Développement Android

Laboratoire n°4: Tâches asynchrones et Coroutines - Galerie d'images

Friedli Jonathan, Marengo Stéphane, Silvestri Géraud

21.12.2022

Introduction

Le but de ce laboratoire est de développer une application android faisant office de galerie d'images. L'application devra permettre de scroller à travers une liste de 10'000 images, qui seront téléchargées en ligne. Le but de ce laboratoire est de se familiariser avec les tâches asynchrones (`coroutines` et `WorkManager`).

1. Détails d'implémentation

1.1. Layout

Dans le layout `activity_main.xml`, nous avons uniquement une `RecyclerView` qui contiendra les images téléchargées. Nous avons ensuite créé un layout `image_grid_item.xml` pour chaque item de la `RecyclerView`. Ce layout contient une `ImageView` et une `ProgressBar` qui sera affichée pendant le téléchargement de l'image.

activity_main.xml:

```
<androidx.constraintlayout.widget.ConstraintLayout
    [...] >

    <androidx.recyclerview.widget.RecyclerView
        android:id="@+id/image_list"
        [...] />

</androidx.constraintlayout.widget.ConstraintLayout>
```

image_grid_item.xml:

```
<androidx.constraintlayout.widget.ConstraintLayout
    [...] >

    <ImageView
        android:id="@+id/image_view"
        android:visibility="gone"
        [...] />

    <ProgressBar
        android:id="@+id/progress_bar"
```

```
        [...] />

</androidx.constraintlayout.widget.ConstraintLayout>
```

Nous avons également un menu qui est composé d'un unique bouton permettant de rafraichir la liste des images.

main_menu.xml:

```
<menu xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto">
    <item
        android:id="@+id/main_menu_cleanup_button"
        [...] />
</menu>
```

1.2. ImageLoader

```
typealias Callback = (Bitmap) -> Unit

class ImageLoader(
    private val lifecycleScope: CoroutineScope,
    private val cacheManager: CacheManager,
    private val maxCacheDuration: Duration
) {
    fun load(imageURL: URL, onComplete: Callback): Job {
        return lifecycleScope.launch {
            if (tryLoadFromCache(imageURL, onComplete)) {
                return@launch
            }

            downloadImage(imageURL)?.let { bytes ->
                decodeImage(bytes)?.let { bitmap ->
                    putInCache(imageURL, bitmap)
                    completeLoading(bitmap, onComplete)
                }
            }
        }
    }

    private suspend fun tryLoadFromCache(imageURL: URL, onComplete: Callback):
Boolean {
        return getFromCache(imageURL)?.let { bytes ->
            decodeImage(bytes)?.let { bitmap ->
                completeLoading(bitmap, onComplete)
                true
            }
        } ?: false
    }
```

```
    private suspend fun completeLoading(bitmap: Bitmap, onComplete: Callback) =
        withContext(Dispatchers.Main) {
            onComplete(bitmap)
        }

    private suspend fun downloadImage(url: URL): ByteArray? =
withContext(Dispatchers.IO) {
        try {
            url.readBytes()
        } catch (e: IOException) {
            Log.e("downloadImage", "Exception while downloading image
${e.message}", e)
            null
        }
    }

    private suspend fun decodeImage(bytes: ByteArray): Bitmap? =
withContext(Dispatchers.Default) {
        try {
            BitmapFactory.decodeByteArray(bytes, 0, bytes.size)
        } catch (e: IOException) {
            Log.e("decodeImage", "Exception while decoding image ${e.message}", e)
            null
        }
    }

    private suspend fun putInCache(url: URL, bitmap: Bitmap) =
withContext(Dispatchers.IO) {
        cacheManager.put(url.hashCode().toString(), bitmap)
    }

    private suspend fun getFromCache(url: URL): ByteArray? =
withContext(Dispatchers.IO) {
        cacheManager.get(url.hashCode().toString(), maxCacheDuration)
    }
}
```

Chaque étape du processus est découpé dans une méthode suspensive respective s'exécutant dans le `Dispatcher` le plus adapté. Pour notifier l'appelant de la fin du téléchargement, nous utilisons un système de callback.

Chaque instance de la classe est associée à une `CoroutineScope`.

1.3. CacheManager

```
class CacheManager(private val directory: File) {
    companion object {
        private val MIN_INTERVAL = Duration.ofMinutes(15)
        private const val PERIODIC_TAG = "PeriodicCleanup_"
        private const val ONE_TIME_TAG = "OneTimeCleanup_"
```

```
    }

    init {
        if (!directory.exists() || !directory.isDirectory) {
            throw IllegalArgumentException("$directory must be a directory")
        }

        if (!directory.canWrite()) {
            throw IllegalArgumentException("$directory must be writable")
        }
    }

    fun registerPeriodicCleanup(cleanInterval: Duration, context: Context) {
        if (cleanInterval < MIN_INTERVAL) {
            throw IllegalArgumentException("interval cannot be smaller than
${MIN_INTERVAL.toMinutes()} minutes")
        }

        val workManager = WorkManager.getInstance(context)
        val tag = PERIODIC_TAG + directory.name

        val workRequest =
            PeriodicWorkRequestBuilder<DirectoryCleanerWorker>(cleanInterval)
                .addTag(tag)
                .setBackoffCriteria(
                    BackoffPolicy.EXPONENTIAL,
                    PeriodicWorkRequest.MIN_BACKOFF_MILLIS,
                    TimeUnit.MILLISECONDS
                )
                .setInputData(DirectoryCleanerWorker.createInputData(directory))
                .build()

        workManager.enqueueUniquePeriodicWork(tag,
ExistingPeriodicWorkPolicy.KEEP, workRequest)
    }

    fun cleanup(context: Context) {
        val workManager = WorkManager.getInstance(context)
        val tag = ONE_TIME_TAG + directory.name

        val workRequest = OneTimeWorkRequestBuilder<DirectoryCleanerWorker>()
            .addTag(tag)
            .setInputData(DirectoryCleanerWorker.createInputData(directory))
            .build()

        workManager.enqueue(workRequest)
    }

    fun get(key: String, expirationTime: Duration): ByteArray? {
        val file = File(directory, key)
        if (!file.exists() || file.lastModified() + expirationTime.toMillis() <=
System.currentTimeMillis()) {
            return null
        }
```

```
            return try {
                file.readBytes()
            } catch (e: IOException) {
                Log.e("readImageFile", "Exception while reading image file
${e.message}", e)
                null
            }
        }

        fun put(key: String, bitmap: Bitmap) {
            val file = File(directory, key)
            file.outputStream().use { bitmap.compress(Bitmap.CompressFormat.JPEG, 100,
it) }
        }
}
```

Cette classe gère l'enregistrement des tâches de nettoyage périodiques et ponctuelles à l'aide de `WorkManager`. Elle permet également de lire et d'écrire des images dans le cache.

Pour pouvoir différencier les tâches dans la console, nous leur avons ajouté un tag.

1.4. DirectoryCleanerWorker

```
class DirectoryCleanerWorker(context: Context, params: WorkerParameters) :
    Worker(context, params) {
    companion object {
        const val DIRECTORY_KEY = "DIRECTORY_KEY"

        fun createInputData(directory: File): Data {
            if (!directory.exists() || !directory.isDirectory) {
                throw IllegalArgumentException("$directory is not a directory")
            }

            if (!directory.canWrite()) {
                throw IllegalArgumentException("$directory is not writable")
            }

            return Data.Builder()
                .putString(DIRECTORY_KEY, directory.absolutePath)
                .build()
        }

    }

    override fun doWork(): Result {
        val pathStr = inputData.getString(DIRECTORY_KEY)
        val directory = pathStr?.let { File(it) }

        if (directory == null || !directory.isDirectory || !directory.canWrite())
{
```

```
            return Result.failure()
        }

        val result = directory.listFiles()?.fold(true) { result, file ->
            result && file.deleteRecursively()
        }

        return if (result != null && result) {
            Result.success()
        } else {
            Result.failure()
        }
    }
}
```

C'est la classe qui effectue le nettoyage du cache. Pour lui passer des données, en l'occurence le dossier à vider, nous avons utilisé un `InputData`.

Nous avons créé une méthode statique `createInputData` pour créer les `InputData` à partir d'un `File` passé en paramètre et ainsi transférer le chemin du dossier à vider de la requête à la tâche.

1.5. ImageViewAdapter

```
class ImageViewAdapter(
    private val items: List<URL>,
    private val imageLoader: ImageLoader
) : RecyclerView.Adapter<ImageViewAdapter.ViewHolder>() {

    fun reload() {
        notifyDataSetChanged()
    }

    override fun onCreateViewHolder(parent: ViewGroup, viewType: Int): ViewHolder
{
        val layout = R.layout.image_grid_item
        return ViewHolder(
            LayoutInflater.from(parent.context).inflate(layout, parent, false)
        )
    }

    override fun onBindViewHolder(holder: ViewHolder, position: Int) {
        holder.bind(items[position])
    }

    override fun onViewRecycled(holder: ViewHolder) {
        holder.unbind()
    }

    override fun getItemCount(): Int {
        return items.size
    }
```

```
    inner class ViewHolder(view: View) : RecyclerView.ViewHolder(view) {
        // [...] récupération des vues

        fun bind(imageUrl: URL) {
            job = imageLoader.load(imageUrl, ::showImage)
        }

        fun unbind() {
            imageView.setImageBitmap(null)
            imageView.visibility = View.GONE
            progressBar.visibility = View.VISIBLE
            job?.cancel()
        }

        // Callback appelé lorsque l'image est téléchargée
        private fun showImage(bitmap: Bitmap) {
            imageView.setImageBitmap(bitmap)
            imageView.visibility = View.VISIBLE
            progressBar.visibility = View.GONE
        }
    }
}
```

Lorsqu'une vue est bindée à un `ViewHolder`, le processus de téléchargement est initié par ce dernier à l'aide de la méthode `load` de l'`ImageLoader`. Une fois le processus terminé, le `ViewHolder` est notifié à l'aide d'un callback.

La méthode `unbind` est décrite plus bas.

1.6. MainActivity

```
class MainActivity : AppCompatActivity() {
    companion object {
        const val NB_COLUMNS = 3
        const val IMAGE_URL = "https://daa.iict.ch/images/%d.jpg"
        const val NB_IMAGES = 10_000
        val CACHE_DURATION: Duration = Duration.ofMinutes(5)
        val CACHE_CLEAN_INTERVAL: Duration = Duration.ofMinutes(15)
    }

    private val cacheManager by lazy { CacheManager(cacheDir) }
    private lateinit var adapter: ImageViewAdapter

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)

        val imageLoader = ImageLoader(lifecycleScope, cacheManager,
CACHE_DURATION)
        val images = (1..NB_IMAGES).map { URL(IMAGE_URL.format(it)) }
```

```
        adapter = ImageViewAdapter(images, imageLoader)

        val recyclerView = findViewById<RecyclerView>(R.id.image_list)
        recyclerView.adapter = adapter
        recyclerView.layoutManager = GridLayoutManager(this, NB_COLUMNS)

        cacheManager.registerPeriodicCleanup(CACHE_CLEAN_INTERVAL, this)
    }

    override fun onCreateOptionsMenu(menu: Menu?): Boolean {
        menuInflater.inflate(R.menu.main_menu, menu)
        return super.onCreateOptionsMenu(menu)
    }

    override fun onOptionsItemSelected(item: MenuItem): Boolean {
        return when (item.itemId) {
            R.id.main_menu_cleanup_button -> {
                cacheManager.cleanup(this)
                adapter.reload()
                true
            }
            else -> super.onOptionsItemSelected(item)
        }
    }

    override fun onStop() {
        super.onStop()
        lifecycleScope.coroutineContext.cancelChildren()
    }
}
```

Dans la méthode `onCreate`, les différents composants sont initialisés et la tâche périodique enregistrée.

2. Questions sur l'Adapteur et les coroutines

2.1 **Veuillez expliquer comment votre solution s'assure qu'une éventuelle Couroutine associée à une vue (item) de la RecyclerView soit correctement stoppée lorsque l'utilisateur scrolle dans la galerie et que la vue est recyclée.**

Voici le code de l'`ImageViewAdapter`:

```
override fun onViewRecycled(holder: ViewHolder) {
    holder.unbind()
}


inner class ViewHolder(view: View) : RecyclerView.ViewHolder(view) {
    private val imageView by lazy { view.findViewById<ImageView>(R.id.image_view)
}

    [...]

    private var job: Job? = null

    fun bind(imageUrl: URL) {
        job = imageLoader.load(imageUrl, ::showImage)
    }

    fun unbind() {
        imageView.setImageBitmap(null)
        imageView.visibility = View.GONE
        progressBar.visibility = View.VISIBLE
        job?.cancel()
    }

    [...]
}
```

Dans la méthode unbind, on annule la coroutine associée à l'item de la RecyclerView. Ainsi, lorsque l'utilisateur scrolle dans la galerie et que la vue est recyclé, l'éventuelle coroutine associée est stoppée.

2.2 **Comment pouvons-nous nous assurer que toutes les Coroutines soient correctement stoppées lorsque l'utilisateur quitte l'Activité ? Veuillez expliquer la solution que vous avez mis en oeuvre, est-ce la plus adaptée ?**

Dans la méthode `load` de `ImageLoader`, nous lançons toutes les coroutines dans le scope qui nous est passé à la construction de l'objet (le `LifecycleScope` de l'activité dans notre cas).

Dans l'activité:

```
val imageLoader = ImageLoader(lifecycleScope, cacheManager, CACHE_DURATION)
```

Dans `ImageLoader`:

```
fun load(imageURL: URL, onComplete: Callback): Job {
        return lifecycleScope.launch {
            [...]
        }
    }
```

Nous lançons toutes les coroutines dans le `LifecycleScope` de l'activité. Par défaut toutes les coroutines sont arrêtées lorsque l'activité est détruite (lorsque l'owner du `LifecycleScope` est détruit). Ceci s'effectue dans la méthode `onDestroy`. Nous ne possédons aucune garantie que ladite méthode soit appelée.

Dans notre cas, nous avons donc redéfini la méthode `onStop` de l'activité:

```
override fun onStop() {
    super.onStop()
    lifecycleScope.coroutineContext.cancelChildren()
}
```

2.3 **Est-ce que l'utilisation du Dispatchers.IO est le plus adapté pour des tâches de téléchargement ? Ne faudrait-il pas plutôt utiliser un autre Dispatcher, si oui lequel ? Veuillez illustrer votre réponse en effectuant quelques tests.**

Selon la documentation d'android, le `Dispatcher.IO` est le plus adapté pour les entrées/sorties sur le réseau et de lectures/écritures sur les disques.

Alors que le `Dispatcher.Default` est le plus adapté pour les tâches de calculs lourds. Comme par exemple, tri de liste ou parsing de JSON.

Si nous lançons un téléchargement d'image dans le `Dispatcher.Main`, nous obtenons une exception car les opérations liées au réseau ne peuvent pas être effectuées dans le `Dispatcher.Main`.

On constate que le `Dispatcher.IO` est un peu plus rapide que le `Dispatcher.Default` pour télécharger une image.

3. Questions sur le nettoyage automatique du cache

3.1 **Lors du lancement de la tâche ponctuelle, comment pouvons nous faire en sorte que la galerie soit raffraîchie ?**

Nous avons créé une méthode `reload` dans l'adapter qui permet de rafraîchir la galerie. Cette méthode est appelée dans l'activité lorsque l'utilisateur clique sur le bouton de nettoyage du cache.

```
fun reload() {
    notifyDataSetChanged()
}
```

Dans l'activité:

```
override fun onOptionsItemSelected(item: MenuItem): Boolean {
        return when (item.itemId) {
            R.id.main_menu_cleanup_button -> {
                cacheManager.cleanup(this)
                adapter.reload()
                true
            }
            else -> super.onOptionsItemSelected(item)
        }
    }
```

3.2 **Comment pouvons-nous nous assurer que la tâche périodique ne soit pas enregistrée plusieurs fois ? Vous expliquerez comment la librairie WorkManager procède pour enregistrer les différentes tâches périodiques et en particulier comment celles-ci sont ré-enregistrées lorsque le téléphone est redémarré.**

La librairie WorkManager permet d'enregister des tâches périodiques uniques permettant de spécifier le comportement à adopter si une tâche périodique avec le même nom existe déjà. Dans notre cas, nous utilisons `ExistingPeriodicWorkPolicy.KEEP`. Cela signifie que si une tâche périodique avec le même nom existe déjà, elle sera conservée et la nouvelle tâche périodique ne sera pas enregistrée.

```
fun registerPeriodicCleanup(cleanInterval: Duration, context: Context) {

    [...]

    val workRequest = [...]

    workManager.enqueueUniquePeriodicWork(tag, ExistingPeriodicWorkPolicy.KEEP,
workRequest)
}
```

En interne, `WorkManager` utilise une base de données `SQLite` pour stocker les tâches périodiques. Lorsque le téléphone est redémarré, les tâches périodiques sont donc automatiquement ré-enregistrées par la librairie.