Часть третья, RxJava

GoncharovA Follow
Jan 1 · 7 min read

Продолжение ответника на замечательные вопросы. Третья часть — вопросы о фреймворке RxJava.

. . .

22. «hot» vs «cold» observable:

В RxJava есть «горячие» и «холодные» Observables. Горячий Observable порождает данные постоянно, даже если на него никто не подписан. Холодный Observable, соответственно, порождает данные только если у него есть хотя бы один подписчик.

23. Операторы: map, flatMap, concatMap, switchMap — для чего и в чем отличия:

Map к каждому излучаемому элементу применяет функцию и возвращает ее результат.

FlatMap так же применяет функцию к каждому излучаемому элементу, но эта функция функция возвращает тип Observable. Т.е. 1 излучаемый элемент может через flatMap породить множество излучаемых элементов или не одного.

SwitchMap Как flatMap, но он будет испускать только элементы из нового наблюдаемого, пока очередное новое событие не будет испущено из источника наблюдаемого.

ConcatMap также подписывается на внутренний Observable. Но в отличие от switchMap, который отписывается от текущего Observable, если появляется новый Observable, concatMap не будет подписываться на следующий Observable, пока не завершится текущий.

24. Операторы: from, just — для чего и в чем отличия:

Вы можете превратить любые входные данные в Observable при помощи create. Для распространенных типов данных, существуют уже готовые методы(from), призванные облегчить этот процесс.

just создает Observable, который выдаст определенное заранее множество объектов, после чего завершится.

25. Операторы: doOn...Next, ...Error и так далее — для чего:

doOnNext()позволяет нам добавить некоторое дополнительное действие, происходящее всякий раз, как мы получаем новый элемент данных.

doOnError не обрабатывает ошибку, в том смысле, что она ее не потребляет. Он просто что-то делает с нею, например, записывает её. (То же самое верно для doOnNext — он также не потребляет элемент, и элемент все еще заканчивается в onNext Subscriber).

Ошибка(Или другой объект для других методов doOn) по-прежнему отправляется по цепочке и все равно заканчивается на onError(OnNext, OnCompleted) вашего Subscriber.

26. Observable vs Flowable:

В Rx 2.0 Observable — класс без обработки проблемы backpressure, а новый Flowable был наделен поддержкой обработки backpressure из коробки.

27. Flowable: стратегии работы с backpressure:

BackpressureOverflowStrategy — это enum, который предлагает 3 статических поля с реализациями, представляющими типичные действия при:

ERROR — это дефолтное поведение при всех предыдущих перегрузках, выбрасывающее BufferOverflowException.

DROP_OLDES — по сути, это означает обновление буфера, при его переполнении. Все старые значения выбрасываются и буфер наполняется новым стеком.

DROP_LATEST — немного иное поведение, отличающееся тем, что, если произойдет переполнение, текущее значение будет просто проигнорировано, и только старые значения будут доставлены после запросов нисходящего потока.

28. Single, Completable, Maybe — случаи использования:

Single представляют Observable которые выдают одно значение или ошибку. Single может быть подходящим, когда у вас есть задача, ориентированная на наблюдаемое, и вы ожидаете одно значение, например сетевой запрос, который выполняется один раз, и возвращаемое значение (или ошибка), сетевой вызов работает в один раз, что означает, что вы не ожидаете, что он вернет дополнительные значения с течением времени. Другой пример-операция выборки данных из БД.

Completable представляют Observable, которые не выдает никакого значения, но только терминальные события, либо onError или onCompleted. Completable уместен, когда у вас есть Observable а значение возвращаемого результата не интересно. Примеры-это обновление кэша, например, операция может либо завершиться успешно/завершиться неудачно, но не возвращает никакого значения. Другим примером является некоторая длительная операция инициализации, которая ничего не возвращает. Это может быть сетевой вызов UPDATE/PUT, который привел только к индикации успеха.

С Maybe мы можем либо иметь некоторое значение точно так же, как Single или ничего не возвращать-так же, как Completable. Кроме того, как и у обоих, у нас есть ошибка. Maybe ценно, когда мы хотим отметить, что Observable может не иметь значения и будет просто завершена.

29. Операторы: defer, groupBy, debounce, combineLatest, withLatestFrom, merge, concat, zip, switchOnNext — как работают:

defer не создает новый Observable, но позволяет определить каким образом Observable будет создан при появлении подписчиков. Подумайте, как бы вы создали Observable который будет выдавать текущее время? Так как значение только одно, похоже, что здесь нам может помочь just.

groupBy разделяет Observable на множество других Observable, из которых каждая испускает подмножество элементов из исходной Observable.

debounce позволяет Observable испустить следующий элемент только если прошло определённое количество времени после предыдущего элемента.

combineLatest — когда элемент испускается одним из двух Observables, объединяет последний элемент, испускаемый каждым Observable с помощью указанной функции, и создавать элементы на основе результатов этой функции. CombineLatest испускает элемент всякий раз, когда какой-либо из исходных Observables испускает элемент (при условии, что каждый из исходных Observables испускает хотя бы один элемент). Когда какой-либо из исходных Observables испускает элемент, CombineLatest объединяет самые последние испущенные элементы из каждого из других исходных Observable, используя предоставленную вами функцию, и выдает возвращаемое значение из этой функции.

withLatestFrom — Как только Observable1 генерирует элемент проверяется сгенерирован ли хоть один элемент в Observable2, если да, то берутся последние элементы из Observable1 и Observable2 и используются в качестве аргументов для переданной функции, результат которой генерируется в качестве элемента.

merge — Объединяет несколько Observable в одну, испуская элементы всех объединённых Observable.concat — Объединяет несколько Observable в одну, испуская элементы последующей Observable после завершения предыдущей.

zip — Объединяет несколько Observable в одну, используя функцию, которая формирует испускаемый элемент на основании элементов, полученных по одному от каждой из объединённых Observable.

switchOnNext — Преобразует Observable, которая испускает другие Observable, в Observable, которая испускает элементы, испущенные самой последней из этих Observable.

30. Операторы: observeOn(), subscribeOn() — как работают:

Оператор subscribeOn() будет иметь тот же эффект независимо от того, где вы поместите его в цепочку observable; однако вы не можете использовать несколько операторов subscribeOn() в одной цепочке. Если вы включили в цепочку более одного subscribeOn(), ваша цепочка будет использовать только subscribeOn(), который ближе всего к observable источнику.

В отличие от subscribeOn(), имеет значение, куда в цепочку вы помещаете функцию observeOn(), так как этот оператор только изменяет поток, который используется observables, которые следуют ниже. Например, если вы вставляете в свою цепочку следующий код, то каждый observable, который появляется в цепочке с этого момента, будет использовать новый поток. Эта цепочка будет продолжать работать в новом потоке, пока не встретится другой оператор observOn(), после чего она переключится на поток, указанный этим оператором. Вы можете управлять потоком, куда конкретные observables отправляют свои уведомления, путём вставки в вашу цепочку нескольких операторов observeOn().

31. Типы Subjects:

PublishSubject Излучает все последующие элементы наблюдаемого источника в момент подписки. Здесь, если студент вошел в аудиторию, он просто хочет слушать с того момента, когда он вошел в аудиторию. Так, Publish будет лучшим выбором для использования.

ReplaySubject Излучает все элементы источника наблюдаемого(Observable), независимо от того, когда подписчик(subscriber) подписывается. Здесь, если студент с опозданием вошел в аудиторию, он хочет послушать лекцию с самого начала. Для этого мы должны использовать Replay.

BehaviorSubject излучает совсем недавно созданный элемент и все последующие элементы наблюдаемого источника, когда наблюдатель(observer) присоединяется к нему. Здесь, если студент вошел в аудиторию, он хочет слушать самые последние вещи(не с начала) преподаваемые профессором таким образом, что он получает идею контекста.

AsyncSubject выдает только последнее значение наблюдаемого источника(и только последнее). Здесь, если студент пришел в любой момент времени в

аудиторию, и он хочет слушать только о последней вещи(и только последней) которую учат.

32. Типы Processors:

ReplayProcessor — Воспроизводит события для подписчиков.

PublishProcessor — передает все последующие Observable элементы своим текущим подписчикам.

BehaviorProcessor — испускает самый последний Observable элемент и все последующие Observable элементы для каждого подписанного подписчика.

33. Виды Schedulers:

Schedulers.io() — Этот планировщик основывается на неограниченном пуле потоков и используется для интенсивной работы с вводом-выводом, например, доступ к файловой системе, выполнение сетевых вызовов, доступ к базе данных и так далее. Количество потоков в этом планировщике неограниченно и может расти по мере необходимости.

Schedulers.computation() — Этот планировщик используется для выполнения работы, высоко нагружающей ЦП, такой как обработка больших объемов данных, изображений и так далее. Планировщик основывается на ограниченном пуле потоков с размером в количество доступных процессоров. Так как этот планировщик подходит только для интенсивной работы с ЦП — количество его потоков ограничено. Сделано это для того, чтобы потоки не конкурировали за процессорное время и не простаивали.

Schedulers.newThread() — Этот планировщик создает совершенно новый поток при каждом вызове. В данном случае использование пула потоков не принесет никакой выгоды. Потоки очень затратно создавать и уничтожать. Вы должны быть осторожны и не злоупотреблять чрезмерным созданием потоков, так как это может привести в замедлению работы системы и переполнению памяти. Новый поток будет создаваться для обработки каждого элемента, полученного из observable-источника. В идеале вы должны использовать этот планировщик довольно редко, в основном для выведения в отдельный поток долго работающих частей программы.

Schedulers.single() — Этот планировщик основывается на единственном потоке, который используется для последовательного выполнения задач. Он может быть очень полезен, когда у вас есть набор фоновых заданий в разных местах вашего приложения, но нельзя допустить одновременного выполнения более чем одного из этих заданий.

Schedulers.from(Executor executor) — Этот планировщик будет основываться на вашем собственном Executor. Может возникнуть ситуация, в которой необходимо будет выполнять определенные задачи в планировщике на основании собственной логики распределения потоков. Допустим, вы хотите ограничить число параллельных сетевых вызовов, которые делает ваше приложение. Можно создать собственный планировщик, который будет работать на базе ограниченного в размерах пула потоков (Scheduler.from(Executors.newFixedThreadPool(n))) и использовать его во всех местах, связанных с сетевыми вызовами.

AndroidSchedulers.mainThread() — Это специальный планировщик, который недоступен в библиотеке RxJava. Необходимо использовать расширяющую библиотеку RxAndroid для доступа к этому планировщику. Этот планировщик полезен в Android приложениях для выполнения действий в потоке пользовательского интерфейса. По умолчанию этот планировщик ставит задания в очередь в Looper, связанный с основным потоком, но есть возможность переопределения: AndroidSchedulers.from(Looper looper).

34. Как работает связка операторов publish и connect:

Когда применен оператор publish(), то Observable трансформируется в Connectable Observable. Connectable Observable похож на обычный Observable за исключением одного момента. Он начинает производить элементы не тогда, когда на него подписываются, а только тогда, когда на нем вызван оператор connect(). share() — это обертка над publish().refcount().

refcount() вызывает connect() автоматически, когда подписывается первый Observer, так что нет нужды делать это самостоятельно. publish().refcount()/share() следит за том, как много Observer'ов подписано на Observable и не отключает первых от последнего до тех пор, пока существует хотя

бы один подписчик. Другими словами, когда счетчик подписчиков падает до нуля, Observable «умирает» и перестает производить какие-либо элементы.

. . .

Все ответы собраны из следующих источников:

https://riptutorial.com/ru/rx-java/

https://askdev.ru/

http://reactivex.io/documentation/

https://www.codeproject.com/

https://proglib.io/

Androi Development Rxjava

Medium About Help Legal