尽管 Coroutines 在开发人员中引发了很多问题，但如今几乎每个新项目都不再包含对 RxJava 的依赖。而Flow在其中发挥了重要作用。

CoroutineScope(context = Dispatchers.Main.immediate).launch() {

doAction()

flowOf("Hey")

.onEach { doAction() }

.map { it.length }

.onStart { doAction() }

.flowOn(Dispatchers.Default)

.flatMapMerge {

doAction()

flowOf(1)

.flowOn(Dispatchers.Main)

.onEach { doAction() }

}

.flowOn(Dispatchers.IO)

.collect {

doAction()

}

}

}

我们的目标是确定每个动作将在哪个顺序和哪个线程上执行。

生命周期：

Flow 仅建立在两个接口上：

public interface Flow<out T> {

public suspend fun collect(collector: FlowCollector<T>)

}

public fun interface FlowCollector<in T> {

public suspend fun emit(value: T)

}

它们共同构成了消费者和生产者模式实现的基础。

每个 Flow 链都由一组特定的运算符组成。每个运算符创建一个新的 Flow 实例，但同时存储对位于上面的另一个 Flow 实例的引用。在调用 collect 方法之前，运算符不会启动。

生命周期分为五个重要阶段：

启动⬇️

一个协程在我们传输到 CoroutineScope 的 Dispatcher 上启动。

之后，将执行以下步骤：Flow Creation、Operators Collection和Data Emission。最终结果将在传输的 Dispatcher 上进行处理。

Flow Creation⬇️

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。）

Operators Collection⬆️

自下而上进行。每个Operators 收集上一个

Data Emission⬇️

当所有Operators 都成功地在顶部流中调用 collect 时开始。从上往下走。

取消/完成。整个链条都死了

执行取消/完成

创建：

val job = CoroutineScope(Dispatchers.Main.immediate).launch {

doAction()

/\*.../\*

}

我们创建了一个在主线程上运行协程的作用域。 doAction() 方法也在这个协程上启动。

Scope 返回 Job，我们可以使用它来管理生命周期。 （例如，我们可以通过执行 cancel() 方法来停止所有工作。）

在 Android 中，将线程切换到 Main 的唯一方法是使用 Handler/Looper/MessageQueue 链。这个逻辑隐藏在 HandlerContext 后面，它隐藏在 Dispatchers.Main 后面：

//handler here is created with Looper.mainLooper()

override fun dispatch(context: CoroutineContext, block: Runnable) {

if (!handler.post(block)) {

cancelOnRejection(context, block)

}

}

现在假设我们已经在主线程上，但我们仍然使用 handler.post 切换到主线程

在这种情况下，我们的代码不会立即执行，这可能会影响用户体验，例如导致屏幕闪烁。

代码必须通过可能忙于处理其他命令的 MessageQueue 遍历整个链。 Dispatchers.Main.immediate 的主要思想是跳过这个队列并立即执行代码。

override fun isDispatchNeeded(context: CoroutineContext): Boolean {

return !invokeImmediately || Looper.myLooper() != handler.looper

}

Dispatchers.Main.immediate 在 HandlerContext 上创建一个新实例，它将 invokeImmediately 切换为 true。因此，主线程的 Looper 将始终与当前线程的 Looper 进行比较，从而防止对 handler.post 的额外调用。

Flow Creation：

我们链的起点是 flowOf(“hey”)。

在后台，我们可以看到我们显式地创建了一个 Flow 的新实例并将值存储在一个 lambda 中，该 lambda 将在收集阶段被调用：

import kotlinx.coroutines.flow.internal.unsafeFlow as flow

public fun <T> flowOf(value: T): Flow<T> = flow {

emit(value)

}

internal inline fun <T> unsafeFlow(crossinline block: suspend FlowCollector<T>.() -> Unit): Flow<T> {

return object : Flow<T> {

override suspend fun collect(collector: FlowCollector<T>) {

collector.block()

}

}

}

之后，将以相同的方式创建 onEach 运算符。

然而，它是一个扩展并且明确地保留了对前一个 Flow 的引用。

直到 collect() 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：

flowOf("Hey")

→ 缓存传输的值

onEach { doAction() }

→ 缓存将在发射阶段执行的 lambda

map {...}

→ 使用映射器缓存 lambda

onStart { doAction() }

→ 缓存将在收集阶段执行的 lambda

flowOn(Dispatchers.Default)

→ 将分配缓存到调度程序

flatMapMerge { ... }

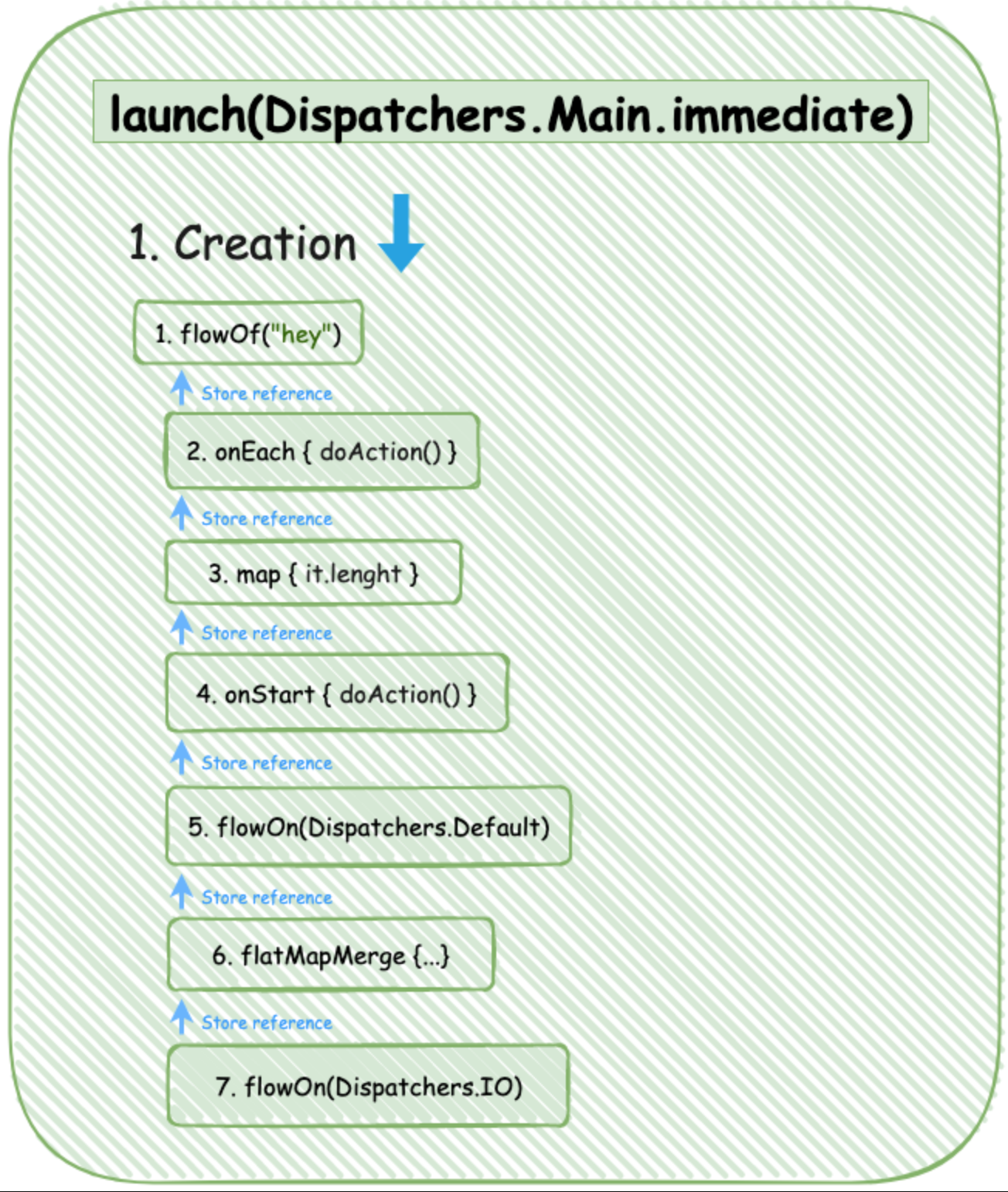
→ 缓存将在发射步骤执行的 lambda

flowOn(Dispatchers.IO)

→ 将分配缓存到调度程序

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。

创建将在主线程中执行。



Operators Collection：

收集过程自下而上进行，并在终端Operators 调用后立即开始。

collect()

first()

toList()

toSet()

reduce()

fold()

当我们调用collect时，collection并不适用于整条链，而只适用于flowOn，后者更高。

然后 flowOn 调用 collect 到它在创建时保存其引用的Operators — flatMapMerge。这就是Operators 保留对上游流的引用的原因。

链在收集阶段的行为：

flowOn(Dispatchers.IO)

→ 在 IO 上创建一个新的协程，将上下文更改为创建的协程

→ 调用 collect 到上游

flatMapMerge { ... }

→ 调用 collect 到上游

flowOn(Dispatchers.Default)

→ 在 Default 上创建一个新的协程，将上下文更改为创建的协程

→ 调用 collect 到上游

onStart { doAction() }

→ 在默认调度程序上执行操作

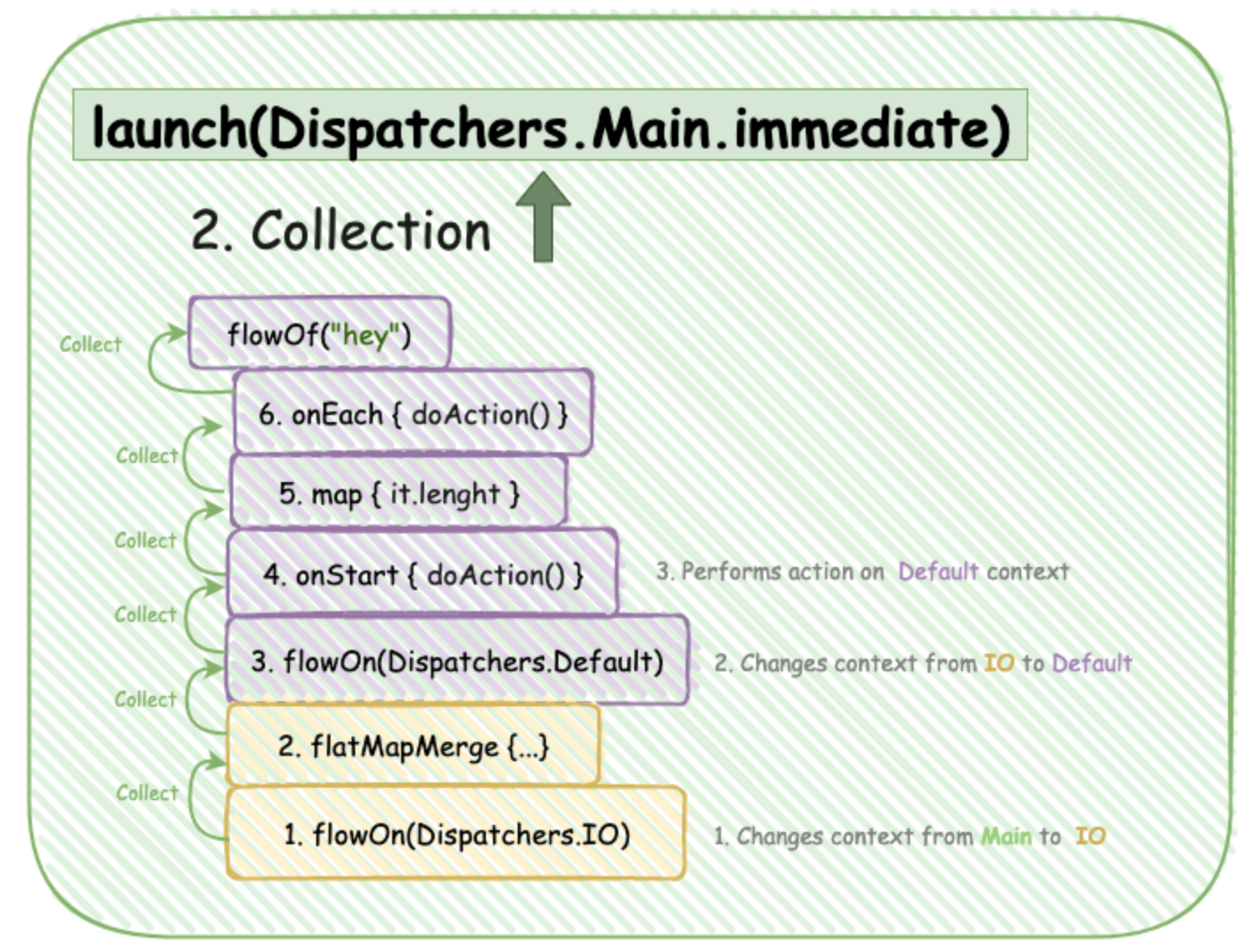
→ 调用 collect 到上游

map { ... }

→ 调用 collect 到上游

onEach { ... }

→ 调用 collect 到上游



在所有操作符中，只有一个 onStart 和两个 flowOn 被执行。

线程上下文改变了两次：首先是 IO，然后是 Default。

也就是说，flowOn 将被执行多次，并会创建一些协程实例。

⚠️ 然而，flowOn 并不总是在底层创建新的协程。让我们看一下下面的例子。

CoroutineScope(Dispatchers.Main.immediate).launch {

flowOf("Hey")

.onStart { doAction() }

.flowOn(Dispatchers.IO)

.onStart { doAction() }

.flowOn(Dispatchers.IO)

.onStart { doAction() }

.flowOn(Dispatchers.IO)

.collect()

}

//onStart1

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Job: ProducerCoroutine{Active}@53f45ab)

Thread: DefaultDispatcher-worker-1,5,main

//onStart2

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Job: ProducerCoroutine{Active}@53f45ab)

Thread: DefaultDispatcher-worker-1,5,main

//onStart3

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Job: ProducerCoroutine{Active}@53f45ab)

Thread: DefaultDispatcher-worker-1,5,main

只创建了一个协程实例，并且链接到一个线程。

Data Emission：

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

flowOf("Hey")

→ 发出嘿，默认调度程序

onEach { doAction() }

→ 执行一个动作，默认调度程序

map {...}

→ 进行映射，默认调度程序

onStart { doAction() }

→ 发出 3，默认调度程序

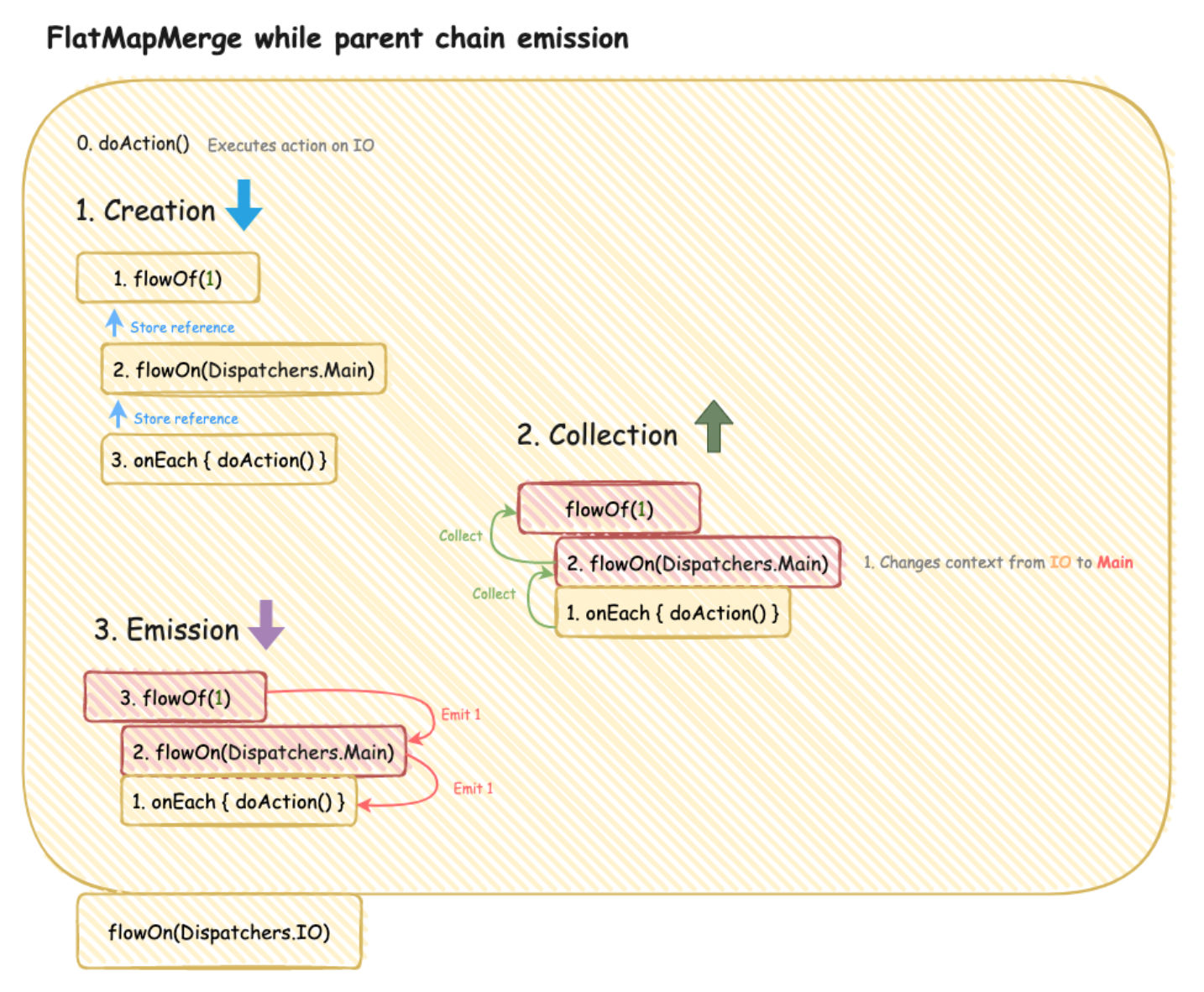
flowOn(Dispatchers.Default)

→ 发出 3，默认调度程序

flatMapMerge { ... }

这个很棘手。首先，让我们回顾一下它的内容：

flatMapMerge 内部的链会经历创建、收集和发射的所有阶段。之后，最终值将被发送到下游。



请注意，onEach 将在 IO Dispatcher 上执行。 （它将在块执行之前恢复。）

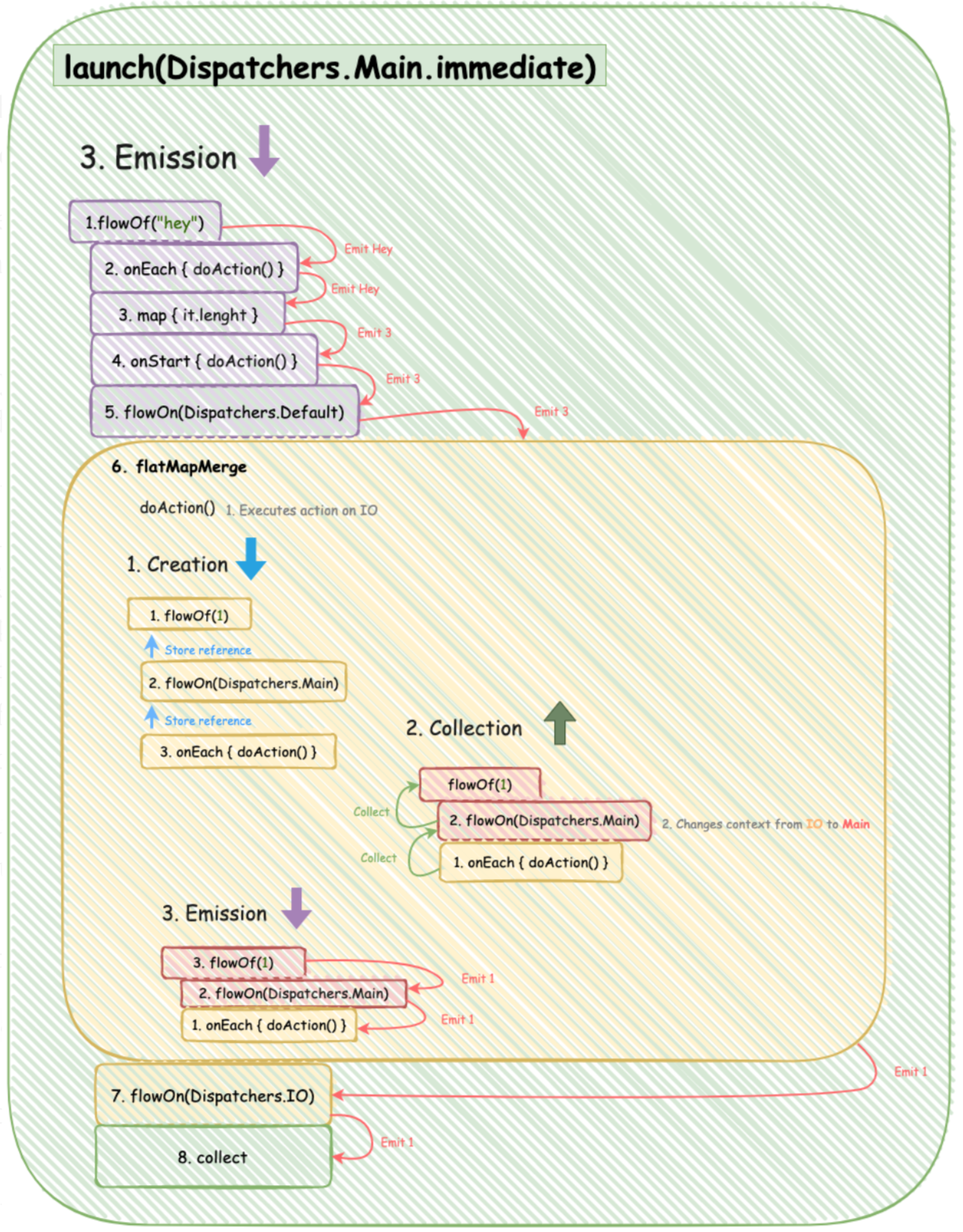
与 RxJava 不同的是，Kotlin Flow 实现了 Context Preservation 的概念，保证了上层流的上下文不会影响到下层流。

flowOn(Dispatchers.IO)

→ 发出 1，IO 调度程序

collect

→ 尽管上游的上下文发生了变化，但仍调用 Main 上的收集器



结论：

1、collect() 方法被暂停，这迫使我们提前决定在哪个上下文中处理我们的链的结果。

2、所有操作符都是从上到下创建的，从下到上一个接一个地互相调用collect。因此，它们形成了一个单独的 LinkedList。发送从根流开始从上到下。

3、一些操作符可能会在收集阶段执行。例如， onStart 执行次数与写入次数一样多。

4、flowOn 创建一个新的协程，在参数中传输 Dispatcher 并更改上下文。 （但是，如果我们有多个 flowOn 使用同一个调度程序，则实际上只会创建一个协程。）

它只影响上游，从而保证符合上下文保留原则。

协程的创建和上下文的改变都可以在两个阶段执行：收集和发射。

5、一个线程可以在多个协程之间使用。如果你写了flowOn，如果当前上下文不同，肯定会创建一个新的协程。但是，不能保证线程会不同于之前协程的线程。

6、flatMapMerge/flatMapConcat 仅在父链数据发射期间启动链。

在根流收集过程中不执行任何操作。