Table of contents

浅谈Kotlin中的协程

引言

在现代软件开发中，并发编程和异步任务处理变得越来越重要。无论是处理大量的网络请求、管理后台任务，还是实现实时响应的用户界面，开发者都需要一种高效且简洁的方式来处理这些挑战。在传统的编程语言中，实现并发和异步操作通常需要使用复杂的线程模型和回调机制，它们臃肿，增加了无谓的复杂度，难以维护。

Kotlin作为一种现代编程语言，为开发者提供了一种强大，优雅的解决方案——协程。协程是一种轻量级的线程，能够以简洁的代码实现复杂的并发和异步操作。与传统线程相比，协程具有更低的开销和更高的性能，能够显著提升程序的响应速度和资源利用效率。

我们将逐步深入探讨Kotlin协程的各个方面，从基础概念到高级用法。

由于Kotlin协程本身属于无栈协程，本文更多介绍无栈协程。

基本概念

协程是一种用于并发编程的轻量级工具，可以被挂起/恢复。它具有以下特点：

- 轻量级：协程并不需要负责新建线程/进程资源，因此相较于传统的线程更为轻量
- 可挂起/恢复：协程可以在执行IO等耗时操作时挂起，避免阻塞
- 结构化并发： 协程的生命周期可以被显式控制，方便管理
- 允许顺序编程：协程在实际应用时就像普通的同步代码一样，有助于提高可读性

Kotlin协程

suspend 函数

在Kotlin中，定义一个suspend函数非常容易：

```
suspend fun runWithCoroutines(): String {
    delay(1000)
    return "data"
```

```
}
val value = runWithCoroutines()
```

它会模拟一个耗时一秒的操作之后返回数据，同时不会阻塞主线程，无需使用回调。

协程构建器

Kotlin提供了一些协程构建器来启动协程。最常用的构建器有`launch` `async` `runBlocking`

launch:

launch构建器用于启动一个新协程，它类似于传统的新建线程，适用于不需要返回结果的后台任务。

```
GlobalScope.launch{
    val data = runWithCoroutines()
    println(data)
}
```

这段代码启动了一个新的协程，在GlobalScope(全局作用域)中运行`runWithCoroutines()`函数，并且在获取到结果之后打印出来。

async:

async会返回一个`Deferred`对象，类似于JavaScript中的Promise和Java中的Future，通过await这个对象来获取结果，类似于：

```
val deferred:Deferred<String> = GlobalScope.async {
    runWithCoroutines()
}
runBlocking {
    val data = deferred.await()
    println(data)
}
```

在这个示例中，async启动了一个新协程，并返回一个Deferred对象。

runBlocking:

runBlocking用于启动一个新的协程，并且会阻塞线程直到协程执行完成，通常用来在非协程环境中使用协程.

它的用法可以参见 `async` 的代码块。

协程上下文和调度器

协程上下文是协程执行时的环境，包含协程的调度器、工作模式和其它上下文元素。Dispatchers是Kotlin提供的调度器，用于指定协程在哪个线程池执行。

通过在`GlobalScope.launch()`方法中传入Dispatchers参数来选择调度器，可选的调度器有：

- Dispatchers.Default

Dispatchers.Default用于CPU密集型任务，默认情况下会使用共享的后台线程池。

- Dispatchers.IO

Dispatchers.IO用于IO密集型任务，如文件读写和网络请求，它会使用一个共享的线程池，线程数比Dispatchers.Default更多。

- Dispatchers.Main

Dispatchers.Main用于在主线程上执行UI相关的任务，通常在Android开发中使用。

- Dispatchers.Unconfined

Dispatchers.Unconfined不限制协程的执行线程，协程会在当前线程执行，直到遇到第一个挂起点。之后，协程恢复执行时的线程将由具体的挂起函数决定。例如：

协程的使用场景

协程在并发编程、异步任务和长时间运行的任务中作用巨大，假设我们现在有一个并发网络请求:

并发编程

```
suspend fun fetchServerData(serverUrl: String): String {
    delay(1000) // 模拟网络延迟
    return "Data from $serverUrl"
}
```

```
fun main() = runBlocking {
    val urls = listOf("server1.com", "server2.com", "server3.com")
    val deferreds = urls.map { url ->
        async { fetchServerData(url) }
    }
    val results = deferreds.awaitAll()
    println(results)
}
```

在这个示例中，我们使用async并发地发送请求，并通过awaitAll汇总结果。这样可以显著减少请求时间，提高效率。

异步任务

在许多应用中，IO等任务通常需要异步执行，使用协程可以有效优化异步执行的效率，同时避免臭名昭著的回调地狱。还是来看代码：

```
suspend fun readFileAsync(filePath: String): String {
    return withContext(Dispatchers.IO) {
        File(filePath).readText()
    }
}

fun main() = runBlocking {
    val filePath = "file.txt"
    val fileContent = readFileAsync(filePath)
    println(fileContent)
}
```

在这个示例中，我们使用withContext切换到IO调度器进行文件读取操作，从而避免阻塞主线程。

长时间任务

同样，如果现在有一个耗时很长的任务，比如IO和计算密集型任务，通过协程避免阻塞主线程也是一个很好的解决方案。

```
suspend fun processData(): String {
    delay(3000)
    return "Processing complete"
```

```
}

fun main() = runBlocking {
    val job = GlobalScope.launch {
        val result = processData()
        println(result)
    }
    println("Processing started")
    job.join()
    println("Processing ended")
}
```

在这个示例中，我们使用launch启动了一个后台任务，并在任务完成后打印结果。通过`job.join()`等待任务完成，这样可以确保任务在后台顺利运行，不阻塞主线程。

协程的控制结构

协程作用域

协程作用域(CoroutineScope)定义了协程的生命周期和上下文，它是管理协程的基础，确保协程在作用域内启动并在作用域结束时自动取消。

常用的作用域有：

- GlobalScope

它的生命周期和应用程序一样长，除非手动调用`cancel`方法，否则它启动的携程在应用程序启动期间不会自动取消。

- CoroutineScope

它为协程的生命周期提供了上下文。通过 CoroutineScope 启动的协程会受到作用域的管理，作用域结束时会自动取消所有未完成的协程。CoroutineScope 是结构化并发的基础，它确保协程的生命周期与其作用域绑定，从而提高代码的可靠性和可维护性。

```
class MyCoroutineScope : CoroutineScope {
    private val job = Job()
    override val coroutineContext = Dispatchers.Default + job
```

```
    fun doWork() {
        launch {
            delay(1000)
            println("Task from custom CoroutineScope")
        }
    }

    fun cancel() {
        job.cancel()
    }
}

fun main() {
    val myScope = MyCoroutineScope()
    myScope.doWork()

    Thread.sleep(2000) //阻塞主线程来等待
    myScope.cancel()
    println("Main thread ends")
}
```

在这个示例中，我们创建了一个自定义的 CoroutineScope，并在其中启动了一个协程。自定义 CoroutineScope 使用 Job 和 Dispatchers.Default 作为协程上下文，确保协程在 Job 取消时自动结束。

```
suspend fun doWork() = coroutineScope {
    launch {
        delay(1000)
        println("Task 1 from coroutineScope")
    }
    launch {
        delay(2000)
        println("Task 2 from coroutineScope")
    }
    println("Tasks started in coroutineScope")
}

fun main() = runBlocking {
```

```
    doWork()
    println("All tasks completed")
}
```

在这个示例中，coroutineScope 构建器创建了一个局部作用域，确保所有在该作用域内启动的协程在作用域结束前完成。

协程的取消和超时

和普通的线程一样，协程也可以取消，当调用`cancel`方法时，协程会被取消并抛出`CancellationException`， 可以检查`isActive`属性来确定协程是否活动。

```
fun main() = runBlocking {
    val job = launch {
        repeat(1000) { i ->
            println("working $i ...")
            delay(500)
        }
    }
    delay(1300)
    println("what can I say?")
    job.cancelAndJoin()
    println("job out")
}
```

主线程在等待一段时间之后取消了子协程。

协程超时 (withTimeout)

withTimeout 函数用于指定协程的最长执行时间，如果协程在指定时间内未完成，则会抛出 TimeoutCancellationException。

```
fun main() = runBlocking {
    try {
        withTimeout(1300) {
            repeat(1000) { i ->
                println("working $i ...")
                delay(500)
            }
        }
```

```
    } catch (e: TimeoutCancellationException) {
        println("job out")
    }
}
```

在这个示例中，withTimeout 限制了协程的执行时间，如果超过1300毫秒，协程将被取消。

协程中的异常处理

好吧，是程序都会抛出异常。在传统的多线程编程中，通常会在每个 Runnable 内部使用 try-catch 块来处理可能抛出的异常

```
Runnable tryCatchRunnable = () -> {
    try {
        throw new Exception();
    } catch (Exception e) {
        System.out.println("Caught exception: " + e.getMessage());
    }
};
```

在Kotlin协程中，异常处理也是一个重要的方面。

异常传播:

子协程的异常会传播到父协程，所以在coroutineScope中可以捕获它：

```
fun main() = runBlocking {
    try {
        coroutineScope {
            launch {
                delay(1000)
                throw Exception("Error in child coroutine")
            }
        }
    } catch (e: Exception) {
        println("Caught exception: ${e.message}")
    }
}
```

异常处理：

最简单有效的方法是直接`try-catch`，在此不再做举例。

另一种是使用`supervisorScope`来进行处理： supervisorScope 是一种特殊的作用域，它确保子协程之间的异常不会相互传播。这意味着一个子协程的失败不会影响其他子协程。

```
fun main() = runBlocking {
    supervisorScope {
        val child1 = launch {
            try {
                delay(1000)
                throw Exception("Error in child1")
            } catch (e: Exception) {
                println("Caught exception in child1: ${e.message}")
            }
        }
        val child2 = launch {
            delay(2000)
            println("Child2 completed successfully")
        }
        child1.join()
        child2.join()
    }
    println("Supervisor scope completed")
}
```

在这个示例中，supervisorScope 确保子协程 child2 不会受到 child1 异常的影响，并且能够正常完成。

同时，可以使用`CoroutineExceptionHandler`来处理和捕获没有在`try-catch`中捕获的异常。

```
fun main() = runBlocking {
    val handler = CoroutineExceptionHandler { _, exception ->
        println("Caught exception: ${exception.message}")
    }

    val job = GlobalScope.launch(handler) {
```

```
        throw Exception("Error in GlobalScope coroutine")
    }
    job.join()
}
```

在这个示例中，CoroutineExceptionHandler 用于捕获 GlobalScope 协程中的未捕获异常，并进行处理。

协程的高级用法

通道和流

通道（Channel） 和 流（Flow） 是 Kotlin 协程中用于处理数据流的两种重要概念。

- 通道（Channel）：通道是一种基于挂起函数的生产者-消费者模式的实现。它允许在不同协程之间安全地传输数据，并支持缓冲区和无缓冲区的模式。通道提供了 send 和 receive 操作，可以在协程间进行数据传递，支持多个发送者和多个接收者的模式。

```
fun main() = runBlocking {
    val channel = Channel<Int>()

    launch {
        for (x in 1..5) channel.send(x * x)
        channel.close()
    }

    repeat(5) {
        println(channel.receive())
    }

    println("Done!")
}
```

- 流（Flow）：流是一种冷的数据流，类似于响应式编程中的序列。流的操作符可以进行连续的操作，如 map、filter、transform 等，流的数据传递是惰性的，只有在收集时才会触发。流在处理连续数据流时非常有用，特别是需要处理异步操作的情况。

```
import kotlinx.coroutines.*
import kotlinx.coroutines.flow.*

fun simple(): Flow<Int> = flow {
    for (i in 1..3) {
        delay(100) // 模拟异步操作
        emit(i)
    }
}
fun main() = runBlocking {
    simple()
        .map { it * it }
        .collect { println(it) }
}
```

关于Flow，限于篇幅和主题这里只做提及，日后还会有一篇文章专门介绍它(画饼ing...)

协程与回调

好吧我不知道为什么还会有人用回调，但是Kotlin协程可以与回调结合使用，通过 suspendCoroutine 和 suspendCancellableCoroutine 等函数将回调风格的异步操作转换为挂起函数。

```
fun fetchData(callback: (Result<String>) -> Unit) {
    callback(Result.success("Data fetched successfully"))
    // callback(Result.failure(Exception("Failed to fetch data")))
}

suspend fun fetchDataAsync(): String = suspendCoroutine { cont ->
    fetchData { result ->
        result.fold(
            onSuccess = { data -> cont.resume(data) },
            onFailure = { exception -> cont.resumeWithException(exception) }
        )
    }
}

fun main() = runBlocking {
```

```
    val data = fetchDataAsync()
    println(data)
}
```

其实在这里，Kotlin中的协程基本用法已经介绍的差不多了，但是还不够。接下来我们会进一步分析协程，最终实现一个简陋的协程。

进一步探究： 如何实现协程?