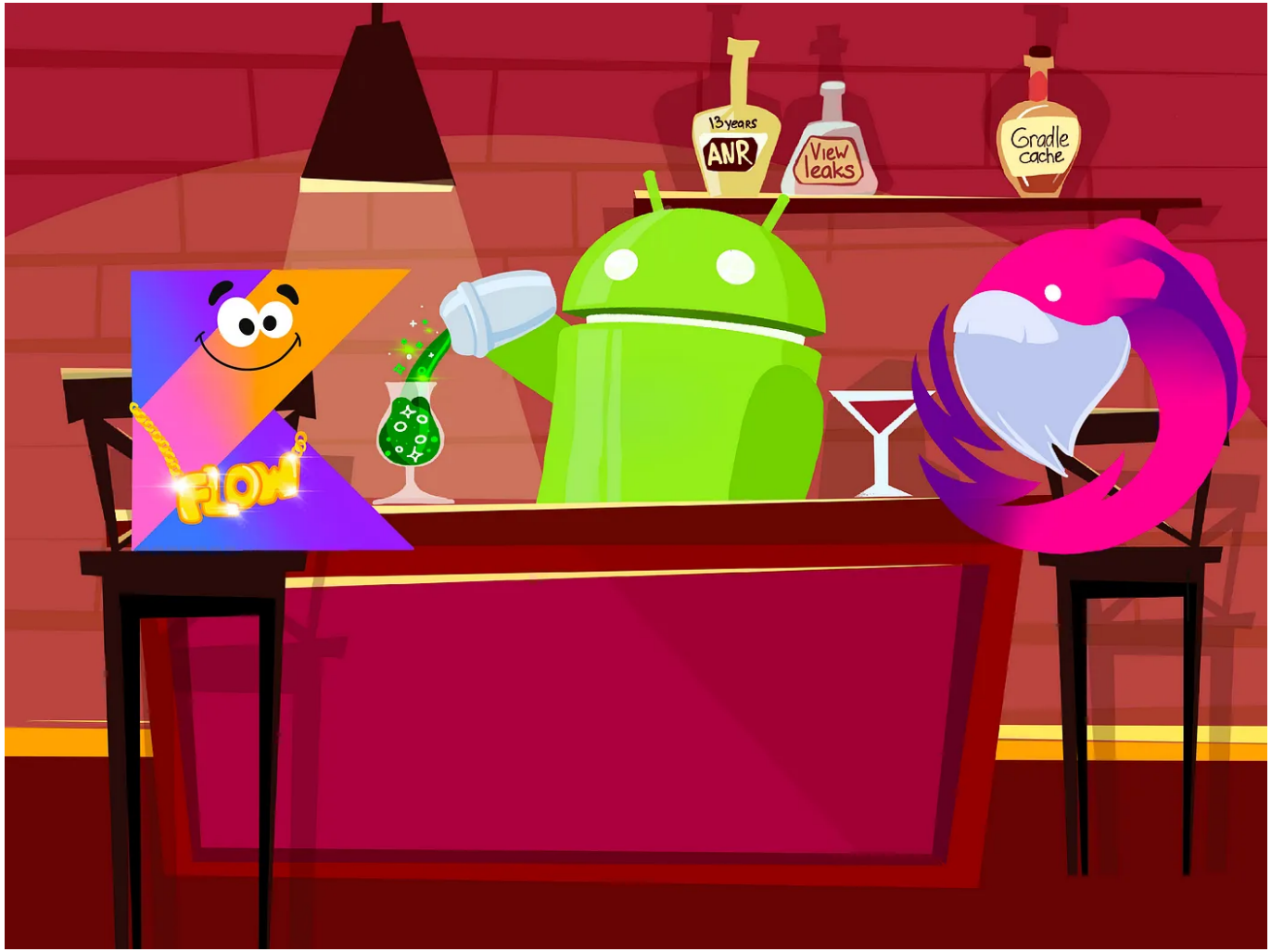
Kotlin Coroutines Flow in a nutshell

Alexey Bykov · Follow

Published in ProAndroidDev · 8 min read · Feb 16, 2022



In my previous post, I described how RxJava actually works. For a long time, it has been a non-official standard and the main tool for working with streams and multithreading in Android. But now we have an alternative from Jetbrains, recommended by Google — `Coroutines`.

Despite the fact that `Coroutines` have been rising a lot of questions among developers, nowadays almost every new project no longer contains a dependency on `RxJava`. And `Flow` played a significant role in this.

What this post is about

In this post, I will tell you how `Flow` works. You will learn about the basic principle and its lifecycle, as well as about dispatching.

This post is addressed both to those who take their first steps trying to understand `Flow`, and those who have years of practice.

The chain

```
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()
            }
    }
}
```

Our goal is to determine in which sequence and on which thread each action will be performed.

Basics and lifecycle

`Flow` is built on just two interfaces:

```
public interface Flow<out T> {
    public suspend fun collect(collector: FlowCollector<T>)
}
public fun interface FlowCollector<in T> {
    public suspend fun emit(value: T)
}
```

Together, they form the basis for the Consumer & Producer pattern implementation.

Before going further, I highly recommend you to take a look at an example of writing a stream API, based on the two interfaces.

Each `Flow` chain consists of a specific set of operators. Each operator creates a new `Flow` instance but at the same time stores a reference to another `Flow` instance that is located above. Operators do not start until the collect method is called. (*Cold operators*).

The lifecycle is presented by five important stages:

1. **Launching** ⬇️
   A coroutine launches on the Dispatcher we transmitted to CoroutineScope. After that, the following steps take place: Flow Creation, Operators Collection & Data Emission. The final result will be processed on the transmitted Dispatcher.
2. **Flow creation** ⬇️
   Operators are created from top to bottom on the current execution thread. (Similar to the Builder pattern.)

3. **Operators collection** ⬆️
   Performed from the bottom up. Each operator collects the upper one
4. **Data emission** ⬇️
   Starts when all operators have successfully called collect at the top stream. Goes from the top down.
5. **Cancellation/Completion. The entire chain dies** 😵
   Execution cancelled/Completed

Let's take a closer look at each of the stages.

Launching

```
val job = CoroutineScope(Dispatchers.Main.immediate).launch {
     doAction()
     //....
}
```

Everything seems clear at the first glance. We create a scope that runs a coroutine on the Main thread. The `doAction()` method is also launched on

this coroutine.

Scope returns `Job` which we can use to manage the lifecycle. *(For example, we can stop all the work by executing the `cancel()` method.)*

Role of immediate dispatcher

In Android, the only way to switch the thread to Main is to use the `Handler/Looper/MessageQueue` chain.

This logic hides behind `HandlerContext`, which hides behind `Dispatchers.Main`

```
//handler here is created with Looper.mainLooper()
override fun dispatch(context: CoroutineContext, block: Runnable) {
    if (!handler.post(block)) {
        cancelOnRejection(context, block)
    }
}
```

Now suppose that we are already on the Main thread, but we are still switching to the Main thread using `handler.post`

In this case, our code will not be executed immediately, which may affect the user experience, for example, cause the screen to blink.

The code will have to go through the entire chain through `MessageQueue` that can be busy processing other commands. The main idea of `Dispatchers.Main.immediate` is to skip this queue and execute the code immediately.

`Dispatcher` has an `isDispatchNeeded` method that helps us out in such a case. In `HandlerContext`, this method is implemented as follows:

```
override fun isDispatchNeeded(context: CoroutineContext): Boolean {
    return !invokeImmediately || Looper.myLooper() != handler.looper
}
```

`Dispatchers.Main.immediate` creates a new instance at `HandlerContext`, which switches `invokeImmediately` to true. As a result, the `Looper` of the Main thread will always be compared with the `Looper` of the current thread, thus preventing an extra call to `handler.post`.

Flow creation

The starting point of our chain is `flowOf("hey")`.
Under the hood, we can see that we explicitly create a new instance of `Flow` and store the value in a lambda, which will be called at the collection stage:

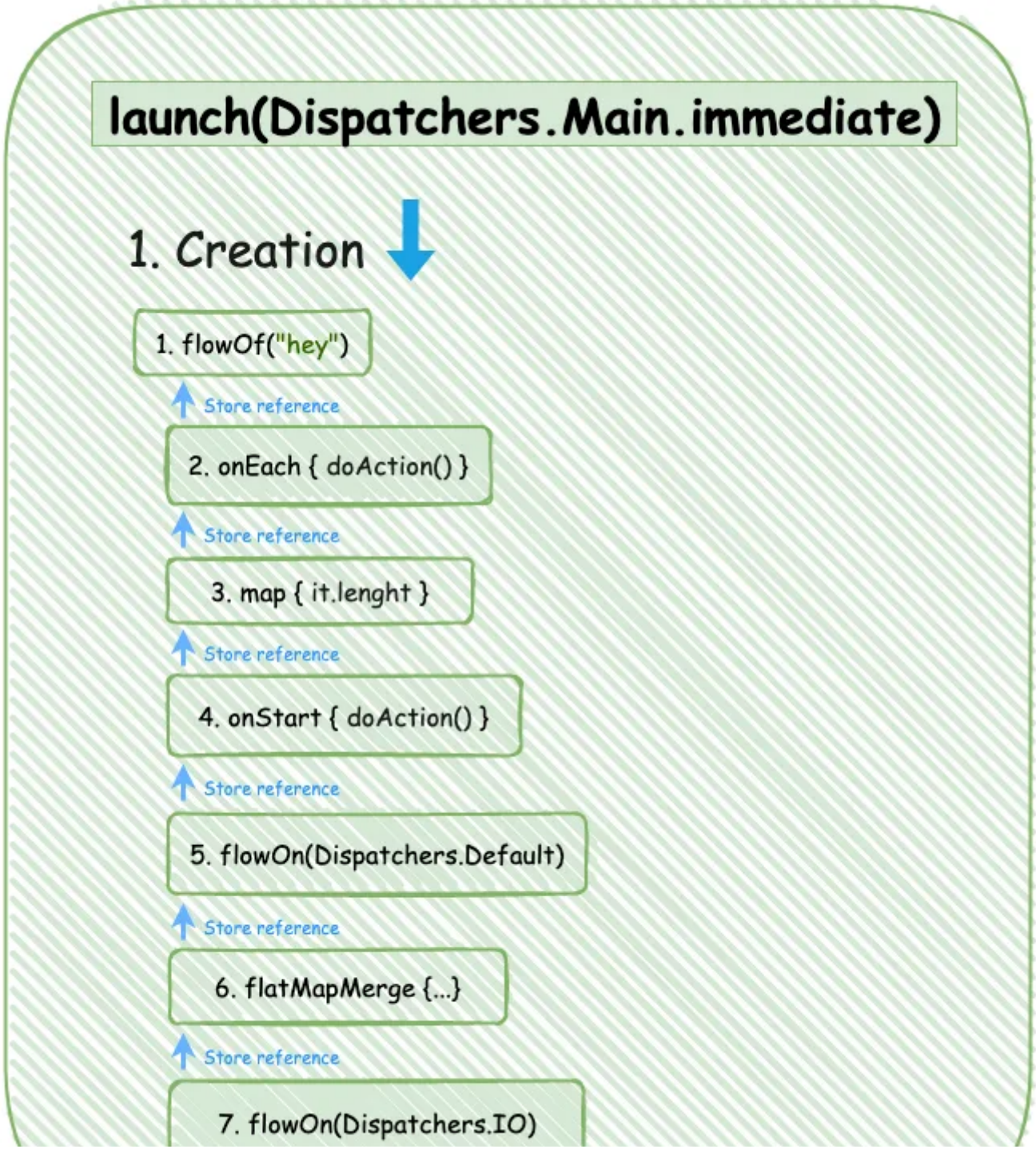
```
import kotlinx.coroutines.flow.internal.unsafeFlow as flow

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

internal inline fun <T> unsafeFlow(crossinline block: suspend FlowCollector<T>.(
    return object : Flow<T> {
        override suspend fun collect(collector: FlowCollector<T>) {
            collector.block()
        }
    }
}
```

After that, the `onEach` operator will be created in the same way.
However, it is an extension and explicitly retains a reference to the previous `Flow`.
All other operators up to `collect()` are created in the same way and do not perform any actions.

The chain's behaviour at the creation stage:

1. `flowOf("Hey")`
   → *caches the transmitted value*
2. `onEach { doAction() }`
   → *caches the lambda that will be executed at the emission stage*
3. `map {...}`
   → *caches a lambda with a mapper*
4. `onStart { doAction() }`
   → *caches the lambda that will be executed at the collecting stage*
5. `flowOn(Dispatchers.Default)`
   → *caches an assignment to the dispatcher*
6. `flatMapMerge { ... }`
   → *caches the lambda that will be executed at the emission step*
7. `flowOn(Dispatchers.IO)`
   → *caches an assignment to the dispatcher*

As a result, each operator, except for the first one, contains a reference to the previous one, thus forming a `LinkedList`.
Creation will be executed at the Main thread.


launch(Dispatchers.Main.immediate)
1. Creation
1. flowOf("hey")
Store reference
2. onEach { doAction() }
Store reference
3. map { it.lenght }
Store reference
4. onStart { doAction() }
Store reference
5. flowOn(Dispatchers.Default)
Store reference
6. flatMapMerge {...}
Store reference
7. flowOn(Dispatchers.IO)

Collection

The collecting process goes from the bottom up and starts immediately after the terminal operator call.

Terminal operators in flow:

- collect()
- first()
- toList()
- toSet()
- reduce()
- fold()

When we call `collect`, the collection does not apply to the entire chain, but only to the `flowOn`, which is higher.
Then `flowOn` calls collect to the operator whose reference it saved while creating — `flatMapMerge`. This is why operators keep references to the upper streams.

The chain's behaviour at the collection stage:

1. `flowOn(Dispatchers.IO)`
   → *creates a new coroutine on IO, changes the context to the created coroutine*
   → *calls collect to the upper stream*

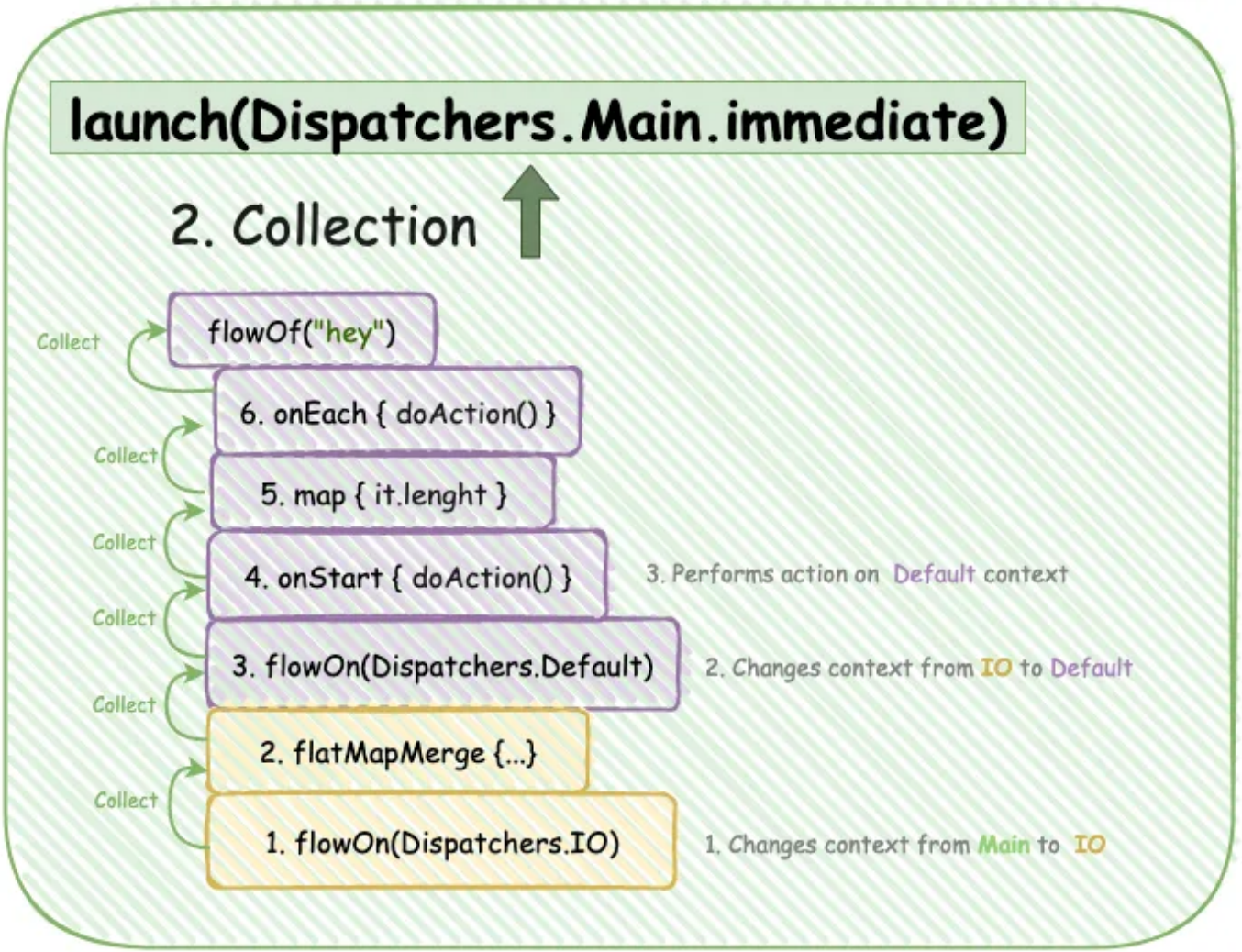
2. `flatMapMerge { ... }`
   → c*alls collect to the upper stream*

3. `flowOn(Dispatchers.Default)`
   → *creates a new coroutine on Default, changes the context to the created coroutine*
   → *calls collect to the upper stream*

4. `onStart { doAction() }`
   → *executes an action on the Default dispatcher*
   → *calls collect to the upper stream*

5. `map { ... }`
   → *calls collect to the upper stream*

6. `onEach { ... }`
   → *calls collect to the upper stream*



Of all the operators, only one `onStart` and two `flowOn` were executed.
The thread context changed twice: first to `IO`, and then to `Default`.
That is, `flowOn` will be executed as many times as it is written and will create a few instances of coroutines.

⚠️ However, `flowOn` does not always create new coroutine under the hood. Let's take a look at the example below.

```
CoroutineScope(Dispatchers.Main.immediate).launch {
    flowOf("Hey")
        .onStart { doAction() }
        .flowOn(Dispatchers.IO)
        .onStart { doAction() }
        .flowOn(Dispatchers.IO)
        .onStart { doAction() }
        .flowOn(Dispatchers.IO)
        .collect()
}
```

We have intentionally written `flowOn` with the same dispatcher many times.

Result

```
//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
```

As you can see, only one instance of the coroutine has been created, and it is linked to one thread.

Emission

As soon as we reach `Flow` that does not have a reference to the parent stream, the emission process starts. It goes from the root `Flow` to the lowest one.

1. **`flowOf("Hey")`**
   → *emits `hey` , Default Dispatcher*
2. **`onEach { doAction() }`**
   → *performs an action, Default Dispatcher*

3. **`map {...}`**
   → *does mapping, Default Dispatcher*

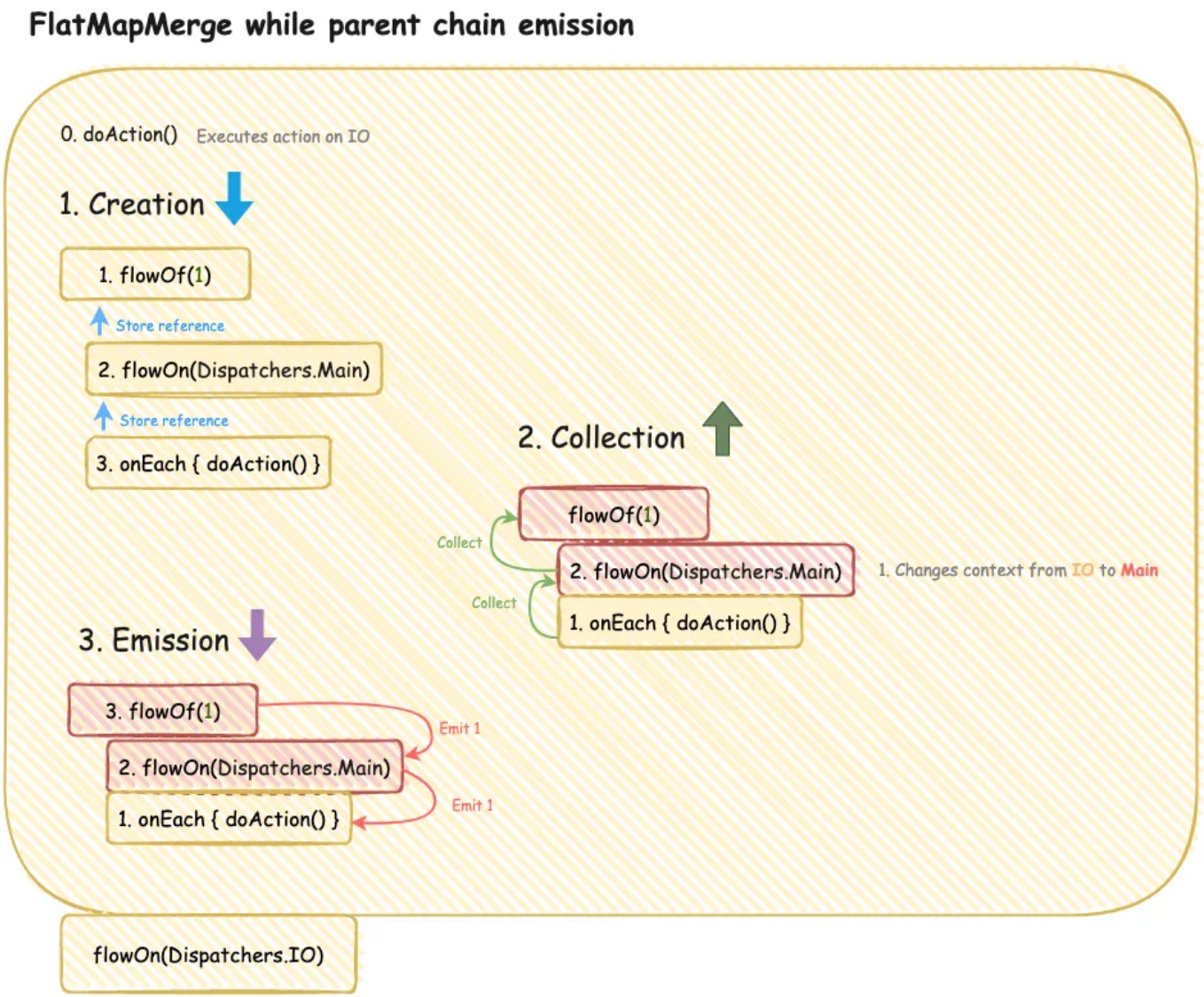
4. **`onStart { doAction() }`**
   → *emits `3`, Default Dispatcher*

5. **`flowOn(Dispatchers.Default)`**
   → *emits `3`, Default Dispatcher*

6. **`flatMapMerge { ... }`**
   This one is tricky. First, let's recall its contents:

```
//...
flatMapMerge {
  doAction()
  flowOf(1)
    .flowOn(Dispatchers.Main)
    .onEach { doAction() }
}
```

The chain inside `flatMapMerge` will go through all the stages of `creation`, `collection` and `emission`. After that, the final value will be emitted to the downstream.



Note that `onEach` will be executed on `IO Dispatcher`. (*It will be restored before the block is executed.*)

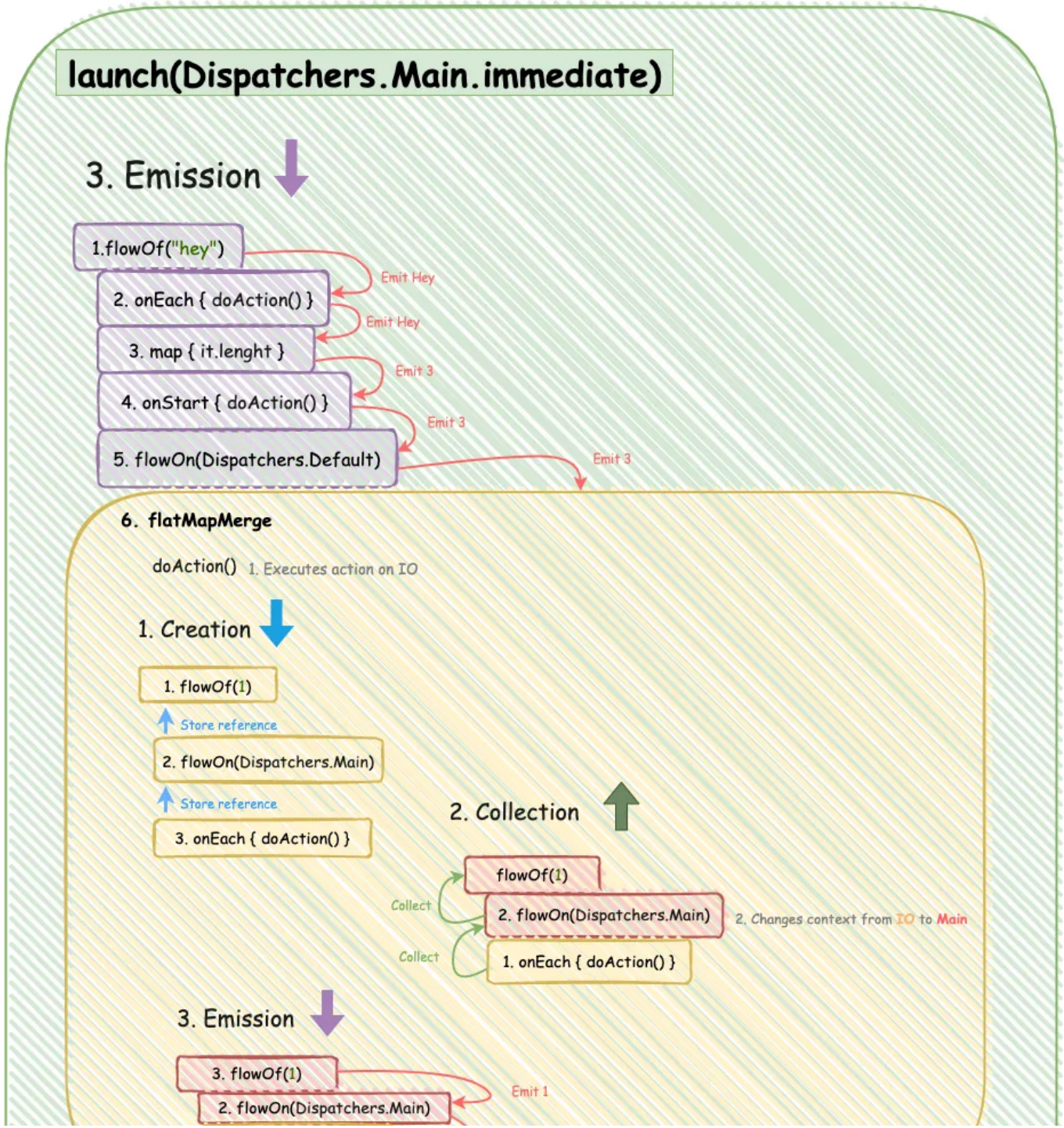
Unlike `RxJava`, `Kotlin Flow` implements the concept of `Context Preservation`, which ensures that **the context of the upper streams cannot affect the lower ones.**

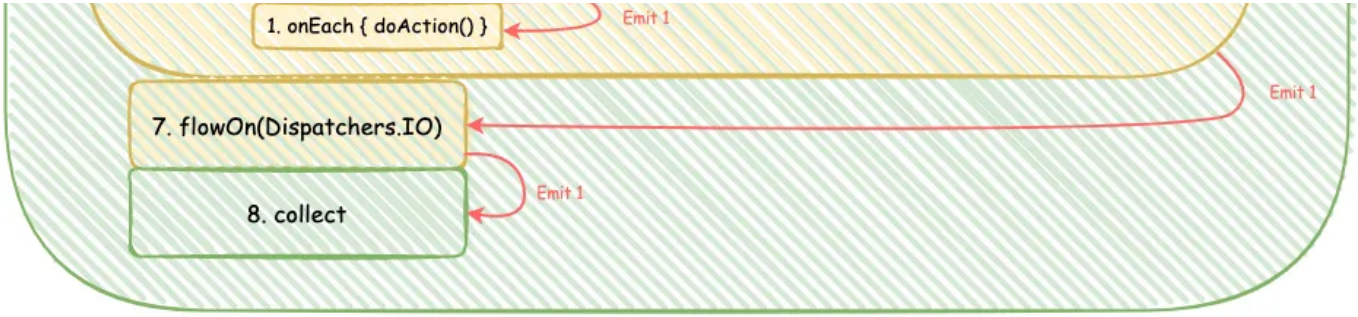
7. **`flowOn(Dispatchers.IO)`**

→ *emits 1, IO Dispatcher*

8. **`collect`**

→ *invokes collector on Main despite changed context in the upstream*


launch(Dispatchers.Main.immediate)
3. Emission
1.flowOf("hey")
2. onEach { doAction() }
3. map { it.lenght }
4. onStart { doAction() }
5. flowOn(Dispatchers.Default)
Emit Hey
Emit Hey
Emit 3
Emit 3
Emit 3
6. flatMapMerge
doAction()
1. Executes action on IO
1. Creation
1. flowOf(1)
Store reference
2. flowOn(Dispatchers.Main)
Store reference
3. onEach { doAction() }
2. Collection
flowOf(1)
Collect
2. flowOn(Dispatchers.Main)
2. Changes context from IO to Main
Collect
1. onEach { doAction() }
3. Emission
3. flowOf(1)
2. flowOn(Dispatchers.Main)
Emit 1



Conclusion

- The `collect()` method is suspended, which forces us to decide in advance on which context the result of our chain will be processed.
- All operators are created from top to bottom and call collect to each other one by one from bottom to top. Thus they form a singly `LinkedList`. Emission goes from top to bottom starting from the root `Flow`.
- Some operators may be executed during the collection stage. For instance, `onStart` is executed as many times as it's written.
- `flowOn` creates a new coroutine transmitting the Dispatcher in arguments and also changes the context. (*However, if we have several flowOns with the same dispatcher, only one coroutine will actually be created.*)
  It affects only the upper streams, thereby guaranteeing compliance with the principle of `Context Preservation`.
  Both the creation of a coroutine and the change of context can be performed at two stages: collection and emission.

- One thread can be used between multiple coroutines. If you have written `flowOn`, a new coroutine will surely be created if the current context is different. However, there is no guarantee that the thread will be different from the thread of the previous coroutine.
- `flatMapMerge/flatMapConcat` starts the chain only during parent chain data emission.
  No actions are performed during the root stream collection process.

In my next post, I'll show how coroutines work under the hood and why they are more efficient than regular threads.

Stay tuned!

Thank you to @kost.maksym, who reviewed the content.

Android Kotlin Kotlin Coroutines