Programación asíncrona en Android

Thread en las coroutines

{desafío} latam_

Utilizar elementos de la programación asíncrona acorde al lenguaje Kotlin para dar solución a un problema

- Unidad 1: Kotlin para el desarrollo de aplicaciones.
- Unidad 2: Ciclo de vida de componentes.
- Unidad 3: Arquitectura en Android.
- Unidad 4: Programación asíncrona en Android.

Te encuentras aquí

{desafío} latam_

¿Qué aprenderás en esta sesión?

- *Utilizar Coroutines para el manejo asíncrono acorde al lenguaje Kotlin*

¿Sabías que puedes especificar en qué thread se ejecuta una coroutine?

/* Thread en las coroutines */

Dispatchers & threads en detalle

Ya mencionamos anteriormente que el contexto de coroutine incluye un **dispatcher** de coroutine el cual determina qué subproceso o subprocesos utiliza la coroutine correspondiente para su ejecución.

El dispatcher o despachador de coroutine puede limitar la ejecución de coroutines a un subproceso específico, enviarlo a un grupo de subprocesos o dejar que se ejecute sin restricciones.

Todos los constructores de coroutine como `launch` y `async` aceptan un parámetro `CoroutineContext` opcional que se puede usar para especificar explícitamente el despachador para el nuevo coroutine y otros elementos de contexto.

Ejemplos Dispatchers & threads

A continuación podemos ver algunos ejemplos de cómo usar los dispatchers para indicar en qué thread correrá un proceso

- `viewModelScope.launch(Dispatchers.IO) {  }`
- `lifecycleScope.launch(Dispatchers.Main) {  }`
- `CoroutineScope(Dispatchers.Default).launch { }`
- `async(Dispatchers.Unconfined) { }`
- `withContext(Dispatchers.Main){ }`

Tipos de Dispatchers

Dispatcher	Descripción	Uso
`Dispatchers.Main`	• Hilo principal o Main Thread	• Llamado de suspend functions • Llamado de UI • Actualizacion de StateFlow o LiveData
`Dispatchers.IO`	• Lectura y escritura de disco • Base de datos • Network	• Comunicación con bases de datos • Comunicación API / Networking • Lectura y escritura de archivos
`Dispatchers.Default`	• Trabajos intensos en CPU	• Ordenamiento de listas o arreglos y otros algoritmos • Mapeo de json • DiffUtils (RecyclerViews)

Dispatcher.Main

Dispatcher.Main es un dispatcher de corrutina que se limita al subproceso principal que opera con objetos de interfaz de usuario. Por lo general, dicho dispatcher es de un solo subproceso.

Ejemplo:

```
fun main() = runBlocking<Unit> {
    launch {
        println("Thread: ${Thread.currentThread().name}")
    }
}
```

Resultado:

```
Thread: main
```

Dispatcher.Default

Es el `CoroutineDispatcher` predeterminado que utilizan todos los constructores estándar como `launch`, `async`, etc. si no se especifica un despachador ni ningún otro `ContinuationInterceptor` en su contexto.

Está respaldado por un grupo compartido de subprocesos en JVM. De forma predeterminada, la cantidad máxima de subprocesos utilizados por este despachador es igual a la cantidad de núcleos de CPU, pero es al menos dos.

Ejemplo:

```
fun main() = runBlocking<Unit> {
    launch(Dispatchers.Default) {
        println("Thread: ${Thread.currentThread().name}")
    }
}
```

Resultado:

```
Thread: DefaultDispatcher-worker-1
```

Dispatcher.IO

Es el CoroutineDispatcher que está diseñado para descargar tareas de I/O de bloqueo en un grupo compartido de subprocesos.

Se crean subprocesos adicionales en este grupo y se cierran a pedido. La cantidad de subprocesos utilizados por las tareas en este despachador está limitada por el valor de la propiedad del sistema "kotlinx.coroutines.io.parallelism" (IO_PARALLELISM_PROPERTY_NAME).

El valor predeterminado es el límite de 64 subprocesos o el número de núcleos (el que sea mayor).

Dispatcher.IO - Elasticidad para paralelismo limitado

Dispatchers.IO tiene una propiedad única de elasticidad:

Sus vistas obtenidas con `CoroutineDispatcher.limitedParallelism` no están restringidas por el paralelismo de Dispatchers.IO. Conceptualmente, hay un despachador respaldado por un conjunto ilimitado de subprocesos, y tanto Dispatchers.IO como las vistas de Dispatchers.IO son en realidad vistas de ese despachador. En la práctica, esto significa que, a pesar de no cumplir con las restricciones de paralelismo de Dispatchers.IO, sus vistas comparten hilos y recursos con él.

Dispatcher.IO - Elasticidad para paralelismo limitado

// **100** threads para la conexión de DBEngine1

```
val dbEngine1Dispatcher = Dispatchers.IO.limitedParallelism(100)
```

// **60** threads para la conexión de DBEngine2

```
val dbEngine2Dispatcher = Dispatchers.IO.limitedParallelism(60)
```

El sistema puede tener hasta 64 + **100** + **60** subprocesos dedicados a tareas de bloqueo durante picos de carga, pero durante su estado estable solo hay una pequeña cantidad de subprocesos compartidos entre **`Dispatchers.IO`**, **`dbEngine1Dispatcher`** y **`dbEngine2Dispatcher`**.

Dispatchers - Unconfined vs confined

El despachador de coroutine **`Dispatchers.Unconfined`** inicia una coroutine en el subproceso que lo llama, pero solo hasta el primer punto de suspensión. Después de la suspensión, se reanuda la coroutine en el subproceso que está completamente determinado por la función de suspensión que se invocó. El despachador no confinado es apropiado para corrutinas que no consumen tiempo de CPU ni actualizan datos compartidos (como la interfaz de usuario) confinados a un subproceso específico.

Por otro lado, el despachador se hereda del CoroutineScope externo de forma predeterminada. El despachador predeterminado para la corrutina runBlocking, en particular, se limita al subproceso del invocador, por lo que heredarlo tiene el efecto de limitar la ejecución a este subproceso con una programación FIFO predecible.

Dispatcher.Unconfined

Es un dispatcher que no está confinado a ningún subproceso específico.

Ejecuta la continuación inicial de una coroutine en el marco de llamada actual y permite que la coroutine se reanude en cualquier subproceso que utilice la función de suspensión correspondiente, sin exigir ninguna política específica de subprocesos. Las coroutines anidadas lanzadas en este despachador forman un bucle de eventos para evitar desbordamientos de pila (stack overflows).

```
withContext(Dispatchers.Unconfined) {
   println(1)
   withContext(Dispatchers.Unconfined) { // unconfined anidado
       println(2)
   }
   println(3)
}
println("Done")
```

Puede imprimir tanto "1 2 3" como "1 3 2". Este es un detalle de implementación que se puede cambiar. Sin embargo, se garantiza que "Done" se imprimirá solo cuando se completen ambas llamadas withContext.

Demostración
"Dispatchers en coroutines"

{desafío}
latam_

Dispatchers

En el ejercicio de la Guía - Programación asíncrona en Android (II), realizamos una consulta a una base de datos con el siguiente código:

```
init {
    viewModelScope.launch {
        repository.getTasks().collectLatest {
            _data.value = it
        }
    }
}
```

Ahora vamos a mejorar ese código de acuerdo a lo aprendido

```
private val dispatcherIO : CoroutineDispatcher =
Dispatchers.IO
init {
    viewModelScope.launch(dispatcherIO) {
        repository.getTasks().collectLatest {
            _data.value = it
        }
    }
}
```

Recuerda que si no especificas en que thread quieres que se ejecute una tarea, por defecto se hará en Default lo cual puede conllevar a un crash

Próxima sesión...

- *Guía de ejercicios*

{desafío}
latam_

{desafío}
latam_

Academia de
talentos digitales