Programación asíncrona en Android

Job 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*

¿Recuerdas qué son las coroutines?

{desafío} latam_

/* El Job en las coroutines */

¿Qué es Job en Coroutines?

Un Job es un identificador de una coroutine. Cada coroutine que crea con `launch` o `async` devuelve una instancia de `Job` que identifica de forma única la coroutine y administra su ciclo de vida. También puede pasar un Job a un `CoroutineScope` para administrar aún más su ciclo de vida, como se muestra en el siguiente ejemplo:

```
class ExampleClass {
    fun exampleMethod() {
        val job = scope.launch {
            // Nueva coroutine
        }

        if (...) {
            // si las condiciones son las necesarias, entonces se puede cancelar la coroutine
            job.cancel()
        }
    }
}
```

{desafío} latam_

Importante: Conceptualmente, la ejecución de un trabajo no produce un valor de resultado. Los trabajos se lanzan únicamente por sus efectos secundarios.

Job States

State	isActive	isCompleted	isCancelled
New (optional initial state)	false	false	false
Active (default initial state)	true	false	false
Completing (transient state)	true	false	false
Cancelling (transient state)	false	false	true
Cancelled (final state)	false	true	true
Completed (final state)	false	true	false

¿Cómo crear un Job?

Del ejemplo anterior podemos ver que una coroutine retorna un job, otro ejemplo más simple podría ser:

```
private val dispatcherIO: CoroutineDispatcher = Dispatchers.IO
fun someCoroutineFunction(){
    val job = viewModelScope.launch(dispatcherIO) {
    }
}
```

¿Cómo crear un Job?

Otra forma de crear un Job podría ser así:

```
val job = Job()
```

Ahora podemos tomar ese objeto y usarlos una coroutine

```
CoroutineScope(dispatcherIO + job).launch
{
  ...
}
```

Uno de los beneficios de especificar el job en una coroutine es al momento de cancelar la ejecución del Job, por ejemplo, si tomamos el valor que retorna `CoroutineScope(dispatcherIO + job)` y lo cancelamos, este solo afectara al que se especifica en la función.

Ejemplos de Job Cancellation

```
val job1 = Job()
val job2 = CoroutineScope(dispatcherIO + job1).launch {
    // ...
}
job2.cancel()

val job3 = CoroutineScope(dispatcherIO).launch {
// ...
}
job3.cancel()
```

Según lo que se explicó anteriormente, al cancelar **`job2`**, se cancela solamente **`job1`**, en cambio, **`job3`** cancela todos los **`jobs`**

Causa de cancelación

Se dice que un `Job` de coroutine se completa excepcionalmente cuando su cuerpo genera una excepción; un `CompletableJob` se completa excepcionalmente llamando a `CompletableJob.completeExceptionally`.

Un `Job` excepcionalmente se cancela y la excepción correspondiente se convierte en la causa de cancelación del trabajo.

Causa de cancelación

La cancelación normal de un `Job` se distingue de su falla por el tipo de esta excepción que causó su cancelación. Una coroutine que lanzó `CancellationException` se considera cancelada normalmente.

Si una causa de cancelación es un tipo de excepción diferente, se considera que el `Job` ha fallado. Cuando un `Job` falla, el `Job` padre se cancela con la excepción del mismo tipo, lo que garantiza la transparencia al delegar partes del `Job` a sus hijos.

/* SupervisorJob */

SupervisorJob

SupervisorJob: crea un objeto "*supervisor*" de `Job` en un estado activo. Los hijos de un de `SupervisorJob` pueden fallar independientemente uno del otro.

Una falla o cancelación de un hijo no hace que el `SupervisorJob` falle y no afecta a sus otros hijos, por lo que un supervisor puede implementar una política personalizada para manejar las fallas de sus hijos:

- Una falla de un job secundario que se creó usando el **`launch`** se puede manejar a través de `CoroutineExceptionHandler` en el contexto.

- Una falla de un `Job` secundario que se creó usando **`async`** se puede manejar a través de `Deferred.await` en el valor diferido resultante.

Si se especifica un `Job` principal, este `SupervisorJob` se convierte en un job secundario de su principal y se cancela cuando su principal falla o se cancela. Todos los hijos de este supervisor también están cancelados en este caso. La invocación de cancelar con excepción (que no sea CancellationException) en este`SupervisorJob` también cancela el padre.

Demostración "Job"

Job

Podemos ejecutar más de un job al mismo tiempo y a su vez esperar que ambos completen antes de mostrar algún resultado al usuario

```
val job1 = coroutineScope.launch {
    delay(3000L)
    Log.d(TAG, "startCoroutine - job1")
}

val job2 = coroutineScope.launch {
    delay(3000L)
    Log.d(TAG, "startCoroutine - job2")
}
```

Job

Luego podemos usar `invokeOnCompletation{}` para mostrar un mensaje al usuario o actualizar UI

```
job1.invokeOnCompletion {
            it?.message.let { msg ->
                if (msg.isNullOrBlank()) {
                    Log.d(TAG, "invokeOnCompletion - Ok")
                } else {
                    Log.d(TAG, "invokeOnCompletion - Error
$msg")
                }
            }
        }
```

Job

Finalmente, apoyándonos con el ciclo de vida del viewModel, podemos cancelar el job.

```
override fun onCleared() {
        super.onCleared()
        parentJob.cancel()
    }
```

Job - Código completo

```
class MainViewModel : ViewModel() {

    private val parentJob = SupervisorJob()
    private val coroutineScope = CoroutineScope(parentJob + Dispatchers.Default)

    fun startCoroutine() {
        val job1 = coroutineScope.launch {
            delay(3000L)
            Log.d(TAG, "startCoroutine - job1")
        }

        val job2 = coroutineScope.launch {
            delay(3000L)
            Log.d(TAG, "startCoroutine - job2")
        }

        job1.invokeOnCompletion {
            it?.message.let { msg ->
                if (msg.isNullOrBlank()) {
                    Log.d(TAG, "invokeOnCompletion - Ok")
                } else {
                    Log.d(TAG, "invokeOnCompletion - Error $msg")
                }
            }
        }

        Log.d(TAG, "job1 status = ${job1.notifyJobStatus()}")
        Log.d(TAG, "job2 status = ${job2.notifyJobStatus()}")
    }

    private fun Job.notifyJobStatus(): String = when {
        isCancelled -> "cancelado"
        isActive -> "activo"
        isCompleted -> "completado"
        else -> ""
    }

    override fun onCleared() {
        super.onCleared()
        parentJob.cancel()
    }
}
```

{desafío} latam_

Los Jobs son mucha ayuda cuando necesitas controlar una tarea que corre en una coroutine y que puede tomar tiempo en completar

Próxima sesión...

- *Thread en las coroutines*

{desafío}
latam_

Academia de
talentos digitales