EBook Gratuito

APPRENDIMENTO rx-java

Free unaffiliated eBook created from
Stack Overflow contributors.

#rx-java

Sommario

Di

Capitolo 1: Iniziare con rx-java

Osservazioni

Questa sezione fornisce una panoramica di base e un'introduzione superficiale a rx-java.

RxJava è un'implementazione Java VM di Reactive Extensions : una libreria per la composizione di programmi asincroni e basati su eventi utilizzando sequenze osservabili.

Ulteriori informazioni su RxJava sulla Wiki Home .

Versioni

Versione	Stato	Ultima versione stabile	Data di rilascio
1.x	Stabile	1.3.0	2017/05/05
2.x	Stabile	2.1.1	2017/06/21

Examples

Installazione o configurazione

rx-java impostato

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. Edera

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

4. Istantanee di JFrog

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

```
}

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

5. Se devi scaricare i vasi invece di usare un sistema di build, crea un file Maven `pom` come questo con la versione desiderata:

```
<?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>
```

Quindi eseguire:

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

Questo comando scarica `rxjava-*.jar` e le sue dipendenze in `./target/dependency/`.

Hai bisogno di Java 6 o versioni successive.

Ciao mondo!

Quanto segue stampa il messaggio `Hello, World!` per consolare

```
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 notazione lambda Java 8

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

Un'introduzione a RxJava

I concetti chiave di RxJava sono i suoi `Observables` e `Subscribers` . Un `Observable` emette oggetti, mentre un `Subscriber` li consuma.

Osservabile

`Observable` è una classe che implementa il modello di progettazione reattiva. Questi oggetti osservabili forniscono metodi che consentono ai consumatori di iscriversi alle modifiche degli eventi. Le modifiche all'evento sono innescate dall'osservabile. Non c'è alcuna limitazione al numero di abbonati che può avere un `Observable` , o il numero di oggetti che un `Observable` può emettere.

Prendi ad esempio:

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

Qui, un oggetto osservabile chiamato `integerObservable` e `stringObservable` vengono creati dal metodo factory `just` fornito dalla libreria Rx. Si noti che `Observable` è generico e può quindi emettere qualsiasi oggetto.

abbonato

Un `Subscriber` è il consumatore. Un `Subscriber` può sottoscrivere ***una sola*** osservabile. The `Observable` chiama i `onNext()` , `onCompleted()` e `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);
```

```
        }
    };
```

Si noti che `Subscriber` è anche generico e può supportare qualsiasi oggetto. Un `Subscriber` deve iscriversi alla osservabile chiamando il `subscribe` metodo sul osservabile.

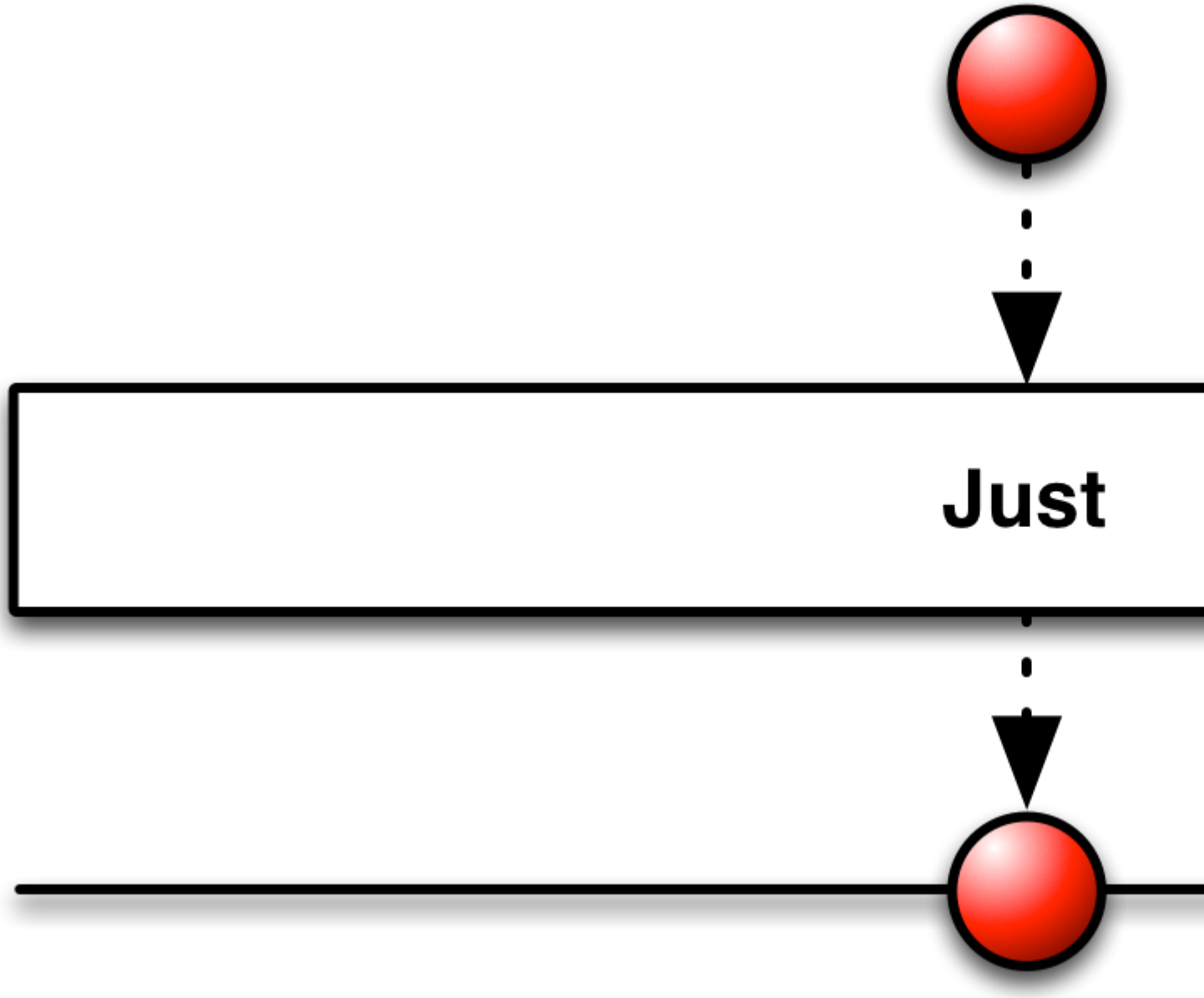
```
integerObservable.subscribe(mSubscriber);
```

Quanto sopra, quando eseguito, produrrà il seguente risultato:

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

Capire i diagrammi di marmo

Un osservabile può essere pensato come un semplice flusso di eventi. Quando definisci un Observable, hai tre listener: onNext, onComplete e onError. onNext sarà chiamato ogni volta che l'osservabile acquisisce un nuovo valore. onComplete sarà chiamato se il genitore Observable notifica che ha finito di produrre altri valori. onError viene chiamato se viene lanciata un'eccezione in qualsiasi momento durante l'esecuzione della catena Observable. Per mostrare gli operatori in Rx, il diagramma di marmo viene utilizzato per visualizzare ciò che accade con una particolare operazione. Di seguito è riportato un esempio di un operatore osservabile semplice "Giusto".



I diagrammi di marmo hanno un blocco orizzontale che rappresenta l'operazione eseguita, una barra verticale per rappresentare l'evento completato, una X per rappresentare un errore e qualsiasi altra forma rappresenta un valore. Con questo in mente, possiamo vedere che "Just" prenderà semplicemente il nostro valore e fare un onNext e poi terminare con onComplete. Ci sono molte più operazioni quindi solo "Just". Puoi vedere tutte le operazioni che fanno parte del progetto ReactiveX e le implementazioni in RxJava sul sito ReativeX . Ci sono anche diagrammi di marmo interattivi tramite il sito RxMarbles .

Leggi Iniziare con rx-java online: https://riptutorial.com/it/rx-java/topic/974/iniziare-con-rx-java

Capitolo 2: Android con RxJava

Osservazioni

RxAndroid era una libreria con molte funzionalità. È stato diviso in molte librerie diverse passando dalla versione 0.25.0 a 1.x.

Un elenco delle librerie che implementano le funzioni disponibili prima che il 1.0 è mantenuto qui .

Examples

RxAndroid - AndroidSchedulers

Questa è letteralmente l'unica cosa di cui hai bisogno per iniziare a utilizzare RxJava su Android.

Includere RxJava e RxAndroid nelle dipendenze gradle:

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

L'aggiunta principale di RxAndroid a RxJava è un programma di pianificazione per il thread principale Android o il thread dell'interfaccia utente.

Nel tuo codice:

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

Oppure puoi creare uno Scheduler per un `Looper` personalizzato:

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

Per la maggior parte di tutto il resto è possibile fare riferimento alla documentazione RxJava standard.

Componenti RxLifecycle

La libreria RxLifecycle semplifica l'associazione degli abbonamenti osservabili alle attività Android e al ciclo di vita dei frammenti.

Tieni presente che dimenticare di annullare l'iscrizione a un Osservabile può causare perdite di memoria e mantenere attivo l'evento attività / frammento dopo che è stato distrutto dal sistema.

Aggiungi la libreria alle dipendenze:

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

Quindi estende le classi `Rx*` :

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

Sei tutto pronto, quando ti iscrivi a un osservabile puoi ora:

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

Se lo si esegue nel metodo `onCreate()` dell'attività, verrà automaticamente annullato l'iscrizione in `onDestroy()` .

Lo stesso accade per:

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

In alternativa è possibile specificare l'evento quando si desidera che la disiscrizione avvenga:

Da un'attività:

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

Da un frammento:

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

È inoltre possibile ottenere il ciclo di vita osservabile utilizzando il metodo `lifecycle()` per ascoltare direttamente gli eventi del ciclo di vita.

RxLifecycle può anche essere utilizzato passando direttamente ad esso il ciclo di vita osservabile:

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

Se devi gestire `Single` o `Completable` puoi farlo aggiungendo rispettivamente `forSingle()` o `forCompletable` dopo il metodo bind:

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

Può anche essere utilizzato con la libreria Navi .

Rxpermissions

Questa libreria consente l'utilizzo di RxJava con il nuovo modello di autorizzazione di Android M.

Aggiungi la libreria alle dipendenze:

Rxjava

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

Rxjava2

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

uso

Esempio (con Retrolambda per brevità, ma non richiesto):

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

Ulteriori informazioni: https://github.com/tbruyelle/RxPermissions .

Leggi Android con RxJava online: https://riptutorial.com/it/rx-java/topic/7125/android-con-rxjava

Capitolo 3: Contropressione

Examples

introduzione

La contropressione avviene quando in una pipeline di elaborazione `Observable` alcuni stadi asincroni non sono in grado di elaborare i valori abbastanza velocemente e hanno bisogno di un modo per dire al produttore a monte di rallentare.

Il caso classico della necessità di contropressione è quando il produttore è una fonte di calore:

```
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);
```

In questo esempio, il thread principale produrrà 1 milione di elementi per un consumatore finale che lo sta elaborando su un thread in background. È probabile che il metodo di `compute(int)` richieda del tempo, ma il sovraccarico della catena di operatori `Observable` può anche aumentare il tempo necessario per elaborare gli articoli. Tuttavia, il thread di produzione con il ciclo for non può sapere ciò e continua a `onNext` .

Internamente, gli operatori asincroni dispongono di buffer per conservare tali elementi fino a quando non possono essere elaborati. Nel classico Rx.NET e all'inizio RxJava, questi buffer erano illimitati, il che significa che probabilmente conserverebbero quasi 1 milione di elementi dall'esempio. Il problema inizia quando ci sono, per esempio, 1 miliardo di elementi o la stessa sequenza di 1 milione appare 1000 volte in un programma, portando a `OutOfMemoryError` e generalmente rallentamenti a causa dell'overhead eccessivo del GC.

Analogamente a come la gestione degli errori è diventata un cittadino di prima classe e ha ricevuto operatori per affrontarla (tramite operatori `onErrorXXX`), la backpressure è un'altra proprietà dei flussi di dati che il programmatore deve pensare e gestire (tramite `onBackpressureXXX` operatori di `onBackpressureXXX`).

Oltre a `PublishSubject` sopra, ci sono altri operatori che non supportano la contropressione, principalmente per ragioni funzionali. Ad esempio, l' `interval` operatore emette valori periodicamente, la contropressione porterebbe a spostarsi nel periodo relativo a un orologio da parete.

Nel moderno RxJava, la maggior parte degli operatori asincroni ora ha un buffer interno limitato,

come `observeOn` sopra e qualsiasi tentativo di overflow di questo buffer terminerà l'intera sequenza con `MissingBackpressureException` . La documentazione di ciascun operatore ha una descrizione del suo comportamento di contropressione.

Tuttavia, la contropressione è presente in modo più sottile nelle sequenze fredde regolari (che non producono e non dovrebbero produrre `MissingBackpressureException`). Se il primo esempio è stato riscritto:

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

Thread.sleep(10_000);
```

Non ci sono errori e tutto funziona senza problemi con l'utilizzo di memoria di piccole dimensioni. La ragione di ciò è che molti operatori sorgente possono "generare" valori su richiesta e quindi l'operatore `observeOn` può dire che l' `range` genera al massimo così tanti valori che il buffer `observeOn` può contenere contemporaneamente senza overflow.

Questa trattativa si basa sul concetto di informatica delle co-routine (io ti chiamo, tu mi chiami). L'operatore `range` invia un callback, sotto forma di un'implementazione del `Producer` dell'interfaccia, alla `observeOn` chiamando il suo (interna `Subscriber` s') `setProducer` . In cambio, `observeOn` chiama `Producer.request(n)` con un valore per indicare `range` che è autorizzato a produrre (cioè, su `onNext`) molti **altri** elementi. È quindi responsabilità `observeOn` chiamare il metodo di `request` nel momento giusto e con il giusto valore per mantenere i dati fluidi ma non straripanti.

Raramente la necessità di esprimere una contropressione nei consumatori finali (perché sono sincroni rispetto al loro upstream immediato e alla contropressione avviene naturalmente a causa del blocco delle chiamate), ma potrebbe essere più facile capirne il funzionamento:

```
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!");
    }
});
```

Qui l'implementazione di `onStart` indica l' `range` per produrre il suo primo valore, che viene quindi ricevuto in `onNext` . Una volta terminato il `compute(int)` , viene richiesto un altro valore `range` . In un'implementazione ingenua `range` , tale chiamata chiamerebbe ricorsivamente `onNext` , portando a `StackOverflowError` che ovviamente non è desiderabile.

Per evitare ciò, gli operatori utilizzano la cosiddetta logica del trampolino che impedisce tali chiamate di rientro. Nei termini `range` , ricorderà che c'era una chiamata di `request(1)` mentre chiamava `onNext()` e una volta `onNext()` restituisce, farà un altro giro e chiamerà `onNext()` con il valore intero successivo. Pertanto, se i due vengono scambiati, l'esempio funziona ancora allo stesso modo:

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

    compute(v);
}
```

Tuttavia, questo non è vero per `onStart` . Sebbene l'infrastruttura `Observable` garantisca che venga chiamata al più una volta su ciascun `Subscriber` , la chiamata alla `request(1)` può far scattare immediatamente l'emissione di un elemento. Se si ha una logica di inizializzazione dopo la chiamata alla `request(1)` che è necessaria per `onNext` , si può finire con eccezioni:

```
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
});
```

In questo caso sincrono, una `NullPointerException` verrà lanciata immediatamente mentre sta ancora eseguendo `onStart` . Un bug più sottile si verifica se la chiamata alla `request(1)` innesca una chiamata asincrona a `onNext` su qualche altro thread e il `name` lettura in `onNext` gare lo scrive in `onStart` `request` post- `onStart` .

Pertanto, si dovrebbe eseguire l'inizializzazione di tutti i campi in `onStart` o anche prima e chiamare la `request()` ultimo. Le implementazioni di `request()` negli operatori assicurano la corretta relazione happening-before (o in altri termini, rilascio di memoria o recinzione completa) quando

necessario.

Gli operatori di onBackpressureXXX

La maggior parte degli sviluppatori incontra la contropressione quando la loro applicazione fallisce con `MissingBackpressureException` e l'eccezione di solito punta all'operatore `observeOn` . La causa effettiva è solitamente l'uso non `PublishSubject` di `PublishSubject` , `timer()` o `interval()` o operatori personalizzati creati tramite `create()` .

Esistono diversi modi per gestire tali situazioni.

Aumentando le dimensioni del buffer

A volte tali traboccamenti si verificano a causa di fonti esplosive. All'improvviso, l'utente tocca lo schermo troppo velocemente e `observeOn` buffer interno di 16 elementi predefinito su overflow Android.

La maggior parte degli operatori sensibili alla retropressione nelle versioni recenti di RxJava ora consente ai programmatori di specificare la dimensione dei loro buffer interni. I parametri rilevanti sono solitamente chiamati `bufferSize` , `prefetch` o `capacityHint` . Dato l'esempio traboccante nell'introduzione, possiamo semplicemente aumentare la dimensione del buffer di `observeOn` per avere spazio sufficiente per tutti i valori.

```
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);
}
```

Si noti tuttavia che, in generale, questa può essere solo una correzione temporanea poiché l'overflow può ancora verificarsi se l'origine del prodotto supera la dimensione del buffer prevista. In questo caso, è possibile utilizzare uno dei seguenti operatori.

Valori batch / saltati con operatori standard

Nel caso in cui i dati di origine possano essere elaborati in modo più efficiente in batch, è possibile ridurre la probabilità di `MissingBackpressureException` utilizzando uno degli operatori di batch standard (per dimensione e / o tempo).

```
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);
}
```

Se alcuni dei valori possono essere tranquillamente ignorati, è possibile utilizzare il campionamento (con il tempo o un altro Osservabile) e gli operatori di limitazione (`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);
}
```

Si noti che hovewer questi operatori riducono solo il tasso di ricezione del valore da parte del downstream e quindi possono ancora portare a `MissingBackpressureException` .

onBackpressureBuffer ()

Questo operatore nella sua forma senza parametri reintroduce un buffer illimitato tra l'origine upstream e l'operatore downstream. Essere illimitati significa che finché la JVM non esaurisce la memoria, può gestire quasi tutte le quantità provenienti da una fonte esplosiva.

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

In questo esempio, `observeOn` una dimensione del buffer molto bassa, ma non c'è `MissingBackpressureException` poiché `onBackpressureBuffer` assorbe tutti i 1 milione di valori e passa a piccoli lotti di esso per `observeOn` .

Si noti tuttavia che `onBackpressureBuffer` consuma la propria origine in modo illimitato, cioè senza applicare alcuna contropressione ad esso. Ciò ha come conseguenza che anche una sorgente che supporta la contropressione come l' `range` sarà completamente realizzata.

Ci sono 4 sovraccarichi aggiuntivi di `onBackpressureBuffer`

onBackpressureBuffer (capacità int)

Questa è una versione limitata che segnala `BufferOverflowError` nel caso in cui il buffer raggiunga la capacità specificata.

```
Observable.range(1, 1_000_000)
```

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

La rilevanza di questo operatore è in diminuzione poiché sempre più operatori consentono ora di impostare le dimensioni del buffer. Per il resto, questo dà l'opportunità di "estendere il loro buffer interno" avendo un numero maggiore con `onBackpressureBuffer` rispetto al loro valore predefinito.

onBackpressureBuffer (int capacity, Action0 onOverflow)

Questo overload richiama un'azione (condivisa) nel caso in cui si verifichi un overflow. La sua utilità è piuttosto limitata in quanto non vi sono altre informazioni fornite sull'overflow rispetto allo stack di chiamate corrente.

onBackpressureBuffer (int capacity, Action0 onOverflow, BackpressureOverflow.Strategy strategy)

Questo sovraccarico è in realtà più utile in quanto consente di definire cosa fare nel caso in cui la capacità sia stata raggiunta. `BackpressureOverflow.Strategy` è in realtà un'interfaccia ma la classe `BackpressureOverflow` offre 4 campi statici con implementazioni di esso che rappresentano azioni tipiche:

- `ON_OVERFLOW_ERROR` : questo è il comportamento predefinito dei due overload precedenti, segnalando `BufferOverflowException`
- `ON_OVERFLOW_DEFAULT` : attualmente è uguale a `ON_OVERFLOW_ERROR`
- `ON_OVERFLOW_DROP_LATEST` : se dovesse verificarsi un overflow, il valore corrente verrà semplicemente ignorato e solo i vecchi valori verranno consegnati una volta che le richieste downstream.
- `ON_OVERFLOW_DROP_OLDEST` : rilascia l'elemento più vecchio nel buffer e aggiunge il valore corrente ad esso.

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

Si noti che le ultime due strategie causano discontinuità nel flusso mentre eliminano gli elementi. Inoltre, non segnalano `BufferOverflowException` .

onBackpressureDrop ()

Ogni volta che il downstream non è pronto a ricevere i valori, questo operatore lo farà cadere dalla sequenza. Si può pensare a una capacità 0 `onBackpressureBuffer` con la strategia `ON_OVERFLOW_DROP_LATEST` .

Questo operatore è utile quando si possono tranquillamente ignorare i valori da una sorgente (come mosse del mouse o segnali di posizione GPS correnti) poiché ci saranno più valori

aggiornati in seguito.

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

Può essere utile in congiunzione con l' `interval()` operatore sorgente `interval()` . Ad esempio, se si desidera eseguire un'attività di background periodica ma ciascuna iterazione può durare più a lungo del periodo, è possibile rilasciare la notifica di intervallo in eccesso in quanto ci sarà più tardi su:

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

Esiste un overload di questo operatore: `onBackpressureDrop(Action1<? super T> onDrop)` dove viene chiamata l'azione (condivisa) con il valore che viene eliminato. Questa variante consente di ripulire i valori stessi (ad esempio, rilasciando risorse associate).

onBackpressureLatest ()

L'operatore finale mantiene solo l'ultimo valore e praticamente sovrascrive i valori meno recenti e non consegnati. Si può pensare a questa come una variante di `onBackpressureBuffer` con una capacità di 1 e una strategia di `ON_OVERFLOW_DROP_OLDEST` .

A differenza di `onBackpressureDrop` c'è sempre un valore disponibile per il consumo se il downstream è in ritardo. Questo può essere utile in alcune situazioni simili alla telemetria in cui i dati possono presentarsi in qualche schema di raffica, ma solo l'ultima è interessante per l'elaborazione.

Ad esempio, se l'utente fa clic molto sullo schermo, vorremmo comunque rispondere al suo ultimo input.

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

L'uso di `onBackpressureDrop` in questo caso porterebbe a una situazione in cui l'ultimo clic viene rilasciato e lascia all'utente chiedersi perché la logica di business non è stata eseguita.

Creazione di origini dati retropresse

La creazione di origini dati backpressured è il compito relativamente più semplice quando si ha a che fare con la backpressure in generale, poiché la libreria offre già metodi statici su `Observable` che gestiscono la contropressione per lo sviluppatore. Possiamo distinguere due tipi di metodi di

fabbrica: "generatori" freddi che restituiscono e generano elementi basati sulla domanda a valle e "spacciatori" a caldo che di solito collegano fonti di dati non reattivi e / o non contropressurabili e sovrappongono la gestione della contropressione in cima a loro.

appena

La sorgente di backpressure più elementare è creata `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
}
```

Dal momento che non richiediamo esplicitamente in `onStart` , questo non stampa nulla. è `just` fantastico quando c'è un valore costante in cui vorremmo avviare una sequenza.

Purtroppo, `just` viene spesso scambiato per un modo per calcolare dinamicamente qualcosa per essere consumato da `Subscriber` s:

```
int counter;

int computeValue() {
   return ++counter;
}

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

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

Sorprendente per alcuni, questa stampa 1 due volte invece di stampare 1 e 2 rispettivamente. Se la chiamata viene riscritta, diventa ovvio il motivo per cui funziona così:

```
int temp = computeValue();

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

Il `computeValue` viene chiamato come parte della routine principale e non in risposta agli abbonati che sottoscrivono.

fromCallable

Ciò di cui le persone hanno effettivamente bisogno è il metodo `fromCallable` :

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

Qui il `computeValue` viene eseguito solo quando un sottoscrittore sottoscrive e per ciascuno di essi, stampando l'1 e il 2 previsti. Naturalmente, anche `fromCallable` supporta correttamente la contropressione e non emetterà il valore calcolato se non richiesto. Si noti comunque che il calcolo avviene comunque. Nel caso in cui il calcolo stesso debba essere ritardato fino a quando il downstream richiede effettivamente, possiamo usare `just` con la `map` :

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

`just` non emetterà il suo valore costante fino a quando richiesto quando è mappato al risultato del valore `computeValue` , ancora chiamato singolarmente per ogni utente.

a partire dal

Se i dati sono già disponibili come una matrice di oggetti, un elenco di oggetti o qualsiasi sorgente `Iterable` , i rispettivi `from` sovraccarichi gestiranno la contropressione e l'emissione di tali fonti:

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

Per comodità (ed evitando gli avvertimenti sulla creazione di array generici) ci sono da 2 a 10 sovraccarichi di argomenti `just` per delegare internamente a `from` .

Il `from(Iterable)` offre anche un'opportunità interessante. Molte generazioni di valore possono essere espresse in una forma di macchina di stato. Ogni elemento richiesto attiva una transizione di stato e il calcolo del valore restituito.

La scrittura di macchine di stato come `Iterable` s è alquanto complicata (ma ancora più semplice della scrittura di un `Observable` per consumarla) ea differenza del C #, Java non ha alcun supporto dal compilatore per costruire tali macchine di stato semplicemente scrivendo codice di aspetto classico (con `yield return` e `yield break`). Alcune librerie offrono qualche aiuto, come `AbstractIterable` Google Guava e `Ix.generate()` e `Ix.forloop()` . Questi sono di per sé degni di una serie completa, quindi vediamo una sorgente `Iterable` molto semplice che ripete indefinitamente un valore costante:

```
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);
```

Se consumassimo l' `iterator` tramite il classico ciclo for, ciò comporterebbe un ciclo infinito. Dal momento che costruiamo un `Observable` fuori da esso, possiamo esprimere la nostra volontà di consumare solo i primi 5 di esso e quindi smettere di richiedere qualsiasi cosa. Questo è il vero potere di valutare e calcolare pigramente dentro s `Observable` .

creare (SyncOnSubscribe)

A volte, l'origine dei dati da convertire nel mondo reattivo stesso è sincrona (bloccante) e pull-like, vale a dire, dobbiamo chiamare qualche metodo `get` o `read` per ottenere il prossimo pezzo di dati. Si potrebbe, ovviamente, trasformarlo in un `Iterable` ma quando tali risorse sono associate a risorse, potremmo perdere tali risorse se il downstream annulla la sequenza prima che finisca.

Per gestire tali casi, RxJava ha la classe `SyncOnSubscribe` . Si può estenderlo e implementare i suoi metodi o usare uno dei suoi metodi di fabbrica basati su lambda per costruire un'istanza.

```
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);
```

In generale, `SyncOnSubscribe` utilizza 3 callback.

I primi callback consentono di creare uno stato per sottoscrittore, come `FileInputStream` nell'esempio; il file verrà aperto indipendentemente per ogni singolo utente.

Il secondo callback accetta questo oggetto stato e fornisce un `Observer` output i cui metodi `onXXX` possono essere chiamati per emettere valori. Questo callback viene eseguito tante volte quanto richiesto dal downstream. Ad ogni chiamata, deve chiamare `onNext` Al più al massimo una volta facoltativamente seguito da `onError` o `onCompleted` . Nell'esempio chiamiamo `onCompleted()` se il byte letto è negativo, indicante e fine del file, e chiama `onError` nel caso in cui il read lanci una

`IOException` .

Il callback finale viene richiamato quando il downstream si annulla (chiudendo l'inputstream) o quando il callback precedente ha chiamato i metodi del terminale; consente di liberare risorse. Dal momento che non tutte le fonti hanno bisogno di tutte queste funzionalità, i metodi statici di `SyncOnSubscribe` consentono di creare istanze senza di esse.

Sfortunatamente, molte chiamate di metodo attraverso la JVM e altre librerie generano eccezioni controllate e devono essere racchiuse in esecuzioni `try-catch` quanto le interfacce funzionali utilizzate da questa classe non consentono il lancio di eccezioni controllate.

Naturalmente, possiamo imitare altre fonti tipiche, ad esempio un intervallo illimitato con esso:

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

In questa configurazione, la `current` inizia con `0` e la volta successiva che viene richiamata la lambda, il parametro `current` ora contiene `1` .

Esiste una variante di `SyncOnSubscribe` chiamata `AsyncOnSubscribe` che sembra abbastanza simile con l'eccezione che il callback centrale richiede anche un valore lungo che rappresenta l'importo della richiesta da downstream e la callback dovrebbe generare un `Observable` con la stessa lunghezza esatta. Questa fonte quindi concatena tutti questi `Observable` in una singola sequenza.

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

C'è una discussione (accesa) in corso sull'utilità di questa classe e generalmente non raccomandata perché rompe abitualmente le aspettative su come effettivamente emetterà quei valori generati e su come risponderà, o anche quale tipo di valori di richiesta riceverà in scenari consumer più complessi.

creare (emettitore)

A volte, la sorgente da includere in un `Observable` è già calda (come le mosse del mouse) o fredda ma non è backpressurable nella sua API (come un callback di rete asincrono).

Per gestire casi del genere, una versione recente di RxJava ha introdotto il metodo factory `create(emitter)` . Ci vogliono due parametri:

- un callback che verrà chiamato con un'istanza dell'interfaccia `Emitter<T>` per ciascun utente in entrata,
- un'enumerazione `Emitter.BackpressureMode` che impone allo sviluppatore di specificare il comportamento di contropressione da applicare. Ha le solite modalità, simili a `onBackpressureXXX` oltre a segnalare una `MissingBackpressureException` o semplicemente ignorare del tutto l'overflow al suo interno.

Si noti che al momento non supporta parametri aggiuntivi per tali modalità di contropressione. Se uno ha bisogno di quelli di personalizzazione, utilizzando `NONE` come modalità di contropressione e applicando le pertinenti `onBackpressureXXX` sul risultante `Observable` è la strada da percorrere.

Il primo caso tipico per il suo utilizzo quando si desidera interagire con una fonte basata su push, come gli eventi della GUI. Queste API presentano alcune forme di chiamate `addListener` / `removeListener` che è possibile utilizzare:

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

    button.addActionListener(al);

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

}, BackpressureMode.BUFFER);
```

L' `Emitter` è relativamente semplice da usare; si può chiamare su `onNext` , su `onError` e su `onCompleted` su di esso e l'operatore gestisce da solo la gestione della contropressione e della cancellazione. Inoltre, se l'API `setCancellation` supporta la cancellazione (come la rimozione del listener nell'esempio), è possibile utilizzare `setCancellation` (o `setSubscription` per le risorse simili a `Subscription`) per registrare un callback di annullamento che viene richiamato quando l'annullamento di sottoscrizione o l' `onError` / `onCompleted` viene chiamato `onCompleted` `Emitter` fornita.

Questi metodi consentono di associare una sola risorsa all'emettitore alla volta e impostarne una nuova annulla automaticamente la vecchia. Se si devono gestire più risorse, creare una `CompositeSubscription` , associarla all'emettitore e quindi aggiungere ulteriori risorse alla stessa `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);
```

Il secondo scenario di solito prevede alcune API asincrone basate su callback che devono essere convertite in `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);
```

In questo caso, la delega funziona allo stesso modo. Sfortunatamente, di solito queste classiche API di callback non supportano la cancellazione, ma se lo fanno, si può impostare la cancellazione proprio come negli esempi precedenti (con un modo forse più complesso). Si noti l'uso del `LATEST` modalità contropressione; se sappiamo che ci sarà solo un singolo valore, non abbiamo bisogno della strategia `BUFFER` poiché assegna un buffer lungo 128 elementi predefinito (che cresce come necessario) che non sarà mai completamente utilizzato.

Leggi Contropressione online: https://riptutorial.com/it/rx-java/topic/2341/contropressione

Capitolo 4: operatori

Osservazioni

Questo documento descrive il comportamento di base di un operatore.

Examples

Operatori, un'introduzione

Un operatore può essere utilizzato per manipolare il flusso di oggetti da `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);
```

L'output sarebbe:

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

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

L'operatore della `map` cambiato il numero `Integer` osservabile in una `String` osservabile, manipolando il flusso degli oggetti.

Concatenamento dell'operatore

Più operatori possono essere `chained` insieme per trasformazioni e manipolazioni più potenti.

```
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
```

È possibile aggiungere un numero qualsiasi di operatori tra `Observable` e `Subscriber` .

operatore flatMap

L'operatore `flatMap` ti aiuta a trasformare un evento in un altro `Observable` (o trasformare un evento in zero, uno o più eventi).

È un operatore perfetto quando vuoi chiamare un altro metodo che restituisce 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` serializzare `perform` **abbonamenti, ma** gli eventi emited da `perform` non può essere ordinato. Pertanto, è possibile ricevere eventi emessi dall'ultima chiamata di esecuzione **prima degli** eventi dalla prima chiamata di `perform` (è necessario utilizzare invece `concatMap`).

Se crei un altro `Observable` nel tuo abbonato, **dovresti** usare `flatMap` . L'idea principale è: **non lasciare mai l'osservabile**

Per esempio :

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

può essere facilmente sostituito da:

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

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

filtro Operatore

È possibile utilizzare l'operatore `filter` per filtrare gli elementi dal flusso dei valori in base a un risultato di un metodo di predicati.

In altre parole, gli elementi che passano dall'Observer al Sottoscrittore verranno scartati in base al `filter` Function pass, se la funzione restituisce `false` per un determinato valore, tale valore verrà filtrato.

Esempio:

```
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
    });
```

Questo codice verrà stampato

```
0.0
4.0
16.0
36.0
64.0
```

mappa Operatore

È possibile utilizzare la `map` all'operatore di mappare i valori di un flusso di valori diversi sulla base del risultato per ogni valore dalla funzione passata a `map` . Il flusso di risultati è una nuova copia e non modificherà il flusso di valori fornito, il flusso di risultati avrà la stessa lunghezza del flusso di input ma potrebbe essere di tipi diversi.

La funzione passata a `.map()` , deve restituire un valore.

Esempio:

```
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
```

```
    });
```

Questo codice verrà stampato

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

In questo esempio Observable accetta un `List<Integer>` l'elenco verrà trasformato in un `List<String>` nella pipeline e il `.subscribe` emetterà `String` 's

Operatore doOnNext

`doOnNext` operatore `doOnNext` chiamato ogni volta quando Source `Observable` emette un elemento. Può essere utilizzato per scopi di debug, applicando alcune azioni all'elemento emesso, alla registrazione, ecc ...

```
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();
```

Nell'esempio seguente `doOnNext` non viene mai chiamato perché l' `Observable` origine non emette nulla perché `Observable.empty()` chiama `onCompleted` dopo la sottoscrizione.

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

ripetere operatore

`repeat` operatore di `repeat` consente di ripetere l'intera sequenza dalla sorgente `Observable` .

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

Uscita dell'esempio sopra

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

Questa sequenza ripete un numero infinito di volte e non completa mai.

Per ripetere il numero finito di volte in sequenza basta passare il numero intero come argomento per `repeat` operatore.

```
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")
    );
```

Questo esempio stampa

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

È molto importante capire che l'operatore di `repeat` iscrive nuovamente alla fonte `Observable` quando la sequenza `Observable` fonte completa. Riscriveremo l'esempio sopra 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")
        );
```

Questo esempio stampa

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

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

Quando si utilizza operatore concatenamento è importante sapere che `repeat` operatore ripete **intera sequenza** piuttosto che precede operatore.

```
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")
        );
```

Questo esempio stampa

```
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
```

Questo esempio mostra che `repeat` operatore ripete intera sequenza reiscriversi alle `Observable` invece di ripetere ultima `map` operatore e non importa in quale posizione nella sequenza `repeat` operatore usato.

Questa sequenza

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

```
        /*....*/
    );
```

è uguale a questa sequenza

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

Leggi operatori online: https://riptutorial.com/it/rx-java/topic/2316/operatori

Capitolo 5: Osservabile

Examples

Crea un osservabile

Esistono diversi modi per creare un Observable in RxJava. Il modo più efficace è utilizzare il metodo `Observable.create` . Ma è anche il modo più **complicato** . Quindi devi **evitare di usarlo** , per quanto possibile.

Emissione di un valore eccitante

Se hai già un valore, puoi utilizzare `Observable.just` per emettere il tuo valore.

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

Emissione di un valore che dovrebbe essere calcolato

Se si desidera emettere un valore che non è già calcolato, o che può richiedere molto tempo per essere calcolato, è possibile utilizzare `Observable.fromCallable` per emettere il valore successivo.

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

`longComputation()` sarà chiamato solo quando ti iscrivi al tuo `Observable` . In questo modo, il calcolo sarà *pigro* .

Modo alternativo per emettere un valore che dovrebbe essere calcolato

`Observable.defer` costruisce un `Observable` come `Observable.fromCallable` ma viene utilizzato quando è necessario restituire un `Observable` anziché un valore. È utile quando vuoi gestire gli errori nella tua chiamata.

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

Osservabili caldi e freddi

Gli osservabili sono generalmente classificati come `Hot` o `Cold` , a seconda del loro comportamento di emissione.
Un `Cold Observable` è uno che inizia a emettere su richiesta (abbonamento), mentre un `Hot Observable` è uno che emette indipendentemente dalle sottoscrizioni.

Osservabile a freddo

```
/* 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
```

L'output del codice sopra sembra (può variare):

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

Si noti che anche se `sub2` inizia tardi, riceve valori dall'inizio. Per concludere, un `Cold Observable` emette solo gli articoli quando richiesto. Più richiesta avvia più pipeline.

Osservabile caldo

Nota: le osservabili Hot emettono valori indipendenti dalle singole sottoscrizioni. Hanno la loro sequenza temporale e gli eventi si verificano se qualcuno sta ascoltando o meno.

Un `Cold Observale` può essere convertito in `Hot Observable` con una semplice `publish` .

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

`publish` restituisce un `ConnectableObservable` che aggiunge funzionalità per *connettersi* e *disconnettersi* dall'osservabile.

```
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));
```

I suddetti rendimenti:

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

Notare che anche se `sub2` inizia a osservare in ritardo, è sincronizzato con `sub1` .
La disconnessione è un po 'più complicata! La disconnessione avviene `Subscription` e non `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();
```

Quanto sopra produce:

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

Dopo la disconnessione, l' `Observable` essenzialmente "termina" e si riavvia quando viene aggiunto un nuovo abbonamento.

`Hot Observable` può essere usato per creare un `EventBus` . Tali Eventbus sono generalmente leggeri e super veloci. L'unico lato negativo di un RxBus è che tutti gli eventi devono essere implementati manualmente e passati al bus.

Leggi Osservabile online: https://riptutorial.com/it/rx-java/topic/1418/osservabile

Capitolo 6: Retrofit e RxJava

Examples

Imposta Retrofit e RxJava

Retrofit2 include il supporto per molteplici meccanismi di esecuzione pluggable, uno dei quali è RxJava.

Per utilizzare il retrofit con RxJava è necessario prima aggiungere l'adattatore Retrofit RxJava al progetto:

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

quindi è necessario aggiungere l'adattatore durante la creazione dell'istanza di modifica successiva:

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

Nella tua interfaccia quando definisci l'API il tipo di `Observable` dovrebbe essere `Observable` ad esempio:

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

Puoi anche usare `Single` invece di `Observable` .

Fare richieste seriali

RxJava è utile quando si effettua una richiesta seriale. Se si desidera utilizzare il risultato di una richiesta per `flatMap` un altro, è possibile utilizzare l'operatore `flatMap` :

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

Fare richieste parallele

Puoi usare l'operatore `zip` per fare richieste in parallelo e combinare i risultati ad esempio:

```
Observable.zip(api.getRepo(repoId1), api.getRepo(repoId2), (repo1, repo2) ->
    {
        //here you can combine the results
```

```
    }).subscribe(/*do something with the result*/);
```

Leggi Retrofit e RxJava online: https://riptutorial.com/it/rx-java/topic/2950/retrofit-e-rxjava

Capitolo 7: RxJava2 Flowable and Subscriber

introduzione

Questo argomento mostra esempi e documentazione relativi ai concetti reattivi di Flowable e Subscriber introdotti in rxjava versione2

Osservazioni

l'esempio ha bisogno di rxjava2 come dipendenza, le coordinate di maven per la versione utilizzata sono:

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

Examples

esempio di consumatore produttore con supporto per la contropressione nel produttore

Il `TestProducer` di questo esempio produce oggetti `Integer` in un determinato intervallo e li spinge al suo `Subscriber` . Estende la classe `Flowable<Integer>` . Per un nuovo sottoscrittore, crea un oggetto `Subscription` cui metodo `request(long)` viene utilizzato per creare e pubblicare i valori Integer.

È importante per l' `Subscription` che viene passato al `subscriber` che il metodo `request()` che chiama `onNext()` sul `onNext()` sottoscrizione possa essere chiamato in modo ricorsivo da questa chiamata a `onNext()` . Per impedire uno stack overflow, l'implementazione mostrata utilizza il contatore `outStandingRequests` e il flag `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;
            }
        });
    }
}
```

Il consumatore in questo esempio estende `DefaultSubscriber<Integer>` e all'avvio e dopo aver consumato un intero richiede il successivo. Al momento di consumare i valori Integer, c'è un piccolo ritardo, quindi la contropressione verrà creata per il produttore.

```
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");
    }
}
```

nel seguente metodo principale di una classe di test vengono creati e cablati il produttore e il consumatore:

```
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();
    }
}
```

Quando si esegue l'esempio, il file di registro mostra che il consumatore viene eseguito continuamente, mentre il produttore diventa attivo solo quando è necessario riempire il buffer interno Flowable di rxjava2.

Leggi RxJava2 Flowable and Subscriber online: https://riptutorial.com/it/rx-java/topic/9810/rxjava2-flowable-and-subscriber

Capitolo 8: schedulatori

Examples

Esempi di base

Gli scheduler sono un'astrazione RxJava sull'unità di elaborazione. Un programma di pianificazione può essere supportato da un servizio Executor, ma è possibile implementare la propria implementazione dello scheduler.

Uno `Scheduler` dovrebbe soddisfare questo requisito:

- Dovrebbe elaborare in sequenza l'attività non ritardata (ordine FIFO)
- L'attività può essere ritardata

Un `Scheduler` può essere utilizzato come parametro in alcuni operatori (esempio: `delay`) o utilizzato con il metodo `subscribeOn` / `observeOn` .

Con qualche operatore, lo `Scheduler` verrà utilizzato per elaborare l'attività dell'operatore specifico. Ad esempio, il `delay` programmerà un'attività ritardata che emetterà il valore successivo. Questo è uno `Scheduler` che lo manterrà ed eseguirà in seguito.

Il `subscribeOn` può essere utilizzato una volta per `Observable` . Definirà in quale `Scheduler` il codice dell'abbonamento sarà eseguito.

`observeOn` può essere utilizzato più volte per `Observable` . `observeOn` in quale `Scheduler` verrà utilizzato per eseguire tutte le attività definite **dopo** il metodo `observeOn` . `observeOn` ti aiuterà a eseguire il thread hopping.

iscriviti a uno specifico programmatore

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

observOn con uno Scheduler specifico

```
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()));
```

Specifica di uno Schedulatore specifico con un operatore

Alcuni operatori possono prendere uno `Scheduler` come parametro.

```
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);
```

Pubblica nell'iscritto:

```
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 di thread:

```
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();
```

Leggi schedulatori online: https://riptutorial.com/it/rx-java/topic/2321/schedulatori

Capitolo 9: Soggetti

Sintassi

- Soggetto <T, R> subject = AsyncSubject.create (); // AsyncSubject predefinito
- Soggetto <T, R> subject = BehaviorSubject.create (); // Default BehaviorSubject
- Soggetto <T, R> subject = PublishSubject.create (); // Default PublishSubject
- Soggetto <T, R> subject = ReplaySubject.create (); // Default ReplaySubject
- mySafeSubject = new SerializedSubject (unSafeSubject); // Converti un unsafeSubject in un safeSubject - generalmente per soggetti multi-threaded

Parametri

parametri	Dettagli
T	Tipo di input
R	Tipo di uscita

Osservazioni

Questa documentazione fornisce dettagli e spiegazioni `Subject` . Per ulteriori informazioni e ulteriori letture, si prega di visitare la documentazione ufficiale .

Examples

Soggetti di base

Un `Subject` in RxJava è una classe che è sia un `Observable` che un `Observer` . Ciò significa fondamentalmente che può agire come `Observable` e passare input agli abbonati e come `Observer` per ottenere input da un altro Observable.

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

Le stampe sopra "Hello, World!" consolare usando `Subjects` .

Spiegazione

1. La prima riga di codice definisce un nuovo `Subject` di tipo `PublishSubject`

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

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

2. La seconda riga si abbona al soggetto, mostrando il comportamento `Observer` .

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

   Ciò consente al `Subject` di prendere input come un normale abbonato

3. La terza riga chiama il metodo `onNext` dell'oggetto, mostrando il comportamento `Observable` .

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

   Ciò consente al `Subject` di fornire input a tutti coloro che lo sottoscrivono.

tipi

Un `Subject` (in RxJava) può essere di uno di questi quattro tipi:

- AsyncSubject
- BehaviorSubject
- PublishSubject
- ReplaySubject

Inoltre, un `Subject` può essere di tipo `SerializedSubject` . Questo tipo garantisce che il `Subject` non violi il *Contratto Osservabile* (che specifica che tutte le chiamate devono essere serializzate)

Ulteriori letture:

- Utilizzare o non usare Subject dal blog di Dave Sexton

PublishSubject

`PublishSubject` emette su un `Observer` solo quegli elementi che sono emessi dalla `Observable` origine successivamente al momento dell'abbonamento.

Un semplice esempio `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);
```

Produzione:

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

Nell'esempio precedente, `PublishSubject` iscrive a un `Observable` che agisce come un orologio ed emette elementi (Long) ogni 500 milli secondi. Come visto nell'output, `PublishSubject` passa i `PublishSubject` che ottiene dalla sorgente (`clock`) ai suoi abbonati (`sub1` e `sub2`).

Un `PublishSubject` può iniziare a emettere elementi non appena viene creato, senza alcun osservatore, che corre il rischio di perdere uno o più oggetti fino a quando un osservatore può oscurarsi.

```
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
```

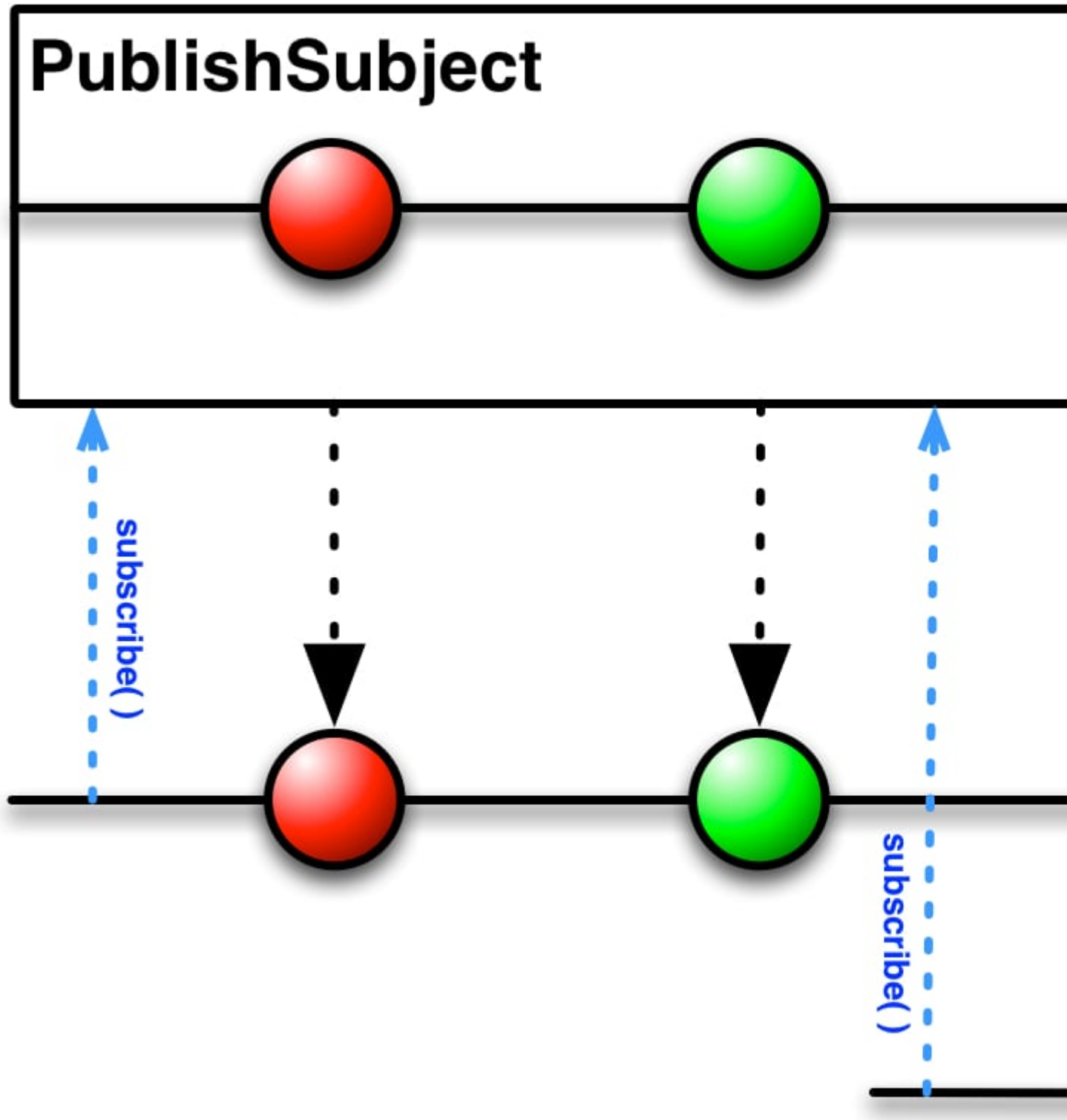
Produzione:

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

Si noti che `sub1` emette valori a partire da `10` . Il ritardo di 5 secondi introdotto ha causato una *perdita* di articoli. Questi non possono essere riprodotti. Questo rende essenzialmente `PublishSubject` un `Hot Observable` .

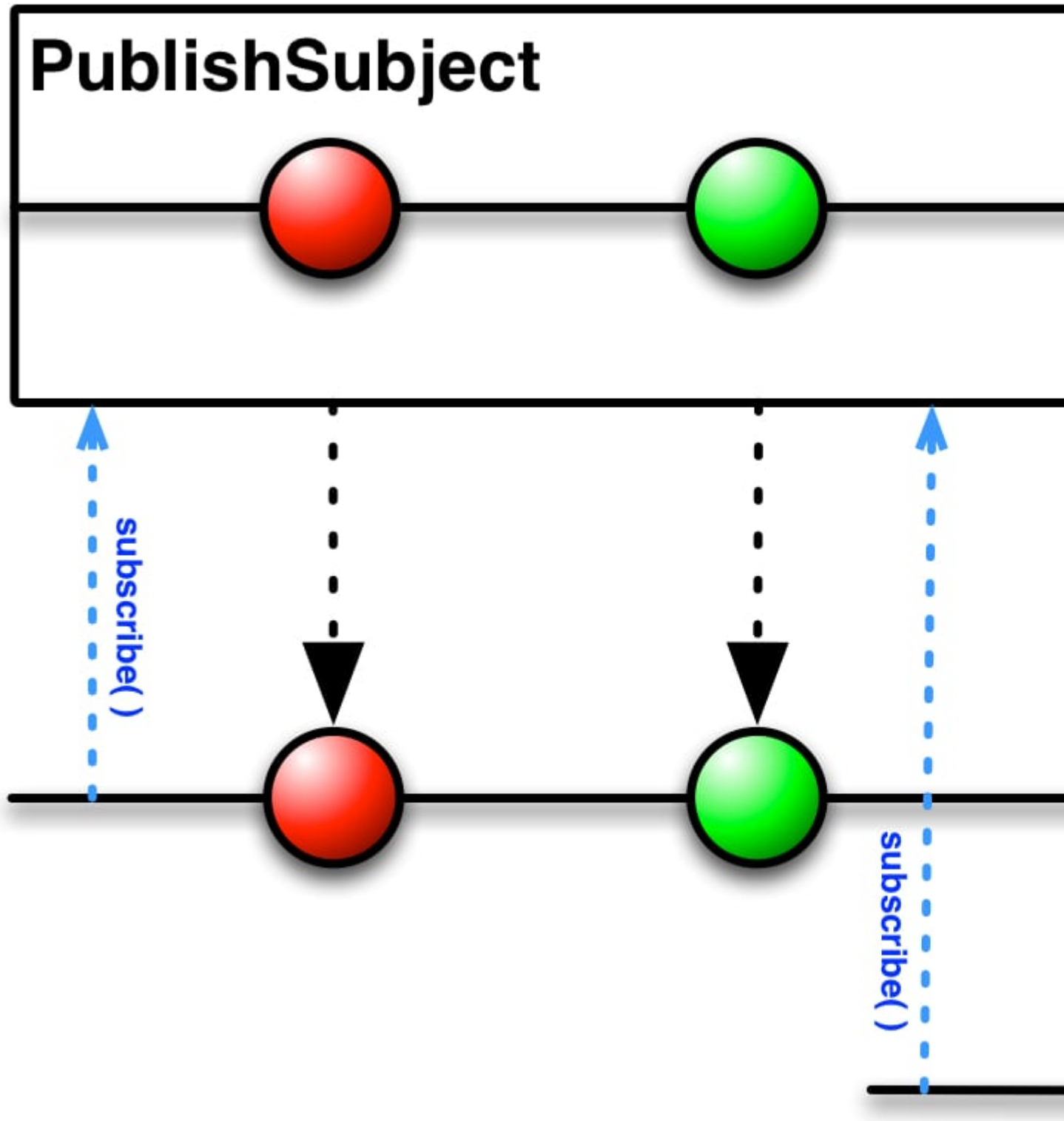
Si noti inoltre che se un osservatore si iscrive a `PublishSubject` dopo aver emesso ***n*** elementi, questi ***n*** elementi *non possono* essere riprodotti per questo osservatore.

Di seguito è riportato il diagramma di marmo di `PublishSubject`



`PublishSubject` emette elementi a tutti quelli che si sono iscritti, in qualsiasi momento prima che `onCompleted` l' `onCompleted` della `Observable` origine.

Se l' `Observable` origine termina con un errore, `PublishSubject` non emetterà alcun elemento agli osservatori successivi, ma passerà semplicemente la notifica di errore dalla Observable di origine.



Caso d'uso

Supponiamo che tu voglia creare un'applicazione che monitori i prezzi delle azioni di una certa azienda e inoltrala a tutti i clienti che ne fanno richiesta.

```
/* 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();
```

Nell'esempio di utilizzo sopra riportato, `PublishSubject` funge da bridge per trasferire i valori dal tuo server a tutti i client che si abbonano al tuo `watcher` .

Ulteriori letture:

- PublishSubject javadocs
- Blog di Thomas Nield (Lettura avanzata)

Leggi Soggetti online: https://riptutorial.com/it/rx-java/topic/3287/soggetti

Capitolo 10: Test unitario

Osservazioni

Poiché tutti i metodi Schedulers sono statici, i test delle unità che utilizzano gli hook RxJava non possono essere eseguiti in parallelo sulla stessa istanza JVM. Se loro dove, un TestScheduler verrebbe rimosso nel bel mezzo di un test unitario. Questo è fondamentalmente il lato negativo dell'uso della classe Schedulers.

Examples

TestSubscriber

TestSubscribers ti consente di evitare che il lavoro crei il tuo Subscriber o abbonati all'azione <?> Per verificare che alcuni valori siano consegnati, quanti ne siano, se l'Observable sia completato, sia stata sollevata un'eccezione e molto altro ancora.

Iniziare

Questo esempio mostra solo un'asserzione secondo cui i valori 1,2,3 e 4 sono passati all'Osservable tramite onNext.

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

`assertValues` afferma che il conteggio è corretto. Se dovessi passare solo alcuni dei valori, l'asserzione fallirebbe.

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

`assertValues` usa il metodo `equals` quando fa affermazioni. Ciò consente di testare facilmente classi considerate come dati.

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

Questo esempio mostra una classe che ha un uguaglianza definita e che asserisce i valori dall'osservabile.

```
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
```

Inoltre, prendi nota che utilizziamo il valore `assertValue` più `assertValue` perché dobbiamo solo controllare un articolo.

Ottenere tutti gli eventi

Se necessario, puoi anche chiedere tutti gli eventi come 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();
```

Affermando sugli eventi

Se vuoi fare test più estesi sui tuoi eventi, puoi combinare `getOnNextEvents` (o `getOn*Events`) con la tua libreria di asserzione preferita:

```
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 di `Observable#error`

Puoi assicurarti che venga emessa la classe di eccezione corretta:

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

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

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

Puoi anche assicurarti che sia stata lanciata l'eccezione esatta:

```
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 ti permette di controllare il tempo e l'esecuzione degli Osservabili invece di dover fare attese occupate, unire discussioni o qualsiasi cosa per manipolare il tempo del sistema. Questo è MOLTO importante se si desidera scrivere test unitari prevedibili, coerenti e veloci. Poiché stai manipolando il tempo, non c'è più la possibilità che un thread sia affamato, che il tuo test fallisca su una macchina più lenta o che rifiuti tempo di esecuzione occupato ad aspettare un risultato.

TestSchedulers può essere fornito tramite il sovraccarico che richiede un Utilità di pianificazione per tutte le operazioni 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
```

Il TestScheduler è piuttosto semplice. Consiste solo di tre metodi.

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

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

Questo ti consente di manipolare quando il TestScheduler deve attivare tutte le azioni relative a un certo periodo di tempo in futuro.

Mentre passa lo schedulatore funziona, questo non è il modo in cui il TestScheduler viene comunemente usato a causa di quanto sia inefficace. Passare scheduler in classi finisce per fornire un sacco di codice extra per poco guadagno. Invece, puoi collegarti a Schedulers.io () / computation () / etc di RxJava. Questo è fatto con RxJava's Hooks. Ciò consente di definire cosa viene restituito da una chiamata da uno dei metodi Pianificatori.

```
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;
    }
}
```

Questa classe consente all'utente di ottenere lo scheduler di test che verrà collegato per tutte le chiamate agli Scheduler. Un test unitario avrebbe solo bisogno di ottenere questo programmatore nella sua configurazione. Si consiglia vivamente di acquisirlo nel setup e non come un semplice campo vecchio perché il TestScheduler potrebbe tentare di attivare le azioni da un altro test unitario quando si avanza in anticipo. Ora il nostro esempio sopra diventa

```
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)
```

È così che puoi rimuovere efficacemente l'orologio di sistema dal test dell'unità (almeno per quanto riguarda RxJava))

Leggi Test unitario online: https://riptutorial.com/it/rx-java/topic/5207/test-unitario

Titoli di coda

S. No	Capitoli	Contributors
1	Iniziare 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	Contropressione	akarnokd, Bartek Lipinski, Chris A, Cristian, dwursteisen, Niklas, Sebas LG
4	operatori	dwursteisen, hello_world, svarog, Vadeg
5	Osservabile	Aki K, dwursteisen, hello_world, JonesV
6	Retrofit e RxJava	LordRaydenMK
7	RxJava2 Flowable and Subscriber	P.J.Meisch
8	schedulatori	dwursteisen, Gal Dreiman
9	Soggetti	hello_world, mavHarsha
10	Test unitario	Buttink, Sir Celsius