Kotlin Coroutines(Part — 5): Exception Handling

How to handle exceptions thrown by coroutines?

Aditi Katiyar · Follow
Published in DeHaat · 3 min read · Sep 15, 2021

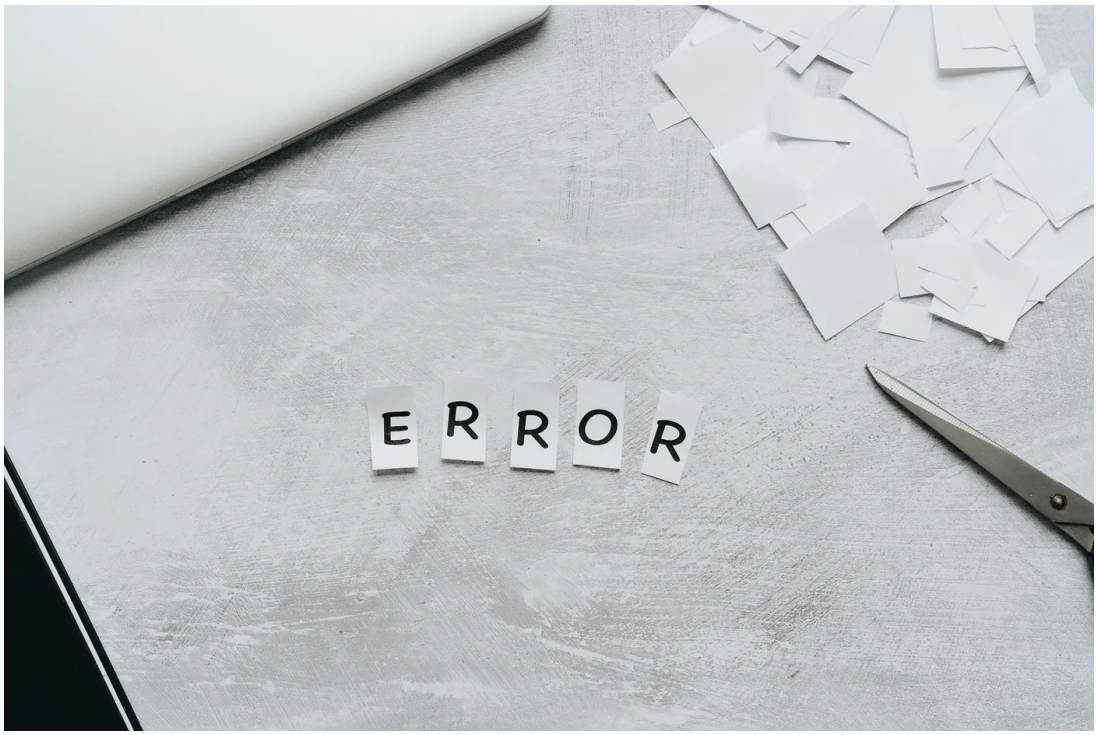

Photo by alleksana from Pexels

One needs to handle exceptions in a coroutine to facilitate a smooth user experience. We can handle exceptions in 2 ways — by using a try-catch block or *CoroutineExceptionHandler*. Which one to use depends on what we want to do when an exception occurs.

Let's look at an example job hierarchy in which *Job1* throws an exception.

```kotlin
fun main() {
    val scope = CoroutineScope(Dispatchers.IO)
    scope.launch {
        // Job1
        someExceptionThrowingWork()
    }

    scope.launch {
        // Job2
        delay(500)
        println("Job2 finished")
    }.invokeOnCompletion { throwable ->
        if (throwable is CancellationException)
            println("Job2 is cancelled")
    }

    Thread.sleep(1000) // wait for the coroutines to finish(just for demonstration purpo
}

private suspend fun someExceptionThrowingWork() {
    delay(200)
    throw RuntimeException()
}
```

trycatchincoroutines.kt hosted with ❤️ by GitHub view raw

Both *Job1* & *Job2* are running in the same scope. Note that *Job1* takes 200ms and *Job2* takes 500ms to finish(Job2 takes more time than Job1). We are simulating this with the help of *delay()* function.

If we run the above code, we get:

```
"C:\Program Files\Java\jdk1.8.0_261\bin\java.exe" ...
Job2 is cancelled
Exception in thread "DefaultDispatcher-worker-2" java.lang.RuntimeException Create breakpo
    at com.example.coroutinesexamples.ExceptionHandling_7Kt.someExceptionThrowingWork(
    at com.example.coroutinesexamples.ExceptionHandling_7Kt$someExceptionThrowingWork$
    at kotlin.coroutines.jvm.internal.BaseContinuationImpl.resumeWith(ContinuationImpl
    at kotlinx.coroutines.DispatchedTask.run(DispatchedTask.kt:106)
    at kotlinx.coroutines.scheduling.CoroutineScheduler.runSafely(CoroutineScheduler.k
    at kotlinx.coroutines.scheduling.CoroutineScheduler$Worker.executeTask(CoroutineSc
    at kotlinx.coroutines.scheduling.CoroutineScheduler$Worker.runWorker(CoroutineSche
    at kotlinx.coroutines.scheduling.CoroutineScheduler$Worker.run(CoroutineScheduler.

Process finished with exit code 0
```

Job2 is cancelled because it was running when *Job1* threw an exception. The moment *Job1* threw the exception, it forced the entire job hierarchy to shut down and the code crashed.

try-catch block

It is used when we want to recover from exceptions. Using this will not propagate the exception upwards in the job hierarchy and the remaining portion of the job tree can continue execution.

```kotlin
fun main() {
    val scope = CoroutineScope(Dispatchers.IO)
    scope.launch {
        // Job1
        try {
            someExceptionThrowingWork()
        } catch (e: Exception) {
            println("Job1 threw $e")
        }
    }

    scope.launch {
        // Job2
        delay(500)
        println("Job2 finished")
    }.invokeOnCompletion { throwable ->
        if (throwable is CancellationException)
            println("Job2 is cancelled")
    }

    Thread.sleep(1000) // wait for the coroutines to finish(just for demonstration purpo
}

private suspend fun someExceptionThrowingWork() {
    delay(200)
    throw RuntimeException()
}
```

trycatchincoroutines2.kt hosted with ❤️ by GitHub view raw

Here, we put try-catch around *someExceptionThrowingWork()* function in *Job1*. If we run the above code, we'll get:

```
"C:\Program Files\Java\jdk1.8.0_261\bin\java.exe" ...
Job1 threw java.lang.RuntimeException
Job2 finished

Process finished with exit code 0
```

The code did not crash, and *Job2* was able to finish even though *Job1* threw an exception. The try-catch helped to recover from the exception and the remaining job was able to finish.

CoroutineExceptionHandler

It is one of the context elements which is optional to pass while creating a coroutine. It helps us handle uncaught exceptions. We use this when we do not want to recover from the exception. If any coroutine fails, the entire job hierarchy shuts down and the exception is handled by our *CoroutineExceptionHandler*.

```
1   fun main() {
7           // Job1
8           someExceptionThrowingWork()
9       }
10
11      scope.launch {
12          // Job2
13          delay(500)
14          println("Job2 finished")
15      }.invokeOnCompletion { throwable ->
16          if (throwable is CancellationException)
17              println("Job2 is cancelled")
18      }
19
20      Thread.sleep(1000) // wait for the coroutines to finish(just for demonstration purpo
21  }
22
23  private suspend fun someExceptionThrowingWork() {
24      delay(200)
25      throw RuntimeException()
26  }
```

corotuineexceptionhandler.kt hosted with ❤️ by GitHub view raw

Here, we are installing a *CoroutineExceptionHandler* at the parent level scope. If we run the above code, we'll get:

```
"C:\Program Files\Java\jdk1.8.0_261\bin\java.exe" ...
Job2 is cancelled
caught exception java.lang.RuntimeException

Process finished with exit code 0
```

Again, the code did not crash. But, *Job2* is cancelled(unlike try-catch where *Job2* is finished). This is because when *Job1* threw an exception, it propagated up the job-tree, the *CoroutineExceptionHandler* at the parent's scope cancelled all the running jobs and shut down the entire job hierarchy.

CoroutineExceptionHandler should be used as a last resort for global 'catch-all' behavior.

If you want to learn more about coroutines, check these out:

- Kotlin Coroutines(Part — 1): The Basics
- Kotlin Coroutines(Part — 2): The 'Suspend' Function