Бесплатная электронная книга

учусь

rx-java

Free unaffiliated eBook created from
Stack Overflow contributors.

#rx-java

Около

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

глава 1: Начало работы с rx-java

замечания

В этом разделе представлен общий обзор и поверхностное введение в rx-java.

RxJava - это реализация виртуальных расширений Java VM: библиотека для составления асинхронных и основанных на событиях программ с использованием наблюдаемых последовательностей.

Узнайте больше о RxJava в Wiki Home .

Версии

Версия	Статус	Последняя стабильная версия	Дата выхода
1.x	стабильный	1.3.0	2017-05-05
2.x	стабильный	2.1.1	2017-06-21

Examples

Установка или настройка

Настройка rx-java

1. Gradle

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

2. специалист

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

3. плющ

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

4. Снимки от JFrog

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

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

5. Если вам нужно загрузить банки вместо того , чтобы использовать систему сборки, создать Maven `pom` файл , как это с нужной версии:

```
<?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>
```

Затем выполните:

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

Эта команда загружает `rxjava-*.jar` и ее зависимости в `./target/dependency/`.

Вам нужна Java 6 или более поздняя.

Привет, мир!

Следующее выводит сообщение `Hello, World!` утешить

```
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);
       }

    });
```

```
}
```

Или используя Java 8 лямбда-нотацию

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

Введение в RxJava

Основные понятия RxJava являются его `Observables` и `Subscribers` . `Observable` испускает объекты, а `Subscriber` их потребляет.

наблюдаемый

`Observable` - класс, который реализует реактивный шаблон дизайна. Эти Observables предоставляют методы, позволяющие потребителям подписываться на изменения событий. Наблюдаемые изменения события инициируются. Нет ограничений на количество подписчиков, которое может иметь `Observable` , или количество объектов, которое может наблюдаться `Observable` .

Возьмем, к примеру:

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

Здесь, наблюдаемый объект , называемый `integerObservable` и `stringObservable` создается из фабричного метода `just` при условии библиотеки Rx. Обратите внимание, что `Observable` является общим и может, таким образом, испускать любой объект.

подписчик

`Subscriber` - это потребитель. `Subscriber` может подписаться ***только на один*** наблюдаемый. `Observable` вызывает `onNext()` , `onCompleted()` и `onError()` для `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);
        }
    };
```

Обратите внимание, что `Subscriber` также является общим и может поддерживать любой объект. `Subscriber` должен подписаться на наблюдаемый, вызвав метод `subscribe` на наблюдаемый.

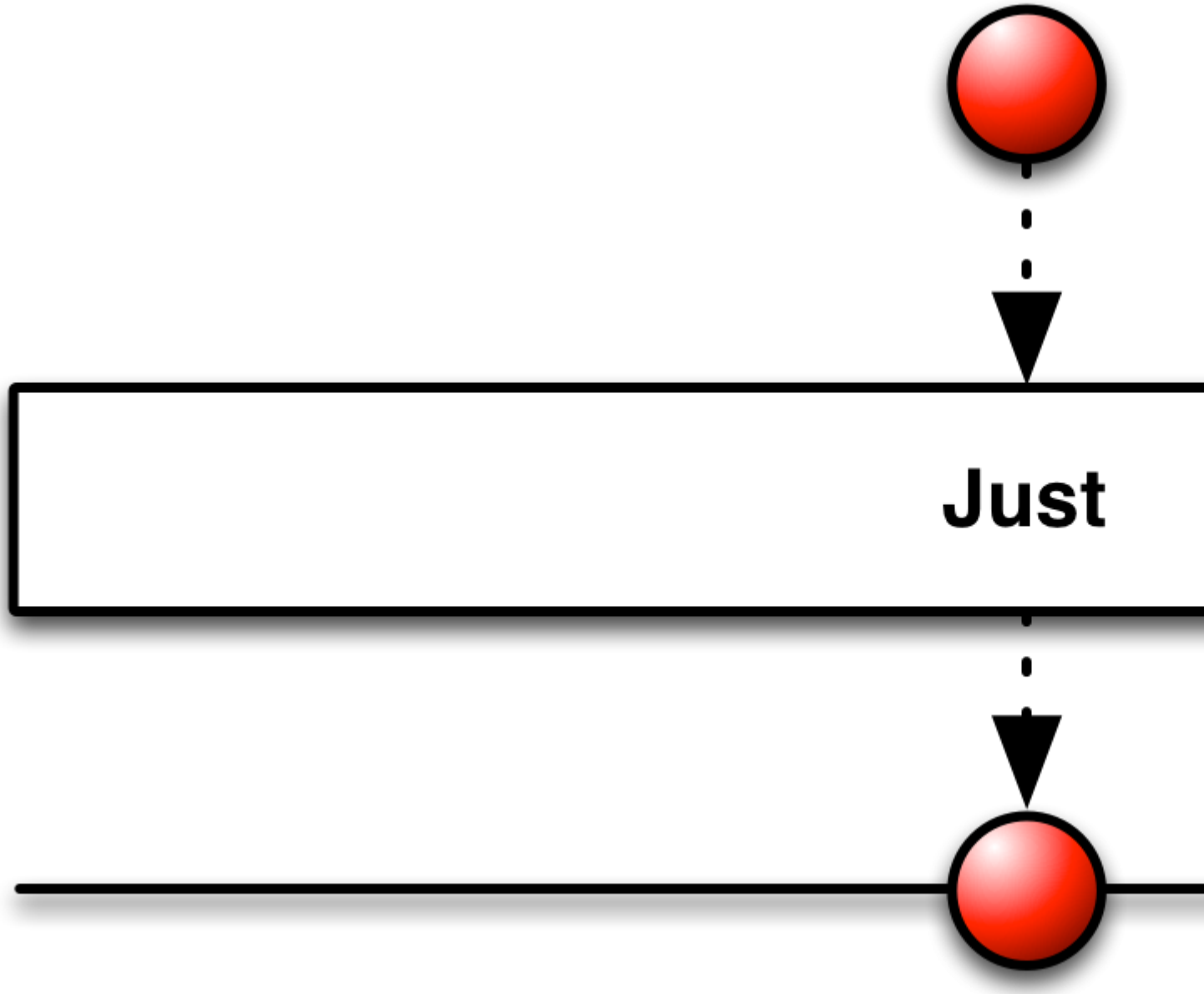
```
integerObservable.subscribe(mSubscriber);
```

Вышеприведенный при запуске будет производить следующий вывод:

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

Понимание мраморных диаграмм

Наблюдаемый можно рассматривать как просто поток событий. Когда вы определяете Observable, у вас есть три прослушивателя: onNext, onComplete и onError. onNext будет вызываться каждый раз, когда наблюдаемое приобретает новое значение. onComplete будет вызываться, если родительский Observable уведомит, что он завершил создание каких-либо других значений. onError вызывается, если исключение вызывается в любое время во время выполнения цепи Observable. Чтобы показать операторов в Rx, мраморная диаграмма используется для отображения того, что происходит с конкретной операцией. Ниже приведен пример простого наблюдаемого оператора «Just».



Мраморные диаграммы имеют горизонтальный блок, который представляет выполняемую операцию, вертикальную полосу для представления завершенного события, X для представления ошибки, а любая другая форма представляет значение. Имея это в виду, мы можем видеть, что «Just» просто возьмет наше значение и сделает onNext, а затем закончит с onComplete. Есть намного больше операций, затем просто «Just». Вы можете увидеть все операции, которые являются частью проекта ReactiveX, а также реализации в RxJava на сайте ReativeX . Существуют также интерактивные мраморные диаграммы через сайт RxMarbles .

Прочитайте Начало работы с rx-java онлайн: https://riptutorial.com/ru/rx-java/topic/974/начало-работы-с-rx-java

глава 2: Android с RxJava

замечания

RxAndroid раньше была библиотекой с множеством функций. Он был разделен во многих библиотеках, начиная от версии 0.25.0 до 1.x.

Список библиотек, реализующих функции, доступные до версии 1.0, поддерживается здесь .

Examples

RxAndroid - AndroidSchedulers

Это буквально единственное, что вам нужно, чтобы начать использовать RxJava на Android.

Включите RxJava и RxAndroid в зависимостях градиента:

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

Основное дополнение RxAndroid к RxJava - это планировщик для главной темы Android или пользовательского интерфейса.

В вашем коде:

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

Или вы можете создать Планировщик для пользовательского `Looper` :

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

Для большинства остальных вы можете обратиться к стандартной документации RxJava.

Компоненты RxLifecycle

Библиотека RxLifecycle упрощает привязку наблюдаемых подписки к действиям Android и жизненному циклу фрагмента.

Имейте в виду, что забывание отказаться от подписки Observable может привести к утечке памяти и сохранению вашего события активности / фрагмента после его разрушения системой.

Добавьте библиотеку в зависимости:

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

Затем расширяет классы `Rx*` :

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

Вы все настроены, когда подписываетесь на Observable, вы можете сейчас:

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

Если вы выполните это в `onCreate()` он автоматически `onDestroy()` в `onDestroy()` .

То же самое происходит и для:

- `onStart()` -> `onStop()`
- `onResume()` -> `onPause()`
- `onAttach()` -> `onDetach()` *(только для фрагмента)*
- `onViewCreated()` -> `onDestroyView()` *(только для фрагмента)*

В качестве альтернативы вы можете указать событие, когда хотите, чтобы произошла отмена подписки:

Из деятельности:

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

Из фрагмента:

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

Вы также можете получить наблюдаемый жизненный цикл с помощью метода `lifecycle()` для непосредственного прослушивания событий жизненного цикла.

RxLifecycle также может использоваться непосредственно, передавая ему наблюдаемый жизненный цикл:

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

Если вам нужно обрабатывать `Single` или `Completable` вы можете сделать это, просто добавив соответственно `forSingle()` или `forCompletable` после метода привязки:

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

Его также можно использовать с библиотекой Navi .

Rxpermissions

Эта библиотека позволяет использовать RxJava с новой моделью разрешения Android M.

Добавьте библиотеку в зависимости:

Rxjava

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

Rxjava2

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

использование

Пример (с Retrolambda для краткости, но не обязательно):

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

Подробнее: https://github.com/tbruyelle/RxPermissions .

Прочитайте Android с RxJava онлайн: https://riptutorial.com/ru/rx-java/topic/7125/android-c-rxjava

глава 3: RxJava2 Flowable и подписчик

Вступление

В этом разделе приведены примеры и документация в отношении реактивных концепций Flowable и Subscriber, которые были введены в rxjava version2

замечания

для примера нужен rxjava2 как зависимость, координаты maven для используемой версии:

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

Examples

пример потребителя-производителя с поддержкой противодавления у производителя

`TestProducer` из этого примера создает объекты `Integer` в заданном диапазоне и подталкивает их к своему `Subscriber` . Он расширяет класс `Flowable<Integer>` . Для нового абонента он создает объект `Subscription` , метод `request(long)` используется для создания и публикации значений Integer.

Важно, чтобы `Subscription` передана `subscriber` что метод `request()` который вызывает `onNext()` на подписчике, может быть рекурсивным образом вызван из этого `onNext()` . Для того, чтобы предотвратить переполнение стека, показанная реализация использует `outStandingRequests` счетчик и `isProducing` флаг.

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

Потребитель в этом примере расширяет `DefaultSubscriber<Integer>` и в начале и после использования Integer запрашивает следующий. При потреблении значений Integer существует небольшая задержка, поэтому противодавление будет создано для производителя.

```
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");
    }
}
```

в следующем основном методе тестового класса создаются и подключаются производитель и потребитель:

```
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();
    }
}
```

При запуске примера файл журнала показывает, что потребитель работает непрерывно, в то время как производитель только активируется, когда внутренний буфер Flowable rxjava2 необходимо пополнить.

Прочитайте RxJava2 Flowable и подписчик онлайн: https://riptutorial.com/ru/rx-java/topic/9810/rxjava2-flowable-и-подписчик

глава 4: Дооснащение и RxJava

Examples

Настроить дооснащение и RxJava

Retrofit2 поставляется с поддержкой нескольких подключаемых механизмов выполнения, один из которых - RxJava.

Чтобы использовать модификацию с RxJava, вам сначала нужно добавить адаптер Retrofit RxJava в свой проект:

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

то вам нужно добавить адаптер при создании экземпляра модификации:

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

В вашем интерфейсе, когда вы определяете API, возвращаемый тип должен быть `Observable` например:

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

Вы также можете использовать `Single` вместо `Observable` .

Выполнение последовательных запросов

RxJava удобен при выполнении последовательного запроса. Если вы хотите использовать результат из одного запроса для создания другого, вы можете использовать оператор `flatMap` :

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

Выполнение параллельных запросов

Вы можете использовать `zip` оператор для выполнения запроса параллельно и объединить результаты, например:

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

Прочитайте Дооснащение и RxJava онлайн: https://riptutorial.com/ru/rx-java/topic/2950/дооснащение-и-rxjava

глава 5: наблюдаемый

Examples

Создать наблюдаемый

Существует несколько способов создания Observable в RxJava. Самый мощный способ - использовать метод `Observable.create` . Но это также самый **сложный способ** . Поэтому вы должны **избегать его использования** , насколько это возможно.

Исходящее значение

Если у вас уже есть значение, вы можете использовать `Observable.just` чтобы исправить ваше значение.

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

Вычисление значения, которое должно быть вычислено

Если вы хотите испустить значение, которое еще не было вычислено, или что может потребоваться много времени для вычисления, вы можете использовать `Observable.fromCallable` для испускания следующего значения.

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

`longComputation()` будет вызываться только при подписке на ваш `Observable` . Таким образом, вычисление будет *ленивым* .

Альтернативный способ испускать значение, которое должно быть вычислено

`Observable.defer` создает `Observable` же, как `Observable.fromCallable` но используется, когда вам нужно вернуть `Observable` вместо значения. Это полезно, когда вы хотите управлять ошибками в своем вызове.

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

Горячие и холодные наблюдения

Наблюдения широко классифицируются как `Hot` или `Cold` , в зависимости от их поведения выбросов.
« `Cold Observable` - это тот, который начинает излучать по запросу (подписке), тогда как « `Hot Observable` - это тот, который испускает независимо от подписки.

Холодный наблюдаемый

```
/* 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
```

Вывод вышеуказанного кода выглядит (может варьироваться):

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

Обратите внимание: несмотря на то, что `sub2` запускается с опозданием, он получает значения с самого начала. В заключение, `Cold Observable` только испускает предметы по запросу. Несколько запросов запускают несколько конвейеров.

Горячие наблюдаемые

Примечание. Горячие наблюдаемые значения излучают независимо от отдельных подписей. У них есть своя временная шкала, и события происходят независимо от того, слушает кто-то или нет.

`Cold Observale` может быть преобразована в `Hot Observable` с простой `publish` .

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

`publish` возвращает `ConnectableObservable` который добавляет функции для *подключения* и *отключения* от наблюдаемого.

```
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));
```

Вышеизложенное дает:

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

Обратите внимание, что даже несмотря на то, что `sub2` начинает наблюдение в последнее время, он синхронизируется с `sub1` .
Отключение немного сложнее! Отключение происходит в `Subscription` а не в `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();
```

Вышеизложенное дает:

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

```
sub1, 0
...
```

При отключении `Observable` существу «завершается» и перезапускается при добавлении новой подписки.

`EventBus Hot Observable` может использоваться для создания `EventBus` . Такие EventBuses, как правило, легкие и супер быстрые. Единственным недостатком RxBus является то, что все события должны быть вручную реализованы и переданы в шину.

Прочитайте наблюдаемый онлайн: https://riptutorial.com/ru/rx-java/topic/1418/наблюдаемый

глава 6: Обратное давление

Examples

Вступление

Противодавление происходит, когда в конвейере обработки `Observable` некоторые асинхронные этапы не могут обрабатывать значения достаточно быстро и требуют способа замедлить работу восходящего производителя.

Классический случай необходимости противодавления заключается в том, что производитель является горячим источником:

```
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);
```

В этом примере основной поток будет производить 1 миллион элементов для конечного потребителя, который обрабатывает его на фоновом потоке. Вероятно, метод `compute(int)` занимает некоторое время, но накладные расходы цепной цепи `Observable` также могут увеличивать время, затрачиваемое на обработку элементов. Тем не менее, производящий поток с циклом for не может этого знать и продолжает `onNext` .

Внутри асинхронные операторы имеют буферы для хранения таких элементов до тех пор, пока они не будут обработаны. В классических Rx.NET и ранних RxJava эти буферы были неограниченными, что означает, что они, вероятно, будут содержать почти все 1 миллион элементов из примера. Проблема начинается, когда есть, например, 1 миллиард элементов или одна и та же 1 миллионная последовательность, которая появляется 1000 раз в программе, что приводит к `OutOfMemoryError` и, как правило, замедлению из-за чрезмерной нагрузки на `OutOfMemoryError` .

Подобно тому, как обработка ошибок стала первоклассным гражданином и получила операторов для работы с ней (через операторы `onErrorXXX`), противодавление - это еще одно свойство потоков данных, о которых программист должен думать и обрабатывать (через операторы `onBackpressureXXX`).

Помимо вышеперечисленного `PublishSubject` существуют другие операторы, которые не поддерживают противодавление, в основном из-за функциональных причин. Например,

`interval` оператора `interval` испускает значения, что приводит к сдвигу в периоде относительно настенных часов.

В современной RxJava большинство асинхронных операторов теперь имеют ограниченный внутренний буфер, например, как `observeOn` выше, и любая попытка переполнения этого буфера завершает всю последовательность с помощью `MissingBackpressureException` . В документации каждого оператора есть описание противодавления.

Однако противодавление присутствует более тонко в обычных холодных последовательностях (которые не дают и не должны давать `MissingBackpressureException`). Если первый пример переписан:

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

Thread.sleep(10_000);
```

Нет ошибки, и все работает плавно при использовании небольшой памяти. Причиной этого является то, что многие операторы источников могут «генерировать» значения по требованию, и, таким образом, `observeOn` за оператором. Можно сказать, что `range` генерирует не более чем столько значений, `observeOn` буфер `observeOn` может удерживать сразу без переполнения.

Это согласование основано на концепции компьютерной науки о совместных подпрограммах (я называю вас, вы называете меня). Оператор `range` посылает обратный вызов, в форме осуществления `Producer` интерфейса, к `observeOn` путем вызова (внутренний `Subscriber` «ов) `setProducer` . В свою очередь, `observeOn` вызывает `Producer.request(n)` со значением, чтобы сообщить `range` которому разрешено создавать (т. `onNext` it), что многие **дополнительные** элементы. Именно тогда ответственность `observeOn` заключается в `observeOn` , чтобы вызвать метод `request` в нужное время и с правильным значением, чтобы сохранить поток данных, но не переполняться.

Выражение противодавления в конечных потребителях редко необходимо (поскольку они синхронны по отношению к их непосредственному восходящему и обратному давлению, естественно, происходит из-за блокировки стека вызовов), но может быть легче понять его работу:

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

Здесь реализация `onStart` указывает `range` для создания своего первого значения, которое затем принимается в `onNext` . После завершения `compute(int)` другое значение запрашивается из `range` . В наивной реализации `range` такой вызов рекурсивно вызывает `onNext` , что приводит к `StackOverflowError` что, конечно, нежелательно.

Чтобы предотвратить это, операторы используют так называемую логику батутинга, которая предотвращает такие повторные вызовы. В терминах `range` он будет помнить, что был `request(1)` когда он вызывал `onNext()` и как только `onNext()` возвращает, он сделает другой раунд и вызовет `onNext()` со следующим целочисленным значением. Поэтому, если они меняются, пример по-прежнему работает одинаково:

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

    compute(v);
}
```

Однако это не так для `onStart` . Хотя инфраструктура `Observable` гарантирует, что она будет вызываться не более одного раза на каждого `Subscriber` , запрос на `request(1)` может сразу вызвать эмиссию элемента. Если у вас есть логика инициализации после запроса на `request(1)` который необходим для `onNext` , у вас могут быть исключения:

```
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
});
```

В этом синхронном случае `onStart` `NullPointerException` будет `onStart` немедленно, пока выполняется `onStart` . Более тонкая ошибка возникает, если запрос на `request(1)` вызывает асинхронный вызов `onNext` в другом потоке и чтение `name` в `onNext` гонках, записывающих его в `onStart` post `request` .

Таким образом, нужно выполнить всю инициализацию поля в `onStart` или даже до этого и `request()` последним. Реализации `request()` в операторах обеспечивают надлежащее выполнение - до отношения (или, в других отношениях, освобождения памяти или полного забора), когда это необходимо.

Операторы onBackpressureXXX

Большинство разработчиков сталкиваются с противодавлением, когда их приложение терпит неудачу с `MissingBackpressureException` и исключение обычно указывает на оператор `observeOn` . Фактической причиной обычно является не- `PublishSubject` использование `PublishSubject` , `timer()` или `interval()` или пользовательских операторов, созданных с помощью `create()` .

Существует несколько способов решения таких ситуаций.

Увеличение размеров буфера

Иногда такие переполнения происходят из-за взрывоопасных источников. Внезапно пользователь слишком быстро `observeOn` экран и `observeOn` за внутренним буфером 16-элементного стандарта по умолчанию на переполнениях Android.

Большинство чувствительных к давлению операторов в последних версиях RxJava теперь позволяют программистам указывать размер своих внутренних буферов. Соответствующие параметры обычно называются `bufferSize` , `prefetch` или `capacityHint` . Учитывая переполненный пример во введении, мы можем просто увеличить размер буфера для `observeOn` чтобы иметь достаточно места для всех значений.

```
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);
}
```

Обратите внимание, однако, что в общем случае это может быть только временное исправление, поскольку переполнение может все же произойти, если источник

перепроизводит размер предсказанного буфера. В этом случае можно использовать один из следующих операторов.

Группировка / пропускание значений со стандартными операторами

В случае, если исходные данные могут быть обработаны более эффективно в пакетном режиме, можно уменьшить вероятность `MissingBackpressureException` с использованием одного из стандартных операторов `MissingBackpressureException` обработки (по размеру и / или по времени).

```
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);
}
```

Если некоторые из значений можно безопасно игнорировать, можно использовать выборку (со временем или другим наблюдаемым) и операторы дросселирования (`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);
}
```

Обратите внимание на то, что эти операторы только снижают скорость приема значений по нисходящему потоку и, следовательно, могут по-прежнему приводить к `MissingBackpressureException` .

onBackpressureBuffer ()

Этот оператор в своей безпараметрической форме повторно вводит неограниченный буфер между исходным источником и оператором нисходящего потока. Быть неограниченным означает, что пока JVM не исчерпывает память, он может обрабатывать практически любую сумму, поступающую из разрывающегося источника.

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

В этом примере `observeOn` идет с очень низким размером буфера, но отсутствует `MissingBackpressureException` как `onBackpressureBuffer` впитывает все 1 миллион значений и передает небольшим партиям его для `observeOn` .

Обратите внимание, однако, что `onBackpressureBuffer` потребляет свой источник неограниченным образом, то есть без применения к нему противодавления. Это приводит к тому , что даже противодавлению поддерживающего источник , таким как `range` будет полностью реализован.

Есть 4 дополнительных перегрузки `onBackpressureBuffer`

onBackpressureBuffer (int capacity)

Это ограниченная версия, которая сигнализирует `BufferOverflowError` в том случае, если ее буфер достигает заданной емкости.

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

Уместность этого оператора уменьшается, поскольку все больше и больше операторов теперь позволяют устанавливать размеры своих буферов. В остальном это дает возможность «расширить свой внутренний буфер», имея большее число с `onBackpressureBuffer` чем их значение по умолчанию.

onBackpressureBuffer (int capacity, Action0 onOverflow)

Эта перегрузка вызывает (совместное) действие в случае переполнения. Его полезность довольно ограничена, поскольку нет другой информации о переполнении, чем текущий стек вызовов.

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

Эта перегрузка на самом деле более полезна, поскольку позволяет определить, что делать, если емкость была достигнута. `BackpressureOverflow.Strategy` - это интерфейс на самом деле, но класс `BackpressureOverflow` предлагает 4 статических поля, реализация которых представляет собой типичные действия:

- `ON_OVERFLOW_ERROR` : это поведение по умолчанию предыдущих двух перегрузок, сигнализирующее об `BufferOverflowException`

- `ON_OVERFLOW_DEFAULT` : в настоящее время это то же самое, что и `ON_OVERFLOW_ERROR`
- `ON_OVERFLOW_DROP_LATEST` : если произойдет переполнение, текущее значение будет просто проигнорировано, и только старые значения будут доставлены после запросов нисходящего потока.
- `ON_OVERFLOW_DROP_OLDEST` : удаляет самый старый элемент в буфере и добавляет к нему текущее значение.

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

Обратите внимание, что последние две стратегии вызывают разрыв в потоке по мере удаления элементов. Кроме того, они не будут сигнализировать об `BufferOverflowException` .

onBackpressureDrop ()

Всякий раз, когда нисходящий поток не готов принимать значения, этот оператор отбрасывает этот элемент из последовательности. Можно подумать об этом как о вместимости 0 `onBackpressureBuffer` со стратегией `ON_OVERFLOW_DROP_LATEST` .

Этот оператор полезен, когда можно смело игнорировать значения из источника (например, перемещения мыши или текущие сигналы GPS-местоположения), так как в дальнейшем они будут иметь более современные значения.

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

Это может быть полезно в сочетании с `interval()` оператора источника `interval()` . Например, если вы хотите выполнить некоторую периодическую фоновую задачу, но каждая итерация может длиться дольше, чем период, можно с уверенностью отказаться от уведомления об избыточном интервале, поскольку будет более поздно:

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

Существует одна перегрузка этого оператора: `onBackpressureDrop(Action1<? super T> onDrop)` где `onBackpressureDrop(Action1<? super T> onDrop)` (совместно используемое) действие с отбрасываемым значением. Этот вариант позволяет самостоятельно очищать значения (например, освобождать связанные ресурсы).

onBackpressureLatest ()

Конечный оператор сохраняет только последнее значение и практически перезаписывает старые, недопустимые значения. Можно подумать об этом как о варианте `onBackpressureBuffer` с емкостью 1 и стратегии `ON_OVERFLOW_DROP_OLDEST` .

В отличие от `onBackpressureDrop` всегда есть ценность, доступная для потребления, если нисходящий `onBackpressureDrop` оказался отстающим. Это может быть полезно в некоторых ситуациях, связанных с телеметрией, где данные могут возникать в виде нескольких всплесков, но только самые последние интересны для обработки.

Например, если пользователь нажимает много на экране, мы все равно хотим отреагировать на его последний ввод.

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

Использование `onBackpressureDrop` в этом случае приведет к ситуации, когда последний клик будет сброшен и не позволит пользователю задаться вопросом, почему бизнес-логика не была выполнена.

Создание резервных источников данных

Создание резервных источников данных является относительно простой задачей при обратном противодавлении вообще, потому что библиотека уже предлагает статические методы `Observable` которые обрабатывают противодавление для разработчика. Мы можем различать два типа заводских методов: холодные «генераторы», которые либо возвращают, либо генерируют элементы, основанные на потреблении вниз по течению и горячих «толкателях», которые обычно соединяют нереактивные и / или невосстановимые источники данных и накладывают некоторое обратное давление на верх их.

просто

Самый основное противодавление известен источник создается с помощью `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
}
```

Поскольку мы явно не запрашиваем в `onStart` , это ничего не печатает. `just` отлично, когда есть постоянная ценность, которую мы хотели бы начать с начала.

К сожалению, `just` ошибочно принимают за способ вычислить что-то динамически, чтобы потреблять `Subscriber` s:

```
int counter;

int computeValue() {
   return ++counter;
}

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

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

Удивительно для некоторых, это печатает 1 дважды вместо печати 1 и 2 соответственно. Если вызов переписан, становится очевидным, почему он работает так:

```
int temp = computeValue();

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

`computeValue` называется частью основной процедуры, а не в ответ на подписку подписчиков.

fromCallable

То, что действительно нужно людям, это метод `fromCallable` :

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

Здесь `computeValue` выполняется только тогда, когда абонент подписывается и для каждого из них печатает ожидаемые 1 и 2. Естественно, `fromCallable` также правильно поддерживает противодавление и не будет `fromCallable` вычисленное значение, если не запрошено. Обратите внимание, однако, что вычисления все равно происходят. В случае, если само вычисление должно быть отложено до тех пор, пока на самом деле не потребуются нисходящие потоки, мы можем использовать `just` с `map` :

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

`just` не будет выдавать свое постоянное значение до тех пор, пока не будет запрошено, когда оно будет сопоставлено с результатом `computeValue` , все равно будет `computeValue` для

каждого абонента индивидуально.

от

Если данные уже доступны в виде массива объектов, список объектов или любого `Iterable` источника, соответствующий `from` перегрузок будет обрабатывать противодавления и излучение таких источников:

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

Для удобства (и избежать предупреждений о создании общего массива) есть 2 до 10 аргументов перегруженных к `just` что внутренне делегируют к `from` .

The `from(Iterable)` также дает интересную возможность. Многие генерирующие ценности могут быть выражены в форме государственной машины. Каждый запрошенный элемент запускает переход состояния и вычисляет возвращаемое значение.

Написание таких состояний машин, как `Iterable` s, несколько сложнее (но все же проще, чем писать `Observable` для его потребления), и в отличие от C #, Java не имеет никакой поддержки от компилятора для создания таких состояний машин, просто написав классически выглядящий код (с `yield return` и `yield break`). Некоторые библиотеки предлагают некоторую помощь, такие как Google гуавы в `AbstractIterable` и IxJava в `Ix.generate()` и `Ix.forloop()` . Они сами по себе достойны полной серии, поэтому давайте посмотрим на очень простой исходный источник `Iterable` который бесконечно повторяет некоторое постоянное значение:

```
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);
```

Если бы мы использовали `iterator` через классический цикл for, это привело бы к бесконечному циклу. Поскольку мы создаем `Observable` из этого, мы можем выразить нашу волю потреблять только первые 5 из них, а затем прекратить запрашивать что-либо. Это истинная сила ленивой оценки и вычисления внутри `Observable` s.

создание (SyncOnSubscribe)

Иногда источник данных, который должен быть преобразован в самый реактивный мир, является синхронным (блокирующим) и pull-like, то есть мы должны вызвать некоторый метод `get` или `read` чтобы получить следующий фрагмент данных. Разумеется, можно было бы превратить это в `Iterable` но когда такие источники связаны с ресурсами, мы можем утечка этих ресурсов, если нисходящий поток не подписывает последовательность до ее окончания.

Для обработки таких случаев RxJava имеет класс `SyncOnSubscribe` . Его можно расширить, реализовать его методы или использовать один из его основанных на лямбдах методов для создания экземпляра.

```
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);
```

Как правило, `SyncOnSubscribe` использует 3 обратных вызова.

Первые обратные вызовы позволяют создать состояние для каждого абонента, такое как `FileInputStream` в примере; файл будет открыт независимо каждому отдельному абоненту.

Второй обратный вызов принимает этот объект состояния и предоставляет выходной `Observer` чьи методы `onXXX` могут быть вызваны для испускания значений. Этот обратный вызов выполняется столько раз, сколько запрашивается нисходящий поток. При каждом вызове он должен вызывать `onNext` не более одного раза, по выбору, либо с помощью `onError` либо `onCompleted` . В примере мы вызываем `onCompleted()` если прочитанный байт отрицателен, указывает и заканчивает файл, и вызывает `onError` в случае, если read вызывает `onError IOException` .

Последний обратный вызов активируется, когда нисходящий поток отменяет подписку

(закрытие входного потока) или когда предыдущий обратный вызов называется терминальными методами; он позволяет освободить ресурсы. Поскольку не всем источникам нужны все эти функции, статические методы `SyncOnSubscribe` позволяют создавать экземпляры без них.

К сожалению, многие вызовы методов в JVM и других библиотеках выдают проверенные исключения и должны быть завернуты в `try-catch` es, поскольку функциональные интерфейсы, используемые этим классом, не позволяют бросать проверенные исключения.

Конечно, мы можем имитировать другие типичные источники, такие как неограниченный диапазон:

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

В этой настройке `current` начинается с `0` а в следующий раз, когда вызывается лямбда, `current` равен `1` .

Существует вариант `SyncOnSubscribe` под названием `AsyncOnSubscribe` который выглядит очень похожим, за исключением того, что средний обратный вызов также принимает длинное значение, которое представляет собой сумму запроса из нисходящего потока, а обратный вызов должен генерировать `Observable` с такой же длиной. Этот источник затем объединяет все эти `Observable` в одну последовательность.

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

Существует продолжительная (горячая) дискуссия о пользе этого класса и вообще не рекомендуется, поскольку она регулярно нарушает ожидания относительно того, как она действительно будет генерировать эти сгенерированные ценности и как она будет реагировать, или даже какие значения запроса, которые она получит в более сложные потребительские сценарии.

создание (эмиттер)

Иногда источник, который должен быть обернут в `Observable` , уже горячий (например,

перемещение мыши) или холодный, но не протирающийся в его API (такой как асинхронный обратный вызов сети).

Чтобы обработать такие случаи, в последней версии RxJava был введен заводский метод `create(emitter)` . Он принимает два параметра:

- обратный вызов, который вызывается с экземпляром интерфейса `Emitter<T>` для каждого входящего абонента,
- перечисление `Emitter.BackpressureMode` которое обязывает разработчика указывать поведение противодавления, которое должно применяться. Он имеет обычные режимы, похожие на `onBackpressureXXX` в дополнение к `MissingBackpressureException` или просто игнорируя такое переполнение внутри него.

Обратите внимание, что в настоящее время он не поддерживает дополнительные параметры для этих режимов противодавления. Если вам нужна эта настройка, использование `NONE` в качестве режима противодавления и применение соответствующего `onBackpressureXXX` на полученном `Observable` - это путь.

Первый типичный случай для его использования, когда вы хотите взаимодействовать с источником на основе push, таким как события GUI. Эти API имеют некоторую форму `addListener` / `removeListener` которые можно использовать:

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

    button.addActionListener(al);

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

}, BackpressureMode.BUFFER);
```

`Emitter` относительно прост в использовании; можно вызвать `onNext` , `onError` и `onCompleted` а оператор самостоятельно обрабатывает управление противодавлением и отменой подписки. Кроме того, если обернута API - поддерживает отмену (например, удаление слушателя в данном примере), можно использовать `setCancellation` (или `setSubscription` для `Subscription` -подобных ресурсов) для регистрации отмены обратного вызова , которая вызывается , когда вниз по течению отписывается или `onError` / `onCompleted` вызывается на предоставленный экземпляр `Emitter` .

Эти методы позволяют одновременно связывать только один ресурс с эмиттером и устанавливать новый, который автоматически не подписывается на старый. Если нужно обрабатывать несколько ресурсов, создайте `CompositeSubscription` , свяжите ее с эмиттером, а затем добавьте дополнительные ресурсы в саму `CompositeSubscription` подписку:

```
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);
```

Второй сценарий обычно включает в себя некоторый асинхронный API с обратным вызовом, который должен быть преобразован в `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);
```

В этом случае делегация работает одинаково. К сожалению, как правило, эти классические API обратного вызова не поддерживают отмену, но если это так, можно настроить их отмену, как в примерах previoius (хотя, возможно, более увлекательный способ). Обратите внимание на использование режима `LATEST` противодавления; если мы знаем, что будет только одно значение, нам не нужна стратегия `BUFFER` , поскольку он выделяет стандартный буфер длиной 128 элементов (который растет по мере необходимости), который никогда не будет полностью использован.

Прочитайте Обратное давление онлайн: https://riptutorial.com/ru/rx-java/topic/2341/обратное-давление

глава 7: операторы

замечания

В этом документе описывается основное поведение оператора.

Examples

Операторы, введение

Оператор может использоваться для управления потоком объектов от `Observable` to `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);
```

Результатом будет:

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

Оператор `map` изменил наблюдаемое `Integer` наблюдаемый `String` , тем самым манипулируя потоком объектов.

Цепочка операторов

Несколько операторов могут быть `chained` вместе для более мощных преобразований и манипуляций.

```
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
```

Любое число операторов может быть добавлено между `Observable` и `Subscriber` .

Оператор FlatMap

Оператор `flatMap` помогает преобразовать одно событие в другое `Observable` (или преобразовать событие в ноль, один или несколько событий).

Это идеальный оператор, если вы хотите вызвать другой метод, который возвращает `Observable`

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

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

`flatMap` будет сериализовать `perform` подписки, **но** события, выдаваемые `perform` не могут быть заказаны. Таким образом , вы можете получить события , испускаемые последним выполнение вызова , **прежде чем** события от первого `perform` вызова (вы должны использовать `concatMap` вместо этого).

Если вы создаете еще один `Observable` в своем подписчике, вместо этого вы **должны** использовать `flatMap` . Основная идея: **никогда не покидайте Наблюдаемый**

Например :

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

можно легко заменить на:

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

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

фильтр-оператор

Вы можете использовать оператор `filter` для фильтрации элементов из потока значений на основе результата предикатного метода.

Другими словами, элементы, передаваемые от наблюдателя к подписчику, будут отброшены на основе `filter` , который вы передадите, если функция возвращает `false` для определенного значения, это значение будет отфильтровано.

Пример:

```
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
    });
```

Этот код распечатает

```
0.0
4.0
16.0
36.0
64.0
```

Оператор карты

Вы можете использовать оператор `map` для сопоставления значений потока с разными значениями в зависимости от результата для каждого значения от функции, переданной на `map` . Исходный поток представляет собой новую копию и не будет изменять предоставленный поток значений, поток результатов будет иметь одинаковую длину входного потока, но может быть разных типов.

Функция, переданная в `.map()` , должна вернуть значение.

Пример:

```
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
    });
```

Этот код распечатает

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

В этом примере Observable принял `List<Integer>` список будет преобразован в `List<String>` в конвейере, а `.subscribe` будет `.subscribe String` 's

Оператор doOnNext

Оператор `doOnNext` каждый раз, когда источник `Observable` испускает элемент. Он может использоваться для целей отладки, применяя некоторые действия к испускаемому элементу, протоколированию и т. Д. ...

```
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();
```

В приведенном ниже примере `doOnNext` никогда не вызывается, потому что источник `Observable` испускает ничего, потому что `Observable.empty()` вызывает `onCompleted` после подписки.

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

повторить оператор

оператор `repeat` позволяет повторить целую последовательность из источника `Observable` .

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

```
    );
```

Вывод примера выше

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

Эта последовательность повторяется бесконечно много раз и никогда не завершается.

Для повторения последовательности конечное число раз просто передайте целое число в качестве аргумента для `repeat` оператора.

```
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")
    );
```

Этот пример печатает

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

Это очень важно понимать , что `repeat` оператора resubscribes к источнику `Observable` когда источник `Observable` последовательность завершается. Давайте перепишем пример выше, используя `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")
        );
```

Этот пример печатает

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

При использовании оператора сцепления, важно знать , что `repeat` оператор повторяет **всю последовательность** , а не предшествующий оператор.

```
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")
        );
```

Этот пример печатает

```
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
```

Этот пример показывает , что `repeat` оператор повторяет всю последовательность в

Повторная подписка `Observable` , а не повторяющуюся последнюю `map` оператора , и это не имеет значения , в каком месте в последовательности `repeat` оператора используется.

Эта последовательность

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

равна этой последовательности

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

Прочитайте операторы онлайн: https://riptutorial.com/ru/rx-java/topic/2316/операторы

глава 8: Планировщики

Examples

Основные примеры

Планировщики - это абстракция RxJava о процессоре. Планировщик может быть защищен службой Executor, но вы можете реализовать свою собственную реализацию планировщика.

`Scheduler` должен соответствовать этому требованию:

- Должен обрабатывать неустановленную задачу последовательно (порядок FIFO)
- Задание может быть отложено

`Scheduler` может использоваться как параметр для некоторых операторов (пример: `delay`) или используется с методом `subscribeOn` / `observeOn` .

С помощью некоторого оператора `Scheduler` будет использоваться для обработки задачи конкретного оператора. Например, `delay` будет планировать отложенную задачу, которая испустит следующее значение. Это `Scheduler` , который сохранит и выполнит его позже.

`subscribeOn` можно использовать один раз для `Observable` . Он определит, в каком `Scheduler` будет выполняться код подписки.

`observeOn` может быть использовано несколько раз за `Observable` . Он определит, в каком `Scheduler` будет использоваться для выполнения всех задач, определенных **после** метода `observeOn` . `observeOn` поможет вам выполнить скачок резьбы.

subscribeOn конкретный планировщик

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

наблюдать со специальным планировщиком

```
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()));
```

Указание конкретного Планировщика с оператором

Некоторые операторы могут принимать параметр `Scheduler` качестве параметра.

```
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);
```

Опубликовать подписчику:

```
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);
```

Пул потоков:

```
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);
```

Наблюдаемый веб-разъем:

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

Прочитайте Планировщики онлайн: https://riptutorial.com/ru/rx-java/topic/2321/планировщики

глава 9: Предметы

Синтаксис

- Тема <T, R> subject = AsyncSubject.create (); // По умолчанию AsyncSubject
- Тема <T, R> subject = BehaviorSubject.create (); // Default BehaviorSubject
- Тема <T, R> subject = PublishSubject.create (); // Default PublishSubject
- Тема <T, R> subject = ReplaySubject.create (); // По умолчанию ReplaySubject
- mySafeSubject = новый SerializedSubject (unSafeSubject); // Преобразование unsafeSubject в safeSubject - как правило, для многопоточных объектов

параметры

параметры	подробности
T	Тип ввода
p	Тип выхода

замечания

В этой документации приводятся подробные сведения и пояснения по `Subject` . Для получения дополнительной информации и дальнейшего ознакомления, пожалуйста, посетите официальную документацию .

Examples

Основные темы

`Subject` в RxJava - это класс, который является `Observable` и `Observer` . Это в основном означает, что он может выступать в качестве `Observable` и передавать входные данные подписчикам и в качестве `Observer` получать данные от другого наблюдаемого.

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

Вышеприведенные отпечатки «Привет, мир!» для консольного использования `Subjects` .

объяснение

1. Первая строка кода определяет новый `Subject` типа `PublishSubject`

```
Subject<String, String> subject = PublishSubject.create();
    |      |       |       |                  |
 subject<input, output>  name   = default publish subject
```

2. Вторая строка подписывается на объект, отображая поведение `Observer` .

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

   Это позволяет `Subject` принимать входные данные, такие как обычный абонент

3. Третья строка вызывает метод `onNext` объекта, отображающий поведение `Observable` .

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

   Это позволяет `Subject` давать входные данные всем подписчикам на него.

Типы

`Subject` (в RxJava) может быть любого из этих четырех типов:

- AsyncSubject
- BehaviorSubject
- PublishSubject
- ReplaySubject

Кроме того, объект `Subject` может иметь тип `SerializedSubject` . Этот тип гарантирует, что `Subject` не нарушит *Договор* о *наблюдении* (который указывает, что все вызовы должны быть сериализованы)

Дальнейшее чтение:

- Использовать или не использовать тему из блога Дейва Секстона

PublishSubject

`PublishSubject` `Observer` только те элементы, которые испускаются источником. `Observable` после времени подписки.

Простой пример `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);
```

Выход:

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

В приведенном выше примере `PublishSubject` подписывается на `Observable` который действует как часы, и испускает элементы (Long) каждые 500 миллисекунд. Как видно на выходе, `PublishSubject` передает значения, которые он получает от источника (`clock`) до его подписчиков (`sub1` и `sub2`).

`PublishSubject` может запускать испускающие элементы, как только он будет создан, без какого-либо наблюдателя, который рискует потерять один или несколько предметов до тех пор, пока наблюдатель не сможет записаться.

```
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
```

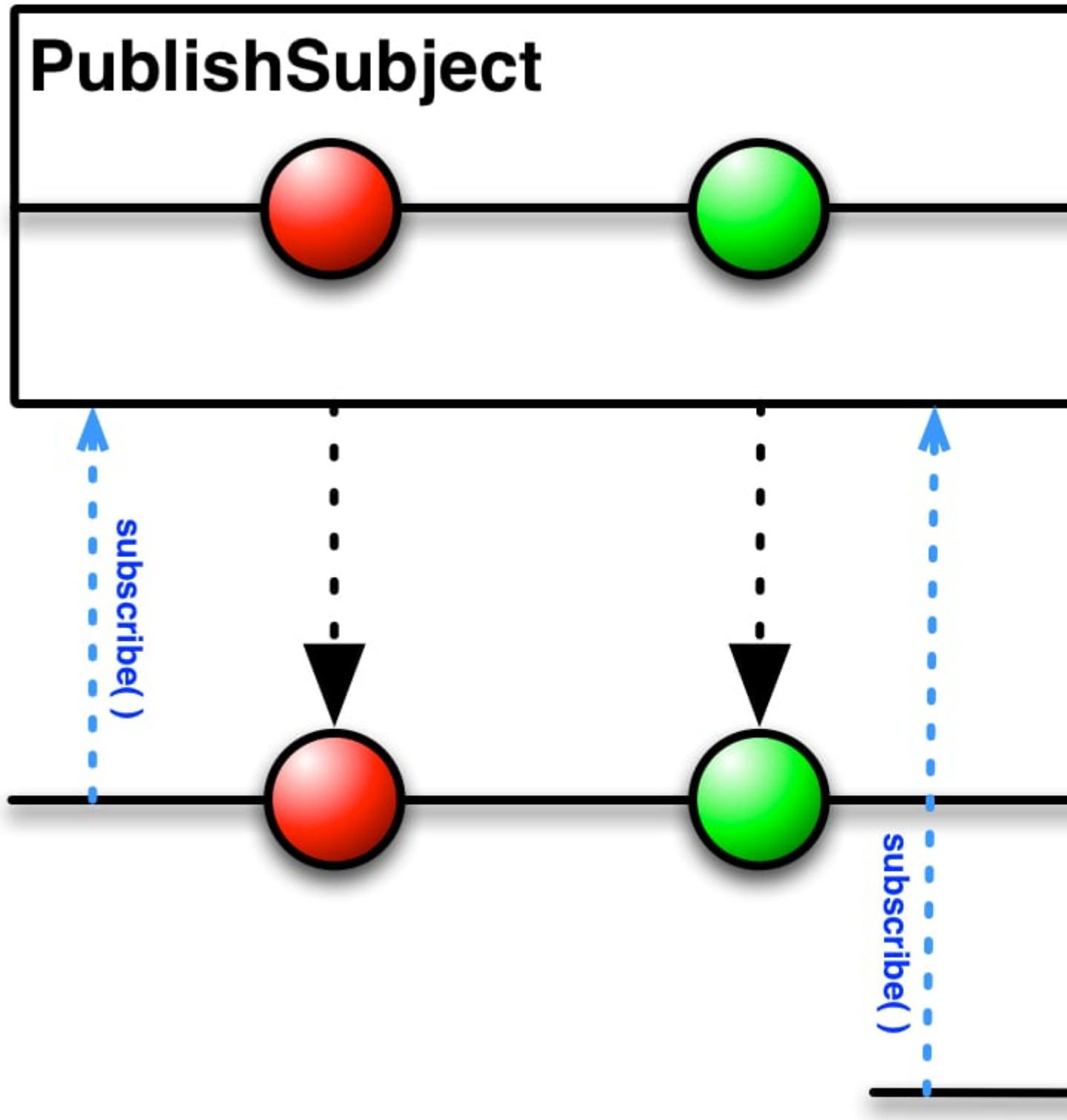
Выход:

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

Обратите внимание, что `sub1` испускает значения, начиная с `10` . Внесенная 5-секундная задержка вызвала *потерю* предметов. Они не могут воспроизводиться. Это по существу делает `PublishSubject` `Hot Observable` .

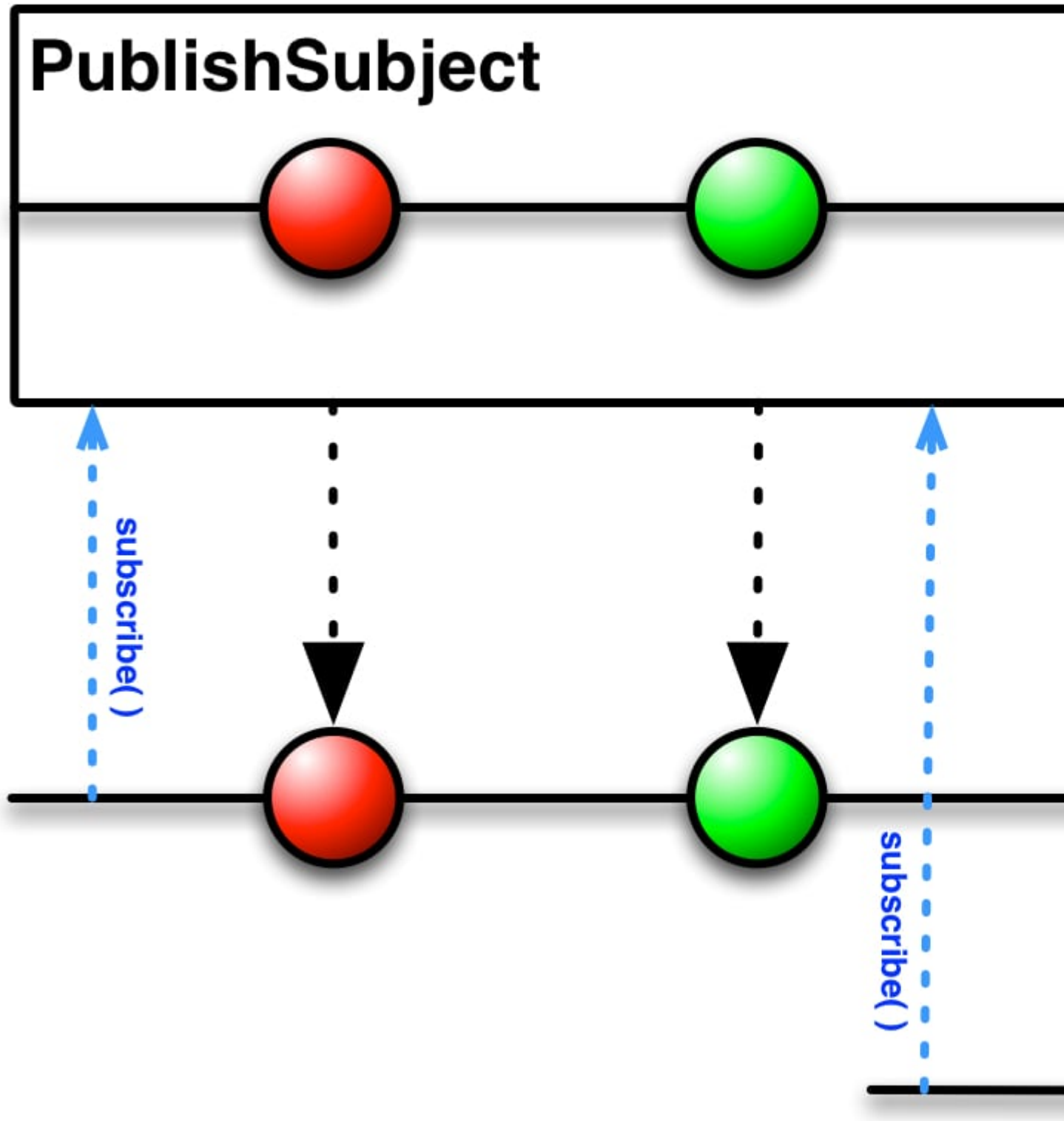
Также обратите внимание, что если наблюдатель подписывается на `PublishSubject` после того, как он выпустил ***n*** элементов, эти ***n*** элементов *не могут* быть воспроизведены для этого наблюдателя.

Ниже представлена мраморная диаграмма `PublishSubject`



`PublishSubject` элементы всем, кто подписался, в любой момент времени до `onCompleted` источника `Observable` .

Если источник `Observable` завершается с ошибкой, `PublishSubject` не будет `PublishSubject` какие-либо элементы последующим наблюдателям, а просто будет передавать уведомление об ошибке из источника Observable.



Случай использования

Предположим, вы хотите создать приложение, которое будет контролировать цены акций определенной компании и перенаправлять их всем клиентам, которые запрашивают ее.

```
/* 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();
```

В приведенном выше примере использования, `PublishSubject` действует как мост для передачи значений с вашего сервера всем клиентам, которые подписываются на ваш `watcher` .

Дальнейшее чтение:

- PublishSubject javadocs
- Блог Томаса Нильда (расширенное чтение)

Прочитайте Предметы онлайн: https://riptutorial.com/ru/rx-java/topic/3287/предметы

глава 10: Тестирование устройства

замечания

Поскольку все методы Schedulers статичны, модульные тесты, использующие крючки RxJava, не могут запускаться параллельно в одном экземпляре JVM. Если они, где один TestScheduler будет удален в середине единичного теста. В основном это недостаток использования класса Schedulers.

Examples

TestSubscriber

TestSubscribers позволяют избежать работы, создающей вашего собственного Абонента, или подписаться на Action <?>, Чтобы убедиться, что определенные значения, которые были доставлены, сколько есть, если Observable завершено, исключение было поднято и намного больше.

Начиная

Этот пример просто показывает утверждение, что значения 1,2,3 и 4, которые передаются в Observable через onNext.

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

`assertValues` утверждает, что подсчет верен. Если вы должны были передавать только некоторые из значений, утверждение будет терпеть неудачу.

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

`assertValues` использует метод `equals` при выполнении `assertValues` . Это позволяет легко тестировать классы, которые рассматриваются как данные.

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

В этом примере показан класс, который имеет равные значения и утверждает значения из Observable.

```
public class Room {

    public String floor;
    public String number;

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

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

Также обратите внимание, что мы используем более короткий `assertValue` потому что нам нужно только проверить один элемент.

Получение всех событий

В случае необходимости вы также можете запросить все события в виде списка.

```
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();
```

Утверждение событий

Если вы хотите сделать более обширные тесты своих событий, вы можете комбинировать `getOnNextEvents` (или `getOn*Events`) с вашей любимой библиотекой утверждений:

```
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));
```

Observable#error проверки Observable#error

Вы можете убедиться, что исправит правильный класс исключения:

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

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

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

Вы также можете убедиться, что было выбрано точное исключение:

```
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 позволяет вам контролировать время и выполнение Observables вместо того, чтобы делать занятые ожидания, присоединять потоки или что-либо, чтобы манипулировать системным временем. Это ОЧЕНЬ важно, если вы хотите написать единичные тесты, которые являются предсказуемыми, последовательными и быстрыми. Поскольку вы манипулируете временем, нет больше шансов, что нить станет голодной, что ваш тест завершится неудачей на более медленной машине или что вы тратите время на занятие, ожидая результата.

TestSchedulers могут быть предоставлены через перегрузку, которая принимает Планировщик для любых операций 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
```

TestScheduler довольно простой. Он состоит только из трех методов.

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

Это позволяет вам манипулировать, когда TestScheduler должен запускать все действия, относящиеся к некоторому времени в будущем.

При прохождении планировщика это не так, как обычно используется TestScheduler из-за того, насколько он неэффективен. Передача планировщиков в классы заканчивается предоставлением большого количества дополнительного кода для небольшого выигрыша. Вместо этого вы можете подключиться к плагину RxJava Schedulers.io () / computation () / etc. Это делается с помощью крючков RxJava. Это позволяет вам определить, что возвращается от вызова от одного из методов Schedulers.

```
public final class TestSchedulers {

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

Этот класс позволяет пользователю получить планировщик тестирования, который будет подключен для всех вызовов Планировщиков. Единичный тест просто должен был получить этот планировщик в его настройке. Настоятельно рекомендуется использовать его в настройке, а не как любое обычное старое поле, потому что ваш TestScheduler может попытаться вызвать триггер из другого модульного теста, когда вы продвигаете время. Теперь наш пример выше становится

```
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)
```

Вот как вы можете эффективно удалить системные часы из вашего модульного теста (по

крайней мере, до RxJava))

Прочитайте Тестирование устройства онлайн: https://riptutorial.com/ru/rx-java/topic/5207/тестирование-устройства

кредиты

S. No	Главы	Contributors
1	Начало работы с rx-java	Buttink, Community, dimsuz, Dmitry Avtonomov, Hans Wurst, hello_world, Omar Al Halabi, Saulius Next, Sneh Pandya, svarog, Tom
2	Android с RxJava	akarnokd, Athafoud, Daniele Segato, Eugen Martynov, Geng Jiawen, Sneh Pandya
3	RxJava2 Flowable и подписчик	P.J.Meisch
4	Дооснащение и RxJava	LordRaydenMK
5	наблюдаемый	Aki K, dwursteisen, hello_world, JonesV
6	Обратное давление	akarnokd, Bartek Lipinski, Chris A, Cristian, dwursteisen, Niklas, Sebas LG
7	операторы	dwursteisen, hello_world, svarog, Vadeg
8	Планировщики	dwursteisen, Gal Dreiman
9	Предметы	hello_world, mavHarsha
10	Тестирование устройства	Buttink, Sir Celsius