Rakamin Academy

schoters

Virtual Internship Experience

Android Coroutines

- Memahami konsep Coroutines
- Memahami Dispatchers

1. Memahami konsep Coroutines

Coroutine adalah pola desain serentak yang dapat Anda gunakan di Android untuk menyederhanakan kode yang dieksekusi secara asinkron. Coroutine ditambahkan pada Kotlin dalam versi 1.3 dan didasarkan pada konsep yang telah ditetapkan dari bahasa lain.

Di Android, coroutine berguna untuk mengelola tugas yang berjalan lama dan mungkin memblokir thread utama serta menyebabkan aplikasi tidak responsif. Lebih dari 50% developer profesional yang menggunakan coroutine telah melaporkan peningkatan produktivitas. Topik ini menjelaskan bagaimana Anda dapat menggunakan coroutine Kotlin untuk mengatasi masalah ini, sehingga Anda dapat menulis kode aplikasi yang lebih rapi dan ringkas.

Fitur Coroutine adalah solusi yang direkomendasikan untuk pemrograman asinkron di Android. Fitur penting meliputi:

- Ringan: Anda dapat menjalankan banyak coroutine pada satu thread karena adanya dukungan untuk penangguhan, yang tidak memblokir thread tempat coroutine berjalan. Penangguhan menghemat memori melalui pemblokiran sekaligus mendukung banyak operasi serentak.
- Lebih sedikit kebocoran memori: Gunakan struktur serentak untuk menjalankan operasi dalam cakupan.
- Dukungan pembatalan bawaan: Pembatalan otomatis disebarkan melalui hierarki coroutine yang sedang berjalan.
- Integrasi Jetpack: Banyak library Jetpack dilengkapi ekstensi yang menyediakan dukungan penuh coroutine. Beberapa library juga menyediakan cakupan coroutine sendiri yang dapat Anda gunakan untuk membuat struktur serentak.

Ringkasan contoh

Berdasarkan Panduan arsitektur aplikasi, contoh dalam topik ini membuat permintaan jaringan dan menampilkan hasilnya ke thread utama, tempat aplikasi kemudian dapat menampilkan hasilnya kepada pengguna.

Secara khusus, komponen Arsitektur ViewModel memanggil lapisan repositori pada thread utama untuk memicu permintaan jaringan. Panduan ini akan melakukan iterasi melalui berbagai solusi yang menggunakan coroutine untuk mencegah thread utama terblokir.

ViewModel menyertakan serangkaian ekstensi KTX yang berfungsi langsung dengan coroutine. Ekstensi tersebut adalah library lifecycle-viewmodel-ktx dan digunakan dalam panduan ini.

Info dependensi

Untuk menggunakan coroutine dalam project Android, tambahkan dependensi berikut ke file build.gradle aplikasi Anda:

```
dependencies {
  implementation("org.jetbrains.kotlinx:kotlinx-coroutines-android:1.3.9")
}
```

Menggunakan coroutine untuk main-safety

Kami menganggap sebuah fungsi bersifat main-safe jika tidak memblokir update UI pada thread utama. Fungsi makeLoginRequest tidak main-safe, karena memanggil makeLoginRequest dari thread utama akan memblokir UI. Gunakan fungsi withContext() dari library coroutine untuk memindahkan eksekusi coroutine ke thread lain:

```
class LoginRepository(...) {
    ...
    suspend fun makeLoginRequest(
        jsonBody: String
    ): Result<LoginResponse> {

        // Move the execution of the coroutine to the I/O dispatcher
        return withContext(Dispatchers.IO) {
            // Blocking network request code
        }
    }
}
```

Gambar 1. withContext Example.

withContext(Dispatchers.IO) memindahkan eksekusi coroutine ke thread I/O sehingga fungsi panggilan kami menjadi main-safe dan memungkinkan UI diupdate sesuai kebutuhan.

makeLoginRequest juga ditandai dengan kata kunci suspend. Kata kunci ini adalah cara Kotlin menerapkan fungsi yang akan dipanggil dari dalam coroutine.

Dalam contoh berikut, coroutine dibuat di LoginViewModel. Saat makeLoginRequest mengeluarkan eksekusi dari thread utama, coroutine dalam fungsi login kini dapat dijalankan pada thread utama:

```
class LoginViewModel(
    private val loginRepository: LoginRepository
): ViewModel() {

    fun login(username: String, token: String) {

        // Create a new coroutine on the UI thread
        viewModelScope.launch {
            val jsonBody = "{ username: \"$username\", token: \"$token\"}"

            // Make the network call and suspend execution until it finishes
            val result = loginRepository.makeLoginRequest(jsonBody)

            // Display result of the network request to the user
            when (result) {
                is Result.Success<LoginResponse> -> // Happy path
                else -> // Show error in UI
            }
        }
    }
}
```

Gambar 2. LoginViewModel Example

Perlu diingat bahwa coroutine masih diperlukan di sini, karena makeLoginRequest adalah fungsi suspend, dan semua fungsi suspend harus dijalankan dalam coroutine.

Dalam beberapa hal, kode ini berbeda dengan contoh login sebelumnya:

- launch tidak menggunakan parameter Dispatchers.IO. Jika Dispatcher tidak diteruskan ke launch, setiap coroutine yang diluncurkan dari viewModelScope akan dijalankan di thread utama.
- Kini hasil permintaan jaringan ditangani untuk menampilkan UI yang berhasil atau gagal.

Kini fungsi login berjalan seperti berikut:

- Aplikasi memanggil fungsi login() dari lapisan View pada thread utama.
- launch membuat coroutine baru untuk membuat permintaan jaringan pada thread utama, dan coroutine memulai eksekusi.
- Dalam coroutine, panggilan ke loginRepository.makeLoginRequest() kini menangguhkan eksekusi coroutine lebih lanjut hingga blok withContext dalam makeLoginRequest() selesai berjalan.
- Setelah blok withContext selesai, coroutine pada login() akan melanjutkan eksekusi pada thread utama dengan hasil permintaan jaringan.

Menangani pengecualian

Untuk menangani pengecualian yang dapat ditampilkan oleh lapisan Repository, gunakan dukungan pengecualian bawaan Kotlin. Dalam contoh berikut, kami menggunakan blok try-catch:

```
class LoginViewModel(
    private val loginRepository: LoginRepository
): ViewModel() {

    fun makeLoginRequest(username: String, token: String) {
        viewModelScope.launch {
            val jsonBody = "{ username: \"$username\", token: \"$token\"}"
            val result = try {
                loginRepository.makeLoginRequest(jsonBody)
            } catch(e: Exception) {
                Result.Error(Exception("Network request failed"))
            }
            when (result) {
                is Result.Success<LoginResponse> -> // Happy path
                else -> // Show error in UI
            }
        }
    }
}
```

Gambar 3. Try-catch example

Dalam contoh ini, setiap pengecualian tidak terduga yang ditampilkan oleh panggilan makeLoginRequest() ditangani sebagai error dalam UI.

Mengelola tugas yang berjalan lama

Coroutine membuat fungsi reguler dengan menambahkan dua operasi untuk menangani tugas yang berjalan lama. Selain invoke (atau call) dan return, coroutine menambahkan suspend dan resume:

- suspend menjeda eksekusi coroutine saat ini yang menyimpan semua variabel lokal.
- resume melanjutkan eksekusi coroutine yang ditangguhkan dari titik penangguhannya.

Anda dapat memanggil fungsi suspend hanya dari fungsi suspend lainnya atau dengan menggunakan builder coroutine seperti launch untuk memulai coroutine baru.

Contoh berikut menunjukkan implementasi coroutine sederhana untuk tugas hipotesis yang berjalan lama:

```
suspend fun fetchDocs() {                              // Dispatchers.Main
    val result = get("https://developer.android.com") // Dispatchers.IO for `get`
    show(result)                                       // Dispatchers.Main
}

suspend fun get(url: String) = withContext(Dispatchers.IO) { /* ... */ }
```

Gambar 4. Suspend example

Dalam contoh ini, get() masih berjalan pada thread utama, tetapi menangguhkan coroutine sebelum memulai permintaan jaringan. Saat permintaan jaringan selesai, get akan melanjutkan coroutine yang ditangguhkan, bukan menggunakan callback untuk memberi tahu thread utama.

Kotlin menggunakan frame stack untuk mengelola fungsi mana yang berjalan bersama dengan variabel lokal apa pun. Saat menangguhkan coroutine, frame stack saat ini akan disalin dan disimpan untuk nanti. Saat melanjutkan, frame stack akan disalin kembali dari tempatnya disimpan, dan fungsi mulai berjalan kembali. Meskipun kode mungkin terlihat seperti permintaan pemblokiran berurutan biasa, coroutine memastikan agar permintaan jaringan menghindari pemblokiran thread utama.

Konsep coroutine

CoroutineScope

CoroutineScope memantau setiap coroutine yang dibuat olehnya menggunakan launch atau async. Pekerjaan yang sedang berlangsung (misalnya, coroutine yang berjalan) dapat dibatalkan dengan memanggil scope.cancel() kapan saja. Di Android, beberapa library KTX menyediakan CoroutineScope sendiri untuk class siklus proses tertentu. Misalnya, ViewModel memiliki viewModelScope, dan Lifecycle memiliki lifecycleScope. Namun, tidak seperti dispatcher, CoroutineScope tidak menjalankan coroutine.

viewModelScope juga digunakan dalam contoh yang ditemukan di Mengelola thread latar belakang di Android dengan Coroutine. Namun, jika perlu membuat CoroutineScope sendiri untuk mengontrol siklus proses coroutine pada lapisan aplikasi tertentu, Anda dapat membuatnya sebagai berikut:

```
class ExampleClass {

    // Job and Dispatcher are combined into a CoroutineContext which
    // will be discussed shortly
    val scope = CoroutineScope(Job() + Dispatchers.Main)

    fun exampleMethod() {
        // Starts a new coroutine within the scope
        scope.launch {
            // New coroutine that can call suspend functions
            fetchDocs()
        }
    }

    fun cleanUp() {
        // Cancel the scope to cancel ongoing coroutines work
        scope.cancel()
    }
}
```

Gambar 5. CoroutineScope example

Cakupan yang dibatalkan tidak dapat membuat lebih banyak coroutine. Oleh karena itu, sebaiknya hanya panggil scope.cancel() ketika class yang mengontrol siklus prosesnya sedang dihapus. Saat menggunakan viewModelScope, class ViewModel akan membatalkan cakupan secara otomatis di metode onCleared() ViewModel.

Tugas

Job adalah tuas untuk coroutine. Setiap coroutine yang Anda buat dengan launch atau async akan menampilkan instance Job yang mengidentifikasi coroutine secara unik dan mengelola siklus prosesnya. Anda juga dapat meneruskan Job ke CoroutineScope untuk mengelola siklus prosesnya lebih lanjut, seperti pada contoh berikut:

```
class ExampleClass {
    ...
    fun exampleMethod() {
        // Handle to the coroutine, you can control its lifecycle
        val job = scope.launch {
            // New coroutine
        }

        if (...) {
            // Cancel the coroutine started above, this doesn't affect the scope
            // this coroutine was launched in
            job.cancel()
        }
    }
}
```

Gambar 6. Job concept in coroutines example

CoroutineContext

CoroutineContext menentukan perilaku coroutine menggunakan serangkaian elemen berikut:

- Job: Mengontrol siklus proses coroutine.
- CoroutineDispatcher: Mengirimkan pekerjaan ke thread yang sesuai.
- CoroutineName: Nama coroutine, berguna untuk proses debug.
- CoroutineExceptionHandler: Menangani pengecualian yang tidak terdeteksi.

Untuk coroutine baru yang dibuat dalam cakupan, instance Job baru akan ditetapkan ke coroutine baru, dan elemen CoroutineContext lainnya diwariskan dari cakupan yang menampungnya. Anda dapat mengganti elemen yang diwariskan dengan meneruskan CoroutineContext baru ke fungsi launch atau async. Perlu diketahui bahwa meneruskan Job ke launch atau async tidak akan berpengaruh, karena instance baru Job selalu ditetapkan ke coroutine baru.

```
class ExampleClass {
    val scope = CoroutineScope(Job() + Dispatchers.Main)

    fun exampleMethod() {
        // Starts a new coroutine on Dispatchers.Main as it's the scope's default
        val job1 = scope.launch {
            // New coroutine with CoroutineName = "coroutine" (default)
        }

        // Starts a new coroutine on Dispatchers.Default
        val job2 = scope.launch(Dispatchers.Default + "BackgroundCoroutine") {
            // New coroutine with CoroutineName = "BackgroundCoroutine" (overridden)
        }
    }
}
```

Gambar 7. CoroutineContext concept in coroutines example

2. Memahami Dispatchers

Coroutine Kotlin menggunakan dispatcher untuk menentukan thread yang digunakan untuk eksekusi coroutine. Untuk menjalankan kode di luar thread utama, Anda dapat memberi tahu coroutine Kotlin untuk melakukan pekerjaan pada dispatcher Default atau IO. Di Kotlin, semua coroutine harus dijalankan dalam dispatcher, meskipun sedang berjalan di thread utama. Coroutine dapat menangguhkan dirinya sendiri, dan dispatcher bertanggung jawab untuk melanjutkannya.

Untuk menentukan tempat menjalankan coroutine, Kotlin menyediakan tiga dispatcher yang dapat Anda gunakan:

- Dispatchers.Main - Gunakan dispatcher ini untuk menjalankan coroutine pada thread utama Android. Dispatcher ini harus digunakan hanya untuk berinteraksi dengan UI dan melakukan pekerjaan cepat. Contohnya mencakup pemanggilan fungsi suspend, menjalankan operasi framework UI Android, dan mengupdate objek LiveData.
- Dispatchers.IO - Dispatcher ini dioptimalkan untuk menjalankan disk atau I/O jaringan di luar thread utama. Contohnya termasuk menggunakan Komponen room, membaca dari atau menulis ke file, dan menjalankan operasi jaringan apa pun.
- Dispatchers.Default - Dispatcher ini dioptimalkan untuk melakukan pekerjaan yang membebani CPU di luar thread utama. Contoh kasus penggunaan mencakup pengurutan daftar dan penguraian JSON.

Melanjutkan contoh sebelumnya, Anda dapat menggunakan dispatcher untuk menentukan ulang fungsi get. Di dalam isi get, panggil withContext(Dispatchers.IO) untuk membuat blok yang berjalan di pool thread IO. Setiap kode yang Anda masukkan ke dalam blok tersebut selalu dijalankan melalui dispatcher IO. Karena withContext itu sendiri memegang fungsi penangguhan, fungsi get juga merupakan fungsi penangguhan.

```
suspend fun fetchDocs() {                      // Dispatchers.Main
    val result = get("developer.android.com")  // Dispatchers.Main
    show(result)                               // Dispatchers.Main
}

suspend fun get(url: String) =                 // Dispatchers.Main
    withContext(Dispatchers.IO) {              // Dispatchers.IO (main-safety block)
        /* perform network IO here */          // Dispatchers.IO (main-safety block)
    }                                          // Dispatchers.Main
}
```

Gambar 8. Dispatchers concept in coroutines example

Dengan coroutine, Anda dapat mengirim thread dengan kontrol yang sangat baik. Karena withContext() memungkinkan Anda mengontrol kumpulan thread dari setiap baris kode tanpa memasukkan callback, Anda dapat menerapkannya pada fungsi yang sangat kecil seperti membaca dari database atau melakukan permintaan jaringan. Praktik yang baik adalah menggunakan withContext() untuk memastikan setiap fungsi berada dalam main-safe, yang berarti Anda dapat memanggil fungsi tersebut dari thread utama. Dengan cara ini, pemanggil tidak perlu memikirkan thread mana yang harus digunakan untuk menjalankan fungsi.

Pada contoh sebelumnya, fetchDocs() mengeksekusi di thread utama; tetapi dapat memanggil get dengan aman, yang melakukan permintaan jaringan di latar belakang. Karena coroutine mendukung suspend dan resume, coroutine pada thread utama dilanjutkan dengan hasil get segera setelah blok withContext selesai.

Memasukkan Dispatcher

Jangan meng-hardcode Dispatchers saat membuat coroutine baru atau memanggil withContext.

```
// DO inject Dispatchers
class NewsRepository(
    private val defaultDispatcher: CoroutineDispatcher = Dispatchers.Default
) {
    suspend fun loadNews() = withContext(defaultDispatcher) { /* ... */ }
}

// DO NOT hardcode Dispatchers
class NewsRepository {
    // DO NOT use Dispatchers.Default directly, inject it instead
    suspend fun loadNews() = withContext(Dispatchers.Default) { /* ... */ }
}
```

Gambar 9. Combining Dispatchers and withContext example

Pola injeksi dependensi ini mempermudah pengujian karena Anda dapat mengganti dispatcher tersebut dalam uji unit dan instrumentasi dengan TestCoroutineDispatcher untuk membuat pengujian Anda menjadi lebih deterministik.

Daftar Pustaka

https://developer.android.com/kotlin/coroutines
https://developer.android.com/kotlin/coroutines/coroutines-adv#main-safety
https://developer.android.com/kotlin/coroutines/coroutines-best-practices