemos expresar nuestra voluntad de consumir solo los primeros 5 y dejar de solicitar cualquier cosa. Este es el verdadero poder de evaluar y computar perezosamente dentro de `Observable` s.

crear (SyncOnSubscribe)

A veces, la fuente de datos que se convertirá en el mundo reactivo en sí es sincrónica (de bloqueo) y de tipo pull, es decir, tenemos que llamar a algún método de `get` o `read` para obtener el siguiente dato. Uno podría, por supuesto, convertir eso en un `Iterable` pero cuando tales fuentes están asociadas con recursos, podemos filtrar esos recursos si el flujo descendente anula la suscripción de la secuencia antes de que finalice.

Para manejar tales casos, RxJava tiene la clase `SyncOnSubscribe` . Uno puede ampliarlo e implementar sus métodos o usar uno de sus métodos de fábrica basados en lambda para crear una instancia.

```
SyncOnSubscribe<Integer, InputStream> binaryReader = SyncOnSubscribe.createStateful(
     () -> new FileInputStream("data.bin"),
     (inputstream, output) -> {
         try {
             int byte = inputstream.read();
             if (byte < 0) {
                 output.onCompleted();
             } else {
                 output.onNext(byte);
             }
         } catch (IOException ex) {
             output.onError(ex);
         }
         return inputstream;
     },
     inputstream -> {
         try {
             inputstream.close();
         } catch (IOException ex) {
             RxJavaHooks.onError(ex);
         }
     }
 );

 Observable<Integer> o = Observable.create(binaryReader);
```

En general, `SyncOnSubscribe` utiliza 3 devoluciones de llamada.

Las primeras devoluciones de llamada permiten crear un estado por suscriptor, como `FileInputStream` en el ejemplo; El archivo se abrirá de forma independiente para cada suscriptor individual.

La segunda devolución de llamada toma este objeto de estado y proporciona un `Observer` salida cuyos métodos `onXXX` pueden llamarse para emitir valores. Esta devolución de llamada se ejecuta tantas veces como se solicite en sentido descendente. En cada invocación, debe llamar `onNext` como máximo una vez, opcionalmente seguido de `onError` o `onCompleted` . En el ejemplo, llamamos

`onCompleted()` si el byte de lectura es negativo, lo que indica el final del archivo, y llamamos `onError` en caso de que la lectura lance una `IOException` .

La devolución de llamada final se invoca cuando las subscripciones subsiguientes (cerrando la entrada) o cuando la devolución de llamada anterior llamó a los métodos del terminal; Permite liberar recursos. Dado que no todas las fuentes necesitan todas estas características, los métodos estáticos de `SyncOnSubscribe` permiten crear instancias sin ellos.

Desafortunadamente, muchas llamadas de métodos a través de la JVM y otras bibliotecas arrojan excepciones comprobadas y deben ajustarse en `try-catch` ya que las interfaces funcionales utilizadas por esta clase no permiten lanzar excepciones controladas.

Por supuesto, podemos imitar otras fuentes típicas, como un rango ilimitado con él:

```
SyncOnSubscribe.createStateful(
     () -> 0,
     (current, output) -> {
         output.onNext(current);
         return current + 1;
     },
     e -> { }
);
```

En esta configuración, la `current` comienza con `0` y la próxima vez que se invoca la lambda, el parámetro `current` ahora tiene `1` .

Hay una variante de `SyncOnSubscribe` llamada `AsyncOnSubscribe` que parece bastante similar, con la excepción de que la devolución de llamada central también toma un valor largo que representa la cantidad solicitada y la devolución de llamada debe generar un `Observable` con la misma longitud. Esta fuente luego concatena todos estos `Observable` en una sola secuencia.

```
 AsyncOnSubscribe.createStateful(
     () -> 0,
     (state, requested, output) -> {
         output.onNext(Observable.range(state, (int)requested));
         return state + 1;
     },
     e -> { }
 );
```

Existe una discusión continua (acalorada) sobre la utilidad de esta clase y, en general, no se recomienda porque rompe las expectativas sobre cómo emitirá esos valores generados y cómo responderá, o incluso qué tipo de valores de solicitud recibirá. Escenarios de consumo más complejos.

crear (emisor)

A veces, la fuente que se envuelve en un `Observable` ya está caliente (como los movimientos del mouse) o fría, pero no se puede volver a comprimir en su API (como una devolución de llamada de red asíncrona).

Para manejar estos casos, una versión reciente de RxJava introdujo el método de fábrica `create(emitter)` . Toma dos parámetros:

- una devolución de llamada que se llamará con una instancia de la interfaz del `Emitter<T>` para cada suscriptor entrante,
- una enumeración `Emitter.BackpressureMode` que obliga al desarrollador a especificar el comportamiento de contrapresión que se aplicará. Tiene los modos habituales, similares a `onBackpressureXXX` además de señalar una `MissingBackpressureException` o simplemente ignorar dicho desbordamiento dentro de él.

Tenga en cuenta que actualmente no admite parámetros adicionales para esos modos de contrapresión. Si se necesita esa personalización, usar `NONE` como modo de contrapresión y aplicar `onBackpressureXXX` relevante en el `Observable` resultante es el camino a seguir.

El primer caso típico para su uso cuando uno quiere interactuar con una fuente basada en push, como los eventos GUI. Esas API presentan alguna forma de llamadas `addListener` / `removeListener` que se pueden utilizar:

```
Observable.create(emitter -> {
    ActionListener al = e -> {
        emitter.onNext(e);
    };

    button.addActionListener(al);

    emitter.setCancellation(() ->
        button.removeListener(al));

}, BackpressureMode.BUFFER);
```

El `Emitter` es relativamente sencillo de usar; se puede llamar a `onNext` , `onError` y `onCompleted` en él y el operador se encarga de la gestión de la contrapresión y la baja de suscripción por su cuenta. Además, si la API ajustada admite la cancelación (como la eliminación del oyente en el ejemplo), se puede usar `setCancellation` (o `setSubscription` para los recursos de `Subscription`) para registrar una devolución de llamada de cancelación que se invoca cuando el `onError` la `onError` la suscripción o `onError` / `onCompleted` se llama en el proporcionado `Emitter` ejemplo.

Estos métodos permiten que solo se asocie un solo recurso con el emisor a la vez, y al configurar uno nuevo se cancela la suscripción del antiguo de forma automática. Si uno tiene que manejar múltiples recursos, cree una `CompositeSubscription` a `CompositeSubscription` , asóciela con el emisor y luego agregue más recursos a la `CompositeSubscription` a `CompositeSubscription` sí:

```
Observable.create(emitter -> {
    CompositeSubscription cs = new CompositeSubscription();

    Worker worker = Schedulers.computation().createWorker();

    ActionListener al = e -> {
        emitter.onNext(e);
    };

    button.addActionListener(al);
```

```
    cs.add(worker);
    cs.add(Subscriptions.create(() ->
        button.removeActionListener(al));

    emitter.setSubscription(cs);

}, BackpressureMode.BUFFER);
```

El segundo escenario usualmente involucra alguna API asíncrona basada en la devolución de llamada que se debe convertir en un `Observable` .

```
Observable.create(emitter -> {

    someAPI.remoteCall(new Callback<Data>() {
        @Override
        public void onSuccess(Data data) {
            emitter.onNext(data);
            emitter.onCompleted();
        }

        @Override
        public void onFailure(Exception error) {
            emitter.onError(error);
        }
    });

}, BackpressureMode.LATEST);
```

En este caso, la delegación trabaja de la misma manera. Desafortunadamente, por lo general, estas API de estilo de devolución de llamada clásicas no admiten la cancelación, pero si lo hacen, se puede configurar su cancelación como en los ejemplos anteriores (aunque quizás sea una forma más complicada). Observe el uso de la `LATEST` modo de contrapresión; Si sabemos que solo habrá un único valor, no necesitamos la estrategia `BUFFER` , ya que asigna un búfer de 128 elementos predeterminado (que crece según sea necesario) que nunca se utilizará por completo.

Lea Contrapresion en línea: https://riptutorial.com/es/rx-java/topic/2341/contrapresion

Capítulo 5: Examen de la unidad

Observaciones

Debido a que todos los métodos de Programadores son estáticos, las pruebas unitarias que utilizan los enganches RxJava no se pueden ejecutar en paralelo en la misma instancia de JVM. Si estuvieran donde, un TestScheduler se eliminaría en medio de una prueba de unidad. Esa es básicamente la desventaja de usar la clase Schedulers.

Examples

Suscriptor de prueba

Los Suscriptores de Prueba le permiten evitar el trabajo creando su propio Suscriptor o Acción de suscripción <?> Para verificar que ciertos valores se entregaron, cuántos hay, si el Observable se completó, se generó una excepción y mucho más.

Empezando

Este ejemplo solo muestra una afirmación de que los valores 1,2,3 y 4 se transfirieron al Observable a través de onNext.

```
TestSubscriber<Integer> ts = TestSubscriber.create();
Observable.just(1,2,3,4).subscribe(ts);
ts.assertValues(1,2,3,4); // Success
```

`assertValues` afirma que el recuento es correcto. Si solo pasara algunos de los valores, la afirmación fallaría.

```
TestSubscriber<Integer> ts = TestSubscriber.create();
Observable.just(1,2,3,4).subscribe(ts);
ts.assertValues(1,2,3); // Fail
```

`assertValues` utiliza el método `equals` cuando hace aserciones. Esto le permite probar fácilmente las clases que se tratan como datos.

```
TestSubscriber<Object> ts = TestSubscriber.create();
Observable.just(new Object(), new Object()).subscribe(ts);
ts.assertValues(new Object(), new Object()); // Fail
```

Este ejemplo muestra una clase que tiene un igual definido y afirma los valores del Observable.

```
public class Room {

    public String floor;
```

```
    public String number;

    @Override
    public boolean equals(Object o) {
        if (o == this) {
            return true;
        }
        if (o instanceof Room) {
            Room that = (Room) o;
            return (this.floor.equals(that.floor))
                    && (this.number.equals(that.number));
        }
        return false;
    }
}

TestSubscriber<Room> ts = TestSubscriber.create();
Observable.just(new Room("1", "10")).subscribe(ts);
ts.assertValue(new Room("1", "10"); // Success
```

También tenga en cuenta que usamos `assertValue` más `assertValue` porque solo necesitamos verificar un elemento.

Conseguir todos los eventos

Si es necesario, también puede solicitar todos los eventos como una lista.

```
TestSubscriber<Integer> ts = TestSubscriber.create();
Observable.just(1,2,3,4).subscribe(ts);
List<Integer> onNextEvents = ts.getOnNextEvents();
List<Throwable> onErrorEvents = ts.getOnErrorEvents();
List<Notification<Integer>> onCompletedEvents = ts.getOnCompletedEvents();
```

Afirmación de eventos

Si desea realizar pruebas más exhaustivas en sus eventos, puede combinar `getOnNextEvents` (u `getOn*Events`) con su biblioteca de aserción favorita:

```
Observable<Integer> obs = Observable.just(1,2,3,4)
    .filter( x -> x % 2 == 0);

// note that we instanciate TestSubscriber via the constructor here
TestSubscriber<Integer> ts = new TestSubscriber();
obs.subscribe(ts);

// Note that we are not using Observable#forEach here
// but java.lang.Iterable#forEach.
// You should never use Observable#forEach unless you know
// exactly what you're doing
ts.getOnNextEvents()
    .forEach( integer -> assertTrue(integer % 2 == 0));
```

Prueba de `Observable#error`

Puede asegurarse de que se emita la clase de excepción correcta:

```
Observable<Integer> obs = Observable.error(new Exception("I am a Teapot"));

TestSubscriber<Integer> ts = new TestSubscriber<>();
obs.subscribe(ts);

ts.assertError(Exception.class);
```

También puede asegurarse de que se haya lanzado la excepción exacta:

```
Exception e = new Exception("I am a Teapot");
Observable<Integer> obs = Observable.error(e);

TestSubscriber<Integer> ts = new TestSubscriber<>();
obs.subscribe(ts);

ts.assertError(e);
```

TestScheduler

TestSchedulers le permite controlar el tiempo y la ejecución de los Observables en lugar de tener que hacer esperas ocupadas, unir hilos o cualquier cosa para manipular la hora del sistema. Esto es MUY importante si desea escribir pruebas unitarias que sean predecibles, consistentes y rápidas. Debido a que está manipulando el tiempo, ya no existe la posibilidad de que un subproceso se pierda de hambre, que su prueba falle en una máquina más lenta o que pierda el tiempo de ejecución ocupado esperando un resultado.

TestSchedulers se puede proporcionar a través de la sobrecarga que toma un Programador para cualquier operación de RxJava.

```
TestScheduler testScheduler = new TestScheduler();
TestSubscriber<Integer> subscriber = TestSubscriber.create();
Observable.just(1,2,3)
          .delay(10, TimeUnit.SECONDS, testScheduler)
          .subscribe(subscriber);

try {
    Thread.sleep(TimeUnit.SECONDS.toMillis(11));
} catch (InterruptedException ignored) { }
subscriber.assertValues(1,2,3); // fails

testScheduler.advanceTimeBy(10, TimeUnit.SECONDS);
subscriber.assertValues(1,2,3); // success
```

El TestScheduler es bastante básico. Solo consiste en tres métodos.

```
testScheduler.advanceTimeBy(amount, timeUnit);
testScheduler.advanceTimeTo(when, timeUnit);
```

```
testScheduler.triggerActions();
```

Esto le permite manipular cuando el TestScheduler debe disparar todas las acciones correspondientes a algún tiempo en el futuro.

Si bien el paso del programador funciona, no es así como se usa comúnmente el TestScheduler debido a su ineficacia. Pasar programadores a clases termina proporcionando una gran cantidad de código adicional para obtener poca ganancia. En su lugar, puede conectarse a los Schedulers.io () / computation () / etc de RxJava. Esto se hace con los ganchos de RxJava. Esto le permite definir lo que se devuelve de una llamada desde uno de los métodos de los programadores.

```
public final class TestSchedulers {

    public static TestScheduler test() {
        final TestScheduler testScheduler = new TestScheduler();
        RxJavaHooks.reset();
        RxJavaHooks.setOnComputationScheduler((scheduler) -> {
            return testScheduler;
        });
        RxJavaHooks.setOnIOScheduler((scheduler) -> {
            return testScheduler;
        });
        RxJavaHooks.setOnNewThreadScheduler((scheduler) -> {
            return testScheduler;
        });
        return testScheduler;
    }
}
```

Esta clase permite al usuario obtener el programador de prueba que se conectará para todas las llamadas a los programadores. Una prueba de unidad solo necesitaría tener este programador en su configuración. Se recomienda encarecidamente adquirirlo en la configuración y no como un campo antiguo, ya que su TestScheduler podría intentar activar Acciones desde otra prueba de unidad cuando avance el tiempo. Ahora nuestro ejemplo anterior se convierte en

```
TestScheduler testScheduler = new TestScheduler();
TestSubscriber<Integer> subscriber = TestSubscriber.create();
Observable.just(1,2,3)
          .delay(10, TimeUnit.SECONDS, testScheduler)
          .subscribe(subscriber);
testScheduler.advanceTimeBy(9, TimeUnit.SECONDS);
subscriber.assertValues(); // success (delay hasn't finished)
testScheduler.advanceTimeBy(10, TimeUnit.SECONDS);
subscriber.assertValues(1,2,3); // success (delay has finished)
```

Así es como puede eliminar efectivamente el reloj del sistema de su prueba de unidad (al menos en lo que respecta a RxJava)

Lea Examen de la unidad en línea: https://riptutorial.com/es/rx-java/topic/5207/examen-de-la-unidad

Capítulo 6: Los operadores

Observaciones

Este documento describe el comportamiento básico de un operador.

Examples

Operadores, una introducción

Se puede usar un operador para manipular el flujo de objetos de `Observable` a `Subscriber` .

```
Observable<Integer> integerObservable = Observable.just(1, 2, 3); // creating a simple Integer
observable
Subscriber<String> mSubscriber = new Subscriber<String>() {
    @Override
    public void onCompleted() {
        System.out.println("onCompleted called!");
    }

    @Override
    public void onError(Throwable throwable) {
        System.out.println("onError called");
    }

    @Override
    public void onNext(String string) {
        System.out.println("onNext called with: " + string);
    }
}; // a simple String subscriber

integerObservable
    .map(new Func1<Integer, String>() {
        @Override
        public String call(Integer integer) {
            switch (integer) {
                case 1:
                    return "one";
                case 2:
                    return "two";
                case 3:
                    return "three";
                default:
                    return "zero";
            }
        }
}).subscribe(mSubscriber);
```

La salida sería:

```
onNext called with: one
onNext called with: two
```

```
onNext called with: three
onCompleted called!
```

El operador del `map` cambió el `Integer` observable a una `String` observable, manipulando así el flujo de objetos.

Encadenamiento del operador

Se pueden `chained` varios operadores para obtener transformaciones y manipulaciones más poderosas.

```
integerObservable // emits 1, 2, 3
            .map(i -> i + 10) // adds 10 to each item; emits 11, 12, 13
            .filter(i -> i > 11) // emits items that satisfy condition; 12, 13
            .last() // emits last item in observable; 13
            // unlimited operators can be added ...
            .subscribe(System.out::println); // prints 13
```

Se puede agregar cualquier número de operadores entre el `Observable` y el `Subscriber` .

Operador FlatMap

El operador `flatMap` ayuda a transformar un evento en otro `Observable` (o transformar un evento en cero, uno o más eventos).

Es un operador perfecto cuando desea llamar a otro método que devuelve un `Observable`

```
public Observable<String> perform(int i) {
     // ...
}

Observable.just(1, 2, 3)
          .flatMap(i -> perform(i))
          .subscribe(result -> System.out.println("result ->" + result);
```

`flatMap` serializará las suscripciones de `perform` **pero los** eventos emitidos por `perform` pueden no ordenarse. Por lo que puede recibir eventos emitidos por la última llamada **antes de** realizar eventos desde el primer `perform` llamadas (que puedes usar `concatMap` en su lugar).

Si está creando otro `Observable` en su suscriptor, **debe** utilizar `flatMap` en `flatMap` lugar. La idea principal es: **nunca dejar lo observable.**

Por ejemplo :

```
Observable.just(1, 2, 3)
          .subscribe(i -> perform(i));
```

Se puede reemplazar fácilmente por:

```
Observable.just(1, 2, 3)
          .flatMap(i -> perform(i))
```

```
            .subscribe();
```

Documentación de Reactivex.io: http://reactivex.io/documentation/operators/flatmap.html

Operador de filtro

Se puede utilizar el `filter` operador para filtrar los elementos de la corriente de valores basado en el resultado de un método predicado.

En otras palabras, los elementos que pasan del Observador al Suscriptor se descartarán en función de la Función que pase el `filter` , si la función devuelve `false` para un determinado valor, ese valor se eliminará.

Ejemplo:

```
List<Integer> integers = Arrays.asList(0, 1, 2, 3, 4, 5, 6, 7, 8, 9);

Observable.from(integers)
    .filter(number -> {
        return (number  % 2 == 0);
        // odd numbers will return false, that will cause them to be filtered
    })
    .map(i -> {
        return Math.pow(i, 2); // take each number and multiply by power of 2
    })
    .subscribe(onNext -> {
         System.out.println(onNext); // print out the remaining numbers
    });
```

Este código se imprimirá

```
0.0
4.0
16.0
36.0
64.0
```

operador de mapas

Se puede utilizar el `map` operador para asignar los valores de una corriente a diferentes valores basados en los resultados para cada valor de la función pasada a `map` . El flujo de resultados es una nueva copia y no modificará el flujo de valores proporcionado, el flujo de resultados tendrá la misma longitud del flujo de entrada, pero puede ser de diferentes tipos.

La función pasada a `.map()` , debe devolver un valor.

Ejemplo:

```
List<Integer> numbers = Arrays.asList(1, 2, 3);
Observable.from(numbers)
    .map(number -> {
        return number.toString(); // convert each integer into a string and return it
```

```
    })
    .subscribe(onNext -> {
         System.out.println(onNext); // print out the strings
    });
```

Este código se imprimirá

```
"1"
"2"
"3"
```

En este ejemplo, el Observable aceptó una `List<Integer>` la lista se transformará en una `List<String>` en la tubería y el `.subscribe` emitirá `String`

Operador doOnNext

`doOnNext` operador `doOnNext` llama cada vez que la fuente `Observable` emite un elemento. Se puede utilizar para fines de depuración, aplicando alguna acción al elemento emitido, al registro, etc.

```
Observable.range(1, 3)
    .doOnNext(value -> System.out.println("before transform: " + value))
    .map(value -> value * 2)
    .doOnNext(value -> System.out.println("after transform: " + value))
    .subscribe();
```

En el siguiente ejemplo, nunca se llama a `doOnNext` porque la fuente `Observable` no emite nada porque `Observable.empty()` llama `onCompleted` después de suscribirse.

```
Observable.empty()
    .doOnNext(item -> System.out.println("item: " + item))
    .subscribe();
```

operador de repetición

`repeat` operador de `repeat` permite repetir una secuencia completa desde la fuente `Observable` .

```
Observable.just(1, 2, 3)
    .repeat()
    .subscribe(
        next -> System.out.println("next: " + next),
        error -> System.out.println("error: " + error),
        () -> System.out.println("complete")
    );
```

Salida del ejemplo anterior.

```
next: 1
next: 2
next: 3
next: 1
next: 2
```

```
next: 3
```

Esta secuencia se repite un número infinito de veces y nunca se completa.

Para repetir un número finito de secuencias, simplemente pase un entero como argumento para `repeat` operador.

```
Observable.just(1, 2, 3)
    // Repeat three times and complete
    .repeat(3)
    .subscribe(
        next -> System.out.println("next: " + next),
        error -> System.out.println("error: " + error),
        () -> System.out.println("complete")
    );
```

Este ejemplo imprime

```
next: 1
next: 2
next: 3
next: 1
next: 2
next: 3
next: 1
next: 2
next: 3
complete
```

Es muy importante comprender que el operador `repeat` vuelve a suscribirse a la fuente `Observable` cuando se completa la secuencia `Observable` fuente. Reescribamos el ejemplo anterior usando `Observable.create` .

```
Observable.<Integer>create(subscriber -> {

    //Same as Observable.just(1, 2, 3) but with output message
    System.out.println("Subscribed");
    subscriber.onNext(1);
    subscriber.onNext(2);
    subscriber.onNext(3);
    subscriber.onCompleted();
})
        .repeat(3)
        .subscribe(
                next -> System.out.println("next: " + next),
                error -> System.out.println("error: " + error),
                () -> System.out.println("complete")
        );
```

Este ejemplo imprime

```
Subscribed
next: 1
next: 2
next: 3
```

```
Subscribed
next: 1
next: 2
next: 3
Subscribed
next: 1
next: 2
next: 3
complete
```

Cuando se utiliza el operador de encadenamiento es importante saber que `repeat` operador repite **toda la secuencia** anterior en lugar de operador.

```
Observable.<Integer>create(subscriber -> {
    System.out.println("Subscribed");
    subscriber.onNext(1);
    subscriber.onNext(2);
    subscriber.onNext(3);
    subscriber.onCompleted();
})
        .map(value -> value * 2) //First chain operator
        .map(value -> "modified " + value) //Second chain operator
        .repeat(3)
        .subscribe(
                next -> System.out.println("next: " + next),
                error -> System.out.println("error: " + error),
                () -> System.out.println("complete")
        );
```

Este ejemplo imprime

```
Subscribed
next: modified 2
next: modified 4
next: modified 6
Subscribed
next: modified 2
next: modified 4
next: modified 6
Subscribed
next: modified 2
next: modified 4
next: modified 6
complete
```

Este ejemplo muestra que `repeat` operador repite toda la secuencia resubscribing a `Observable` en lugar de repetir último `map` operador y no importa en que lugar en la secuencia `repeat` operador utilizado.

Esta secuencia

```
Observable.<Integer>create(subscriber -> {
        //...
    })
    .map(value -> value * 2) //First chain operator
    .map(value -> "modified " + value) //Second chain operator
```

```
    .repeat(3)
    .subscribe(
        /*....*/
    );
```

es igual a esta secuencia

```
Observable.<Integer>create(subscriber -> {
        //...
    })
    .repeat(3)
    .map(value -> value * 2) //First chain operator
    .map(value -> "modified " + value) //Second chain operator
    .subscribe(
        /*....*/
    );
```

Lea Los operadores en línea: https://riptutorial.com/es/rx-java/topic/2316/los-operadores

Capítulo 7: Observable

Examples

Crear un observable

Hay varias formas de crear un Observable en RxJava. La forma más poderosa es usar el método `Observable.create` . Pero también es la forma más **complicada** . Así que debes **evitar usarlo** , tanto como sea posible.

Emitiendo un valor de salida.

Si ya tiene un valor, puede usar `Observable.just` para emitir su valor.

```
Observable.just("Hello World").subscribe(System.out::println);
```

Emitiendo un valor que debe ser calculado.

Si desea emitir un valor que aún no está calculado, o que puede tardar mucho tiempo en calcularse, puede usar `Observable.fromCallable` para emitir su próximo valor.

```
Observable.fromCallable(() -> longComputation()).subscribe(System.out::println);
```

`longComputation()` solo se llamará cuando te suscribas a tu `Observable` . De esta manera, el cálculo será *perezoso* .

Forma alternativa de emitir un valor que debe ser computado.

`Observable.defer` creado un `Observable` como `Observable.fromCallable` pero se usa cuando se necesita devolver un `Observable` lugar de un valor. Es útil cuando desea administrar los errores en su llamada.

```
Observable.defer(() -> {
          try {
                return Observable.just(longComputation());
          } catch(SpecificException e) {
                return Observable.error(e);
          }).subscribe(System.out::println);
```

Observables fríos y calientes

Los observables se clasifican ampliamente en `Hot` o `Cold` , dependiendo de su comportamiento de emisión.
Un `Cold Observable` es uno que comienza a emitir a pedido (suscripción), mientras que un `Hot Observable` es uno que emite independientemente de las suscripciones.

Frio observable

```
/* Demonstration of a Cold Observable */
Observable<Long> cold = Observable.interval(500, TimeUnit.MILLISECONDS); // emits a long every
500 milli seconds
cold.subscribe(l -> System.out.println("sub1, " + l)); // subscriber1
Thread.sleep(1000); // interval between the two subscribes
cold.subscribe(l -> System.out.println("sub2, " + l)); // subscriber2
```

La salida del código anterior parece (puede variar):

```
sub1, 0    -> subscriber1 starts
sub1, 1
sub1, 2
sub2, 0    -> subscriber2 starts
sub1, 3
sub2, 1
sub1, 4
sub2, 2
```

Tenga en cuenta que aunque `sub2` comienza tarde, recibe valores desde el principio. Para concluir, un `Cold Observable` solo emite elementos cuando se solicita. Múltiples solicitudes inician múltiples tuberías.

Caliente observable

Nota: los observables en caliente emiten valores independientes de las suscripciones individuales. Tienen su propia línea de tiempo y los eventos ocurren ya sea que alguien esté escuchando o no.

Un `Cold Observale` se puede convertir en un `Hot Observable` con una `publish` simple.

```
Observable.interval(500, TimeUnit.MILLISECONDS)
    .publish(); // publish converts cold to hot
```

`publish` devuelve un `ConnectableObservable` que agrega funcionalidades para *conectarse* y *desconectarse* de lo observable.

```
ConnectableObservable<Long> hot = Observable
                                    .interval(500, TimeUnit.MILLISECONDS)
                                    .publish(); // returns ConnectableObservable
hot.connect(); // connect to subscribe

hot.subscribe(l -> System.out.println("sub1, " + l));
Thread.sleep(1000);
```

```
hot.subscribe(l -> System.out.println("sub2, " + l));
```

Los rendimientos anteriores:

```
sub1, 0  -> subscriber1 starts
sub1, 1
sub1, 2
sub2, 2  -> subscriber2 starts
sub1, 3
sub2, 3
```

Tenga en cuenta que aunque `sub2` comienza a observar tarde, está sincronizado con `sub1` . ¡Desconectar es un poco más complicado! La desconexión ocurre en la `Subscription` y no en el `Observable` .

```
ConnectableObservable<Long> hot = Observable
                                    .interval(500, TimeUnit.MILLISECONDS)
                                    .publish(); // same as above
Subscription subscription = hot.connect(); // connect returns a subscription object, which we
store for further use

hot.subscribe(l -> System.out.println("sub1, " + l));
Thread.sleep(1000);
hot.subscribe(l -> System.out.println("sub2, " + l));
Thread.sleep(1000);
subscription.unsubscribe(); // disconnect, or unsubscribe from subscription

System.out.println("reconnecting");
/* reconnect and redo */
subscription = hot.connect();
hot.subscribe(l -> System.out.println("sub1, " + l));
Thread.sleep(1000);
hot.subscribe(l -> System.out.println("sub2, " + l));
Thread.sleep(1000);
subscription.unsubscribe();
```

Lo anterior produce:

```
sub1, 0   -> subscriber1 starts
sub1, 1
sub1, 2
sub2, 2   -> subscriber2 starts
sub1, 3
sub2, 3
reconnecting  -> reconnect after unsubscribe
sub1, 0
...
```

Al desconectarse, el `Observable` esencialmente "termina" y se reinicia cuando se agrega una nueva suscripción.

`Hot Observable` se puede utilizar para crear un `EventBus` . Tales EventBuses son generalmente ligeros y súper rápidos. El único inconveniente de un RxBus es que todos los eventos deben implementarse manualmente y pasarse al bus.

Lea Observable en línea: https://riptutorial.com/es/rx-java/topic/1418/observable

Capítulo 8: Programadores

Examples

Ejemplos básicos

Los programadores son una abstracción de RxJava sobre la unidad de procesamiento. Un programador puede respaldar un planificador, pero puede implementar su propia implementación de planificador.

Un `Scheduler` debe cumplir con este requisito:

- Debería procesar la tarea no demorada secuencialmente (orden FIFO)
- La tarea puede retrasarse

Un `Scheduler` puede usarse como parámetro en algunos operadores (ejemplo: `delay`), o puede usarse con el método `subscribeOn` / `observeOn` .

Con algún operador, el `Scheduler` se utilizará para procesar la tarea del operador específico. Por ejemplo, el `delay` programará una tarea retrasada que emitirá el siguiente valor. Este es un `Scheduler` que lo retendrá y ejecutará más tarde.

El `subscribeOn` se puede utilizar una vez por `Observable` . Se definirá en qué `Scheduler` se ejecutará el código de la suscripción.

El `observeOn` se puede utilizar varias veces por `Observable` . `observeOn` en qué `Scheduler` se utilizará para ejecutar todas las tareas definidas **después** del método de `observeOn` . `observeOn` le ayudará a realizar el salto de hilo.

suscribirse en el programador específico

```
// this lambda will be executed in the `Schedulers.io()`
Observable.fromCallable(() -> Thread.currentThread().getName())
          .subscribeOn(Schedulers.io())
          .subscribe(System.out::println);
```

observar con programador específico

```
Observable.fromCallable(() -> "Thread -> " + Thread.currentThread().getName())
         // next tasks will be executed in the io scheduler
         .observeOn(Schedulers.io())
         .map(str -> str + " -> " + Thread.currentThread().getName())
          // next tasks will be executed in the computation scheduler
         .observeOn(Schedulers.computation())
         .map(str -> str + " -> " + Thread.currentThread().getName())
         // next tasks will be executed in the io scheduler
         .observeOn(Schedulers.newThread())
         .subscribe(str -> System.out.println(str + " -> " +
Thread.currentThread().getName()));
```

Especificar un programador específico con un operador

Algunos operadores pueden tomar un `Scheduler` como parámetro.

```
Observable.just(1)
          // the onNext method of the delay operator will be executed in a new thread
          .delay(1, TimeUnit.SECONDS, Schedulers.newThread())
          .subscribe(System.out::println);
```

Publicar Para Suscriptor:

```
TestScheduler testScheduler = Schedulers.test();
EventBus sut = new DefaultEventBus(testScheduler);
TestSubscriber<Event> subscriber = new TestSubscriber<Event>();
sut.get().subscribe(subscriber);
sut.publish(event);
testScheduler.advanceTimeBy(1, TimeUnit.SECONDS);
```

Grupo de subprocesos:

```
this.poolName = schedulerFig.getIoSchedulerName();
final int poolSize = schedulerFig.getMaxIoThreads();
final BlockingQueue<Runnable> queue = new ArrayBlockingQueue<Runnable>(poolSize);
final MaxSizeThreadPool threadPool = new MaxSizeThreadPool( queue, poolSize );
this.scheduler = Schedulers.from(threadPool);
```

Web Socket observable:

```
final Subscription subscribe = socket.webSocketObservable()
        .subscribeOn(Schedulers.io())
        .doOnNext(new Action1<RxEvent>() {
            @Override
            public void call(RxEvent rxEvent) {
                System.out.println("Event: " + rxEvent);
            }
        })
        .subscribe();
```

Lea Programadores en línea: https://riptutorial.com/es/rx-java/topic/2321/programadores

Capítulo 9: Retrofit y RxJava

Examples

Configurar Retrofit y RxJava

Retrofit2 viene con soporte para múltiples mecanismos de ejecución conectables, uno de ellos es RxJava.

Para usar la actualización con RxJava, primero debe agregar el adaptador RxJava de actualización a su proyecto:

```
compile 'com.squareup.retrofit2:adapter-rxjava:2.1.0'
```

luego necesita agregar el adaptador al crear su instancia de actualización:

```
Retrofit retrofit = new Retrofit.Builder()
    .baseUrl("https://api.example.com")
    .addCallAdapterFactory(RxJavaCallAdapterFactory.create())
    .build();
```

En su interfaz, cuando define la API, el tipo de retorno debe ser `Observable` por ejemplo:

```
public interface GitHubService {
  @GET("users/{user}/repos")
  Observable<List<Repo>> listRepos(@Path("user") String user);
}
```

También puedes usar `Single` lugar de `Observable` .

Haciendo solicitudes seriales

RxJava es útil cuando se realiza una solicitud en serie. Si desea usar el resultado de una solicitud para hacer otra, puede usar el operador `flatMap` :

```
api.getRepo(repoId).flatMap(repo -> api.getUser(repo.getOwnerId())
    .subscribe(/*do something with the result*/);
```

Haciendo peticiones paralelas

Puede utilizar el operador `zip` para realizar una solicitud en paralelo y combinar los resultados, por ejemplo:

```
Observable.zip(api.getRepo(repoId1), api.getRepo(repoId2), (repo1, repo2) ->
    {
        //here you can combine the results
    }).subscribe(/*do something with the result*/);
```

Lea Retrofit y RxJava en línea: https://riptutorial.com/es/rx-java/topic/2950/retrofit-y-rxjava

Capítulo 10: RxJava2 Flowable y Suscriptor

Introducción

Este tema muestra ejemplos y documentación con respecto a los conceptos reactivos de Flowable y Subscriber que se introdujeron en rxjava versión 2.

Observaciones

el ejemplo necesita rxjava2 como una dependencia, las coordenadas de Maven para la versión utilizada son:

```
    <dependency>
        <groupId>io.reactivex.rxjava2</groupId>
        <artifactId>rxjava</artifactId>
        <version>2.0.8</version>
    </dependency>
```

Examples

Ejemplo de consumidor productor con soporte de contrapresión en el productor.

El `TestProducer` de este ejemplo produce objetos `Integer` en un rango dado y los empuja a su `Subscriber` . Extiende la clase `Flowable<Integer>` . Para un nuevo suscriptor, crea un objeto de `Subscription` cuyo método de `request(long)` se utiliza para crear y publicar los valores de enteros.

Es importante para la `Subscription` que se transmite al `subscriber` que el método `request()` que llama `onNext()` en el suscriptor se puede llamar recursivamente desde esta llamada `onNext()` . Para evitar un desbordamiento de pila, la implementación que se muestra utiliza el contador `outStandingRequests` y el indicador `isProducing` .

```
class TestProducer extends Flowable<Integer> {
    static final Logger logger = LoggerFactory.getLogger(TestProducer.class);
    final int from, to;

    public TestProducer(int from, int to) {
        this.from = from;
        this.to = to;
    }

    @Override
    protected void subscribeActual(Subscriber<? super Integer> subscriber) {
        subscriber.onSubscribe(new Subscription() {

            /** the next value. */
            public int next = from;
            /** cancellation flag. */
```

```
            private volatile boolean cancelled = false;
            private volatile boolean isProducing = false;
            private AtomicLong outStandingRequests = new AtomicLong(0);

            @Override
            public void request(long n) {
                if (!cancelled) {

                    outStandingRequests.addAndGet(n);

                    // check if already fulfilling request to prevent call  between request()
an subscriber .onNext()
                    if (isProducing) {
                        return;
                    }

                    // start producing
                    isProducing = true;

                    while (outStandingRequests.get() > 0) {
                        if (next > to) {
                            logger.info("producer finished");
                            subscriber.onComplete();
                            break;
                        }
                        subscriber.onNext(next++);
                        outStandingRequests.decrementAndGet();
                    }
                    isProducing = false;
                }
            }

            @Override
            public void cancel() {
                cancelled = true;
            }
        });
    }
}
```

El Consumidor en este ejemplo extiende `DefaultSubscriber<Integer>` y en el inicio y después de consumir un Integer solicita el siguiente. Al consumir los valores de Integer, hay un pequeño retraso, por lo que se creará la contrapresión para el productor.

```
class TestConsumer extends DefaultSubscriber<Integer> {

    private static final Logger logger = LoggerFactory.getLogger(TestConsumer.class);

    @Override
    protected void onStart() {
        request(1);
    }

    @Override
    public void onNext(Integer i) {
        logger.info("consuming {}", i);
        if (0 == (i % 5)) {
            try {
                Thread.sleep(500);
```

```
            } catch (InterruptedException ignored) {
                // can be ignored, just used for pausing
            }
        }
        request(1);
    }

    @Override
    public void onError(Throwable throwable) {
        logger.error("error received", throwable);
    }

    @Override
    public void onComplete() {
        logger.info("consumer finished");
    }
}
```

En el siguiente método principal de una clase de prueba, el productor y el consumidor se crean y conectan:

```
public static void main(String[] args) {
    try {
        final TestProducer testProducer = new TestProducer(1, 1_000);
        final TestConsumer testConsumer = new TestConsumer();

        testProducer
                .subscribeOn(Schedulers.computation())
                .observeOn(Schedulers.single())
                .blockingSubscribe(testConsumer);

    } catch (Throwable t) {
        t.printStackTrace();
    }
}
```

Cuando se ejecuta el ejemplo, el archivo de registro muestra que el consumidor se ejecuta continuamente, mientras que el productor solo se activa cuando se debe rellenar el búfer interno Flowable de rxjava2.

Lea RxJava2 Flowable y Suscriptor en línea: https://riptutorial.com/es/rx-java/topic/9810/rxjava2-flowable-y-suscriptor

Creditos

S. No	Capítulos	Contributors
1	Empezando con rx-java	Buttink, Community, dimsuz, Dmitry Avtonomov, Hans Wurst, hello_world, Omar Al Halabi, Saulius Next, Sneh Pandya, svarog, Tom
2	Android con RxJava	akarnokd, Athafoud, Daniele Segato, Eugen Martynov, Geng Jiawen, Sneh Pandya
3	Asignaturas	hello_world, mavHarsha
4	Contrapresion	akarnokd, Bartek Lipinski, Chris A, Cristian, dwursteisen, Niklas, Sebas LG
5	Examen de la unidad	Buttink, Sir Celsius
6	Los operadores	dwursteisen, hello_world, svarog, Vadeg
7	Observable	Aki K, dwursteisen, hello_world, JonesV
8	Programadores	dwursteisen, Gal Dreiman
9	Retrofit y RxJava	LordRaydenMK
10	RxJava2 Flowable y Suscriptor	P.J.Meisch