Sombras do Kotlin

Coroutines Imortais

Sérgio Ribeiro

Bem-vindo(a) às Sombras do Kotlin

O Despertar do Poder das Coroutines

Agora você entrará na jornada sombria e fascinante pelo jardim selvagem das coroutines, uma ferramenta poderosa e essencial para o desenvolvimento mobile Android. Inspirado pela série “Entrevista com o Vampiro”, baseada na obra literária de Anne Rice, este E-book foi elaborado para oferecer um vislumbre inicial à pessoa dev. iniciante, através de alguns conceitos e práticas fundamentais das coroutines.
Coroutines simplificam o código, melhoram a performance e oferecem um controle preciso sobre tarefas paralelas, sendo essencial para qualquer desenvolvedor Android.
Prepare-se para explorar este universo imortal, transformando suas habilidades de programação, tendo como guia as sombras das coroutines. Que este seja apenas o início de sua caminhada como dev. Android.

01

O NASCER DA LUA

Fundamentos de Coroutines

Fundamentos

Coroutines são uma maneira simples e eficiente de escrever código assíncrono em Kotlin. Elas permitem que você execute tarefas em paralelo sem bloquear a thread principal, diferente das abordagens tradicionais com threads, que podem ser complexas e propensas a erros. Com coroutines, você pode escrever código sequencial que é fácil de entender e manter.

Vantagens das Coroutines:

1.Simplicidade: Código mais limpo e fácil de ler.
2.Performance: Menor sobrecarga em comparação com threads.
3.Controle: Melhor gerenciamento do ciclo de vida das tarefas.

```kotlin
fun main() = runBlocking {
    launch {
        delay(1000L)
        println("Coroutines são simples!")
    }
    println("Início")
}

```

Neste exemplo, launch inicia uma coroutine que espera 1 segundo antes de imprimir a mensagem. A função runBlocking bloqueia a thread principal até que a coroutine termine.

02

O CAMINHO DA NOITE

Manipulação de Coroutines com Scopes

Manipulação com Scopes

Coroutines são gerenciadas por CoroutineScope, que define o contexto no qual elas serão executadas. Usar CoroutineScope corretamente é essencial para garantir que as coroutines sejam gerenciadas eficientemente e que seus recursos sejam liberados quando necessário.

CoroutineScope e GlobalScope

CoroutineScope é uma interface que define um escopo para novas coroutines. Ao lançar uma coroutine dentro dele, ela herda o contexto e o ciclo de vida daquele escopo. Já o GlobalScope, é um escopo global não vinculado ao ciclo de vida de nenhum componente.

```
1  fun main() {
2      GlobalScope.launch {
3          delay(1000L)
4          println("GlobalScope Coroutine")
5      }
6      println("Main Function")
7      Thread.sleep(2000L)
8  }
9
```

Usar GlobalScope pode ser perigoso, pois suas coroutines vivem enquanto a aplicação estiver rodando, o que pode levar a vazamentos de memória.

viewModelScope

viewModelScope é um escopo fornecido pela biblioteca de ciclo de vida do Android. Ele é vinculado ao ciclo de vida do ViewModel e é automaticamente cancelado quando o ViewModel é destruído, garantindo que suas coroutines não vazem memória.

```
1  class MyViewModel : ViewModel() {
2      fun fetchData() {
3          viewModelScope.launch {
4              val data = fetchFromNetwork()
5              // Atualizar UI com os dados
6          }
7      }
8  }
9
```

Neste exemplo, a coroutine será cancelada automaticamente se o ViewModel for destruído, evitando problemas de vazamento de memória.

Utilizando CoroutineScope de Forma Eficiente

Criar seu próprio CoroutineScope pode ser útil para tarefas que precisam de um controle específico sobre o ciclo de vida. É importante cancelar manualmente esse escopo para liberar recursos corretamente.

```kotlin
class MyActivity : AppCompatActivity() {
    private val scope = CoroutineScope(Dispatchers.Main + Job())

    override fun onDestroy() {
        super.onDestroy()
        scope.cancel() // Cancela todas as coroutines
    }

    fun loadData() {
        scope.launch {
            val data = fetchData()
            // Atualizar UI com os dados
        }
    }
}
```

Aqui, um escopo personalizado é criado com Dispatchers.Main e um Job. Cancelando o escopo no onDestroy, todas as coroutines são canceladas, impedindo que haja vazamento de memória. Escolher o escopo apropriado para cada tarefa garante que suas coroutines sejam executadas de maneira controlada, segura e eficiente, aproveitando ao máximo o poder das coroutines em Kotlin.

03

O CHAMADO SOMBRIO

Funções Suspensas

Funções Suspensas

Funções suspensas são funções que podem pausar sua execução sem bloquear a thread. Elas são marcadas com a palavra-chave suspend e podem ser usadas apenas dentro de coroutines ou outras funções suspensas. Isso facilita o trabalho com operações assíncronas, como chamadas de rede.

Criando uma Função Suspensa:

```
1  suspend fun fetchData(): String {
2      delay(2000L)
3      return "Dados recebidos!"
4  }
5
```

Usando em um Projeto Android:

```
1  fun loadData() {
2      CoroutineScope(Dispatchers.IO).launch {
3          val data = fetchData()
4          withContext(Dispatchers.Main) {
5              println(data)
6          }
7      }
8  }
9
```

Neste exemplo, fetchData simula uma chamada de rede, e loadData usa coroutines para buscar dados em um thread de I/O e atualizar a UI na thread principal.

04

O RITMO DAS TREVAS

Controle e Sincronização

Controle e Sincronização

Controlar o fluxo de execução das coroutines é essencial para tarefas complexas. Você pode usar funções como withContext, yield e join para gerenciar a execução e sincronização.

Exemplo de Uso do withContext:

```
1  suspend fun processData() {
2      withContext(Dispatchers.Default) {
3          // Código pesado
4      }
5  }
6
```

Uso do yield para Cooperatividade:

```kotlin
suspend fun compute() {
    repeat(1000) {
        // Tarefa pesada
        yield() // Permite que outras coroutines rodem
    }
}

```

Exemplo de join:

```kotlin
fun main() = runBlocking {
    val job = launch {
        delay(1000L)
        println("Coroutine completada")
    }
    job.join() // Espera até a coroutine terminar
    println("Fim")
}

```

Com funções assim, você pode otimizar a execução das coroutines, garantindo que elas funcionem de maneira eficiente e cooperativa.

05

A NOITE SILENCIOSA

Cancelamento e Exceções

Cancelamento e Exceções

Cancelar coroutines corretamente é crucial para liberar recursos e evitar comportamentos inesperados. Você pode usar job.cancel() para cancelar uma coroutine e try/catch para lidar com exceções dentro de coroutines.

Cancelando uma Coroutine:

```
1  fun main() = runBlocking {
2      val job = launch {
3          repeat(1000) {
4              println("Trabalhando $it")
5              delay(500L)
6          }
7      }
8      delay(1300L)
9      job.cancelAndJoin() // Cancela e espera o fim da coroutine
10     println("Coroutine cancelada")
11 }
12
```

Tratamento de Exceções:

```kotlin
fun main() = runBlocking {
    val job = launch {
        try {
            repeat(1000) {
                println("Trabalhando $it")
                delay(500L)
            }
        } catch (e: CancellationException) {
            println("Coroutine cancelada com exceção: $e")
        } finally {
            println("Liberação de recursos")
        }
    }
    delay(1300L)
    job.cancelAndJoin()
}

```

Esses exemplos mostram como cancelar e lidar com exceções, garantindo que suas coroutines sejam seguras e robustas.

Conclusão

Muito obrigado por ler até aqui!

Esse Ebook foi feito e diagramado por um humano, com auxílio de inteligência artificial na elaboração, revisão e geração de imagens. Esse conteúdo foi desenvolvido sem fins lucrativos, com objetivo de estudo, seguindo a programação do curso “Formação ChatGPT for Devs”, da DIO (Digital Inovation One), ministrado pelo Felipe Aguiar.

Espero que tenha gostado do conteúdo e que ele tenha te ajudado de alguma forma. A gente se despede por aqui, mas se quiser trocar ideias sobre programação, android, kotlin (e porque não sobre vampiros? Hahaha) é só me adicionar no LinkedIn. Um abraço e tudo de bom!