Coroutines

Contenido

- Coroutines
- Scope
- Jobs
- Dispatcher
- Integraciones con ViewModel
- Llamando desde iOS

Coroutines

Coroutines

¿Qué son?

Son esencialmente light-weight (livianos) threads que nos permiten escribir ***código asincrónico*** de una manera ***secuencial***

¿Son realmente más livianas que los threads?

Coroutines

Comparación: Coroutines

```
@Test
fun `Create 100_000 coroutines`() = runBlockingTest {
    val time = measureTimeMillis {
        val jobs = List(100_000) {
            launch {
                delay(10_000L)
                println("$it")
            }
        }
        jobs.forEach { it.join() }
    }
    println("Time for coroutines: $time")
}
```

≈ 2150 ms*

**delay es ignorado dentro de runBlockingTest { }*

Coroutines
Comparación: Threads

```
@Test
fun `Create 100_000 threads`() {
    val time = measureTimeMillis {
        val jobs = List(100_000) {
            thread {
                Thread.sleep(10_000L)
                println("$it")
            }
        }
        jobs.forEach { it.join() }
    }
    println("Time for threads: $time")
}
```

java.lang.OutOfMemoryError: unable to create new native thread

¿Son realmente más livianas que los threads?

Coroutines

¿Qué características tienen?

- Su programación es más sencilla, de manera secuencial (¡¡no más callbacks!! 🎉)
- Son una implementación a nivel lenguaje => El SO no las conoce
- Permiten ejecutar suspending functions

Coroutines

Ejemplo

Suspending functions

Coroutines
Suspending functions

- Son funciones que se pueden *suspender* y *continuar* más adelante
- Se ejecutan dentro de una coroutine o dentro de otra suspending function
- Deben ser seguras, capaces de ser llamadas **sin bloquear el thread que las invoca**
- Internamente, el compilador traduce el modificador *suspend* en una función con **callbacks** utilizando una **máquina de estados finitos**

Coroutines
Suspending functions

Suspend & Resume

Este ejemplo muestra una supending function *fetchDocs()* que se comunica con un servicio externo para obtener un documento y luego se muestra el mismo en la UI.

La función get("...") es también una suspending function

Scope

Coroutines

Scope: ¿Para qué usarlos?

- El scope define el **contexto** sobre el que van a ejecutarse las coroutines
- Las coroutines lanzadas por el scope heredan su contexto
- Toda aplicación tiene al menos un scope asociado, llamado GlobalScope, que permite ejecutar coroutines durante todo el tiempo de vida de la aplicación

Coroutines

Scope: ¿Cómo creamos coroutines?

Existen *coroutines builders* declarados como extension functions del scope.

Los 2 más comunes son launch { } y async { }

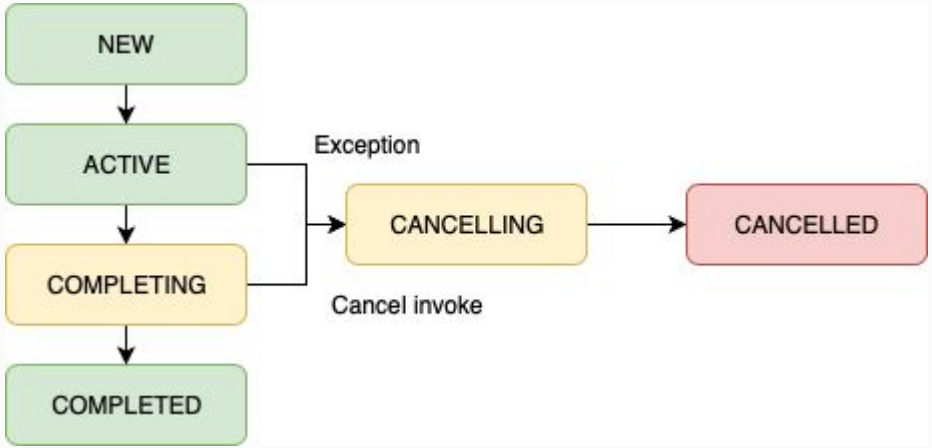
- launch { }: Crea una nueva coroutine sin bloquear el thread actual y devuelve una referencia a la coroutine como un Job
- async { }: Crea una nueva coroutine sin bloquear el thread actual y nos devuelve una referencia a un Deferred value (que es un Job también). Se utiliza para obtener un valor como resultado de una ejecución, mediante un llamado a .await().

Existe una función withContext() que nos permite cambiar de dispatcher, pero sin crear una nueva coroutine

Jobs

Jobs

- Un Job representa a una coroutine
- Es el encargado del ciclo de vida, cancelación y relación parent-child de las coroutines



Dispatcher

Dispatcher

- Determina que thread o threads va a utilizar la coroutine para su ejecución
- Existen varios dispatchers que podemos utilizar:
  - Dispatchers.Default: Es el utilizado por defecto si no se especifica otro. Está respaldado por un pool de threads compartidos.
  - Dispatchers.Main: Está relacionado al Main Thread que opera con objetos de UI. Depende de la plataforma, hay que agregar la dependencia correspondiente.
  - Dispatchers.Unconfined: No limita a las coroutines a ningún thread en particular. Continúa en el thread de quien llamó a su resume().
  - Dispatchers.IO: Utilizado para realizar operaciones de Entrada/Salida mediante un pool de threads compartido.

Integraciones en Android

Integraciones en Android

Existe una extensión de Kotlin que nos provee un *scope* asociado al ciclo de vida de un ViewModel. Esto nos da una ventaja importante, que es la de cancelar las coroutines cuando este ViewModel se destruye.

Para ello basta con incluir la dependencia

```
implementation "androidx.lifecycle:lifecycle-viewmodel-ktx:$version"
```

Ej:

```
class MyViewModel(private val profileService: ProfileService) : ViewModel() {

    val profile = MutableStateFlow<Profile?>(null)

    fun showMyProfile() {
        viewModelScope.launch {
            val myProfile = profileService.findMyProfile()
            profile.value = myProfile
        }
    }
}
```

Llamando desde iOS

Llamando desde iOS

- En iOS no existen las corrutinas, por lo tanto no se puede llamar una función *suspend* “directamente”
- Sin embargo, las funciones suspend se traducen como funciones con callbacks o *completion handlers* (en terminología de Swift/Objective-C).

Llamando desde iOS

Por ejemplo, la siguiente función suspend de Kotlin:

```
@Throws(Exception::class)
suspend fun getAllProducts(): List<Product> {
    val products = productsRepository.findAll()
    return products
}
```

Puede ser llamada desde Swift como:

```
productsService.getAllProducts { list, error in
    if let products = list {
        self.productsList = products
    }
    if let error = error {
       // Manejar caso de error
    }
}
```

Llamando desde iOS

- A partir de Swift 5.5, tenemos la posibilidad de usar un mecanismo llamado async await, muy similar a las corrutinas de Kotlin.
- Podemos entonces transformar el callback generado por Kotlin en una task asincrónica de Swift de la siguiente manera:

```
func getProductsAsync() async -> [Product] {
    return await withCheckedContinuation{ continuation in
        productsService.getAllProducts { list, error in
            if let products = list {
                continuation.resume(returning: products)
            }
            if let errorReal = error {
                continuation.resume(throwing: mapError(errorReal))
            }
        }
    }
}
```

- Y por último llamarlo:

```
async {
    self.products = await getProductsAsync()
}
```

Dependencias

- En el módulo compartido, debemos agregar la siguiente dependencia en *commonMain*:

```
val commonMain by getting {
    dependencies {
        implementation("org.jetbrains.kotlinx:kotlinx-coroutines-core:1.6.0-native-mt") {
            version {
                // Necesario para evitar que otras dependencias reemplacen coroutines, por ej: Ktor
                strictly("1.6.0-native-mt")
            }
        }
    }
}
```

Links

- Processes and threads overview
- Coroutines Guide
- Use Kotlin Coroutines in your Android App
- Roman Elizarov – Medium
- Corrutinas en Kotlin 1.3: suspending functions, contexts, builders y scope
- Concurrency and coroutines in KMM
- Completion handler & async en iOS
- Concurrency in Swift (async await Swift 5.5)
- Swift async await and Kotlin coroutines

Algo práctico:

- Kotlin Hands On: Coroutines and Channels