Kostenloses eBook

LERNEN

rx-java

Free unaffiliated eBook created from
Stack Overflow contributors.

#rx-java

Inhaltsverzeichnis

Über

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

Kapitel 1: Erste Schritte mit rx-java

Bemerkungen

Dieser Abschnitt bietet einen grundlegenden Überblick und eine oberflächliche Einführung in RX-Java.

RxJava ist eine Java VM-Implementierung von Reactive Extensions : Eine Bibliothek zum Erstellen asynchroner und ereignisbasierter Programme mithilfe von beobachtbaren Sequenzen.

Erfahren Sie mehr über RxJava auf der Wiki-Startseite .

Versionen

Ausführung	Status	Letzte stabile Version	Veröffentlichungsdatum
1.x	Stabil	1.3.0	2017-05-05
2.x	Stabil	2.1.1	2017-06-21

Examples

Installation oder Setup

rx-java eingerichtet

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. Efeu

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

4. Schnappschüsse von JFrog

```
repositories {
```

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

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

5. Wenn Sie die Gläser anstelle eines Build-Systems herunterladen müssen, erstellen Sie eine Maven- `pom` Datei mit der gewünschten Version wie `pom` :

```
<?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>
```

Dann führe aus:

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

Dieser Befehl lädt `rxjava-*.jar` und seine Abhängigkeiten in `./target/dependency/`.

Sie benötigen Java 6 oder höher.

Hallo Welt!

Folgendes druckt die Nachricht `Hello, World!` zu trösten

```
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);
       }

    });
}
```

Oder mit der Java 8-Lambda-Notation

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

Eine Einführung in RxJava

Die Kernkonzepte von RxJava sind die `Observables` und `Subscribers` . Ein `Observable` gibt Objekte aus, während ein `Subscriber` verbraucht.

Beobachtbar

`Observable` ist eine Klasse, die das reaktive Entwurfsmuster implementiert. Diese Observables bieten Methoden, mit denen Verbraucher Ereignisänderungen abonnieren können. Die Ereignisänderungen werden vom Beobachtbaren ausgelöst. Die Anzahl der Abonnenten, die ein `Observable` kann, oder die Anzahl der Objekte, die ein `Observable` ausgeben kann, ist nicht beschränkt.

Nehmen Sie zum Beispiel:

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

Hier wird ein beobachtbares Objekt namens `integerObservable` und `stringObservable` aus der Factory-Methode erstellt, die `just` von der Rx-Bibliothek bereitgestellt wird. Beachten Sie, dass `Observable` generisch ist und somit jedes Objekt ausgeben kann.

Teilnehmer

Ein `Subscriber` ist der Verbraucher. Ein `Subscriber` kann ***nur einen*** beobachtbaren abonnieren. Das `Observable` ruft die `onNext()` , `onCompleted()` und `onError()` des `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);
        }
    };
```

Beachten Sie, dass der `Subscriber` auch generisch ist und jedes Objekt unterstützen kann. Ein `Subscriber` muss das Observable abonnieren, indem er die `subscribe` Methode für das Observable aufruft.

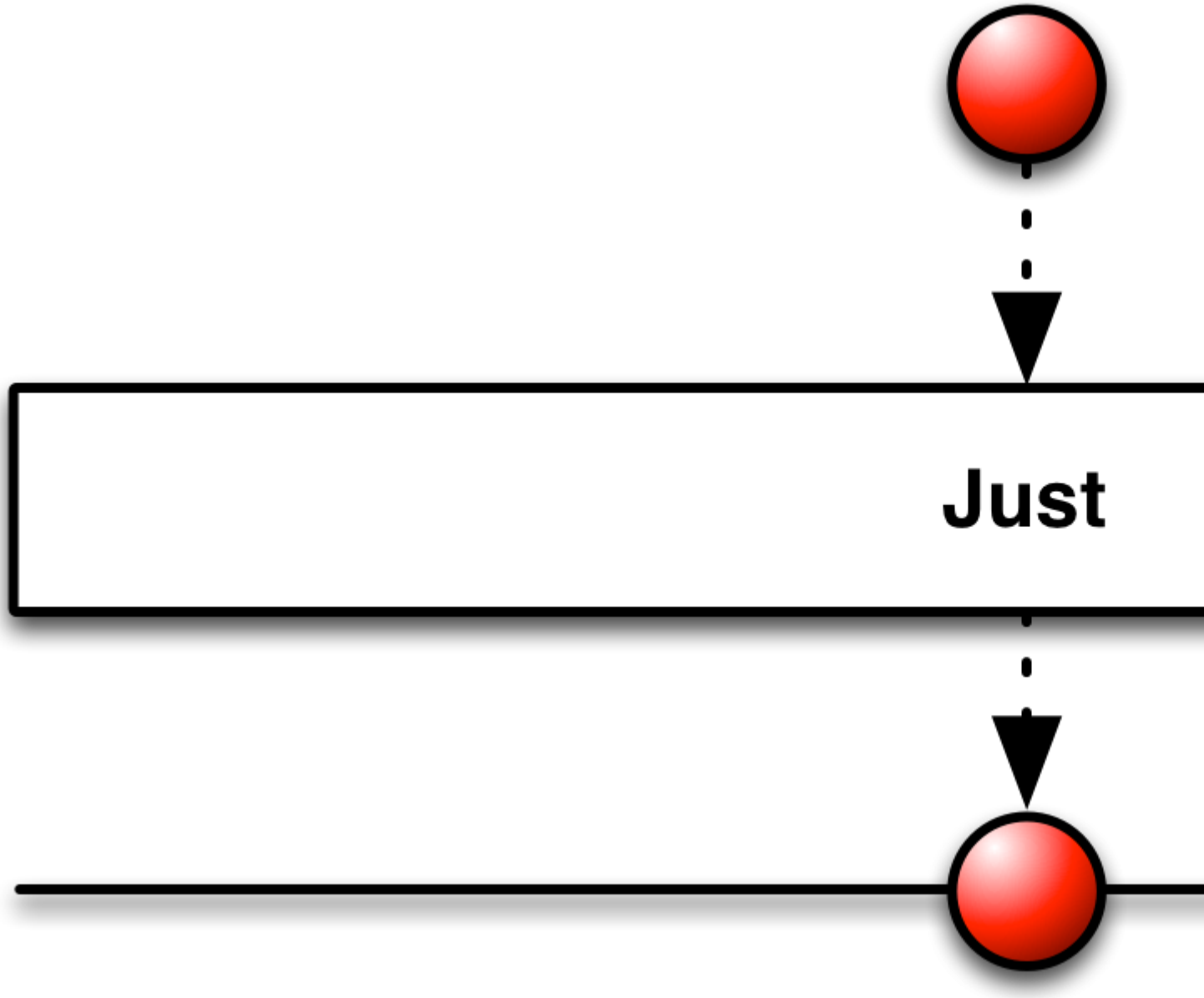
```
integerObservable.subscribe(mSubscriber);
```

Wenn das obige ausgeführt wird, wird folgende Ausgabe erzeugt:

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

Marmordiagramme verstehen

Ein Observable kann als nur ein Ereignisstrom betrachtet werden. Wenn Sie ein Observable definieren, haben Sie drei Listener: onNext, onComplete und onError. onNext wird jedes Mal aufgerufen, wenn das Observable einen neuen Wert erhält. onComplete wird aufgerufen, wenn das übergeordnete Observable mitteilt, dass es keine weiteren Werte mehr erzeugt. onError wird aufgerufen, wenn während der Ausführung der Observable-Kette eine Ausnahme ausgelöst wird. Um Operatoren in Rx anzuzeigen, wird das Marmordiagramm verwendet, um anzuzeigen, was mit einer bestimmten Operation passiert. Nachfolgend finden Sie ein Beispiel für einen einfachen beobachtbaren Operator "Just".



Marmordiagramme haben einen horizontalen Block, der die ausgeführte Operation darstellt, einen vertikalen Balken, der das abgeschlossene Ereignis darstellt, ein X, um einen Fehler darzustellen, und jede andere Form repräsentiert einen Wert. In diesem Sinne können wir sehen, dass "Just" nur unseren Wert übernimmt und onNext ausführt und dann mit onComplete endet. Es gibt viel mehr Operationen als nur "Just". Sie können alle Vorgänge, die Teil des ReactiveX-Projekts sind, und die zugehörigen Implementierungen in RxJava auf der ReativeX-Site anzeigen . Es gibt auch interaktive Marmordiagramme über die RxMarbles-Site .

Erste Schritte mit rx-java online lesen: https://riptutorial.com/de/rx-java/topic/974/erste-schritte-mit-rx-java

Kapitel 2: Android mit RxJava

Bemerkungen

RxAndroid war früher eine Bibliothek mit vielen Funktionen. Es wurde in viele verschiedene Bibliotheken aufgeteilt, die von Version 0.25.0 auf 1.x verschoben wurden.

Eine Liste der Bibliotheken, die die vor der Version 1.0 verfügbaren Funktionen implementieren, wird hier beibehalten.

Examples

RxAndroid - AndroidSchedulers

Dies ist buchstäblich das einzige, was Sie benötigen, um RxJava unter Android zu verwenden.

Beziehen Sie RxJava und RxAndroid in Ihre Abstufungsabhängigkeiten ein:

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

Der wichtigste Zusatz von RxAndroid zu RxJava ist ein Scheduler für den Android-Haupt- oder UI-Thread.

In Ihrem Code:

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

Oder Sie können einen Scheduler für einen benutzerdefinierten `Looper` erstellen:

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

Für alles andere können Sie sich auf die Standarddokumentation von RxJava beziehen.

RxLifecycle-Komponenten

Die RxLifecycle- Bibliothek erleichtert das Binden beobachtbarer Abonnements an Android-Aktivitäten und den Fragment-Lebenszyklus.

Denken Sie daran, dass das Vergessen der Abmeldung eines Observable-Speichers zu Speicherverlusten führen kann und Ihr Aktivitäts- / Fragmentereignis erhalten bleibt, nachdem es vom System zerstört wurde.

Fügen Sie die Bibliothek den Abhängigkeiten hinzu:

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

Dann erweitert `Rx*` -Klassen:

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

Wenn Sie ein Observable abonnieren, können Sie jetzt Folgendes tun:

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

Wenn Sie dies in der `onCreate()` -Methode der Aktivität ausführen, wird sie automatisch in `onDestroy()` abbestellt.

Dasselbe passiert für:

- `onStart()` -> `onStop()`
- `onResume()` -> `onPause()`
- `onAttach()` -> `onDetach()` *(nur Fragment)*
- `onViewCreated()` -> `onDestroyView()` *(nur Fragment)*

Als Alternative können Sie das Ereignis angeben, zu dem die Abbestellung erfolgen soll:

Aus einer Tätigkeit:

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

Aus einem Fragment:

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

```
    .subscribe();
```

Sie können den beobachtbaren Lebenszyklus auch mit der Methode `lifecycle()` abrufen, um Lebenszyklusereignisse direkt abzuhören.

RxLifecycle kann auch verwendet werden, um den beobachtbaren Lebenszyklus direkt weiterzuleiten:

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

Wenn Sie `Single` oder `Completable` , können Sie dies tun, indem Sie `forSingle()` bzw. `forCompletable` nach der Bind-Methode hinzufügen:

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

Es kann auch mit der Navi- Bibliothek verwendet werden.

Rxpermissions

Diese Bibliothek ermöglicht die Verwendung von RxJava mit dem neuen Berechtigungsmodell für Android M.

Fügen Sie die Bibliothek den Abhängigkeiten hinzu:

Rxjava

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

Rxjava2

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

Verwendungszweck

Beispiel (mit Retrolambda der Kürze halber, aber nicht erforderlich):

```
// 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
        }
```

```
    });
```

Lesen Sie mehr: https://github.com/tbruyelle/RxPermissions .

Android mit RxJava online lesen: https://riptutorial.com/de/rx-java/topic/7125/android-mit-rxjava

Kapitel 3: Beobachtbar

Examples

Erstellen Sie ein Observable

Es gibt mehrere Möglichkeiten, ein Observable in RxJava zu erstellen. Die leistungsfähigste Methode ist die Verwendung der `Observable.create` Methode. Es ist aber auch der **komplizierteste Weg** . Sie müssen **es daher** so weit wie möglich **vermeiden** .

Einen hervorragenden Wert ausgeben

Wenn Sie bereits einen Wert haben, können Sie `Observable.just` , um Ihren Wert `Observable.just` .

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

Einen Wert ausgeben, der berechnet werden soll

Wenn Sie einen Wert ausgeben möchten, der noch nicht berechnet wurde oder dessen Berechnung lange dauern kann, können Sie mit `Observable.fromCallable` den nächsten Wert ausgeben.

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

`longComputation()` wird nur aufgerufen, wenn Sie Ihr `Observable` abonnieren. Auf diese Weise wird die Berechnung *faul* .

Alternative zum Senden eines Wertes, der berechnet werden soll

`Observable.defer` ein `Observable` wie `Observable.fromCallable` , es wird jedoch verwendet, wenn Sie statt eines Werts ein `Observable` zurückgeben müssen. Dies ist nützlich, wenn Sie die Fehler in Ihrem Anruf verwalten möchten.

```
Observable.defer(() -> {
          try {
                return Observable.just(longComputation());
          } catch(SpecificException e) {
                return Observable.error(e);
```

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

Warme und kalte Observables

Observables werden in Abhängigkeit von ihrem Emissionsverhalten allgemein als `Hot` oder `Cold` klassifiziert.
Ein `Cold Observable` ist dasjenige, das auf Anfrage zu `Cold Observable` beginnt (Abonnement), während ein `Hot Observable` unabhängig von Abonnements aussendet.

Kalt beobachtbar

```
/* 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
```

Die Ausgabe des obigen Codes sieht folgendermaßen aus (kann variieren):

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

Beachten Sie, dass obwohl `sub2` spät startet, die Werte von Anfang an empfangen werden. `Cold Observable` gibt ein `Cold Observable` nur Elemente aus, wenn dies verlangt wird.
Mehrfachanforderung startet mehrere Pipelines.

Heiß beobachtbar

Hinweis: Heiße Observables geben Werte unabhängig von den einzelnen Abonnements aus. Sie haben ihre eigene Zeitleiste und Ereignisse treten auf, egal ob jemand zuhört oder nicht.

Ein `Cold Observale` kann mit einem einfachen `publish` in ein `Hot Observable` umgewandelt werden.

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

`publish` gibt ein `ConnectableObservable` , das Funktionen zum *Verbinden* und *Trennen* der *Verbindung* mit dem Observable hinzufügt.

```
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));
```

Die oben genannten Erträge:

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

`sub2` beachte, dass, obwohl `sub2` spät zu beobachten beginnt, es mit 1 `sub1` .
Die Trennung ist etwas komplizierter! Das Trennen der Verbindung erfolgt beim `Subscription` und nicht beim `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();
```

Das oben genannte produziert:

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

Beim Trennen der Verbindung wird `Observable` Wesentlichen beendet und neu gestartet, wenn ein neues Abonnement hinzugefügt wird.

`Hot Observable` kann zum Erstellen eines `EventBus` . Solche EventBusse sind im Allgemeinen leicht und superschnell. Der einzige Nachteil eines RxBus besteht darin, dass alle Ereignisse manuell implementiert und an den Bus übergeben werden müssen.

Beobachtbar online lesen: https://riptutorial.com/de/rx-java/topic/1418/beobachtbar

Kapitel 4: Gegendruck

Examples

Einführung

Gegendruck ist , wenn in einer `Observable` Verarbeitungspipeline, einige asynchronen Phasen nicht die Werte schnell genug verarbeiten können und brauchen einen Weg , um die Upstream - Hersteller zu sagen zu verlangsamen.

Der klassische Fall der Notwendigkeit eines Rückstaus ist, wenn der Hersteller eine heiße Quelle ist:

```
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 diesem Beispiel erzeugt der Haupt-Thread 1 Million Artikel für einen Endverbraucher, der ihn in einem Hintergrund-Thread verarbeitet. Es ist wahrscheinlich, dass die Methode `compute(int)` einige Zeit in `compute(int)` nimmt, der Overhead der Operatorkette `Observable` kann jedoch auch die Zeit für die Verarbeitung von Elementen erhöhen. Der produzierende Thread mit der for-Schleife kann dies jedoch nicht wissen und bleibt auf der `onNext` .

Intern verfügen asynchrone Operatoren über Puffer, um solche Elemente zu speichern, bis sie verarbeitet werden können. Im klassischen Rx.NET und frühen RxJava waren diese Puffer unbegrenzt, was bedeutet, dass sie wahrscheinlich fast alle 1 Million Elemente des Beispiels enthalten würden. Das Problem beginnt, wenn beispielsweise 1 Milliarde Elemente vorhanden sind oder die gleiche 1-Millionen-Sequenz 1000-mal in einem Programm vorkommt, was zu `OutOfMemoryError` und generell zu `OutOfMemoryError` aufgrund von übermäßigem GC-Overhead führt.

Ähnlich wie die Fehlerbehandlung zu einem erstklassigen Bürger wurde und Operatoren zur Bearbeitung erhielt (über `onErrorXXX` Operatoren), ist `onErrorXXX` eine weitere Eigenschaft von Datenflüssen, über die der Programmierer (über `onBackpressureXXX` Operatoren) nachdenken und sie handhaben `onBackpressureXXX` .

Neben dem `PublishSubject` genannten `PublishSubject` gibt es andere Operatoren, die den Gegendruck hauptsächlich aus funktionalen Gründen nicht unterstützen. Zum Beispiel kann das Bediener `interval` emittiert periodisch Werte, es backpressuring führen würde , in der Zeit relativ zu einer Wanduhr zu verschieben.

In modernen RxJava verfügen die meisten asynchronen Operatoren jetzt über einen begrenzten internen Puffer, wie beispielsweise `observeOn` oben. Jeder Versuch, diesen Puffer zu überlaufen, beendet die gesamte Sequenz mit `MissingBackpressureException` . Die Dokumentation jedes Bedieners enthält eine Beschreibung seines Gegendruckverhaltens.

In normalen kalten Abläufen ist der Gegendruck jedoch subtiler vorhanden (was `MissingBackpressureException` nicht ergeben sollte und sollte). Wenn das erste Beispiel neu geschrieben wird:

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

Thread.sleep(10_000);
```

Es gibt keinen Fehler und alles läuft reibungslos mit wenig Speicherbedarf. Der Grund dafür ist, dass viele `observeOn` Werte auf Anforderung "generieren" können, und der Operator `observeOn` kann dem `range` mitteilen, dass höchstens so viele Werte generiert werden, die der `observeOn` Puffer ohne Überlauf auf einmal halten kann.

Diese Verhandlung basiert auf dem Informatikkonzept der Co-Routinen (ich rufe Sie an, Sie rufen mich an). Der Bediener `range` sendet einen Rückruf, in Form einer Implementierung der `Producer` -Schnittstelle, an die `observeOn` durch ihren (inneren Aufruf `Subscriber` ,s) `setProducer` . Im Gegenzug ruft das `observeOn` `Producer.request(n)` mit einem Wert auf, mit dem der `range` wird, den es erzeugen darf (dh `onNext`), und zwar viele **zusätzliche** Elemente. Es ist dann die `observeOn` , die `request` zum richtigen Zeitpunkt und mit dem richtigen Wert aufzurufen, damit die Daten fließen, aber nicht überlaufen.

Das Ausdrücken eines Rückdrucks bei Endverbrauchern ist selten notwendig (weil sie in Bezug auf ihren unmittelbaren Upstream synchron sind, und der Rückdruck geschieht natürlich aufgrund der Blockierung der Aufrufstapel), es kann jedoch einfacher sein, die Funktionsweise des Endverbrauchers zu verstehen:

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

Die `onStart` Implementierung gibt hier den `range` an, um den ersten Wert zu erzeugen, der dann in `onNext` . Sobald die `compute(int)` , wird der andere Wert aus dem `range` angefordert. In einer naiven Implementierung `range` würde eine solche Aufforderung rekursiv rufen `onNext` , an führende `StackOverflowError` was natürlich unerwünscht ist .

Um dies zu verhindern, verwenden Operatoren eine sogenannte Trampolin-Logik, die solche wiedereintrittsbedingten Aufrufe verhindert. In `range` wird daran erinnert, dass während des Aufrufs von `onNext()` ein `request(1) onNext()` Sobald `onNext()` zurückkehrt, wird eine weitere Runde ausgeführt und `onNext()` mit dem nächsten Ganzzahlwert `onNext()` . Wenn also beide ausgetauscht werden, funktioniert das Beispiel immer noch gleich:

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

    compute(v);
}
```

Dies gilt jedoch nicht für `onStart` . Obwohl die `Observable` Infrastruktur garantiert wird es höchstens einmal auf jedem heißen `Subscriber` , der Anruf zu `request(1)` kann direkt die Emission eines Elements auslösen. Wenn nach dem Aufruf von `request(1)` eine Initialisierungslogik vorhanden ist, die von `onNext` benötigt wird, kann es zu Ausnahmen kommen:

```
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 diesem synchronen Fall wird sofort eine `NullPointerException` ausgelöst, während `onStart` noch ausgeführt `onStart` . Ein subtilerer Fehler tritt auf, wenn der Aufruf von `request(1)` einen asynchronen Aufruf von `onNext` in einem anderen Thread auslöst und den `name` in `onNext` Races `onNext` , indem er in der `onStart` post- `request` .

Daher sollte man die gesamte `onStart` in `onStart` oder sogar davor durchführen und `request()` zuletzt aufrufen. Implementierungen von `request()` in Operatoren stellen sicher, dass bei Bedarf eine ordnungsgemäße Beziehung vor dem Ereignis (oder in anderen Begriffen: Speicherfreigabe oder vollständiger Zaun) besteht.

Die onBackpressureXXX-Operatoren

Die meisten Entwickler stoßen auf einen Gegendruck, wenn ihre Anwendung mit `MissingBackpressureException` und die Ausnahme normalerweise auf den `observeOn` Operator `observeOn` . Die eigentliche Ursache ist normalerweise die nicht- `PublishSubject` Verwendung von `PublishSubject` , `timer()` oder `interval()` oder benutzerdefinierten Operatoren, die mit `create()` .

Es gibt verschiedene Möglichkeiten, mit solchen Situationen umzugehen.

Vergrößern der Puffergrößen

In manchen Fällen entstehen solche Überläufe aufgrund von bersten Quellen. Plötzlich tippt der Benutzer zu schnell auf den Bildschirm und `observeOn` bei Android-Überläufen den standardmäßigen internen 16-Element-Puffer von On.

Die meisten gegen den Druckausdruck empfindlichen Operatoren in den aktuellen Versionen von RxJava ermöglichen es Programmierern nun, die Größe ihrer internen Puffer anzugeben. Die relevanten Parameter werden üblicherweise als `bufferSize` , `prefetch` oder `capacityHint` . In Anbetracht des überfließenden Beispiels in der Einführung können wir die Puffergröße von `observeOn` einfach erhöhen, um genügend Platz für alle Werte zu haben.

```
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);
}
```

Beachten Sie jedoch, dass dies im Allgemeinen nur eine temporäre Lösung ist, da der Überlauf immer noch auftreten kann, wenn die Quelle die vorhergesagte Puffergröße überproduziert. In diesem Fall kann einer der folgenden Operatoren verwendet werden.

Batching / Überspringen von Werten mit Standardoperatoren

`MissingBackpressureException` die Quelldaten im Stapel effizienter verarbeitet werden können, können Sie die Wahrscheinlichkeit von `MissingBackpressureException` verringern, indem Sie einen der standardmäßigen Stapeloperatoren (nach Größe und / oder Zeit) verwenden.

```
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);
}
```

Wenn einige der Werte sicher ignoriert werden können, können Sie die Sampling-Funktion (mit der Zeit oder ein anderes Observable) und die Throttling-Operatoren (`throttleFirst` , `throttleLast` , `throttleWithTimeout`) verwenden.

```
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);
}
```

Beachten Sie jedoch, dass diese Operatoren nur die Rate des Empfangens von Werten durch den Downstream reduzieren und daher zu `MissingBackpressureException` führen `MissingBackpressureException` .

onBackpressureBuffer ()

Dieser Operator führt in seiner parameterlosen Form einen unbegrenzten Puffer zwischen der vorgelagerten Quelle und dem nachgelagerten Operator ein. Unbegrenzt sein bedeutet, solange die JVM nicht über genügend Arbeitsspeicher verfügt, kann sie mit fast jeder Menge umgehen, die aus einer quälenden Quelle kommt.

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

In diesem Beispiel geht das `observeOn` mit einer sehr geringen Puffergröße `MissingBackpressureException` gibt jedoch keine `MissingBackpressureException` da `onBackpressureBuffer` alle 1 Million Werte `onBackpressureBuffer` und kleine Batches davon zur `observeOn` .

Beachten Sie jedoch, dass `onBackpressureBuffer` seine Quelle auf unbegrenzte Weise verbraucht, d. `onBackpressureBuffer` , Ohne dass ein Gegendruck darauf `onBackpressureBuffer` wird. Dies hat zur Folge, dass auch eine Gegendruck unterstützende Quelle wie `range` vollständig realisiert wird.

Es gibt 4 zusätzliche Überladungen von `onBackpressureBuffer`

onBackpressureBuffer (int. Kapazität)

Dies ist eine begrenzte Version, die `BufferOverflowError` signalisiert, falls der Puffer die angegebene Kapazität erreicht.

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

Die Bedeutung dieses Operators nimmt ab, da immer mehr Operatoren die Einstellung der Puffergröße zulassen. Im Übrigen bietet dies die Möglichkeit, den internen Puffer zu erweitern, indem eine größere Anzahl mit `onBackpressureBuffer` als die Standardeinstellung verwendet wird.

onBackpressureBuffer (int capacity, Action0 onOverflow)

Diese Überladung ruft eine (gemeinsam genutzte) Aktion für den Fall eines Überlaufs auf. Seine Nützlichkeit ist eher begrenzt, da über den Überlauf keine anderen Informationen als der aktuelle Aufrufstapel bereitgestellt werden.

onBackpressureBuffer (int capacity, Action0 onOverflow, BackpressureOverflow.Strategy-Strategie)

Diese Überlastung ist tatsächlich sinnvoller, da sie definieren soll, was zu tun ist, wenn die Kapazität erreicht ist. Die `BackpressureOverflow.Strategy` ist eigentlich eine Schnittstelle, aber die Klasse `BackpressureOverflow` bietet 4 statische Felder, deren Implementierungen typische Aktionen darstellen:

- `ON_OVERFLOW_ERROR` : Dies ist das Standardverhalten der vorherigen beiden Überladungen, das eine `BufferOverflowException` signalisiert
- `ON_OVERFLOW_DEFAULT` : Momentan ist es das gleiche wie `ON_OVERFLOW_ERROR`
- `ON_OVERFLOW_DROP_LATEST` : Wenn ein Überlauf auftreten würde, wird der aktuelle Wert einfach ignoriert und nur die alten Werte werden einmalig von den Downstream-Anforderungen geliefert.
- `ON_OVERFLOW_DROP_OLDEST` : `ON_OVERFLOW_DROP_OLDEST` das älteste Element im Puffer und fügt den aktuellen Wert hinzu.

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

Beachten Sie, dass die letzten beiden Strategien eine Diskontinuität im Stream verursachen, wenn Elemente ausgeblendet werden. Darüber hinaus signalisieren sie keine `BufferOverflowException` .

onBackpressureDrop ()

Wenn der Downstream nicht bereit ist, Werte zu empfangen, wird dieser Operator dieses Element aus der Sequenz entfernen. Man kann sich das als einen `onBackpressureBuffer` mit der Kapazität `onBackpressureBuffer ON_OVERFLOW_DROP_LATEST` .

Dieser Operator ist nützlich, wenn Werte aus einer Quelle (z. B. Mausbewegungen oder aktuelle GPS-Positionssignale) sicher ignoriert werden können, da später aktuellere Werte angezeigt werden.

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

Dies kann in Verbindung mit dem `interval()` nützlich sein. Wenn beispielsweise eine periodische Hintergrundaufgabe ausgeführt werden soll, die Iteration jedoch länger als die Periode dauern kann, kann die Benachrichtigung über das überschüssige Intervall gelöscht werden, da später mehr angezeigt wird:

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

Es gibt eine Überladung dieses Operators: `onBackpressureDrop(Action1<? super T> onDrop)` bei der die (gemeinsam genutzte) Aktion aufgerufen wird, wobei der Wert fallen gelassen wird. Diese Variante ermöglicht das Bereinigen der Werte selbst (z. B. Freigeben zugehöriger Ressourcen).

onBackpressureLatest ()

Der letzte Operator behält nur den neuesten Wert und überschreibt praktisch ältere, nicht gelieferte Werte. Man kann sich dies als eine Variante des `onBackpressureBuffer` mit einer Kapazität von 1 und der Strategie `ON_OVERFLOW_DROP_OLDEST` .

Im Gegensatz zu `onBackpressureDrop` steht immer ein Wert für den Verbrauch zur Verfügung, wenn der Downstream hinterherhinkt. Dies kann in einigen telemetrieähnlichen Situationen nützlich sein, in denen die Daten in einem unruhigen Muster erscheinen, aber nur die neuesten Daten für die Verarbeitung interessant sind.

Wenn der Benutzer zum Beispiel viel auf den Bildschirm klickt, möchten wir trotzdem auf seine neuesten Eingaben reagieren.

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

Die Verwendung von `onBackpressureDrop` in diesem Fall dazu führen, dass der letzte Klick abgeworfen wird und der Benutzer sich wundert, warum die Geschäftslogik nicht ausgeführt

wurde.

Backpressured-Datenquellen erstellen

Das Erstellen von komprimierten Datenquellen ist im Allgemeinen die relativ einfache Aufgabe im Umgang mit dem Backpressure, da die Bibliothek bereits statische Methoden für `Observable` anbietet, die den Backpressure für den Entwickler handhaben. Wir können zwei Arten von Factory-Methoden unterscheiden: kalte "Generatoren", die entweder Elemente zurückgeben und basierend auf der nachgelagerten Nachfrage erzeugen, und "heiße" Pusher ", die normalerweise nicht reaktive und / oder nicht-unterdrückbare Datenquellen überbrücken und einige Rückdruckhandlungen überlagern Sie.

gerade

Die grundlegendste Rückstau bewusst Quelle über erstellt `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
}
```

Da wir in `onStart` ausdrücklich keine Anfragen `onStart` , wird nichts gedruckt. `just` toll, wenn es einen konstanten Wert gibt, mit dem wir eine Sequenz starten möchten.

Leider `just` ist oft für eine Art und Weise falsch , etwas zu berechnen dynamisch durch verbraucht werden `Subscriber` s:

```
int counter;

int computeValue() {
   return ++counter;
}

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

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

Überraschend für einige ist, dass 1 zweimal gedruckt wird, anstatt 1 bzw. 2 zu drucken. Wenn der Anruf umgeschrieben wird, wird offensichtlich, warum er so funktioniert:

```
int temp = computeValue();
```

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

Der `computeValue` wird als Teil der Hauptroutine und nicht als Antwort auf die Abonnenten der Abonnenten aufgerufen.

vonCallable

Was die Leute wirklich brauchen, ist die Methode von `fromCallable` :

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

Hier wird `computeValue` nur ausgeführt, wenn ein Abonnent abonniert und für jeden von ihnen die erwartete 1 und 2 `fromCallable` . Natürlich unterstützt `fromCallable` auch den Gegendruck ordnungsgemäß und gibt den berechneten Wert nicht aus, wenn dies nicht angefordert wird. Beachten Sie jedoch, dass die Berechnung trotzdem stattfindet. Für den Fall, dass die Berechnung selbst verzögert werden soll, bis der Downstream tatsächlich angefordert wird, können wir `just` mit `map` :

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

`just` wird der konstante Wert erst `computeValue` , wenn er dem Ergebnis des `computeValue` , der noch für jeden Teilnehmer einzeln aufgerufen wird.

von

Wenn die Daten bereits verfügbar ist als ein Array von Gegenständen, eine Liste von Objekten oder einer `Iterable` Quelle, wobei die jeweiligen `from` Überlastungen den Gegendruck und die Emission von solchen Quellen handhaben :

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

Der Einfachheit halber (und Warnungen über generische Array - Erstellung zu vermeiden) gibt es 2 bis 10 Argument Überlastungen `just` , dass intern delegieren `from` .

Das `from(Iterable)` bietet auch eine interessante Gelegenheit. Viele Wertschöpfungen können in Form einer Zustandsmaschine ausgedrückt werden. Jedes angeforderte Element löst einen Zustandsübergang und eine Berechnung des zurückgegebenen Werts aus.

Das Schreiben solcher Zustandsmaschinen als `Iterable` s ist etwas kompliziert (aber immer noch einfacher als das Schreiben eines `Observable` zum Konsumieren). Im Gegensatz zu C # bietet Java keine Unterstützung des Compilers, um solche Zustandsmaschinen zu erstellen, indem einfach klassisch aussehender Code (mit `yield return` und `yield break`). Einige Bibliotheken bieten einige Hilfestellungen, z. B. `AbstractIterable` Google Guava und `Ix.generate()` `Ix.forloop()` und `Ix.forloop()` . Diese sind an sich schon einer ganzen Reihe würdig. Sehen wir uns also eine sehr grundlegende `Iterable` Quelle an, die einen konstanten Wert auf unbestimmte Zeit wiederholt:

```
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);
```

Wenn wir den `iterator` über die klassische for-Schleife verbrauchen, führt dies zu einer Endlosschleife. Da wir daraus ein `Observable` bauen, können wir unseren Willen zum Ausdruck bringen, nur die ersten fünf davon zu konsumieren und dann aufhören, irgendetwas anzufordern. Dies ist die wahre Macht der trägen Auswertung und Berechnung innerhalb von `Observable` s.

erstellen (SyncOnSubscribe)

Manchmal ist die Datenquelle, die in die reaktive Welt umgewandelt werden soll, synchron (blockierend) und pull-artig, d. H. Wir müssen eine `get` oder `read` Methode aufrufen, um die nächsten Daten zu erhalten. Man könnte das natürlich in eine `Iterable` aber wenn solche Quellen mit Ressourcen `Iterable` sind, können diese Ressourcen auslaufen, wenn der Downstream die Sequenz abbestellt, bevor sie enden würde.

Für solche Fälle verfügt RxJava über die `SyncOnSubscribe` Klasse. Man kann es erweitern und seine Methoden implementieren oder eine seiner Lambda-basierten Factory-Methoden verwenden, um eine Instanz zu erstellen.

```
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);
```

Im `SyncOnSubscribe` verwendet `SyncOnSubscribe` 3 Rückrufe.

Die ersten Rückrufe ermöglichen das Erstellen eines Zustands pro Teilnehmer, z. B. `FileInputStream` im Beispiel. Die Datei wird für jeden einzelnen Teilnehmer unabhängig geöffnet.

Der zweite Callback nimmt dieses `onXXX` und stellt einen Ausgabe- `Observer` `onXXX` dessen `onXXX` Methoden `onXXX` werden können, um Werte `onXXX` . Dieser Callback wird so oft ausgeführt, wie der Downstream angefordert wurde. Bei jedem Aufruf muss `onNext` höchstens einmal `onNext` werden, optional gefolgt von `onError` oder `onCompleted` . In dem Beispiel rufen wir `onCompleted()` wenn das gelesene Byte negativ ist, das Ende der Datei `onError` , und `onError` falls der `IOException` eine `IOException` .

Der letzte Rückruf wird aufgerufen, wenn der Downstream seine Abmeldung aufhebt (den Eingangsstrom schließt) oder wenn der vorherige Rückruf die Terminalmethoden aufgerufen hat. es ermöglicht die Freigabe von Ressourcen. Da nicht alle Quellen alle diese Funktionen benötigen, können die statischen Methoden von `SyncOnSubscribe` Instanzen ohne sie erstellen.

Leider lösen viele Methodenaufrufe in der gesamten JVM und anderen Bibliotheken geprüfte Ausnahmen aus und müssen in `try-catch` da die von dieser Klasse verwendeten funktionalen Schnittstellen keine geprüften Ausnahmen zulassen.

Natürlich können wir andere typische Quellen imitieren, wie zum Beispiel eine unbeschränkte Reichweite:

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

In diesem Setup beginnt der `current` mit `0` und beim nächsten Aufruf von Lambda hält der Parameter `current` jetzt `1` .

Es gibt eine Variante von `SyncOnSubscribe` Namen `AsyncOnSubscribe` , die ziemlich ähnlich aussieht, mit der Ausnahme, dass der mittlere Rückruf auch einen langen Wert annimmt, der den Anforderungsbetrag von Downstream darstellt, und der Rückruf sollte ein `Observable` mit der gleichen Länge generieren. Diese Quelle verkettet dann alle diese `Observable` zu einer einzigen Sequenz.

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

```
);
```

Es gibt eine anhaltende (hitzige) Diskussion über die Nützlichkeit dieser Klasse und wird im Allgemeinen nicht empfohlen, da sie routinemäßig die Erwartungen darüber, wie diese generierten Werte tatsächlich ausgegeben werden und wie sie darauf reagieren werden, oder sogar die Art der Anforderungswerte, die sie empfängt, widerlegt komplexere Verbraucherszenarien.

erstellen (Sender)

Manchmal ist die Quelle, die in ein `Observable` soll, bereits heiß (z. B. Mausbewegungen) oder kalt, jedoch nicht in ihrer API (z. B. ein asynchroner Netzwerkrückruf) unter Druck gesetzt.

In solchen Fällen wurde in einer aktuellen Version von RxJava die Factory-Methode `create(emitter)` . Es sind zwei Parameter erforderlich:

- einen Rückruf, der mit einer Instanz der `Emitter<T>` -Schnittstelle für jeden eingehenden Teilnehmer aufgerufen wird,
- eine `Emitter.BackpressureMode` Enumeration, in der der Entwickler das anzuwendende `Emitter.BackpressureMode` angeben muss. Es hat die üblichen Modi, ähnlich wie bei `onBackpressureXXX` zusätzlich zu einer `MissingBackpressureException` oder zum einfachen Ignorieren eines solchen Überlaufs.

Beachten Sie, dass derzeit keine zusätzlichen Parameter für diese Gegendruckmodi unterstützt werden. Wenn Sie diese Anpassung benötigen, verwenden Sie `NONE` als `onBackpressureXXX` und wenden den entsprechenden `onBackpressureXXX` auf das resultierende `Observable` an.

Der erste typische Fall für die Verwendung, wenn Sie mit einer Push-basierten Quelle interagieren möchten, beispielsweise mit GUI-Ereignissen. Diese APIs verfügen über eine Form von `addListener` / `removeListener` Aufrufen, die verwendet werden können:

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

    button.addActionListener(al);

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

}, BackpressureMode.BUFFER);
```

Der `Emitter` ist relativ einfach zu benutzen; `onNext` , `onError` und `onCompleted` können darauf `onNext` , und der Operator übernimmt die Verwaltung des `onCompleted` und der Abmeldung selbst. Darüber hinaus, wenn die umwickelte API cancellation (wie beispielsweise der Hörer Entfernung in dem Beispiel) unterstützt, kann man die Verwendung `setCancellation` (oder `setSubscription` für `Subscription` -ähnlichen Ressourcen) einen Annullierung Rückruf zu registrieren, wenn der stromabwärtige abmeldet oder die aufgerufen wird `onError` / `onCompleted` wird in der bereitgestellten `Emitter` Instanz aufgerufen.

Bei diesen Methoden kann jeweils nur eine einzige Ressource mit dem Emitter verknüpft werden, und durch das Einrichten einer neuen Ressource wird die alte automatisch deaktiviert. Wenn mehrere Ressourcen behandelt werden müssen, erstellen Sie ein `CompositeSubscription` , verknüpfen Sie es mit dem Emitter und fügen Sie dem `CompositeSubscription` selbst weitere Ressourcen hinzu:

```
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);
```

Das zweite Szenario umfasst normalerweise eine asynchrone, auf Rückruf basierende API, die in ein `Observable` konvertiert werden muss.

```
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 diesem Fall arbeitet die Delegation auf dieselbe Weise. Leider unterstützen diese klassischen APIs im Callback-Stil normalerweise keine Stornierung, aber wenn dies der Fall ist, können Sie ihre Stornierung genau wie in den vorangegangenen Beispielen einrichten (mit einer etwas aufwendigeren Methode). Beachten Sie die Verwendung des `LATEST` Gegendruckmodus. Wenn wir wissen, dass es nur einen einzigen Wert gibt, brauchen wir die `BUFFER` Strategie nicht, da sie einen standardmäßig langen 128-Element-Puffer (der bei Bedarf `BUFFER` , der niemals vollständig ausgenutzt wird.

Gegendruck online lesen: https://riptutorial.com/de/rx-java/topic/2341/gegendruck

Kapitel 5: Nachrüstung und RxJava

Examples

Richten Sie Retrofit und RxJava ein

Retrofit2 unterstützt mehrere steckbare Ausführungsmechanismen. Einer davon ist RxJava.

Um Retrofit mit RxJava verwenden zu können, müssen Sie zunächst den Retrofit RxJava-Adapter zu Ihrem Projekt hinzufügen:

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

Dann müssen Sie den Adapter hinzufügen, wenn Sie Ihre Retrofit-Instanz erstellen:

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

In Ihrer Schnittstelle , wenn Sie definieren sollten die API der Rückgabetyp sein `Observable` zB:

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

Sie können auch `Single` anstelle von `Observable` .

Serienanfragen stellen

RxJava ist praktisch, wenn Sie eine Serienanforderung durchführen. Wenn Sie das Ergebnis einer Anfrage für eine andere verwenden `flatMap` können Sie den `flatMap` Operator verwenden:

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

Parallele Anfragen stellen

Sie können den `zip` Operator verwenden, um eine Anfrage parallel zu stellen und die Ergebnisse zu kombinieren, zB:

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

Nachrüstung und RxJava online lesen: https://riptutorial.com/de/rx-java/topic/2950/nachrustung-und-rxjava

Kapitel 6: Operatoren

Bemerkungen

Dieses Dokument beschreibt das grundlegende Verhalten eines Bedieners.

Examples

Betreiber, eine Einführung

Ein Operator kann verwendet werden, um den Fluss von Objekten von `Observable` zu `Subscriber` zu manipulieren.

```
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);
```

Die Ausgabe wäre:

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

Der `map` geändert , um die `Integer` beobachtbaren an einen `String` beobachtbar, wodurch der Fluss von Objekten zu manipulieren.

Operator-Verkettung

Mehrere Operatoren können miteinander `chained` werden, um leistungsfähigere Transformationen und Manipulationen durchzuführen.

```
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
```

Zwischen `Observable` und `Subscriber` kann eine beliebige Anzahl von Operatoren hinzugefügt werden.

flatMap Operator

Der `flatMap` Operator hilft Ihnen, ein Ereignis in ein anderes `Observable` (oder ein Ereignis in null, ein oder mehrere Ereignisse) umzuwandeln.

Es ist ein perfekter Operator, wenn Sie eine andere Methode aufrufen möchten, die ein `Observable`

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

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

`flatMap` serialisiert `perform` Abonnements, **aber** Ereignisse, die von `perform` ausgegeben werden `perform` können möglicherweise nicht bestellt werden. Möglicherweise empfangen Sie Ereignisse, die vom letzten Perform-Aufruf `concatMap` , **vor** Ereignissen vom ersten `perform` Aufruf (Sie sollten stattdessen `concatMap` verwenden).

Wenn Sie ein anderes `Observable` in Ihrem Abonnenten erstellen, **sollten** `flatMap` stattdessen `flatMap` verwenden. Die Hauptidee lautet: **Verlassen Sie niemals das Observable**

Zum Beispiel :

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

kann leicht ersetzt werden durch:

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

Dokumentation zu Reactivex.io: http://reactivex.io/documentation/operators/flatmap.html

Filter Operator

Sie können den `filter` , um Elemente aus dem Wertestrom basierend auf einem Ergebnis einer Prädikatmethode herauszufiltern.

Mit anderen Worten, die Elemente, die vom Observer an den Abonnenten übergeben werden, werden auf der Grundlage des `filter` , den Sie übergeben, verworfen. Wenn die Funktion für einen bestimmten Wert `false` zurückgibt, wird dieser Wert herausgefiltert.

Beispiel:

```
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
    });
```

Dieser Code wird ausgedruckt

```
0.0
4.0
16.0
36.0
64.0
```

Kartenoperator

Sie können den `map` , um die Werte eines Streams basierend auf dem Ergebnis für jeden Wert aus der an `map` Funktion verschiedenen Werten `map` . Der Ergebnisstrom ist eine neue Kopie und ändert den bereitgestellten Wertestrom nicht. Der Ergebnisstrom hat dieselbe Länge wie der Eingabestrom, kann jedoch unterschiedliche Typen aufweisen.

Die an `.map()` Funktion muss einen Wert zurückgeben.

Beispiel:

```
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
    });
```

Dieser Code wird ausgedruckt

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

Die beobachtbare eine akzeptierte In diesem Beispiel `List<Integer>` die Liste wird in einen umgewandelt werden `List<String>` in der Pipeline und die `.subscribe` emittieren `String` ,s

doOnNext-Operator

`doOnNext` Operator genannt , jedes Mal , wenn die Quelle `Observable` ein Element emittiert. Es kann für Debug-Zwecke verwendet werden, eine Aktion für das emittierte Element Anwendung, Protokollierung, 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();
```

Im Beispiel unten `doOnNext` heißt nie , weil die Quelle `Observable` nichts abgibt , weil `Observable.empty()` ruft `onCompleted` nach der Anmeldung.

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

Operator wiederholen

`repeat` ermöglicht die Wiederholung der gesamten Sequenz aus der Quelle `Observable` .

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

Ausgabe des obigen Beispiels

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

Diese Sequenz wiederholt sich unendlich oft und wird nie abgeschlossen.

Um eine endliche Anzahl von Wiederholungen zu wiederholen, übergeben Sie einfach eine Ganzzahl als Argument für den `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")
    );
```

Dieses Beispiel wird gedruckt

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

Es ist sehr wichtig zu verstehen, dass der `repeat` die Quelle `Observable` abonniert, wenn die Quelle `Observable` Source abgeschlossen ist. Lassen Sie uns das obige Beispiel mit `Observable.create` neu schreiben.

```
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")
        );
```

Dieses Beispiel wird gedruckt

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

Bei der Verwendung der Operator-Verkettung ist es wichtig zu wissen, dass der `repeat` **gesamte Sequenz** und nicht den vorhergehenden Operator wiederholt.

```
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")
        );
```

Dieses Beispiel wird gedruckt

```
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
```

Dieses Beispiel zeigt , dass `repeat` Operator wiederholt ganze Sequenz resubscribing `Observable` und nicht zuletzt Wiederholung `map` Operator und es spielt keine Rolle , an welcher Stelle in der Sequenz `repeat` Operator verwendet.

Diese Reihenfolge

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

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

ist gleich dieser Reihenfolge

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

Operatoren online lesen: https://riptutorial.com/de/rx-java/topic/2316/operatoren

Kapitel 7: RxJava2 Fließfähig und Abonnent

Einführung

Dieses Thema zeigt Beispiele und Dokumentation zu den reaktiven Konzepten von Flowable und Subscriber, die in rxjava Version2 eingeführt wurden

Bemerkungen

Das Beispiel benötigt rxjava2 als Abhängigkeit, die Maven-Koordinaten für die verwendete Version sind:

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

Examples

Beispiel eines Produzenten-Verbrauchers mit Unterstützung des Gegendrucks beim Hersteller

Der `TestProducer` aus diesem Beispiel erzeugt `Integer` Objekte in einem bestimmten Bereich und überträgt sie an seinen `Subscriber` . Es erweitert die Klasse `Flowable<Integer>` . Bei einem neuen Abonnenten wird ein `Subscription` , dessen Methode `request(long)` zum Erstellen und Veröffentlichen der Integer-Werte verwendet wird.

Für das `Subscription` , das an den `subscriber` , ist es wichtig, dass die `request()` Methode, die `onNext()` des Abonnenten `onNext()` aus diesem `onNext()` Aufruf rekursiv aufgerufen werden kann. Um einen Stapelüberlauf zu verhindern, verwendet die gezeigte Implementierung den Zähler `outStandingRequests` und das 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;
            }
        });
    }
}
```

Der Consumer in diesem Beispiel erweitert `DefaultSubscriber<Integer>` und fordert beim Start und nach Verbrauch einer Integer-Anforderung die nächste an. Beim Verbrauch der Integer-Werte tritt eine kleine Verzögerung auf, sodass der Gegendruck für den Hersteller aufgebaut wird.

```
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");
    }
}
```

In der folgenden Hauptmethode einer Testklasse werden Hersteller und Verbraucher erstellt und verkabelt:

```
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();
    }
}
```

Beim Ausführen des Beispiels zeigt die Protokolldatei, dass der Consumer kontinuierlich ausgeführt wird, während der Produzent nur dann aktiv wird, wenn der interne fließfähige Puffer von rxjava2 nachgefüllt werden muss.

RxJava2 Fließfähig und Abonnent online lesen: https://riptutorial.com/de/rx-java/topic/9810/rxjava2-flie-fahig-und-abonnent

Kapitel 8: Scheduler

Examples

Grundlegende Beispiele

Scheduler sind eine RxJava-Abstraktion über die Verarbeitungseinheit. Ein Scheduler kann durch einen Executor-Service gesichert werden, Sie können jedoch Ihre eigene Scheduler-Implementierung implementieren.

Ein `Scheduler` sollte diese Anforderung erfüllen:

- Soll unverzögerte Task sequentiell abarbeiten (FIFO-Reihenfolge)
- Task kann verzögert werden

Ein `Scheduler` kann in einigen Operatoren als Parameter verwendet werden (Beispiel: `delay`) oder mit der `subscribeOn` / `observeOn` Methode verwendet werden.

Bei einigen Bedienern wird der `Scheduler` dazu verwendet, die Aufgabe des jeweiligen Bedieners zu bearbeiten. `delay` plant beispielsweise eine verzögerte Task, die den nächsten Wert ausgeben wird. Dies ist ein `Scheduler` , der später beibehalten und ausgeführt wird.

Das `subscribeOn` kann einmal pro `Observable` . Es wird festgelegt, in welchem `Scheduler` der Code des Abonnements ausgeführt wird.

Der `observeOn` kann pro `Observable` mehrfach verwendet werden. Hier wird festgelegt, in welchem `Scheduler` alle **nach** der `observeOn` Methode definierten Aufgaben `observeOn` werden. `observeOn` hilft Ihnen, Thread-Hopping auszuführen.

subscribeOn bestimmter Scheduler

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

observOn mit einem bestimmten Scheduler

```
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()));
```

Festlegen eines bestimmten Schedulers mit einem Operator

Einige Operatoren können einen `Scheduler` als Parameter verwenden.

```
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);
```

Für Abonnenten veröffentlichen:

```
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);
```

Thread-Pool:

```
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();
```

Scheduler online lesen: https://riptutorial.com/de/rx-java/topic/2321/scheduler

Kapitel 9: Themen

Syntax

- Betreff <T, R> subject = AsyncSubject.create (); // Standard AsyncSubject
- Betreff <T, R> subject = BehaviorSubject.create (); // Standard BehaviorSubject
- Betreff <T, R> subject = PublishSubject.create (); // Default PublishSubject
- Betreff <T, R> subject = ReplaySubject.create (); // Standard ReplaySubject
- mySafeSubject = neues SerializedSubject (unSafeSubject); // Konvertieren Sie ein unsafeSubject in ein safeSubject - im Allgemeinen für Betreffs mit mehreren Threads

Parameter

Parameter	Einzelheiten
T	Eingabetyp
R	Ausgabetyp

Bemerkungen

Diese Dokumentation enthält Details und Erklärungen zum `Subject` . Weitere Informationen und weiterführende Informationen finden Sie in der offiziellen Dokumentation .

Examples

Grundthemen

Ein `Subject` in RxJava ist eine Klasse , die sowohl eine ist `Observable` und einen `Observer` . Dies bedeutet im Wesentlichen, dass es als `Observable` fungieren und Eingaben an Abonnenten und als `Observer` , um Eingaben von einem anderen Observable zu erhalten.

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

Die obigen Drucke "Hallo, Welt!" zu konsultieren mit `Subjects` .

Erläuterung

1. Die erste Codezeile definiert einen neuen `Subject` des Typs `PublishSubject`

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

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

2. Die zweite Zeile abonniert das Thema und zeigt das Verhalten des `Observer` .

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

   Dies ermöglicht dem `Subject` , Eingaben wie ein normaler Teilnehmer zu machen

3. Die dritte Zeile ruft die `onNext` Methode des Subjekts, das zeigt `Observable` Verhalten.

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

   Dies ermöglicht dem `Subject` , Eingaben für alle Abonnenten zu geben.

Typen

Ein `Subject` (in RxJava) kann einen der folgenden vier Typen haben:

- AsyncSubject
- BehaviorSubject
- PublishSubject
- ReplaySubject

Ein `Subject` kann auch vom Typ `SerializedSubject` . Dieser Typ stellt sicher, dass der `Subject` nicht gegen den beobachtbaren *Vertrag* verstößt (der besagt, dass alle Anrufe serialisiert werden müssen).

Lesen Sie weiter:

- Betreff aus Dave Sextons Blog verwenden oder nicht verwenden

PublishSubject

`PublishSubject` gibt nur diejenigen Elemente an einen `Observer` , die nach dem Abonnement von der Quelle `Observable` `PublishSubject` .

Ein einfaches `PublishSubject` Beispiel:

```
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);
```

Ausgabe:

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

Im obigen Beispiel abonniert ein `PublishSubject` ein `Observable` das sich wie eine Uhr `PublishSubject` und alle 500 `PublishSubject` Elemente (Long) ausgibt. Wie in der Ausgabe zu sehen ist, gibt `PublishSubject` die `PublishSubject` weiter, die es von der Quelle (`clock`) an seine Subskribenten (`sub1` und `sub2`) `sub2` .

Ein `PublishSubject` kann mit dem Ausgeben von Objekten beginnen, sobald es erstellt wurde, ohne dass ein Beobachter die Gefahr besteht, dass ein oder mehrere Objekte verloren gehen, bis ein Beobachter sich sonnen kann.

```
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
```

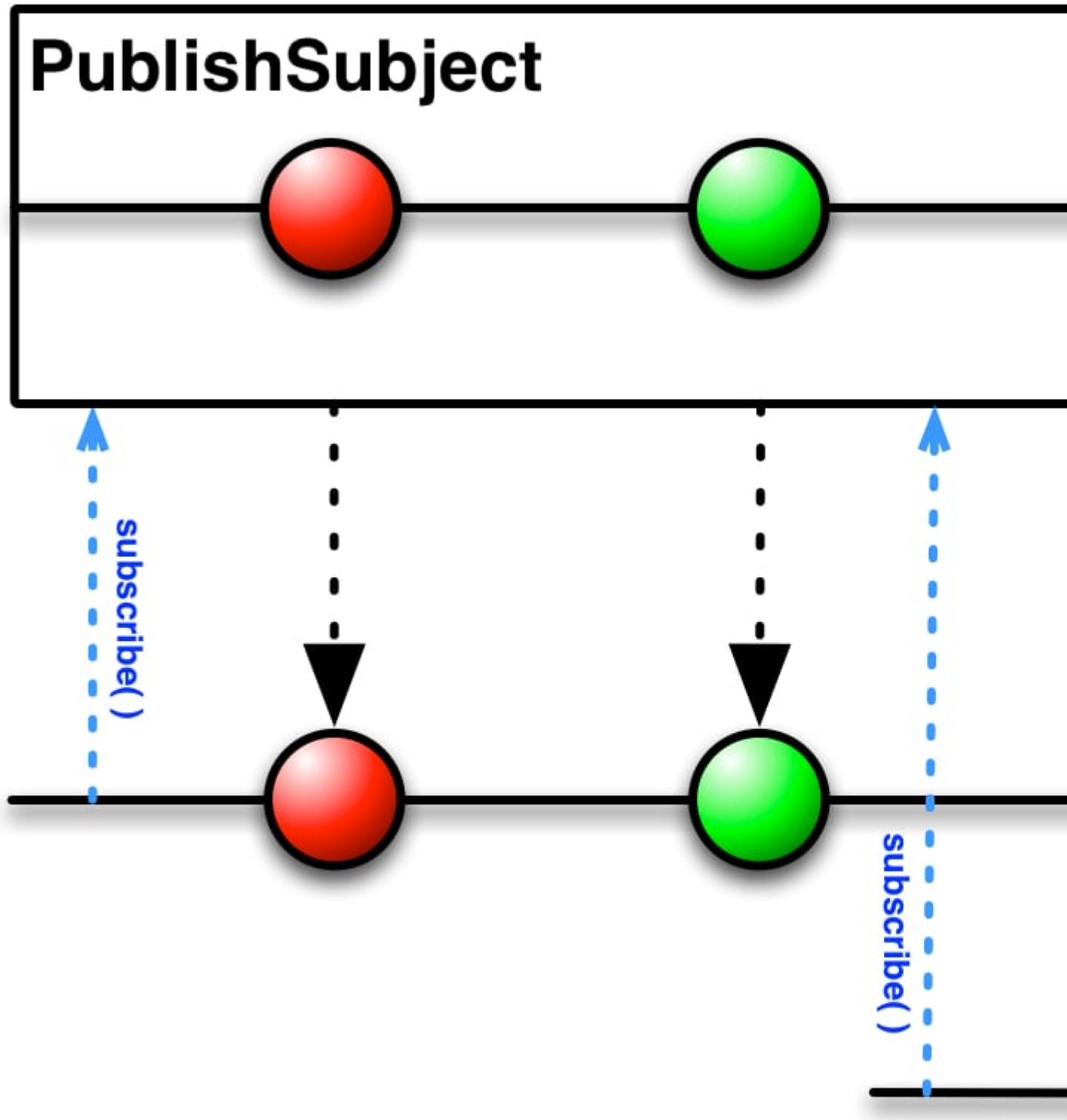
Ausgabe:

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

Beachten Sie, dass `sub1` Werte ab `10 sub1` . Die Verzögerung von 5 Sekunden führte zum *Verlust* von Artikeln. Diese können nicht reproduziert werden. Dies macht `PublishSubject` Wesentlichen zu einem `Hot Observable` .

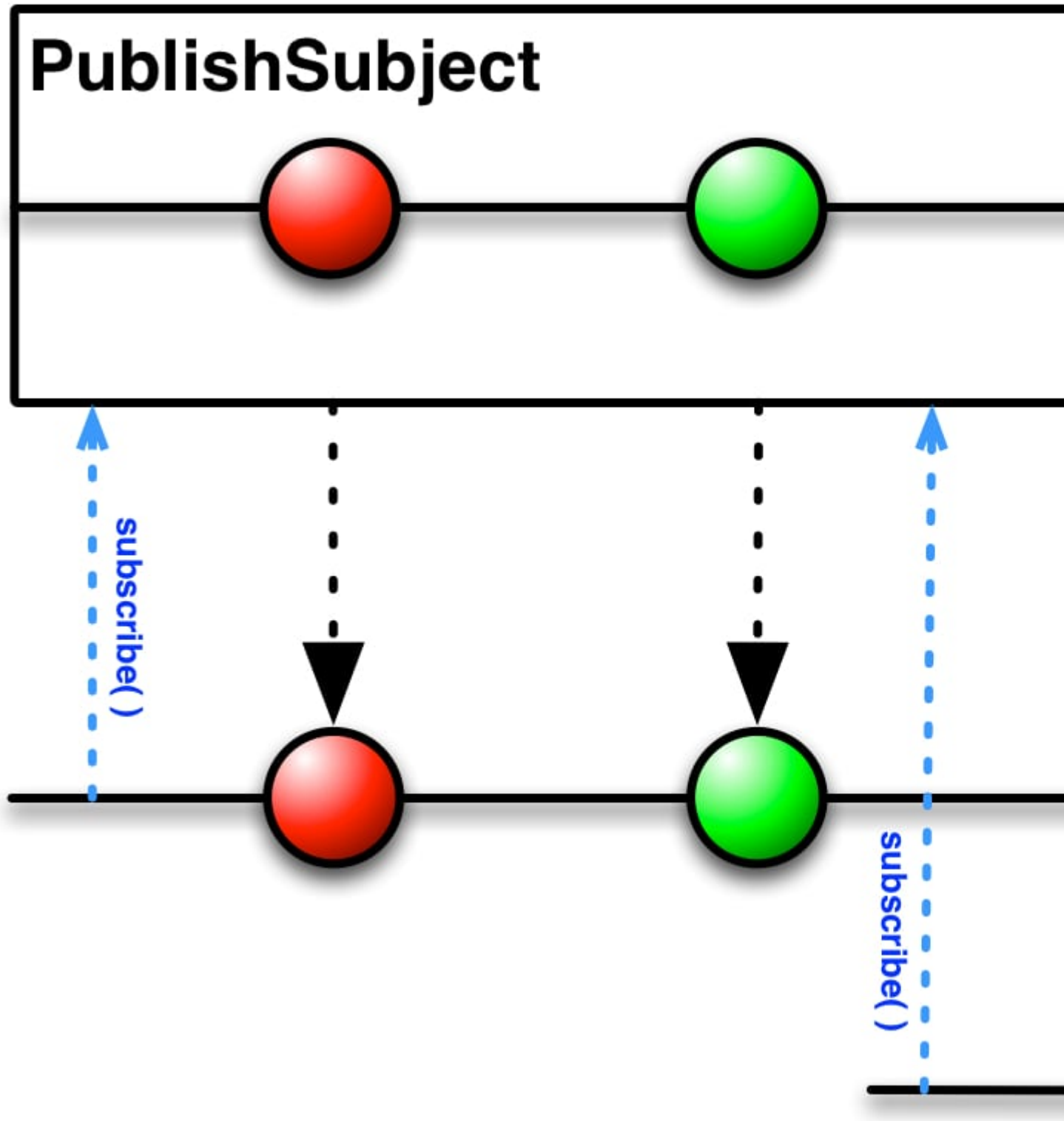
Beachten Sie auch, dass, wenn ein Beobachter das `PublishSubject` abonniert, nachdem ***n*** Elemente `PublishSubject` wurden, diese ***n*** Elemente für diesen Beobachter *nicht* reproduziert werden *können* .

Unten sehen Sie das `PublishSubject` von `PublishSubject`



`PublishSubject` gibt Elemente an alle Abonnenten aus, die zu einem beliebigen Zeitpunkt vor dem `onCompleted` von `onCompleted` der Quelle `Observable` aufgerufen wurden.

Wenn die Quelle `Observable` mit einem Fehler `PublishSubject` , `PublishSubject` keine Elemente an nachfolgende Beobachter, sondern leitet einfach die Fehlerbenachrichtigung von der Quelle Observable weiter.



Anwendungsfall
Angenommen, Sie möchten eine Anwendung erstellen, die die Aktienkurse eines bestimmten Unternehmens überwacht und an alle Kunden weiterleitet, die dies verlangen.

```
/* 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();
```

Im obigen Anwendungsbeispiel dient `PublishSubject` als Brücke, um die Werte von Ihrem Server an alle Clients weiterzuleiten, die Ihren `watcher` abonnieren.

Lesen Sie weiter:

- PublishSubject- Javadocs
- Blog von Thomas Nield (Fortgeschrittene Lektüre)

Themen online lesen: https://riptutorial.com/de/rx-java/topic/3287/themen

Kapitel 10: Unit Testing

Bemerkungen

Da alle Scheduler-Methoden statisch sind, können Komponententests, die die RxJava-Hooks verwenden, nicht parallel in derselben JVM-Instanz ausgeführt werden. Wenn dies der Fall ist, wird ein TestScheduler während eines Komponententests entfernt. Das ist im Grunde der Nachteil der Verwendung der Scheduler-Klasse.

Examples

TestSubscriber

Mit TestSubscribers können Sie die Erstellung eines eigenen Abonnenten vermeiden oder Aktion <?> Abonnieren, um zu überprüfen, ob bestimmte Werte geliefert wurden, wie viele vorhanden sind, ob das Observable abgeschlossen wurde, eine Ausnahme ausgelöst wurde und vieles mehr.

Fertig machen

Dieses Beispiel zeigt nur eine Behauptung, dass die Werte 1, 2, 3 und 4 über onNext an das Observable übergeben wurden.

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

`assertValues` versichert, dass die Anzahl korrekt ist. Wenn Sie nur einige der Werte übergeben würden, würde die Bestätigung fehlschlagen.

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

`assertValues` verwendet die `equals` Methode, wenn `assertValues` werden. So können Sie auf einfache Weise Klassen testen, die als Daten behandelt werden.

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

Dieses Beispiel zeigt eine Klasse, für die eine Gleichheit definiert ist und die Werte aus Observable bestätigt.

```
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
```

Beachten Sie auch, dass wir den kürzeren `assertValue` da nur ein Element `assertValue` werden muss.

Alle Ereignisse erhalten

Bei Bedarf können Sie alle Veranstaltungen auch als Liste abfragen.

```
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();
```

Feststellung von Ereignissen

Wenn Sie umfangreichere Tests für Ihre Ereignisse durchführen möchten, können Sie `getOnNextEvents` (oder `getOn*Events`) mit Ihrer bevorzugten Assertionsbibliothek kombinieren:

```
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 Observable#error

Sie können sicherstellen, dass die richtige Ausnahmeklasse ausgegeben wird:

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

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

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

Sie können auch sicherstellen, dass die exakte Ausnahme ausgelöst wurde:

```
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 ermöglicht es Ihnen, Zeit und Ausführung von Observablen zu steuern, anstatt busy waits zu tun zu haben, Fäden oder irgendetwas Verbindungssystem Zeit zu manipulieren. Dies ist sehr wichtig, wenn Sie Komponententests schreiben möchten, die vorhersehbar, konsistent und schnell sind. Weil Sie Zeit manipulieren, gibt es nicht mehr die Chance, dass ein Thread wurde ausgehungert, dass Ihr Test auf einem langsameren Maschine ausfällt oder dass Sie die Ausführungszeit beschäftigt Warten auf ein Ergebnis verschwenden.

TestScheduler können über die Überladung bereitgestellt werden, die einen Scheduler für alle RxJava-Vorgänge benötigt.

```
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
```

Der TestScheduler ist ziemlich einfach. Es besteht nur aus drei Methoden.

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

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

Auf diese Weise können Sie manipulieren, wann der TestScheduler alle Aktionen auslösen soll, die sich auf einige Zeit in der Zukunft beziehen.

Während des Bestehens des Schedulers wird der TestScheduler normalerweise nicht verwendet, da er unwirksam ist. Wenn Sie die Scheduler in Klassen übergeben, erhalten Sie viel zusätzlichen Code für wenig Gewinn. Stattdessen können Sie sich in RxJavas Schedulers.io () / computation () / etc einhaken. Dies geschieht mit RxJavas Hooks. Auf diese Weise können Sie definieren, was von einem Anruf von einer der Scheduler-Methoden zurückgegeben wird.

```
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;
    }
}
```

Mit dieser Klasse kann der Benutzer den Test-Scheduler abrufen, der für alle Aufrufe an Scheduler angeschlossen wird. Ein Komponententest müsste diesen Scheduler einfach in sein Setup bringen. Es wird dringend empfohlen es im Setup aquiring und nicht als ein gutes altes Feld, weil Ihr TestScheduler zu triggerActions von einer anderen Gerät zu testen versuchen können, wenn Sie vorher Zeit. Nun wird unser Beispiel oben

```
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)
```

So können Sie die Systemuhr effektiv aus Ihrem Unit-Test entfernen (zumindest was RxJava betrifft)

Unit Testing online lesen: https://riptutorial.com/de/rx-java/topic/5207/unit-testing

Credits

S. No	Kapitel	Contributors
1	Erste Schritte mit rx-java	Buttink, Community, dimsuz, Dmitry Avtonomov, Hans Wurst, hello_world, Omar Al Halabi, Saulius Next, Sneh Pandya, svarog, Tom
2	Android mit RxJava	akarnokd, Athafoud, Daniele Segato, Eugen Martynov, Geng Jiawen, Sneh Pandya
3	Beobachtbar	Aki K, dwursteisen, hello_world, JonesV
4	Gegendruck	akarnokd, Bartek Lipinski, Chris A, Cristian, dwursteisen, Niklas, Sebas LG
5	Nachrüstung und RxJava	LordRaydenMK
6	Operatoren	dwursteisen, hello_world, svarog, Vadeg
7	RxJava2 Fließfähig und Abonnent	P.J.Meisch
8	Scheduler	dwursteisen, Gal Dreiman
9	Themen	hello_world, mavHarsha
10	Unit Testing	Buttink, Sir Celsius