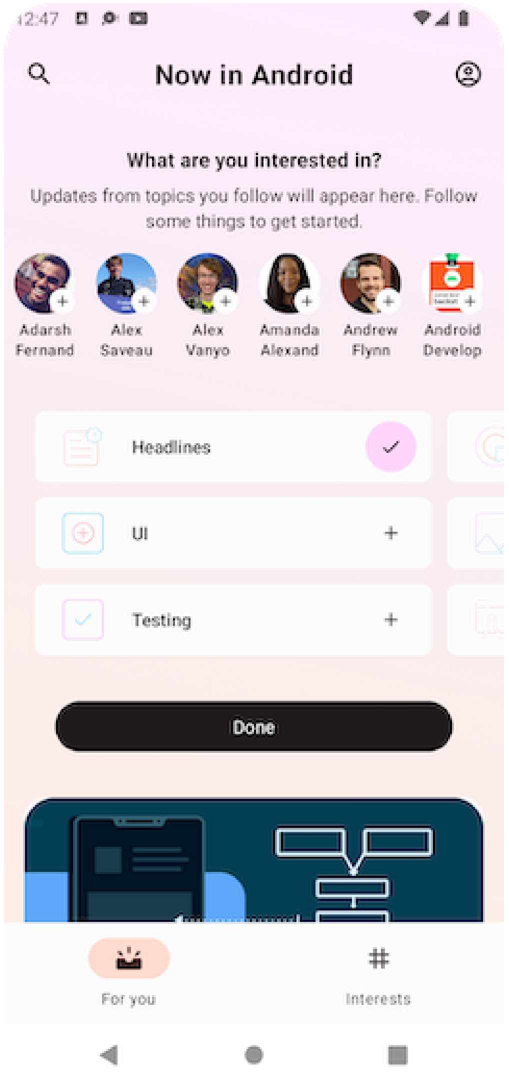
Kotlin Flow and Compose Snapshot

Software Studio 2022

Goal: Reactive Programming

Clone the project "*Now in Android*" from this link



Outline

1. Flow
2. Snapshot
3. Trace code

1. Flow

1. Basic Flow
2. Flow Operator
3. `StateFlow`
4. `combine` Flow

1.1 Basic Flow

```kotlin
val countDownFlow = flow<Int> {
  // Inside coroutine scope, can call suspend function
  val startingValue = 10
  var currentValue = startingValue
  emit(startingValue)
  while (currentValue > 0) {
    delay(1000L)
    currentValue--
    emit(currentValue)
  }
}

private fun collectFlow() {
  // Don't just use it like this in UI layer
  viewModelScope.launch {
    countDownFlow.collect { time ->
      println("The current time is $time")
    }
  }
}
```

1.1 Basic FLow

```
private fun collectFlowLaunchIn() {
  countDownFlow.onEach { time ->
    println(time)
  }.launchIn(viewModelScope)
}
```

```
private fun collectLatestFlow() {
  viewModelScope.launch {
    // Cancel the block if new emit arrived, e.g. show the latest state
    countDownFlow.collectLatest { time ->
      delay(1500L)
      println("The current time is $time")
    }
  }
}
```

1.2 Flow Operator

```kotlin
private fun collectFlow() {
  viewModelScope.launch {
    countDownFlow.filter { time ->
      time % 2 == 0
    }.map { time ->
      time * time
    }.onEach { time ->
      println(time)
    }.collect { time ->
      println("The current time is $time")
    }
  }
}
```

1.3 `StateFlow`

StateFlow is a specialized configuration of SharedFlow optimized for sharing state: the last emitted item is replayed to new collectors, and items are conflated

```kotlin
// Similar to live data, hot flow
private val _stateFlow = MutableStateFlow(0)
val stateFlow = _stateFlow.asStateFlow()

fun incrementCounter() {
  _stateFlow.value += 1
}
```

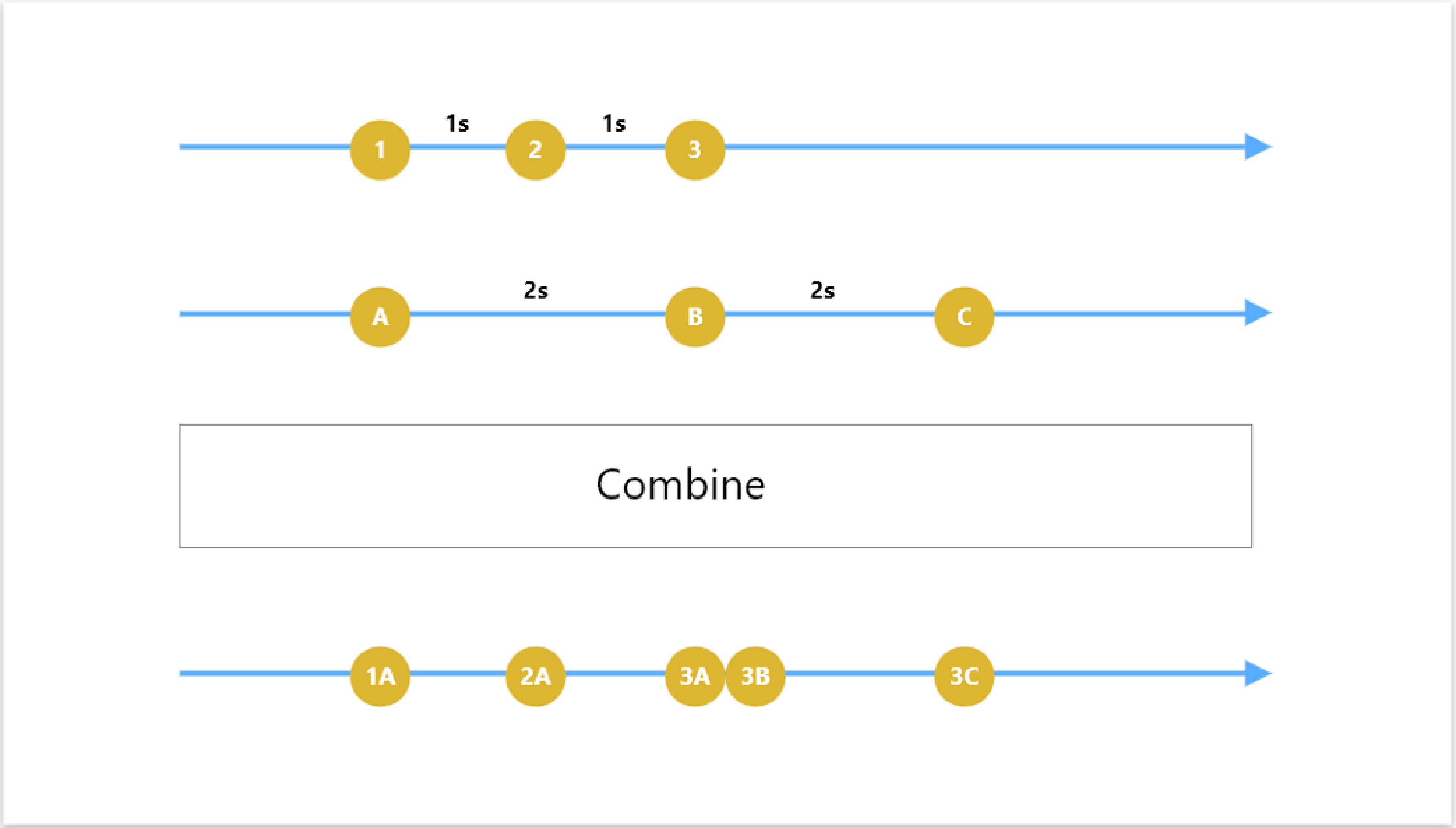
1.3 `StateFlow`

`Flow.stateIn` caches and replays the last emitted item to a new collector.

```kotlin
class LocationRepository(
  private val locationDataSource: LocationDataSource,
  private val externalScope: CoroutineScope
) {
  val locations: Flow<Location> =
    locationDataSource.locationsSource.stateIn(externalScope, WhileSubscribed(), EmptyLocation)
}
```

1.4 `combine` Flow

```
fun flowsWithCombine() = runBlocking {
  val numbersFlow = flowOf(1, 2, 3).delayEach(1000)
  val lettersFlow = flowOf("A", "B", "C").delayEach(2000)
  numbersFlow.combine(lettersFlow) { number, letter ->
    "$number$letter"
  }.collect {
    println(it)
  }
}
```


1s
1s
1
2
3
2s
2s
A
B
C
Combine
1A
2A
3A
3B
3C

2. Snapshot

1. What is Snapshot
2. Mutable Snapshot
3. Tracking reads and writes
4. The global Snapshot
5. Flow + Snapshot

2.1 What is Snapshot

```
fun main() {
  val dog = Dog()
  dog.name.value = “Spot”
  val snapshot = Snapshot.takeSnapshot()
  dog.name.value = “Fido”

  println(dog.name.value)
  snapshot.enter { println(dog.name.value) }
  println(dog.name.value)
}

// Output:
Fido
Spot
Fido
```

2.2 Mutable Snapshot

```
fun main() {
  val dog = Dog()
  dog.name.value = “Spot”
  val snapshot = Snapshot.takeSnapshot()
  dog.name.value = “Fido”

  println(dog.name.value)
  snapshot.enter { println(dog.name.value) }
  println(dog.name.value)
}

// Output:
Fido
Spot
Fido
```

2.2 Mutable Snapshot

- After calling `apply()` the new value is applied to the direct parent snapshot

```
fun main() {
  val dog = Dog()
  dog.name.value = "Spot"

  val snapshot = Snapshot.takeMutableSnapshot()
  snapshot.enter {
    dog.name.value = "Fido"
    println(dog.name.value)
  }
  snapshot.apply()
  println(dog.name.value)
}

// Output:
Fido
Fido
```

2.2 Mutable Snapshot

Use `withMutableSnapshot` to achieve the same result

```kotlin
fun main() {
  val dog = Dog()
  dog.name.value = "Spot"

  Snapshot.withMutableSnapshot {
    dog.name.value = "Fido"
    println(dog.name.value)
  }
  println(dog.name.value)
}
```

2.3 Tracking reads and writes

```kotlin
fun main() {
  val dog = Dog()
  dog.name.value = "Spot"

  val readObserver: (Any) -> Unit = { readState ->
    if (readState == dog.name) println("dog name was read")
  }
  val writeObserver: (Any) -> Unit = { writtenState ->
    if (writtenState == dog.name) println("dog name was written")
  }

  val snapshot = Snapshot.takeMutableSnapshot(readObserver, writeObserver)
  println("name before snapshot: " + dog.name.value)
  snapshot.enter {
    dog.name.value = "Fido"
    println("name before applying: ")
    // This could be inlined, but I've separated the actual state read
    // from the print statement to make the output sequence more clear.
    val name = dog.name.value
    println(name)
  }
  snapshot.apply()
  println("name after applying: " + dog.name.value)

}
```

2.4 The global snapshot

```
fun main() {
  val dog = Dog()
  Snapshot.registerApplyObserver { changedSet, snapshot ->
    if (dog.name in changedSet) println("dog name was changed")
  }

  println("before setting name")
  dog.name.value = "Spot"
  println("after setting name")

  println("before sending apply notifications")
  Snapshot.sendApplyNotifications()
  println("after sending apply notifications")
}

// Output:
before setting name
after setting name
before sending apply notifications
dog name was changed
after sending apply notifications
```

2.5 Flow + Snapshot

- `snapshotFlow`: convert `State` objects into a cold Flow, run its block when collected.

```kotlin
val listState = rememberLazyListState()

LazyColumn(state = listState) {
    // ...
}

LaunchedEffect(listState) {
  snapshotFlow { listState.firstVisibleItemIndex }
    .map { index -> index > 0 }
    .distinctUntilChanged()
    .filter { it == true }
    .collect {
      MyAnalyticsService.sendScrolledPastFirstItemEvent()
    }
}
```

3. Trace Code

Try to answer the following questions:

Q1: How many states does `feedState` observe on?

Q2: What is the value of the `feedState` when `followedInterestState` is still processing?
Q3: What is the value of the `feedState` when `followedInterestState` finishes processing?
Q4: What is the detail process from selecting a topic to showing the UI change?

3. Trace Code

The files to focus on:

- *ForYouScreen.kt*
- *ForYouViewModel.kt*
- *OfflineFirstNewsRepository.kt*
- *NewsResourceDao.kt*

Reference

- How to Combine Kotlin Flows
- Migrating from LiveData to Kotlin's Flow
- Side-effects in Compose
- Introductino to the Compose Snapshot System
- 一文看懂 Jetpack Compose 快照系统

Other great resources

Overview

- Offical Document
- Compose Tutorial (YouTube Playlist)

Layout

- Deep dive into Jetpack Compose layouts
- Lazy layouts in Compose

Performance

- Performance best practices for Jetpack Compose