ANDROID

Laurent Provot <laurent.provot@uca.fr>

Janvier 2023

Coroutines Kotlin

Coroutines

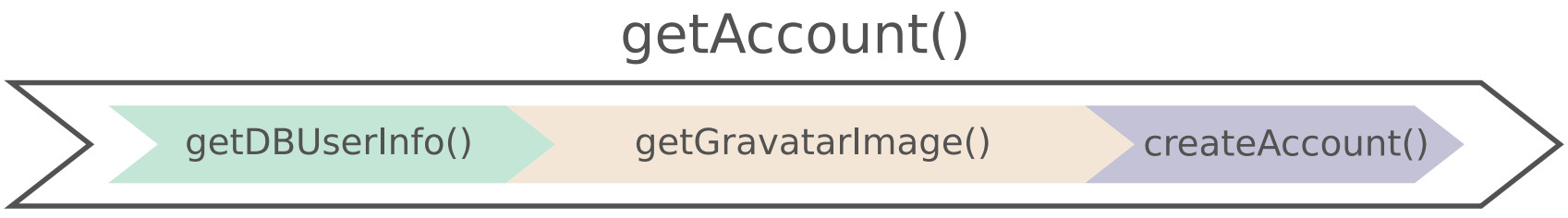
- Concept ancien, début des années 60 (Melvin Conway)
- Remis au goût du jour pour faire de l'asynchrone à « faible coût »
- Coroutine = exécution qui peut être suspendue, puis reprise
- De base dans le langage Kotlin et sa lib standard :
  - mot-clé `suspend`
  - quelques types de base, i.e. `CoroutineContext`, `Continuation`
  - quelques fonctions, i.e. `createCoroutine()`, `startCoroutine()`, `suspendCoroutine()`, `resume()`
- Pour tout le reste : intégration à travers une bibliothèque (`kotlinx.coroutines`)
  https://github.com/Kotlin/kotlinx.coroutines
- Lectures conseillées : celles de Roman Elizarov sur Medium :-)

Séquentiel vs concurrent

- Séquentiel : on fait les opérations les unes après les autres

```
fun getAccount(id: Int) : Account {
    val userInfo = getDBUserInfo(id)
    val avatar = getGravatarImage(id)

    return createAccount(userInfo, avatar)
}
```



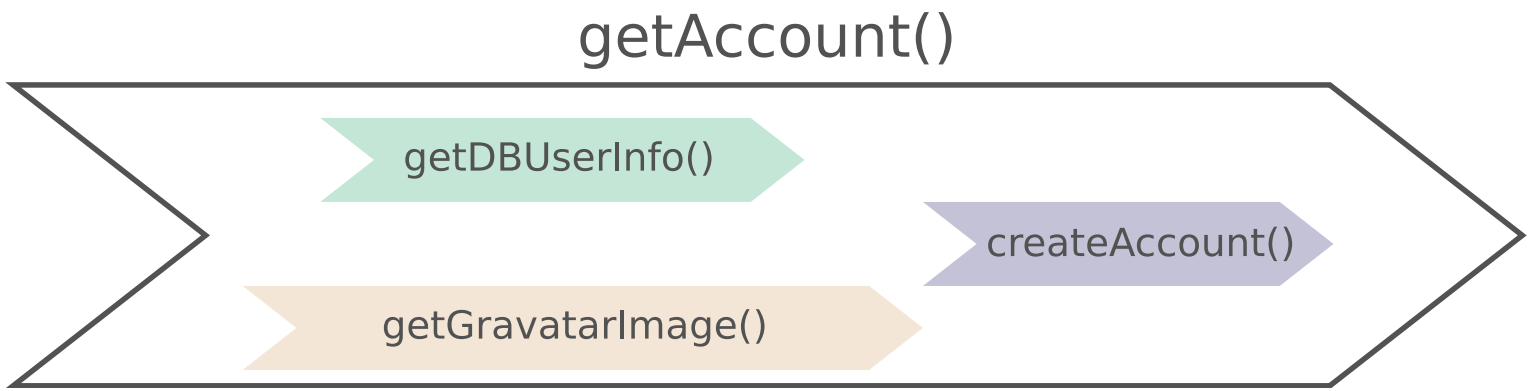
- Chaque opération doit attendre que la précédente soit terminée

Séquentiel vs concurrent

- Concurrent : variabilité dans l'ordre de certaines opérations, mais résultat déterministe

```
suspend fun getAccount(id: Int) : Account {
    val userInfo = async { getDBUserInfo(id) }
    val avatar = async { getGravatarImage(id) }

    return createAccount(userInfo.await(), avatar.await())
}
```

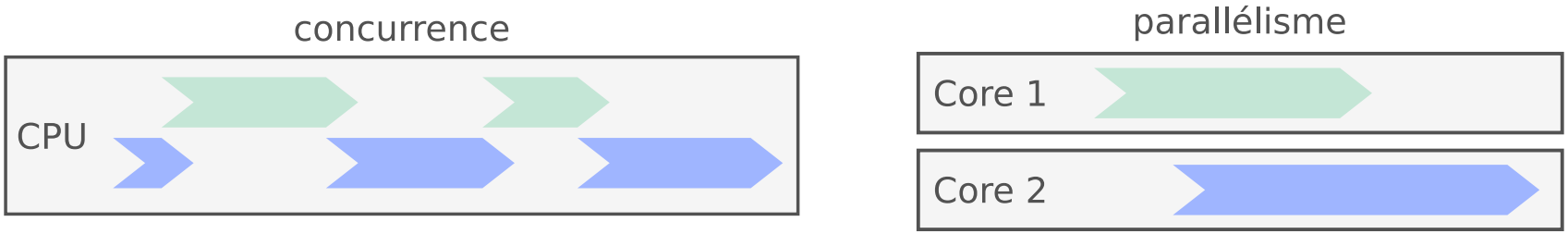


- Nécessite de l'indépendance entre certaines fonctions

Concurrence vs parallélisme

- Coroutines souvent comparées aux threads (maladroit)
- Thread : multi-tâche préemptif
- Coroutines : multi-tâche coopératif

- Coroutines utilisées pour mettre en place de la concurrence...
- ... mais profite du parallélisme



Problèmes classiques quand on fait de la concurrence

- Structuration du code difficile
- Callback hell, gestion des cas d'erreur difficile

```
const verifyUser = function(username, password, callback){
   dataBase.verifyUser(username, password, (error, userInfo) => {
       if (error) {
           callback(error)
       }else{
           dataBase.getRoles(username, (error, roles) => {
               if (error){
                   callback(error)
               }else {
                   dataBase.logAccess(username, (error) => {
                       if (error){
                           callback(error);
                       }else{
                           callback(null, userInfo, roles);
                       }
                   })
               }
           })
       }
   })
};
```

- Nouvelle API à appréhender (Fluent API des Promise/Future)
- Que avec threads : couteux

Apport des coroutines

- Structuration de code classique
- Utilisation des threads mais sans les bloquer
- Amélioration des performances

- Coroutines sont utiles dans 2 principaux cas sous Android :

1. Les tâches longues qui peuvent notamment bloquer le main thread
2. Préserver la « Main-safety » (s'assurer que les fonctions `suspend` puissent être appelées dans le main thread)

Réfléchir à l'utilisation

1. Algo utilisant beaucoup le CPU (*CPU bound*)
   - Profite du parallélisme
   - Ne profite pas vraiment de la concurrence en mono thread
   - Peut même en souffrir à cause des changements de contexte

1. Algo faisant beaucoup d'entrées/sorties (*IO bound*)
   - Profite beaucoup du parallélisme et de la concurrence (si E/S indépendantes)
   - Sera quasiment toujours mieux que du séquentiel

- On essaye de tirer partie des deux mondes
- Coroutines utilisent judicieusement les threads

Utilisation des coroutines en Kotlin

1 Création

- Utilisation de *coroutine builders*
- Les plus courants :
  - `launch()` : crée une coroutine qui ne renvoie pas de résultat (*fire-and-forget*); retourne le `Job` représentant la coroutine
  - `async()` : crée une coroutine permettant d'obtenir un résultat de type `T`; retourne une `Deferred<T>` qui est un job et une *future* qui encapsule le résultat
  - `runBlocking()` : crée une coroutine et bloque le thread courant tant qu'elle n'est pas terminée; utilisée pour faire le lien entre du code bloquant et du code non bloquant (entre le `main` et les *suspending functions* par exemple)

Utilisation des coroutines en Kotlin

2 Utilisation

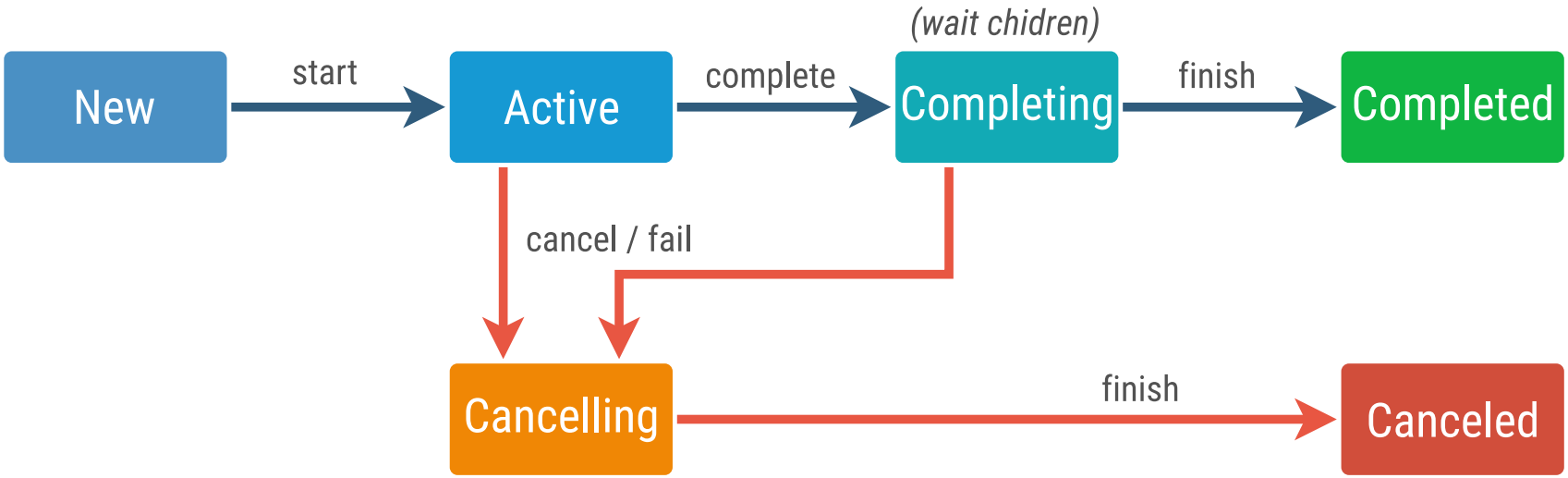
- Appel de fonctions `suspend` : travail long mais non boquant
- Attente de terminaison d'une coroutine `join()`
- Attente d'un résultat `await()`, `awaitAll()`
- Annulation d'un coroutine `cancel()`, `cancelAndJoin()`
- Lancement d'autres coroutines `withContext()`, `coroutineScope()`, ...
- Laisser la main à d'autres coroutines `yield`

Exemple « tout en 1 »

```
suspend fun getName(): String {
    delay(1000)
    return "John"
}
suspend fun getLastName(): String {
    delay(1000)
    return "Smith"
}
fun main() {
    lateinit var job: Job
    val time = measureTimeMillis {
        job = GlobalScope.launch {
            val name = async { getName() }
            val lastName = getLastName()
            println("Hello, ${name.await()} ${lastName}")
        }
    }
    println("Execution took $time ms")
    runBlocking {
       println("Join took " + measureTimeMillis { job.join() } + " ms")
    }
}
```

Le `Job`

- Coroutines représentées à travers le concept de `Job`
- `Deferred<T>` hérite de `Job`
- Gère le cycle de vie, l'annulation et les relations de parenté des coroutines



- Permet de mettre en place le mécanisme de *structured concurrency*

Structured concurrency

- Utiliser les caractéristiques du langage et des *best practices* pour conserver la trace des coroutines
- Pour Android cela revient à essayer de :
  - Garder un contrôle sur les tâches en cours d'exécution
  - Signaler les erreurs quand la coroutine échoue
  - Annuler les exécutions qui ne sont plus nécessaires

`CoroutineContext`

- Une map d'éléments
- Les éléments permettent de customiser la coroutine
- `CoroutineName`, `CoroutineDispatcher`, `CoroutineExceptionHandler`, `ContinuationInterceptor`
- Modifiable en les mixant grâce à l'opérateur `+` (les éléments de même clé sont remplacés)

```
launch(Dispatcher.IO +
       CoroutineName("DBAcces") +
       CoroutineExceptionHandler(…) ) {
  …
}
```

`CoroutineScope`

- Contient seulement un contexte (`CoroutineContext`)
- Un `Job` est un `CoroutineContext`
- Permet de définir des portés de code gérant l'existence de coroutines
- Appui la *structured concurrency*
- Intéressant car tous les builders (hormis `runBlocking()`) sont des fonctions d'extension de `CoroutineScope`

```
fun CoroutineScope.launch(
    context: CoroutineContext = EmptyCoroutineContext,
    start: CoroutineStart = CoroutineStart.DEFAULT,
    block: suspend CoroutineScope.() -> Unit
): Job
```

Exemple pour Android

```
class MyActivity : AppCompatActivity(), CoroutineScope {
  lateinit var job: Job
  override val coroutineContext: CoroutineContext
  get() = Dispatchers.Main + job

  override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    job = Job()
  }

  override fun onDestroy() {
    super.onDestroy()
    job.cancel()
  }

  fun loadDataFromUI() = launch {
    val ioData = async(Dispatchers.IO) { /* opération d'E/S */ }
    // ici des choses se font en concurrence avec ioData
    val data = ioData.await() // attente du résultat d'E/S
    updateUI(data) // mise à jour de la vue dans le main thread
  }
}
```

`CoroutineDispatcher`

- Permet de définir sur quels threads s'exécutent les coroutines
- Plusieurs implémentations existent de base :
  - `Dispatchers.Default` : celui utilisé par défaut quand rien n'est spécifié. C'est un pool de threads partagés. Approprié pour les coroutines qui consomment du CPU (parser du Json, trier une liste, ...)
  - `Dispatchers.IO` : utilise un pool partagé de thread créés à la demande. Parfait pour les opérations d'E/S bloquantes (lecture fichier, accès BdD, requête réseau, ...)
  - `Dispatchers.Unconfined` : démarré dans le thread qui lance la coroutine, puis si coroutine suspendue, redémarré dans le thread courant (pas forcément le même). *Ne devrait pas être utilisé normalement dans votre code*

`CoroutineDispatcher`

- Pour les frameworks ayant un main thread (ou UI thread) :
  - `Dispatchers.Main` (JavaFx, Android, ...) : utilisé pour des opérations légères (appeler des fonctions `suspend` main-safe, mettre à jour l'UI, ...)
- Possibilité de créer ses propres dispatchers : `newSingleThreadContext`, `newFixedThreadPoolContext`
- Possibilité de convertir un `Executor` en dispatcher avec `asCoroutineDispatcher`

Coroutines en interne

- Fonctionnent sur le principe de *Continuation-Passing Style*
- Utilisation d'un type spécial, `Continuation`
- Permet de stocker les informations pour continuer une fonction `suspend`
- Une machine à état pour :
  - Découper la fonction `suspend` suivant certains points de césure (appels `suspend`)
  - Sauvegarder les infos vitales (valeur de retour, paramètres, ...)

```
suspend fun createPost(token: Token, item: Item): Post { … }
```

devient pour la JVM

```
Object createPost(Token token, Item item, Continuation<Post> cont) { … }
```

Pour plus d'infos détaillés : https://resources.jetbrains.com/storage/products/kotlinconf2017/slides/2017+KotlinConf+-+Deep+dive+into+Coroutines+on+JVM.pdf

Coroutines sous Android

- On pourrait utiliser la méthode décrite plus haut en exemple de `CoroutineScope`
- Mais il faudrait écrire le même code souvent pour gérer ces `CoroutineScope`
- Les fonctions d'extension Kotlin relatives au cycle de vie évitent ce boilerplate code
- Ajouter les dépendances adéquates dans le `build.gradle`

```
dependencies {
    def lfccl_vers = "2.5.1"
    // ViewModel
    implementation "androidx.lifecycle:lifecycle-viewmodel-ktx:$lfccl_vers"
    // LiveData
    implementation "androidx.lifecycle:lifecycle-livedata-ktx:$lfccl_vers"
    // Lifecycles seulement (sans ViewModel ou LiveData)
    implementation "androidx.lifecycle:lifecycle-runtime-ktx:$lfccl_vers"
    …
}
```

KTX dans les `LifecycleOwner`

- Un `LifecycleScope` est définit pour chaque `Lifecycle`
- Coroutines lancées dans ce scope annulées s'il est détruit
- Accessible à travers la propriété :
  - `coroutineScope` du `Lifecycle`
  - `lifecycleScope` du `LifecycleOwner` (activité, fragment)
- Scopes disponibles pour certains états du cycle de vie :
  - `lifecycle.whenCreated` `lifecycle.whenStarted` et `lifecycle.whenResumed`
  - les coroutines lancés dans ces scopes sont suspendues tant que l'état en question n'est pas atteint

KTX dans la `ModelView`

- Chaque `ViewModel` définit un `ViewModelScope`
- Les coroutines lancées dans ce dernier sont annulées si la `ViewModel` passe dans son `onCleared()`
- Accessible à travers la propriété `viewModelScope`

```
class MyViewModel: ViewModel() {
    init {
        viewModelScope.launch {
            …
        }
    }
}
```

KTX pour les `LiveData`

```
val user: LiveData<User> = liveData {
    val data = database.loadUser()  // fonction suspend
    emit(data)
}
```

- `liveData` crée un `LiveDataScope` permettant de lancer des fonctions `suspend`
- `emit()` mettra à jour la valeur de la `LiveData` une fois le résultat obtenu
- Possibilité d'utiliser `emitSource()` pour remplacer la source de la `LiveData` (à la manière d'un `MediatorLiveData.addSource()` unique)

Coroutines dans Room ou Retrofit

- Room est compatible avec les coroutines
- Il suffit de mettre des fonctions `suspend` dans la DAO

```
implementation "androidx.room:room-ktx:$room_version"
```

- Retrofit, après la version 2.6.0 est compatible avec les coroutines
- Il suffit de mettre des fonctions `suspend` dans l'interface de requêtes

Pour aller plus loin

Cf. doc et guides Kotlin des coroutines

- Flow
- Actors
- Channel
- Sequences, itérateurs
- Producteur / consommateur