Fresher Android

Kotlin Basics – Day 7

FPT | GLOBAL SMART TECHNOLOGIES

Threading Networking Advance

1. Asynchronous Flow

- **Representing multiple values**
- **To represent the stream of values that are being asynchronously computed**

```
fun foo(): Flow<Int> = flow { // flow builder
        for (i in 1..3) {
             delay(100) // pretend we are doing something useful here
            emit(i) // emit next value
      }
}
fun main() = runBlocking<Unit> {
// Launch a concurrent coroutine to check if the main thread is blocked
          launch {
               for (k in 1..3) {
                    println("I'm not blocked $k")
                    delay(100)
               }
          }
          // Collect the flow
         foo().collect { value -> println(value) }
}
```

1. Asynchronous Flow

- Flows are *cold* streams similar to sequences — the code inside a flow builder does not run until the flow is collected.
- Flow cancellation: Flow adheres to the general cooperative cancellation of coroutines. However, flow infrastructure does not introduce additional cancellation points. It is fully transparent for cancellation

1. Asynchronous Flow

- Flow builders
  - ✓ flow {}
  - ✓ flowOf builder that defines a flow emitting a fixed set of values.
  - ✓ Various collections and sequences can be converted to flows using asFlow() extension functions

```
// Convert an integer range to a flow
(1..3).asFlow().collect { value -> println(value) }
```

1. Asynchronous Flow

- Intermediate flow operators
  - ✓ Flows can be transformed with operators, just as you would with collections and sequences. Intermediate operators are applied to an upstream flow and return a downstream flow
  - ✓ The basic operators have familiar names like map and filter.

```
suspend fun performRequest(request: Int): String {
    delay(1000) // imitate long-running asynchronous work
    return "response $request"
}

fun main() = runBlocking<Unit> {
    (1..3).asFlow() // a flow of requests
        .map { request -> performRequest(request) }
        .collect { response -> println(response) }
}
```

1. Asynchronous Flow

- **Intermediate flow operators**
  - ✓ **Transform operator:** initiate simple transformations like map and filter, as well as implement more complex transformations

```
(1..3).asFlow() // a flow of requests
    .transform { request ->
        emit("Making request $request")
        emit(performRequest(request))
    }
    .collect { response -> println(response) }
```

  - ✓ **Size-limiting operators:** cancel the execution of the flow when the corresponding limit is reached

```
fun numbers(): Flow<Int> = flow {
    try {
        emit(1)
        emit(2)
        println("This line will not execute")
        emit(3)
    } finally {
        println("Finally in numbers")
    }
}
fun main() = runBlocking<Unit> {
    numbers()
        .take(2) // take only the first two
        .collect { value -> println(value) }
}
```

1. Asynchronous Flow

- Terminal flow operators: Terminal operators on flows are ***suspending functions*** that start a collection of the flow
  - ✓ The collect operator is the most basic one
  - ✓ Conversion to various collections like toList and toSet.
  - ✓ Operators to get the first value and to ensure that a flow emits a single value.
  - ✓ Reducing a flow to a value with reduce and fold.

```
val sum = (1..5).asFlow()
    .map { it * it } // squares of numbers from 1 to 5
    .reduce { a, b -> a + b } // sum them (terminal operator)
println(sum)
```

1. Asynchronous Flow

- Flows are sequential
  - ✓ Each individual collection of a flow is performed sequentially unless special operators that operate on multiple flows are used

```
(1..5).asFlow()
    .filter {
        println("Filter $it")
        it % 2 == 0
    }
    .map {
        println("Map $it")
        "string $it"
    }.collect {
        println("Collect $it")
    }
```

1. Asynchronous Flow

- **Flow context**

  Collection of a flow always happens in the context of the calling coroutine

  ✓ **Wrong emission withContext**

  Code in the flow {} builder has to honor the context preservation property and is not allowed to emit from different context

  ✓ **flowOn operator**

```
// function that shall be used to change the context of the flow emission
fun foo(): Flow<Int> = flow {
    for (i in 1..3) {
        Thread.sleep(100) // pretend we are computing it in CPU-consuming way
        log("Emitting $i")
        emit(i) // emit next value
    }
}.flowOn(Dispatchers.Default)
// RIGHT way to change context for CPU-consuming code in flow builder
fun main() = runBlocking<Unit> {
    foo().collect { value ->
        log("Collected $value")
    }
}
```

1. Asynchronous Flow

- **Buffering**

```
val time = measureTimeMillis {
    foo()
        .buffer() // buffer emissions, don't wait
        .collect { value ->
            delay(300) // pretend we are processing it for 300 ms
            println(value)
        }
}
println("Collected in $time ms")
```

- **Conflation:** the conflate operator can be used to skip intermediate values when a collector is too slow to process them

```
val time = measureTimeMillis {
    foo()
        .conflate() // conflate emissions, don't process each one
        .collect { value ->
            delay(300) // pretend we are processing it for 300 ms
            println(value)
        }
}
println("Collected in $time ms")
```

Processing the latest value: collectLatest

1. Asynchronous Flow

- **Composing multiple flows**
  - ✓ **Zip:** combines the corresponding values of two flows

```
val nums = (1..3).asFlow() // numbers 1..3
val strs = flowOf("one", "two", "three") // strings
nums.zip(strs) { a, b -> "$a -> $b" } // compose a single string
.collect { println(it) } // collect and print
```

  - ✓ **Combine**
- **Flattening flows**: flows (Flow<Flow<String>>) that needs to be flattened into a single flow. Flatten and flatMap operators for this.
  - ✓ **flatMapConcat:** wait for the inner flow to complete before starting to collect the next one
  - ✓ **flatMapMerge:** concurrently collect all the incoming flows and merge their values into a single flow so that values are emitted as soon as possible
  - ✓ **flatMapLatest:** flattening mode where a collection of the previous flow is cancelled as soon as new flow is emitted

1. Asynchronous Flow

- **Flow exceptions**
  - ✓ **Collector try and catch**

```
fun foo(): Flow<Int> = flow {
    for (i in 1..3) {
        println("Emitting $i")
        emit(i) // emit next value
    }
}

fun main() = runBlocking<Unit> {
    try {
        foo().collect { value ->
            println(value)
            check(value <= 1) { "Collected $value" }
        }
    } catch (e: Throwable) {
        println("Caught $e")
    }
}
```

1. Asynchronous Flow

- **Flow completion**
  - ✓ **Imperative finally block**

```
fun foo(): Flow<Int> = (1..3).asFlow()
fun main() = runBlocking<Unit> {
    try {
        foo().collect { value -> println(value) }
    } finally {
        println("Done")
    }
}
```

  - ✓ **Declarative handling**

```
foo()
    .onCompletion { println("Done") }
    .collect { value -> println(value) }
```

2. Channels

- Deferred values provide a convenient way to transfer a single value between coroutines. Channels provide a way to transfer a stream of values

- Channel basics: A Channel is conceptually very similar to BlockingQueue. One Key difference is that instead of blocking take it has a supending receive

```
val channel = Channel<Int>()
launch {
    // this might be heavy CPU-consuming computation or async logic, we'll just send five squares
    for (x in 1..5) channel.send(x * x)
}
// here we print five received integers:
repeat(5) { println(channel.receive()) }
println("Done!")
```

2. Channels

- Closing and iteration over channels

```
val channel = Channel<Int>()
launch {
    for (x in 1..5) channel.send(x * x)
    channel.close() // we're done sending
}
// here we print received values using `for` loop (until the channel is closed)
for (y in channel) println(y)
println("Done!")
```

2. Channels

- **Pipelines**
  - ✓ **A pipeline is a pattern where one coroutine is producing, possibly infinite, stream of values:**

```
fun CoroutineScope.produceNumbers() = produce<Int> {
    var x = 1
    while (true) send(x++) // infinite stream of integers starting from 1
}
```

  - ✓ **And another coroutine or coroutines are consuming that stream, doing some processing, and producing some other results**

```
fun CoroutineScope.square(numbers: ReceiveChannel<Int>): ReceiveChannel<Int> = produce {
    for (x in numbers) send(x * x)
}
The main code starts and connects the whole pipeline:
val numbers = produceNumbers() // produces integers from 1 and on
val squares = square(numbers) // squares integers
repeat(5) {
    println(squares.receive()) // print first five
}
println("Done!") // we are done
coroutineContext.cancelChildren() // cancel children coroutines
```

2. Channels

- **Fan-out:**
  - ✓ **Multiple coroutines may receive from the same channel, distributing work between themselves**

```
fun CoroutineScope.produceNumbers() = produce<Int> {
    var x = 1 // start from 1
    while (true) {
        send(x++) // produce next
        delay(100) // wait 0.1s
    }
}
```

  - ✓ **Then we can have several processor coroutines**

```
fun CoroutineScope.launchProcessor(id: Int, channel: ReceiveChannel<Int>) = launch {
    for (msg in channel) {
        println("Processor #$id received $msg")
    }
}
```

  - ✓ **launch five processors and let them work for almost a second**

```
val producer = produceNumbers()
repeat(5) { launchProcessor(it, producer) }
delay(950)
producer.cancel() // cancel producer coroutine and thus kill them all
```

2. Channels

- **Fan-in**
  - ✓ **Multiple coroutines may send to the same channel**

```
suspend fun sendString(channel: SendChannel<String>, s: String, time: Long) {
    while (true) {
        delay(time)
        channel.send(s)
    }
}
val channel = Channel<String>()
launch { sendString(channel, "foo", 200L) }
launch { sendString(channel, "BAR!", 500L) }
repeat(6) { // receive first six
    println(channel.receive())
}
coroutineContext.cancelChildren() // cancel all children to let main finish
```

2. Channels

- Buffered channels
  - ✓ Unbuffered channels transfer elements when sender and receiver meet each other (aka rendezvous). If send is invoked first, then it is suspended until receive is invoked, if receive is invoked first, it is suspended until send is invoked.
  - ✓ Both Channel() factory function and produce builder take an optional capacity parameter to specify buffer size.Buffer allows sender to send multiple elements before suspending

```
val channel = Channel<Int>(4) // create buffered channel
val sender = launch { // launch sender coroutine
    repeat(10) {
        println("Sending $it") // print before sending each element
        channel.send(it) // will suspend when buffer is full
    }
}
// don't receive anything... just wait....
delay(1000)
sender.cancel() // cancel sender coroutine
```

2. Channels

Ticker channels: Ticket channel is a special rendezvous channel that produces Unit every time given delay passes since last consumption from this channel

```
fun main() = runBlocking<Unit> {
    val tickerChannel = ticker(delayMillis = 100, initialDelayMillis = 0) // create ticker channel
    var nextElement = withTimeoutOrNull(1) { tickerChannel.receive() }
    println("Initial element is available immediately: $nextElement") // initial delay hasn't passed yet
    nextElement = withTimeoutOrNull(50) { tickerChannel.receive() } // all subsequent elements has 100ms delay
    println("Next element is not ready in 50 ms: $nextElement")
    nextElement = withTimeoutOrNull(60) { tickerChannel.receive() }
    println("Next element is ready in 100 ms: $nextElement")
    // Emulate large consumption delays
    println("Consumer pauses for 150ms")
    delay(150)
    // Next element is available immediately
    nextElement = withTimeoutOrNull(1) { tickerChannel.receive() }
    println("Next element is available immediately after large consumer delay: $nextElement")
    // Note that the pause between `receive` calls is taken into account and next element arrives faster
    nextElement = withTimeoutOrNull(60) { tickerChannel.receive() }
    println("Next element is ready in 50ms after consumer pause in 150ms: $nextElement")
    tickerChannel.cancel() // indicate that no more elements are needed
}
```

Lesson Summary

- Asynchronous Flow
- Channels

Thank you