Resumen B10. Corrutinas y Servicios

Descargar estos apuntes

Índice

Corrutinas Kotlin

Una corrutina de Koytlin es un conjunto de sentencias que realizan una tarea específica, con la capacidad de suspender o resumir su ejecución sin bloquear un hilo. Esto permite que tengas diferentes corrutinas cooperando entre ellas y no significa que exista un hilo por cada corrutina, al contrario, puedes ejecutar varias en un solo.

Dependencia correspondiente:

```
implementation "org.jetbrains.kotlinx:kotlinx-coroutines-android:1.5.1"
```

Corrutinas en el desarrollo de Android

Alcance de las corrutinas

El uso de `GlobalScope` permite crea corrutinas de nivel superior, esto quiere decir que tienen la capacidad de vivir hasta que termine la aplicación y es trabajo del desarrollador el control para su cancelación, por lo que no se aconseja usar este ámbito.

```
private fun ejemploGlobalScope() {
          GlobalScope.launch(Dispatchers.Main) {
               // primero ejecuta el delay y al acabar el Toast
               delay(3000)

               Toast.makeText(requireActivity(), "GlobalScope: He terminado",
                   Toast.LENGTH_SHORT).show()
       }
   }
```

Otro ejemplo:

```
private fun ejemploGlobalScope() {
          GlobalScope.launch(Dispatchers.Main) {
            //lanzamos una segunda corrutina
            launch(Dispatchers.IO) {
               //en ella lanzamos un delay de 3000
               delay(3000)
            }
            //sin esperar a terminar el delay, lanza el Toast
            Toast.makeText(requireActivity(), "GlobalScope",
            Toast.LENGTH_SHORT).show()
        }
    }
```

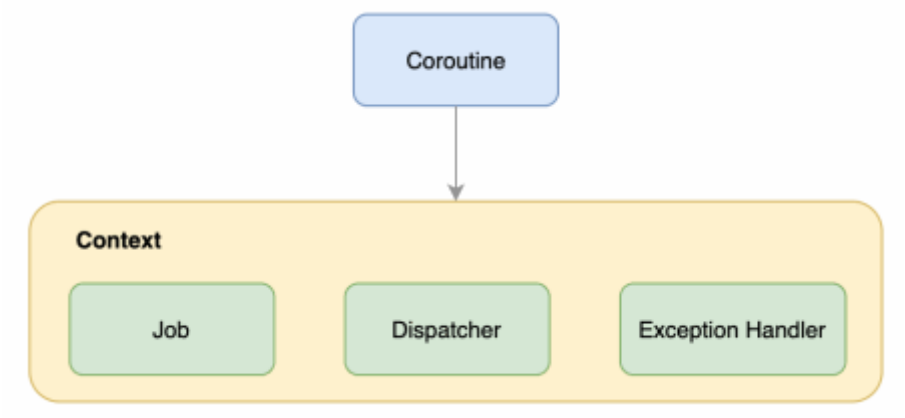
Para reducir este alcance, Kotlin nos permite crear los espacios donde queremos que se dé la concurrencia. Esto lo haremos mediante `CoroutineScope`.

```
    var coroutineScope= CoroutineScope(Dispatchers.Main)
    private fun ejemploCoroutineScopeConMain()
    {
       coroutineScope?.launch {
        //Ponemos un bucle que isActive será cierto hasta que se cancela la corr
        while (isActive) {
            delay(3000)
            Toast.makeText(requireActivity(), "[CoroutineScope-Main] I'm alive o
            "${Thread.currentThread().name}!", Toast.LENGTH_SHORT).show()
        }
      }
    }
    //Al salir del fragment se cancela la corutina
    override fun onDetach() {
        super.onDetach()
        coroutineScope?.cancel()
    }
```

Cada vez que usamos un constructor de coroutines en realidad estamos haciendo una llamada a una función que recibe como primer parámetro un objeto de tipo `CoroutineContext`.

CoroutineContext

Las coroutines siempre se ejecutan en algún contexto que está representado por un valor del tipo CoroutineContext.



Funciones de Suspensión

Las corrutinas se basan en las **funciones de suspension** que son funciones normales a las que se les agrega el modificador `suspend`, las funciones de suspensión tienen la capacidad de bloquear la ejecución de la corrutina mientras están haciendo su trabajo. Una vez que termina, el resultado de la operación se devuelve y se puede utilizar en la siguiente línea.

withContext

```
        CoroutineScope(Dispatchers.Main).launch {
            ....
            withContext(Dispatchers.IO) {
                //With the context Dispatchers.IO
                 ....
            }
            ....
        }
```

Builders de corrutinas

- `runBlocking`, este builder **bloquea** el hilo actual hasta que se terminen todas las tareas dentro de esa corrutina.

```
    fun ejemploRunBlocking()
    {
            runBlocking (Dispatchers.IO){
                //Lanzamos tres corrutinas con launch
                for (i in 1..3) {
                    launch(Dispatchers.Default) {
                        //delay es una funcion de suspensión por lo que al
                        //lanzar las tres corrutinas el tiempo no duran 5seg
                        //ya que se hacen las tres intercaladas
                        //a diferencia de sleep
                        Log.d("CORRUTINA","${Thread.currentThread().name}")
                        delay(5000)
                    }
                }
            }
        Toast.makeText(requireActivity(),"Ya han terminado las tareas lanzad
        "con el constructor runBlocking." +
        "\nSe desbloquea el hilo principal y se muestra este texto",
        Toast.LENGTH_LONG).show()
    }
```

- `launch`, este es el builder más usado. A diferencia de runBlocking, **no bloqueará** el subproceso actual (si se usan los dispatchers adecuados). Launch **devuelve un Job**

```
...
//launch especificando el alcance mediante CoroutineScope
CoroutineScope(Dispatchers.Main).launch()
{
//Al estar dentro de una corrutina padre, se puede omitir
//el alcance e incluso el Dispatchers si no queremos cambiar
//el contexto del padre, en este caso si lo cambiamos
launch(Dispatchers.IO){  ...    }
}
...
```

- `async`, este otro builder permite ejecutar varias tareas en **segundo plano en paralelo**. No es una función de suspensión en sí misma, por lo que cuando ejecutamos async, el proceso en segundo plano se inicia, pero **la siguiente línea se ejecuta de inmediato**. `async` siempre debe llamarse dentro de otra corrutina, y devuelve un job especializado que se llama `Deferred`. `Deferred` tiene una nueva función de suspensión llamada `await()` que es la que bloquea. Llamaremos a await() solo cuando necesitemos el resultado. Si el resultado aún no esta listo, la corrutina se suspende en ese punto. Si ya tenemos el resultado, simplemente lo devolverá y continuará.

```
private fun ejemploAsyncAway()
{
    var cadena:String?=null
    CoroutineScope(Dispatchers.Main).launch {
        val job=async {
            for (i in 1..5) {
                tiempo.text =i.toString()
                withContext(Dispatchers.IO) { delay(1000)}
            }
            "Ha terminado la corrutina async"
        }
        cadena=job.await()
        Toast.makeText(requireActivity(),cadena,Toast.LENGTH_SHORT).show()
    }
}
```

ViewModel y Corrutinas

ViewModel es extendido con una propiedad `viewModelScope` que se encarga de cancelar las corrutinas cuando ya no son necesarias.

```
//propiedad viewModelScope extendida de CoroutineScope y
//que a su vez extiende el ViewModel
val ViewModel.viewModelScope: CoroutineScope
   get()
   set()
```

Gracias al comportamiento de `CoroutineScope` a través de la propiedad `viewModelScope` se realiza un seguimiento de todas las corrutinas que se crean en este ambito. Por lo tanto, si se cancela un alcance, se cancelan todas las corrutinas creadas en él.

Por ejemplo, en el siguiente código creamos un ViewModel que nos gestionará un `MutableLiveData` de tipo ArrayList de Strings y que tiene un método que simula una descarga de datos que se guardarán en el objeto:

```
...
import androidx.lifecycle.viewModelScope
...
class ItemViewModel : ViewModel() {
    private val valores =
        arrayOf("item1", "item2", "item3", "item4", "item5", "item6", "item7", "
    private var liveData =
        MutableLiveData<ArrayList<String>>()
    val datos: LiveData<ArrayList<String>> get() = liveData

    fun descargarDatos() {
        val random = Random()
        var aux = ArrayList<String>()
        //Simulación de una descarga lenta de datos
        val numeroElementos = random.nextInt(10)
        viewModelScope.launch {
            for (i in 0..numeroElementos) {
                aux.add(valores[random.nextInt(valores.size - 1)])
                delay(1000)
            }
            liveData.value = aux
        }
    }
}
```

Servicios y tareas de larga duración

Servicios

Servicios

Construyendo un Servicio

Ejemplo de un servicio sencillo, podemos analizar su funcionamiento visualizando las lineas de LogCat:

```
class MyService: Service() {
    override fun onCreate() {
        super.onCreate()
        Log.d("DATO","Creando servicio....")
    }
    override fun onStartCommand
                (intent: Intent?, flags: Int, startId: Int): Int {
        super.onStartCommand(intent, flags, startId)
        Log.d("DATO","Recibiendo Intent ....")
        while(true)  Log.d("DATO","Mostrando intent ....+" +
                            "${intent?.getStringExtra("CADENA")}")
        return START_STICKY
    }
    override fun onBind(p0: Intent?): IBinder? {
        Log.d("DATO","Enlazando Intent ....")
        return null
    }
    override fun onDestroy() {
        Log.d("DATO","Destruyendo Intent ....")
        super.onDestroy()
    }
}
```

Una manera de inicializar este servicio, es desde una actividad y de forma muy parecida a iniciar una activity:

```
class MainActivity : AppCompatActivity() {
    lateinit var intento:Intent
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
        findViewById<MaterialButton>(R.id.boton).setOnClickListener{
            intento=Intent(applicationContext,MyService::class.java)
            intento.putExtra("CADENA","Dato Pasado desde Main")
            startService(intento)
        }
    }
    override fun onDestroy() {
        super.onDestroy()
        stopService(intento)
    }
}
```

> ⚠ No olvidar que se debe declarar los servicios usados en el manifest.
>
> **`<service android:name=".MyService"/>`**

Todo este código funcionaría en primer plano, si deseamos crear un hilo para realizar tareas en segundo plano, habría que hacerlo en onStartCommand:

```
class MyServiceSegundoPlano: Service() {
     var hilo=Thread()
    override fun onStartCommand
                (intent: Intent?, flags: Int, startId: Int): Int {
        super.onStartCommand(intent, flags, startId)
        val cadena=intent?.getStringExtra("CADENA")
        Log.d("DATO","Recibiendo Intent ....")
        if(hilo==null || !hilo?.isAlive) {
            hilo = Thread {
                while (true) Log.d(
                    "DATO", "Mostrando intent ...." +
                            "${cadena}"
                )
            }
            hilo.start()
        }

        return START_STICKY
    }
    override fun onBind(p0: Intent?): IBinder? {
        return null
    }
    override fun onDestroy() {
        Log.d("DATO","Destruyendo Intent ....")
        super.onDestroy()
    }
}
```

Aplicación ejemplo para crear un servicio para poder consumirlo desde una actividad

Vamos a enlazar el servicio enlazándolo con la actividad (Bound)

1. Creamos un nuevo proyecto llamado MiServicio.
2. Como dato adicional, a la actividad principal del proyecto la he llamado Consumer:
3. Creamos una nueva clase llamada MiServicio que es en dónde vamos a escribir todo el código correspondiente a lo que hace el servicio. A continuación te proporciono el código que deberá tener esta clase:

```
  class MiService : Service() {
    var timer: Timer?=null
    var miBinder: IBinder = MiBinder()
    lateinit var lista: ArrayList<String>
    var array = arrayOf("blanco", "azul", "verde")
    var pos = 0
    override  fun onCreate() {
        pos = 0
        super.onCreate()
        timer=Timer()
        datos = ArrayList()
        //LLamamos a la funcionalidad que queramos ejecutar
    }

    fun miFuncionalidad() {
        ...
    }
    override  fun onDestroy() {
        super.onDestroy()

    }
    override fun onBind(intent: Intent?): IBinder {
        return miBinder
    }
    fun getDatos(): List<String> {
        return datos
    }

    internal inner class MiBinder : Binder() {
        val service: MiService
            get() = this@MiService
    }
    companion object {
        private const val UPDATE_INTERVAL: Long = 5000
    }
}
```

4. Ahora pasemos a codificar la actividad que consumirá este servicio. Para la cuál definimos la interfaz gráfica con un botón para refrescar la información del servicio y una ListView para desplegar los datos del mismo.

```
class MainActivity : Activity() {
    var miservicio: MiService?=null
    lateinit  var values: ArrayList<String>
    lateinit  var adapter: ArrayAdapter<String>
    lateinit  var lista: ListView
    lateinit var boton: Button
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
        lista = findViewById(R.id.listView)
        boton = findViewById(R.id.button)
        doBindService()
        values = ArrayList()
        adapter = ArrayAdapter<String>(this, android.R.layout.
                                    simple_list_item_1, values)
        lista.setAdapter(adapter)
        boton.setOnClickListener {  showServiceData()}
    }
    private val mConnection: ServiceConnection = object : ServiceConnection
    {
        override fun onServiceConnected(name: ComponentName,
                                        service: IBinder) {
            miservicio = (service as MiBinder).service
            Toast.makeText(this@MainActivity, "Connectado",
                        Toast.LENGTH_LONG).show()
        }
        override fun onServiceDisconnected(name: ComponentName) {
            miservicio = null
        }
    }
    fun doBindService() {
        bindService(Intent(this, MiService::class.java), mConnection,
                            BIND_AUTO_CREATE)
    }
    fun showServiceData() {
        if (miservicio != null) {
            val listaTrabajo: List<String> = miservicio!!.getDatos()
            values.clear()
            values.addAll(listaTrabajo)
            adapter.notifyDataSetChanged()
        }
    }
}
```

5. IMPORTANTE , una vez que hemos terminado la lógica de la aplicación nos faltará dar de alta el servicio creado, en el archivo AndroidManifest.xml.

BroadCastReceiver

Aplicación ejemplo

```
class BroadCastAlarma : BroadcastReceiver() {
    @RequiresApi(Build.VERSION_CODES.O)
    override fun onReceive(context: Context, intent: Intent?) {
        Toast.makeText(
            context,
            "Vibración activa porque el tiempo se ha terminado",
            Toast.LENGTH_LONG
        ).show()
        val vibrator = context.getSystemService(Context.VIBRATOR_SERVICE)
                                                        as Vibrator
        vibrator.vibrate(VibrationEffect.createOneShot(8000,
                                        VibrationEffect.DEFAULT_AMPLITUDE))
    }
}
```

```
class MainActivity :AppCompatActivity() {
    lateinit var texto:EditText
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
        texto=findViewById<EditText>(R.id.editText)
        texto.setText("0")
        findViewById<Button>(R.id.button).setOnClickListener {  activar()}
    }

    private fun activar() {
        val tiempo = texto.text.toString().toInt()
        val intent = Intent(this,BroadCastAlarma::class.java)
        val pendingIntent = PendingIntent.getBroadcast(this,
                                                REQUESTCODE, intent, 0)
        val alarmManager = getSystemService(ALARM_SERVICE) as AlarmManager
        alarmManager[AlarmManager.RTC_WAKEUP, System.currentTimeMillis()
                                                    + tiempo * 1000] =
                                                         pendingIntent
        Toast.makeText(this, "Has fijado la alarma en "+tiempo+"segundos",
                                        Toast.LENGTH_LONG).show()
    }
    companion object{
        val REQUESTCODE=1111
    }
}
```

WorkManager

WorkManager

```
   implementation 'androidx.work:work-runtime-ktx:2.6.0'
```

Crear un trabajo

```
class WorkManager(val context: Context, workerParams: WorkerParameters) :
    Worker(context, workerParams){
    override fun doWork(): Result {
        ...
        var intento=Intent()
        intento.setComponent(
            ComponentName("com.ejemplos.b10.ejemploviewmodelcorrutinas",
            "com.ejemplos.b10.ejemploviewmodelcorrutinas.MainActivity")
        )
        //Se añade el Flag para que se pueda abrir una activity
        // desde un hilo en segundo plano
        intento.addFlags(Intent.FLAG_ACTIVITY_NEW_TASK)
        if (intento.resolveActivity(context.packageManager) != null)
            startActivity(context,intento,null)
        return Result.success()
    }
}
```

Iniciar un trabajo

```
val work = OneTimeWorkRequestBuilder<MiWorkManager>()
val workPeriodico =PeriodicWorkRequestBuilder<MiWorkManager>(16, TimeUnit.MINUTE
                                                              .build()
```

Una vez creado el objeto, para ejecutar la solicitud de trabajo necesitamos llamar al método `enqueue()` desde una instancia de `WorkManager` y pasar el `WorkRequest`:

```
val workPeriodico = PeriodicWorkRequestBuilder<WorkPeriodico>(16, TimeUnit.MINUT
                                                              .build()
WorkManager.getInstance(this).enqueue(workPeriodico)
```

El método `enqueue()` pone en cola una o más `WorkRequests` para que se ejecuten en segundo plano.

Funcionalidad de WorkManager

```
val workManager = WorkManager.getInstance(this)
val id:UUID
id = workPeriodico.id
if(!workManager.getWorkInfoById(id).isCancelled)
                workManager.cancelWorkById(id)
//En este caso se cancelan todas las tareas lanzadas por esta actividad
workManager.cancelAllWork()
```

```
class MiUnicoWork(val context: Context, workerParams: WorkerParameters) :
    Worker(context, workerParams){
    override fun doWork(): Result {
     ...
     //isStopped devolverá true cuando el sistema informe
     //que el trabajo ha sido cancelado desde código
        while(!isStopped)
        {
            Thread.sleep(1000)
            Log.d("DATOS", "Realizar tarea larga")
        }
        ...
        return Result.success()
    }
}
```

Restricciones de WorkManager

Restringir

```
//Se construyen las restricciones
val constraints = Constraints.Builder()
            .setRequiresBatteryNotLow(true)
            .setRequiredNetworkType(NetworkType.CONNECTED)
            .setRequiresCharging(true)
            .setRequiresStorageNotLow(true)
            .setRequiresDeviceIdle(true)
            .build()
val workUnico = OneTimeWorkRequestBuilder<MiUnicoWork>()
//Se añaden al trabajo
workUnico.setConstraints(constraints)
WorkManager.getInstance(this).enqueue(workUnico.build())
```

Comunicación con WorkManager

```
val inputData = Data.Builder()
                .putString("USER", user)
                .putString("PASS", pass)
                .build()
val workUnico = OneTimeWorkRequestBuilder<MiUnicoWork>()
workUnico.setInputData(inputData)
WorkManager.getInstance(this).enqueue(workUnico.build())
```

```
class MiUnicoWork(val context: Context, workerParams: WorkerParameters) :
    Worker(context, workerParams){
    override fun doWork(): Result {
     ...
    val user = inputData.getString("USER")
    val pass = inputData.getString("PASS")
     ...
        return Result.success()
    }
}
```