

COROUTINES

Vou explicar um pouco sobre uma ferramenta que adotamos na nova arquitetura nos nossos projetos no Itau, e um dos motivos da escolha é a curva de aprendizagem que é bem menor que a do RX, e com conceitos bem básicos e simples ja te ajuda criar varias possibilidades.

Nos exemplos vou utilizar uma aplicação mobile kotlin utilizando o MVVM para auxiliar a explicação

Também vou usar retrofit e live data para simplificar alguns exemplos

O QUE SÃO COROUTINES?

- Operações que são executadas sem travar uma thread.
- Uma forma de escrever código Assíncrono
- Ja estão disponíveis desde a versão 1.1 do Kotlin



Ja estão disponíveis desde a versão 1.1. do kotlin

É um tipo de light weight thread(significa que não é uma thread mapeada na thread nativa e por isso não requer uma troca de contexto do processador e este o maior motivo de ser rápido)

ENTENDENDO

ANTES DE CRIAR PRECISAMOS ENTENDER

- Suspend Function
- Context e Dispatchers
- Scopes e Jobs
- Tipos de Execução

SUSPEND FUNCTIONS

- Qualquer função que pode ser pausada e retomada
- Exemplo usando Room:

```
@Dao
interface ListaDAO {

    @Query("SELECT * FROM gitHubRepo")
    suspend fun getRepositoriosCache(): MutableList<GitRepositorioEntity>

    @Insert(onConflict = OnConflictStrategy.REPLACE)
    suspend fun insertRepositorio(repositorio: GitRepositorioEntity)

    @Insert(onConflict = OnConflictStrategy.REPLACE)
    suspend fun insertAllRepositorio(repositorio: MutableList<GitRepositorioEntity>)
}
```

Basicamente ela executa e informa a thread que esta esperando algo terminar e que depois eles voltam a conversar

Isto é possível pq as Coroutines usam um padrão chamado CPS Continuation-Passing Style Que basicamente uma forma bonita de falar callback
Ele trabalha como se houvesse uma interface por trás, onde haverá um fim com sucesso e um fim com falha

Hoje o Retrofit e o Room ja nos devolvem suspend functions para que a gente possa trabalhar
Vamos criar um exemplo usando uma interface do room

A criação é bem similar a qualquer função no kotlin - cria a suspend function

SUSPEND FUNCTIONS

- Exemplo Usando RETROFIT

```
interface Service {  
    @GET("search/repositories")  
    suspend fun getHubRepos(  
        @Query(QUERY, encoded = true) query: String = KOTLIN_LANGUAGE,  
        @Query(SORT) sort: String = STARS,  
        @Query(PAGE) page: Int = INITIAL_PAGE  
    ): GitListaRepositoryResponse  
}
```

Basicamente ela executa e informa a thread que esta esperando algo terminar e que depois eles voltam a conversar

Isto é possível pq as Coroutines usam um padrão chamado CPS Continuation-Passing Style Que basicamente uma forma bonita de falar callback
Ele trabalha como se houvesse uma interface por trás, onde haverá um fim com sucesso e um fim com falha

Hoje o Retrofit e o Room ja nos devolvem suspend functions para que a gente possa trabalhar
Vamos criar um exemplo usando uma interface do room

A criação é bem similar a qualquer função no kotlin - cria a suspend function

CONTEXT & DISPATCHERS

- Dispatcher.UI
- Dispatcher.IO
- Dispatcher.Unconfined
- Dispatcher.Default

Um Dispatcher pode ser declarado no momento da chamada da coroutine, ou no momento em que você declara a suspend function, se não declarar o Dispatcher ele irá executar no Default

UI - assim como o nome diz iremos utilizar para atualização de qualquer objeto de ui

um bom exemplo é quando estamos usando uma coroutine dentro da viewmodel, e vamos atualizar a tela com alguma informação, ou vamos pedir para exibir o loading ou algo do tipo

IO - para qualquer objeto de entrada e saída, uma chamada de API, uma chamada

DEFAULT - qualquer outro processamento pesado, filtrar uma lista, popular um objeto, fazer parse de algo

Unconfined - Não é presa a nenhum tipo de Thread, ele irá sempre iniciar na thread atual, e caso ele mude o contexto em algum momento da execução, ele pode terminar em outro tipo de contexto de Thread.

DISPATCHERS EXEMPLO



SCOPES

- Coroutine Scope
- Global Scope
- ViewModel Scope
- LifeCycle Scope
- Main Scope

COROUTINE SCOPE

- É uma interface que somente possui o Coroutine Context
- É recomendado a utilização deste escopo

```
fun testaCoroutines(dispatcher: CoroutineDispatcher, tipolog: String) {
    runBlocking { this: CoroutineScope
        CoroutineScope(dispatcher).launch { this: CoroutineScope
            println(tipolog +"1 - Thread ao iniciar coroutine ${Thread.currentThread()} executou.")
        }
    }
}
```

Há também uma interface chamada CoroutineScope que consiste em uma única propriedade - val coroutineContext: CoroutineContext. Não tem mais nada além de um contexto. Então, por que existe e como é diferente de um contexto em si? A diferença entre um contexto e um escopo está na finalidade pretendida.

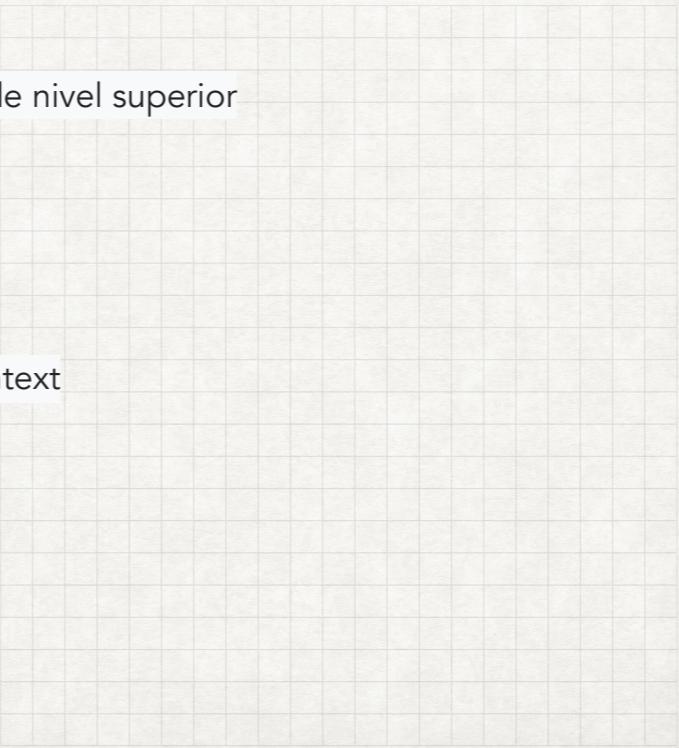
O importante é que você saiba que sempre que um novo escopo de rotina é criado, um novo Job é criado e associado a ele.

Todas as coroutines criadas usando esse escopo tornam-se filhos desse trabalho. É assim que um relacionamento pai-filho é criado entre corotinas.

Se alguma das coroutines lançar uma exceção não tratada, o trabalho dos pais será cancelado, o que acabará por cancelar todos os seus filhos.

GLOBAL SCOPE

- É usado para iniciar coroutines de nível superior
- Não é Recomendado
- Recebe Empty no CoroutineContext



- Um CoroutineScope global não é vinculado a nenhum trabalho.

O escopo global é usado para iniciar as corotinas de nível superior

Ou seja coroutines que estão operando em toda a vida útil do aplicativo e não são canceladas prematuramente.

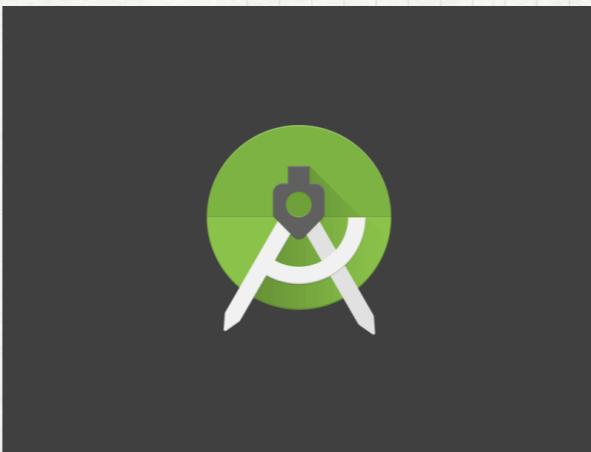
- Outro uso do escopo global são os operadores em Dispatchers. Não confinado, que não tem nenhum trabalho associado a eles.

- NÃO É RECOMENDADO por passar a responsabilidade de acompanhar a sua vida útil ao desenvolvedor

- Podemos usar uma abordagem com o GlobalScope, mantendo o controle manual das corotinas lançadas e aguardando sua conclusão usando join

MAIN SCOPE, VIEWMODEL SCOPE
E LIFECYCLE SCOPE

- São derivações do Coroutine Scope
- É recomendado a utilização deste escopo



São criados a partir do coroutine scope

O Main Scope

O viewmodel scope é um coroutine scope associado ao Dispatcher.main

E assim que viewmodel se encerra, ele cancela todos os trabalhos

Esta disponível na versão

Para ViewModelScope, use androidx.lifecycle:viewmodel-ktx:2.1.0-beta01 ou posterior.

O Lifecycle Scope

Um coroutine scope que pode ser controlado pelo ciclo de vida da View

ESTA DISPONIVEL NA VERSAO

androidx.lifecycle:lifecycle-runtime-ktx:2.2.0-alpha01 ou posterior.

ALGUMAS FORMAS DE EXECUÇÕES

- RunBlocking
- Launch
- Async

Existem várias maneiras possíveis de criar coroutines com base em diferentes requisitos. Nesta seção, vamos dar uma olhada em alguns deles

Os Coroutines Builders mais comuns são o launch e o async
Para iniciar uma execução de coroutine é necessário ter um escopo associado

RUNBLOCKING

- Executa uma nova rotina e bloqueia a thread atual até a conclusão da execução

```
@Test
fun `quando chamar a Api para a proxima pagina retornar sucesso e retornar os dados da api`() {
    val repository = ListaRepository(service, database)
    runBlocking {
        val lista = repository.listaRepositoriosProximaPagina(1)
        Assert.assertEquals(2, lista.size)
        Assert.assertFalse(repository.dadosCache)
    }
}
```

Ele foi projetado para conectar o código de bloqueio regular a bibliotecas escritas no estilo de suspensão, para serem usadas nas funções principais e nos testes.

A corotina é cancelada quando o job resultante é cancelado.

LAUNCH

Inicia uma nova corotina sem bloquear a thread atual e retorna uma referência à corotina como um trabalho.

```
private fun proximaPaginaRepositorio() {
    viewModelScope.launch {
        try {
            if (useCase.aListaEstaLocal()) {
                event.value = ListaGitHubEvent.ExibeInformacaoCache(View.VISIBLE)
            } else {
                state.value = ListaGitHubStates.ListaGitHubSucesso(useCase.listarProximaPaginaRepositoriosGitHub())
            }
        } catch (exception: Exception) {
            exception.printStackTrace()
            Log.e("Viewmodel", exception.message)
        }
    }
}
```

Inicia uma nova corotina sem bloquear a thread atual e retorna uma referência à corotina como um trabalho.

A coroutine é cancelada quando o seu trabalho é cancelado

ENTENDENDO

ANTES DO ASYNC - DEFERRED

- Objeto Futuro
- Promessa
- Uso com o. Async



É uma promessa de um resultado, um bom exemplo é feito quando usamos o. Async da coroutine, uma forma de iniciar varias rotinas de forma asincronas, e o retorno deles não é um objeto contrato mas o que chamamos de deferred

Uma forma de obter o resultado seria usando o .await() que ira aguardar a execução desta coroutine e ter um objeto concreto

ASYNC

- Executa uma nova rotina e bloqueia a thread atual até a conclusão da execução

EXEMPLO



Ele foi projetado para conectar o código de bloqueio regular a bibliotecas escritas no estilo de suspensão, para serem usadas nas funções principais e nos testes.

A corotina é cancelada quando o job resultante é cancelado.

JOB

- É uma atividade cancelável
- Podem ser divididos em Hierarquias
- Um escopo sempre é associado a um Job

Um JOB é um trabalho em segundo plano.

Conceitualmente, um Job é uma coisa cancelável com um ciclo de vida que culmina na sua conclusão.

Os trabalhos podem ser organizados em hierarquias pai-filho, onde o cancelamento de um pai leva ao cancelamento imediato de todos os seus filhos. A falha ou cancelamento de um filho com uma exceção diferente de CancellationException cancela imediatamente seu pai. Dessa forma, um pai ou mãe pode cancelar seus próprios filhos (incluindo todos os filhos recursivamente) sem se cancelar.

SUPERVISOR JOB

- Os filhos deste job podem falhar de forma independente
- É o mesmo job que o viewmodelscope e o lifecycle usam

Um JOB é um trabalho em segundo plano.

Conceitualmente, um Job é uma coisa cancelável com um ciclo de vida que culmina na sua conclusão.

Os trabalhos podem ser organizados em hierarquias pai-filho, onde o cancelamento de um pai leva ao cancelamento imediato de todos os seus filhos. A falha ou cancelamento de um filho com uma exceção diferente de CancellationException cancela imediatamente seu pai. Dessa forma, um pai ou mãe pode cancelar seus próprios filhos (incluindo todos os filhos recursivamente) sem se cancelar.

COROUTINES EXEMPLO



E TEM MAIS

- Channel
- Flow
- ...

Um CoroutineScope global não é vinculado a nenhum trabalho.

O escopo global é usado para iniciar as corotinas de nível superior que estão operando em toda a vida útil do aplicativo e não são canceladas prematuramente. Outro uso do escopo global são os operadores em Dispatchers. Não confinado, que não tem nenhum trabalho associado a eles.

O código do aplicativo geralmente deve usar um CoroutineScope definido pelo aplicativo. O uso de assíncrono ou inicialização na instância do GlobalScope é altamente desencorajado.

O uso dessa interface pode ser assim:

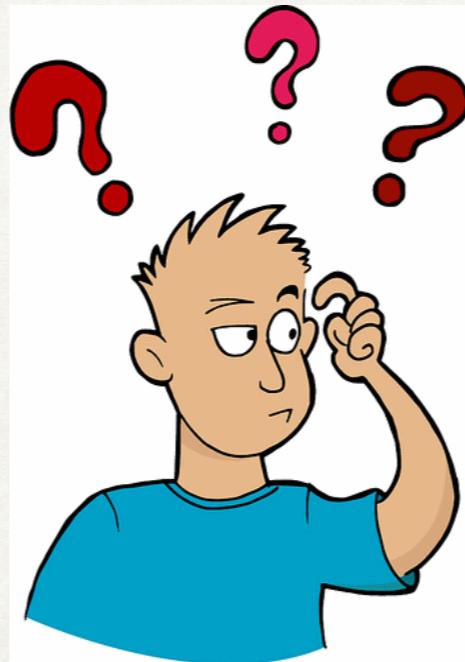
LINKS UTEIS
PARA MAIS CONHECIMENTO DE COROUTINES



 SCAN ME

A black rounded rectangular button containing a white smartphone icon on the left and the text "SCAN ME" in white capital letters on the right.

DÚVIDAS



CONTATOS

RENATO RIBEIRO DOS SANTOS

- github: <https://github.com/rribesa>
- Linkedin: <https://www.linkedin.com/in/renato-santos-9a25b221/>

