RxJava 广泛用于大型 Android 项目，一个使用可观察序列组成异步和基于事件的程序库，作为管理流和多线程的主要工具。

它扩展了观察者模式以支持数据/事件序列，并添加了允许以声明方式将序列组合在一起的运算符，同时抽象出对低级线程、同步、线程安全和并发数据结构等事物的关注。

上游、下游：

RxJava 中的数据流由一个源、零个或多个中间步骤组成，然后是数据消费者或组合器步骤（其中该步骤负责通过某种方式使用数据流）：

source.operator1().operator2().operator3().subscribe(consumer);

source.flatMap(value -> source.operator1().operator2().operator3());

在这里，如果我们想象自己在 operator2 上，向左看向源称为上游。向右看订阅者/消费者称为下游。

运动中的物体：

在 RxJava 的文档中，emission、emits、item、event、signal、data 和 message 被认为是同义词，代表沿着数据流传播的对象。

背压：

当数据流通过异步步骤运行时，每个步骤可能以不同的速度执行不同的事情。为了避免压倒这些步骤，这通常表现为由于临时缓冲或需要跳过/丢弃数据而增加的内存使用量，应用了所谓的背压，这是一种流控制形式，其中步骤可以表达多少项他们准备好处理了吗？这允许在步骤通常无法知道上游将发送给它的项目数量的情况下限制数据流的内存使用情况。

在 RxJava 中，专用的 Flowable 类被指定用于支持背压，而 Observable 专用于非背压操作（短序列、GUI 交互等）。其他类型 Single、Maybe 和 Completable 不支持背压，也不应该支持；总有空间暂时存放一件物品。

装配时间：

通过应用各种中间运算符来准备数据流发生在所谓的组装时间：

Flowable<Integer> flow = Flowable.range(1, 5)

.map(v -> v \* v)

.filter(v -> v % 3 == 0);

此时，数据还没有流动，也没有发生任何副作用。

订阅时间：

这是在内部建立处理步骤链的流程上调用 subscribe() 时的临时状态：

flow.subscribe(System.out::println)

这是触发订阅副作用的时间（请参阅 doOnSubscribe）。在这种状态下，一些源会立即阻止或开始发射项目。

运行：

这是流主动发出项目、错误或完成信号时的状态：

Observable.create(emitter -> {

while (!emitter.isDisposed()) {

long time = System.currentTimeMillis();

emitter.onNext(time);

if (time % 2 != 0) {

emitter.onError(new IllegalStateException("Odd millisecond!"));

break;

}

}

})

.subscribe(System.out::println, Throwable::printStackTrace);

实际上，这是上面给定示例的主体执行的时间。

建造者模式：

import io.reactivex.rxjava3.schedulers.Schedulers;

Flowable.fromCallable(() -> {

Thread.sleep(1000); // imitate expensive computation

return "Done";

})

.subscribeOn(Schedulers.io())

.observeOn(Schedulers.single())

.subscribe(System.out::println, Throwable::printStackTrace);

Thread.sleep(2000); // <--- wait for the flow to finish

这种链接方法的风格称为流式 API，类似于构建器模式。然而，RxJava 的响应式类型是不可变的；每个方法调用都会返回一个带有附加行为的新 Flowable。为了说明，示例可以重写如下：

Flowable<String> source = Flowable.fromCallable(() -> {

Thread.sleep(1000); // imitate expensive computation

return "Done";

});

Flowable<String> runBackground = source.subscribeOn(Schedulers.io());

Flowable<String> showForeground = runBackground.observeOn(Schedulers.single());

showForeground.subscribe(System.out::println, Throwable::printStackTrace);

Thread.sleep(2000);

可以通过 subscribeOn 将计算或阻塞 IO 移动到其他线程。数据准备好后，您可以确保它们通过 observeOn 在前台或 GUI 线程上得到处理。

Schedulers：

RxJava 操作符不直接与线程或 ExecutorServices 一起工作，而是与所谓的调度程序一起工作，这些调度程序在统一的 API 后面抽象出并发源。 RxJava 3 具有几个标准调度程序，可通过 Schedulers 实用程序类访问。

Schedulers.computation()：在后台在固定数量的专用线程上运行计算密集型工作。大多数异步操作符使用它作为他们的默认调度程序。

Schedulers.io()：在一组动态变化的线程上运行类似 I/O 或阻塞操作。

Schedulers.single()：以顺序和先进先出的方式在单个线程上运行工作。

Schedulers.trampoline()：在其中一个参与线程中以顺序和 FIFO 方式运行工作，通常用于测试目的。

这些在所有 JVM 平台上都可用，但某些特定平台（例如 Android）定义了自己的典型调度程序：AndroidSchedulers.mainThread()、SwingScheduler.instance() 或 JavaFXSchedulers.gui()。

此外，还有一个选项可以通过 Schedulers.from(Executor) 将现有的 Executor（及其子类型，例如 ExecutorService）包装到 Scheduler 中。例如，这可以用于拥有更大但仍然固定的线程池（分别不同于computation()和 io()）。

流内的并发：

RxJava 中的流程本质上是顺序的，分为可能彼此并发运行的处理阶段：

Flowable.range(1, 10)

.observeOn(Schedulers.computation())

.map(v -> v \* v)

.blockingSubscribe(System.out::println);

此示例流程在计算调度程序上对从 1 到 10 的数字进行平方，并在“主”线程（更准确地说，blockingSubscribe 的调用者线程）上使用结果。但是，lambda v -> v \* v 不会在此流程中并行运行；它在同一个计算线程上一个接一个地接收值 1 到 10。

并行处理：

并行处理数字 1 到 10 涉及更多：

Flowable.range(1, 10)

.flatMap(v ->

Flowable.just(v)

.subscribeOn(Schedulers.computation())

.map(w -> w \* w)

)

.blockingSubscribe(System.out::println);

实际上，RxJava 中的并行性意味着运行独立的流程并将其结果合并回单个流程。运算符 flatMap 通过首先将每个数字从 1 到 10 映射到它自己的单独的 Flowable 中，运行它们并合并计算的平方。

但是请注意，flatMap 不保证任何顺序，并且来自内部流的项目最终可能会交错。有替代运算符：

concatMap 一次映射和运行一个内部流，并且

concatMapEager “一次”运行所有内部流，但输出流将按照创建这些内部流的顺序排列。

或者，Flowable.parallel() 运算符和 ParallelFlowable 类型有助于实现相同的并行处理模式：

Flowable.range(1, 10)

.parallel()

.runOn(Schedulers.computation())

.map(v -> v \* v)

.sequential()

.blockingSubscribe(System.out::println);

相关子流：

flatMap 是一个强大的运算符，可以在很多情况下提供帮助。例如，给定一个返回 Flowable 的服务，我们想用第一个服务发出的值调用另一个服务：

Flowable<Inventory> inventorySource = warehouse.getInventoryAsync();

inventorySource

.flatMap(inventoryItem -> erp.getDemandAsync(inventoryItem.getId())

.map(demand -> "Item " + inventoryItem.getName() + " has demand " + demand))

.subscribe(System.out::println);

依赖：

最典型的场景是给定一个值，调用另一个服务，等待并继续其结果：

service.apiCall()

.flatMap(value -> service.anotherApiCall(value))

.flatMap(next -> service.finalCall(next))

通常情况下，后面的序列需要来自早期映射的值。这可以通过将外部 flatMap 移动到前一个 flatMap 的内部来实现，例如：

service.apiCall()

.flatMap(value ->

service.anotherApiCall(value)

.flatMap(next -> service.finalCallBoth(value, next))

)

原始值将在内部 flatMap 中可用，由 lambda 变量捕获提供。

非依赖：

在其他情况下，第一个源/数据流的结果是不相关的，并且希望继续使用准独立的另一个源。在这里，flatMap 也可以：

Observable continued = sourceObservable.flatMapSingle(ignored -> someSingleSource)

continued.map(v -> v.toString())

.subscribe(System.out::println, Throwable::printStackTrace);

然而，在这种情况下，延续仍然是 Observable，而不是可能更合适的 Single。 （这是可以理解的，因为从 flatMapSingle 的角度来看，sourceObservable 是一个多值源，因此映射也可能导致多个值）。

通常，通过使用 Completable 作为中介和它的操作符 andThen 用其他东西来恢复，通常有一种更具表现力（并且开销也更低）的方法：

sourceObservable

.ignoreElements() // returns Completable

.andThen(someSingleSource)

.map(v -> v.toString())

sourceObservable 和 someSingleSource 之间的唯一依赖关系是前者应该正常完成才能使用后者。

延迟依赖：

有时，前一个序列和新序列之间存在隐含的数据依赖关系，由于某种原因，它没有通过“常规通道”流动。有人会倾向于编写如下的延续：

AtomicInteger count = new AtomicInteger();

Observable.range(1, 10)

.doOnNext(ignored -> count.incrementAndGet())

.ignoreElements()

.andThen(Single.just(count.get()))

.subscribe(System.out::println);

不幸的是，这会打印 0，因为 Single.just(count.get()) 在数据流尚未运行时在组装时进行评估。我们需要一些东西，将这个单一来源的评估推迟到运行时，当主来源完成时：

AtomicInteger count = new AtomicInteger();

Observable.range(1, 10)

.doOnNext(ignored -> count.incrementAndGet())

.ignoreElements()

.andThen(Single.defer(() -> Single.just(count.get())))

.subscribe(System.out::println);

或者：

AtomicInteger count = new AtomicInteger();

Observable.range(1, 10)

.doOnNext(ignored -> count.incrementAndGet())

.ignoreElements()

.andThen(Single.fromCallable(() -> count.get()))

.subscribe(System.out::println);

执行流程：

Observable.just("Hey")

.subscribeOn(Schedulers.io())

.map(String::length)

.subscribeOn(Schedulers.computation())

.observeOn(AndroidSchedulers.mainThread())

.doOnSubscribe { doAction() }

.flatMap {

doAction()

Observable.timer(1, TimeUnit.SECONDS)

.subscribeOn(Schedulers.single())

.doOnSubscribe { doAction() }

}.subscribe {

doAction()

}

生命周期：初始化、运行、死亡

生命周期包括四个重要步骤：

1、创建 Observables ⬇️

运算符是在当前执行线程上从上到下创建的。 （类似于 Builder 模式。）

2、运算符订阅⬆️

自下而上进行。每个运算符订阅上一个

3、数据发送⬇️

当所有操作员成功订阅对方时开始。从上往下走。

1. 退订。整个链条都死了

所有运算符的执行停止。

1. 创建 Observables ⬇️

Observable.just("Hey") 是用值“Hey”创建的

如果我们查看源代码，我们会发现我们的值只是简单地缓存在 ObservableJust 中：

@CheckReturnValue

@NonNull

@SchedulerSupport(SchedulerSupport.NONE)

public static <T> Observable<T> just(T item) {

ObjectHelper.requireNonNull(item, "item is null");

return RxJavaPlugins.onAssembly(new ObservableJust<T>(item));

}

之后，subscribeOn() 的创建类似于 Observable.just()。

Observable.just("Hey")

.subscribeOn(Schedulers.io())

请注意，在此阶段仅创建运算符，而无需更改线程。

与前一个运算符的主要区别在于，该流存储了对上一个 Observable 的引用：

@CheckReturnValue

@SchedulerSupport(SchedulerSupport.CUSTOM)

public final Observable<T> subscribeOn(Scheduler scheduler) {

ObjectHelper.requireNonNull(scheduler, "scheduler is null");

//"this" it's top observable

return RxJavaPlugins.onAssembly(new ObservableSubscribeOn<T>(this, scheduler));

}

直到 subscribe() 的所有其他运算符都以相同的方式创建，并且不执行任何操作。

链在创建阶段的行为：

Observable.just("Hey")

→ 缓存传递的值

subscribeOn（Schedulers.io（））

→ 缓存订阅调度程序

map（String::length）

→ 使用映射器缓存 lambda

subscribeOn(Scheduler.computation())

→ 缓存订阅调度程序

observeOn（AndroidSchedulers.mainThread（））

→ 缓存发射调度器

doOnSubscribe { doAction() }

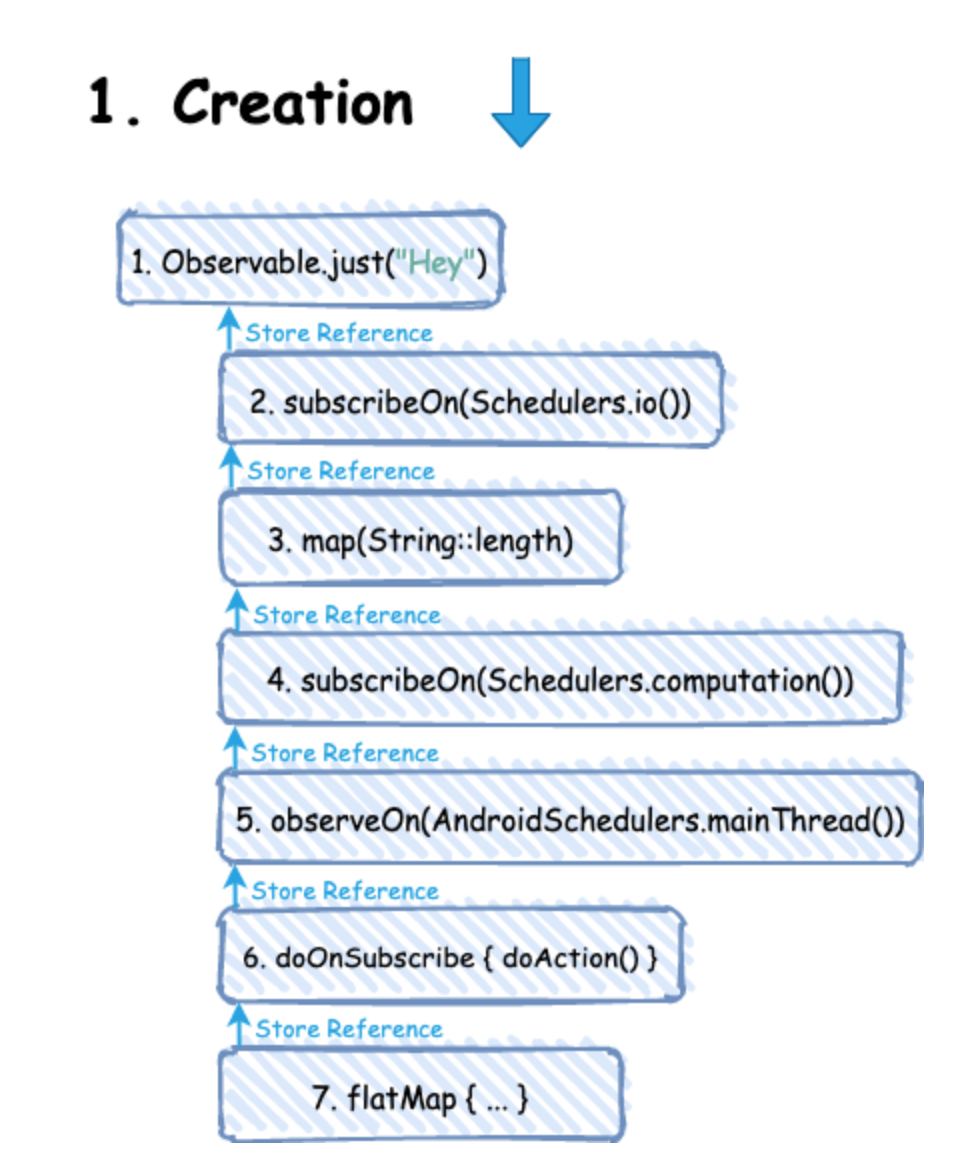
→ 缓存将在订阅时执行的 lambda

flatMap { ... }

→ 缓存将在订阅时执行的 lambda

结果，除了第一个运算符之外，每个运算符都包含对前一个运算符的引用，从而形成一个 LinkedList。

创建将在链调用线程中执行。



2、运算符订阅⬆️

订阅过程自下而上，并在订阅调用后立即开始。

在这个阶段，一些运算符已经可以执行了。

当我们调用 subscribe 时，订阅并不适用于整个链，而只适用于 flatMap，后者更高。

然后 flatMap 订阅它在创建 doOnSubscribe 时保存其引用的运算符。这就是运算符保留对上游流的引用的原因。

Rx 就像一棵逐渐开放的卷心菜，一片一片。

链在订阅阶段的行为：

flatMap { ... }

→ 订阅上游

doOnSubscribe{ ... }

→ 执行 lambda 中的代码

→ 订阅上游

→ 执行线程：调用 subscribe 的线程

observeOn（AndroidSchedulers.mainThread（））

→ 订阅上游

subscribeOn(Scheduler.computation())

→ 将线程更改为computation（上一个线程 = 订阅线程）

→ 订阅上游

map(String::length)

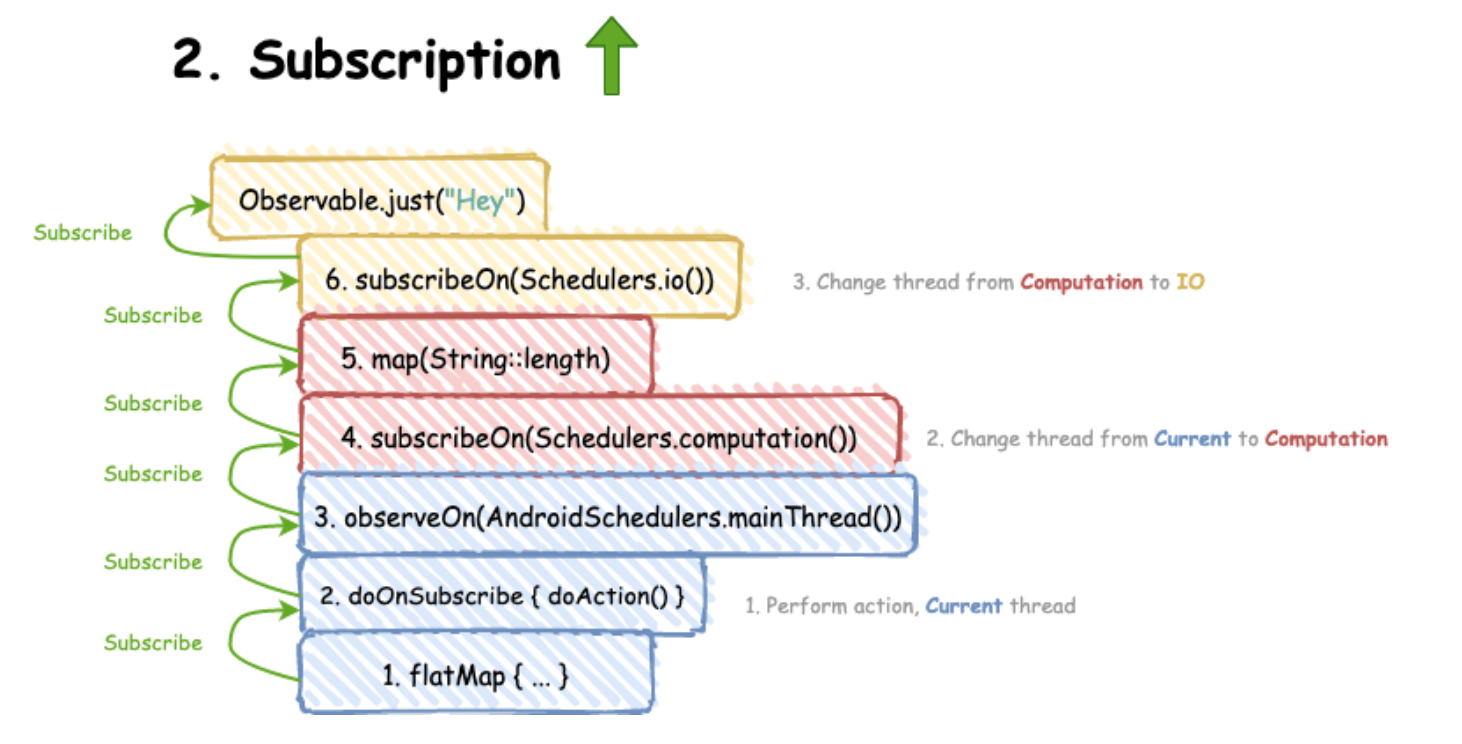
→ 订阅上游

→ 在computation订阅

subscribeOn（Schedulers.io（））

→ 将线程更改为 IO（前一个线程 = computation）

→ 订阅上游



在所有运算符中，只有一个 doOnSubscribe 和两个 subscribeOn 被执行。线程上下文改变了两次：首先是computation，然后是 IO。也就是说，subscribeOn 将被执行多次。

即使你连续写了 3 次 subscribeOn(Schedulers.io())，前 2 次也不会被忽略。

让我们看一下下面的例子：

Observable.just("Hey")

.doOnSubscribe { doAction() }

.subscribeOn(Schedulers.io())

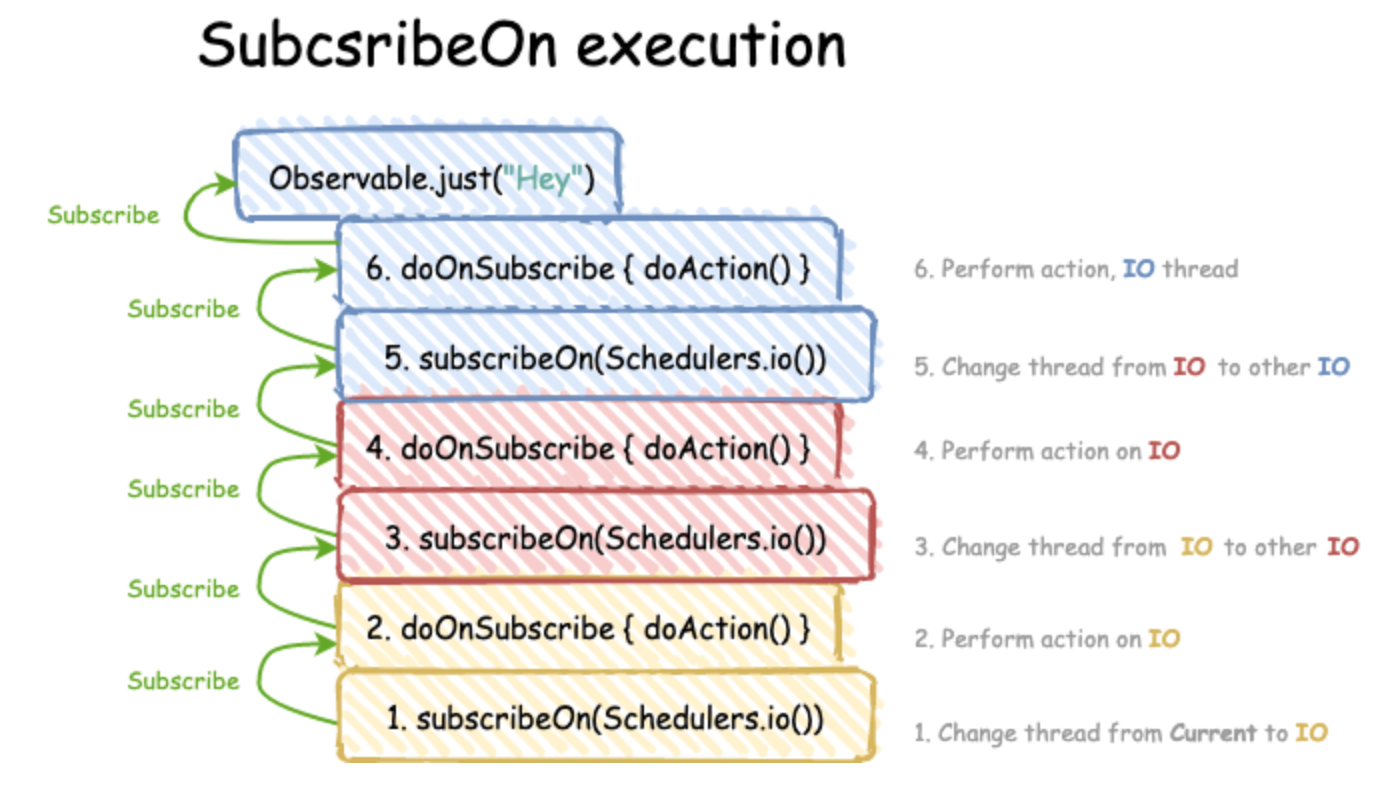
.doOnSubscribe { doAction() }

.subscribeOn(Schedulers.io())

.doOnSubscribe { doAction() }

.subscribeOn(Schedulers.io())

.subscribe { }



请注意，Schedulers.io()的实现是使用 Executor 动态线程池。有时线程实例可能会同时发生，因为繁忙的线程可能会变得空闲并再次可用。

例如，如果现在只运行这个 RX 代码，则线程输出将如下所示：

Thread[RxCachedThreadScheduler-1,5,main] //blue onsubscribe

Thread[RxCachedThreadScheduler-2,5,main] //red onsubscribe

Thread[RxCachedThreadScheduler-1,5,main] //yellow onsubscribe

在计算方面，线程的数量是有限的，直接取决于处理器及其内核的数量。

我们应该只在其上执行消耗资源的任务，例如图像压缩或加密。但绝对不是输入输出操作或文件系统的任何任务。

3、数据发送⬇️

一旦我们到达没有对父流的引用的 Observable，发射过程就开始了。它从根 Observable 到最低的 Observable 。

链的行为：

Observable.just("Hey")

→ 在 IO 上发出“Hey”

subscribeOn（Schedulers.io（））

→ 在现有 IO 上代理“Hey”，并且不更改线程

map(String::length)

→ 在 IO 上运行 lambda 中的代码

subscribeOn(Scheduler.computation())

→ 代理值，仍在 IO 上

observeOn（AndroidSchedulers.mainThread（））

→ 将线程更改为 main，发出值。

重要的是要记住，如果我们已经在主线程上，则不能立即发出元素，因为无论如何它都会通过 Handler/Looper/MessageQueue。

doOnSubscribe { doAction() }

→ 将值代理到下面的流中，在 main

flatMap { ... }

这个很棘手。首先，让我们回顾一下它的内容：

.flatMap {

doAction()

Observable.timer(1, TimeUnit.SECONDS)

.subscribeOn(Schedulers.single())

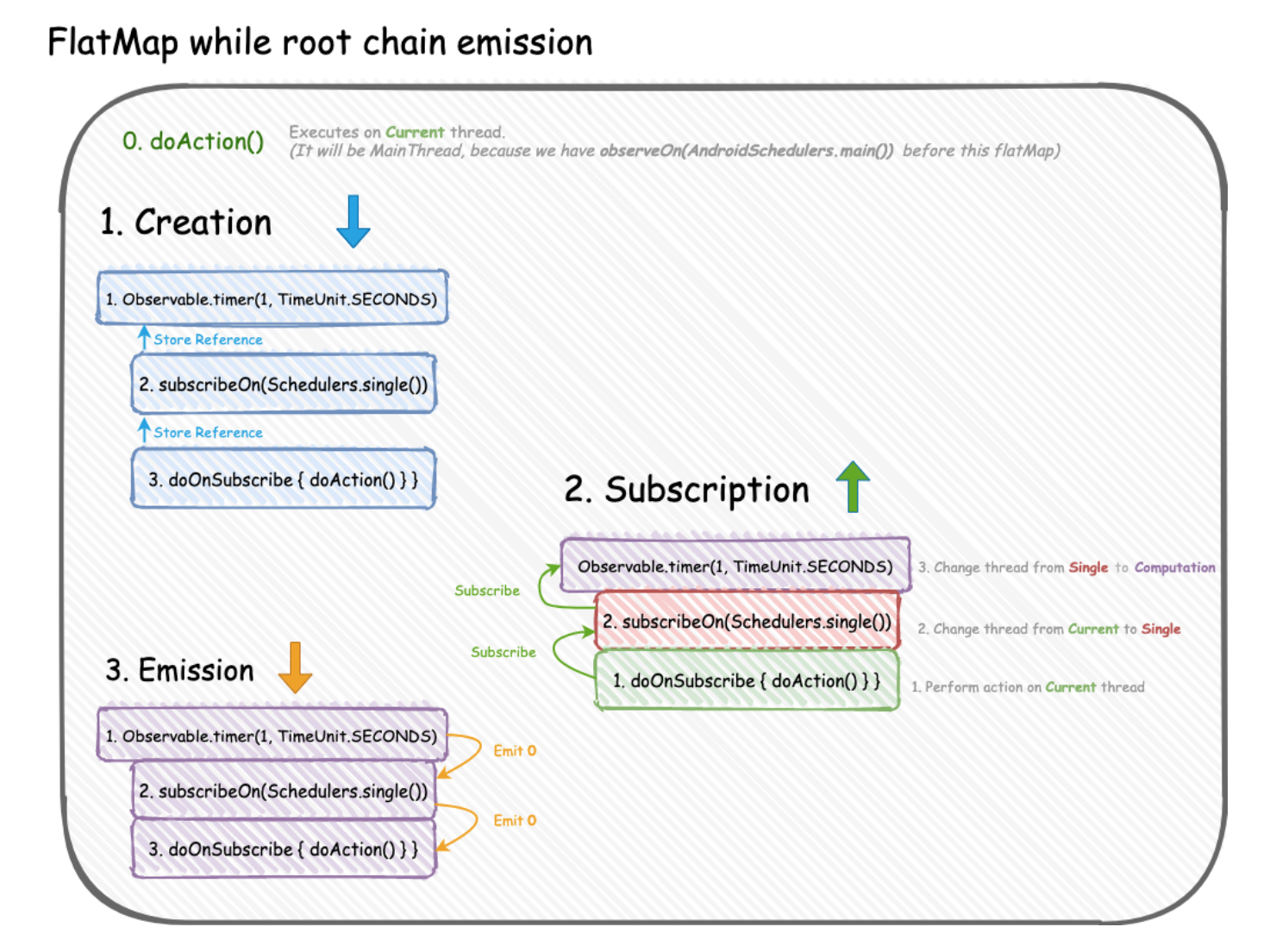
.doOnSubscribe { doAction() }

}

flatMap 内部的链会经历创建、订阅、发射的所有阶段。在这种情况下，不会取消订阅，因为计时器将继续发出值。

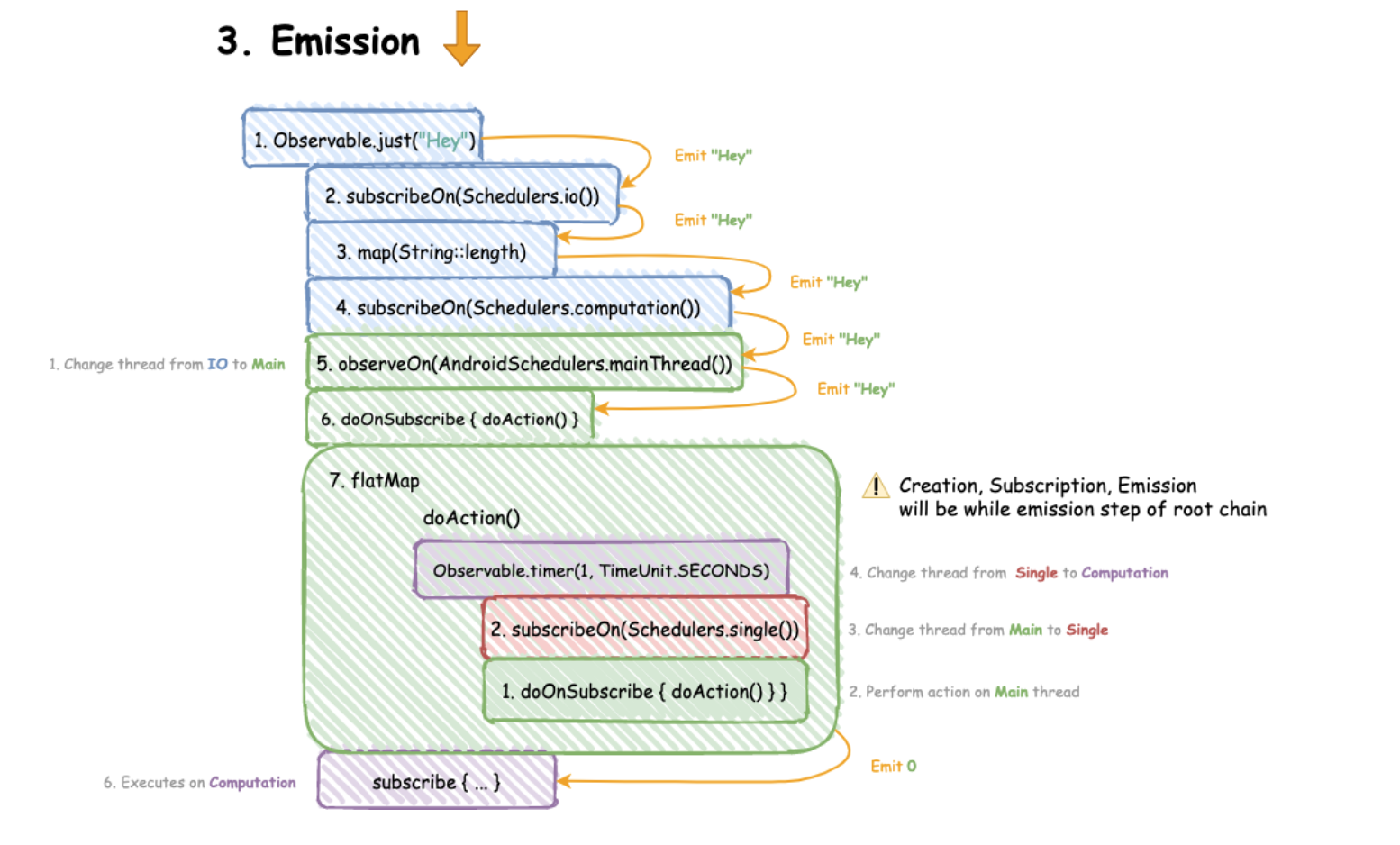
运算符 timer/delay/interval 在后台订阅计算。这一步并不明显，但会影响下游。

因此请记住，如果您将这些运算符与 http/database 请求结合使用，请在执行此请求之前将调度程序更改为 IO。否则，将订阅调度程序作为参数直接传递给 timer/delay/interval。



subscribe{ doAction() }

对计算执行操作，返回 Disposable。通过使用 Disposable，可以管理整个订阅的生命周期。



总结：

所有运算符都是自上而下创建的，自下而上一一订阅。因此，它们形成了一个单独的 LinkedList。

发射从根 Observable 开始从上到下。

订阅过程中会执行 startWith/doOnSubscribe 等运算符。更重要的是，它们的执行次数与编写它们的次数一样多。它们的执行线程可能会受到下面写的 subscribeOn 的影响。

subscribeOn 在订阅过程中执行，同时影响上游和下游。它还可以在编写线程时多次更改线程。应用最接近链顶部的那个。

observeOn 只影响较低的流，并在发射期间执行。它会在编写线程时多次更改线程。

flatMap 仅在根链数据发射期间启动链。

在根流订阅过程中不执行任何操作。

默认情况下，运算符interval/delay/timer订阅底层计算。当将这些运算符用于 http/db 请求时，请记住将调度程序更改为 IO。