# Освобождение потоков (disposing)

Disposable Является  связующим звеном между Observable и active Observer, вызов его dispose()метода останавливает выбросы и удаляет все ресурсы, используемые для этого Observer:

**fun** **main**() {

**val** seconds = **Observable**.**interval**(1, **TimeUnit**.**SECONDS**)

**val** disposable = seconds.**subscribe** { **println**("Received: $it") }

**TimeUnit**.**SECONDS**.**sleep**(5)

*// Dispose and stop emissions.*

disposable.**dispose**()

*// Sleep 5 secs: no new emissions.*

**TimeUnit**.**SECONDS**.**sleep**(5)

}

Received: 0

Received: 1

Received: 2

Received: 3

Received: 4

**Disposable в наблюдателе**

Observer получает свое собственное при подписке в качестве Disposable аргумента onSubscribe(d: Disposable). приведен пример Observer самоутилизации после потребления выбросов:

**fun** **main**() {

**val** seconds = **Observable**.**interval**(1, **TimeUnit**.**SECONDS**)

seconds.**subscribe**(**MyObserver**())

**TimeUnit**.**SECONDS**.**sleep**(5)

}

**class** **MyObserver**<**T**> : **Observer**<**T**> {

**private** **lateinit** **var** disposable: **Disposable**

**override** **fun** **onSubscribe**(disposable: **Disposable**) {

**this**.disposable = disposable

}

**override** **fun** **onNext**(t: **T**) {

**println**("Received: $t")

disposable.**dispose**() *// Self dispose*

}

**override** **fun** **onError**(e: **Throwable**) {

e.**printStackTrace**()

}

**override** **fun** **onComplete**() {

**println**("Done !")

}

}

Received: 0

По умолчанию вызов subscribe() с Observer экземпляром не возвращает a Disposable, но его можно получить, расширив Resource Observer и подписавшись с помощью subscribeWith():

**fun** **main**() {

**val** seconds = **Observable**.**interval**(1, **TimeUnit**.**SECONDS**)

**val** disposable: **Disposable** = seconds.**subscribeWith**(**MyResourceObserver**())

**TimeUnit**.**SECONDS**.**sleep**(5)

*// Dispose and stop emissions.*

disposable.**dispose**()

*// Sleep 5 secs: no new emissions.*

**TimeUnit**.**SECONDS**.**sleep**(5)

}

**class** **MyResourceObserver**<**T**> : **ResourceObserver**<**T**>() {

**override** **fun** **onNext**(t: **T**) {

**println**("Received: $t")

}

**override** **fun** **onError**(e: **Throwable**) {

e.**printStackTrace**()

}

**override** **fun** **onComplete**() {

**println**("Done !")

}

}

Received: 0

Received: 1

Received: 2

Received: 3

Received: 4

**CompositeDisposable**

Для управления и удаления нескольких подписок CompositeDisposable полезно:

**fun** **main**() {

**val** disposables = **CompositeDisposable**()

**val** seconds = **Observable**.**interval**(1, **TimeUnit**.**SECONDS**)

*// Subscribe and capture disposables*

**val** disposable1 = seconds.**subscribe** { **println**("Observer 1: $it") }

**val** disposable2 = seconds.**subscribe** { **println**("Observer 2: $it") }

*// Put disposables into CompositeDisposable*

disposables.**addAll**(disposable1, disposable2)

*// Sleep 5 secs*

**TimeUnit**.**SECONDS**.**sleep**(3)

*//dispose all disposables*

disposables.**dispose**()

*//sleep 5 seconds: no emissions.*

**TimeUnit**.**SECONDS**.**sleep**(3)

}

Observer 1: 0

Observer 2: 0

Observer 1: 1

Observer 2: 1

Observer 1: 2

Observer 2: 2

**Удаление с помощью Observable.create()**

В случае вызова Observable.create() и результат возврата long-running или infinite Observable , функция isDisposed() должна регулярно проверять, продолжать ли отправлять запросы:

**fun** **main**() {

**val** source: **Observable**<**Int**> = **Observable**.**create** { observableEmitter ->

**try** {

**for** (i **in** 0**..**1\_000\_000) {

**if** (observableEmitter.isDisposed)

**return**@create

observableEmitter.**onNext**(i)

}

observableEmitter.**onComplete**()

} **catch** (e: **Exception**) {

observableEmitter.**onError**(e)

}

}

}

В случае , если Observable.create() обернут вокруг какого-либо ресурса, удаление этого ресурса должно быть обработано для предотвращения утечек. ObservableEmitter имеет setCancellable() и setDisposable()методы для этого:

**fun** <**T**> **valuesOf**(fxObservable: **ObservableValue**<**T**>): **Observable**<**T**> {

**return** **Observable**.**create** { observableEmitter ->

*// Emit initial state*

observableEmitter.**onNext**(fxObservable.value)

*// Emit value changes uses a listener*

**val** listener = **ChangeListener**<**T**> { \_, \_, current -> observableEmitter.**onNext**(current) }

*// Add listener to ObservableValue*

fxObservable.**addListener**(listener)

*// Handle disposing by specifying cancellable*

observableEmitter.**setCancellable** { fxObservable.**removeListener**(listener) }

}

}

# Реактивные операторы (полный каталог)

Каждая языковая реализация ReactiveX реализует набор операторов. Несмотря на то, что между реализациями существует много общего, есть также некоторые операторы, которые реализованы только в определенных реализациях. Кроме того, каждая реализация имеет тенденцию называть свои операторы так, чтобы они напоминали аналогичные методы, которые уже знакомы из других контекстов этого языка.

**Цепочки операторов**

Большинство операторов работают с Observable и возвращают Observable. Это позволяет применять эти операторы один за другим, в цепочке. Каждый оператор в цепочке изменяет Observable, являющийся результатом операции предыдущего оператора.

Существуют и другие шаблоны, такие как шаблон Builder, в котором множество методов определенного класса воздействуют на элемент того же класса, изменяя этот объект посредством операции метода. Эти шаблоны также позволяют вам связывать методы аналогичным образом. Но в то время как в Builder Pattern порядок появления методов в цепочке обычно не имеет значения, с Observable операторами порядок имеет значение .

Цепочка операторов Observable не работает независимо от исходного Observable, который создает цепочку, но они работают по очереди , каждый из которых работает с Observable, сгенерированным оператором, непосредственно предшествующим в цепочке.

**Операторы ReactiveX**

Сначала перечислим операторы, которые можно считать «основными» в ReactiveX. Далее «дерево решений», которое может помочь выбрать оператора, наиболее подходящего для вашего варианта использования. Есть алфавитный список большинства операторов, доступных во многих языковых реализациях ReactiveX.

**Операторы по категориям**

**Создание наблюдаемых**

Операторы, которые создают новые Observables.

* [**Create**](https://reactivex.io/documentation/operators/create.html)— создать Observable с нуля, программно вызывая методы наблюдателя
* [**Defer**](https://reactivex.io/documentation/operators/defer.html)— не создавать Observable, пока наблюдатель не подпишется, и создавать новый Observable для каждого наблюдателя
* [**Empty// Never—Throw**](https://reactivex.io/documentation/operators/empty-never-throw.html) создаем Observable с очень точным и ограниченным поведением
* [**From**](https://reactivex.io/documentation/operators/from.html)— преобразовать какой-либо другой объект или структуру данных в Observable
* [**Interval**](https://reactivex.io/documentation/operators/interval.html)— создать Observable, который выдает последовательность целых чисел, разделенных определенным интервалом времени
* [**Just**](https://reactivex.io/documentation/operators/just.html)— конвертировать объект или набор объектов в Observable, который излучает те или иные объекты
* [**Range**](https://reactivex.io/documentation/operators/range.html)— создать Observable, который выдает диапазон последовательных целых чисел
* [**Repeat**](https://reactivex.io/documentation/operators/repeat.html)- создать Observable, который многократно испускает определенный элемент или последовательность элементов
* [**Start**](https://reactivex.io/documentation/operators/start.html)— создать Observable, который выдает возвращаемое значение функции
* [**Timer**](https://reactivex.io/documentation/operators/timer.html)— создать Observable, который испускает один элемент после заданной задержки

**Преобразование наблюдаемых**

Операторы, преобразующие элементы, испускаемые Observable.

* [**Buffer**](https://reactivex.io/documentation/operators/buffer.html)— периодически собирать элементы из Observable в пакеты и выпускать эти пакеты, а не выпускать элементы по одному
* [**FlatMap**](https://reactivex.io/documentation/operators/flatmap.html)— преобразовать элементы, испускаемые Observable, в Observable, а затем свести выбросы из них в один Observable.
* [**GroupBy**](https://reactivex.io/documentation/operators/groupby.html)- разделить Observable на набор Observable, каждый из которых испускает другую группу элементов из исходного Observable, организованную по ключу
* [**Map**](https://reactivex.io/documentation/operators/map.html)- преобразовать элементы, испускаемые Observable, применяя функцию к каждому элементу
* [**Scan**](https://reactivex.io/documentation/operators/scan.html)— применить функцию к каждому элементу, испускаемому Observable, последовательно, и испускать каждое последующее значение
* [**Window**](https://reactivex.io/documentation/operators/window.html)— периодически подразделять элементы из Observable на окна Observable и создавать эти окна, а не создавать элементы по одному

**Фильтрация наблюдаемых**

Операторы, которые выборочно испускают элементы из исходного Observable.

* [**Debounce**](https://reactivex.io/documentation/operators/debounce.html)— испускать элемент из Observable только в том случае, если в течение определенного промежутка времени он не излучал другой элемент
* [**Distinct**](https://reactivex.io/documentation/operators/distinct.html)— подавлять повторяющиеся элементы, испускаемые Observable
* [**ElementAt**](https://reactivex.io/documentation/operators/elementat.html)— испускать только элемент *n* , испускаемый Observable
* [**Filter**](https://reactivex.io/documentation/operators/filter.html)— испускать только те элементы из Observable, которые проходят предикатную проверку
* [**First**](https://reactivex.io/documentation/operators/first.html)— испускать только первый элемент или первый элемент, отвечающий условию, из Observable
* [**IgnoreElements**](https://reactivex.io/documentation/operators/ignoreelements.html)— не испускать никаких элементов из Observable, но отражать его уведомление о прекращении
* [**Last**](https://reactivex.io/documentation/operators/last.html)— испускать только последний элемент, испускаемый Observable
* [**Sample**](https://reactivex.io/documentation/operators/sample.html)— испускать самый последний элемент, испускаемый Observable в течение периодических интервалов времени
* [**Skip**](https://reactivex.io/documentation/operators/skip.html)— подавить первые *n* элементов, испускаемых Observable
* [**SkipLast**](https://reactivex.io/documentation/operators/skiplast.html)— подавить последние *n* элементов, испускаемых Observable
* [**Take**](https://reactivex.io/documentation/operators/take.html)— испускать только первые *n* элементов, испускаемых Observable
* [**TakeLast**](https://reactivex.io/documentation/operators/takelast.html)— испускать только последние *n* элементов, испускаемых Observable

**Объединение наблюдаемых**

Операторы, которые работают с несколькими исходными Observable для создания одного Observable

* [**And// Then—When**](https://reactivex.io/documentation/operators/and-then-when.html) объединить наборы элементов, испускаемых двумя или более Observable с помощьюPatternиPlanпосредников
* [**CombineLatest**](https://reactivex.io/documentation/operators/combinelatest.html)— когда элемент испускается одним из двух Observable, объединяйте последний элемент, испускаемый каждым Observable с помощью указанной функции, и испускайте элементы на основе результатов этой функции.
* [**Join**](https://reactivex.io/documentation/operators/join.html)- объединять элементы, испускаемые двумя наблюдаемыми, всякий раз, когда элемент из одного наблюдаемого испускается в течение временного окна, определенного в соответствии с элементом, испускаемым другим наблюдаемым
* [**Merge**](https://reactivex.io/documentation/operators/merge.html)— объединить несколько Observables в один, объединив их выбросы
* [**StartWith**](https://reactivex.io/documentation/operators/startwith.html)— испускать указанную последовательность элементов, прежде чем начать испускать элементы из исходного Observable
* [**Switch**](https://reactivex.io/documentation/operators/switch.html)- преобразовать Observable, который испускает Observables, в один Observable, который испускает элементы, испускаемые самым последним из этих Observables
* [**Zip**](https://reactivex.io/documentation/operators/zip.html)- объединять выбросы нескольких Observable вместе с помощью указанной функции и испускать отдельные элементы для каждой комбинации на основе результатов этой функции.

**Операторы обработки ошибок**

Операторы, которые помогают восстанавливаться после уведомлений об ошибках от Observable

* [**Catch**](https://reactivex.io/documentation/operators/catch.html)— восстановиться после onErrorуведомления, продолжив последовательность без ошибок
* [**Retry**](https://reactivex.io/documentation/operators/retry.html)— если исходный Observable отправляет onErrorуведомление, повторно подпишитесь на него в надежде, что оно завершится без ошибок

**Наблюдаемые служебные операторы**

Набор полезных операторов для работы с Observables

* [**Delay**](https://reactivex.io/documentation/operators/delay.html)— сдвинуть выбросы от Observable вперед во времени на определенную величину
* [**Do**](https://reactivex.io/documentation/operators/do.html)— зарегистрировать действие для различных событий жизненного цикла Observable
* [**Materialize/Dematerialize**](https://reactivex.io/documentation/operators/materialize-dematerialize.html) — представлять как отправленные элементы, так и уведомления, отправленные как отправленные элементы, или отменить этот процесс
* [**ObserveOn**](https://reactivex.io/documentation/operators/observeon.html)— указать планировщик, на котором наблюдатель будет наблюдать за этим Observable
* [**Serialize**](https://reactivex.io/documentation/operators/serialize.html)- заставить Observable делать сериализованные вызовы и вести себя хорошо
* [**Subscribe**](https://reactivex.io/documentation/operators/subscribe.html)- работать с выбросами и уведомлениями от Observable
* [**SubscribeOn**](https://reactivex.io/documentation/operators/subscribeon.html)— указать планировщик, который должен использовать Observable, когда он подписан
* [**TimeInterval**](https://reactivex.io/documentation/operators/timeinterval.html)- преобразовать Observable, который испускает элементы, в тот, который испускает указания количества времени, прошедшего между этими выбросами
* [**Timeout**](https://reactivex.io/documentation/operators/timeout.html)— зеркально отразить исходный Observable, но выдавать уведомление об ошибке, если в течение определенного периода времени отсутствуют какие-либо испускаемые элементы.
* [**Timestamp**](https://reactivex.io/documentation/operators/timestamp.html)— прикрепить временную метку к каждому элементу, испускаемому Observable
* [**Using**](https://reactivex.io/documentation/operators/using.html)— создать одноразовый ресурс, который имеет тот же срок службы, что и Observable

**Условные и логические операторы**

Операторы, которые оценивают один или несколько Observables или элементов, испускаемых Observables

* [**All**](https://reactivex.io/documentation/operators/all.html)— определить, соответствуют ли все элементы, испускаемые Observable, некоторым критериям
* [**Amb**](https://reactivex.io/documentation/operators/amb.html)- при наличии двух или более исходных Observables испускать все элементы только из первого из этих Observables, чтобы испускать элемент
* [**Contains**](https://reactivex.io/documentation/operators/contains.html)— определить, испускает ли Observable конкретный элемент или нет
* [**DefaultIfEmpty**](https://reactivex.io/documentation/operators/defaultifempty.html)— испускать элементы из исходного Observable или элемент по умолчанию, если исходный Observable ничего не испускает
* [**SequenceEqual**](https://reactivex.io/documentation/operators/sequenceequal.html)— определить, испускают ли два Observable одну и ту же последовательность элементов
* [**SkipUntil**](https://reactivex.io/documentation/operators/skipuntil.html)- отбрасывать элементы, испускаемые Observable, до тех пор, пока второй Observable не испустит предмет
* [**SkipWhile**](https://reactivex.io/documentation/operators/skipwhile.html)— отбрасывать элементы, испускаемые Observable, пока указанное условие не станет ложным
* [**TakeUntil**](https://reactivex.io/documentation/operators/takeuntil.html)- отбрасывать элементы, испускаемые Observable после того, как второй Observable испускает элемент или завершается
* [**TakeWhile**](https://reactivex.io/documentation/operators/takewhile.html)— отбрасывать элементы, испускаемые Observable после того, как указанное условие становится ложным

**Математические и агрегатные операторы**

Операторы, которые работают со всей последовательностью элементов, испускаемых Observable

* [**Average**](https://reactivex.io/documentation/operators/average.html)— вычисляет среднее число чисел, испускаемых Observable, и выдает это среднее
* [**Concat**](https://reactivex.io/documentation/operators/concat.html)— испускать выбросы от двух или более Observable, не чередуя их
* [**Count**](https://reactivex.io/documentation/operators/count.html)— подсчитать количество элементов, испускаемых исходным Observable, и испускать только это значение
* [**Max**](https://reactivex.io/documentation/operators/max.html)- определить и испустить элемент с максимальным значением, испускаемый Observable
* [**Min**](https://reactivex.io/documentation/operators/min.html)- определить и выдать элемент с минимальным значением, испускаемый Observable
* [**Reduce**](https://reactivex.io/documentation/operators/reduce.html)— применить функцию к каждому элементу, испускаемому Observable, последовательно, и выдать окончательное значение
* [**Sum**](https://reactivex.io/documentation/operators/sum.html)— вычислить сумму чисел, испускаемых Observable, и испустить эту сумму

**Операторы противодавления**

* [**операторы обратного давления**](https://reactivex.io/documentation/operators/backpressure.html) — стратегии для работы с наблюдаемыми, которые производят элементы быстрее, чем их наблюдатели потребляют их.

**Подключаемые наблюдаемые операторы**

Специализированные Observables с более точным контролем динамики подписки.

* [**Connect**](https://reactivex.io/documentation/operators/connect.html)- указать подключаемому Observable начать отправку элементов своим подписчикам
* [**Publish**](https://reactivex.io/documentation/operators/publish.html)— преобразовать обычный Observable в подключаемый Observable
* [**RefCount**](https://reactivex.io/documentation/operators/refcount.html)— заставить Connectable Observable вести себя как обычный Observable
* [**Replay**](https://reactivex.io/documentation/operators/replay.html)— убедитесь, что все наблюдатели видят одну и ту же последовательность испускаемых элементов, даже если они подписываются после того, как Observable начал излучать элементы

**Операторы для преобразования наблюдаемых**

* [**To**](https://reactivex.io/documentation/operators/to.html)— преобразовать Observable в другой объект или структуру данных

**Дерево решений наблюдаемых операторов**

Это дерево может помочь вам найти нужный оператор ReactiveX Observable.

**Я хочу создать новый Observable**

* который излучает определенный предмет [Просто](https://reactivex.io/documentation/operators/just.html)
* который был возвращен из функции, вызванной во время подписки [Начинать](https://reactivex.io/documentation/operators/start.html)
* который был возвращен из Action, Callable, Runnableили чего-то в этом роде, вызванного во время подписки [Из](https://reactivex.io/documentation/operators/from.html)
* после указанной задержки [Таймер](https://reactivex.io/documentation/operators/timer.html)
* который извлекает свои выбросы из определенного Array, Iterableили что -то в этом роде [Из](https://reactivex.io/documentation/operators/from.html)
* извлекая его из будущего [Начинать](https://reactivex.io/documentation/operators/start.html)
* который получает свою последовательность из Future [Из](https://reactivex.io/documentation/operators/from.html)
* который многократно выдает последовательность элементов [Повторение](https://reactivex.io/documentation/operators/repeat.html)
* с нуля, со своей логикой [Создавать](https://reactivex.io/documentation/operators/create.html)
* за каждого подписавшегося наблюдателя [Отложить](https://reactivex.io/documentation/operators/defer.html)
* который выдает последовательность целых чисел [Диапазон](https://reactivex.io/documentation/operators/range.html)
* в определенные промежутки времени [Интервал](https://reactivex.io/documentation/operators/interval.html)
* после указанной задержки [Таймер](https://reactivex.io/documentation/operators/timer.html)
* который завершается без испускания элементов [Пустой](https://reactivex.io/documentation/operators/empty-never-throw.html)
* это вообще ничего не делает [Никогда](https://reactivex.io/documentation/operators/empty-never-throw.html)

**Я хочу создать Observable, объединив другие Observables**

и испускать все элементы из всех Observables в любом порядке их получения [Объединить](https://reactivex.io/documentation/operators/merge.html)

и испускать все элементы из всех Observables, по одному Observable за раз [Конкат](https://reactivex.io/documentation/operators/concat.html)

последовательно комбинируя элементы из двух или более Observable, чтобы получить новые элементы для испускания

всякий раз , когда каждый из Observables испускает новый элемент [Почтовый индекс](https://reactivex.io/documentation/operators/zip.html)

всякий раз , когда какой- либо из Observables испускает новый элемент [ОбъединитьПоследние](https://reactivex.io/documentation/operators/combinelatest.html)

всякий раз, когда элемент испускается одним Observable в окне, определяемом элементом, испускаемым другим [Присоединиться](https://reactivex.io/documentation/operators/join.html)

с помощью Patternи Planпосредников [И/Тогда/Когда](https://reactivex.io/documentation/operators/and-then-when.html)

и испускать элементы только из самых последних испущенных из этих Observables [Выключатель](https://reactivex.io/documentation/operators/switch.html)

**Я хочу испускать элементы из Observable после их преобразования**

по одному с функцией [карта](https://reactivex.io/documentation/operators/map.html)

испуская все элементы, испускаемые соответствующими Observables [Плоская карта](https://reactivex.io/documentation/operators/flatmap.html)

один Observable за раз, в порядке их испускания [ConcatMap](https://reactivex.io/documentation/operators/flatmap.html)

на основе всех предметов, которые им предшествовали [Сканировать](https://reactivex.io/documentation/operators/scan.html)

прикрепив к ним метку времени [Отметка времени](https://reactivex.io/documentation/operators/timestamp.html)

в показатель количества времени, прошедшего до выброса предмета [Временной интервал](https://reactivex.io/documentation/operators/timeinterval.html)

**Я хочу сдвинуть элементы, испускаемые Observable, вперед во времени, прежде чем переиздавать их** [Задерживать](https://reactivex.io/documentation/operators/delay.html)

**Я хочу преобразовать элементы и уведомления из Observable в элементы и повторно отправить их**

оборачивая их в Notification объекты [Материализоваться](https://reactivex.io/documentation/operators/materialize-dematerialize.html)

который я могу снова развернуть с помощью [Дематериализоваться](https://reactivex.io/documentation/operators/materialize-dematerialize.html)

**Я хочу игнорировать все элементы, испускаемые Observable, и передавать только его уведомление о завершении/ошибке**

[ИгнорЭлементс](https://reactivex.io/documentation/operators/ignoreelements.html)

**Я хочу отразить Observable, но префикс элементов к его последовательности** [Начать с](https://reactivex.io/documentation/operators/startwith.html)

только если его последовательность пуста [По умолчанию, если пусто](https://reactivex.io/documentation/operators/defaultifempty.html)

**Я хочу собирать элементы из Observable и повторно отправлять их в виде буферов элементов.** [Буфер](https://reactivex.io/documentation/operators/buffer.html)

содержащий только последние испущенные элементы [TakeLastBuffer](https://reactivex.io/documentation/operators/takelast.html)

**Я хочу разделить один Observable на несколько Observables** [Окно](https://reactivex.io/documentation/operators/window.html)

чтобы похожие элементы попадали в один и тот же Observable [Группа по](https://reactivex.io/documentation/operators/groupby.html)

**Я хочу получить определенный элемент, испускаемый Observable:**

последний элемент, выпущенный перед его завершением [Последний](https://reactivex.io/documentation/operators/last.html)

единственный предмет, который он испустил [Одинокий](https://reactivex.io/documentation/operators/first.html)

первый элемент, который он испустил [Первый](https://reactivex.io/documentation/operators/first.html)

**Я хочу перевыпустить только определенные элементы из Observable**

путем фильтрации тех, которые не соответствуют какому-либо предикату [Фильтр](https://reactivex.io/documentation/operators/filter.html)

то есть только первый пункт [Первый](https://reactivex.io/documentation/operators/first.html)

то есть только первый элемент s[Брать](https://reactivex.io/documentation/operators/take.html)

то есть только последний пункт [Последний](https://reactivex.io/documentation/operators/last.html)

то есть только пункт *n* [ЭлементВ](https://reactivex.io/documentation/operators/elementat.html)

то есть только те элементы после первых элементов

то есть после первых *n* элементов [Пропускать](https://reactivex.io/documentation/operators/skip.html)

то есть до тех пор, пока один из этих элементов не будет соответствовать предикату [Пропустить во время](https://reactivex.io/documentation/operators/skipwhile.html)

то есть после начального периода времени [Пропускать](https://reactivex.io/documentation/operators/skip.html)

то есть после второго Observable испускает элемент [SkipUntil](https://reactivex.io/documentation/operators/skipuntil.html)

то есть те элементы, кроме последних элементов

то есть, кроме последних *n* элементов [ПропуститьПоследнее](https://reactivex.io/documentation/operators/skiplast.html)

то есть до тех пор, пока один из этих элементов не будет соответствовать предикату

[TakeWhile](https://reactivex.io/documentation/operators/takewhile.html)

то есть, за исключением элементов, испускаемых в течение периода времени до завершения источника [ПропуститьПоследнее](https://reactivex.io/documentation/operators/skiplast.html)

то есть, за исключением элементов, испускаемых после того, как второй Observable испускает элемент [TakeUntil](https://reactivex.io/documentation/operators/takeuntil.html)

путем периодической выборки Observable [Образец](https://reactivex.io/documentation/operators/sample.html)

только испуская элементы, за которыми не следуют другие элементы в течение некоторого времени [Отказаться](https://reactivex.io/documentation/operators/debounce.html)

путем подавления элементов, которые являются дубликатами уже выпущенных элементов [Отчетливый](https://reactivex.io/documentation/operators/distinct.html)

если они сразу следуют за элементом, они являются дубликатами [DistinctionUntilChanged](https://reactivex.io/documentation/operators/distinct.html)

отложив мою подписку на него на некоторое время после того, как он начнет выдавать предметы [Задержка подписки](https://reactivex.io/documentation/operators/delay.html)

**Я хочу перевыпускать элементы из Observable только при условии, что он был первым из набора Observable, испустившим элемент** [амб](https://reactivex.io/documentation/operators/amb.html)

**Я хочу оценить всю последовательность элементов, испускаемых Observable**

и выдать одно логическое значение, указывающее, проходят ли все элементы какой-либо тест [Все](https://reactivex.io/documentation/operators/all.html)

и выдать одно логическое значение, указывающее, выдал ли Observable какой -либо элемент (который проходит какой-либо тест) [Содержит](https://reactivex.io/documentation/operators/contains.html)

и выдать одно логическое значение, указывающее, не выдал ли Observable никаких элементов [Пустой](https://reactivex.io/documentation/operators/contains.html)

и выдать одно логическое значение, указывающее, идентична ли последовательность последовательности, испускаемой вторым Observable [SequenceEqual](https://reactivex.io/documentation/operators/sequenceequal.html)

и выдать среднее значение всех их значений [Средний](https://reactivex.io/documentation/operators/average.html)

и выдать сумму всех их значений [Сумма](https://reactivex.io/documentation/operators/sum.html)

и выдать число, указывающее, сколько элементов было в последовательности [Считать](https://reactivex.io/documentation/operators/count.html)

и выпустить элемент с максимальным значением [Максимум](https://reactivex.io/documentation/operators/max.html)

и выдать элемент с минимальным значением [Мин.](https://reactivex.io/documentation/operators/min.html)

применяя функцию агрегирования к каждому элементу по очереди и выдавая результат [Сканировать](https://reactivex.io/documentation/operators/scan.html)

**Я хочу преобразовать всю последовательность элементов, испускаемых Observable, в какую-то другую структуру данных.** [К](https://reactivex.io/documentation/operators/to.html)

**Я хочу, чтобы оператор работал с определенным**[**планировщиком**](https://reactivex.io/scheduler.html)[Подпишитесь на](https://reactivex.io/documentation/operators/subscribeon.html)

когда он уведомляет наблюдателей [Наблюдение за](https://reactivex.io/documentation/operators/observeon.html)

**Я хочу, чтобы Observable вызывал определенное действие, когда происходят определенные события.** [Делать](https://reactivex.io/documentation/operators/do.html)

**Я хочу Observable, который уведомит наблюдателей об ошибке** [Бросать](https://reactivex.io/documentation/operators/empty-never-throw.html)

если указанный период времени истекает без создания элемента [Тайм-аут](https://reactivex.io/documentation/operators/timeout.html)

**Я хочу, чтобы Observable изящно восстановился**

от тайм-аута, переключившись на резервный Observable [Тайм-аут](https://reactivex.io/documentation/operators/timeout.html)

из восходящего уведомления об ошибке [Ловить](https://reactivex.io/documentation/operators/catch.html)

пытаясь повторно подписаться на вышестоящий Observable [Повторить попытку](https://reactivex.io/documentation/operators/retry.html)

**Я хочу создать ресурс с таким же сроком службы, как и у Observable.** [С использованием](https://reactivex.io/documentation/operators/using.html)

**Я хочу подписаться на Observable и получать Futureблоки, пока Observable не завершится** [Начинать](https://reactivex.io/documentation/operators/start.html)

**Мне нужен Observable, который не начинает отправлять элементы подписчикам, пока их не попросят** [Публиковать](https://reactivex.io/documentation/operators/publish.html)

а затем выдает только последний элемент в своей последовательности [ОпубликоватьПоследний](https://reactivex.io/documentation/operators/publish.html)

а затем выдает полную последовательность даже тем, кто подписался после того, как последовательность началась [Повтор](https://reactivex.io/documentation/operators/replay.html)

но я хочу, чтобы он исчез, как только все его подписчики отпишутся [RefCount](https://reactivex.io/documentation/operators/refcount.html)

а потом я хочу попросить его начать [Соединять](https://reactivex.io/documentation/operators/connect.html)

# Subjects

*Subject — это своего рода мост или прокси, доступный в некоторых реализациях ReactiveX, который действует как наблюдатель(Observer) и наблюдаемый(Observable).Так как он является наблюдателем(observer), он может подписаться на один и более наблюдаемых(Observables), и потому как он наблюдатель(Observer), он может пройти через все элементы, за которыми он наблюдает, повторно передав их, а также может излучать(emit) новые элементы.*

Subject которые есть в RxJava.

* Publish Subject
* Replay Subject
* Behavior Subject
* Async Subject

**Observable**: Предположим, что профессор является наблюдаемым(observable). Профессор учит какой-то теме.

**Observer:** Предположим, что студент наблюдатель(observer). Студент слушает(или наблюдает) тему, которую преподает профессор.

**Publish Subject**

Излучает(emit) 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. И так, Publish будет лучшим выбором для использования.

пример:

*PublishSubject<Integer> source = PublishSubject.create();*

*// Получит 1, 2, 3, 4 и onComplete*

*source.subscribe(getFirstObserver());*

*source.onNext(1);*

*source.onNext(2);*

*source.onNext(3);*

*// Получит 4 и onComplete для следующего наблюдателя тоже.source.subscribe(getSecondObserver());*

*source.onNext(4);*

*source.onComplete();*

**Replay Subject**

Излучает(emit 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. И так, для этого мы должны использовать Replay.

пример:

*ReplaySubject<Integer> source = ReplaySubject.create();*

*// Он получит 1, 2, 3, 4*

*source.subscribe(getFirstObserver());*

*source.onNext(1);*

*source.onNext(2);*

*source.onNext(3);*

*source.onNext(4);*

*source.onComplete();*

*// Он также получит 1, 2, 3, 4 так как он использует Replay Subject*

*source.subscribe(getSecondObserver());*

**Behavior Subject**

Он излучает(emit) совсем недавно созданый 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. Итак, здесь мы будем использовать Behavior.

Пример:

*BehaviorSubject<Integer> source = BehaviorSubject.create();*

*// Получит 1, 2, 3, 4 and onComplete*

*source.subscribe(getFirstObserver());*

*source.onNext(1);*

*source.onNext(2);*

*source.onNext(3);*

*// Получит 3(последний элемент) и 4(последующие элементы) и onComplete*

*source.subscribe(getSecondObserver());*

*source.onNext(4);*

*source.onComplete();*

**Async Subject**

Он 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. Итак, здесь мы будем использовать Async.

пример:

*AsyncSubject<Integer> source = AsyncSubject.create();*

*// Получит только 4 и onComplete*

*source.subscribe(getFirstObserver());*

*source.onNext(1);*

*source.onNext(2);*

*source.onNext(3);*

*// Тоже получит только 4 и onComplete*

*source.subscribe(getSecondObserver());*

*source.onNext(4);*

*source.onComplete();*

# Распараллеливание потоков (Schedulers)

Многопоточное приложение состоит из двух или более частей, которые могут работать параллельно. Это позволяет приложению лучше использовать ядра внутри процессора устройства. Это позволяет выполнять задачи быстрее и обеспечивает более плавный и отзывчивый опыт для пользователя.

Кодирование для параллелизма в Java может быть болезненным, но благодаря RxJava теперь это сделать намного проще. С RxJava просто нужно объявить поток, в котором нужно, чтобы задача была выполнена (декларативно) вместо создания и управления потоками (обязательно).

RxJava использует Schedulers вместе с операторами параллельной subscribeOn() на subscribeOn() и observeOn() для достижения этой цели.

**Планировщики в RxJava 2**

Schedulers в RxJava используются для выполнения единицы работы над потоком. Scheduler предоставляет абстракцию для механизма потоков Android и Java. Если хотите запустить задачу и используете Scheduler для ее выполнения, Scheduler переходит в свой пул потоков (набор потоков, готовых к использованию), а затем запускает задачу в доступном потоке.

Можно указать, что задача должна выполняться в одном конкретном потоке. (Существует два оператора, subscribeOn() и observeOn() , которые можно использовать для указания, в каком потоке из пула потоков Scheduler должна быть выполнена задача.)

В Android длительные процессы или задачи с интенсивным использованием процессора не должны выполняться в главном потоке. Если подписка Observer на Observable проводится в основном потоке, любой связанный оператор также будет работать в основном потоке. В случае длительной задачи (например, выполнение сетевого запроса) или задачи с интенсивным использованием ЦП (например, преобразование изображения) это будет блокировать пользовательский интерфейс до завершения задачи, что приведет к ужасному диалоговому окну ANR (приложение не отвечает) и приложение вылетает. Вместо этого эти операторы могут быть переключены на другой поток с observeOn() оператора observeOn() .

**Типы планировщиков**

Вот некоторые из типов Schedulers доступных в RxJava и RxAndroid чтобы указать тип потока для выполнения задач.

* Schedulers.immediate() : возвращает Scheduler который мгновенно выполняет работу в текущем потоке. Имейте в виду, что это заблокирует текущий поток, поэтому его следует использовать с осторожностью.
* Schedulers.trampoline() : планирование задач в текущем потоке. Эти задачи не выполняются сразу, а выполняются после того, как поток завершил свои текущие задачи. Это отличается от Schedulers.immediate() потому что вместо немедленного выполнения задачи он ожидает завершения текущих задач.
* Schedulers.newThread() : запускает новый поток и возвращает Scheduler для выполнения задачи в новом потоке для каждого Observer . Вы должны быть осторожны при использовании этого, потому что новый поток не используется повторно, а вместо этого уничтожается.
* Schedulers.computation() : это дает нам Scheduler который предназначен для вычислительно-интенсивной работы, такой как преобразование изображений, сложные вычисления и т. Д. Эта операция полностью использует ядра ЦП. Этот Scheduler использует фиксированный размер пула потоков, который зависит от ядер ЦП для оптимального использования. Вы должны быть осторожны, чтобы не создавать больше потоков, чем доступные ядра ЦП, поскольку это может снизить производительность.
* Schedulers.io() : создает и возвращает Scheduler предназначенный для работы, связанной с вводом-выводом, такой как выполнение асинхронных сетевых вызовов или чтение и запись в базу данных. Эти задачи не загружают процессор или используют Schedulers.computation() .
* Schedulers.single() : создает и возвращает Scheduler и выполняет несколько задач последовательно в одном потоке.
* Schedulers.from(Executor executor) : это создаст Scheduler который будет выполнять задачу или единицу работы с данным Executor .
* AndroidSchedulers.mainThread() : это создаст Scheduler который выполняет задачу в главном потоке приложения Android. Этот тип планировщика предоставляется библиотекой RxAndroid .

**Оператор subscribeOn()**

Используя оператор одновременной subscribeOn() , если указать, что Scheduler должен выполнять операцию в Observable восходящем потоке. Затем он отправит значения Observers используя тот же поток. Теперь давайте посмотрим на практический пример:

*import android.os.Bundle;*

*import android.support.v7.app.AppCompatActivity;*

*import android.util.Log;*

*import io.reactivex.Observable;*

*import io.reactivex.ObservableOnSubscribe;*

*import io.reactivex.disposables.Disposable;*

*import io.reactivex.schedulers.Schedulers;*

*public class MainActivity extends AppCompatActivity {*

*private static final String[] STATES = { «Lagos», «Abuja», «Abia»,*

*«Edo», «Enugu», «Niger», «Anambra»};*

*private Disposable mDisposable = null;*

*@Override*

*protected void onCreate(Bundle savedInstanceState) {*

*super.onCreate(savedInstanceState);*

*setContentView(R.layout.activity\_main);*

*Observable<String> observable = Observable.create(dataSource())*

*.subscribeOn(Schedulers.newThread())*

*.doOnComplete(() -> Log.d(«MainActivity», «Complete»));*

*mDisposable = observable.subscribe(s -> {*

*Log.d(«MainActivity», «received » + s + » on thread » + Thread.currentThread().getName());*

*});*

*}*

*private ObservableOnSubscribe<String> dataSource() {*

*return(emitter -> {*

*for(String state : STATES) {*

*emitter.onNext(state);*

*Log.d(«MainActivity», «emitting » + state + » on thread » + Thread.currentThread().getName());*

*Thread.sleep(600);*

*}*

*emitter.onComplete();*

*});*

*}*

*@Override*

*protected void onDestroy() {*

*if (mDisposable != null && !mDisposable.isDisposed()) {*

*mDisposable.dispose();*

*}*

*super.onDestroy();*

*}*

*}*

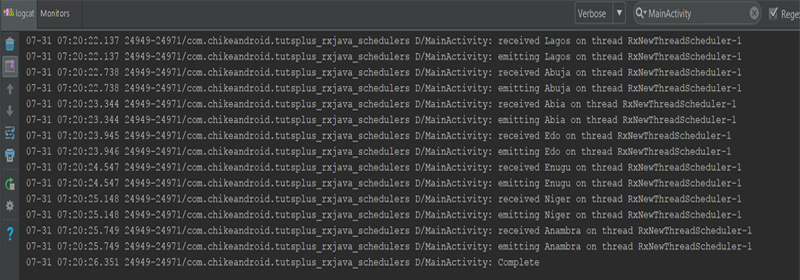
В приведенном выше коде есть статический ArrayList который содержит некоторые города. Также есть поле типа Disposable . Мы получаем экземпляр Disposable , вызывая Observable.subscribe() , и будем использовать его позже, когда вызовем метод dispose() для освобождения любых использованных ресурсов. Это помогает предотвратить утечки памяти. dataSource() (который может возвращать данные из удаленного или локального источника базы данных) вернет ObservableOnSubscribe<T> : это необходимо для того, чтобы мы позже создали собственный Observable используя метод Observable.create() .

Внутри dataSource()перебираем массив, отправляя каждый элемент Observers , вызывая emitter.onNext() . После выдачи каждого значения спим потоком, чтобы смоделировать выполняемую интенсивную работу. Наконец, вызываем метод onComplete() чтобы сообщить Observers что мы завершили передачу значений, и они больше не должны ожидать.

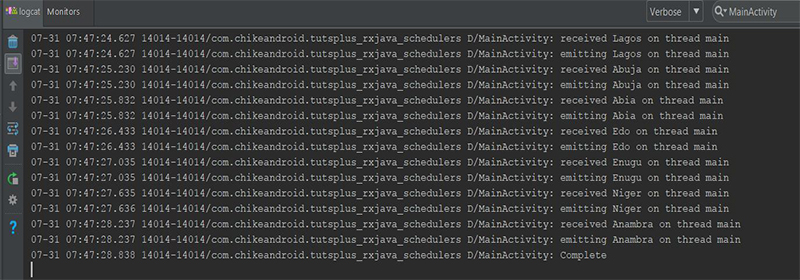
Теперь dataSource() не должен выполняться в основном потоке пользовательского интерфейса. Но как это указано? В приведенном примере предоставили Schedulers.newThread() в качестве аргумента для subscribeOn() . Это означает, что dataSource() будет запущена в новом потоке. Также обратите внимание, что в приведенном выше примере у нас есть только один Observer . Если бы у нас было несколько Observers , каждый из них получал бы свой собственный поток.

Чтобы могли видеть, как это работает, наш Observer распечатывает значения, которые он получает в своем onNext() из Observable .

Когда запустим это и просмотрим наш logcat в Android Studio, видно, что dataSource() метода dataSource() в Observer произошла в том же потоке — RxNewThreadScheduler-1 — в котором Observer их получил.



Если не указать метод .subscribeOn() после метода Observable.create() , он будет выполнен в текущем потоке, который в нашем случае является основным, блокируя таким образом пользовательский интерфейс приложения.

Есть несколько важных деталей, о которых нужно знать, касающихся оператора subscribeOn() . Должен быть только один subscribeOn() в цепочке Observable ; добавление еще одного в любом месте цепочки не будет иметь никакого эффекта вообще. Рекомендуемое место для размещения этого оператора максимально близко к источнику для ясности. Другими словами, поместите его первым в цепочке операторов.

*Observable.create(dataSource())*

*.subscribeOn(Schedulers.computation()) // this has effect*

*.subscribeOn(Schedulers.io()) // has no effect*

*.doOnNext(s -> {*

*saveToCache(s);*

*})*

**Оператор observeOn()**

Оператор параллелизма subscribeOn() будет Observable какой Scheduler использовать, чтобы продвигать выбросы по цепочке Observers к Observers .

С другой стороны, observeOn() оператора одновременного observeOn() состоит в том, чтобы переключать последующие выбросы в другой поток или Scheduler . Этот оператор используется, чтобы контролировать, какой поток нижестоящие потребители будут получать выбросы. Давайте посмотрим на практический пример.

*import android.os.Bundle;*

*import android.support.v7.app.AppCompatActivity;*

*import android.util.Log;*

*import android.widget.TextView;*

*import io.reactivex.Observable;*

*import io.reactivex.ObservableOnSubscribe;*

*import io.reactivex.android.schedulers.AndroidSchedulers;*

*import io.reactivex.disposables.Disposable;*

*import io.reactivex.schedulers.Schedulers;*

*public class ObserveOnActivity extends AppCompatActivity {*

*private Disposable mDisposable = null;*

*@Override*

*protected void onCreate(Bundle savedInstanceState) {*

*super.onCreate(savedInstanceState);*

*setContentView(R.layout.activity\_main);*

*TextView textView = (TextView) findViewById(R.id.tv\_main);*

*Observable<String> observable = Observable.create(dataSource())*

*.subscribeOn(Schedulers.newThread())*

*.observeOn(AndroidSchedulers.mainThread())*

*.doOnComplete(() -> Log.d(«ObserveOnActivity», «Complete»));*

*mDisposable = observable.subscribe(s -> {*

*Log.d(«ObserveOnActivity», «received » + s + » on thread » + Thread.currentThread().getName());*

*textView.setText(s);*

*});*

*}*

*private ObservableOnSubscribe<String> dataSource() {*

*return(emitter -> {*

*Thread.sleep(800);*

*emitter.onNext(«Value»);*

*Log.d(«ObserveOnActivity», «dataSource() on thread » + Thread.currentThread().getName());*

*emitter.onComplete();*

*});*

*}*

*// …*

*}*

В приведенном коде использовали оператор observeOn() а затем передали ему AndroidSchedulers.mainThread() . Переключили поток с Schedulers.newThread() на основной поток Android. Это необходимо, потому что хотим обновить виджет TextView и можем это делать только из основного потока пользовательского интерфейса. Обратите внимание, что если не переключится на основной поток при попытке обновить виджет TextView , приложение завершится CalledFromWrongThreadException и CalledFromWrongThreadException .

В отличие от оператора subscribeOn() оператор observeOn() может применяться несколько раз в цепочке операторов, тем самым изменяя Scheduler более одного раза.

*Observable<String> observable = Observable.create(dataSource())*

*.subscribeOn(Schedulers.newThread())*

*.observeOn(Schedulers.io())*

*.doOnNext(s -> {*

*saveToCache(s);*

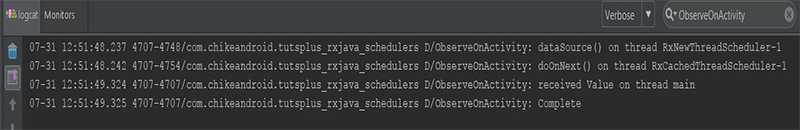
*Log.d(«ObserveOnActivity», «doOnNext() on thread » + Thread.currentThread().getName());*

*})*

*.observeOn(AndroidSchedulers.mainThread())*

*.doOnComplete(() -> Log.d(«ObserveOnActivity», «Complete»));*

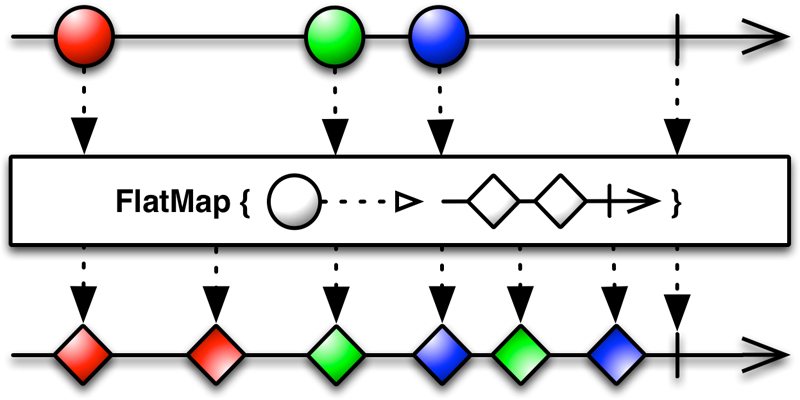
Этот код имеет два оператора observeOn() . Первый использует Schedulers.io() , что означает, что метод saveToCache() будет выполняться в потоке Schedulers.io() . После этого он переключается на AndroidSchedulers.mainThread() где Observers будут получать выбросы от восходящего потока.



**Параллелизм с оператором flatMap ()**

Оператор flatMap() — это еще один очень мощный и важный оператор, который можно использовать для достижения параллелизма. Определение в соответствии с официальной документацией выглядит следующим образом:

Преобразуйте предметы, испускаемые Обсерваторией, в Наблюдаемые, затем сведите выбросы из них в единую Наблюдаемую.



\Посмотрим на практический пример, который использует этот оператор:

*//…*

*@Override*

*protected void onCreate(Bundle savedInstanceState) {*

*// …*

*final String[] states = {«Москва», «Ярославль», «Пушкино», «Н.Новгород»};*

*Observable<String> statesObservable = Observable.fromArray(states);*

*statesObservable.flatMap(*

*s -> Observable.create(getPopulation(s))*

*).subscribe(pair -> Log.d(«MainActivity», pair.first + » population is » + pair.second));*

*}*

*private ObservableOnSubscribe<Pair> getPopulation(String state) {*

*return(emitter -> {*

*Random r = new Random();*

*Log.d(«MainActivity», «getPopulation() for » + state + » called on » + Thread.currentThread().getName());*

*emitter.onNext(new Pair(state, r.nextInt(300000 — 10000) + 10000));*

*emitter.onComplete();*

*});*

*}*

*}*

В Android Studio logcat будет напечатано следующее:

*getPopulation() for Москва called on main*

*Москва population is 80362*

*getPopulation() for Ярославль called on main*

*Ярославль population is 132559*

*getPopulation() for Пушкино called on main*

*Пушкино population is 34106*

*getPopulation() for Н.Новгород called on main*

*Н.Новгород population is 220301*

Из приведенного выше результата видно, что полученные результаты были в том же порядке, что и в массиве. Кроме того, метод getPopulation() для каждого состояния обрабатывался в одном и том же потоке — основном потоке. Это замедляет вывод, потому что они обрабатывались последовательно в главном потоке.

Теперь, чтобы достичь параллелизма с этим оператором, мы хотим, чтобы метод getPopulation() для каждого состояния (эмиссии из statesObservable ) обрабатывался в разных потоках. Это приведет к более быстрой обработке. Для этого мы будем использовать оператор flatMap() , поскольку он создает новую Observable для каждого излучения. Затем мы применяем к нему один оператор параллелизма subscribeOn() , передавая ему Scheduler .

*statesObservable.flatMap(*

*s -> Observable.create(getPopulation(s))*

*.subscribeOn(Schedulers.io())*

*).subscribe(pair -> Log.d(«MainActivity», pair.first + » population is » + pair.second));*

Поскольку каждое излучение создает Observable , flatMap() оператора flatMap() состоит в том, чтобы объединить их вместе и затем отправить их в виде единого потока.

*getPopulation() for Lagos called on RxCachedThreadScheduler-1*

*Lagos population is 143965*

*getPopulation() for Abuja called on RxCachedThreadScheduler-2*

*getPopulation() for Enugu called on RxCachedThreadScheduler-4*

*Abuja population is 158363*

*Enugu population is 271420*

*getPopulation() for Imo called on RxCachedThreadScheduler-3*

*Imo population is 81564*

В приведенном выше результате мы можем наблюдать, что метод getPopulation() каждого состояния обрабатывался в разных потоках. Это значительно ускоряет обработку, но также flatMap() что выбросы от оператора flatMap() , полученные Observer , не соответствуют порядку исходных выбросов в восходящем направлении.

# Противодавление (backpressure) и интерфейс Flowable

Противодавление — это процесс обработки быстрого производителя предметов, например. когда конвейер обработки Observable не может обрабатывать значения достаточно быстро и может столкнуться с исключением Out of Memory , тогда нам нужен способ сообщить вышестоящему производителю о замедлении, этот механизм называется Backpressuring или Backpressure . Проще говоря, мы можем сказать, что когда у нас слишком много данных, передаваемых наблюдателям, и наблюдатели не могут их обработать и сталкиваются с исключением Out of Memory , чтобы предотвратить это, мы должны ( обратное давление ) сказать вышестоящему производителю замедлить работу. .

**Flowable** похож на Observable. Есть два типа классов объектов: Observable и Flowable . Observable

**Разница между Observable и Flowable**: Противодавление было причиной того, что Flowable был введен в RxJava 2.x, поскольку основное различие между Observable и Flowable заключается в том, что Flowables осведомлены о противодавлении, а Observable — нет . Класс Observableимеет неограниченный размер буфера , т.е. Наблюдаемый объект будет буферизовать все и отдавать все подписчику в один момент времени, поэтому в то время, когда Observableон был введен в RxJava 0.x, обратное давление не было включено в него, однако с RxJava 2.x у нас есть новый Observableобъектный класс, называемый Текучесть с включенным противодавлением.

Observable используются, когда у нас есть относительно небольшое количество объектов, которые нужно генерировать с течением времени (<1000), и нет риска , что производитель переполнит потребителей, что может привести кOutOfMemoryException.

Flowable используются, когда у нас относительно большое количество элементов, и нам нужно тщательно контролировать их Producerповедение, чтобы избежать исчерпания ресурсов или перегрузки.

**Терминология обратного давления:**

* Горячие наблюдаемые/источники
* Холодные наблюдаемые/источники
* Многоадресные наблюдаемые/источники

**Горячие наблюдаемые или горячий источник:**

Когда Observables непрерывно испускает объекты в направлении Observers, независимо от того, могут ли Observers обработать это или нет, такие Observables называются Hot Observables или hot sources . Hot Observables обычно плохо справляются с моделью реактивного извлечения и являются лучшими кандидатами для некоторых других стратегий управления потоком, обсуждаемых ниже.

**Холодные наблюдаемые или холодные источники:**

Когда Observables испускают объекты только тогда, когда этого хотят наблюдатели, или мы можем сказать, что объекты, испускаемые Observables, не нуждаются в буферизации , поскольку Observables испускают объект только тогда, когда он нужен наблюдателям, мы также можем назвать это ленивой загрузкой, и этот вид наблюдаемых называется холодными наблюдаемыми или холодными источниками . Холодные наблюдаемые противоположны горячим наблюдаемым.

**Многоадресные наблюдаемые:**

Когда холодный Observable является многоадресным (когда он преобразуется в a ConnectableObservableи connect()вызывается его метод), он фактически становится горячим , и для целей противодавления и управления потоком его следует рассматривать как горячий Observable.

**Предотвращение противодавления:**

Чтобы предотвратить обратное давление, мы преобразуем Observable в Flowable с помощью observable.toFloawable()метода и указываем с ним стратегию буферизации.

**Стратегии обратного давления**

Противодавление встречается на горячих Observables с MissingBackPressureException ie. использование без противодействия или пользовательских операторов PublishProcessor, созданных с помощью , которые происходят в операторе, мы уже знаем, что для предотвращения этого мы используем Flowable и указываем с ним стратегию буферизации.timer()interval()create()observeOn

* Увеличение размеров буфера. Переполнение объектов происходит в наблюдателях из-за скачкообразных источников, пользователь приложения может очень быстро нажимать на представления, поэтому наблюдатели не получают достаточно времени для использования испускаемых объектов из-за этого из RxJava 2.x, чтобы иметь достаточно место для всех значений, программисты могут указать размер своих внутренних буферов, используя чувствительные к обратному давлению операторы с соответствующими параметрами: bufferSize, prefetchили capacityHint. ниже пример. взято из документации RxJava 2.x.

*PublishProcessor<Integer> source = PublishProcessor.create();*

*source.observeOn(Schedulers.computation(), 1024 \* 1024)*

*.subscribe(e -> {}, Throwable::printStackTrace);*

*for (int i = 0; i < 1\_000\_000; i++) {*

*source.onNext(i);*

*}*

Это нельзя рассматривать как постоянное исправление, так как в случае, если источник превышает прогнозируемый размер буфера, может произойти переполнение, для такого сценария мы можем использовать другие операторы, описанные ниже.

* Пакетирование/пропуск значений со стандартными операторами

Чтобы уменьшить вероятность того , что MissingBackpressureExceptionдаже после определения bufferSize, у нас все еще есть возможность использовать стандартные операторы пакетирования (по размеру и/или по времени).

*PublishProcessor<Integer> source = PublishProcessor.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);*

*}*

В тех случаях, когда объекты/значения можно безопасно игнорировать, у нас также есть дросселирующие операторы ( throttleFirst, throttleLast, throttleWithTimeout), однако эти операторы только снижают скорость получения значения нисходящим потоком и, таким образом, они все еще могут привести к MissingBackpressureException.

* onBackpressureBuffer()

Этот оператор без параметров повторно вводит неограниченный буфер между восходящим источником и нижестоящим оператором. Неограниченность означает, что пока у JVM не заканчивается память, она может обрабатывать почти любой объем, поступающий из неустойчивого источника. например.

*Flowable.range(1, 1\_000\_000)*

*.onBackpressureBuffer()*

*.observeOn(Schedulers.computation(), 8)*

*.subscribe(e -> {}, Throwable::printStackTrace);*

Один и тот же оператор можно использовать по-разному, добавив к нему определенные параметры:

* onBackpressureBuffer (целая емкость)
* onBackpressureBuffer (целая емкость, действие при переполнении)
* onBackpressureBuffer (целая емкость, действие при переполнении, стратегия BackpressureOverflowStrategy)

# Тестирование реактивных потоков, протокол RSocket

Это протокол, обеспечивающий семантику Reactive Streams.

Это двоичный протокол связи точка-точка, разработанный для использования в распределенных приложениях. В этом смысле он предоставляет альтернативу другим протоколам, таким как HTTP.

С RSocket можно узнать, когда лучше отправлять запрос, а когда нет. Сделать это с HTTP нельзя.

RSocket имеет реализации на нескольких языках. Библиотека Java построена на Project Reactor и Reactor Netty для транспорта. Это означает, что сигналы от Reactive Streams Publishers в вашем приложении распространяются через RSocket по сети.

Протокол RSocket использует транспортный протокол более низкого уровня для передачи кадров RSocket .

RSocket позволяет общаться с использованием следующих транспортных протоколов:

* TCP
* WebSocket
* Aeron
* HTTP / 2 Stream

RSocket пользуется доверием и поддержкой некоторых крупнейших компаний в Интернете, таких как Netifi, Pivotal, Facebook, Netflix, Alibaba и других.

**Почему RSocket ?**

Использование микросервисов очень популярно сейчас.

Но ситуация усложняется, если решите запустить свои приложения в облаке, где проблемы с сетью, периоды повышенной задержки и большой поток запросов — это то, чего вы не можете полностью избежать. Вместо того, чтобы пытаться исправить проблемы с сетью, лучше сделать вашу архитектуру устойчивой и полностью работоспособной даже при таких условиях.

**Давайте посмотрим на пример:**

У нас есть много микросервисов, которые общаются друг с другом через HTTP. Мы также используем облачные серверы (AWS, GCP, Azure). Каждый компонент предоставляет простые REST API.

***Первая проблема*** — модель взаимодействия запрос/ответ HTTP . Некоторые шаблоны коммуникации трудно реализовать эффективным способом, используя модель взаимодействия запрос / ответ. Даже выполнение простой операции «fire and forget» имеет побочные эффекты — сервер должен отправить ответ обратно клиенту, даже если клиенту это не нужно.

***Вторая проблема*** — производительность.

Обмен сообщениями на основе RabbitMQ, gRPC или даже HTTP 2 с его поддержкой мультиплексирования будет намного лучше с точки зрения производительности и эффективности, чем простой HTTP 1.x.

Дополнительные ресурсы влекут за собой дополнительные расходы, хотя все, что нам нужно, — это простое сообщение “fire and forget”.

Кроме того, множество различных протоколов может создавать серьезные проблемы, связанные с управлением приложениями, особенно если наша система состоит из сотен микросервисов.

Упомянутые выше проблемы являются основными причинами, по которым RSocket был изобретен. Благодаря своей реактивности и встроенной надежной модели взаимодействия RSocket может применяться в различных бизнес-сценариях.

**Разница между gRPC, WebSocket и HTTP**

HTTP.1 — это хорошее решение, проверенное временем, но оно не будет работать на производительность и реактивность.

WebSocket слишком сложен для разработки и в тяжелых стресс-тестах не работает так, как ожидалось.

gRPC — несомненно лучше, чем HTTP и WebSocket, он прост в разработке. Он работает поверх HTTP.1 (то есть, по HTTP.2) — на самом деле, это не совсем так, то есть это просто оболочка над HTTP. 2. Для каждого действия отправляется обычный запрос POST (HTTP). Поэтому это неэффективное использование ресурсов. Он использует прокси (envoy) между браузером и сервером и отправляет регулярные HTTP-запросы POST к прокси, и прокси уже отправляет HTTP.2 на сервер. С помощью больших стресс-тестов он показал себя не очень хорошо, если, например, у нас нет очень мощного сервера.

gRPC и RSocket пытаются решить разные проблемы. gRPC — это среда RPC, использующая HTTP/2.

gRPC и RSocket находятся на разных уровнях в стеке.

gRPC находится на уровне 7 OSI — уровне RPC, построенном поверх HTTP/2.

RSocket — это уровень OSI 5/6, который моделирует семантику Reactive Streams по сети. Реактивные потоки позволяют моделировать асинхронные потоки с backpressure.

gRPC предназначен для работы с семантикой HTTP/2. Если вы хотите отправить его через другой транспорт, вы должны имитировать семантику HTTP / 2. По всей сети он фактически связан с HTTP/2 и TCP.

RSocket требуется только дуплексное соединение — т.е. то, что может отправлять и получать байты. Это может быть TCP, WebSockets и т. д.

У gRPC есть традиционная модель клиент-сервер, потому что она основана на семантике HTTP/2 и RPC. В gRPC клиент подключается к серверу, но сервер не может совершать вызовы клиенту.

RSocket не имеет взаимодействия клиент-сервер в традиционном смысле HTTP. В RSocket, когда клиент подключается к серверу, они оба могут быть запросчиком и ответчиком. Это означает, что после того, как они подключены, RSocket является полностью дуплексным, и запрашивающая сторона сервера может инициировать вызовы запрашивающей стороне клиента — то есть сервер может совершать вызовы в веб-браузер после подключения.

**Разница между WebSocket и RSocket**

Веб-сокеты не обеспечивают backpressure на уровне приложения, только backpressure на уровне байтов на основе TCP. Веб-сокеты также обеспечивают только создание, они не обеспечивают семантику приложения. Разработчик должен разработать протокол приложения для взаимодействия с веб-сокетом.

RSocket обеспечивает кадрирование, семантику приложения, backpressure на уровне приложения, и оно не привязано к конкретному транспорту.

Разница между HTTP и RSocket

RSocket — это протокол приложений для реактивных потоков, который обеспечивает управление потоком приложений по сети, чтобы предотвратить сбои и повысить отказоустойчивость. В отличие от HTTP, RSocket не ожидает ответа или запроса от клиента.

В отличие от этого, RSocket использует идею асинхронной обработки потока с неблокирующим противодавлением , при котором отказавший компонент будет вместо того, чтобы просто отбрасывать трафик, сообщать о своих нагрузках вышестоящим компонентам, заставляя их снижать нагрузку и позволяя системе «Изящно реагировать на нагрузку, а не разрушаться под ней», согласно глоссарию Reactive Manifesto .

**Основные понятия**

RSocket использует кадрирование.

Кадр RSocket — это отдельное сообщение, которое содержит запрос, ответ или обработку протокола. К кадру RSocket может добавляться 24-битное поле длины кадра (Frame Length), представляющее длину кадра в байтах. Зависит от базового транспортного протокола, используемого RSocket, поле длины кадра может не требоваться.

Кадры RSocket начинаются с заголовка RSocket Frame.

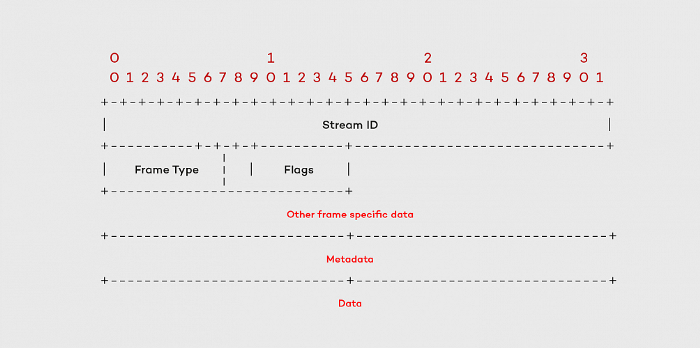
RSocket поддерживает два типа полезных данных: данные и метаданные. Данные и метаданные могут быть закодированы в разных форматах.

В настоящее время в протоколе RSocket имеется 16 типов кадров:

* SETUP - Настройка подключения. Всегда использует идентификатор потока 0.
* REQUEST\_RESPONSE - Используется в модели запроса-ответа. Запрос одного сообщения.
* REQUEST\_STREAM - Запрашивается поток сообщений.
* REQUEST\_FNF - Используется в модели «fire-and-forget». Отправляет сообщение и не ждет ответа.
* REQUEST\_CHANNEL - Используется в модели канала. Запрашивает поток сообщений в обоих направлениях.
* REQUEST\_N - Запрашивает больше данных и используется для контроля потока.
* PAYLOAD - Полезная нагрузка сообщения.
* ERROR - Ошибка на уровне соединения или приложения.
* CANCEL - Отмена невыполненного запроса.

**Принцип работы**

Взаимодействие в RSocket разбито на фреймы. Каждый кадр состоит из заголовка кадра, который содержит идентификатор потока, определение типа кадра и другие данные, относящиеся к типу кадра. За заголовком кадра следуют метаданные и полезная нагрузка — эти части несут данные, указанные пользователем.



Клиент отправляет установочный фрейм на сервер в самом начале связи. Этот кадр можно настроить так, чтобы вы могли добавлять свои собственные правила безопасности или другую информацию, требуемую при инициализации соединения. Следует отметить, что RSocket не различает клиента и сервер после фазы настройки соединения. Каждая сторона может начать отправку данных другой — это делает протокол почти полностью симметричным.

Кадры отправляются в виде потока байтов. Это делает RSocket более эффективным, чем обычные текстовые протоколы. С точки зрения разработчика, легче отлаживать систему, когда JSON летают туда-сюда по сети, но влияние на производительность делает такое удобство очень и очень сомнительным. Протокол не навязывает какой-либо конкретный механизм сериализации/десериализации, он рассматривает кадр как пакет битов, которые могут быть преобразованы во что угодно.

Следующим фактором, который оказывает огромное влияние на производительность RSocket, является мультиплексирование. Протокол создает логические потоки (каналы) поверх единственного физического соединения. Каждый поток имеет свой уникальный идентификатор, который в некоторой степени можно интерпретировать как очередь. Такой дизайн имеет дело с основными проблемами, известными из HTTP 1.x — модель соединения на запрос и слабая производительность «конвейерной обработки».

Более того, RSocket изначально поддерживает передачу больших полезных нагрузок. В таком случае кадр полезной нагрузки разделяется на несколько кадров с дополнительным флагом — порядковым номером данного фрагмента.

RSocket использует Reactor, поэтому на уровне API мы в основном работаем с объектами Mono и Flux.

**Интеграция с Spring**

Maven:

*<dependency>*

*<groupId>org.springframework.integration</groupId>*

*<artifactId>spring-integration-rsocket</artifactId>*

*<version>5.2.3.RELEASE</version>*

*</dependency>*

Gradle:

*compile "org.springframework.integration:spring-integration-rsocket:5.2.3.RELEASE"*

*Spring Boot 2.2 поддерживает установку сервера RSocket через TCP или WebSocket.*

Существует также поддержка клиентов и автоматическая настройка для RSocketRequester.Builder и RSocketStrategies.

**Характеристики**

* Он бинарный.

Вся отправка данных уже оптимизирована по максимуму. Все что надо, это сконвертировать сообщение в байты и потом деконвертировать.

RSocket все сделает за вас.

Если данные большие, RSocket сделает фрейминг сообщения и удостовериться в том, что оно целое и невредимое придет к получателю.

* Он является Multiplexed

Он позволяет только одно соединение для всех логических стримов.

* Bi-directional

Как только мы получили соединение, обе стороны могут как запрашивать данные так и отдавать их.

* Backpressure

RSocket — это имплементация реактивных стримов поверх Network.

Он предоставляет настоящий backpressure.

Например когда вы говорите ему: дай мне 11 элементов, он конвертирует их в фрейм, отправляет на другую сторону (получателю), декодирует его и нотифицирует продьюсера, сколько элементов ему надо отправить (т.е. больше 11 он не сможет отправить никак!).

* Возобновление сессии

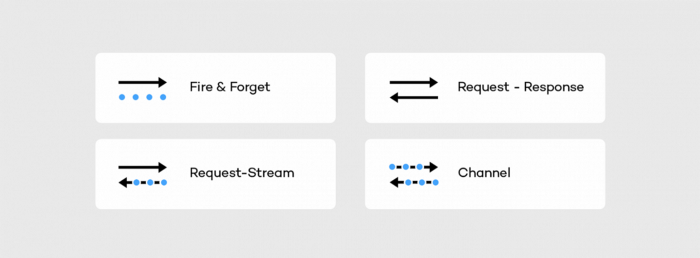
Управление состоянием прозрачно для приложений и хорошо работает в сочетании с backpressure, которое может по возможности остановить производителя и уменьшить количество требуемого состояния. Возобновление — это возможность возобновить работу в случае сбоя (например, восстановление внезапно закрытого соединения).

Это особенно полезно, так как при отправке кадра RESUME, содержащего информацию о последнем принятом кадре, клиент может возобновить соединение и запрашивать только те данные, которые он еще не получил, избегая ненужной нагрузки на сервер и тратя время на попытки восстановить данные, которые уже были получены.

Его следует использовать везде, где это имеет смысл.

**Стратегии использования**

* Request-Response — отправляет сообщение и получает результат
* Request-Stream — отправляет сообщение и получает обратно поток данных
* Channel — отправляет потоки сообщений в обоих направлениях
* Fire-and-Forget — отправляет односторонее сообщение и не ждет ответа



***Fire-and-Forget:***

Fire-and-forget предназначен для передачи данных от отправителя к получателю, в котором отправитель не заботится о результате операции — он отправил запрос и забыл о нем. Такое упрощенное сообщение может быть полезно при отправке уведомлений на мобильную связь например.

В случае операции request stream запрашивающая сторона отправляет ответчику один кадр и ответчик возвращает обратно поток данных. Вместо отправки периодических запросов можно просто подписаться на поток и реагировать на поступающие данные — они будут поступать автоматически, когда они станут доступны.

Благодаря мультиплексированию и поддержке двунаправленной передачи данных, мы можем сделать очень интересную на мой взгляд вещь, используя метод request channel. RSocket может передавать данные от запрашивающей стороны к ответчику и наоборот, используя одно физическое соединение. Такое взаимодействие может быть полезно, когда запрашивающая сторона например обновляет подписку. Без двунаправленного канала клиент должен был бы отменить поток и повторно запросить его с новыми параметрами.

**Основные преимущества**

Можем сделать простой CRUD — нет проблем, т.к. есть простой реквест-респонс.

Можем сделать более advanced CRUD — нет проблем. Например мы пушим какие-то метрики на другой сервер и нам все равно, удлятся они или нет, главное нам отправить их в сеть и забыть о них. Там используется Fire and Forget — отправляем сообщение и скорей забываем про него без ожидания окончания процессинга.

Есть стриминг: можем запросить стрим сервера и сервер отправит данные или мы можем сами про помощи Stream-Stream или Request-Channel Communication пушить данные или пушить стрим данных и сервер нам тоже может отвечать (пушить нам какой-то стрим даных).

Поддержка RSocket практически на всех языках.

Можете использовать или RPC архитектуру или Messaging, не важно.

Вы можете использовать RSocket поверх любого транспортного протокола (WebSocket, HTTP.2, TCP и другие).