RuStore

Реактивный велосипед для SDK

Александр Юдин

Команда:

- Rustore Android Core

Опыт:

- 7+ лет в Android разработке
- 2 года в разработке SDK

План

- Особенности разработки SDK
- Наши стандарты при разработке SDK
- Reactive SDK
  1. Архитектура решения и общие моменты
  2. Single.flatMap
  3. Observable.switchMap
  4. Реализация backpressure
  5. Горячие источники

Проблемы подключения библиотек к проекту

Подключение зависимости в приложение

Activity

```
Observable.create { emitter ->
    emitter.onNext(1)
    emitter.onNext(2)
    emitter.onComplete()
}
.subscribe {
    Log.d("RxTestTag", "subscribe Next $it")
}
```

Decompiled

```
import android.os.Bundle;
import android.util.Log;
import androidx.activity.ComponentActivity;
import androidx.constraintlayout.widget.ConstraintLayout;
import io.reactivex.rxjava3.core.Observable;
import io.reactivex.rxjava3.core.ObservableEmitter;
import io.reactivex.rxjava3.core.ObservableOnSubscribe;
import io.reactivex.rxjava3.functions.Consumer;
import kotlin.Metadata;
import kotlin.jvm.internal.Intrinsics;
```

Что там после компиляции

app-debug.apk
- Код
  - android.support.p000v4
  - androidx
  - com.google
  - io.reactivex.rxjava3
  - kotlin
  - kotlinx.coroutines
  - org
  - ru.rustore

Подключение зависимости в приложение

LibraryClass

```
class LibraryClass {
    operator fun invoke() {
        Observable.create { emitter ->
            emitter.onNext(1)
            emitter.onNext(2)
            emitter.onComplete()
        }
            .subscribe {
                Log.d("RxTestTag", "subscribe Next $it")
            }
    }
}
```

Decompiled

```
import android.util.Log;
import io.reactivex.rxjava3.core.Observable;
import io.reactivex.rxjava3.core.ObservableEmitter;
import io.reactivex.rxjava3.functions.Consumer;
import kotlin.Metadata;
import kotlin.jvm.internal.Intrinsics;
```

Проверяем что попало в AAR

```
mylibrary-debug.aar
▼ Код
  ▼ ru.rustore.mylibrary
    ▼ LibraryClass
        invoke() void
        invoke$lambda$0(ObservableEmitter)
```

Как поставить библиотеку пользователям через maven repository?

AAR/JAR файл + POM файл

содержит код
и логику библиотеки

прописана версия библиотеки
и транзитивные зависимости

Для формирования этих артефактов
используется Gradle plugin maven-publish

POM файл нашей библиотеки

Стандартная зависимость,
если библиотека разрабатывалась
на Kotlin

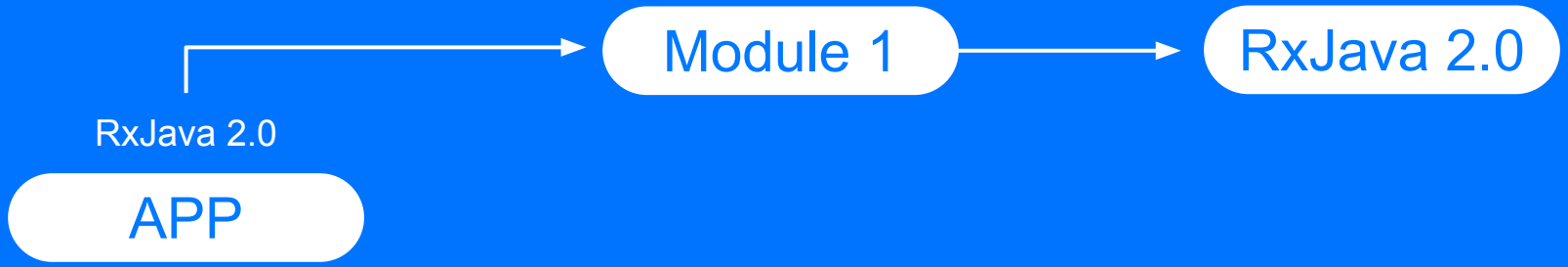
Транзитивная зависимость
на RxJava

```
<groupId>ru.rustore</groupId>
<artifactId>mylibrary</artifactId>
<version>1.0.0</version>
<packaging>aar</packaging>
<dependencies>
  <dependency>
    <groupId>org.jetbrains.kotlin</groupId>
    <artifactId>kotlin-stdlib-jdk8</artifactId>
    <version>1.7.20</version>
    <scope>runtime</scope>
  </dependency>
  <dependency>
    <groupId>io.reactivex.rxjava3</groupId>
    <artifactId>rxkotlin</artifactId>
    <version>3.0.1</version>
    <scope>runtime</scope>
  </dependency>
</dependencies>
```

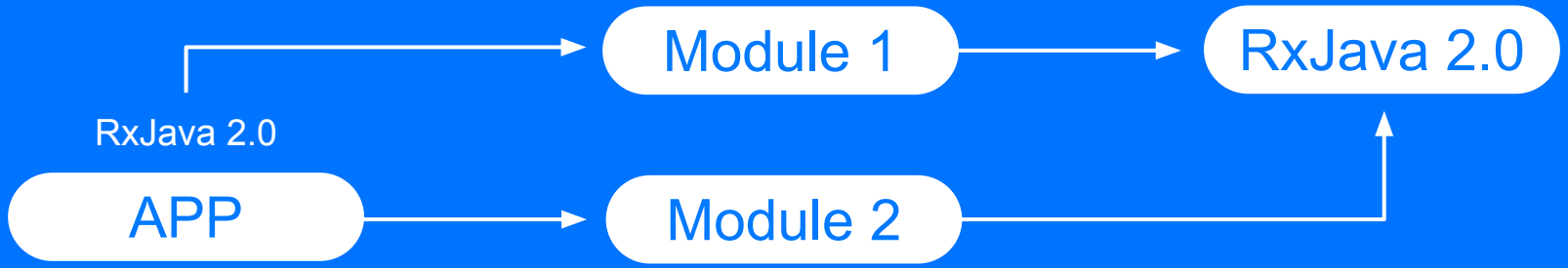
Проблемы транзитивных зависимостей

APP

Проблемы транзитивных зависимостей

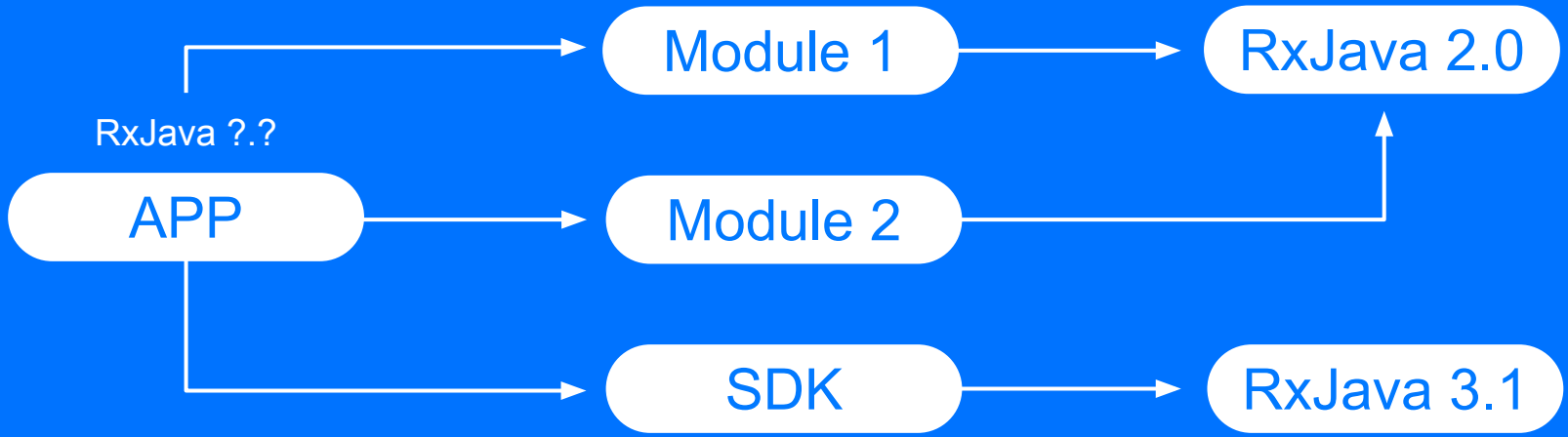


Проблемы транзитивных зависимостей



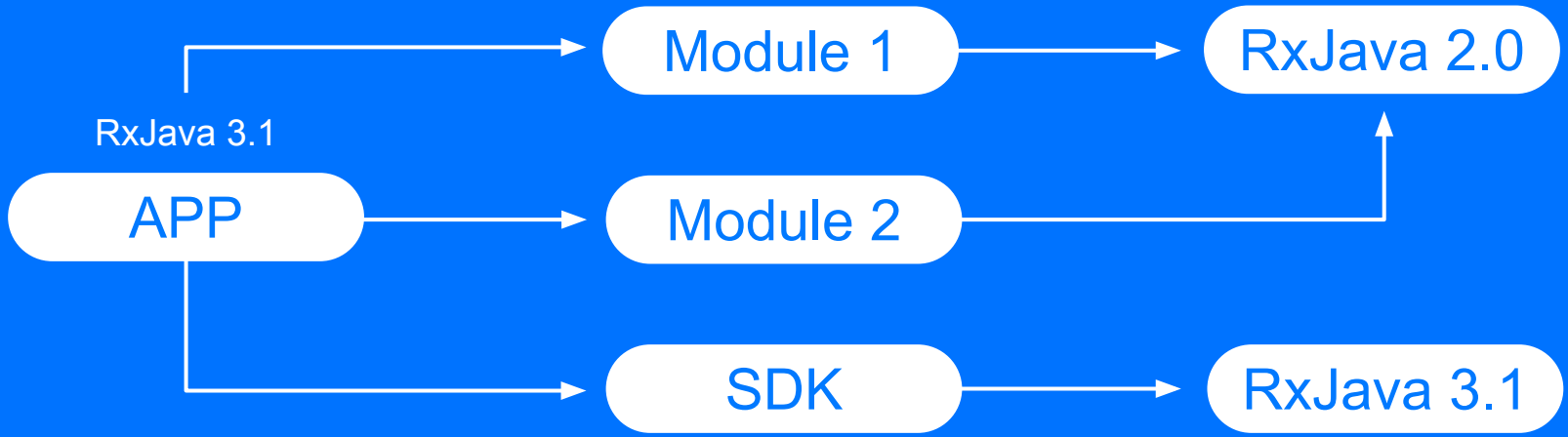
Проблемы транзитивных зависимостей

Чем больше зависимостей используется для разработки SDK,
тем больше проблем можно принести потребителям:



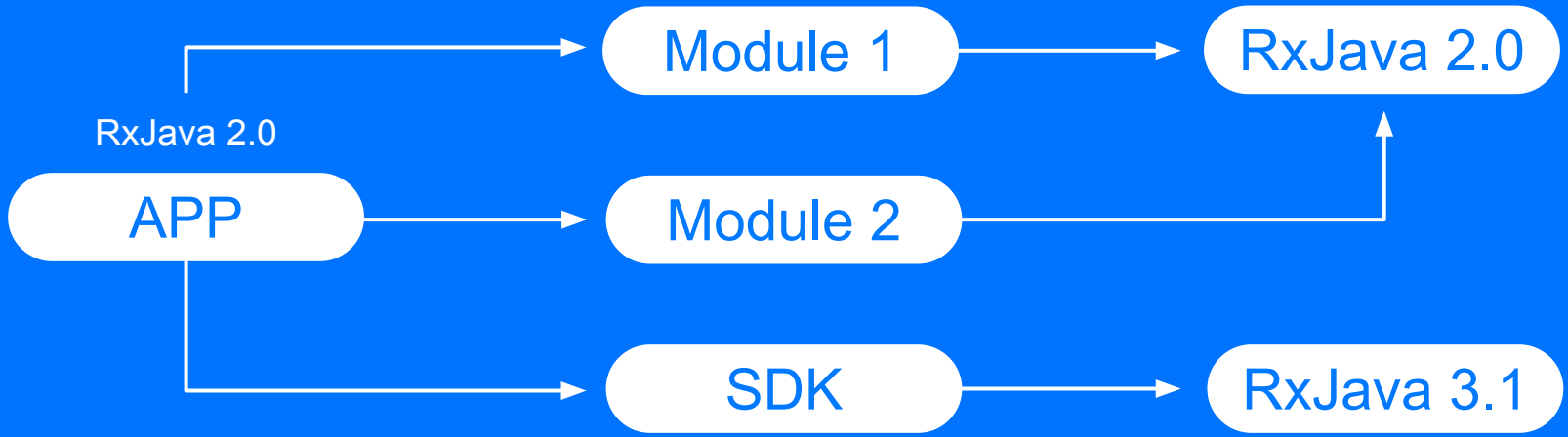
Проблемы транзитивных зависимостей

Стратегия разрешения: Default



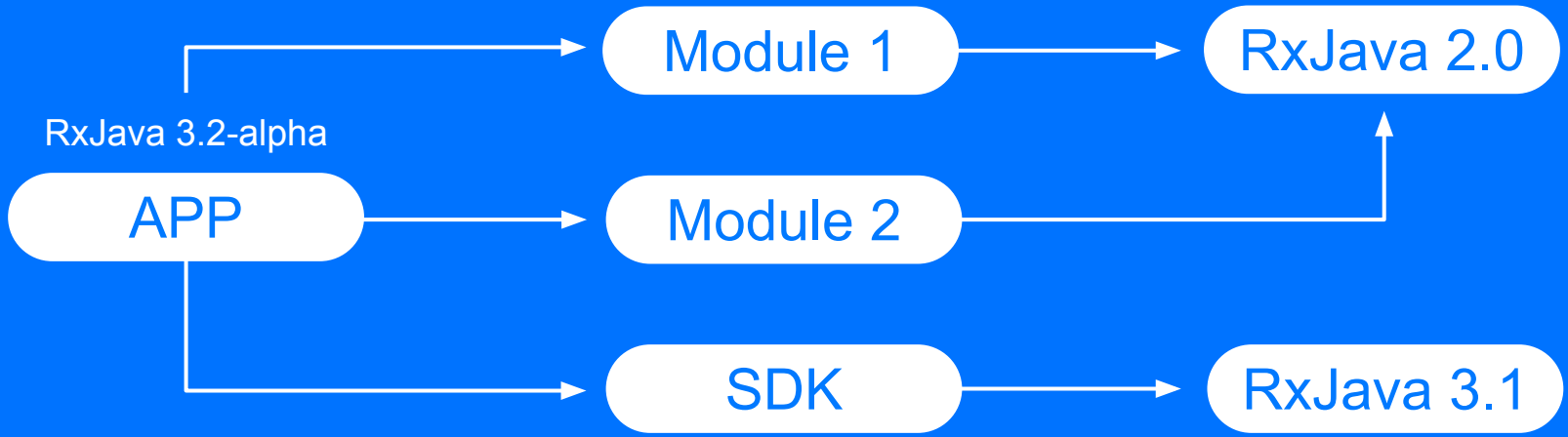
Проблемы транзитивных зависимостей

Стратегия разрешения: Force 2.0



Проблемы транзитивных зависимостей

Стратегия разрешения: Latest



Наши стандарты разработки SDK

При разработке SDK нужно следовать следующим правилам:

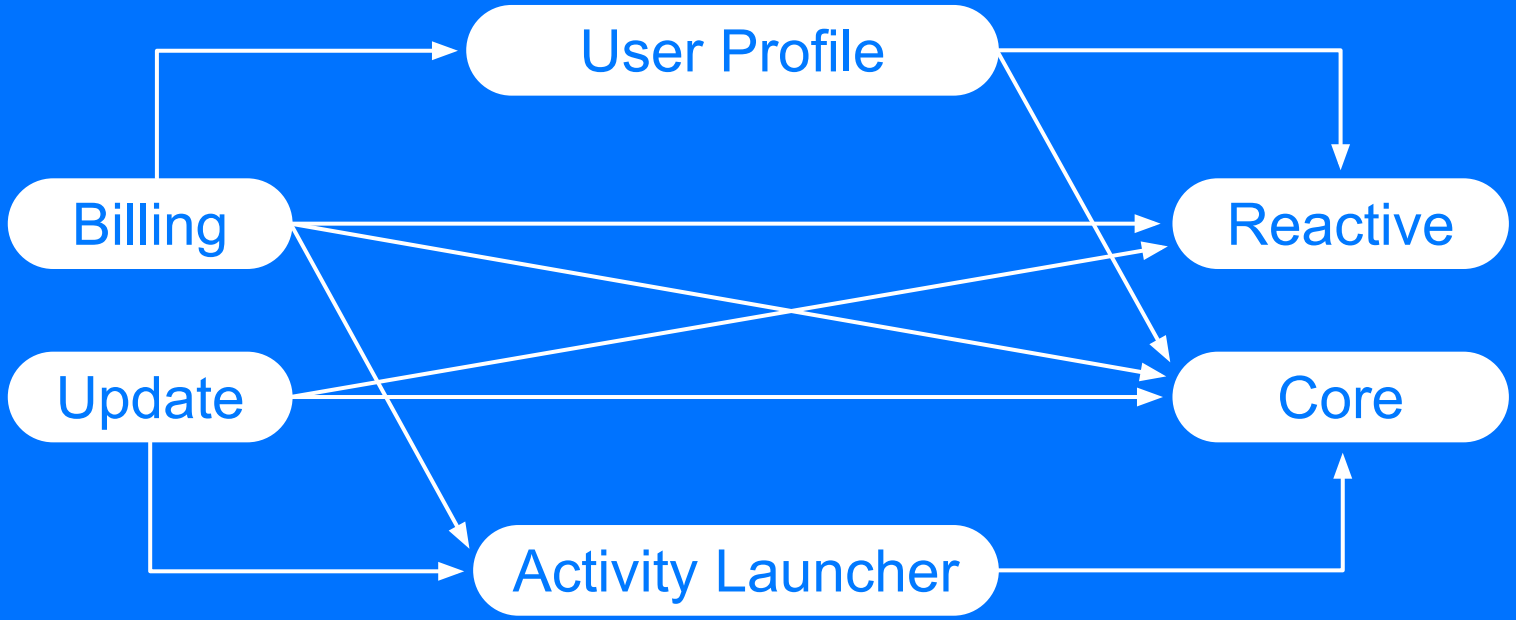
1. Использовать общий публичный контракт
2. Использовать модульный подход
3. Избегать использования фреймворков
4. При разработке ~~изобретать велосипеды:)~~ использовать собственные подходы

Общий контракт

Все API методы SDK должны предоставлять общий контракт.

```
public class Task<T> private constructor() {

    @Throws
    public fun await(): T

    public fun addOnSuccessListener(listener: OnSuccessListener<T>): Task<T>

    public fun addOnFailureListener(listener: OnFailureListener): Task<T>
}
```

Модульный подход



Выметаем фреймворки

Делаем все сами

- Для DI мы стали использовать обычный ManualDI —
обычный Kotlin класс с val полями.

- База данных чистый SQLite —
давно забытый, но достаточно удобный фреймворк.

- Ручные сериализаторы
в JSON и обратно.

- HttpUrlConnection как сетевой слой —
пока используем в местах напрямую,
не оборачивая в какой-то общий модуль.

Асинхронное программирование

Начали мы этот путь с использвания Task api, про который говорил ранее.

Для выполнения задач создали Executor

```
public interface Executor {

    public fun <R> execute(command: () -> R): Future<R>
}
```

Асинхронное программирование

```
public fun <R> Executor.executeTask(
    block: () -> R,
): Task<R> {
    val (task, resultProvider) = Task.create<R>()
    val future = execute {
        try {
            resultProvider.setTaskSuccessResult(block())
        } catch (e: Throwable) {
            resultProvider.setTaskErrorResult(e)
        }
    }
    return task
}
```

Минусы подхода

1. Отсутствие цепочек задач. Если вторую задачу нужно было запускать с результатом первой, то возникали неудобные конструкции.
2. Обработка ошибок параллельных задач неудобна и склонна к ошибкам.
3. Отмена задач усложнена
4. Отсутствие аналога горячих источников. (SharedFlow/PublishSubject)

А может Shading?

ДОКЛАД

Как сделать библиотеку, чтобы ей пользовались

Не будем рассказывать как продать библиотеку. Расскажем как сделать так, чтобы у пользователей был шанс ее подключить.

Игорь Рыбаков
VK / RuStore

Евгений Ковешников
VK / RuStore

RU 

Инфраструктура

Свой фреймворк, свои правила

- Реализуем фреймворк как отдельный модуль и публикуем вместе с релизами SDK. Классы SDK будут выступать в роли контракта для связи между публикуемыми модулями.
- Сделать фреймворк без оверинжениринга, следуем принципам “KISS”
- Сохранение знакомого API обеспечит максимально удобный и быстрый переход.


Reactive
SDK

Выбор пути

1. Реактивный подход
2. Контракт на понятном нейминге из RxJava
3. Не перегружать код наследованием
4. Сразу решить проблему с backpreassure,
   не заводить лишние сущности типа Observable и Flowable
5. Больше Kotlin лайк конструкций для разгрузки кода

Базовый пример

- Один источник
- Нет преобразований
- Потребитель

Базовый пример

```
interface Source<T> {

    fun doWork(resultHandler: ResultHandler<T>)
}


 interface ResultHandler<T> {

     fun workStarted()

     fun onFail(t: Throwable)

     fun onSuccess(value: T)
 }
```

Базовый пример

```
class ExampleSource : Source<Int> {
    override fun doWork(resultHandler: ResultHandler<Int>) {
        resultHandler.workStarted()

        try {
            val result = 1 + 2
            resultHandler.onSuccess(result)
        } catch (t: Throwable) {
            resultHandler.onFail(t)
        }
    }
}
```

Базовый пример

```
class ExampleResultHandler : ResultHandler<Int> {
    override fun workStarted() {
        println("workStarted")
    }

    override fun onFail(t: Throwable) {
        println("onFail $t")
    }

    override fun onSuccess(value: Int) {
        println("onSuccess $value")
    }
}
```

Базовый пример

```
val source = ExampleSource()
val resultHandler = ExampleResultHandler()

source.doWork(resultHandler)
```

```
workStarted
onSuccess 3
```

Построение цепочки

- Один источник
- Преобразование типов
- Потребитель

Построение цепочки

```
public abstract class Single<T> {
    public abstract fun subscribe(downstream: SingleObserver<T>)
}


public interface SingleObserver<T> {

    public fun onSubscribe(d: Disposable)

    public fun onSuccess(item: T)

    public fun onError(e: Throwable)
}
```

Построение цепочки

```
val source: Single<String> = Single.create { emitter ->
        emitter.success(1)
    }
        .map { value ->
            (value * 2).toString()
        }

    source.subscribe { result ->
        println(result)
    }
```

Построение цепочки

```
val source: Single<String> = Single.create { emitter ->
        emitter.success(1)
    }
        .map { value ->
            (value * 2).toString()
        }

    source.subscribe { result ->
        println(result)
    }
```

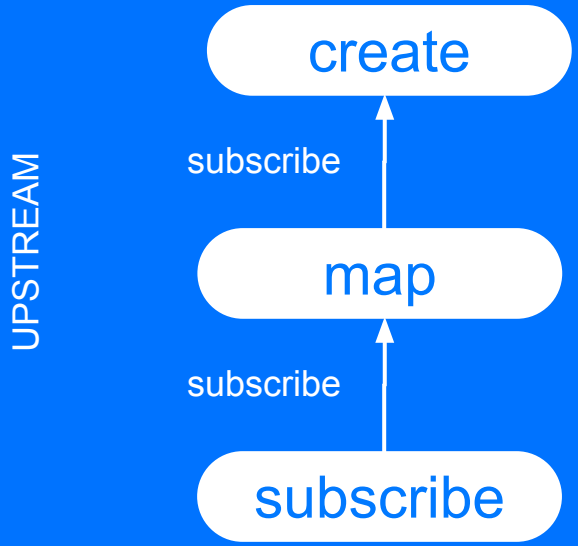
```
private class SingleMap<T, R>(
    private val upstream: Single<T>,
    private val mapper: (T) -> R,
) : Single<R>() {
```

Построение цепочки

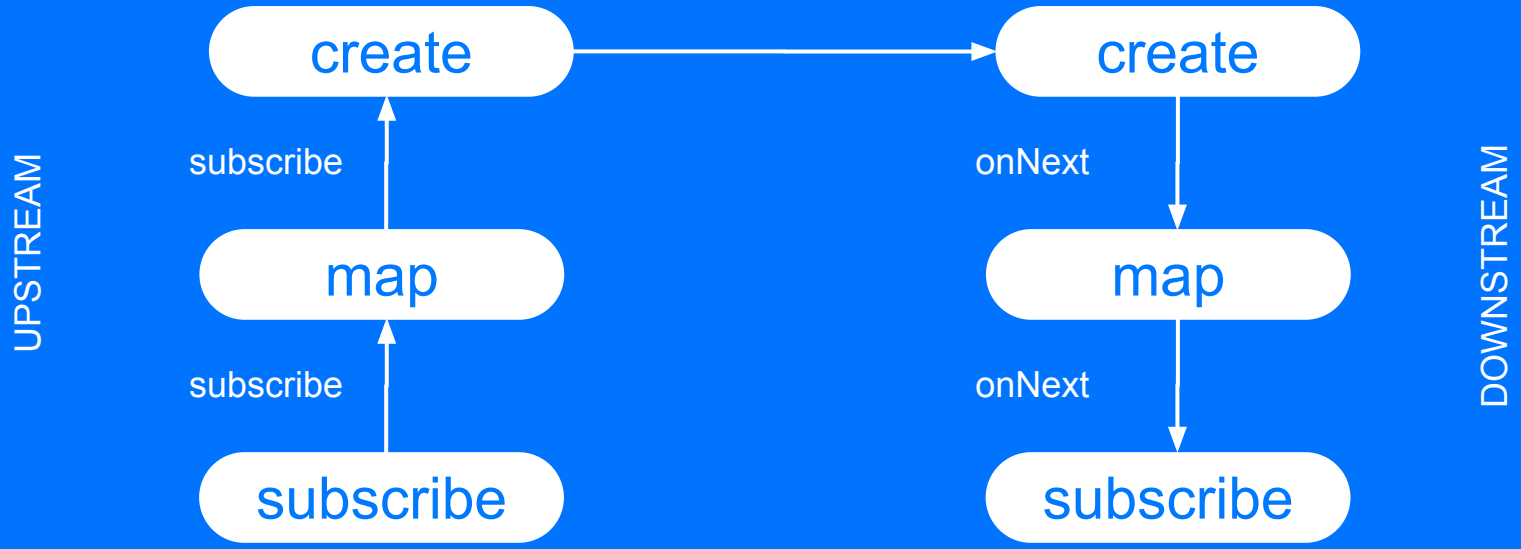
```
val source: Single<String> = Single.create { emitter ->
        emitter.success(1)
    }
        .map { value ->
            (value * 2).toString()
        }

    source.subscribe { result ->
        println(result)
    }
```

```
public fun <T> Single<T>.subscribe(
    onError: (Throwable) -> Unit = errorStub,
    onSuccess: (T) -> Unit,
): Disposable {
    val observer = SingleSubscribeObserver(onError, onSuccess)
    subscribe(observer)
    return observer
}
```

Upstream-downstream



Upstream-downstream



Построение цепочки

```
private class SingleMap<T, R>(
    private val upstream: Single<T>,
    private val mapper: (T) -> R,
) : Single<R>() {
```

```
override fun subscribe(downstream: SingleObserver<R>) {
    val wrappedObserver = object : SingleObserver<T> {
        override fun onSubscribe(d: Disposable) {
            downstream.onSubscribe(d)
        }
        override fun onError(e: Throwable) {
            downstream.onError(e)
        }
        override fun onSuccess(item: T) {
            runCatching { mapper(item) }
                .onSuccess {
                    downstream.onSuccess(it)
                }
                .onFailure {
                    downstream.onError(it)
                }
        }
    }
    upstream.subscribe(wrappedObserver)
}
```

Построение цепочки

```
private class SingleMap<T, R>(
    private val upstream: Single<T>,
    private val mapper: (T) -> R,
) : Single<R>() {
```

```
override fun subscribe(downstream: SingleObserver<R>) {
    val wrappedObserver = object : SingleObserver<T> {
        override fun onSubscribe(d: Disposable) {
            downstream.onSubscribe(d)
        }
        override fun onError(e: Throwable) {
            downstream.onError(e)
        }
        override fun onSuccess(item: T) {
            runCatching { mapper(item) }
                .onSuccess {
                    downstream.onSuccess(it)
                }
                .onFailure {
                    downstream.onError(it)
                }
        }
    }
    upstream.subscribe(wrappedObserver)
}
```

Построение цепочки

```
private class SingleMap<T, R>(
    private val upstream: Single<T>,
    private val mapper: (T) -> R,
) : Single<R>() {
```

```
override fun subscribe(downstream: SingleObserver<R>) {
    val wrappedObserver = object : SingleObserver<T> {
        override fun onSubscribe(d: Disposable) {
            downstream.onSubscribe(d)
        }
        override fun onError(e: Throwable) {
            downstream.onError(e)
        }
        override fun onSuccess(item: T) {
            runCatching { mapper(item) }
                .onSuccess {
                    downstream.onSuccess(it)
                }
                .onFailure {
                    downstream.onError(it)
                }
        }
    }
    upstream.subscribe(wrappedObserver)
}
```

Построение цепочки

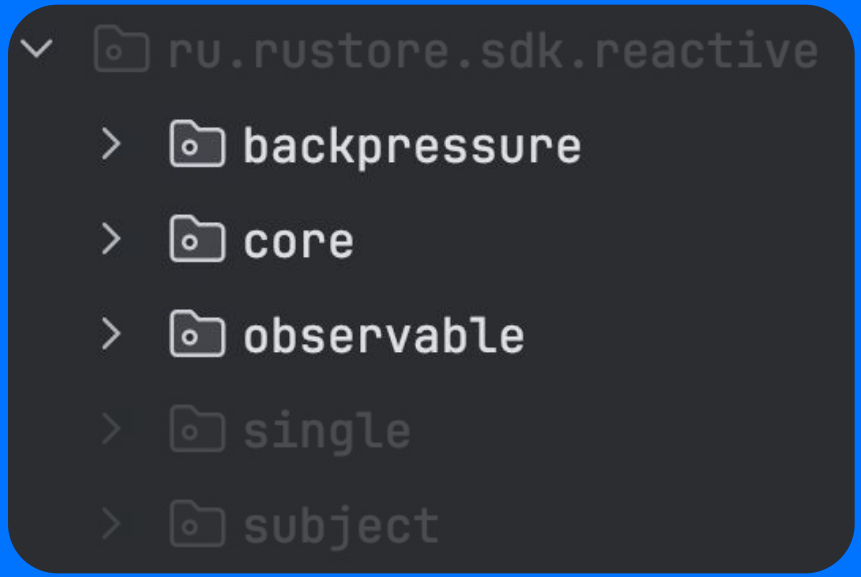
```
private class SingleMap<T, R>(
    private val upstream: Single<T>,
    private val mapper: (T) -> R,
) : Single<R>() {

    override fun subscribe(downstream: SingleObserver<R>) {
        val wrappedObserver = object : SingleObserver<T> {
            override fun onSubscribe(d: Disposable) {
                downstream.onSubscribe(d)
            }
            override fun onError(e: Throwable) {
                downstream.onError(e)
            }
            override fun onSuccess(item: T) {
                runCatching { mapper(item) }
                    .onSuccess {
                        downstream.onSuccess(it)
                    }
                    .onFailure {
                        downstream.onError(it)
                    }
            }
        }
        upstream.subscribe(wrappedObserver)
    }
```

Архитектура

Backpressure — набор классов для управления процессом отправки значений в многопоточной среде.

Core — общие классы.

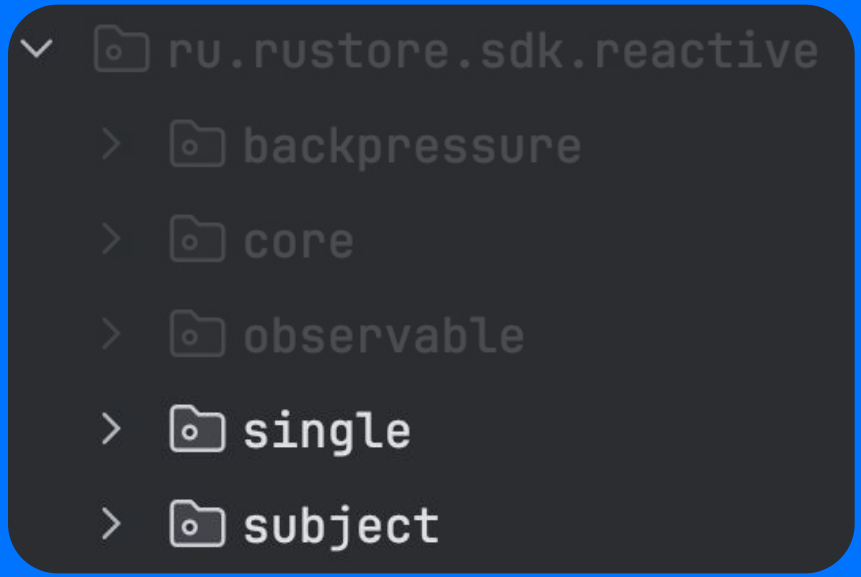
Observable — Базовый класс и набор операторов для потока данных, который может испускать N значений. Из коробки использует поддержку backpressure.



Архитектура

Single — Базовый класс и набор операторов для одного значения.

Subject — Аналог SharedFlow и StateFlow



Сложности

1. Как в одном disposable объединить всю цепочку операторов
2. Как правильно строить substream (flatMap)
3. flatMap **switchMap** для Observable
4. Как побороть backpressure
5. Горячий поток с:
    - Множественной подпиской
    - Повтором значений
    - Потокобезопасен

Сложность №1 — Disposable

```
public fun <T> Single<T>.subscribe(
    onError: (Throwable) -> Unit = errorStub,
    onSuccess: (T) -> Unit,
): Disposable {
    val observer = SingleSubscribeObserver(onError, onSuccess)
    subscribe(observer)
    return observer
}
```

Если посмотреть внимательней, то возвращаемый тип метода — Disposable.
Ну у обсервера **Disposable присутствует только в колбеке onSubscribe**

```
public interface SingleObserver<T> {

    public fun onSubscribe(d: Disposable)
```

Disposable

```
override fun subscribe(downstream: SingleObserver<T>) {
    val wrappedObserver = object : SingleObserver<T>, Disposable {
        private val disposed = AtomicBoolean()
        private val upstreamDisposable = AtomicReference<Disposable?>(null)

        override fun onSubscribe(d: Disposable) {
            upstreamDisposable.compareAndSet(null, d)
            if (isDisposed()) {
                upstreamDisposable.getAndSet(null)?.dispose()
            }
            downstream.onSubscribe(this)
        }

        override fun isDisposed(): Boolean =
            disposed.get()

        override fun dispose() {
            if (disposed.compareAndSet(false, true)) {
                upstreamDisposable.getAndSet(null)?.dispose()
            }
        }
```

Disposable

```
override fun subscribe(downstream: SingleObserver<T>) {
    val wrappedObserver = object : SingleObserver<T>, Disposable {
        private val disposed = AtomicBoolean()
        private val upstreamDisposable = AtomicReference<Disposable?>(null)

        override fun onSubscribe(d: Disposable) {
            upstreamDisposable.compareAndSet(null, d)
            if (isDisposed()) {
                upstreamDisposable.getAndSet(null)?.dispose()
            }
            downstream.onSubscribe(this)
        }

        override fun isDisposed(): Boolean =
            disposed.get()

        override fun dispose() {
            if (disposed.compareAndSet(false, true)) {
                upstreamDisposable.getAndSet(null)?.dispose()
            }
        }
```

Сложность №2 — Single.FlatMap

Требования:

1. Формировать дополнительную цепочку событий (substream) не в upstream, а в downstream
2. Поддержать отмену substream при отмене всей цепочки

Single.FlatMap

FlatMap

```
private class SingleFlatMap<T, R>(
    private val upstream: Single<T>,
    private val mapper: (T) -> Single<R>,
) : Single<R>() {
```

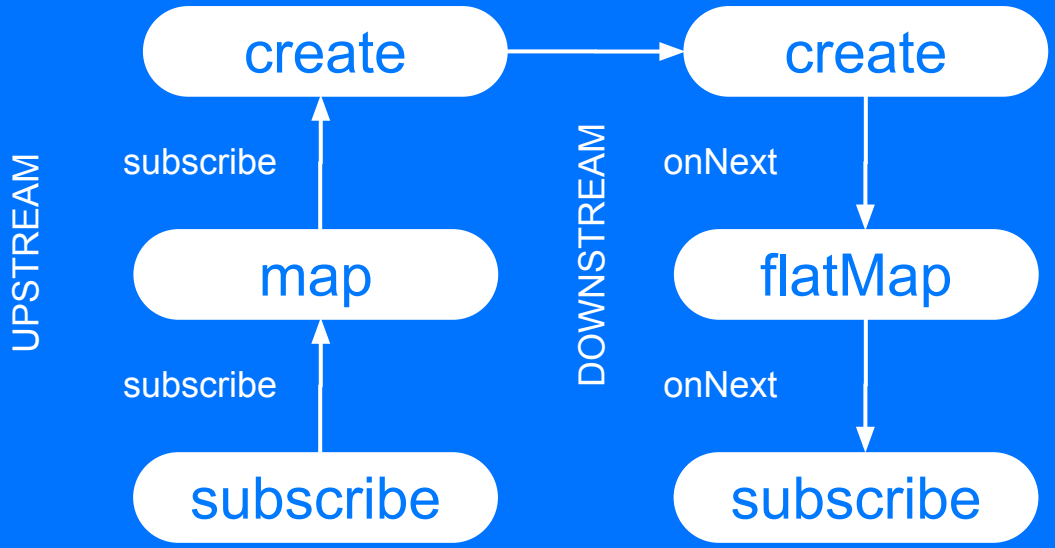
Map

```
private class SingleMap<T, R>(
    private val upstream: Single<T>,
    private val mapper: (T) -> R,
) : Single<R>() {
```

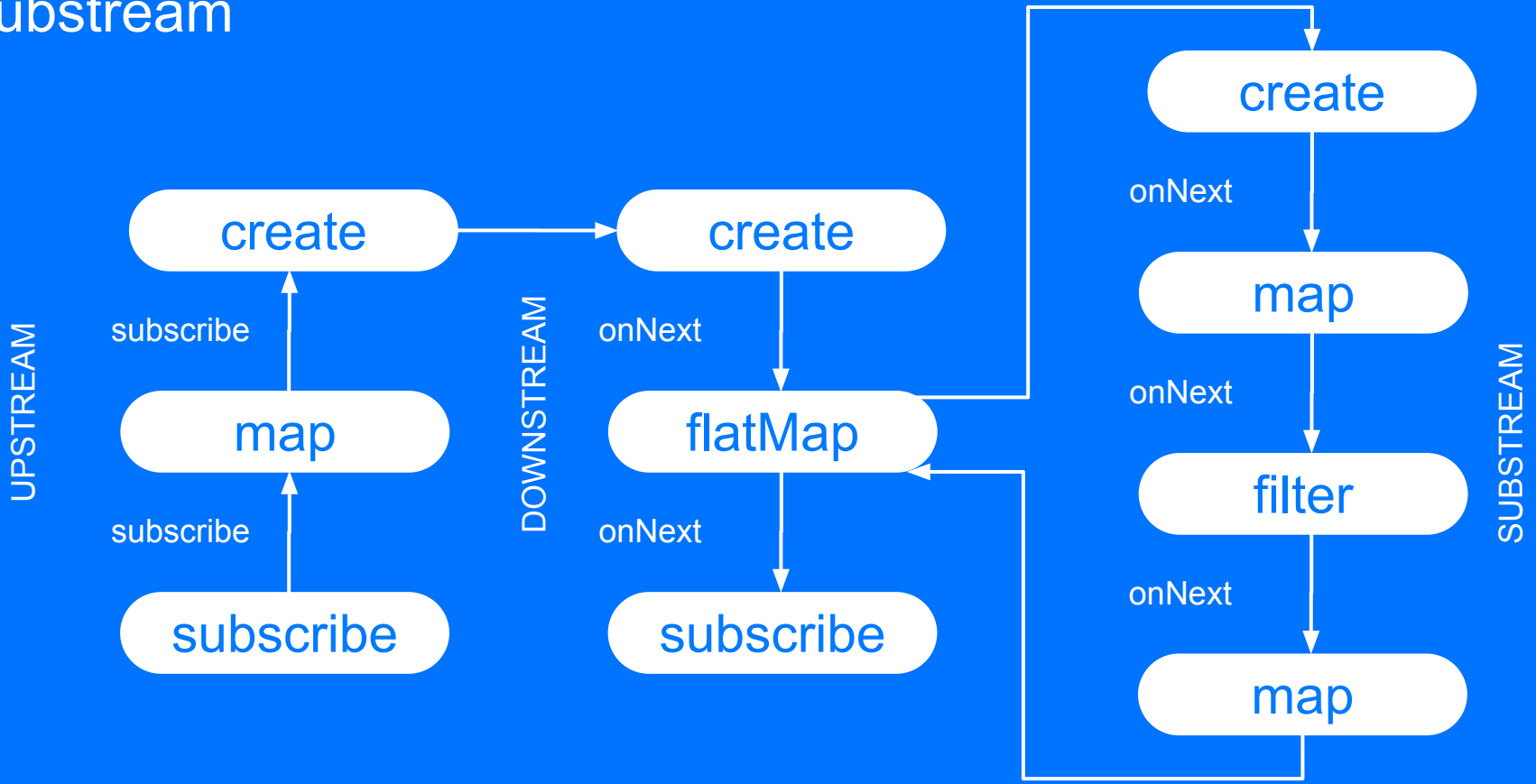
Single.FlatMap

```
Single.from { 1 }
    .flatMap { upstreamValue ->

        Single.from { upstreamValue * 2 }
            .map { value ->
                value.toString()
            }

    }
    .subscribe { result ->
        println(result)
    }
```

Substream


create
create
UPSTREAM
subscribe
map
subscribe
subscribe
DOWNSTREAM
onNext
flatMap
onNext
subscribe

Substream


UPSTREAM
create
subscribe
map
subscribe
subscribe
DOWNSTREAM
create
onNext
flatMap
onNext
subscribe
SUBSTREAM
create
onNext
map
onNext
filter
onNext
map

Single.FlatMap

```
private val disposed = AtomicBoolean()
private val upstreamDisposable = AtomicReference<Disposable?>(null)
private val substreamDisposable = AtomicReference<Disposable?>(null)

override fun onSubscribe(d: Disposable) {
    upstreamDisposable.compareAndSet(null, d)
    if (disposed.get()) {
        upstreamDisposable.getAndSet(null)?.dispose()
        substreamDisposable.getAndSet(null)?.dispose()
    }
    downstream.onSubscribe(this)
}
override fun isDisposed(): Boolean =
    disposed.get()
override fun dispose() {
    if (disposed.compareAndSet(false, true)) {
        upstreamDisposable.getAndSet(null)?.dispose()
        substreamDisposable.getAndSet(null)?.dispose()
    }
}
```

Single.FlatMap

```
override fun subscribe(downstream: SingleObserver<R>) {
    val wrappedObserver = object : SingleObserver<T>, Disposable {
        private val substreamDisposable = AtomicReference<Disposable?>(null)

        override fun onSuccess(item: T) {
            if (disposed.compareAndSet(false, true)) {
                val flatMapSource = SingleFlatMapSubscriber()

                substreamDisposable.set(flatMapSource)

                flatMapSource.subscribe(downstream, item)
            }
        }
```

Single.FlatMap

```
private inner class SingleFlatMapSubscriber : Disposable {
    private val disposed = AtomicBoolean()
    private val upstreamDisposable = AtomicReference<Disposable>(null)

    fun subscribe(downstream: SingleObserver<R>, item: T) {
        val singleFlatMapObserver = object : SingleObserver<R> {...}
        if (!isDisposed()) {
            runCatching { mapper(item) }
                .onSuccess { newSingle ->
                    if (!isDisposed()) {
                        newSingle.subscribe(singleFlatMapObserver)
                    }
                }
                .onFailure { error ->
                    if (!isDisposed()) {
                        singleFlatMapObserver.onError(error)
                    }
                }
        }
    }
```

Single.FlatMap

```
private inner class SingleFlatMapSubscriber : Disposable {
    private val disposed = AtomicBoolean()
    private val upstreamDisposable = AtomicReference<Disposable>(null)

    fun subscribe(downstream: SingleObserver<R>, item: T) {
        val singleFlatMapObserver = object : SingleObserver<R> {...}
        if (!isDisposed()) {
            runCatching { mapper(item) }
                .onSuccess { newSingle ->
                    if (!isDisposed()) {
                        newSingle.subscribe(singleFlatMapObserver)
                    }
                }
                .onFailure { error ->
                    if (!isDisposed()) {
                        singleFlatMapObserver.onError(error)
                    }
                }
        }
    }
```

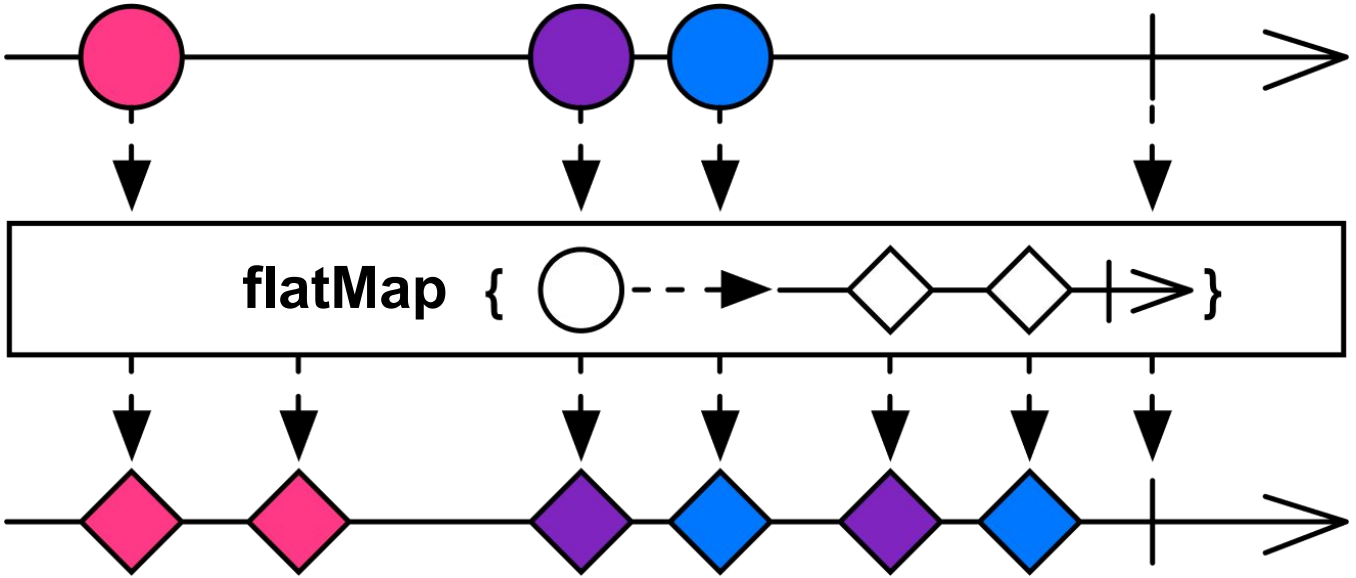
```
private class SingleFlatMap<T, R>(
    private val upstream: Single<T>,
    private val mapper: (T) -> Single<R>,
) : Single<R>() {
```

Single.FlatMap

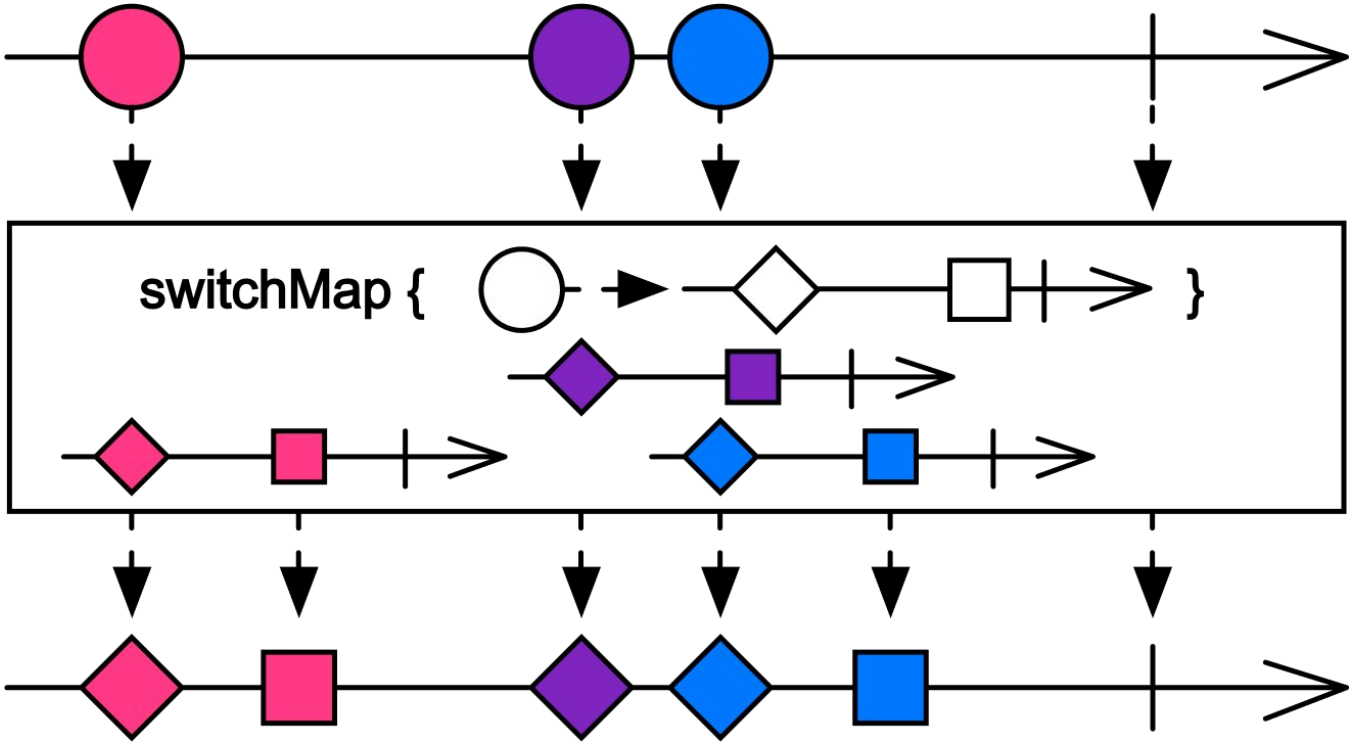
Итоги:

- Поняли как правильно работать с substream
- Включили substream в цепочку disposable
- Выработали базовый подход и можно переходить к более сложным операторам

Сложности №3 — Observable.switchMap



Observable.switchMap



Observable.switchMap

```
Observable.create { emitter ->
    repeat(3) { value ->
        emitter.onNext(value)
        Thread.sleep(10)
    }
}
    .switchMap { upstreamValue ->
        Observable.create { emitter ->
            repeat(3) {
                emitter.onNext(upstreamValue * 10)
                Thread.sleep(8)
            }
        }
            .subscribeOn(Dispatchers.io)
    }
    .subscribe { println(it) }
```

Результат:

0
0
10
10
20
20
20

Observable.switchMap

```
Observable.create { emitter ->
    repeat(3) { value ->
        emitter.onNext(value)
        Thread.sleep(10)
    }
}
    .switchMap { upstreamValue ->
        Observable.create { emitter ->
            repeat(3) {
                emitter.onNext(upstreamValue * 10)
                Thread.sleep(8)
            }
        }
            .subscribeOn(Dispatchers.io)
    }
    .subscribe { println(it) }
```

Результат:

0
0
10
10
20
20
20

Observable.switchMap

```
Observable.create { emitter ->
    repeat(3) { value ->
        emitter.onNext(value)
        Thread.sleep(10)
    }
}
    .switchMap { upstreamValue ->
        Observable.create { emitter ->
            repeat(3) {
                emitter.onNext(upstreamValue * 10)
                Thread.sleep(8)
            }
        }
            .subscribeOn(Dispatchers.io)
    }
    .subscribe { println(it) }
```

Результат:

0
0
10
10
20
20
20

Observable.switchMap

```
Observable.create { emitter ->
    repeat(3) { value ->
        emitter.onNext(value)
        Thread.sleep(10)
    }
    emitter.onComplete()
}
    .switchMap { upstreamValue ->
        Observable.create { emitter ->
            repeat(3) {
                emitter.onNext(upstreamValue * 10)
                Thread.sleep(8)
            }
            emitter.onComplete()
        }
            .subscribeOn(Dispatchers.io)
    }
    .subscribe(onNext = { println(it) }, onComplete = { println("Complete") })
```

Вывод цепочки:

0
0
10
10
20
20
20
Complete

Observable.switchMap

```
Observable.create { emitter ->
    repeat(3) { value ->
        emitter.onNext(value)
        Thread.sleep(10)
    }
    emitter.onComplete()
}
    .switchMap { upstreamValue ->
        Observable.create { emitter ->
            emitter.onNext(upstreamValue * 10)
            emitter.onComplete()
        }
            .subscribeOn(Dispatchers.io)
    }
    .subscribe(onNext = { println(it) }, onComplete = { println("Complete")

})
```

Вывод цепочки:

0
10
20
Complete

Observable.switchMap

```
// OnComplete downstream будет
// вызван в случае уменьшения счетчика до 0
private val completionsLeftCount = AtomicInteger(1)
```

Счетчик в основном Observer

```
override fun onComplete() {
    completionsLeftCount.decrementAndGet()
    innerOnComplete()
}
```

Метод onComlete основного downstream.

```
private fun innerOnComplete() {
    if (completionsLeftCount.get() == 0 && switchMapDisposed.compareAndSet(false, true)) {
        emitProcessor.complete()
        emitProcessor.drain()
    }
}
```

Observable.switchMap

```
private val actualSubstream = AtomicReference<Disposable?>(null)

override fun onNext(item: T) {
    if (isDisposed()) return

    completionsLeftCount.incrementAndGet()

    val substream = SubstreamSubscriber()

    val substreamToDispose = actualSubstream.getAndSet(substream)
    substreamToDispose?.dispose()

    if (!isDisposed()) {
        substream.subscribe(item)
    }
}
```

Observable.switchMap

```
private val actualSubstream = AtomicReference<Disposable?>(null)

override fun onNext(item: T) {
    if (isDisposed()) return

    completionsLeftCount.incrementAndGet()

    val substream = SubstreamSubscriber()

    val substreamToDispose = actualSubstream.getAndSet(substream)
    substreamToDispose?.dispose()

    if (!isDisposed()) {
        substream.subscribe(item)
    }
}
```

Observable.switchMap

```
private val actualSubstream = AtomicReference<Disposable?>(null)

override fun onNext(item: T) {
    if (isDisposed()) return

    completionsLeftCount.incrementAndGet()

    val substream = SubstreamSubscriber()

    val substreamToDispose = actualSubstream.getAndSet(substream)
    substreamToDispose?.dispose()

    if (!isDisposed()) {
        substream.subscribe(item)
    }
}
```

Observable.switchMap

Класс SubstreamSubscriber для SwitchMap

```
override fun dispose() {
    if (substreamDisposed.compareAndSet(false, true)) {
        completionsLeftCount.decrementAndGet()
        substreamDisposable.getAndSet(null)?.dispose()
    }
}
```

Observable.switchMap

Класс SubstreamSubscriber для SwitchMap

```
onComplete = {
    if (substreamDisposed.compareAndSet(false, true)) {
        completionsLeftCount.decrementAndGet()
        innerOnComplete()
    }
},
```

Сам метод onComplete очень похож на основной метод downstream.

Observable.switchMap

Итоги:

Увеличение счетчика onComplete

- Создание main stream
- Создание substream

Уменьшение счетчика onComplete

- Вызов OnComplete main stream
- Вызов OnComplete substream
- Вызов Dispose у substream

Сложность №4 — Backpressure

Быстрый источник 1..N + Медленный оператор + Медленный потребитель

=

OutOfMemoryException

Backpressure

Требования:

1. Создание observable поддерживает backpressure из коробки
2. Закладываем backpressure во все “асинхронные” операторы
3. Заложить функционал “выравнивания”, чтобы данные в onNext шли строго по порядку
4. Добавить возможность смены политики борьбы с backpressure на любом этапе

Backpressure

```
Observable.create { emitter ->
    Thread {
        repeat(10) {
            emitter.onNext(it)
        }
    }.start()
    Thread {
        repeat(10) {
            emitter.onNext(it + 10)
        }
    }.start()
}
    .observeOn(Dispatchers.io)
    .map { upstreamValue ->
      if (upstreamValue == 2) Thread.sleep(100)
        upstreamValue
    }
    .subscribe(onNext = { println(it) }, onComplete = { println("Complete") })
```

Backpressure

```
internal class ObservableCreate<T>(
    private val backpressureStrategy: BackpressureStrategy,
    private val source: (ObservableEmitter<T>) -> Unit,
) : Observable<T>() {
```

```
private class ObservableSwitchMap<T, R>(
    private val upstream: Observable<T>,
    private val mapper: (T) -> Observable<R>,
    private val backpressureStrategy: BackpressureStrategy,
) : Observable<R>() {
```

Backpressure

```
public sealed interface BackpressureStrategy {

    public class BufferDropLast(public val bufferSize: Int) : BackpressureStrategy

    public class BufferDropOldest(public val bufferSize: Int) : BackpressureStrategy
}
```

Backpressure

```
public class ObservableEmitter<T> internal constructor(
    private val emitProcessor: BufferEmitProcessor<T>,
) {
    public fun onNext(item: T) {
        emitProcessor.emit(item)
        emitProcessor.drain()
    }
    public fun onComplete() {
        emitProcessor.complete()
        emitProcessor.drain()
    }
    public fun onError(e: Throwable) {
        emitProcessor.error(e)
        emitProcessor.drain()
    }
}
```

Backpressure

```
private val elements = ArrayDeque<BufferItemType>()


internal sealed class BufferItemType {

    object Complete : BufferItemType()

    class Error(val e: Throwable) : BufferItemType()

    class Item<T>(val item: T) : BufferItemType()
}
```

Backpressure

```
fun emit(item: T) {
    synchronized(monitor) {
        if (streamDone) return

        if (buffer.size() >= bufferSize) {
            onOverflow(buffer, BufferItemType.Item(item))
        } else {
            buffer.offer(BufferItemType.Item(item))
        }
    }
}
```

Backpressure

```
fun emit(item: T) {
    synchronized(monitor) {
        if (streamDone) return

        if (buffer.size() >= bufferSize) {
            onOverflow(buffer, BufferItemType.Item(item))
        } else {
            buffer.offer(BufferItemType.Item(item))
        }
    }
}
```

Backpressure

```
fun complete() {
    synchronized(monitor) {
        if (streamDone) return

        streamDone = true
        buffer.offer(BufferItemType.Complete)
    }
}
```

```
fun error(e: Throwable) {
    synchronized(monitor) {
        if (streamDone) return

        streamDone = true
        buffer.clear()
        buffer.offer(BufferItemType.Error(e))
    }
}
```

Backpressure

```
fun complete() {
    synchronized(monitor) {
        if (streamDone) return

        streamDone = true
        buffer.offer(BufferItemType.Complete)
    }
}
```

```
fun error(e: Throwable) {
    synchronized(monitor) {
        if (streamDone) return

        streamDone = true
        buffer.clear()
        buffer.offer(BufferItemType.Error(e))
    }
}
```

Backpressure

```
fun drain() {
    synchronized(monitor) {
        if (isDrainActive) return

        isDrainActive = true
    }

    if (dispatcher != null) {
        dispatcher.execute { loop() }
    } else {
        loop()
    }
}
```

Backpressure

```
fun drain() {
    synchronized(monitor) {
        if (isDrainActive) return

        isDrainActive = true
    }

    if (dispatcher != null) {
        dispatcher.execute { loop() }
    } else {
        loop()
    }
}
```

Backpressure

```
fun drain() {
    synchronized(monitor) {
        if (isDrainActive) return

        isDrainActive = true
    }

    if (dispatcher != null) {
        dispatcher.execute { loop() }
    } else {
        loop()
    }
}
```

Backpressure

```
while (true) {
    val value = synchronized(monitor) {
        val nextValue = buffer.popFirstOrNull()
        if (nextValue == null) {
            isDrainActive = false
            return
        }
        nextValue
    }
    when (value) {
        is BufferItemType.Item<*> -> downStream.onNext(value.item as T)
        is BufferItemType.Error -> downStream.onError(value.e)
        BufferItemType.Complete -> downStream.onComplete()
    }
}
```

Backpressure

```
while (true) {
    val value = synchronized(monitor) {
        val nextValue = buffer.popFirstOrNull()
        if (nextValue == null) {
            isDrainActive = false
            return
        }
        nextValue
    }
    when (value) {
        is BufferItemType.Item<*> -> downStream.onNext(value.item as T)
        is BufferItemType.Error -> downStream.onError(value.e)
        BufferItemType.Complete -> downStream.onComplete()
    }
}
```

Backpressure

```
while (true) {
    val value = synchronized(monitor) {
        val nextValue = buffer.popFirstOrNull()
        if (nextValue == null) {
            isDrainActive = false
            return
        }
        nextValue
    }
    when (value) {
        is BufferItemType.Item<*> -> downStream.onNext(value.item as T)
        is BufferItemType.Error -> downStream.onError(value.e)
        BufferItemType.Complete -> downStream.onComplete()
    }
}
```

Backpressure

```
while (true) {
    val value = synchronized(monitor) {
        val nextValue = buffer.popFirstOrNull()
        if (nextValue == null) {
            isDrainActive = false
            return
        }
        nextValue
    }
    when (value) {
        is BufferItemType.Item<*> -> downStream.onNext(value.item as T)
        is BufferItemType.Error -> downStream.onError(value.e)
        BufferItemType.Complete -> downStream.onComplete()
    }
}
```

Backpressure

Итоги:

- Все асинхронные операторы поддерживают backpressure
- Не боимся случайного переполнения буфера
- «Выравнивание» потока данных спасает от неожиданных эмитов при смене потоков

Сложность №5 — Subjects(горячие источники)

Subjects

Требования:

1. Поддерживает множество подписчиков
2. Может повторить N последних значений для нового подписчика
3. Поддерживает Backpreassure
4. Потокобезопасен в плане одновременной подписки/отписки/эмитов значений из разных потоков

Subjects

```
public class MutableSubject<T>(
    private val replayCount: Int = 0,
    bufferSize: Int = 128,
) : Subject<T> {

    private val replayBuffer = ArrayDeque<T>()
    ........
}
```

Subjects

```
private val observers = CopyOnWriteArraySet<BufferEmitProcessor<T>>()

private val processor = BufferDropOldestEmitProcessor(downstream, bufferSize)

private val downstream = object : ObservableObserver<T> {
    override fun onSubscribe(d: Disposable) = Unit

    override fun onComplete() = Unit

    override fun onError(e: Throwable) = Unit

    override fun onNext(item: T) {
        observers.forEach { emitter ->
            emitter.emit(item)
            emitter.drain()
        }
    }
}
```

Subjects

```
val downstreamEmitProcessor: BufferEmitProcessor<T>
if (replayCount == 0) {
    downstreamEmitProcessor = backpressureStrategy.createBufferEmitProcessor(downstream)
    observers.add(downstreamEmitProcessor)
} else {
    synchronized(replayBufferMonitor) {
        downstreamEmitProcessor = backpressureStrategy.createBufferEmitProcessor(downstream)
        downstreamEmitProcessor.emitAll(replayBuffer.toList())

        observers.add(downstreamEmitProcessor)
    }
}
downstreamEmitProcessor.drain()
```

Subjects

```
public fun emit(item: T) {
  fillBuffer(item)

   processor.emit(item)
   processor.drain()
}
```

Subjects

```
private fun fillBuffer(item: T) {
    if (replayCount == 0) return

    synchronized(replayBufferMonitor) {
        if (replayBuffer.size >= replayCount) {
            replayBuffer.removeFirstOrNull()
        }
        replayBuffer.addLast(item)
    }
}
```

Subjects

```
val downstreamEmitProcessor: BufferEmitProcessor<T>
if (replayCount == 0) {
    downstreamEmitProcessor = backpressureStrategy.createBufferEmitProcessor(downstream)
    observers.add(downstreamEmitProcessor)
} else {
    synchronized(replayBufferMonitor) {
        downstreamEmitProcessor = backpressureStrategy.createBufferEmitProcessor(downstream)
        downstreamEmitProcessor.emitAll(replayBuffer.toList())

        observers.add(downstreamEmitProcessor)
    }
}
downstreamEmitProcessor.drain()
```

Subjects

Итоги:

1. Реализовали универсальный горячий источник.
2. Источник является потокобезопасным даже с использованием повторов.
3. На основе рассмотренного MutableSubject легко реализовали StateSubject

Подводим итоги

Итоги

- Не нужно тащить привычные фреймворки в SDK.
  Лучше страдать самим, чем ставить под удар ваших потребителей.
- Разработка своего фреймворка для асинхронного программирования — очень нетривиальная задача. Даже хорошо понимая концепцию реактивного программирования, некоторые операторы могут быть реализованы корректно не с первого раза.
- Разработка своего велосипеда — это весело ☺
- Выкладываем ReactiveSDK в OpenSource

Готов ответить на ваши вопросы!

TG-канал для разработчиков

TG-чат с разработчиками