Codando Assíncrono com Coroutines

Arildo Borges Jr / Guilherme Moresco Bisotto

TDC 2019 - Trilha Android e Kotlin

Responsabilidades da main thread

- Carregar layouts
- Desenhar as views
- Eventos de UI

ASYNC ao resgate

E como podemos fazer isso no android?

Callbacks

```
request1 { result1 ->
    request2 { result2 ->
        request3 { result3 ->
            // finally use results
        }
    }
}
```

RxJava

```
sampleObservable
        .subscribeOn(Schedulers.io())
        .observeOn(AndroidSchedulers.mainThread())
        .doOnSubscribe(sample_loader.show())
        .doFinally(sample_loader.hide())
        .subscribeBy(
                onSuccess = { print(it.sampleText) },
                onError = { print(it) }
        )
```

Coroutines

Don’t block
Keep moving

O que são coroutines?

- É um design pattern de **concorrência**, que permite execução de código async de maneira **simples**. Criado na década de 60, possui implementação em **diversas linguagens**;

- **Evita** os chamados callback hells;

- Possibilita execução em **threads secundárias**.

Mas antes, vamos conhecer alguns conceitos...

- Job
- Context
- Scope
- Dispatchers

```
private val job = Job()
private val coroutineScope = CoroutineScope(Job + Dispatchers.Main)

fun someFunction() {

  coroutineScope.launch {

    binding.txtTitle = “”

    val someData = withContext(Dispatchers.IO) {
      someRepo.loadSomeData()
    }

    // updating UI using someData
  }
}

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

Job

- Representa uma **tarefa** ou um **conjunto** de tarefas;
- Pode possuir “filhos”;
- **Launch** devolve um **Job**;
- Pode ser cancelado utilizando o **cancel()**.

```
private val job = Job()
private val coroutineScope = CoroutineScope(Job + Dispatchers.Main)

fun someFunction() {

  coroutineScope.launch {

    binding.txtTitle = “”

    val someData = withContext(Dispatchers.IO) {
      someRepo.loadSomeData()
    }

    // updating UI using someData
  }
}

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

Context

- O conjunto de atributos que configuram as coroutines é representado pela interface **CoroutineContext**;
- Política de threading, tratamento de exceções, etc.

```
private val job = Job()
private val coroutineScope = CoroutineScope(Job + Dispatchers.Main)

fun someFunction() {

  coroutineScope.launch {

    binding.txtTitle = “”

    val someData = withContext(Dispatchers.IO) {
      someRepo.loadSomeData()
    }

    // updating UI using someData
  }
}

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

Scope

- Coroutines **sempre** rodam em um escopo;
- Serve como uma espécie de **ciclo de vida** para um conjunto de coroutines;
- Permite um maior **controle** das tarefas em execução.

```
private val job = Job()
private val coroutineScope = CoroutineScope(Job + Dispatchers.Main)

fun someFunction() {

  coroutineScope.launch {

    binding.txtTitle = ""

    val someData = withContext(Dispatchers.IO) {
      someRepo.loadSomeData()
    }

    // updating UI using someData
  }
}

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

Dispatchers

- **Default**: para processos que usam a CPU mais intensamente;
- **IO**: para tarefas de rede ou arquivos. O pool de threads é compartilhado com o dispatcher DEFAULT.
- **Main**: main thread do Android

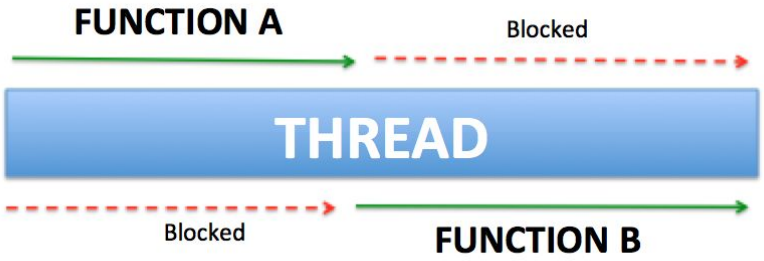
```
private val job = Job()
private val coroutineScope = CoroutineScope(Job + Dispatchers.Main)

fun someFunction() {

  coroutineScope.launch {

    binding.txtTitle = “”

    val someData = withContext(Dispatchers.IO) {
      someRepo.loadSomeData()
    }

    // updating UI using someData
  }
}

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

Suspend

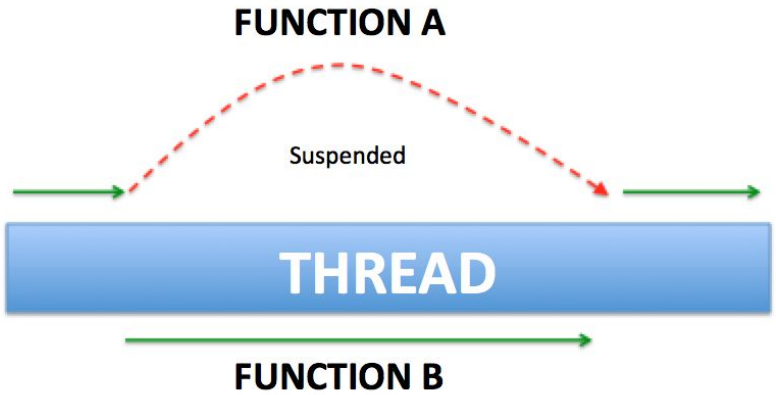
Funções suspensas são basicamente o centro de toda mágica em Coroutines, isto porque:

- Podem ser **pausadas** e **retomadas**;
- Podem **aguardar** a execução de tarefas mais prioritárias **sem bloquear** a thread atual e dar **continuidade** a execução quando houver o retorno do resultado;
- Devem ser chamadas por outras funções suspensas;
- Por default não são "assíncronas";
- Sintaxe simples, pois apenas é adicionado o modificador **suspend** nas funções.

Suspend



blocking function



suspend function

IO manipulando UI

```
launch {

    // get character on IO thread
    withContext(Dispatchers.IO) {
        val character = repository.getCharacter()

        binding.tvCharacterName.text = character.name
        // Error - Only main thread can update UI
    }
}
```

```
2019-11-26 10:34:05.783 8433-8433/com.arildo_guilherme.meetupcoroutines E/AndroidRuntime: FATAL EXCEPTION: main
    Process: com.arildo_guilherme.meetupcoroutines, PID: 8433
    android.view.ViewRootImpl$CalledFromWrongThreadException: Only the original thread that created a view hierarchy can touch its views.
```

Request na Main Thread

```
launch {

    withContext(Dispatchers.Main) {
        val character = repository.getCharacter()

        binding.tvCharacterName.text = character.name
    }

}
```

```
2019-04-23 15:09:19.261 23435-23435/ie.redstudio.dzoneapplicationone E/AndroidRuntime: FATAL EXCEPTION: main
    Process: ie.redstudio.dzoneapplicationone, PID: 23435
    java.lang.IllegalStateException: Cannot access database on the main thread since it may potentially lock the UI for a long period of time.
```

Exemplo de uso

```
interface CharacterApiService {

    @GET("/character")
    suspend fun getCharacter(): Response<Character>

}
```

Retrofit com funções suspensas

```
launch {

    // get character on IO thread
    val character = withContext(Dispatchers.IO) {
        val char = repository.getCharacter()

        Log.d(character.name, character.power)
        char
    }

    // thread UI on Main Thread
    binding.tvCharacterName.text = character.name
}
```

Jetpack + Lifecycle

```
// Activity

// scope

lifecycleScope

// launch

lifecycleScope.launchWhenCreated { }

lifecycleScope.launchWhenStarted { }

lifecycleScope.launchWhenResumed { }
```

```
// Fragment

// scope

lifecycleScope

// Activity Scope for Fragment

viewLifecycleOwner.lifecyleScope

// launch

lifecycleScope.launchWhenCreated { }

lifecycleScope.launchWhenStarted { }

lifecycleScope.launchWhenResumed { }
```

```
// ViewModel

// scope

viewModelScope

// launch

viewModelScope.launch { }
```

Launch vs Async

- "Fire and forget";
- Retorna um Job e não um resultado.

```
val time = measureTimeMillis {
    val char1 = withContext(Dispatchers.IO) {
        repository.getCharacter(1)
    }

    val char2 = withContext(Dispatchers.IO) {
        repository.getCharacter(2)
    }

    val char3 = withContext(Dispatchers.IO) {
        repository.getCharacter(3)
    }
}

// measuredTime: 3.1 seconds
```

- Retorna um Deferred que possibilita obter o resultado.

```
val time = measureTimeMillis {
    val char1 = async(Dispatchers.IO) {
        repository.getCharacter(1)
    }

    val char2 = async(Dispatchers.IO) {
        repository.getCharacter(2)
    }

    val char3 = async(Dispatchers.IO) {
        repository.getCharacter(3)
    }

    Log.d("${char1.await()} / ${char1.await()} / ${char1.await()}")

}
// measuredTime: 1.2 seconds
```

E como é na prática…

```
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)

    binding.viewModel = viewModel
    binding.rvSongs.adapter = songsAdapter

    lifecycleScope.launch {
        viewModel.getSongs()
    }
}
```

Activity

E como é na prática…

```
suspend fun getSongs() = withContext(Dispatchers.IO) {
    try {
        val response = songsRepository.getSongs()

        if (response.isSuccessful) {
            response.body()?.let { songList ->
                _songs.postValue(songList)
            }
        } else {
            _error.postValue(ErrorOnLoadSongs)
        }


    } catch (e: Exception) {
        // error handler
        _error.postValue(ErrorOnLoadSongs)
    }
}
```

ViewModel

E como é na prática…

```
class SongsRepositoryImpl(private val songsRemoteDataSource: SongsDataSource.Remote)  {

    suspend fun getSongs(): Response<List<Song>> {
        return songsRemoteDataSource.getSongs()
    }

}
```

Repositório

E como é na prática…

```
class SongsRemoteDataSource(private val apiService: SongsApiService) : SongsDataSource.Remote {

    override suspend fun getSongs(): Response<List<Song>> {
        return apiService.getSongs()
    }

}
```

DataSource.Remote

E como é na prática…

```
interface SongsApiService {

    @GET("/songs")
    suspend fun getSongs(): Response<List<Song>>

}
```

API

E o que mais?

- Channel
- Flow
- Escopo de LiveData com suporte a CoroutineContext

Referências

- Developer Kotlin Coroutines
- Repositório Meetup Coroutines
- nglauber - Aplicações Assíncronas no Android com Coroutines e Jetpack
- KotlinLang - Composing Suspend Functions
- Coroutines Overview

Obrigado!

Arildo Borges Jr

/arildojr7

/arildoborgesjr

Guilherme M. Bisotto

/GuilhermeMBisotto

/guilhermebisotto

THE DEVELOPER'S CONFERENCE