GeekBrains

Android. Популярные библиотеки

# Реактивное программирование. RxJava

Часть 1

# На этом уроке

1. Узнаем, как работать с асинхронными задачами и не только.

# Оглавление



# Вступление

Формально реактивное программирование определяется как парадигма программирования, ориентированная на потоки данных и распространение изменений.

Поток в нашем случае — не понятие thread, к которому мы привыкли, используя термин «многопоточность». Это обработка любых последовательностей данных, будь то набор файлов, получаемых из облака, или цепочка вводимых в EditText символов. Подобная последовательность и есть поток.

С точки зрения изменений, поток представляет собой последовательность значений одной и той же величины. Например, поток, состоящий из строк текста всё того же EditText целиком после каждого изменения.

RxJava — реализация ReactiveX для Java. ReactiveX — библиотека, посредством которой легче работать с этими потоками информации. Посредством Rx удобно обрабатывать любые последовательности данных, например:

- набор файлов, получаемых из облака;
- цепочка вводимых в EditText символов.

RxJava основывается на паттерне «Наблюдатель» (Observer), где, как известно, есть объект, который формирует и отдаёт данные, и также подписчики, получающие эти данные. Rx организует передачу данных, но не всех сразу, а по очереди.

Одна из главных идей, лежащих в основе Rx — неизвестно когда именно последовательность выдаст новое значение или завершится. Используя эту библиотеку, мы можем это отследить.

В Rx также легко управлять уже известными нам потоками thread, где выполняется обработка цепочек данных. Реализация Rx есть для большинства языков.

Официальный сайт ReactiveX: ReactiveX.io.

Репозиторий RxJava с документацией и примерами.

# Реактивное программирование с RXJava

Подключение:

```
implementation 'io.reactivex.rxjava3:rxandroid:3.0.0'
implementation 'io.reactivex.rxjava3:rxjava:3.0.0'
```

# Асинхронное программирование

Реактивное программирование обеспечивает доступ к асинхронному программированию. Оно используется, чтобы упростить асинхронную обработку длительных операций. Именно это программирование — способ обработки нескольких событий, ошибок и завершения потока событий.

Такой тип программирования обеспечивает также упрощённый способ запуска различных задач в разных потоках.

# Источники данных и слушатели

В RxJava источники представлены двумя основными типами — **Flowable** и **Observable**. Flowable рассмотрим позже, сосредоточимся на Observable. Observable выдаёт 0 или n элементов и завершается успехом или событием ошибки.

Так выглядит создание источника вручную, с применением метода **create**:

```
Observable.create<String> { emitter ->
    try {
        val list: List<String> = someExampleStringHolder.getStrings()
        list.forEach {
            emitter.onNext(it)
        }
    } catch (e: Throwable){
        emitter.onError(e)
    }
    emitter.onComplete()
}
```

В примере someExampleStringHolder — экземпляр класса с методом getStrings, возвращающим список строк. Мы обернули список в поток данных. Здесь это просто визуальный пример, чтобы дать представление о том, как это может выглядеть. Подробности работы оператора create и прочие детали этого примера рассмотрим позже.

В RXJava есть два основных понятия: **Observable** и **Observer**. Observable — это класс, Observer — интерфейс. Реализация интерфейса Observer — класс, объект которого лишь «подписывается» на Observable. А Observable, в свою очередь, «генерирует» некоторый поток данных и уведомляет об этом либо о каком-либо другом событии подписчика Observer.

События, генерируемые Observable, мы рассматриваем как некий поток данных. В этом потоке есть три типа событий:

1. Next — передана очередная порция данных.
2. Error — произошла ошибка.
3. Completed — поток завершился, и данных больше не будет.

Во время изучения RxJava в силу его специфики мы не станем делить занятия на теорию и практику, а рассмотрим операторы по мере их написания. Для удобства работы немного структурируем код. Так

как начнём с формирования Observable, создадим класс Creation и напишем туда следующую заготовку:

```
class Creation {

    fun exec() {
        Consumer(Producer()).exec()
    }

    class Producer {

    }


    class Consumer(val producer: Producer) {
        fun exec() {

        }
    }

}
```

Теперь в Producer мы станем создавать Observable разными способами, а в Consumer — на них подписываться. При запуске функции exec внешнего класса Creation, например, из onCreate нашего главного активити всё это будет запускаться и выводить результат. Рассмотрим простейший способ создания Observable — оператор **just.** Создадим следующую функцию в Producer:

```
class Producer {
    fun just(): Observable<String> {
        return Observable.just("1", "2", "3")
    }
}
```

Это даст источник, готовый передать цепочку строковых данных. Теперь, чтобы на него подписаться, понадобится получатель — Observer. Создадим его в Consumer:

```
class Consumer(val producer: Producer) {
    val stringObserver = object : Observer<String> {
        var disposable: Disposable? = null

        override fun onComplete() {
            println("onComplete")
        }

        override fun onSubscribe(d: Disposable?) {
            disposable = d
            println("onSubscribe")
```

```
        }

        override fun onNext(s: String?) {
            println("onNext: $s")
        }

        override fun onError(e: Throwable?) {
            println("onError: ${e?.message}")
        }
    }


    fun exec() {

    }
}
```

Чтобы принимать строки, создадим Observer<String>. Теперь добавим в Consumer следующую функцию:

```
fun execJust() {
    producer.just()
        .subscribe(stringObserver)
}
```

Посредством Observer подписываемся на Observable, возвращаемый функцией just() нашего Producer. В exec осталось вызвать execJust в Producer. В результате всё приобретает следующий вид:

```
class Creation {

    fun exec() {
        Consumer(Producer()).exec()
    }

    class Producer {
        fun just(): Observable<String> {
            return Observable.just("1", "2", "3")
        }
    }


    class Consumer(val producer: Producer) {
        val stringObserver = object : Observer<String> {
            var disposable: Disposable? = null

            override fun onComplete() {
                println("onComplete")
            }
```

```
            override fun onSubscribe(d: Disposable?) {
                disposable = d
                println("onSubscribe")
            }

            override fun onNext(s: String?) {
                println("onNext: $s")
            }

            override fun onError(e: Throwable?) {
                println("onError: ${e?.message}")
            }
        }


        fun exec() {
            execJust()
        }

        fun execJust() {
            producer.just()
                .subscribe(stringObserver)
        }
    }

}
```

Теперь, когда мы запустим это посредством функции exec(), Observer подпишется на Observable. В этот момент сработает событие onSubscribe(), после чего начнётся передача данных. Во время передачи данных из последовательности сработает событие onNext() Observer. Данные из цепочки станут передаваться как параметр **s** функции onNext() по очереди. В конце сработает событие onCompleted(). При передаче данных может сработать один из двух терминаторов: onCompleted() или onError(). После их активации передача данных прекращается. В итоге программа выведет следующий результат:

```
onSubscribe
onNext: a
onNext: b
onNext: c
onNext: d
onComplete
```

Осталось понять, что за Disposable-аргумент в onSubscribe.

**Disposable** — интерфейс, реализация которого напрямую связана с потоком данных (источником). Этот интерфейс позволяет отписаться от источника посредством вызова у него метода dispose().

Например, пользователь закрыл экран, на котором отображались данные из потока. Данные больше не требуются, надо отписаться от источника и прекратить обработку данных из потока. Чтобы добиться этого, предварительно сохраняем Disposable, полученный в onSubscribe, а затем просто вызываем у него dispose(), например, где-нибудь на этапе onPause.

Подписаться на Observable можно не только посредством экземпляра Observer, но и набора лямбд, каждая из которых отвечает за своё событие: onNext, onError и onComplete. onSubscribe не требуется, так как Disposable вернётся из функции subscribe. В Consumer:

```
fun execLambda() {
    val disposable = producer.just()
        .subscribe({ s ->
            println("onNext: $s")
        }, { e ->
            println("onError: ${e.message}")
        }, {
            println("onComplete")
        })
}
```

Как видно из примера, мы просто используем соответствующую версию функции subscribe, которая принимает на вход три лямбды и возвращает Disposable.

# Операторы

Есть достаточное количество операторов создания Observable. Рассмотрим несколько ключевых.

## just

Его мы рассмотрели выше. В оператор just передаётся набор элементов:

```
fun just(): Observable<String> {
    return Observable.just("1", "2", "3")
}
```

## fromIterable

Оператор fromIterable похож на just, но в него передаётся не набор, а коллекция элементов. В Producer:

```
fun fromIterable(): Observable<String> {
    return Observable.fromIterable(listOf("1", "2", "3"))
}
```

В Consumer:

```
fun execFromIterable() {
    producer.fromIterable()
        .subscribe(stringObserver)
}
```

## Interval

```
val observable = Observable.interval(5, TimeUnit.SECONDS)
```

В Producer:

```
fun interval() = Observable.interval(1, TimeUnit.SECONDS)
```

В Consumer:

```
fun execInterval() {
    producer.interval()
        .subscribe {
            println("onNext: $it")
        }
}
```

В этом примере Observable будет бесконечно выдавать Long-числа начиная с 0, с интервалом 5 секунд. Подписываемся, используя лямбду, чтобы не создавать отдельный Observer.

Мы уже чётко наметили схему, по которой создаём источники в Producer и подписываемся в Consumer. Ниже приводятся только конструкции из Producer, так как для Consumer всё аналогично, если использовать подписку посредством лямбд, как в примере выше.

## Timer

```
fun timer() = Observable.interval(1, TimeUnit.SECONDS)
```

Аналогичен interval, но он выдаст одно значение только один раз.

## Range

```
fun range() = Observable.range(1, 10)
```

Последовательно выдаст числа от первого аргумента до второго. В нашем случае — от 1 до 10.

## fromCallable

Чаще всего нам будет недостаточно операторов создания, принимающих заведомо известные значения. Потребуется создавать источник из какой-либо процедуры, который выдаст нам результат её выполнения в качестве значения в onNext. Для этого используется оператор fromCallable. Сначала в Producer создадим функцию, которая имитирует длительные вычисления с некоторым случайным результатом:

```
//Симуляция длительной операции со случайным исходом
w {
    Thread.sleep(Random.nextLong(1000))
    return listOf(true, false, true)[Random.nextInt(2)]
}
```

Функция будет «спать» случайное время до секунды. После этого в одном случае из трёх она выдаст нам false, а в остальных — true.

Теперь используем её в fromCallable. Добавим следующее в Producer:

```
fun fromCallable() = Observable.fromCallable {
    //Симуляция долгих вычислений
    val result = randomResultOperation()
    return@fromCallable result
}
```

Таким образом, fromCallable принимает на вход процедуру и возвращает источник, который выдаст возвращаемое значение этой процедуры.

## create

Самый мощный оператор создания источника. Оператор create предполагает полностью ручное управление событиями, которые выдаёт источник, и мы в буквальном смысле вызываем onNext, onComplete и onError у подписчиков. В начале урока он уже встречался, но детально не рассматривался. Теперь разберём этот оператор более подробно:

```
fun create() = Observable.create<String> { emitter ->
    try {
        for (i in 0..10) {
            randomResultOperation().let {
                if (it) {
                    emitter.onNext("Success$i")
```

```
                } else {
                    emitter.onError(RuntimeException("Error"))
                    return@create
                }
            }
        }
        emitter.onComplete()
    } catch (t: Throwable) {
        emitter.onError(RuntimeException("Error"))
    }
}
```

Аргумент метода **create** — функциональный интерфейс **ObservableOnSubsribe** с функцией subscribe, имеющей единственный аргумент с типом **ObservableEmitter**. Это позволяет заменить его лямбдой. Вызывая соответствующие методы у ObservableEmitter, заставляем источник выдавать события. Так мы полностью контролируем источник и сами решаем, когда и какой метод у Observer вызовется.

В примере выводим строку, когда имитация randomResultOperation() выдаёт true, и ошибку — в случае false. Правильнее обернуть весь код в try-catch, чтобы передать возможное исключение в onError подписчиков для корректной обработки.

Теперь, когда мы рассмотрели основные операторы создания источников, перейдём к операторам манипуляции над потоком. Создадим класс Operators, аналогичный Creation:

```
class Operators {

    fun exec() {
        Consumer(Producer()).exec()
    }

    class Producer {
        fun createJust() = Observable.just("1", "2", "3", "3")
    }


    class Consumer(val producer: Producer) {

        fun exec() {

        }

    }
}
```

Just выдаёт последовательность строк, над которой совершим преобразования. Тройка выдаётся два раза подряд, далее станет ясно, для чего это.

Теперь в Consumer мы будем подписываться на источники, выдаваемые Producer, и применять к ним операторы, манипулирующие потоком.

## take

Оператор take(count) берёт первые count элементов цепочки. В Consumer:

```
fun execTake() {
    producer.createJust()
        .take(2)
        .subscribe({ s ->
            println("onNext: $s")
        }, {
            println("onError: ${it.message}")
        })
}
```

Код выведет результат для первых двух значений. Во время исполнения один Observable будет завёрнут в другой. Вначале мы получаем первый Observable от producer. Затем посредством оператора take создаётся второй. Второй Observable оборачивает первого, создавая надстройку. Первый производит последовательность чисел, второй берёт только первые два и отдаёт их Observer.

Такой принцип использования операторов позволяет производить над последовательностью разные преобразования одно за другим.

## skip

В противовес оператору take skip позволяет пропустить несколько первых элементов. В Consumer:

```
fun execSkip() {
    producer.createJust()
        .skip(2)
        .subscribe({ s ->
            println("onNext: $s")
        }, {
            println("onError: ${it.message}")
        })
}
```

## map

Один из ключевых операторов — map. Этот оператор преобразует элементы цепочки согласно переданному ему правилу. Чаще всего это какая-нибудь лямбда. В Consumer:

```
fun execMap() {
    producer.createJust()
        .map { it + it }
        .subscribe({ s ->
```

```
            println("onNext: $s")
        }, {
            println("onError: ${it.message}")
        })
}
```

Выражение приписывает к каждой переданной в него строке саму строку. Результат:

```
onNext: 11
onNext: 22
onNext: 33
onNext: 44
```

## distinct

Оператор distinct, как следует из его названия, отсеивает дубликаты. В Consumer:

```
fun execDistinct() {
    producer.createJust()
        .distinct()
        .subscribe({ s ->
            println("onNext: $s")
        }, {
            println("onError: ${it.message}")
        })
}
```

Результат:

```
onNext: 1
onNext: 2
onNext: 3
```

Мы видим, что вторую тройку отсеял оператор.

## filter

Суть оператора — в его названии. Отфильтруем все строки, представляющие числа, меньшие или равные единице. В Consumer:

```
fun execFilter() {
    producer.createJust()
        .filter() { it.toInt() > 1 }
        .subscribe({ s ->
            println("onNext: $s")
        }, {
            println("onError: ${it.message}")
```

```
        })
}
```

Результат:

```
onNext: 2
onNext: 3
onNext: 3
```

## merge

Один из операторов объединения. Создадим ещё одну функцию в Producer, которая вернёт немного другой источник:

```
class Producer {
    fun createJust() = Observable.just("1", "2", "3", "3")
    fun createJust2() = Observable.just("4", "5", "6")
}
```

Теперь применим к этим источникам оператор merge в Producer:

```
fun execMerge(){
    producer.createJust()
        .mergeWith(producer.createJust2())
        .subscribe({ s ->
            println("onNext: $s")
        }, {
            println("onError: ${it.message}")
        })
}
```

Сливаем первый источник со вторым. В результате получаем следующий вывод:

```
onNext: 1
onNext: 2
onNext: 3
onNext: 3
onNext: 4
onNext: 5
onNext: 6
```

Так как у нас довольно простой пример, все значения выдаются последовательно. Однако на самом деле порядок элементов не гарантируется, и элементы второго источника могут выдаваться между элементами первого.

## flatMap

Оператор flatMap похож на map, также применяет функцию к каждому излучаемому элементу, но эта функция возвращает Obsevable. То есть один излучаемый элемент исходного источника через flatMap порождает другие. Проще говоря, flatMap из каждого элемента создаёт новый источник, после чего выполняет слияние этих источников, похожее на применение над ними оператора merge. Разберёмся на примерах. В Consumer:

```
fun execFlatMap() {
    producer.createJust()
        .flatMap {
            val delay = Random.nextInt(1000).toLong()
            return@flatMap Observable.just(it + "x").delay(delay,
TimeUnit.MILLISECONDS)
        }
        .subscribe({ s ->
            println("onNext: $s")
        }, {
            println("onError: ${it.message}")
        })
}
```

Прежде чем мы начнём разбираться с flatMap, поговорим о delay. На указанное время оператор delay откладывает подписку на источник. Это случайное время до одной секунды. Применяем flatMap к just, приписывая в конец каждой получаемой строки x. Для каждого элемента посредством оператора just генерируется новый источник, который будет работать и завершится независимо от остальных. Каждый новый источник отложит подписку на случайное время до секунды, используя оператор delay. Таким образом, все они завершатся в разное время, после чего результаты выполнения сольются обратно в один источник. В итоге получим следующий вывод:

```
onNext: 3x
onNext: 2x
onNext: 1x
onNext: 3x
```

Каждое значение заняло место в последовательности, отталкиваясь от времени завершения работы соответствующего источника .zip.

Оператор zip позволяет подписаться на несколько источников параллельно и получать их значения одновременно. Это удобно, например, когда требуется сделать несколько сетевых вызовов параллельно. Рассмотрим пример. В Consumer:

```
fun execZip() {
    val observable1 = Observable.just("1").delay(1, TimeUnit.SECONDS)
    val observable2 = Observable.just("2").delay(2, TimeUnit.SECONDS)
    val observable3 = Observable.just("3").delay(4, TimeUnit.SECONDS)
    val observable4 = Observable.just("4").delay(6, TimeUnit.SECONDS)

    Observable.zip(observable1, observable2, observable3, observable4,
Function4<String, String, String, String, List<String>> { t1, t2, t3, t4 ->
        return@Function4 listOf(t1, t2, t3, t4)
    })
        .subscribeOn(Schedulers.computation())
        .subscribe({
            println("Zip result: $it" )
        },{
            println("onError: ${it.message}")
        })
}
```

У нас есть четыре источника, каждый из которых излучает по одному элементу. Применим к ним оператор zip, отдав их в качестве аргументов. Последним аргументом отдадим анонимную реализацию интерфейса Function4. Цифра в конце имени интерфейса говорит о количестве источников. Есть аналогичные интерфейсы до 9.

В записи Function4<String, String, String, String, List<String>> первые четыре типа — типы элементов, возвращаемых источниками. Последний — тип, который будет возвращать источник, возвращаемый самим zip.

Оператор работает так:

1. Осуществляется подписка на все переданные источники.
2. От каждого ожидается результат.
3. В функцию передаются результаты в качестве аргументов, где они обрабатываются подходящим образом.

Мы же просто создаём из них список, чем и обусловлен последний тип List<String>. Аргументы расположены в том порядке, в котором указываются источники, то есть в t1 — значение, излучённое источником observable1. Посмотрим на вывод кода:

```
Zip result: [1,2,3,4]
```

Zip дождался значения от каждого источника, после чего отдал их нам в качестве аргументов. Возникает вопрос: что, если наши источники будут излучать разное количество значений? Всё просто. Функция не будет вызываться до тех пор, пока не появится значение от каждого из источника.

Например, если в одном из источников мы добавим один элемент, то результат выполнения никак не поменяется. Мы получим только одно срабатывание оператора с первыми значениями. Второе срабатывание появится, если добавить минимум по одному элементу в каждый источник. Для третьего срабатывания — минимум по три и т. д. Можно поэкспериментировать с этим самостоятельно.

Ещё один важный момент, который вытекает из вышесказанного: при получении ошибки от одного из источников zip больше не сработает.

Операторов очень много. Мы рассмотрели лишь небольшую часть для демонстрации. Полный список операторов — здесь.

### Заключение

1. Каждый последующий оператор не меняет Observable, а создаёт поверх него новый.
2. В цепочке операторов каждый следующий оператор берёт данные предыдущего, меняет их заданным образом и возвращает новый Observable.

# Практическое задание

1. Переделайте взаимодействие модели и логики в коде из второго урока на Rx-паттерн.
2. *Самостоятельно изучите оператор switchMap. Разберитесь, как он работает и чем отличается от flatMap. Сформулируйте и напишите ответ в комментарии к практическому заданию. Для экспериментов воспользуйтесь приведённым на уроке примером с flatMap, замените его на switchMap, а остальное оставьте без изменений.

# Дополнительные материалы

1. Статья «Исследуем RxJava 2 для Android».

# Используемая литература


1. ReactiveX.io.
2. Overview (RxJava Javadoc 2.2.20).
3. RxJava: Reactive Extensions for the JVM.
4. RxJava Wiki.

5. Operators.