Flows

CS 346 Application Development

https://student.cs.uwaterloo.ca/~cs346

Contents

Introduction

Motivation

A suspending function can execute code asynchronously i.e. it can suspend/resume.

- However, otherwise it behaves like any other function.
- Return values are only delivered after the function completes.

In this example, we build a list of values and then return it once the function completes.

- We cannot deliver the first value early but must wait for the function to be “done”.
- What if we wanted to deliver values as they were produced instead? [1]

```
suspend fun createValues():
List<Int> {
  return buildList {
    add(1)
    delay(1.seconds)
    add(2)
    delay(1.seconds)
    add(3)
    delay(1.seconds)
  }
}

fun main() = runBlocking {
  val list = createValues()
  list.forEach {
    log(it)
  }
}
```

See Kotlin Play demo

Motivation

Reactive programming is a declarative programming paradigm that is based on the idea of asynchronous event processing and data streams.

- As a model, it supports processing data asynchronously and delivering results continuously over a period over time.
- Reactive programming is used in many different areas, such as GUI programming, web programming, microservices [2].

Kotlin flows are a coroutine-based abstraction that supports reactive programming [3].

- Think of it as a “pipeline” architecture with a producer and consumer.
  - One function emits values, and another function consumes them.
  - Like RxJava, but for Kotlin. Leverages coroutines nicely.

Kotlin Flows

Introduction

Flows allow to emit values and process then as they are emitted.

```
fun createValues(): Flow<Int> {
  return flow {
    emit(1)
    delay(1000.milliseconds)
    emit(2)
    delay(1000.milliseconds)
    emit(3)
    delay(1000.milliseconds)
  }
}

fun main() = runBlocking {
  val flow = createValues()
  flow.collect { log(it) }
}
```

Values are emitted as they are produced!

```
29 [main] 1
1040 [main] 2
2045 [main] 3
```

Keywords

- `flow` is a coroutine builder that creates a Flow coroutine.
- `emit` places items into the Flow.
- `collect` the returned value.

Introduction

Cold flows represent data streams that only start emitting items when the items will be consumed and processed by a collector.

- Cold flows are “lazy”.
- If “nobody” is listening, then values won’t be emitted.

Hot flows produce items independently of whether anyone is collecting them i.e., they broadcast “blindly”, and items will be “lost” if no one is listening.

- A collector of a hot flow is called a subscriber.
- You can use hot flows when multiple parts of an application need to react to the same stream of updates, such as incoming chat messages, user actions

Cold Flows

The `flow` keyword creates a cold flow `emitter`.

Normally you define an associated collector for each flow.

- The collector captures emitted values using the collect function.
- The lambda that you pass to the collector is executed every time a value is emitted.
- It only starts emitting when the collector starts collecting.

Cold Flows

```
val letters = flow {
  log("Emitting A!")
  emit("A")
  delay(200.milliseconds)
  log("Emitting B!")
  emit("B")
}

fun main() = runBlocking {
  letters.collect {
    log("Collecting $it")
    delay(500.milliseconds)
  }
}

// 18 [main] Emitting A!
// 25 [main] Collecting A
// 741 [main] Emitting B!
// 742 [main] Collecting B
```

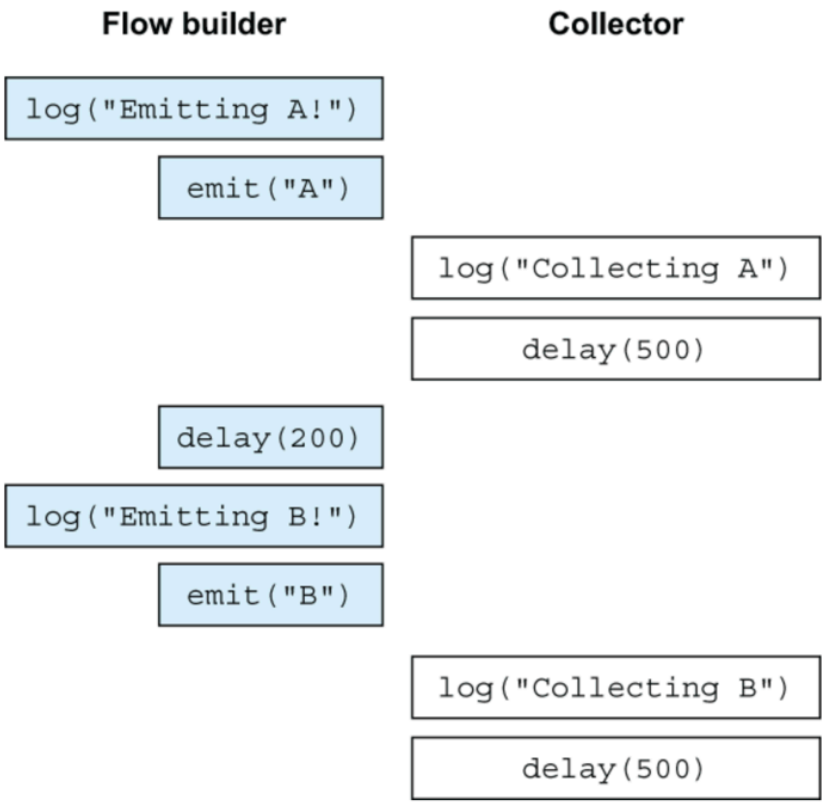

Figure 1: A cold flow won't start until the collector starts to collect the emitted values. Courtesy: Kotlin in Action, 2nd Ed.

Cold Flows

Infinite Flow

Flows can be “infinite”, meaning that they won’t terminate.

- In the case of a cold flow, if you stopped collecting it would stop emitting.

```
val counterFlow = flow {
  var x = 0
  while (true) {
    emit(x++)
    delay(200.milliseconds)
  }
}

launch {
  log("Starting infinite flow")
  counterFlow.collect {
    log("> collected $it")
  }
}
```

```
31 [main] > collected 0
236 [main] > collected 1
438 [main] > collected 2
643 [main] > collected 3
845 [main] > collected 4
1046 [main] > collected 5
1251 [main] > collected 6
```

Cold Flows

Multiple Collectors

A flow will be executed separately for each collector.

```
fun main() = runBlocking {
  letters.collect {
    log("(1) collect $it") // 1
    delay(500.milliseconds)
  }
  letters.collect {
    log("(2) collect $it") // 2
    delay(500.milliseconds)
  }
}
val letters = flow {
  log("Emitting A!"); emit("A")
  delay(200.milliseconds)
  log("Emitting B!"); emit("B")
  }
}
```

```
0 [main] Emitting A!
8 [main] (1) Collecting A
720 [main] Emitting B!
720 [main] (1) Collecting B
1229 [main] Emitting A!
1230 [main] (2) Collecting A
1937 [main] Emitting B!
1937 [main] (2) Collecting B
```

Cold Flows

Cancelling Collectors

If you cancel the collector, you stop collection at the next cancellation point.

```
fun main() = runBlocking {
  val collector = launch {
    cold_flow.collect {
      println(it)
    }
  }
  delay(5.seconds)
  collector.cancel()
}
val cold_flow = flow {
  var x = 0
  while (true) {
    emit(x++)
    delay(200.milliseconds)
  }
}
```

```
// output halts at 24
0
1
2
3
…
24
```

Hot Flows

Hot flows share emitted items across multiple collectors, called subscribers.

- They are suitable when you are emitting events or state changes in your system that happen independently or aren’t tied to a specific collector.
- Emissions happen even with no subscribers present

Kotlin coroutines come with two hot flow implementations out of the box:

- Shared flows, which are used for broadcasting values, and
- State flows, for the special case of communicating state

Hot Flows

Shared Flow

Shared flows broadcast **values**. All subscribers will receive the same values.

```
fun main() = runBlocking {
  Radio().beginBroadcasting(this)
}
class Radio {
  private val _messageFlow = MutableSharedFlow<Int>()
  val messageFlow = _messageFlow.asSharedFlow()
  fun beginBroadcasting(scope: CoroutineScope) {
    scope.launch {
      while(true) {
        delay(500.milliseconds)
        val number = Random.nextInt(0..10)
        log("Emitting $number!")
        _messageFlow.emit(number)
      }
    }
  }
}
```

Hot Flows

Subscriptions

```
fun main() = runBlocking {}
  val radio = RadioStation()
  radio.beginBroadcasting(this)
  delay(600.milliseconds)

  launch {
    radio.messageFlow.collect {
      log("A collecting $it!")
    }
  }

  launch {
    radio.messageFlow.collect {
      log("B collecting $it!")
      }
    }
  }
```

```
559 [main] Emitting 0!
1066 [main] Emitting 9!
1070 [main] A collecting 9!
1071 [main] B collecting 9!
1574 [main] Emitting 0!
1574 [main] A collecting 0!
1575 [main] B collecting 0!
2080 [main] Emitting 0!
2081 [main] A collecting 0!
2081 [main] B collecting 0!
```

Hot Flows

State Flow

A state flow broadcasts a variable's **state**.

output: 3

```
fun main() {
  val vc = ViewCounter()
  vc.increment()
  vc.increment()
  vc.increment()
  println(vc.counter.value)
}

class ViewCounter {
  private val _counter = MutableStateFlow(0) // init
  val counter = _counter.asStateFlow()

  fun increment() {
    _counter.update { it + 1 } // atomic
  }
}
```

Bibliography

[1] S. Aigner, R. Elizarov, S. Isakova, and D. Jemerov, *Kotlin in Action*, Second. New York, USA: Manning Publications, 2024. [Online]. Available: https://www.manning.com/books/kotlin-in-action-second-edition

[2] M. Otta, “What is reactive programming?.” [Online]. Available: https://www.baeldung.com/cs/reactive-programming

[3] D. Leeds, *Kotlin An Illustrated Guide*. USA: TypeAlias Studios LLC, 2025. [Online]. Available: https://typealias.com/start/