<TeachMeSkills/>

курс

Android разработчик

Reactive Streams – RxJava2

Агенда занятия:

Паттерн Observer

Отличия от корутин

Операторы трансформации

Операторы фильтрации

- **Паттерн Observer**
  - Отличия от корутин
  - Операторы трансформации
  - Операторы фильтрации

Паттерн Observer

Паттерн Observer (Наблюдатель) – это шаблон проектирования, который определяет отношение один ко многим между объектами: имеются **Observable** (наблюдаемые объекты, издатели событий) и **Observers** (наблюдатели, подписчики).
Когда Observable изменяет свое состояние или генерирует событие, он оповещает всех подписанных Observer-ов, вызывая у них определенные методы. Это позволяет отделить генерацию данных/событий от их потребления. В классическом виде паттерн Observer реализует модель подписки: наблюдатели подписываются на источник и получают уведомления автоматически, что особенно удобно для асинхронных событий.

Паттерн Observer

В контексте реактивного программирования паттерн Observer – фундамент модели **“Observable-Observer”**.

Библиотека RxJava фактически расширяет паттерн Observer, обеспечивая поддержку мощных возможностей реактивного программирования.

В ReactiveX (семейство библиотек, включая RxJava) взаимодействие строится так: Observer (подписчик) подписывается на Observable (наблюдаемый поток данных) и реагирует на все элементы или события, которые этот Observable испускает.

Паттерн Observer

Это реализуется через три метода, определенные у Observer:

- onNext (вызывается при поступлении нового элемента);
- onError (вызывается при ошибке);
- onComplete (вызывается при завершении последовательности без ошибок).

Таким образом, все события (данные, завершение, ошибка) поступают наблюдателю в виде вызовов этих методов.

RxJava

Библиотека для асинхронного программирования на Java, основанная на концепциях функционального программирования и реактивного программирования.
Основная цель RxJava — облегчить работу с асинхронными потоками данных и событиями.

В RxJava источник (source) — это объект, который испускает (emit) элементы в реактивный поток. В RxJava существует несколько типов источников, каждый со своими особенностями поведения, поддержки backpressure, горячести/холодности и областью применения.

RxJava

В RxJava существует несколько типов источников, каждый со своими особенностями поведения, поддержки backpressure, горячести/холодности и областью применения.

- **Cold observable** создаются по запросу и выдают данные при подписке на них.
- **Hot observable** всегда активны и выдают данные независимо от того подписаны на них или нет.

Backpressure в RxJava — это механизм управления скоростью потока данных, когда источник (producer) испускает элементы быстрее, чем подписчик (consumer) способен их обрабатывать.

Observable

Основной тип источника в RxJava. Он испускает последовательность 0 или более элементов и завершает работу либо вызовом onComplete, либо onError.

Backpressure: не поддерживает

Типичный use-case: UI-события, короткие или управляемые по частоте потоки данных

Горячий или холодный: по умолчанию холодный

```
val observable = Observable.just(1, 2, 3)
observable.subscribe { println(it) }
```

Flowable

Используется при необходимости обработки большого количества данных или частых событий, когда возникает проблема backpressure (переполнение потребителя).

Backpressure: поддерживает

Use-case: быстрые источники (например, сенсоры, запросы к БД), большие потоки

Горячий или холодный: холодный по умолчанию

```
val flowable = Flowable.range(1, 1_000_000)
flowable
    .onBackpressureBuffer()
    .subscribe { println(it) }
```

Single

Источник, который всегда испускает один элемент или ошибку.

Backpressure: не поддерживает

Use-case: запрос к API, который возвращает единственный результат (например, getUserById())

Методы подписки: subscribe(onSuccess, onError)

```
val single = Single.just("Hello")
single.subscribe { value -> println(value) }
```

Maybe

Источник, который может испустить один элемент, ничего или ошибку.

Backpressure: не поддерживает

Use-case: когда результат может быть, а может и не быть (например, поиск записи в кэше)

```
val maybe = Maybe.empty<String>() // может быть just, empty или error
maybe.subscribe(
    { println("Result: $it") },
    { println("Error: $it") },
    { println("Completed without value") }
)
```

Completable

Источник, который ничего не испускает, только сообщает об успешном завершении (onComplete) или ошибке (onError).

Backpressure: не поддерживает

Use-case: выполнение действия без возврата результата (например, saveUser() или deleteItem())

```
val completable = Completable.complete()
completable.subscribe(
    { println("Completed") },
    { println("Error: $it") }
)
```

Subject

Subject — это одновременно и Observable, и Observer. То есть он может принимать события и передавать их дальше подписчикам. Используется как мост между императивным и реактивным кодом.

Тип Subject	Описание
PublishSubject	Передаёт только новые элементы подписчикам (с момента подписки)
BehaviorSubject	Передаёт последний испущенный элемент новому подписчику + всё новое
ReplaySubject	Передаёт все предыдущие элементы каждому новому подписчику
AsyncSubject	Передаёт только последний элемент, и только после завершения

Subject

```
val subject = BehaviorSubject.createDefault("initial")
subject.subscribe { println("1: $it") }
subject.onNext("A")
subject.subscribe { println("2: $it") }
subject.onNext("B")
```

```
1: initial
1: A
2: A
1: B
2: B
```

Сравнение

Тип	Элементы	Завершение	Ошибки	Backpressure	Use-case
Observable	0..n	✅	✅	❌	UI, события
Flowable	0..n	✅	✅	✅	массивы, сенсоры
Single	1	✅	✅	❌	API-запрос
Maybe	0..1	✅	✅	❌	кэш, optional
Completable	none	✅	✅	❌	действия
Subject	n	зависит	✅	❌	маппинг событий

Паттерн Observer

▶ **Отличия от корутин**

Операторы трансформации

Операторы фильтрации

RxJava2 vs Kotlin Coroutines: отличия

Как RxJava, так и корутины Kotlin предназначены для упрощения написания асинхронного кода, но они делают это по-разному.
Рассмотрим ключевые отличия между использованием RxJava2 и Kotlin Coroutines (в частности, Flow API) с точки зрения реализации асинхронности, управления потоками, механизма отмены задач и обработки backpressure (противодавления).

Подход к асинхронности

RxJava предоставляет внешний реактивный API – вы создаете Observable/Flowable и используете цепочки операторов и подписчиков.
Асинхронность достигается за счет использования планировщиков (Schedulers) и неблокирующей обработки последовательностей.

Подход к асинхронности

Kotlin Coroutines – это языковая особенность, встроенная прямо в Kotlin.
Асинхронные операции оформляются как suspend функции, которые могут приостанавливать выполнение без блокировки потока. Для потоковых данных в корутинах используется тип Flow.
Код с корутинами выглядит более последовательным, тогда как код с RxJava носит декларативный характер (описание, что делать с потоком данных).
Оба подхода решают сходные задачи, но корутины – более низкоуровневый и гибкий механизм, в то время как RxJava предоставляет богатый набор готовых высокоуровневых операторов для работы с потоками.

Потоки (Threads) и планирование выполнения

В RxJava для управления потоками используются **Schedulers** – при помощи операторов вроде subscribeOn и observeOn вы указываете, на каком потоке выполнять работу. По умолчанию, без указания Scheduler, весь код Observable выполняется в том потоке, где произошел subscribe.

В Kotlin Coroutines используются **Coroutine Dispatchers**, которые выполняют схожую функцию. Вы запускаете корутину в определенном диспетчере, либо переключаетесь на него внутри корутины с помощью withContext. Главное отличие в том, что переключение потоков и планирование при работе с корутинами интегрировано в язык – корутины более легковесны, их можно запускать десятками тысяч (они не привязаны 1:1 к потокам ОС).

Отмена и управление жизненным циклом

В **RxJava** отмена потока осуществляется через **отписку (dispose)** подписчика. Когда вы подписываетесь (subscribe) на Observable, возвращается объект Disposable, и вызвав dispose() на нём, вы прекращаете получение новых событий (Observable при этом может продолжать выполняться, но результаты вам уже не придут, либо, в случае hot-источников, он может продолжать генерировать данные для других подписчиков).

Отмена и управление жизненным циклом

В Kotlin корутинах используется механизм кооперативной отмены: каждая корутина выполняется в контексте Job. Отмена происходит путем вызова job.cancel(), что отменяет корутину и все ее дочерние задачи. При работе с потоками Flow отмена также простая – если корутина, собирающая Flow, была отменена, то поток прерывается автоматически.

Отмена и управление жизненным циклом

Корутины обеспечивают более прозрачное управление жизненным циклом асинхронных операций: можно привязать их к scope (например, жизненному циклу UI-компонента). В RxJava необходимо вручную отслеживать Disposable (например, добавлять их в CompositeDisposable и отписываться в нужный момент). Это требует дисциплины, иначе возможны утечки (несвоевременная отписка приведет к продолжающемуся фоновому исполнению). С другой стороны, RxJava предоставляет операторы вроде takeUntil/takeWhile для прекращения последовательности по условию, а в coroutines Flow есть аналогичные механизмы отмены по времени (withTimeout) или сигналу.

Backpressure

Backpressure – проблема, возникающая, когда производитель данных генерирует события быстрее, чем потребитель успевает их обрабатывать.

RxJava2 решает эту проблему введением двух видов потоков: **Observable** (не поддерживает backpressure) и **Flowable** (поддерживает backpressure). Если ожидается высокая частота событий, следует использовать Flowable и указывать стратегию противодавления (например, onBackpressureBuffer, onBackpressureDrop и др.), либо использовать встроенные стратегии Flowable по умолчанию.

Backpressure

В Kotlin корутинах тип Flow спроектирован таким образом, чтобы противодавление обрабатывалось автоматически. Как отмечается в документации, Flow представляет асинхронный поток данных, который выдает значения последовательно и завершится нормально или с исключением

Backpressure

Важное свойство Flow – поддержка приостановки: все основные операторы Flow помечены как suspend. Благодаря этому, если поток данных эмитируется и собирается в одной корутине, а потребитель не успевает за производителем, то испускание новых элементов приостановится, пока потребитель не обработает текущие javaadvent.com.

Проще говоря, Flow приостанавливает источник при медленном потребителе, избегая необходимости вручную применять специальные стратегии противодавления. Именно поэтому в coroutines нет разделения на “backpressure-aware” и “не backpressure” варианты – достаточно одного Flow.

Backpressure

В RxJava же без правильного использования Flowable или операторов противодавления быстрый источник и медленный подписчик могут привести к переполнению буфера или потере данных.
Таким образом, корутины предоставляют более естественное решение для противодавления, в то время как в RxJava-разработчик должен явно об этом позаботиться (но взамен получает тонкий контроль – напр. выбор стратегии поведения при переполнении буфера).

Итого

RxJava2 и Kotlin Coroutines (Flows) позволяют реализовывать реактивное поведение, но через разные парадигмы.

RxJava – внешний DSL для описания потоков событий, с множеством готовых операторов, что может ускорять разработку сложной реактивной логики.

Корутины – это встроенный механизм языка Kotlin для асинхронности, который ощущается как пишущийся “обычный” код с поддержкой приостановки.

Итого

При миграции с RxJava на корутины (или при выборе подхода) важно учитывать: требуемые операторы (RxJava богаче экосистемой операторов), модель обработки ошибок (в RxJava ошибки протекают через onError, в корутинах – через исключения), интеграцию с существующим кодом, а также командный опыт.

Паттерн Observer

Отличия от корутин

▶ **Операторы трансформации**

Операторы фильтрации

Операторы трансформации в RxJava2

Одним из сильных аспектов RxJava является богатый набор операторов трансформации, позволяющих преобразовывать данные внутри реактивного потока.

Рассмотрим подробно три важных оператора: **map, flatMap** и **switchMap**, а также примеры их использования на Kotlin.

Оператор map

Оператор map предназначен для преобразования каждого элемента исходного потока. Он применяет заданную функцию к каждому элементу и возвращает новый поток, содержащий результаты этих преобразований. Проще говоря, map проецирует элементы из одного типа в другой, либо изменяет значения.

Если исходный Observable выпустил последовательность элементов x1, x2, ... xn, то после применения map(func) мы получим последовательность func(x1), func(x2), ... func(xn).

```
val numbers = Observable.fromIterable(listOf(1, 2, 3, 4, 5))
numbers
    .map { it * it }  // возводим каждое число в квадрат
    .subscribe(
        { value -> println("Получено: $value") },
        { error -> println("Ошибка: ${error.message}") },
        { println("Готово") }
    )
```

Оператор flatMap

Оператор flatMap – один из самых мощных в реактивном арсенале. Его задача – преобразовать каждый элемент исходного потока в новый поток (Observable), а затем объединить (flatten) полученные внутренние потоки в один результирующий поток.

Таким образом, flatMap может породить расширенную последовательность событий: из каждого исходного элемента – целую серию новых элементов.

Оператор flatMap

Основная схема: исходный Observable эмитирует элемент x → применяется функция, возвращающая Observable O_x → flatMap подписывается на O_x и все элементы, которые тот излучит, передаются дальше подписчику. И так для каждого исходного элемента, параллельно.

Важное свойство: flatMap может interleave (перемешивать) результаты из разных внутренних Observable, т.е. результаты могут приходить в смешанном порядке, если внутренние потоки завершаются асинхронно. (Для ситуаций, где порядок важен, существует оператор concatMap, выполняющий последовательное конкатенирование, но его мы здесь не рассматриваем).

Оператор flatMap

Пример использования flatMap: предположим, есть поток букв, и для каждой буквы мы хотим получить поток, состоящий из этой буквы, объединенной с цифрами. Например, для "A" получить ["A1","A2"], для "B" – ["B1","B2"], и объединить все результаты в один поток.

```
val letters = Observable.just("A", "B", "C")
letters
    .flatMap { letter ->
        // Преобразуем каждую букву в новый Observable из двух элементов
        Observable.fromArray(letter + "1", letter + "2")
    }
    .subscribe { value -> println(value) }
```

Оператор flatMap

flatMap особенно полезен для выполнения асинхронных операций для каждого элемента.

Например, у нас есть список URL, и мы хотим по каждому сделать сетевой запрос и получить Observable с результатом – вот тут flatMap пригодится, чтобы запустить все запросы параллельно и собрать их результаты в единый поток.

Или другой пример: есть база данных пользователей (IDs), и для каждого ID нужно получить Observable с данными профиля – flatMap позволит запустить загрузку всех профилей одновременно и обработать поток возвращенных данных по мере их поступления.

Оператор switchMap

Оператор switchMap похож на flatMap, он тоже проецирует каждый элемент исходного потока в новый Observable-поток, но при поступлении нового элемента отменяет предыдущий внутренний поток и переключается на текущий. Значит, в каждый момент времени switchMap активно испускает элементы только из последнего полученного внутреннего Observable, игнорируя результаты предыдущих, если те вдруг не успели завершиться.

Оператор switchMap

Иными словами: “последний выиграл”.

Приходит элемент x1 – запускается внутренняя последовательность O_x1.

Пока она не завершилась, приходит новый элемент x2 – switchMap отписывается от O_x1 (прерывая его обработку) и переключается на новый O_x2.

Этот механизм очень полезен в ситуациях, когда нас интересует только актуальный результат, а устаревшие не нужны.

Оператор switchMap

Пример: у пользователя есть поле ввода для поиска, и он печатает запросы. Мы хотим отправлять запрос к серверу при изменении ввода, но если пользователь ввел новую букву, предыдущий запрос уже не нужен – его результат не следует обрабатывать. Именно здесь применяется switchMap: каждый новый поисковый запрос отменяет предыдущий.

```
val queryInput: Observable<String> = getTextChangeObservable()  // условно, Observable строк из поля ввода
queryInput
    .distinctUntilChanged()  // игнорировать повторяющиеся подряд идентичные запросы
    .switchMap { query ->
        // эмулируем сетевой поиск: возвращаем Observable, который через задержку выдаёт результат
        fakeSearchApi(query)  // допустим, возвращает Observable<List<Result>>
            .onErrorReturnItem(emptyList())  // в случае ошибки возвращаем пустой список (пример обработки ошибки)
    }
    .observeOn(AndroidSchedulers.mainThread())  // результаты обработаем в главном потоке (для UI)
    .subscribe { results -> showResultsOnUI(results) }
```

Как это всё работает

Смотрим проект

Паттерн Observer

Отличия от корутин

Операторы трансформации

▶ **Операторы фильтрации**

Операторы фильтрации в RxJava2

Помимо трансформаций, часто требуется фильтровать поток данных – т.е. отбирать или пропускать только определенные элементы, либо ограничивать последовательность по размеру. RxJava2 предоставляет множество операторов фильтрации.
Рассмотрим основные: **filter, take, skip, distinctUntilChanged**, их предназначение и примеры использования.

Оператор filter

Оператор filter пропускает только те элементы, которые удовлетворяют заданному условию (предикату). Он фильтрует исходный поток, позволяя пройти дальше лишь элементам, для которых предикат возвращает true. Все остальные элементы отбрасываются.

Простыми словами, **filter** – это аналог условия if для потоков: “передай дальше, если…”.

Пример использования filter: отфильтровать из потока чисел только чётные.

```
Observable.just(1, 2, 3, 4, 5, 6)
    .filter { it % 2 == 0 }  // пропускаем только числа, делящиеся на 2
    .subscribe { println(it) }
```

Оператор take

Оператор take позволяет ограничить количество элементов, которые будут выпущены Observable. Он берет только первые N элементов, а остальные игнорирует и завершается (вызывает onComplete после получения N элементов). Также есть вариант **take(Duration)** – брать элементы в течение определенного времени.
Проще говоря, **take(n)** – “возьми не больше n элементов и завершись”.

```
Observable.range(1, 10)    // эмитирует числа от 1 до 10
    .take(3)
    .subscribe(
        { println("Элемент: $it") },
        { println("Ошибка: $it") },
        { println("Последовательность завершена") }
    )
```

Оператор skip

Оператор skip – это “пропусти N элементов”. Он игнорирует первые N элементов исходного потока, а все последующие пропускает дальше без изменений. Если поток содержит меньше N элементов, то на выход вообще ничего не пройдет (будет только завершение).
Пример **skip**: пропустим первые 4 элемента из последовательности.

```
Observable.just("a", "b", "c", "d", "e", "f")
    .skip(4)
    .subscribe { println(it) }
```

Оператор distinctUntilChanged

Оператор distinctUntilChanged фильтрует дубликаты, отбрасывая идущие подряд повторяющиеся элементы. Он следит за предыдущим эмиттированным значением, и если новое значение идентично предыдущему, то не пропускает его. Как только значение изменилось – оно проходит дальше, и затем новый повторяющийся подряд снова блокируется.

Оператор distinctUntilChanged

Важно, что distinctUntilChanged сравнивает только соседние элементы. Если последовательность была A, A, B, A, то на выходе получится A, B, A – в конце A снова прошел, потому что перед ним был B (элемент изменился). Если нужен полностью уникальный набор элементов за все время – используется оператор distinct, который помнит все увиденные значения.

Практика

Задача 1. Фильтрация и преобразование.

Создайте Observable, генерирующий последовательность чисел от 1 до 20. С помощью операторов filter и map получите поток, в котором:

- остаются только четные числа,
- каждое оставшееся число возводится в куб (например, 2 -> 8, 4 -> 64, ...).

Подпишитесь на полученный поток и выведите результаты. Убедитесь, что в вывод попадают только кубы четных чисел.

Подсказка: используйте Observable.range для генерации исходной последовательности.

Задача 2. Использование flatMap.

У вас есть список имен студентов и функция getScores(name: String): Observable<Int> которая возвращает Observable с оценками данного студента (несколько чисел).

Реализуйте Observable из списка имен и примените flatMap, чтобы для каждого имени подтянуть оценки и вывести их в общем потоке. То есть на выходе должен получиться поток типа "Имя - Оценка".
Можно сначала сгенерировать фейковые данные: вместо реальной функции getScores сделайте Observable.fromIterable(listOf(…)) внутри flatMap или используйте Observable.just с парой-тройкой чисел.

Важно понять сам шаблон: как flatMap помогает по каждому элементу внешнего потока запускать свой внутренний поток данных. Добавьте логирование (например, println) чтобы видеть, какие данные к вам приходят и в каком порядке.

Задача 3. Ограничение и пропуск элементов.

При помощи операторов skip и take выберите определенный *диапазон* элементов из последовательности.

К примеру, есть поток целых чис от 1 до 100. Требуется пропустить первые 50 чисел, из оставшихся взять первые 5. Ожидаемый результат – числа 51–55.

Реализуйте эту логику с помощью RxJava. Проверьте, что количество полученных элементов соответствует ожидаемому, и что последовательность завершается корректно после выбранных элементов.

Задача 4. distinctUntilChanged на практике.

Сгенерируйте Observable, который эмитит следующую последовательность символов (тип Char или строки): "A", "A", "B", "B", "B", "C", "A", "A", "D", "D".

Примените distinctUntilChanged, чтобы в выходном потоке не было подряд идущих повторов. Выведите результаты. Убедитесь, что одинаковые подряд символы сокращаются до одного.

Затем модифицируйте пример: попробуйте использовать вместо простых строк объект, содержащий поле, и передайте в distinctUntilChanged функцию получения этого поля – убедитесь, что оператор может работать с пользовательским типом (например, фильтровать повторяющиеся ID у последовательности объектов с полем id).

Домашнее задание

Задача 1.

Реализовать упрощенный механизм *“поиска с автодополнением”* (search-as-you-type) с использованием RxJava2.

Представьте, что пользователь вводит текст запроса, и после каждой паузы в наборе нужно отправлять запрос на сервер для получения результатов.

Ваша задача – сэмулировать эту ситуацию и обработать её реактивно.

Задача 1.

Требования к решению:
Используйте Observable<String> для представления последовательности вводимых пользователем поисковых запросов (например, можно программно эмулировать серии строк, идущих одна за другой с небольшими задержками).

Для предотвращения слишком частых запросов примените оператор debounce (или throttleWithTimeout) – он позволит отправлять запрос только если пользователь перестал печатать на заданное время (например, 300 мс).

Примечание: оператор debounce пропускает значение, если после него в течении N мс не пришло новое – то, что нужно для “паузы”.
Используйте distinctUntilChanged, чтобы не отправлять повторный запрос, если новый введенный текст идентичен предыдущему (например, пользователь может несколько раз нажать пробел или вернуть прежнее значение).

Задача 1.

Для самой отправки запроса воспользуйтесь оператором switchMap.

Он должен выполнить следующее: взять текущую строку запроса и вернуть Observable с “результатами поиска” (эти результаты можно сгенерировать случайно или взять из заранее подготовленного списка в соответствии с запросом).

Важно: благодаря switchMap предыдущий незавершенный запрос будет отменён, если пришел новый ввод – это и требуется (не должны накапливаться ответы от устаревших запросов).

Задача 1.

Добавьте обработку ошибок: например, если “сервер” вернул ошибку (сэмулируйте Observable.error в каком-то случае), поток не должен завершиться навсегда. Можно воспользоваться оператором onErrorResumeNext или onErrorReturn, чтобы перехватить ошибку и выдать какое-то значение (например, пустой список результатов) или переключиться на другой поток.

Результаты поиска (например, список строк) передавайте на observeOn(Schedulers.io()) (если нужно показать понимание переключения потоков; в консоль можно просто вывести, но представим, что обновляем UI – тогда нужен главный поток, в Android это AndroidSchedulers.mainThread()).

Q&A

Ваши вопросы

Спасибо

<TeachMeSkills/>