

Coroutines in Kotlin

کروتین از واژه‌ی cooperative routines تشکیل شده است. یعنی روتین‌هایی که با یکدیگر همکاری می‌کنند. این بدین معناست که هنگامی که یک روتین در حال اجراست سایر روتین‌ها با suspend کردن خودشان ارتباطشان با cpu ، memory و هر ریسورس دیگری را قطع می‌کنند تا از بلاک کردن آن‌ها جلوگیری شود.

کروتین‌ها یک مکانیسم lightweight و موثر در concurrency هستند که می‌توانند در Kotlin برای نوشتن کدهای ناهمزمان و غیر مسدود کننده استفاده شوند. آن‌ها در مقایسه با مدل‌های threading سنتی مزایای زیادی دارند، از جمله:

- **Simplicity:** فهم کروتین در مقایسه با threading API ها آسان تر است و برای توسعه دهندگان با هر سطحی از دانش قابل دسترس تر است.
- **Efficiency:** کروتین‌ها lightweight هستند و کمترین میزان overhead را دارند. بنابراین می‌توان تعداد بسیار زیادی از آن‌ها را بدون آنکه روی عملکرد سیستم تاثیر گسترده‌ای بگذارد ایجاد کرد.
- **Asynchrony:** کروتین‌ها می‌توانند برای نوشتن کدهای asynchronous و non-blocking استفاده شوند. آن‌ها می‌توانند فعالیت‌های طولانی مدت (long-running) را بدون بلاک کردن ترد اصلی سیستم انجام دهند.
- **Structured concurrency:** کروتین به صورت built-in از قواعد Structured concurrency پشتیبانی می‌کند. این پترن مطمئن می‌شود آغاز، اجرا، پایان یا کنسل شدن کروتین‌ها در یک زمان قابل پیشبینی و در یک راه امن صورت پذیرد. این امر باعث می‌شود تا کروتین‌ها دچار ریسورس leak نشوند و از شر ارورهای رایج در اعمال concurrency در امان بمانند.
- **Exception handling:** کروتین به صورت built-in از قابلیت Exception handling پشتیبانی می‌کند. این بدین معناست که به ارور ها اجازه می‌دهد تا در سلسله مراتب به سمت بالا حرکت کنند تا به handler مناسب خودشان برسند. این کار نوشتن کدی را آسان تر می‌کند که در برابر خطاها مقاوم باشد.

Structured concurrency

یک دیزاین پترن است که به ساده سازی و افزایش قابلیت اطمینان در اجرای فرایندهای concurrent کمک می‌کند. این دیزاین پترن اطمینان حاصل می‌کند تا تمام فرایندهای concurrent به صورت ساختار یافته و مدیریت شده اجرا شوند تا با یکدیگر به تداخل بر نخورند و side effect ای بر جای نگذارند.

این هدف به کمک محدود کردن اجرای فرایندها در یک سلسله مراتب خاص (هر فرایند درون یک parent context انجام می‌شود و تضمین می‌شود تا قبل از آن که parent اش خاتمه یابد فعالیتش را تمام کند) محقق می‌شود.

Structured concurrency در کاتلین بر اساس یک رابطه‌ی سلسله مراتبی بین کروتین‌هاست، هر کروتین یک parent دارد. موقعی که یک کروتین جدید start زده می‌شود این کروتین به child یک کروتین دیگر تبدیل می‌شود.

این رابطه‌ی سلسله مراتبی باقی است تا زمانی که طول عمر تمام کروتین‌ها به پایان برسد. لازم به ذکر است هنگامی که یک کروتین parent کنسل می‌شود تمام child های آن نیز به صورت سلسله مراتبی از پایین ترین قسمت کنسل می‌شوند این کمک می‌کند تا مطمئن شویم هیچ leak و یا کروتین معلق وجود ندارد.

یکی از مزایای مهم Structured concurrency آن است که به کمک محدود کردن اجرای کروتین‌ها به یک context سلسله مراتبی خاص مطمئن می‌شود تا همه‌ی فرایندها مدیریت شده و ریسورس‌ها در یک زمان قابل پیش‌بینی آزاد شوند. به کمک این مورد می‌توان بر بسیاری از مشکلات فرایندهای concurrent مانند race condition و deadlock و resource leak غلبه کرد.

به مثال زیر توجه کنید:

```
import kotlinx.coroutines.*

fun main() = runBlocking<Unit> {
    val job = launch {
        delay(1000)
        println("World!")
    }
    println("Hello, ")
    //job.join() // calling or not calling this line have same o/p
}
// Op => Hello World!
```

جواب در یک کروتین دیگر به نام `runBlocking` قرار گرفته و خود `runBlocking` به فانکشن `main` محدود شده است. در این مثال ابتدا باید `job` به اتمام برسد تا `runBlocking` بسته شود و خود `runBlocking` باید کامل شود تا تابع `main` بتواند `return` کند و بسته شود.

این باعث می‌شود تا مطمئن شویم هنگامی که تابع `main` بسته می‌شود `job` و `runBlocking` به اتمام رسیده است.

Coroutine Cancellation

`Cancellation` فرایند توقف اجرای یک کروتین است قبل از آن که آن کروتین وظیفه‌ی خود را به پایان برساند. موقعی که یک کروتین کنسل می‌شود اجرائیش را از نزدیک‌ترین `cancellation point` به پایان می‌رساند. یک `cancellation point` جایی است که کروتین در آن نقطه به دنبال `signal` توقف می‌گردد تا در صورتی که آن را پیدا کرد به اجرا خاتمه دهد (به طور مثال فانکشن `delay` می‌تواند نقش یک `cancellation point` را بازی کند)

یک کروتین به دو روش می‌تواند کنسل شود:

- **Explicitly:** به کمک فانکشن `cancel()` می‌توان یک کروتین را به صورت `explicit` متوقف کرد. فانکشن `cancel()` یک عضو از اینترفیس `Job` است و می‌تواند کروتین را بدون در نظر گرفتن `state` فعلی آن و به صورت فوری متوقف کند.

```
val job = launch {  
    // Do some work here  
}  
// Cancel the coroutine explicitly  
job.cancel()
```

Automatically: یک کروتین هنگامی که `parent` اش متوقف می‌شود می‌تواند به صورت اتوماتیک متوقف شود.

```
val parentJob = Job()  
  
// Launch a child coroutine in the parentJob  
val childJob = launch(parentJob) {  
    // Do some work here  
}  
  
// Cancel the parent coroutine  
parentJob.cancel()
```

موقعی که یک کروتین کنسل می‌شود، استیت job آن به cancelling تغییر می‌کند و فرزندانش به صورت recursive از پایین ترین قسمت سلسله مراتب شروع به کنسل شدن می‌کنند. در طی فرایند cancelling هر کروتین فرصت این را دارد تا ریسورس های خودش را clean up کند.

توجه شود که وقتی یک کروتین کنسل می‌شود الزاما به این معنا نیست که بلاک آن می‌تواند به طور کامل انجام شود. تنها زمانی می‌توان از اجرای کامل بلاک مطمئن شد که کروتین به صورت normal complete شده باشد. (پس چگونه می‌توان فرصت آزاد سازی منابع را به دست آورد؟)

How to handle the Coroutine Cancellation?

هنگامی که یک کروتین کنسل می‌شود، این نکته حائز اهمیت است که cancellation را به نحوی مدیریت کنیم تا از هر گونه resource leak و خطاهای دیگر جلوگیری شود. چندین راه برای این امر وجود دارد.

- try-catch-finally: این بلاک می‌تواند برای مدیریت cancellation استفاده شود. بلاک finally می‌تواند برای آزاد سازی ریسورس ها استفاده شود زیرا حتی اگر کروتین سیگنال کنسل را دریافت کرده باشد finally همچنان اجرا می‌شود.

```
val job = launch {  
    try {  
        // Do some work here  
    } catch (e: Exception) {  
        // Handle the exception  
    } finally {  
        // Release any resources here  
    }  
}
```

- withContext(NonCancellable): این فانکشن می‌تواند برای اجرای یک بلاک از کدهایی استفاده شود که در coroutine context فعلی قابل کنسل نیستند.

- `kotlinx.coroutines.delay()`: این فانکشن یک فانکشن `cancellable` است و هنگامی که سیگنال `cancel` را دریافت می‌کند `CancellationException` ای را `throws` می‌کند. ما با دریافت این `exception` می‌توانیم فرصت آزاد سازی `resource` ها را به دست آوریم.

```
val job = launch {  
    try {  
        // Do some work here  
        delay(5000)  
    } catch (e: CancellationException) {  
        // Handle the cancellation exception  
    }  
}
```

How to ensure that finally block is also executed even when coroutine is cancelled?

در کاتلین کروتین، یک بلاک `finally` برای اجرای کدهایی به کار می‌رود که باید صرف نظر از آن که `exception` ای اتفاق افتاده است یا خیر اجرا شوند. در مواردی که یک کروتین سیگنال کنسل را دریافت کرده، مهم است تا مطمئن شویم که بلاک `finally` به طور کامل اجرا می‌شود یا خیر

هنگامی که یک کروتین کنسل می‌شود، اجرای آن متوقف می‌شود و یک `cancellation exception` به سمت والد آن پرتاب می‌شود، این `exception` در سلسله مراتب آن قدر بالا می‌رود تا به `root` برسد.

هنگامی که یک کروتین کنسل می‌شود، بلاک `finally` قبل از آن که کروتین به صورت حقیقی متوقف شود (رسیدن `exception` به `root` در سلسله مراتب) اجرا می‌شود. این بلاک برای اعمالی مانند آزاد سازی منابع، بستن ارتباط شبکه یا دیتابیس و ... کاربرد دارد. اما گاهی از اوقات یک یا برخی از این اعمال ممکن است خودشان `cancellable` و زمانبر باشند در چنین شرایطی وقوع `exception` در عملکرد آن‌ها می‌تواند سیستم را در شرایط ناپایداری قرار دهد.

برای آن که مطمئن شویم بلاک `finally` هنگام کنسل شدن کروتین به طور کامل اجرا می‌شود سه راه حل وجود دارد:

- کال کردن `yield()` در بلاک `finally`: هنگامی که تابع `yield()` کال می‌شود کروتین را `suspend` می‌کند و منتظر می‌شود تا مجدد `resume` شود. استفاده از `yield` منجر می‌شود تا مطمئن شویم بلاک `finally` در هنگام کنسل شدن کروتین حتما اجرا می‌شود. اما چگونه؟ هنگامی که تابع `yield()` کال می‌شود کروتین را `suspend` می‌کند و منتظر می‌شود تا مجدد `resume` شود. هنگامی که یک کروتین کنسل می‌شود نمی‌تواند مجدداً `resume` شود، اما کال کردن `yield` در `finally` منجر می‌شود تا کروتین مجدداً در صف `ready coroutine` ها قرار بگیرد به این ترتیب `finally` فرصت پیدا میکند تا قبل از پایان حقیقی کروتین کارش را به پایان برساند.

- استفاده از `withContext(NonCancellable)` در بلاک `finally`: این فانکشن مطمئن می‌شود که کد درون بلاک آن حتی در صورت کنسل شدن کروتین، اجرا می‌شود. اما چگونه؟ این تابع یک `context` جدید می‌سازد که قابل کنسل شدن نیست. استفاده از آن در شرایطی مناسب است که می‌خواهیم بدون در نظر گرفتن این که کروتین کنسل شده است یا خیر بلاکی از کدها را اجرا کنیم.

```
fun main() = runBlocking {
    val job = launch {
        try {
            println("Coroutine started")
            delay(500)
            throw RuntimeException("Oops!")
            println("RuntimeException is thrown")
        } finally {
            withContext(NonCancellable) {
                delay(100)
                println("Finally block executed inside withcontext")
            }
            delay(10)
            println("Finally block executed outside")
        }
    }
    delay(100)
    job.cancelAndJoin()
    println("Coroutine cancelled")
}
```

توجه داشته باشید که قبل از اجرای کد `clean up` (کد های داخل بلاک `withContext`)، `cancelAndJoin` را در کوروتین اصلی فراخوانی می کنیم. این مهم است زیرا فراخوانی `withContext (NonCancellable)` در حالی که کوروتین اصلی هنوز در حال اجرا است می تواند باعث مسدود شدن نامحدود کوروتین جدید شود. با لغو کوروتین اصلی قبل از اجرای کد `clean up`، اطمینان حاصل می کنیم که کوروتین جدید مسدود نشده است و می تواند کار `clean up` را کامل کند.

- استفاده از فانکشن `delay()` در بلاک `try` و دریافت `CancellationException`: به مثال زیر توجه کنید:

```
fun main() = runBlocking {
    val job = launch {
        try {
            println("Coroutine started")
            repeat(10) { i ->
                delay(100)
                println("Iteration $i")
            }
        } catch (e: CancellationException) {
            println("Coroutine cancelled")
        } finally {
            println("Finally block executed")
        }
    }

    delay(500)
    job.cancel()
    println("Cancellation requested")
    job.join()
}
```

در مثال بالا هنگامی که کروتین کنسل می‌شود فانکشن delay یک exception از نوع CancellationException را پرتاب می‌کند. دریافت این exception می‌تواند فرصت مناسبی را در اختیار بلاک finally قرار دهد تا بتواند تسک‌های clean up را بدون در نظر گرفتن اینکه کروتین به صورت normal به اتمام رسیده یا کنسل شده است انجام دهد.