Pengantar Coroutine di Kotlin Playground

Konkurensi

Konkurensi merupakan proses menjalankan beberapa tugas di aplikasi secara bersamaan. Misalnya, aplikasi Anda bisa mendapatkan data dari server web atau menyimpan data pengguna di perangkat, selagi merespons peristiwa input pengguna dan mengupdate UI yang sesuai.

Kode sinkron

Dalam kode **sinkron**, hanya satu tugas konseptual yang sedang berlangsung dalam satu waktu. Satu tugas harus selesai sepenuhnya sebelum tugas berikutnya dimulai. Berikut adalah contoh kode sinkron.

```
fun main() {
    println("Weather forecast")
    println("Sunny")
}
```

Dalam fungsi `main()`,terlebih dulu akan mencetak teks: `Weather forecast`. Lalu, akan mencetak: `Sunny`. Karena setiap panggilan fungsi di `main()` bersifat sinkron, seluruh fungsi `main()` akan sinkron.

Menambahkan penundaan

```
import kotlinx.coroutines.*

fun main() {
    println("Weather forecast")
    delay(1000)   // tunda 1 detik
    println("Sunny")
}
```

Jika Anda mencoba menjalankan program pada tahap ini, akan terjadi error kompilasi: `Suspend function 'delay' should be called only from a coroutine or another suspend function`.

Agar tidak terjadi error tambahkan fungis runBlocking.

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

```
fun main() {
    runBlocking {
        println("Weather forecast")
        delay(1000)
        println("Sunny")
    }
}
```

`runBlocking()` menjalankan loop peristiwa, yang dapat menangani beberapa tugas sekaligus dengan melanjutkan setiap tugas dari posisi terakhir saat tugas siap dilanjutkan.

`runBlocking()` berjalan secara sinkron; kode ini tidak akan ditampilkan hingga semua tugas dalam blok lambda-nya selesai. Artinya, tugas tersebut akan menunggu tugas dalam panggilan `delay()` selesai (hingga satu detik berlalu), lalu melanjutkan dengan mengeksekusi pernyataan cetak `Sunny`. Setelah semua tugas di fungsi `runBlocking()` selesai, fungsi tersebut akan ditampilkan, yang mengakhiri program.

Hasil program:

```
Weather forecast
Sunny
```

Fungsi penangguhan

Jika logika yang sebenarnya dalam melakukan permintaan jaringan untuk mendapatkan data cuaca menjadi lebih kompleks, Anda mungkin ingin mengekstrak logika tersebut ke dalam fungsinya sendiri. Mari kita faktorkan ulang kode untuk melihat efeknya.

```
import kotlinx.coroutines.*

fun main() {
    runBlocking {
        println("Weather forecast")
        printForecast()
    }
}

fun printForecast() {
    delay(1000)
    println("Sunny")
}
```

Fungsi penangguhan hanya dapat dipanggil dari coroutine atau fungsi penangguhan lainnya, jadi tentukan `printForecast()` sebagai fungsi `suspend`.

```
import kotlinx.coroutines.*

fun main() {
    runBlocking {
        println("Weather forecast")
        printForecast()
```

```
    }
}

suspend fun printForecast() {
    delay(1000)
    println("Sunny")
}
```

Ingat bahwa `delay()` adalah fungsi penangguhan, dan sekarang Anda juga telah membuat `printForecast()` sebagai fungsi penangguhan. Fungsi **penangguhan** mirip dengan fungsi biasa, tetapi dapat ditangguhkan dan dilanjutkan lagi nanti.

```
import kotlinx.coroutines.*

fun main() {
    runBlocking {
        println("Weather forecast")
        printForecast()
        printTemperature()
    }
}

suspend fun printForecast() {
    delay(1000)
    println("Sunny")
}

suspend fun printTemperature() {
    delay(1000)
    println("30\u00b0C")
}
```

Hasil program:

```
Weather forecast
Sunny
30°C
```

Dalam kode ini, coroutine terlebih dahulu ditangguhkan dengan penundaan dalam fungsi penangguhan `printForecast()`, lalu dilanjutkan setelah penundaan satu detik tersebut. Teks `Sunny` dicetak ke output. Fungsi `printForecast()` kembali ke pemanggil.

Selanjutnya, fungsi `printTemperature()` akan dipanggil. Coroutine tersebut ditangguhkan saat mencapai panggilan `delay()`, lalu dilanjutkan satu detik kemudian dan selesai mencetak nilai suhu ke output. Fungsi `printTemperature()` telah menyelesaikan semua tugas dan pengembalian.

Jika ingin melihat waktu yang diperlukan untuk menjalankan program ini dengan penundaan, Anda dapat menggabungkan kode dalam panggilan ke `measureTimeMillis()`.

```
import kotlin.system.*
import kotlinx.coroutines.*

fun main() {
    val time = measureTimeMillis {
        runBlocking {
            println("Weather forecast")
            printForecast()
            printTemperature()
        }
    }
    println("Execution time: ${time / 1000.0} seconds")
}
suspend fun printForecast() {
    delay(1000)
    println("Sunny")
}

suspend fun printTemperature() {
    delay(1000)
    println("30\u00b0C")
}
```

Hasil program:

```
Weather forecast
Sunny
30°C
Execution time: 2.098 seconds
```

Output menunjukkan bahwa perlu waktu ~ 2,1 detik untuk dieksekusi. Hal tersebut tampaknya wajar karena setiap fungsi penangguhan memiliki penundaan satu detik.

Kode asinkron

launch()

Gunakan fungsi `launch()` dari library coroutine untuk meluncurkan coroutine baru. Untuk menjalankan tugas secara serentak, tambahkan beberapa fungsi `launch()` ke kode Anda agar beberapa coroutine dapat diproses secara bersamaan.

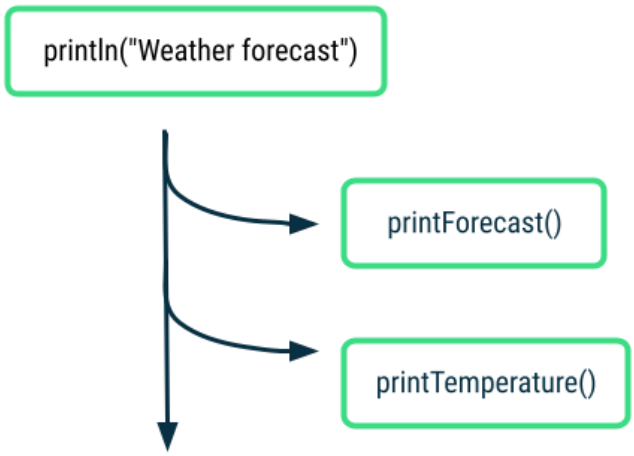
```
import kotlin.system.*
import kotlinx.coroutines.*

fun main() {
    val time = measureTimeMillis {
        runBlocking {
            println("Weather forecast")
            launch {
```

```
                printForecast()
            }
            launch {
                printTemperature()
            }
        }
    }
    println("Execution time: ${time / 1000.0} seconds")
}

suspend fun printForecast() {
    delay(1000)
    println("Sunny")
}

suspend fun printTemperature() {
    delay(1000)
    println("30\u00b0C")
}
```

Hasil program:

```
Weather forecast
Sunny
30°C
Execution time: 1.122 seconds
```

Sebelumnya, Anda harus menunggu fungsi penangguhan `printForecast()` untuk selesai sepenuhnya sebelum beralih ke fungsi `printTemperature()`. Sekarang `printForecast()` dan `printTemperature()` dapat berjalan secara serentak karena keduanya berada di coroutine terpisah.



Ubah kode `runBlocking()` untuk menambahkan pernyataan cetak tambahan sebelum akhir blok tersebut.

```
...
```

```
fun main() {
    runBlocking {
        println("Weather forecast")
        launch {
            printForecast()
        }
        launch {
            printTemperature()
        }
        println("Have a good day!")
    }
}

...
```

Hasil program:

```
Weather forecast
Have a good day!
Sunny
30°C
```

async()

Di dunia nyata, Anda tidak akan tahu berapa lama permintaan jaringan yang diperlukan oleh perkiraan cuaca dan suhu. Jika Anda ingin menampilkan laporan cuaca terpadu saat kedua tugas selesai, pendekatan saat ini dengan `launch()` saja tidak cukup. Di situlah `async()` berperan.

Gunakan fungsi `async()` dari library coroutine jika Anda ingin mengetahui kapan coroutine selesai dan memerlukan nilai yang ditampilkan dari coroutine tersebut.

Fungsi `async()` menghasilkan objek bertipe `Deferred`, yang seperti jaminan bahwa hasilnya akan ada di sana jika sudah siap. Anda dapat mengakses hasilnya pada objek `Deferred` menggunakan `await()`.

```
import kotlinx.coroutines.*

fun main() {
    runBlocking {
        println("Weather forecast")
        val forecast: Deferred<String> = async {
            getForecast()
        }
        val temperature: Deferred<String> = async {
            getTemperature()
        }
        println("${forecast.await()} ${temperature.await()}")
        println("Have a good day!")
    }
```

```
}

suspend fun getForecast(): String {
    delay(1000)
    return "Sunny"
}

suspend fun getTemperature(): String {
    delay(1000)
    return "30\u00b0C"
}
```

Hasil program:

```
Weather forecast
Sunny 30°C
Have a good day!
```

Dekomposisi Paralel

Kita dapat mengembangkan contoh cuaca ini secara lebih detail dan melihat kegunaan coroutine dalam dekomposisi tugas secara paralel. Dekomposisi paralel melibatkan pengambilan masalah dan memecahnya menjadi subtugas yang lebih kecil yang dapat diselesaikan secara paralel. Setelah hasil subtugas siap, Anda dapat menggabungkannya menjadi hasil akhir.

```
import kotlinx.coroutines.*

fun main() {
    runBlocking {
        println("Weather forecast")
        println(getWeatherReport())
        println("Have a good day!")
    }
}

suspend fun getWeatherReport() = coroutineScope {
    val forecast = async { getForecast() }
    val temperature = async { getTemperature() }
    "${forecast.await()} ${temperature.await()}"
}

suspend fun getForecast(): String {
    delay(1000)
    return "Sunny"
}

suspend fun getTemperature(): String {
    delay(1000)
    return "30\u00b0C"
}
```

Hasil program:

```
Weather forecast
Sunny 30°C
Have a good day!
```

`coroutineScope()` hanya akan ditampilkan setelah semua tugasnya, termasuk coroutine yang diluncurkan, telah selesai. Dalam hal ini, coroutine `getForecast()` dan `getTemperature()` perlu menyelesaikan dan menampilkan hasil masing-masing. Kemudian, teks `Sunny` dan `30°C` digabungkan dan ditampilkan dari cakupan. Laporan cuaca `Sunny 30°C` ini akan dicetak ke output, dan pemanggil dapat melanjutkan ke pernyataan cetak terakhir `Have a good day!`.

Dengan `coroutineScope()`, meskipun fungsi tersebut melakukan tugas internal secara serentak, fungsi ini akan muncul bagi pemanggil sebagai operasi sinkron karena `coroutineScope` tidak akan ditampilkan hingga semua tugas selesai.

Pengecualian dan pembatalan

Pengecualian (eksepsi) adalah peristiwa tidak terduga yang terjadi selama eksekusi kode Anda. Anda harus menerapkan cara yang tepat untuk menangani pengecualian ini, guna mencegah aplikasi Anda error dan memengaruhi pengalaman pengguna secara negatif.

```
fun main() {
    val numberOfPeople = 0
    val numberOfPizzas = 20
    println("Slices per person: ${numberOfPizzas / numberOfPeople}")
}
```

Saat program dijalankan:

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
 at FileKt.main (File.kt:4)
 at FileKt.main (File.kt:-1)
 at jdk.internal.reflect.NativeMethodAccessorImpl.invoke0 (:-2)
```

Pengecualian dengan coroutine

```
import kotlinx.coroutines.*

fun main() {
    runBlocking {
        println("Weather forecast")
        println(getWeatherReport())
        println("Have a good day!")
    }
}
```

```
suspend fun getWeatherReport() = coroutineScope {
    val forecast = async { getForecast() }
    val temperature = async { getTemperature() }
    "${forecast.await()} ${temperature.await()}"
}

suspend fun getForecast(): String {
    delay(1000)
    return "Sunny"
}

suspend fun getTemperature(): String {
    delay(1000)
    return "30\u00b0C"
}
```

Dalam salah satu fungsi penangguhan, tampilkan pengecualian secara sengaja untuk melihat efeknya. Tindakan ini menyimulasikan terjadinya error tidak terduga saat mengambil data dari server, yang mungkin saja terjadi.

```
...

suspend fun getTemperature(): String {
    delay(500)
    throw AssertionError("Temperature is invalid")
    return "30\u00b0C"
}
```

Hasil program:

```
Weather forecast
Exception in thread "main" java.lang.AssertionError: Temperature is
invalid
 at FileKt.getTemperature (File.kt:24)
 at FileKt$getTemperature$1.invokeSuspend (File.kt:-1)
 at kotlin.coroutines.jvm.internal.BaseContinuationImpl.resumeWith
(ContinuationImpl.kt:33)
```

Coroutine yang menjalankan `getTemperature()` dan coroutine yang menjalankan `getForecast()` adalah coroutine turunan dari coroutine induk yang sama. Jika salah satu coroutine turunan gagal dengan pengecualian, coroutine tersebut akan disebarkan ke atas. Coroutine induk dibatalkan sehingga membatalkan coroutine turunan lainnya (misalnya, coroutine yang menjalankan `getForecast()` dalam kasus ini). Terakhir, error menyebar ke atas dan program mengalami error dengan `AssertionError`.

Eksepsi try-catch

Jika mengetahui bahwa bagian tertentu dari kode Anda mungkin dapat memunculkan pengecualian, Anda dapat mengapit kode tersebut dengan blok `try-catch`. Anda dapat menangkap pengecualian dan menanganinya dengan lebih baik di aplikasi.

```
try {
    // Some code that may throw an exception
} catch (e: IllegalArgumentException) {
    // Handle exception
}
```

Ubah program cuaca untuk menangkap pengecualian yang Anda tambahkan sebelumnya, lalu cetak pengecualian ke output.

```
import kotlinx.coroutines.*

fun main() {
    runBlocking {
        println("Weather forecast")
        try {
            println(getWeatherReport())
        } catch (e: AssertionError) {
            println("Caught exception in runBlocking(): $e")
            println("Report unavailable at this time")
        }
        println("Have a good day!")
    }
}

suspend fun getWeatherReport() = coroutineScope {
    val forecast = async { getForecast() }
    val temperature = async { getTemperature() }
    "${forecast.await()} ${temperature.await()}"
}

suspend fun getForecast(): String {
    delay(1000)
    return "Sunny"
}

suspend fun getTemperature(): String {
    delay(500)
    throw AssertionError("Temperature is invalid")
    return "30\u00b0C"
}
```

Hasil program:

```
Weather forecast
Caught exception in runBlocking(): java.lang.AssertionError: Temperature
is invalid
Report unavailable at this time
Have a good day!
```

Pindahkan penanganan error agar perilaku try-catch benar-benar terjadi dalam coroutine yang diluncurkan oleh `async()` untuk mengambil suhu. Dengan demikian, laporan cuaca tetap dapat mencetak perkiraan, meskipun suhu gagal.

```
import kotlinx.coroutines.*

fun main() {
    runBlocking {
        println("Weather forecast")
        println(getWeatherReport())
        println("Have a good day!")
    }
}

suspend fun getWeatherReport() = coroutineScope {
    val forecast = async { getForecast() }
    val temperature = async {
        try {
            getTemperature()
        } catch (e: AssertionError) {
            println("Caught exception $e")
            "{ No temperature found }"
        }
    }

    "${forecast.await()} ${temperature.await()}"
}

suspend fun getForecast(): String {
    delay(1000)
    return "Sunny"
}

suspend fun getTemperature(): String {
    delay(500)
    throw AssertionError("Temperature is invalid")
    return "30\u00b0C"
}
```

Hasil program:

```
Weather forecast
Caught exception java.lang.AssertionError: Temperature is invalid
Sunny { No temperature found }
```

```
Have a good day!
```

Pembatalan

Skenario ini biasanya berdasarkan pengguna saat sebuah peristiwa menyebabkan aplikasi membatalkan tugas yang telah dimulai sebelumnya.

```
import kotlinx.coroutines.*

fun main() {
    runBlocking {
        println("Weather forecast")
        println(getWeatherReport())
        println("Have a good day!")
    }
}

suspend fun getWeatherReport() = coroutineScope {
    val forecast = async { getForecast() }
    val temperature = async { getTemperature() }

    delay(200)
    temperature.cancel()

    "${forecast.await()}"
}

suspend fun getForecast(): String {
    delay(1000)
    return "Sunny"
}

suspend fun getTemperature(): String {
    delay(1000)
    return "30\u00b0C"
}
```

Hasil program:

```
Weather forecast
Sunny
Have a good day!
```

Sumber:

https://developer.android.com/codelabs/basic-android-kotlin-compose-coroutines-kotlin-playground?hl=id&continue=https%3A%2F%2Fdeveloper.android.com%2Fcourses%2Fpathways%2Fandroid-basics-compose-unit-5-pathway-1%3Fhl%3Did%23codelab-https%3A%2F%2Fdeveloper.android.com%2Fcodelabs%2Fbasic-android-kotlin-compose-coroutines-kotlin-playground#0