Programación asíncrona en Android

Threads (Parte I)

{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?

- *Explicar el concepto de programación asíncrona reconociendo el problema que resuelve y los mecanismos disponibles en Android*

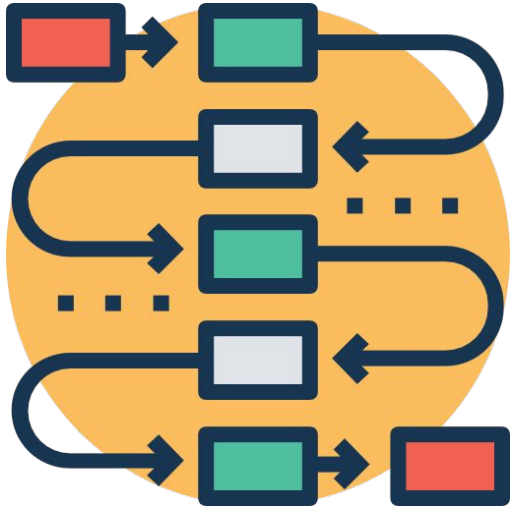
¿Sabías que una aplicación puede realizar varias al mismo tiempo?

/* Qué son los threads */

¿Qué son los threads?

- Un thread o hilo, es la secuencia más pequeña de instrucciones programadas que un planificador puede gestionar de forma independiente, que normalmente forma parte del sistema operativo.

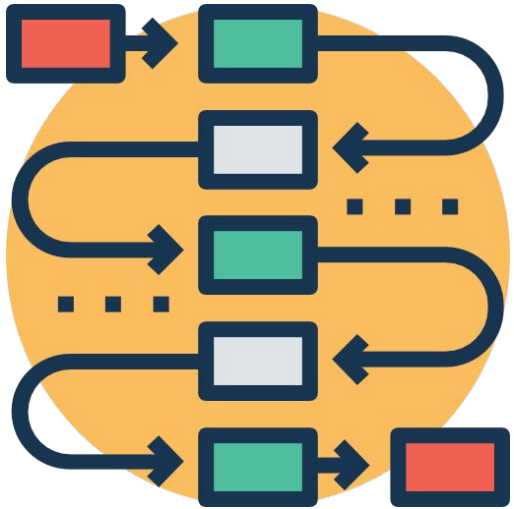
- La implementación de subprocesos y procesos difiere entre sistemas operativos, pero en la mayoría de los casos, un subproceso es un componente de un proceso.
.



¿Qué son los threads?

- Los subprocesos múltiples de un proceso dado pueden ejecutarse simultáneamente (a través de capacidades de subprocesos múltiples), compartiendo recursos como la memoria, mientras que diferentes procesos no comparten estos recursos.

- En particular, los subprocesos de un proceso comparten su código ejecutable y los valores de sus variables asignadas dinámicamente y las variables globales no locales del subproceso en un momento dado.



/* Threads y cómo se manejan */

Uso de Threads en Kotlin

Runnable:

```
class SimpleRunnable: Runnable {
    public override fun run() {
        println("${Thread.currentThread()} has run.")
    }
}
```

Thread:

```
val thread = Thread {
    println("${Thread.currentThread()} has run.")
}
thread.start()
```

Threads y cómo se manejan

Para crear un Thread basta con crear una clase la cual extiende de Thread o Runnable:

Otro ejemplo con Runnable:

```
fun r(f: () -> Unit): Runnable = object : Runnable {override fun run() {f()}}
```

En Kotlin también podemos crear funciones que retornan un Thread:

```
fun thread(
    start: Boolean = true,
    isDaemon: Boolean = false,
    contextClassLoader: ClassLoader? = null,
    name: String? = null,
    priority: Int = -1,
    block: () -> Unit
): Thread
```

/* Diferencias entre AsyncTasks, Service y cronjobs en Android */

{desafío}
latam_

AsyncTasks

- AsyncTask estaba destinado a permitir un uso adecuado y fácil del subproceso de la interfaz de usuario o UI Thread. Sin embargo, el caso de uso más común fue para la integración en la interfaz de usuario, y eso causaría **fugas de contexto**, devoluciones de llamada perdidas o fallas en los cambios de configuración.

- También tiene un comportamiento inconsistente en diferentes versiones de la plataforma, acepta excepciones de doInBackground y no proporciona mucha utilidad sobre el uso directo de Executors.

AsyncTasks ejemplo (Java):

Ya no se recomienda usar AsyncTask, sin embargo, el siguiente ejemplo sirve como base para tener una idea de cómo funciona una clase que crea un nuevo thread para realizar una operación, en este caso descargar archivos.

Se define una clase que extiende de AsyncTask y se definen los parámetros que se usarán en cada uno de los métodos:

- **`doInBackground:`** ejecuta la tarea principal, en este caso recibe como parámetro una lista de URL.
- **`onProgressUpdate:`** con cada update de la tarea se actualiza el valor de progress, usualmente se usa para mostrar el porcentaje de descarga.
- **`onPostExecute:`** una vez finalizada la tarea se llama este método, para mostrar un mensaje al usuario.

```
private class DownloadFilesTask extends AsyncTask<URL, Integer, Long> {
    protected Long doInBackground(URL... urls) {
        int count = urls.length;
        long totalSize = 0;
        for (int i = 0; i < count; i++) {
            totalSize += Downloader.downloadFile(urls[i]);
            publishProgress((int) ((i / (float) count) * 100));
            // Escape early if cancel() is called
            if (isCancelled()) break;
        }
        return totalSize;
    }

    protected void onProgressUpdate(Integer... progress) {
        setProgressPercent(progress[0]);
    }

    protected void onPostExecute(Long result) {
        showDialog("Downloaded " + result + " bytes");
    }
}
```

Services

- Un **Service** es un componente de una aplicación que puede realizar operaciones de larga ejecución en segundo plano y que no proporciona una interfaz de usuario.

- Otro componente de la aplicación puede iniciar un servicio y continuar ejecutándose en segundo plano, aunque el usuario cambie a otra aplicación. Además, un componente puede enlazarse con un servicio para interactuar con él e incluso realizar una comunicación entre procesos (IPC).

- Por ejemplo, un servicio puede manejar transacciones de red, reproducir música, realizar I/O de archivos o interactuar con un proveedor de contenido, todo en segundo plano.

Tipos Services

Existen 3 tipos de services:

- **Foreground**: o servicio en primer plano, realiza alguna operación que es perceptible para el usuario. Por ejemplo, una aplicación de audio usaría un servicio de primer plano para reproducir audio.
- **Background**: o servicio en segundo plano, realiza una operación que el usuario no nota directamente. Por ejemplo, si una aplicación usa un servicio para compactar su almacenamiento, normalmente será un servicio en segundo plano

IMPORTANTE: Los servicios en primer plano deben mostrar una notificación. Los servicios de primer plano continúan ejecutándose incluso cuando el usuario no está interactuando con la aplicación.

Tipos Services

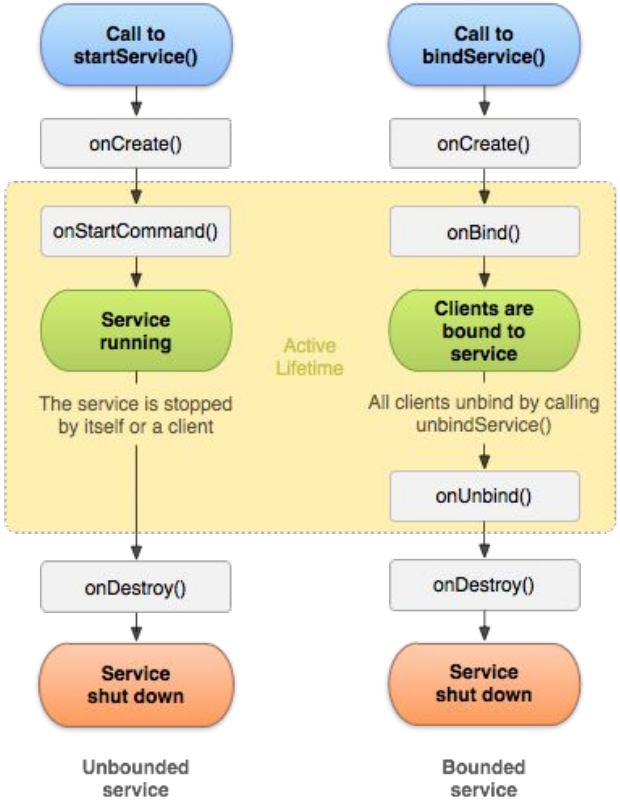
- **Bound**: Un servicio está enlazado cuando un componente de la aplicación lo enlaza llamando a `bindService()`.

  Un servicio enlazado ofrece una interfaz cliente-servidor que permite que los componentes interactúen con el servicio, envíe solicitudes, reciban resultados e incluso lo hagan a través de procesos con comunicación entre procesos (IPC). Un servicio vinculado se ejecuta solo mientras otro componente de la aplicación esté vinculado a él. Varios componentes pueden vincularse al servicio a la vez, pero cuando todos se desvinculan, el servicio se destruye.

Services Lifecycle

En el ciclo de vida de los servicios se ven los métodos de devolución de llamada habituales de un servicio. Si bien en la figura se separan los servicios que se crean mediante `startService()` de los que se crean mediante `bindService()`, recuerda que cualquier servicio, sin importar cómo se inicie, puede potencialmente permitir que se enlacen clientes a él.

Por lo tanto, un servicio que al comienzo se inició con `onStartCommand()` (de parte de un cliente que llamó a `startService()`) puede recibir una llamada a `onBind()` (cuando un cliente llama a `bindService()`).



{desafío} latam_

Demostración
"Hello world (Service)"

Hello world (Service)

Para ver cómo funciona un servicio, puedes descargar el **Material de apoyo - Threads (Parte I).**

La forma en que se llama a ejecutar el servicio puede ser tan simple como, por ejemplo, con un botón:

```
Intent(this,
HelloService::class.java).also {
    startService(it)
}
```

Para entender el código del servicio, se puede dividir en dos partes:

- El servicio y los métodos requeridos
- El servicio y la inner class.

El servicio y los métodos requeridos

Hello World Service parte 1

Como se explicó anteriormente, el servicio debe ser una clase la cual extiende de Service()

```
class HelloService : Service() {}
```

- **onCreate()**: define la creación del servicio
- **onStartCommand()**: define lo que ocurre cuando el servicio inicia, también retorna el tipo de servicio, en nuestro caso “START_STICKY”
- **onBind()**: esta opción se usa cuando el tipo de servicio es Bound
- **onDestroy()**: se llama a este método una vez la app que ejecuta el servicio es terminada

El servicio y los métodos requeridos

Hello World Service parte 1

```
private var serviceLooper: Looper? = null
private var serviceHandler: ServiceTask? = null

override fun onCreate() {
    HandlerThread("HelloService", Process.THREAD_PRIORITY_BACKGROUND).apply {
        start()
        serviceLooper = looper
        serviceHandler = ServiceTask(looper)
    }
}

override fun onStartCommand(intent: Intent?, flags: Int, startId: Int): Int {
    Toast.makeText(this, "Service starting!", Toast.LENGTH_LONG).show()
    serviceHandler?.obtainMessage()?.also { message: Message ->
        message.arg1 = startId
        serviceHandler?.sendMessage(message)
    }
    return START_STICKY
}

override fun onBind(intent: Intent?): IBinder? = null

override fun onDestroy() {Toast.makeText(this, "Service done!", Toast.LENGTH_LONG).show()}
```

El servicio y la inner class

Hello World Service parte 2

Para este ejemplo, creamos una inner class, la cual es la que ejecuta la tarea en el servicio, en este caso, se trata de un mensaje el cual se muestra usando un Toast con un delay de 5 segundos

IMPORTANTE: notaste que el Toast se puede ejecutar fuera del contexto de la aplicación, es decir, el Toast se ejecuta una vez que la app es terminado

El servicio y la inner class

Hello World Service parte 2

```
class HelloService : Service() {
    . . .
    private inner class ServiceTask(looper: Looper) : Handler(looper) {
        override fun handleMessage(msg: Message) {
            try {
                Thread.sleep(5000L)
            } catch (e: InterruptedException) {
                Thread.currentThread().interrupt()
            }
        }
    }
}
```

IMPORTANTE: recuerda que un servicio se ejecuta en el mismo proceso que la aplicación en la que se declara y en el subproceso principal de esa aplicación de forma predeterminada. Si tu servicio realiza operaciones intensivas o de bloqueo mientras el usuario interactúa con una actividad de la misma aplicación, el servicio ralentiza el rendimiento de la actividad. Para evitar afectar el rendimiento de la aplicación, inicia un nuevo subproceso dentro del servicio.

La capacidad de manejar varios threads o hilos al mismo tiempo permite que tu app pueda realizar múltiples tareas, brindando así una mejor experiencia de usuario

Próxima sesión...

- *Reconocer los aspectos fundamentales del patrón de promesas para la ejecución asíncrona.*

{desafío}
latam_

Academia de
talentos digitales