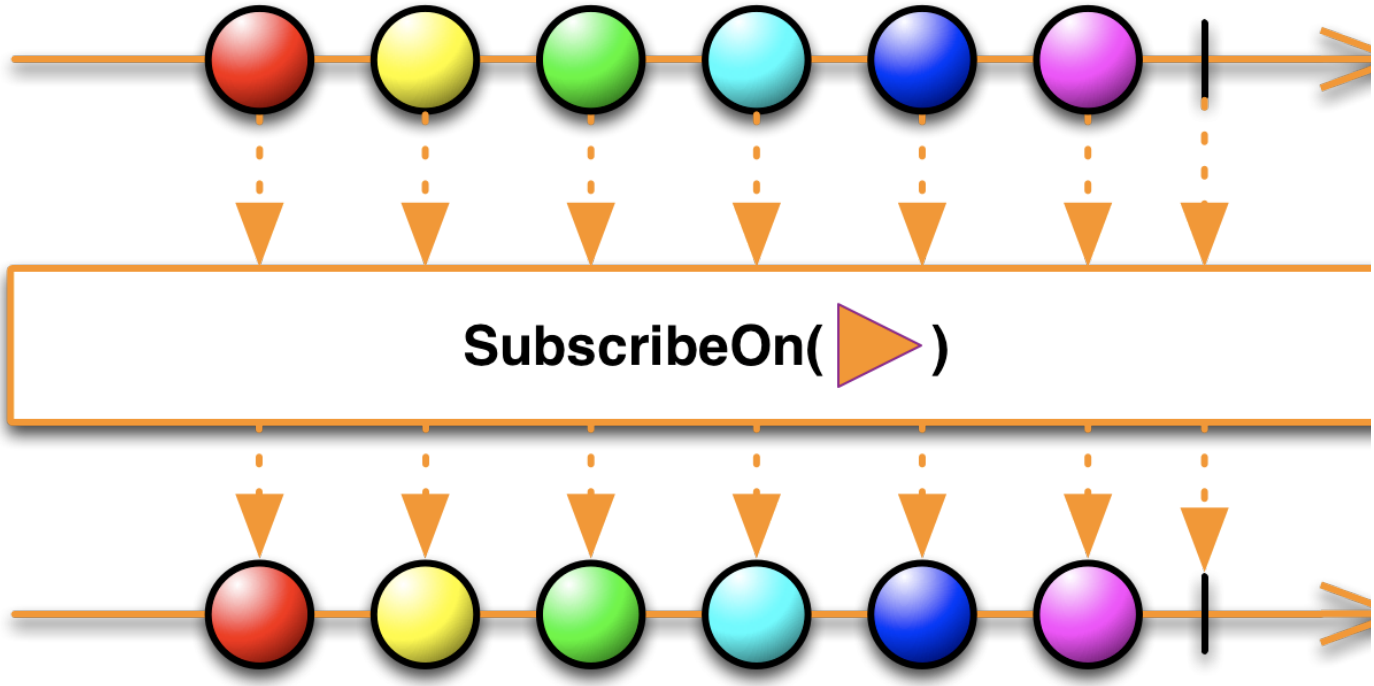
Rambler&Co 85,47
Компания

8 апреля 2016 в 13:58

Разбираемся с многопоточностью в RxJava

Разработка под Android*, Блог компании Rambler&Co



Когда описывают преимущества RxJava, всегда упоминают об удобстве организации работы многопоточного приложения средствами RxJava. как использовать операторы subscribeOn и observeOn, можно прочитать практически в каждой статье, посвященной основам RxJava. Наприме здесь хорошо описаны случаи, когда использовать методы subscribeOn и когда observeOn. Однако, на практике часто приходится сталкивать проблемами, для которых нужно более глубокое понимание того, что именно делают методы subscribeOn и observeOn. В этой статье я хотел рассмотреть ряд вопросов, которые иногда возникают при использовании этих операторов.

Изучать нюансы RxJava можно разными способами: по документации (которая весьма подробна), по исходникам или же на практике. Я выбр последний способ. Для этого я набросал пару тестов, по работе которых я смог лучше разобраться с асинхронным реактивным программиров

Сначала для проверки работы смены потоков я использовал следующий код:

Скрытый текст

```
private void testSchedulersTemplate(Observable.Transformer<String, String> transformer) {
    Observable<String> obs = Observable
            .create(subscriber -> {
                    logThread("Inside observable");
                    subscriber.onNext("Hello from observable");
                    subscriber.onCompleted();
            })
            .doOnNext(s -> logThread("Before transform"))
            .compose(transformer)
            .doOnNext(s -> logThread("After transform"));
    TestSubscriber<String> subscriber = new TestSubscriber<>(new Subscriber<String>() {
        @Override
```

```
        public void onCompleted() {
            logThread("In onComplete");
        }

        @Override
        public void onError(Throwable e) {}

        @Override
        public void onNext(String o) {
            logThread("In onNext");
        }
    });
    obs.subscribe(subscriber);
    subscriber.awaitTerminalEvent();
}
```

Проверим как работает этот код без всяких преобразований:

```
testSchedulersTemplate(stringObservable -> stringObservable);
```

Результат:
Inside observable: main
Before transform: main
After transform: main
Inside doOnNext: main
In onNext: main
In onComplete: main

Как и ожидалось, никакой смены потоков.

1. ObserveOn и SubscribeOn

SubscribeOn
Как можно понять из документации reactivex.io/documentation/operators/subscribeon.html
с помощью этого оператора можно указать Scheduler, в котором будет выполняться процесс Observable.
Проверяем:

```
testSchedulersTemplate(stringObservable -> stringObservable.subscribeOn(Schedulers.io()));
```

Результат:
Inside observable: RxCachedThreadScheduler-1
Before transform: RxCachedThreadScheduler-1
After transform: RxCachedThreadScheduler-1
Inside doOnNext: RxCachedThreadScheduler-1
In onNext: RxCachedThreadScheduler-1
In onComplete: RxCachedThreadScheduler-1

Начиная с выполнения содержимого Observable и до получения результата, все методы выполнялись в потоке, созданном Schedulers.io().

ObserveOn
В документации по этому методу сказано, что применение этого оператора приводит к тому, что последующие операции над “излученными” данными будут выполняться с помощью Scheduler, переданным в этот метод.

Проверяем:

```
testSchedulersTemplate(stringObservable -> stringObservable.observeOn(Schedulers.io()));
```

Результат:
Inside observable: main
Before transform: main
After transform: RxCachedThreadScheduler-1
Inside doOnNext: RxCachedThreadScheduler-1
In onNext: RxCachedThreadScheduler-1
In onComplete: RxCachedThreadScheduler-1

Как и ожидалось, с момента применения метода observeOn поток, в котором производится обработка данных, будет изменен на тот, который выделит указанный Scheduler.

Объединим использование subscribeOn и observeOn:

```
testSchedulersTemplate(stringObservable -> stringObservable
       .subscribeOn(Schedulers.computation())
       .observeOn(Schedulers.io()));
```

Результат:
Inside observable: RxComputationThreadPool-3
Before transform: RxComputationThreadPool-3
After transform: RxCachedThreadScheduler-1
Inside doOnNext: RxCachedThreadScheduler-1
In onNext: RxCachedThreadScheduler-1
In onComplete: RxCachedThreadScheduler-1

Методы, выполняемые до применения оператора observeOn выполнились в Scheduler, указанном в subscribeOn, а после – в scheduler, указан observeOn.

Комбинируя эти два метода, можно добиться асинхронной загрузки данных из интернета и отображения их на экране в главном потоке приложения.

Но что будет, если применить эти методы несколько раз?
Для начала вызовем observeOn несколько раз:

```
testSchedulersTemplate(stringObservable -> stringObservable
       .observeOn(Schedulers.computation())
       .doOnNext(str -> logThread("Between two observeOn"))
       .observeOn(Schedulers.io()));
```

Inside observable: main
Before transform: main
Between two observeOn: RxComputationThreadPool-3
After transform: RxCachedThreadScheduler-1
Inside doOnNext: RxCachedThreadScheduler-1
In onNext: RxCachedThreadScheduler-1
In onComplete: RxCachedThreadScheduler-1

Никаких сюрпризов. После применение observeOn обработка элементов производится с помощью указанного Scheduler.

Теперь вызовем subscribeOn несколько раз.

```
testSchedulersTemplate(stringObservable -> stringObservable
       .subscribeOn(Schedulers.computation())
       .doOnNext(str -> logThread("Between two observeOn"))
       .subscribeOn(Schedulers.io()));
```

Результат:
Inside observable: RxComputationThreadPool-1
Before transform: RxComputationThreadPool-1
Between two observeOn: RxComputationThreadPool-1
After transform: RxComputationThreadPool-1
Inside doOnNext: RxComputationThreadPool-1
In onNext: RxComputationThreadPool-1
In onComplete: RxComputationThreadPool-1

Как видим, применение второго subscribeOn не привело ни каким изменениям. Но совсем ли он бесполезен?

Добавим между вызовами subscribeOn оператор:

```
.lift((Observable.Operator<String, String>) subscriber -> {
   logThread("Inside lift");
   return subscriber;
})
```

Получим первое сообщение в логе:
Inside lift: RxCachedThreadScheduler-1

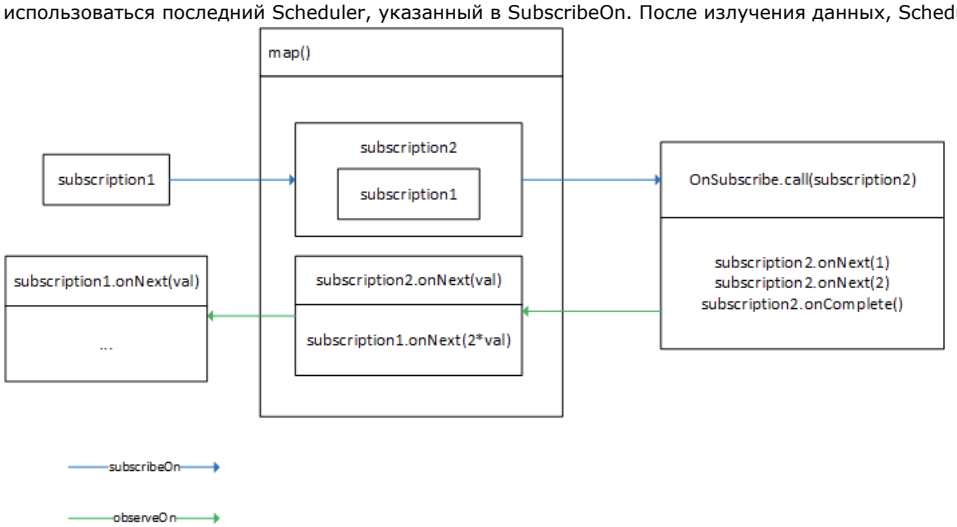
RxCachedThreadScheduler-1 — это именно тот поток, который был получен из Schedulers.io(), указанного во втором вызове subscribeOn.

lift() — это оператор, с помощью которого можно трансформировать subscription.
Можно схематично описать процесс выполнения подписки следующим образом:
Пользователь подписывается на observable, передавая subscription.

Этот subscription доставляется до корневого observable, при этом он может быть преобразован с помощью операторов.
Subscription передается в observable, отправляются onNext, onComplete, onError.
Над произведенными элементами выполняются преобразования
Преобразованные элементы попадают в onNext изначального subscriber.
Таким образом, когда subscription доставляется до observable, изменить поток можно с помощью subscribeOn. А когда элементы доставляются observable в subscription – влияет observeOn.
Для того, что бы это проиллюстрировать рассмотрим код:

```
Observable.create(subscriber -> {
...
})
.map(val-> val*2)
.subscribe(val -> Log.d(TAG, "onNext " + val));
```

Подписчик, созданный в последней строчке, передается в Observable, созданный с помощью Observable.create(). Внутри оператора map вызывается оператор lift, куда передается Operation, который во время подписки декорирует Subscriber. Когда Observable излучает данные, попадают в декорированный Subscriber. Декорированный Subscriber изменяет данные и отправляет их в оригинальный Subscriber.
Без изменения Scheduler весь процесс будет выполняться в потоке, в котором вызывается метод subscribe. Далее, пока Subscriber декорируе помощью subscribeOn можно изменить поток, в котором будет выполняться следующая декорация. В методе call() интерфейса OnSubscribe бу использоваться последний Scheduler, указанный в SubscribeOn. После излучения данных, Scheduler меняется уже с помощью onserveOn.



2. Выполняем задачи параллельно.

Рассмотрим следующий кейс:
Необходимо загрузить с сервера различную информацию, после этого скомпоновать ее и отобразить на экране. При этом, чтобы ускорить пр загружать данные стоит параллельно (если есть такая возможность). Если бы у нас не было RxJava, то эта задача требовала бы значительны усилий. Но с реактивным программированием эта задача тривиальна.

Мы будем выполнять три задачи, каждая из которых ждет 1 секунду, а потом отправляет сообщение в subscription. Далее с помощью операто combineLatest все сообщения будут объединены и переданы в подписку.

Для проверки будем использовать следующий код:
▸ Скрытый текст

Для начала запустим тест без всяких преобразований:

```
template(stringObservable -> stringObservable, null, null, null);
```

Результат:
Inside Observable1: main
Inside Observable2: main
Inside Observable3: main
Inside combining result: main
Before transform: main
After tranform: main
In onNext: main
In onComplete: main

Как видим, все выполняется в одном потоке. Наши три задачи выполняются последовательно.

Добавим subscribeOn и observeOn для observable, полученного с помощью функции zip.

```
template(stringObservable -> stringObservable.subscribeOn(Schedulers.io())
        .observeOn(Schedulers.newThread()), null, null, null);
```

Результат:
Inside Observable1: RxCachedThreadScheduler-1
Inside Observable2: RxCachedThreadScheduler-1
Inside Observable3: RxCachedThreadScheduler-1
Inside combining result: RxCachedThreadScheduler-1
Before transform: RxCachedThreadScheduler-1
After tranform: RxNewThreadScheduler-1
In onNext: RxNewThreadScheduler-1
In onComplete: RxNewThreadScheduler-1

Все так, как и описывалось в предыдущей части статьи про subscribeOn и observeOn.

Теперь каждую из задач будем выполнять в своем потоке. Для этого достаточно указать Scheduler.io(), т.к. внутри него содержится пулл пото оптимальный для загрузки данных.

```
Observable.Transformer<String, String> ioTransformer = stringObservable -> stringObservable.subscribeOn(Schedulers.io());
template(stringObservable -> stringObservable.subscribeOn(Schedulers.newThread())
       .observeOn(Schedulers.computation()), ioTransformer, ioTransformer, ioTransformer);
```

Результат:
Inside Observable1: RxCachedThreadScheduler-1
Inside Observable2: RxCachedThreadScheduler-2
Inside Observable3: RxCachedThreadScheduler-3
Inside combining result: RxCachedThreadScheduler-3
Before transform: RxCachedThreadScheduler-3
After tranform: RxComputationThreadPool-3
In onNext: RxComputationThreadPool-3
In onComplete: RxComputationThreadPool-3

Мы добились того, чего и хотели — три наши задачи выполнились параллельно.

3. Операторы с Schedulers.

В предыдущей главе для эмулирования долгих задач отлично подошел бы оператор delay(), но проблема в том, что этот оператор не так прос может показаться на первый взгляд.
Существует ряд операторов, которые требуют указания Scheduler для свой работы. При этом есть их перегруженные версии, которые в качес Scheduler используют computation(). delay() является примером такого оператора:

```
TestSubscriber<Integer> subscriber = new TestSubscriber<>();
Observable.just(1).delay(1, TimeUnit.SECONDS).subscribe(subscriber);
subscriber.awaitTerminalEvent();
Logger.d("LastSeenThread: " + subscriber.getLastSeenThread().getName());
```

Несмотря на то, что мы не указывали никакой Scheduler, результат будет следующим:
LastSeenThread: RxComputationThreadPool-1

Для того, что бы избежать использования computation scheduler, достаточно третьим параметром передать требуемый scheduler:
.delay(1, TimeUnit.SECONDS, Schedulers.immediate())

Примечание: Schedulers.immediate() — выполняет задачу в том же потоке, в котором выполнялась предыдущая задача.

Результат:
LastSeenThread: main

Кроме delay() существуют и другие операторы, которые могут сами менять Scheduler: interval(), timer(), некоторые перегрузки buffer(), debou skip(), take(), timeout() и некоторые другие.

4. Subjects.

При использовании Subjects стоит учесть то, что по умолчанию цепочка изменений данных, отправленных в onNext subject, будет выполнять том же потоке, в котором был вызван метод onNext(). До тех пор, пока не встретится в цепочке преобразований оператор observeOn.
А вот применить subscribeOn так просто не получится.

Рассмотрим следующий код:

```
        BehaviorSubject<Object> subject = BehaviorSubject.create();
                subject
                .doOnNext(obj -> Logger.logThread("doOnNext"))
                .subscribeOn(Schedulers.io())
                .observeOn(Schedulers.newThread())
                .subscribe(new Subscriber<Object>() {
```

```
                @Override
                public void onCompleted() {
                    Logger.logThread("onComplete");
                }

                @Override
                public void onError(Throwable e) {

                }

                @Override
                public void onNext(Object o) {
                    Logger.logThread("onNext");
                }
            });
    subject.onNext("str");
    Handler handler = new Handler();
    handler.postDelayed(() -> subject.onNext("str"), 1000);
    handler.postDelayed(() -> subject.onNext("str"), 2000);
```

Тут указаны и observeOn и subscribeOn, но результат будет следующим:
doOnNext: RxCachedThreadScheduler-1
onNext: RxNewThreadScheduler-1
doOnNext: main
onNext: RxNewThreadScheduler-1
doOnNext: main
onNext: RxNewThreadScheduler-1

Т.е. когда мы подписываемся на subject, он сразу возвращает значение и оно обрабатывается потоке из Shedulers.io(), а вот когда приходит следующее сообщение в subject, то используется поток, в котором был вызван onNext().

Поэтому, если вы после получения объекта из subject запускаете какую-то долгую операцию, то необходимо явно проставить observeOn меж ними.

5. Backpressure

В этой статье невозможно не упомянуть о таком понятии как backpressure. MissingBackpressureException — ошибка, которая довольно много н мне подпортила. Я не стану тут пересказывать то, что можно прочитать в официальной wiki RxJava: github.com/ReactiveX/RxJava/wiki/Backpr
Но если вы активно используете RxJava, то вам обязательно надо прочитать о backpressure.
Когда у вас в приложении имеется некоторый производитель данных в одном потоке и какой-то потребитель в другом, то стоит учитывать ситуацию, когда потребитель будет не успевать обрабатывать данные. В такой ситуации вам помогут операторы, описанные по приведенной ссылке.

Заключение.

RxJava позволяет очень удобно управлять выполнением задач в разных потоках. Но при использовании стоит хорошо представлять, что имен делают subscribeOn, observeOn, а так же как ведут себя различные операторы.
Следует внимательно изучать документацию к операторам, которые вы используете – там указывают, в каком именно Scheduler выполняется оператор. Так же стоит быть аккуратным с Subject. И не забывать о backpressure.
Так же стоит учитывать один из советов, который когда-то давал Ben Christensen (@benjchristensen) – один из основных авторов RxJava:

> “it makes the most sense for Subscribers to always assume that values are delivered asynchronously, even though on some occasions they may b delivered synchronously.”

> “Для подписчика имеет смысл считать, что данные доставляются асинхронно, даже в тех случаях, когда они могут доставляться синхронно

Ссылка на исходники из статьи: github.com/HotIceCream/GrokkingRxSchedulers

android, rxjava, android development, rxandroid, multithreading, concurrency

ПОХОЖИЕ ПУБЛИКАЦИИ

28 января 2016 в 16:41
Построение Android приложений шаг за шагом, часть первая
+22 103k 894 48

26 октября 2015 в 19:21
Конвейерное производство Android приложений
+12 16,4k 137 18

1 сентября 2015 в 13:02
Открытый митап Rambler.Android
+11 3,8k 16 9

Комментарии (2)

xGromMx 8 апреля 2016 в 16:36 *(комментарий был изменён)*

Welcome to my collection http://xgrommx.github.io/rx-book/content/resources/articles/index.html#rx

withoutuniverse 15 июля 2016 в 18:54

Интересная статья.
Жаль, что была проигнорирована сообществом.
Пожалуй, единственное пожелание — используйте в примерах свои собственные объекты ThreadPoolExecutor, так как названия вроде «RxNewThreadScheduler тяжело читаются. Было бы легче читать имена «ComputationThread», «IoThread» и т.д.

Только зарегистрированные пользователи могут оставлять комментарии. Войдите, пожалуйста.

САМОЕ ЧИТАЕМОЕ Разраб

Сейчас | Сутки | Неделя | Месяц

Получил 1.2K звезд на GitHub с ужасной архитектурой. Как?
+22 36,3k 89 36

Реализация псевдо-3D в гоночных играх
+90 13k 172 16

Индексы в PostgreSQL — 1
+102 11,5k 303 32

Выбор MQ для высоконагруженного проекта
+30 9,9k 131 46

Алгоритм Джонкера-Волгенанта + t-SNE = супер-сила
+63 9,5k 146 2

ИНТЕРЕСНЫЕ ПУБЛИКАЦИИ

Scilab в свободном падении
675 5 4

Библиотека компонентов как инструмент поддержания целостности вебсайта
1,6k 13 6

Победителем первого GameDev-хакатона DataArt стала команда из Киева

1,7k 9 2

Дамп ShadowBrokers: разбираемся в содержимом директории «swift»

2,8k 27 1

Как я дома NAS строил

6k 86 81