eBook Gratuit

APPRENEZ rx-java

eBook gratuit non affilié créé à partir des **contributeurs de Stack Overflow.**

#rx-java

Table des matières

À propos

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

Chapitre 1: Démarrer avec rx-java

Remarques

Cette section fournit un aperçu de base et une introduction superficielle à rx-java.

RxJava est une implémentation Java de Reactive Extensions : une bibliothèque pour composer des programmes asynchrones et basés sur des événements en utilisant des séquences observables.

En savoir plus sur RxJava sur le Wiki Home .

Versions

Version	Statut	Dernière version stable	Date de sortie
1 fois	Stable	1.3.0	2017-05-05
2.x	Stable	2.1.1	2017-06-21

Examples

Installation ou configuration

rx-java mis en place

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. Lierre

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

4. Instantanés de JFrog

```
repositories {
```

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

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

5. Si vous avez besoin de télécharger les fichiers JAR au lieu d'utiliser un système de génération, créez un fichier Maven `pom` comme celui-ci avec la version souhaitée:

```
<?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>
```

Ensuite, exécutez:

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

Cette commande télécharge `rxjava-*.jar` et ses dépendances dans `./target/dependency/`.

Vous avez besoin de Java 6 ou plus tard.

Bonjour le monde!

Ce qui suit imprime le message `Hello, World!` consoler

```
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);
       }

    });
}
```

Ou en utilisant la notation Java 8 lambda

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

Une introduction à RxJava

Les concepts de base de RxJava sont ses `Observables` et ses `Subscribers` . Un objet `Observable` émet des objets, tandis qu'un `Subscriber` consomme.

Observable

`Observable` est une classe qui implémente le modèle de conception réactif. Ces observables fournissent des méthodes permettant aux consommateurs de s'abonner aux modifications apportées aux événements. Les changements d'événement sont déclenchés par l'observable. Il n'y a pas de restriction quant au nombre d'abonnés qu'un `Observable` peut avoir ou au nombre d'objets qu'une `Observable` peut émettre.

Prends pour exemple:

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

Ici, un objet observable appelé `integerObservable` et `stringObservable` sont créés à partir de la méthode de fabrication `just` fourni par la bibliothèque Rx. Notez que `Observable` est générique et peut donc émettre n'importe quel objet.

Abonné

Un `Subscriber` est le consommateur. Un `Subscriber` ne peut souscrire ***qu'à une seule*** observable. `Observable` appelle les `onNext()` , `onCompleted()` et `onError()` de l' `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);
        }
    };
```

Notez que l' `Subscriber` est également générique et peut supporter n'importe quel objet. Un `Subscriber` doit souscrire à l'observable en appelant la méthode `subscribe` sur l'observable.

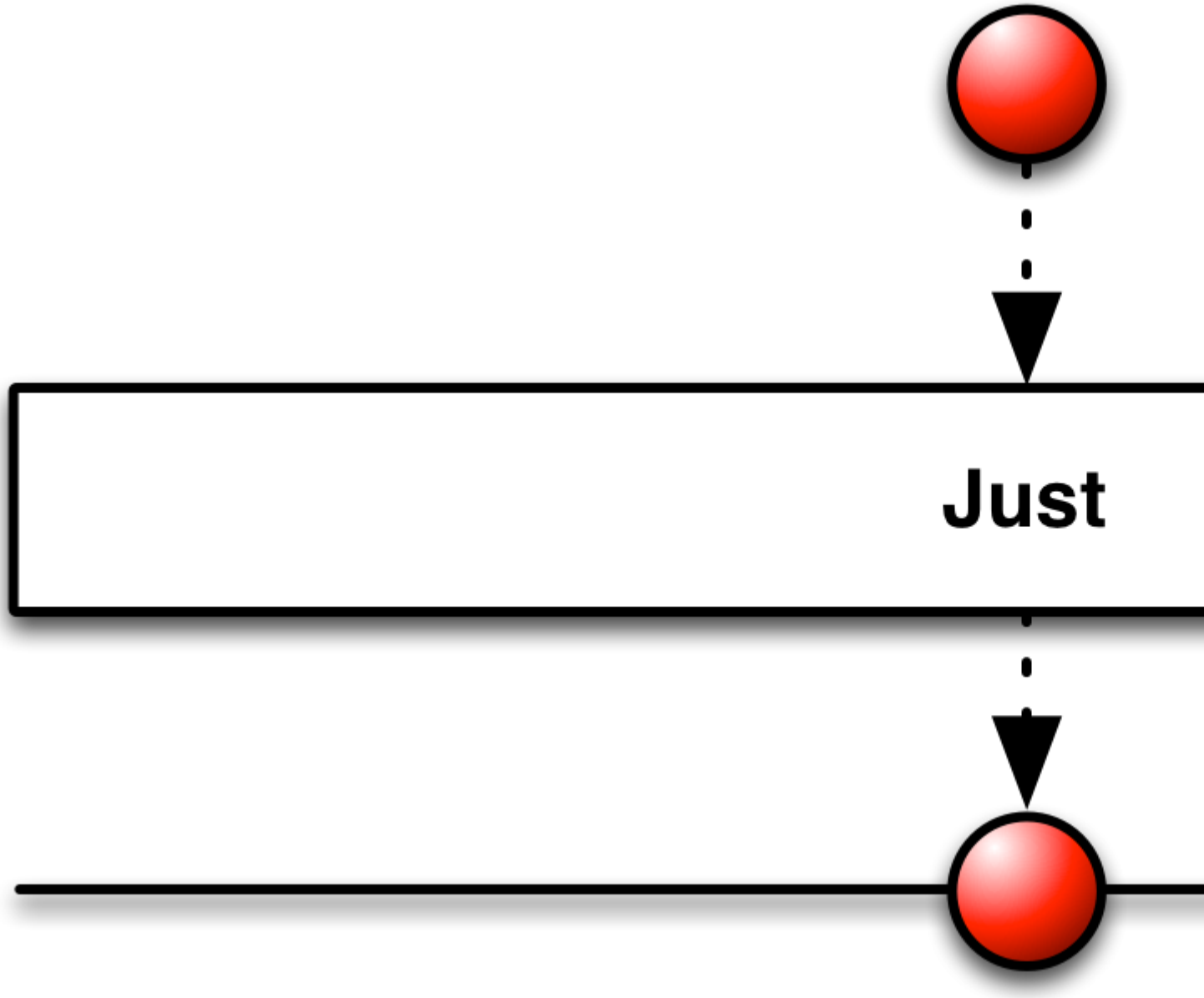
```
integerObservable.subscribe(mSubscriber);
```

Le ci-dessus, lorsqu'il est exécuté, produira la sortie suivante:

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

Comprendre les diagrammes de marbre

Un observable peut être considéré comme un simple flux d'événements. Lorsque vous définissez une observable, vous disposez de trois écouteurs: onNext, onComplete et onError. onNext sera appelé chaque fois que l'observable acquiert une nouvelle valeur. onComplete sera appelé si le parent Observable indique qu'il a fini de produire d'autres valeurs. onError est appelée si une exception est lancée à tout moment pendant l'exécution de la chaîne Observable. Pour afficher les opérateurs dans Rx, le diagramme de marbre est utilisé pour afficher ce qui se passe avec une opération particulière. Vous trouverez ci-dessous un exemple d'un simple opérateur observable "Just".



Les diagrammes de marbre ont un bloc horizontal qui représente l'opération effectuée, une barre verticale représentant l'événement terminé, un X représentant une erreur et toute autre forme représentant une valeur. Dans cet esprit, nous pouvons voir que "Just" va simplement prendre notre valeur et faire un onNext, puis finir avec onComplete. Il y a beaucoup plus d'opérations que juste "juste". Vous pouvez voir toutes les opérations faisant partie du projet ReactiveX et leurs implémentations dans RxJava sur le site ReativeX . Il existe également des diagrammes de marbre interactifs via le site RxMarbles .

Lire Démarrer avec rx-java en ligne: https://riptutorial.com/fr/rx-java/topic/974/demarrer-avec-rx-java

Chapitre 2: Android avec RxJava

Remarques

RxAndroid était une bibliothèque avec de nombreuses fonctionnalités. Il a été divisé en plusieurs bibliothèques différentes allant de la version 0.25.0 à 1.x.

Une liste des bibliothèques qui implémentent les fonctionnalités disponibles avant la version 1.0 est conservée ici .

Examples

RxAndroid - AndroidSchedulers

Ceci est littéralement la seule chose dont vous avez besoin pour commencer à utiliser RxJava sur Android.

Incluez RxJava et RxAndroid dans vos dépendances graduelles:

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

L'ajout principal de RxAndroid à RxJava est un planificateur pour le thread principal Android ou l'interface utilisateur.

Dans votre code:

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

Ou vous pouvez créer un planificateur pour un `Looper` personnalisé:

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

Pour la plupart des choses, vous pouvez vous référer à la documentation RxJava standard.

Composants RxLifecycle

La bibliothèque RxLifecycle facilite la liaison des souscriptions observables aux activités Android et au cycle de vie des fragments.

Gardez à l'esprit que l'oubli de la désinscription d'un Observable peut provoquer des fuites de mémoire et empêcher que votre activité / fragment reste actif après sa destruction par le système.

Ajoutez la bibliothèque aux dépendances:

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

`Rx*` classes `Rx*` :

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

Vous êtes tous ensemble, lorsque vous vous abonnez à un observable, vous pouvez maintenant:

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

Si vous l'exécutez dans la méthode `onCreate()` de l'activité, celle-ci sera automatiquement désabonnée de la `onDestroy()` .

Même chose pour:

- `onStart()` -> `onStop()`
- `onResume()` -> `onPause()`
- `onAttach()` -> `onDetach()` *(fragment uniquement)*
- `onViewCreated()` -> `onDestroyView()` *(fragment uniquement)*

Comme alternative, vous pouvez spécifier l'événement lorsque vous souhaitez que la désinscription se produise:

A partir d'une activité:

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

D'un fragment:

```
someObservable
    .compose(bindUntilEvent(FragmentEvent.DESTROY_VIEW))
```

```
    .subscribe();
```

Vous pouvez également obtenir le cycle de vie observable en utilisant le `lifecycle()` la méthode `lifecycle()` pour écouter directement les événements du cycle de vie.

RxLifecycle peut également être utilisé directement en lui transmettant le cycle de vie observable:

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

Si vous avez besoin de gérer `Single` ou `Completable` vous pouvez le faire simplement en ajoutant respectivement `forSingle()` ou `forCompletable` après la méthode bind:

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

Il peut également être utilisé avec la bibliothèque Navi .

Rxpermissions

Cette bibliothèque permet l'utilisation de RxJava avec le nouveau modèle d'autorisation Android M.

Ajoutez la bibliothèque aux dépendances:

Rxjava

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

Rxjava2

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

Usage

Exemple (avec Retrolambda pour des raisons de concision, mais pas obligatoire):

```
// 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
        }
    });
```

En savoir plus: https://github.com/tbruyelle/RxPermissions .

Lire Android avec RxJava en ligne: https://riptutorial.com/fr/rx-java/topic/7125/android-avec-rxjava

Chapitre 3: Contrepression

Examples

introduction

La contre - **pression** se produit lorsque, dans un pipeline de traitement `Observable` , certaines étapes asynchrones ne peuvent pas traiter les valeurs assez rapidement et nécessitent un moyen de ralentir le producteur en amont.

Le cas classique du besoin de contre-pression est lorsque le producteur est une source chaude:

```
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);
```

Dans cet exemple, le thread principal produira 1 million d'éléments pour un consommateur final qui le traite sur un thread d'arrière-plan. Il est probable que la méthode `compute(int)` prenne un certain temps, mais la surcharge de la chaîne d'opérateurs `Observable` peut également augmenter le temps nécessaire au traitement des éléments. Cependant, le thread produisant avec la boucle for ne peut pas le savoir et continue à `onNext` .

En interne, les opérateurs asynchrones ont des tampons pour contenir de tels éléments jusqu'à ce qu'ils puissent être traités. Dans le Rx.NET classique et au début du RxJava, ces tampons étaient sans limites, ce qui signifiait qu'ils contiendraient probablement presque tous les 1 million d'éléments de l'exemple. Le problème commence lorsque, par exemple, 1 milliard d'éléments ou 1 million de séquences apparaissent 1000 fois dans un programme, conduisant à `OutOfMemoryError` et généralement à des ralentissements dus à une surcharge excessive du GC.

Semblable à la façon dont la gestion des erreurs est devenue un citoyen de première classe et a reçu des opérateurs (via les opérateurs `onErrorXXX`), la contre-pression est une autre propriété des flux de données à laquelle le programmeur doit penser (via les opérateurs `onBackpressureXXX`).

Au-delà du `PublishSubject` ci-dessus, d'autres opérateurs ne prennent pas en charge la contre-pression, principalement pour des raisons fonctionnelles. Par exemple, l' `interval` l'opérateur émet des valeurs périodiquement, ce qui entraîne une modification de la période par rapport à une horloge murale.

Dans RxJava moderne, la plupart des opérateurs asynchrones ont maintenant un tampon interne limité, comme `observeOn` ci-dessus et toute tentative de débordement de ce tampon mettra fin à la

séquence entière avec `MissingBackpressureException` . La documentation de chaque opérateur a une description de son comportement de contre-pression.

Cependant, la contre-pression est présente de manière plus subtile dans les séquences froides régulières (qui ne doivent pas et ne doivent pas générer une `MissingBackpressureException`). Si le premier exemple est réécrit:

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

Thread.sleep(10_000);
```

Il n'y a pas d'erreur et tout fonctionne sans problème avec une faible utilisation de la mémoire. La raison en est que de nombreux opérateurs sources peuvent "générer" des valeurs à la demande et ainsi l'opérateur `observeOn` peut indiquer que la `range` génère autant de valeurs que le tampon `observeOn` peut contenir en une fois sans dépassement.

Cette négociation est basée sur le concept informatique de co-routines (je vous appelle, vous m'appelez). L'opérateur `range` envoie un rappel, sous la forme d'une mise en œuvre du `Producer` interface, à l' `observeOn` en appelant son (intérieur `Subscriber` de) `setProducer` . En retour, le `observeOn` appelle `Producer.request(n)` avec une valeur pour indiquer la `range` qu'il est autorisé à produire (c'est-à-dire, `onNext` it) autant d'éléments **supplémentaires** . Il est alors de la `observeOn` de `observeOn` d'appeler la méthode de `request` au bon moment et avec la bonne valeur pour que les données continuent à circuler mais pas à saturation.

L'expression de la contre-pression chez les consommateurs finaux est rarement nécessaire (car ils sont synchrones par rapport à leur amont immédiat et la contre-pression se produit naturellement par le blocage de la pile d'appels), mais il peut être plus facile de comprendre son fonctionnement:

```
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!");
    }
```

```
});
```

Ici, l'implémentation `onStart` indique la `range` pour produire sa première valeur, qui est ensuite reçue dans `onNext` . Une fois que le `compute(int)` terminé, l'autre valeur est alors demandée à partir de la `range` . Dans une implémentation naïve de la `range` , un tel appel appelle récursivement `onNext` , conduisant à `StackOverflowError` ce qui est évidemment indésirable.

Pour éviter cela, les opérateurs utilisent une logique dite de trampoline qui empêche de tels appels réentrants. En termes de `range` , il se souviendra qu'il y avait un `request(1)` alors qu'il appelait `onNext()` et qu'une fois que `onNext()` reviendrait, il ferait un autre tour et appellerait `onNext()` avec la valeur entière suivante. Par conséquent, si les deux sont échangés, l'exemple fonctionne toujours de la même façon:

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

    compute(v);
}
```

Cependant, ce n'est pas le cas pour `onStart` . Bien que l'infrastructure `Observable` garantisse qu'elle sera appelée au plus une fois sur chaque `Subscriber` , l'appel à la `request(1)` peut déclencher immédiatement l'émission d'un élément. Si l'on dispose d'une logique d'initialisation après l'appel à la `request(1)` nécessaire à `onNext` , vous pouvez vous retrouver avec des exceptions:

```
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
});
```

Dans ce cas synchrone, une `NullPointerException` sera lancée immédiatement lors de l'exécution de `onStart` . Un bug plus subtil se produit si l'appel à la `request(1)` déclenche un appel asynchrone à `onNext` sur un autre thread et lit le `name` dans les courses `onNext` écrivant dans `onStart request` publication `onStart` .

Par conséquent, il faut faire toutes les initialisations de champs dans `onStart` ou même avant cela

et appeler `request()` last. Les implémentations de `request()` dans les opérateurs assurent une relation correcte avant ou après (ou en d'autres termes, une libération de mémoire ou une clôture complète) si nécessaire.

Les opérateurs onBackpressureXXX

La plupart des développeurs rencontrent une contre-pression lorsque leur application échoue avec `MissingBackpressureException` exception `MissingBackpressureException` et que l'exception pointe généralement vers l'opérateur `observeOn` . La cause réelle est généralement l'utilisation non rétroactive de `PublishSubject` , `timer()` ou `interval()` ou des opérateurs personnalisés créés via `create()` .

Il existe plusieurs manières de faire face à de telles situations.

Augmenter la taille des tampons

Parfois, de tels débordements se produisent en raison de sources explosives. Soudain, l'utilisateur tape trop rapidement sur l'écran et `observeOn` le tampon interne de 16 éléments par défaut d' `observeOn` sur les débordements d'Android.

La plupart des opérateurs sensibles à la contre-pression dans les versions récentes de RxJava permettent désormais aux programmeurs de spécifier la taille de leurs tampons internes. Les paramètres pertinents sont généralement appelés `bufferSize` , `prefetch` ou `capacityHint` . Compte tenu de l'exemple débordant dans l'introduction, nous pouvons simplement augmenter la taille du tampon d' `observeOn` pour avoir assez de place pour toutes les valeurs.

```
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);
}
```

Notez toutefois que généralement, cela peut être seulement une solution temporaire, car le dépassement de capacité peut toujours se produire si la source surproduit la taille de la mémoire tampon prévue. Dans ce cas, on peut utiliser l'un des opérateurs suivants.

Valeurs par lots / sauts avec les opérateurs standard

`MissingBackpressureException` les données source peuvent être traitées plus efficacement par lots, il est possible de réduire le risque d' `MissingBackpressureException` en utilisant l'un des opérateurs de traitement par lots standard (par taille et / ou par heure).

```
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 certaines des valeurs peuvent être ignorées en toute sécurité, on peut utiliser l'échantillonnage (avec le temps ou un autre Observable) et les opérateurs de limitation (`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);
}
```

Notez que ces opérateurs ne réduisent que le taux de réception de la valeur en aval et peuvent donc conduire à une `MissingBackpressureException` .

onBackpressureBuffer ()

Cet opérateur sous sa forme sans paramètre réintroduit un tampon non limité entre la source en amont et l'opérateur en aval. Être sans limite signifie que tant que la JVM ne manque pas de mémoire, elle peut gérer presque n'importe quelle quantité provenant d'une source de données en rafale.

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

Dans cet exemple, la `observeOn` va avec une taille de tampon très faible, mais il n'y a aucune `MissingBackpressureException` car `onBackpressureBuffer` absorbe toutes les 1 million de valeurs et en `observeOn` petits lots à `observeOn` .

Notez cependant que `onBackpressureBuffer` consomme sa source de manière illimitée, c'est-à-dire sans y appliquer aucune contre-pression. Cela a pour conséquence que même une source de support de contre-pression telle que la `range` sera complètement réalisée.

Il y a 4 surcharges supplémentaires de `onBackpressureBuffer`

onBackpressureBuffer (capacité int)

C'est une version limitée qui signale `BufferOverflowError` au cas où son tampon atteindrait la capacité donnée.

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

La pertinence de cet opérateur diminue à mesure que de plus en plus d'opérateurs permettent de définir leurs tailles de tampons. Pour le reste, cela donne l'occasion d'étendre leur tampon interne en ayant un nombre plus grand avec `onBackpressureBuffer` que leur valeur par défaut.

onBackpressureBuffer (capacité int, Action0 onOverflow)

Cette surcharge appelle une action (partagée) en cas de dépassement de capacité. Son utilité est plutôt limitée car il n'y a pas d'autres informations sur le débordement que la pile d'appels actuelle.

onBackpressureBuffer (capacité int, action0 onOverflow, stratégie BackpressureOverflow.Strategy)

Cette surcharge est en fait plus utile car elle permet de définir ce qu'il faut faire au cas où la capacité serait atteinte. Le `BackpressureOverflow.Strategy` est une interface, mais la classe `BackpressureOverflow` offre 4 champs statiques avec des implémentations représentant des actions typiques:

- `ON_OVERFLOW_ERROR` : il s'agit du comportement par défaut des deux surcharges précédentes, signalant une `BufferOverflowException`
- `ON_OVERFLOW_DEFAULT` : actuellement, il est identique à `ON_OVERFLOW_ERROR`
- `ON_OVERFLOW_DROP_LATEST` : si un débordement se produisait, la valeur actuelle serait simplement ignorée et seules les anciennes valeurs seraient délivrées une fois les requêtes en aval.
- `ON_OVERFLOW_DROP_OLDEST` : supprime l'élément le plus ancien du tampon et lui ajoute la valeur actuelle.

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

Notez que les deux dernières stratégies entraînent une discontinuité dans le flux lorsqu'elles suppriment des éléments. De plus, ils ne signaleront pas `BufferOverflowException` .

onBackpressureDrop ()

Chaque fois que l'aval n'est pas prêt à recevoir des valeurs, cet opérateur déposera cet élément de la séquence. On peut le considérer comme une capacité `onBackpressureBuffer` 0 avec la stratégie `ON_OVERFLOW_DROP_LATEST` .

Cet opérateur est utile lorsque l'on peut ignorer en toute sécurité des valeurs provenant d'une source (telles que les déplacements de souris ou les signaux de position GPS actuels), car des valeurs plus récentes seront disponibles ultérieurement.

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

Cela peut être utile en conjonction avec l' `interval()` opérateur source `interval()` . Par exemple, si vous souhaitez effectuer une tâche d'arrière-plan périodique mais que chaque itération peut durer plus longtemps que la période, il est conseillé de supprimer la notification d'intervalle supplémentaire car il y en aura plus tard:

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

Il existe une surcharge de cet opérateur: `onBackpressureDrop(Action1<? super T> onDrop)` où l'action (partagée) est appelée avec la valeur en cours de suppression. Cette variante permet de nettoyer les valeurs elles-mêmes (par exemple, libérer des ressources associées).

onBackpressureLatest ()

L'opérateur final conserve uniquement la dernière valeur et remplace pratiquement les anciennes valeurs non livrées. On peut penser à ceci comme une variante du `onBackpressureBuffer` avec une capacité de 1 et une stratégie de `ON_OVERFLOW_DROP_OLDEST` .

Contrairement à `onBackpressureDrop` il existe toujours une valeur disponible pour la consommation si l'aval est en retard. Cela peut être utile dans certaines situations de télémétrie où les données peuvent être irrégulières, mais seule la toute dernière est intéressante pour le traitement.

Par exemple, si l'utilisateur clique beaucoup sur l'écran, nous souhaiterions toujours réagir à sa dernière entrée.

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

L'utilisation de `onBackpressureDrop` dans ce cas conduirait à une situation où le tout dernier clic serait déposé et laisserait l'utilisateur se demander pourquoi la logique métier n'était pas exécutée.

Création de sources de données sous pression

La création de sources de données à contre-courant est la tâche relativement facile pour gérer la

contre-pression en général, car la bibliothèque propose déjà des méthodes statiques sur `Observable` qui gèrent la contre-pression du développeur. Nous pouvons distinguer deux types de méthodes d'usine: les «générateurs» à froid qui renvoient et génèrent des éléments basés sur la demande en aval et les «pousseurs» chauds qui relient généralement les sources de données non réactives et non réversibles. leur.

juste

La source la plus consciente de base est créé contrepression via `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
}
```

Puisque nous ne demandons explicitement pas dans `onStart` , cela n'imprimera rien. `just` est génial quand il y a une valeur constante , nous aimerions relancer une séquence.

Malheureusement, `just` est souvent confondu avec un moyen de calculer quelque chose de dynamique à être consommé par l' `Subscriber` s:

```
int counter;

int computeValue() {
   return ++counter;
}

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

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

Étonnant à certains, cela imprime 1 fois au lieu d’imprimer 1 et 2 respectivement. Si l'appel est réécrit, il devient évident pourquoi cela fonctionne:

```
int temp = computeValue();

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

`computeValue` est appelée dans le cadre de la routine principale et non en réponse aux abonnés abonnés.

fromCallable

Ce dont les gens ont réellement besoin, c'est de la méthode `fromCallable` :

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

Ici, la valeur `computeValue` n'est exécutée que lorsqu'un abonné souscrit et pour chacun d'eux, en imprimant les `fromCallable` attendues 1 et 2. Naturellement, `fromCallable` également correctement en charge la contre-pression et n'émettra pas la valeur calculée sauf demande. Notez cependant que le calcul se produit quand même. Dans le cas où le calcul lui-même devrait être retardé jusqu'à ce que l'aval demande réellement, nous pouvons utiliser `just map` :

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

`just` va `just` pas émettre sa valeur constante jusqu'à ce qu'il soit demandé quand il est mappé au résultat de la valeur de `computeValue` , toujours appelé pour chaque abonné individuellement.

de

Si les données sont déjà disponibles comme un tableau d'objets, une liste d'objets ou d' une `Iterable` source respectif `from` surcharges manipuleront la contre - pression et l' émission de ces sources:

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

Pour plus de commodité (et en évitant les mises en garde sur la création de tableau générique) il y a 2 à 10 argument de surcharges `just` que déléguer en interne à `from` .

Le `from(Iterable)` donne également une opportunité intéressante. Beaucoup de génération de valeur peuvent être exprimées sous la forme d'une machine à états. Chaque élément demandé déclenche une transition d'état et un calcul de la valeur renvoyée.

L' écriture de ces machines d'état comme `Iterable` s est un peu compliqué (mais encore plus facile que d' écrire un `Observable` pour consommer) et contrairement à C #, Java ne pas le soutien du compilateur pour construire ces machines d'état en écrivant le code à la recherche classique (avec un `yield return` et `yield break`). Certaines bibliothèques offrent une aide, telle que `AbstractIterable` Google Guava et `Ix.generate()` et `Ix.forloop()` . Celles-ci sont dignes d'une série complète, voyons donc une source `Iterable` très basique qui répète indéfiniment une valeur constante:

```
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 nous consommions l' `iterator` via for-loop classique, cela entraînerait une boucle infinie. Puisque nous en construisons une `Observable` , nous pouvons exprimer notre volonté de ne consommer que les cinq premiers, puis cesser de demander quoi que ce soit. C'est le véritable pouvoir d'évaluation et de calcul paresseux à l'intérieur de `Observable` .

créer (SyncOnSubscribe)

Parfois, la source de données à convertir dans le monde réactif lui-même est synchrone (bloquant) et ressemble à un pull, c'est-à-dire que nous devons appeler une méthode `get` ou `read` pour obtenir le prochain élément de données. On pourrait, bien sûr, transformer cela en une `Iterable` mais lorsque de telles sources sont associées à des ressources, nous pouvons perdre ces ressources si l'aval se désabonne de la séquence avant sa fin.

Pour gérer de tels cas, RxJava a la classe `SyncOnSubscribe` . On peut l'étendre et implémenter ses méthodes ou utiliser l'une de ses méthodes d'usine basées sur lambda pour créer une instance.

```
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);
```

`SyncOnSubscribe` utilise généralement 3 rappels.

Les premiers rappels permettent de créer un état par abonné, tel que `FileInputStream` dans l'exemple; le fichier sera ouvert indépendamment à chaque abonné individuel.

Le second rappel prend cet objet d'état et fournit une sortie `Observer` dont les méthodes `onXXX` peuvent être appelées pour émettre des valeurs. Ce rappel est exécuté autant de fois que l'aval demandé. A chaque invocation, il doit appeler `onNext` au plus une fois éventuellement suivi par `onError` ou `onCompleted` . Dans l'exemple, nous appelons `onCompleted()` si l'octet de lecture est négatif, en indiquant et en fin de fichier, et appelons `onError` si la lecture renvoie une `IOException` .

Le rappel final est appelé lorsque la désabonnement en aval (fermeture du flux d'entrée) ou lorsque le rappel précédent a appelé les méthodes du terminal; cela permet de libérer des ressources. Toutes les sources n'ayant pas besoin de toutes ces fonctionnalités, les méthodes statiques de `SyncOnSubscribe` permettent de créer des instances sans elles.

Malheureusement, de nombreux appels de méthodes à travers la JVM et d'autres bibliothèques lancent des exceptions vérifiées et doivent être intégrés dans des `try-catch` car les interfaces fonctionnelles utilisées par cette classe ne permettent pas de lancer des exceptions vérifiées.

Bien sûr, nous pouvons imiter d'autres sources typiques, telles qu'une gamme sans limites:

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

Dans cette configuration, le `current` commence avec `0` et la prochaine fois que le lambda est invoqué, le paramètre `current` est maintenant `1` .

Il existe une variante de `SyncOnSubscribe` appelée `AsyncOnSubscribe` qui ressemble beaucoup à l'exception que le callback du milieu prend également une valeur longue qui représente le montant de la requête en aval et que le rappel doit générer une `Observable` de même longueur. Cette source concatène ensuite tous ces `Observable` en une seule séquence.

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

Il y a une discussion en cours (passionnée) sur l'utilité de cette classe et généralement non recommandée car elle rompt régulièrement les attentes sur la façon dont elle va réellement émettre ces valeurs générées et comment elle va répondre, ou même sur le type de demande qu'elle recevra. scénarios de consommation plus complexes.

créer (émetteur)

Parfois, la source à encapsuler dans un objet `Observable` est déjà chaude (comme les déplacements de souris) ou froide mais ne peut pas être rétrogradée dans son API (comme un rappel réseau asynchrone).

Pour gérer de tels cas, une version récente de RxJava a introduit la méthode de `create(emitter)` . Il faut deux paramètres:

- un rappel qui sera appelé avec une instance de l'interface `Emitter<T>` pour chaque abonné entrant,
- une énumération `Emitter.BackpressureMode` qui oblige le développeur à spécifier le comportement de contre-pression à appliquer. Il possède les modes habituels, similaires à `onBackpressureXXX` en plus de signaler une `MissingBackpressureException` ou d'ignorer simplement un tel débordement à l'intérieur.

Notez qu'il ne prend actuellement pas en charge les paramètres supplémentaires pour ces modes de contre-pression. Si vous avez besoin de cette personnalisation, vous devez utiliser `NONE` comme mode de contre-pression et appliquer le `onBackpressureXXX` approprié à l' `Observable` résultant.

Le premier cas typique d'utilisation de celui-ci lorsque l'on souhaite interagir avec une source basée sur les commandes, comme les événements d'interface graphique. Ces API comportent une forme d' `addListener` / `removeListener` que l'on peut utiliser:

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

    button.addActionListener(al);

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

}, BackpressureMode.BUFFER);
```

L' `Emitter` est relativement simple à utiliser; on peut appeler `onNext` , `onError` et `onCompleted` dessus et l'opérateur gère lui-même la gestion de la contre-pression et de la désinscription. En outre, si l'API encapsulée prend en charge l'annulation (telle que la suppression de l'écouteur dans l'exemple), on peut utiliser `setCancellation` (ou `setSubscription` pour `Subscription` like Ressource) pour enregistrer un rappel d'annulation qui est `onError` ou `onError` / `onCompleted` est appelé sur l'instance `Emitter` fournie.

Ces méthodes permettent à une seule ressource d'être associée à l'émetteur à la fois et de définir une nouvelle pour désabonner l'ancienne. Si vous devez gérer plusieurs ressources, créez un `CompositeSubscription` , associez-le à l'émetteur, puis ajoutez des ressources supplémentaires à `CompositeSubscription` :

```
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);
```

Le second scénario implique généralement une API asynchrone basée sur le rappel qui doit être convertie en une `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);
```

Dans ce cas, la délégation fonctionne de la même manière. Malheureusement, ces API classiques de style callback ne prennent généralement pas en charge l'annulation, mais si tel est le cas, vous pouvez configurer leur annulation comme dans les exemples précédents (avec peut-être un moyen plus complexe). Notez l'utilisation du `LATEST` mode contre - pression; Si nous savons qu'il n'y aura qu'une seule valeur, nous n'avons pas besoin de la stratégie `BUFFER` car elle alloue un tampon long de 128 éléments par défaut (qui croît au besoin) qui ne sera jamais pleinement utilisé.

Lire Contrepression en ligne: https://riptutorial.com/fr/rx-java/topic/2341/contrepression

Chapitre 4: Les opérateurs

Remarques

Ce document décrit le comportement de base d'un opérateur.

Examples

Opérateurs, une introduction

Un opérateur peut être utilisé pour manipuler le flux d'objets de `Observable` à `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 sortie serait:

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

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

L'opérateur de `map` changé le `Integer` observable en une `String` observable, manipulant ainsi le flux d'objets.

Chaînage de l'opérateur

Plusieurs opérateurs peuvent être `chained` pour des transformations et des manipulations plus puissantes.

```
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
```

Un nombre quelconque d'opérateurs peut être ajouté entre l' `Observable` et l' `Subscriber` .

FlatMap Operator

L'opérateur `flatMap` vous aide à transformer un événement en un autre `Observable` (ou à transformer un événement en zéro, un ou plusieurs événements).

C'est un opérateur parfait quand vous voulez appeler une autre méthode qui retourne une `Observable`

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

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

`flatMap` sérialise `perform` des abonnements , **mais** les événements emited par `perform` ne peuvent pas être commandés. Ainsi , vous pouvez recevoir des événements émis par le dernier appel **avant** effectuer les événements de la première `perform` l' appel (vous devez utiliser `concatMap` à la place).

Si vous créez une autre `Observable` dans votre abonné, vous **devriez** plutôt utiliser `flatMap` . L'idée principale est la suivante: **ne jamais laisser l'observable**

Par exemple :

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

peut facilement être remplacé par:

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

Documentation Reactivex.io: http://reactivex.io/documentation/operators/flatmap.html

opérateur de filtre

Vous pouvez utiliser le `filter` opérateur pour filtrer les éléments du flux de valeurs en fonction du résultat d'une méthode sous - jacente.

En d'autres termes, les éléments passant de l'observateur à l'abonné seront supprimés en fonction du `filter` fonction que vous transmettez, si la fonction renvoie `false` pour une certaine valeur, cette valeur sera filtrée.

Exemple:

```
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
    });
```

Ce code sera imprimé

```
0.0
4.0
16.0
36.0
64.0
```

Opérateur de carte

Vous pouvez utiliser la `map` opérateur pour cartographier les valeurs d'un flux de valeurs différentes en fonction du résultat pour chaque valeur de la fonction passée à la `map` . Le flux de résultats est une nouvelle copie et ne modifiera pas le flux de valeurs fourni, le flux de résultats aura la même longueur que le flux d'entrée, mais pourra être de différents types.

La fonction transmise à `.map()` doit renvoyer une valeur.

Exemple:

```
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
    });
```

Ce code sera imprimé

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

Dans cet exemple, l'observable a accepté une `List<Integer>` la liste sera transformée en une `List<String>` dans le pipeline et le `.subscribe` émettra des `String` .

Opérateur doOnNext

Opérateur `doOnNext` appelé à chaque fois que la source `Observable` émet un élément. Il peut être utilisé à des fins de débogage, en appliquant une action à l'élément émis, la journalisation, 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();
```

Dans l'exemple ci-dessous, `doOnNext` n'est jamais appelé, car la source `Observable` n'émet rien car `Observable.empty()` appelle `onCompleted` après s'être abonné.

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

opérateur de répétition

`repeat` opérateur de `repeat` permet de répéter toute la séquence de la source `Observable` .

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

Sortie de l'exemple ci-dessus

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

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

Cette séquence répète un nombre infini de fois et ne se termine jamais.

Pour répéter la séquence nombre fini de fois, il suffit de passer un entier comme argument à l'opérateur de `repeat` .

```
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")
    );
```

Cet exemple imprime

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

Il est très important de comprendre que l'opérateur de `repeat` réabonne à la source `Observable` lorsque la séquence source `Observable` terminée. Réécrivons l'exemple ci-dessus en utilisant `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")
        );
```

Cet exemple imprime

```
Subscribed
next: 1
```

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

Lors de l'utilisation du chaînage d'opérateur, il est important de savoir que l'opérateur `repeat` répète **toute la séquence** plutôt que l'opérateur précédent.

```
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")
        );
```

Cet exemple 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
```

Cet exemple montre que la `repeat` opérateur répète la séquence entière réabonnant à `Observable` plutôt que de répéter la dernière `map` opérateur et il n'a pas d' importance à quel endroit dans la séquence `repeat` opérateur utilisé.

Cette séquence

```
Observable.<Integer>create(subscriber -> {
        //...
    })
```

```
    .map(value -> value * 2) //First chain operator
    .map(value -> "modified " + value) //Second chain operator
    .repeat(3)
    .subscribe(
        /*....*/
    );
```

est égal à cette séquence

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

Lire Les opérateurs en ligne: https://riptutorial.com/fr/rx-java/topic/2316/les-operateurs

Chapitre 5: Les ordonnanceurs

Examples

Exemples de base

Les ordonnanceurs sont une abstraction de RxJava sur l'unité de traitement. Un planificateur peut être sauvegardé par un service Executor, mais vous pouvez implémenter votre propre implémentation de planificateur.

Un `Scheduler` doit répondre à cette exigence:

- Devrait traiter la tâche non retardée séquentiellement (ordre FIFO)
- La tâche peut être retardée

Un `Scheduler` peut être utilisé comme paramètre dans certains opérateurs (exemple: `delay`) ou utilisé avec la méthode `subscribeOn` / `observeOn` .

Avec certains opérateurs, le `Scheduler` sera utilisé pour traiter la tâche de l'opérateur spécifique. Par exemple, `delay` planifiera une tâche différée qui émettra la valeur suivante. Ceci est un `Scheduler` qui le conservera et l'exécutera plus tard.

Le `subscribeOn` peut être utilisé une fois par `Observable` . Il définira dans quel `Scheduler` le code de l'abonnement sera executer.

`observeOn` peut être utilisé plusieurs fois par `Observable` . Il définira dans quel `Scheduler` sera utilisé pour exécuter toutes les tâches définies **après** la méthode `observeOn` . `observeOn` vous aidera à effectuer des sauts de fil.

SubscribeOn Scheduler spécifique

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

observer sur planificateur spécifique

```
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()));
```

Spécifier un planificateur spécifique avec un opérateur

Certains opérateurs peuvent prendre un `Scheduler` comme paramètre.

```
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);
```

Publier sur l'abonné:

```
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);
```

Pool de threads:

```
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();
```

Lire Les ordonnanceurs en ligne: https://riptutorial.com/fr/rx-java/topic/2321/les-ordonnanceurs

Chapitre 6: Observable

Examples

Créer une observable

Il existe plusieurs façons de créer une observable dans RxJava. Le moyen le plus efficace consiste à utiliser la méthode `Observable.create` . Mais c'est aussi la **manière** la plus **compliquée** . Vous devez donc **éviter de l'utiliser** autant que possible.

Émettre une valeur existante

Si vous avez déjà une valeur, vous pouvez utiliser `Observable.just` pour émettre votre valeur.

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

Emettre une valeur qui devrait être calculée

Si vous voulez émettre une valeur qui n'est pas déjà calculée ou qui peut prendre du temps à être calculée, vous pouvez utiliser `Observable.fromCallable` pour émettre votre prochaine valeur.

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

`longComputation()` ne sera appelé que lorsque vous vous abonnez à votre `Observable` . De cette façon, le calcul sera *paresseux* .

Autre manière d'émettre une valeur à calculer

`Observable.defer` construit un `Observable` tout comme `Observable.fromCallable` mais il est utilisé lorsque vous devez retourner un `Observable` au lieu d'une valeur. C'est utile quand vous voulez gérer les erreurs dans votre appel.

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

Observables chauds et froids

Les observables sont généralement classés comme `Hot` ou `Cold` , selon leur comportement en

matière d'émission.
Un objet `Cold Observable` est celui qui commence à émettre sur demande (abonnement), tandis qu'un objet `Hot Observable` est celui qui émet indépendamment des abonnements.

Observable à froid

```
/* 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 sortie du code ci-dessus ressemble (peut varier):

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

Notez que même si `sub2` commence en retard, il reçoit des valeurs dès le début. Pour conclure, un `Cold Observable` n'émet que des éléments demandés. Une demande multiple démarre plusieurs pipelines.

Observable à chaud

Remarque: Les observables à chaud émettent des valeurs indépendantes des abonnements individuels. Ils ont leur propre calendrier et les événements se produisent que quelqu'un écoute ou non.

Une `Cold Observale` peut être convertie en une `Hot Observable` avec une simple `publish` .

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

`publish` renvoie un objet `ConnectableObservable` qui ajoute des fonctionnalités pour se *connecter* et se *déconnecter* de l'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));
```

Les rendements ci-dessus:

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

Notez que même si `sub2` commence à observer tard, il est synchronisé avec `sub1` .
Déconnecter est un peu plus compliqué! La déconnexion se produit sur l' `Subscription` et non sur l' `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();
```

Le ci-dessus produit:

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

Lors de la déconnexion, l' `Observable` "termine" essentiellement et redémarre lorsqu'un nouvel abonnement est ajouté.

`Hot Observable` peut être utilisé pour créer un `EventBus` . Ces EventBuses sont généralement légers et super rapides. Le seul inconvénient d'un RxBus est que tous les événements doivent être implémentés manuellement et transmis au bus.

Lire Observable en ligne: https://riptutorial.com/fr/rx-java/topic/1418/observable

Chapitre 7: Rénovation et RxJava

Examples

Configurer la mise à niveau et RxJava

Retrofit2 prend en charge plusieurs mécanismes d'exécution enfichables, dont RxJava.

Pour utiliser la mise à niveau avec RxJava, vous devez d'abord ajouter l'adaptateur Retrofit RxJava à votre projet:

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

Vous devez ensuite ajouter l'adaptateur lors de la création de votre instance de mise à niveau:

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

Dans votre interface lorsque vous définissez l'API, le type de retour doit être `Observable` par exemple:

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

Vous pouvez également utiliser `Single` au lieu de `Observable` .

Faire des requêtes en série

RxJava est pratique lorsque vous effectuez une requête série. Si vous souhaitez utiliser le résultat d'une requête pour en créer une autre, vous pouvez utiliser l'opérateur `flatMap` :

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

Faire des demandes parallèles

Vous pouvez utiliser l'opérateur `zip` pour faire une requête en parallèle et combiner les résultats, par exemple:

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

Lire Rénovation et RxJava en ligne: https://riptutorial.com/fr/rx-java/topic/2950/renovation-et-rxjava

Chapitre 8: RxJava2 Flowable et Subscriber

Introduction

Cette rubrique présente des exemples et de la documentation concernant les concepts réactifs de Flowable et Subscriber introduits dans la version 2 de rxjava.

Remarques

l'exemple nécessite rxjava2 comme dépendance, les coordonnées maven pour la version utilisée sont:

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

Examples

exemple de consommateur producteur avec support de contrepression dans le producteur

Le `TestProducer` de cet exemple produit des objets `Integer` dans une plage donnée et les transmet à son `Subscriber` . Il étend la classe `Flowable<Integer>` . Pour un nouvel abonné, il crée un objet `Subscription` dont la méthode `request(long)` est utilisée pour créer et publier les valeurs Integer.

Il est important pour l' `Subscription` qui est transmis à l' `subscriber` que la méthode `request()` qui appelle `onNext()` sur l'abonné puisse être appelée de manière récursive dans cet appel `onNext()` . Pour empêcher un débordement de pile, l'implémentation affichée utilise le compteur `outStandingRequests` et l'indicateur `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;
            }
        });
    }
}
```

Le consommateur dans cet exemple étend `DefaultSubscriber<Integer>` et au démarrage et après avoir consommé un Integer demande le suivant. En consommant les valeurs Integer, il y a un petit délai, donc la contre-pression sera créée pour le producteur.

```
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");
    }
}
```

Dans la méthode principale suivante d'une classe de test, le producteur et le consommateur sont créés et connectés:

```
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();
    }
}
```

Lors de l'exécution de l'exemple, le fichier journal indique que le consommateur s'exécute en continu, tandis que le producteur ne devient actif que lorsque le tampon Flowable interne de rxjava2 doit être rempli.

Lire RxJava2 Flowable et Subscriber en ligne: https://riptutorial.com/fr/rx-java/topic/9810/rxjava2-flowable-et-subscriber

Chapitre 9: Sujets

Syntaxe

- Sujet <T, R> subject = AsyncSubject.create (); // AsyncSubject par défaut
- Sujet <T, R> subject = BehaviorSubject.create (); // Comportement par défaut
- Sujet <T, R> subject = PublishSubject.create (); // PublishSubject par défaut
- Sujet <T, R> subject = ReplaySubject.create (); // ReplaySubject par défaut
- mySafeSubject = new SerializedSubject (unSafeSubject); // Convertit un unsafeSubject en un safeSubject - généralement pour les sujets multi-threadés

Paramètres

Paramètres	Détails
T	Type d'entrée
R	Le type de sortie

Remarques

Cette documentation fournit des détails et des explications sur le `Subject` . Pour plus d'informations et de lectures supplémentaires, veuillez consulter la documentation officielle .

Examples

Sujets de base

Un `Subject` dans RxJava est une classe à la fois `Observable` et `Observer` . Cela signifie essentiellement qu'il peut agir en tant `Observable` et transmettre des données aux abonnés et en tant `Observer` pour obtenir des données d'un autre observable.

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

Les copies ci-dessus "Hello, World!" pour consoler en utilisant des `Subjects` .

Explication

1. La première ligne de code définit un nouveau `Subject` de type `PublishSubject`

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

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

2. La deuxième ligne s'abonne au sujet, montrant le comportement de l' `Observer` .

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

   Cela permet au `Subject` de prendre des entrées comme un abonné régulier

3. La troisième ligne appelle la méthode `onNext` du sujet, montrant le comportement `Observable` .

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

   Cela permet au `Subject` de donner des informations à tous les abonnés.

Les types

Un `Subject` (dans RxJava) peut être de l'un de ces quatre types:

- AsyncSubject
- Comportement
- PublishSubject
- ReplaySubject

De plus, un `Subject` peut être de type `SerializedSubject` . Ce type garantit que l' `Subject` ne viole pas le *contrat observable* (qui spécifie que tous les appels doivent être sérialisés)

Lectures complémentaires:

- Utiliser ou ne pas utiliser le sujet du blog de Dave Sexton

PublishSubject

`PublishSubject` n'émet sur un `Observer` que les éléments émis par l' `Observable` source après l'heure de l'abonnement.

Un exemple 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);
```

Sortie:

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

Dans l'exemple ci-dessus, un objet `PublishSubject` souscrit à une `Observable` qui agit comme une horloge et émet des éléments (Long) toutes les 500 millisecondes. Comme on le voit dans la sortie, le `PublishSubject` transmet les `PublishSubject` qu'il reçoit de la source (`clock`) à ses abonnés (`sub1` et `sub2`).

Un objet `PublishSubject` peut commencer à émettre des éléments dès qu'il est créé, sans aucun observateur, ce qui risque d'entraîner la perte d'un ou de plusieurs éléments jusqu'à ce qu'un observateur puisse s'inscrire.

```
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
```

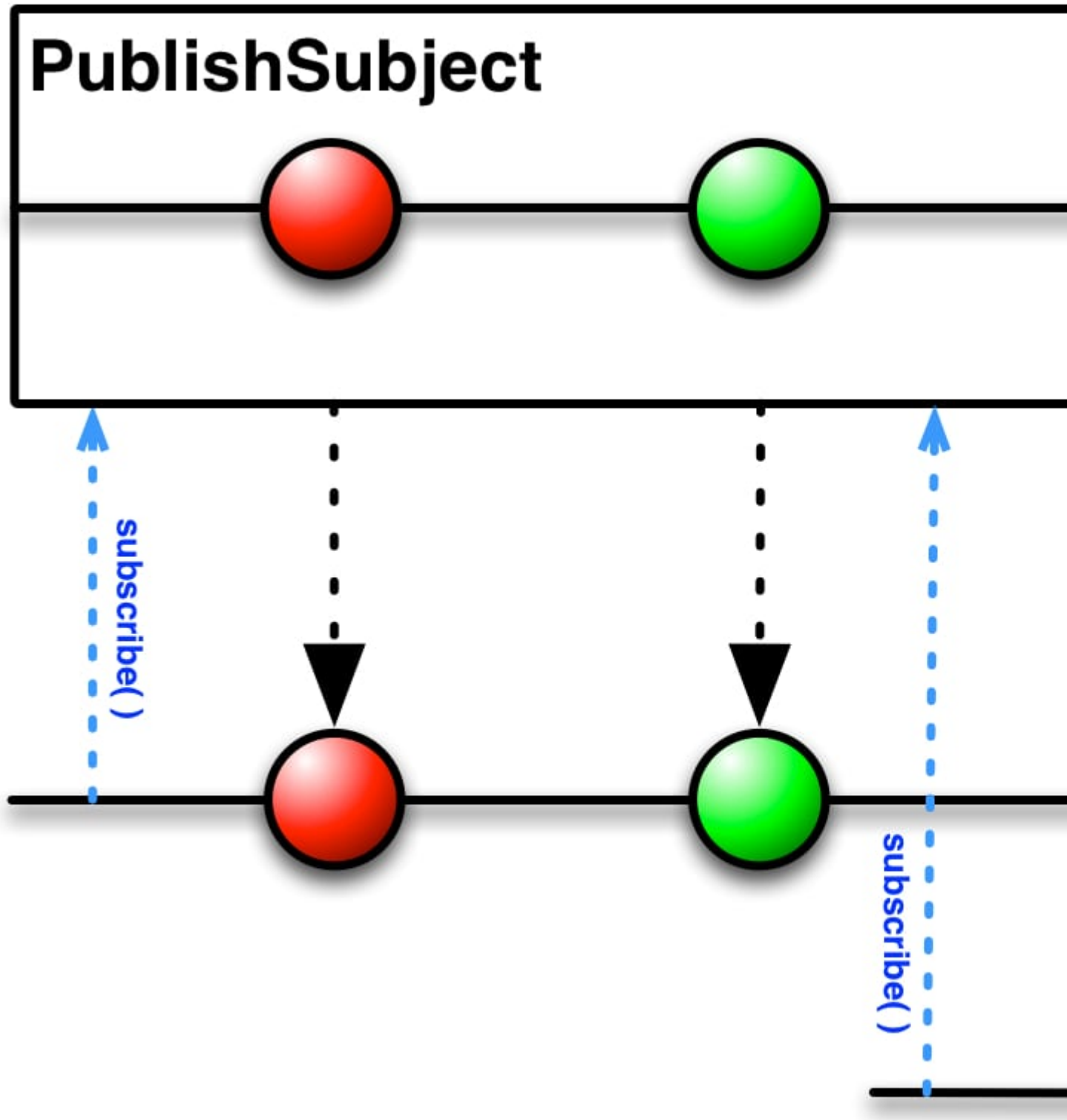
Sortie:

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

Notez que `sub1` émet des valeurs à partir de `10` . Le délai de 5 secondes introduit a entraîné une *perte* d'articles. Ceux-ci ne peuvent pas être reproduits. Cela fait essentiellement de `PublishSubject` une `Hot Observable` .

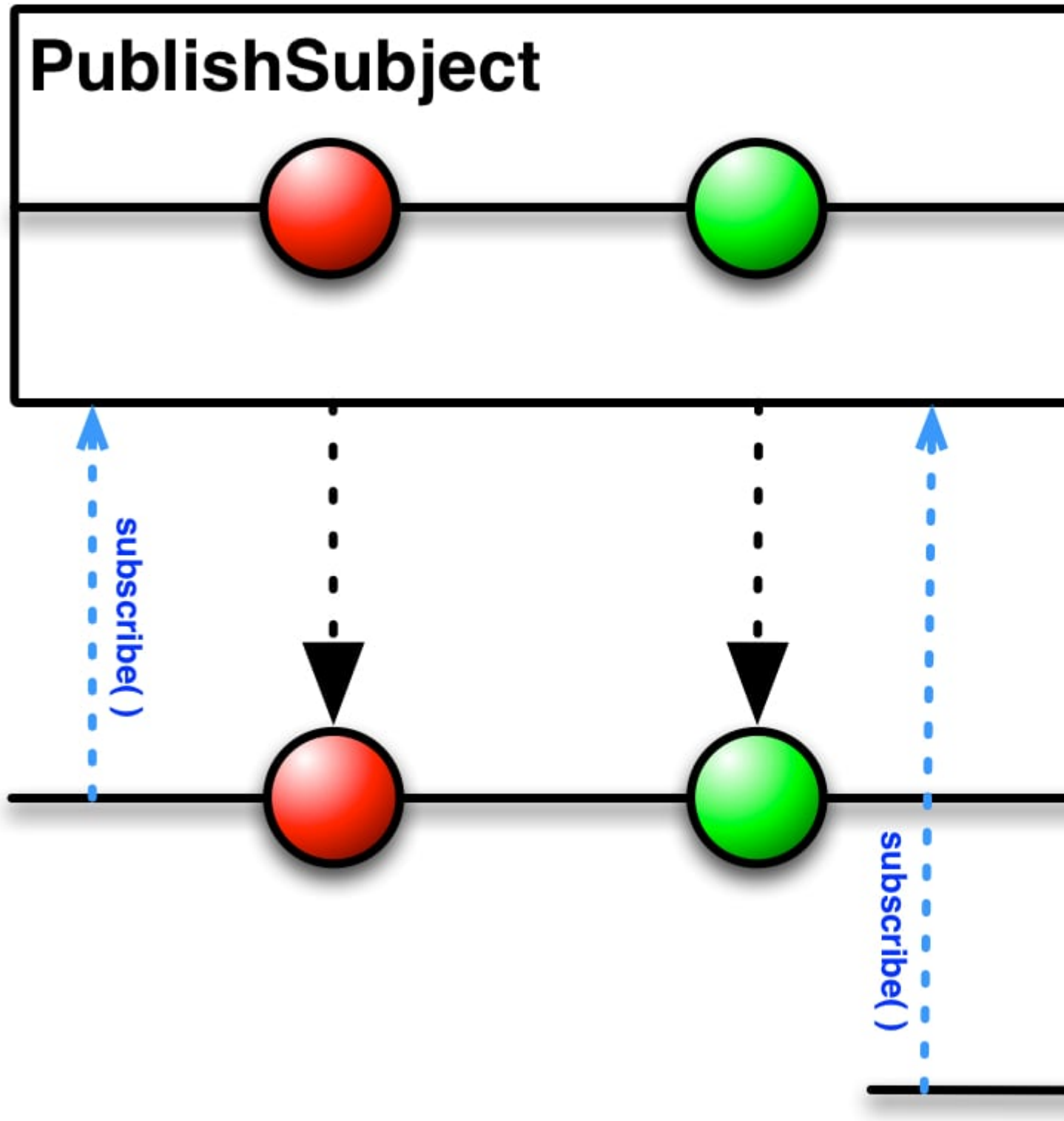
Notez également que si un observateur s'abonne à `PublishSubject` après avoir émis ***n*** éléments, ces ***n*** éléments *ne peuvent pas* être reproduits pour cet observateur.

Ci-dessous le diagramme de marbre de `PublishSubject`



Le `PublishSubject` émet des éléments vers tous ceux qui ont souscrit, à tout moment avant l' `onCompleted` de `onCompleted` de la source `Observable` .

Si la source `Observable` termine par une erreur, le `PublishSubject` n'émettra aucun élément aux observateurs suivants, mais transmettra simplement la notification d'erreur de la source Observable.



Cas d'utilisation

Supposons que vous souhaitiez créer une application qui surveillera les cours des actions d'une certaine société et la transmettra à tous les clients qui en feront la demande.

```
/* 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();
```

Dans l'exemple ci-dessus, le `PublishSubject` fait office de passerelle pour transmettre les valeurs de votre serveur à tous les clients abonnés à votre `watcher` .

Lectures complémentaires:

- Publier des javadocs
- Blog de Thomas Nield (Lecture avancée)

Lire Sujets en ligne: https://riptutorial.com/fr/rx-java/topic/3287/sujets

Chapitre 10: Test d'unité

Remarques

Toutes les méthodes de Schedulers étant statiques, les tests unitaires utilisant les hooks RxJava ne peuvent pas être exécutés en parallèle sur la même instance de JVM. Si c'est le cas, un seul TestScheduler serait supprimé au milieu d'un test unitaire. C'est fondamentalement l'inconvénient d'utiliser la classe Schedulers.

Examples

TestSubscriber

TestSubscribers vous permet d'éviter le travail de création de votre propre abonné ou de vous inscrire Action <?> Pour vérifier que certaines valeurs ont été livrées, combien il y en a, si l'observable est terminé, une exception a été soulevée et beaucoup plus.

Commencer

Cet exemple montre simplement que les valeurs 1, 2, 3 et 4 sont passées dans l'observable via onNext.

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

`assertValues` affirme que le compte est correct. Si vous ne deviez transmettre que certaines des valeurs, l'assertion échouerait.

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

`assertValues` utilise la méthode `equals` lors des `assertValues` . Cela vous permet de tester facilement les classes traitées comme des données.

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

Cet exemple montre une classe qui a un égal défini et qui affirme les valeurs de l'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
```

Notez également que nous utilisons la valeur `assertValue` la plus `assertValue` car nous n'avons besoin que de vérifier un élément.

Obtenir tous les événements

Si nécessaire, vous pouvez également demander tous les événements sous forme de liste.

```
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();
```

Affirmer sur des événements

Si vous souhaitez effectuer des tests plus approfondis sur vos événements, vous pouvez combiner `getOnNextEvents` (ou `getOn*Events`) avec votre bibliothèque d'assertions préférée:

```
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));
```

Test de l' `Observable#error`

Vous pouvez vous assurer que la classe d'exception correcte est émise:

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

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

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

Vous pouvez également vous assurer que l'exception exacte a été lancée:

```
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 vous permet de contrôler le temps et l'exécution des observables au lieu d'avoir à faire des attentes occupées, à joindre des threads ou à manipuler l'heure du système. Ceci est très important si vous voulez écrire des tests unitaires prévisibles, cohérents et rapides. Parce que vous manipulez le temps, il n'y a plus de chance qu'un thread soit affamé, que votre test échoue sur une machine plus lente ou que vous perdiez du temps à attendre un résultat.

TestSchedulers peut être fourni via la surcharge qui prend un planificateur pour toutes les opérations 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
```

Le TestScheduler est assez basique. Il ne comprend que trois méthodes.

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

Cela vous permet de manipuler lorsque le TestScheduler doit déclencher toutes les actions concernant un certain temps dans le futur.

En passant le planificateur fonctionne, ce n'est pas la façon dont le TestScheduler est couramment utilisé en raison de son inefficacité. Passer des ordonnanceurs en classes finit par fournir beaucoup de code supplémentaire pour un gain minime. Au lieu de cela, vous pouvez accéder à Schedulers.io () / computation () / etc de RxJava. Ceci est fait avec les crochets de RxJava. Cela vous permet de définir ce qui est renvoyé par un appel provenant d'une des méthodes Schedulers.

```
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;
    }
}
```

Cette classe permet à l'utilisateur d'obtenir le planificateur de test qui sera connecté à tous les appels aux planificateurs. Un test unitaire devrait simplement avoir ce planificateur dans sa configuration. Il est fortement recommandé de l'acquérir dans le programme d'installation et de ne pas utiliser de vieux champ, car votre TestScheduler peut essayer de déclencher des actions à partir d'un autre test unitaire lorsque vous avancez. Maintenant, notre exemple ci-dessus devient

```
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)
```

C'est ainsi que vous pouvez supprimer efficacement l'horloge système de votre test unitaire (au moins pour ce qui concerne RxJava)

Lire Test d'unité en ligne: https://riptutorial.com/fr/rx-java/topic/5207/test-d-unite

Crédits

S. No	Chapitres	Contributeurs
1	Démarrer avec rx-java	Buttink, Community, dimsuz, Dmitry Avtonomov, Hans Wurst, hello_world, Omar Al Halabi, Saulius Next, Sneh Pandya, svarog, Tom
2	Android avec RxJava	akarnokd, Athafoud, Daniele Segato, Eugen Martynov, Geng Jiawen, Sneh Pandya
3	Contrepression	akarnokd, Bartek Lipinski, Chris A, Cristian, dwursteisen, Niklas, Sebas LG
4	Les opérateurs	dwursteisen, hello_world, svarog, Vadeg
5	Les ordonnanceurs	dwursteisen, Gal Dreiman
6	Observable	Aki K, dwursteisen, hello_world, JonesV
7	Rénovation et RxJava	LordRaydenMK
8	RxJava2 Flowable et Subscriber	P.J.Meisch
9	Sujets	hello_world, mavHarsha
10	Test d'unité	Buttink, Sir Celsius