Cơ bản về Kotlin Coroutines
Áp dụng trong ứng dụng Android

ONE LOVE. ONE FUTURE.

Nội dung

- Giới thiệu chung
- Ví dụ đầu tiên
- Khái niệm hàm suspend
- Khởi tạo coroutine
- Chờ và hủy coroutine
- Async và await
- Coroutine context
- Coroutine scope

Giới thiệu về coroutines

- Coroutine là cơ chế cho phép thực hiện các đoạn mã nguồn đồng thời và xử lý kết quả bất đồng bộ.
- Cách sử dụng đơn giản và hiệu quả hơn so với các kỹ thuật truyền thống như Thread, AsyncTask, Callback.
- Sử dụng ít bộ nhớ hơn
- Cho phép chuyển đổi linh hoạt giữa các ngữ cảnh (chạy nền, cập nhật giao diện)
- Được sử dụng để thực hiện kết nối CSDL (ROOM), kết nối Internet.

Ví dụ đầu tiên

- Để tạo và chạy coroutine cần thêm các thư viện vào project.
- Đối với project Kotlin

```
implementation("org.jetbrains.kotlinx:kotlinx-coroutines-core-jvm:1.5.0")
```

- Đối với project Android

```
implementation 'org.jetbrains.kotlinx:kotlinx-coroutines-core:1.2.1'

implementation 'org.jetbrains.kotlinx:kotlinx-coroutines-android:1.1.1'
```

Ví dụ đầu tiên

Tạo file **test.kt** và chạy đoạn mã sau:

```
import kotlinx.coroutines.*

fun main() = runBlocking {
    launch {
        delay(5000L)
        println("World")
    }

    println("Hello")
}
```

Kết quả thực hiện:

```
Hello
World
```

=> Từ “World” xuất hiện sau từ “Hello” 5 giây.

Ví dụ đầu tiên

Tạo file **test.kt** và chạy đoạn mã sau:

```
import kotlinx.coroutines.*

fun main() = runBlocking {
    launch {
        delay(5000L)
        println("World")
    }

    println("Hello")
}
```

Giải thích:

launch – tạo và chạy một coroutine mới đồng thời với phần còn lại của mã nguồn

delay – hàm suspend dừng coroutine trong một khoảng thời gian, việc tạm dừng coroutine không làm dừng luồng, cho phép các coroutine khác vẫn có thể hoạt động

runBlocking – được dùng để kết nối mã nguồn không dùng coroutine và mã nguồn sử dụng coroutine, tạo và chạy một coroutine, hàm trả về kết quả khi các coroutine bên trong kết thúc

Ví dụ đầu tiên

Cập nhật file **test.kt** như sau:

```
import kotlinx.coroutines.*

fun main() = runBlocking {
    launch { doWorld() }
    println("Hello")
}

suspend fun doWorld() {
    delay(5000L)
    println("World")
}
```

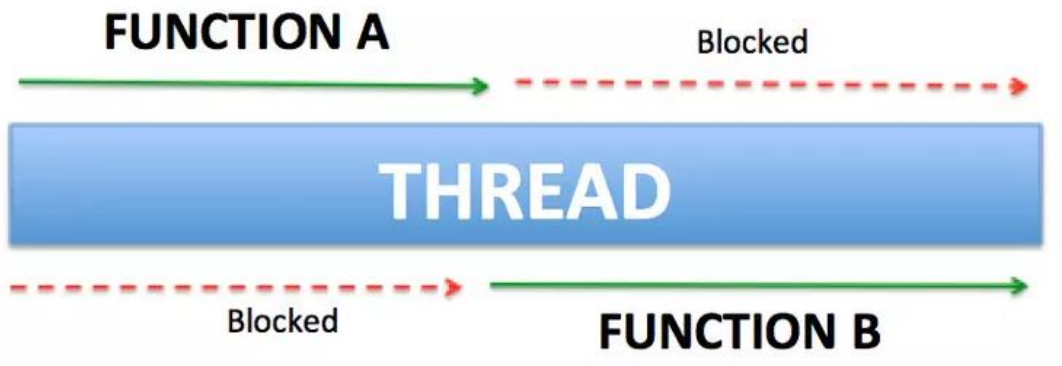
doWorld() là ví dụ về hàm suspend được gọi trong các coroutines, nội dung hàm có thể gọi các hàm suspend khác cho phép tạm dừng coroutine.

Khái niệm hàm suspend

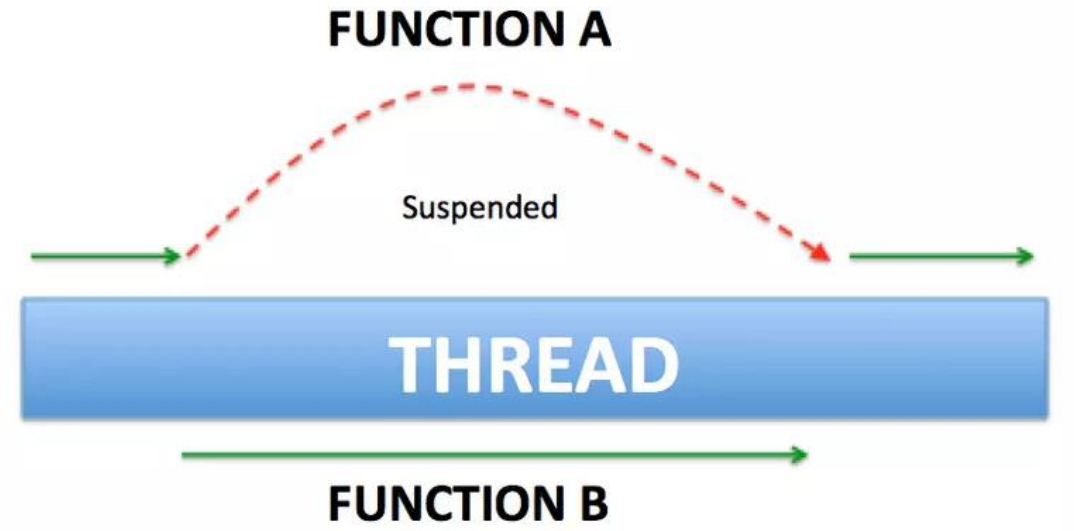
- Là hàm được sử dụng trong các coroutines hoặc trong các hàm suspend khác.
- Có khả năng tạm dừng việc thực thi và tiếp tục khi cần thiết.

Khái niệm hàm suspend

Hàm block: dừng luồng cho đến khi kết thúc hàm, các hàm khác phải chờ.



Hàm suspend: nếu chưa có kết quả thì sẽ tạm dừng nhưng không dừng luồng, cho phép các hàm khác không phải chờ.



Cách thức khởi tạo coroutine

Để tạo và chạy các coroutine, có thể sử dụng các hàm

- **launch()** – khởi tạo coroutine thực hiện công việc mà không trả về kết quả
- **runBlocking()** – khởi tạo coroutine và đợi cho đến khi hoàn tất
- **async()** – khởi tạo coroutine thực hiện công việc và có trả về kết quả

Hàm launch()

```
fun CoroutineScope.launch(
    context: CoroutineContext = EmptyCoroutineContext,
    start: CoroutineStart = CoroutineStart.DEFAULT,
    block: suspend CoroutineScope.() -> Unit
): Job
```

- Tạo và chạy coroutine mới mà không block luồng hiện tại và trả về kết quả là đối tượng tham chiếu đến coroutine thuộc lớp **Job**. Quản lý coroutine thông qua đối tượng này.

- Ngữ cảnh thực thi được kế thừa từ **CoroutineScope**, có thể được bổ sung bằng tham số **context**.

- Mặc định coroutine được xếp lịch để thực hiện ngay lập tức, hoặc theo các lựa chọn khác bằng tham số **start**.

Ví dụ

```
launch {
    delay(2000L)
    println("Third message!")
}
launch {
    delay(1000L)
    println("Second message!")
}
println("First message!")
```

Kết quả khi chạy:

- “First message!” được hiển thị đầu tiên.
- “Second message!” hiển thị sau 1 giây.
- “Third message!” hiển thị sau 2 giây.

Ví dụ

```
val job1 = lifecycleScope.launch(Dispatchers.IO) {
    // Thực hiện kết nối Internet
}

val job2 = lifecycleScope.launch(Dispatchers.Main) {
    // Cập nhật giao diện
}
```

- **job1** chạy ở luồng background, thực hiện kết nối CSDL, kết nối Internet
- **job2** chạy ở luồng giao diện, thực hiện cập nhật giao diện
- **lifecycleScope** là phạm vi gắn với activity, fragment

Hàm runBlocking()

```
expect fun <T> runBlocking(
    context: CoroutineContext = EmptyCoroutineContext,
    block: suspend CoroutineScope.() -> T
): T
```

- Tạo và chạy coroutine mới và dừng luồng cho đến khi hoàn thành.
- Hàm được thiết kế để chuyển đổi giữa kiểu mã nguồn blocking và mã nguồn suspending.
- Được sử dụng chủ yếu trong hàm **main()** với mục đích thử nghiệm.

Ví dụ

```
fun main() {
    runBlocking {
        delay(2000L)
        println("First message!")
    }
    println("Second message!")
}
```

Kết quả khi chạy:

- “First message!” hiển thị sau 2 giây.
- “Second message!” hiển thị sau cùng.

Chờ và hủy coroutine đang chạy

- Hàm **join()** – chờ coroutine hoàn thành
- Hàm **cancel()** – hủy thực hiện coroutine
- Hàm **withTimeout()** – chạy coroutine trong khoảng thời gian giới hạn

Chờ coroutine hoàn thành

- Sử dụng hàm **join()** của lớp **Job**

Ví dụ: **job1.join()** sẽ tạm dừng **job2** cho đến khi **job1** hoàn thành

```
runBlocking {
    val job1 = launch {
        repeat(10) {
            delay(1000L)
            println("Job1: ${it + 1}")
        }
    }
    val job2 = launch {
        println("Waiting for job1 complete...")
        job1.join()
        repeat(10) {
            delay(1000L)
            println("Job2: ${it + 1}")
        }
    }
}
```

Hủy coroutine đang thực hiện

- Sử dụng hàm **cancel()** của lớp **Job, CoroutineScope** hoặc **CoroutineContext**

```
fun CoroutineScope.cancel(cause: CancellationException? = null)
fun CoroutineScope.cancel(message: String, cause: Throwable? = null)
fun CoroutineContext.cancel(cause: CancellationException? = null)
fun Job.cancel(message: String, cause: Throwable? = null)
```

Hủy các coroutine trong một phạm vi, ngữ cảnh hoặc một coroutine cụ thể. Các tham số kèm theo cho biết nguyên nhân việc hủy coroutine.

Chú ý: hàm **cancel()** không hủy được coroutine chạy với ngữ cảnh **NonCancellable**

Ví dụ

```
runBlocking {
    val job1 = launch {
        repeat(10) {
            delay(1000L)
            println("Job1: ${it + 1}")
        }
    }

    val job2 = launch {
        println("Waiting for 5 seconds then cancel Job1")
        delay(5000L)
        job1.cancel()
    }
}
```

job2 hủy **job1** sau 5 giây, **job1** đếm đến 4 thì bị dừng.

Hàm withTimeout()

```
suspend fun <T> withTimeout(timeMillis: Long, block: suspend
CoroutineScope.() -> T): T
```

- Chạy đoạn chương trình (có thể bị suspend) trong một coroutine với thời gian giới hạn (tính theo ms).
- Nếu hết thời gian mà đoạn chương trình chưa hoàn thành thì tạo exception thuộc kiểu TimeoutCancellationException.

Ví dụ

```
runBlocking {
    withTimeout(5000L) {
        repeat(10) {
            delay(800L)
            println("Counting: ${it + 1}")
        }
    }
}
```

Kết quả thực hiện

```
Counting: 1
Counting: 2
Counting: 3
Counting: 4
Counting: 5
Counting: 6
Exception in thread "main" kotlinx.coroutines.TimeoutCancellationException: Timed out waiting for 5000 ms
```

Hàm async()

```
fun <T> CoroutineScope.async(
    context: CoroutineContext = EmptyCoroutineContext,
    start: CoroutineStart = CoroutineStart.DEFAULT,
    block: suspend CoroutineScope.() -> T
): Deferred<T>
```

- Được sử dụng để tạo và chạy một coroutine mới với các tham số tương tự hàm **launch()**
- Trả về kết quả trong tương lai của coroutine trong một đối tượng thuộc lớp **Deferred** (là lớp con của lớp Job)
- Kết quả của coroutine có được khi hoàn thành và được trả về bởi hàm **await()**

Hàm await()

- Đợi cho đến khi có kết quả của đối tượng **Deferred** (coroutine được tạo bởi async hoàn thành)

```
runBlocking {
    val value1: Deferred<Int> = async {
        println("Compute value 1")
        delay(3000L)
        println("Return value 1")
        return@async Random.nextInt(10, 100)
    }

    val value2: Deferred<Int> = async {
        println("Compute value 2")
        delay(1000L)
        println("Return value 2")
        return@async Random.nextInt(10, 100)
    }

    println("Value 1: ${value1.await()}")
    println("Value 2: ${value2.await()}")
}
```

Kết quả khi chạy:

```
Compute value 1
Compute value 2
Return value 2
Return value 1
Value 1: 57
Value 2: 84
```

CoroutineContext

- Các coroutine luôn hoạt động trong ngữ cảnh được mô tả bởi giá trị thuộc kiểu CoroutineContext.
- Các thành phần chính của một CoroutineContext
  - Job – nắm giữ thông tin về lifecycle của coroutine
  - Dispatcher – quyết định luồng (thread) nào mà coroutine hoạt động trên đó
- Các loại dispatcher
  - Dispatchers.Main – chạy trên luồng giao diện
  - Dispatchers.IO – chạy trên luồng background
  - Dispatchers.Defaults – chạy trên luồng background
  - newSingleThreadContext("thread_name") – chạy trên luồng mới với tên là **thread_name**

Ví dụ

```
runBlocking {
    launch { // context of the parent, main runBlocking coroutine
        println("main runBlocking       : I'm working in thread ${Thread.currentThread().name}")
    }
    launch(Dispatchers.Default) { // will get dispatched to DefaultDispatcher
        println("Default                : I'm working in thread ${Thread.currentThread().name}")
    }
    launch(Dispatchers.IO) { // will get dispatched to DefaultDispatcher
        println("IO                     : I'm working in thread ${Thread.currentThread().name}")
    }
    launch(newSingleThreadContext("MyOwnThread")) { // will get its own new thread
        println("newSingleThreadContext : I'm working in thread ${Thread.currentThread().name}")
    }
}
```

Kết quả thực hiện:

```
Default                : I'm working in thread DefaultDispatcher-worker-1
IO                     : I'm working in thread DefaultDispatcher-worker-2
newSingleThreadContext : I'm working in thread MyOwnThread
main runBlocking       : I'm working in thread main
```

Chuyển đổi giữa các ngữ cảnh

Sử dụng hàm **withContext()**

```
runBlocking {
    println("Started in ${Thread.currentThread().name}")
    withContext(Dispatchers.IO) {
        println("Changed to ${Thread.currentThread().name}")
    }
    println("Back to ${Thread.currentThread().name}")
}
```

Kết quả thực hiện:

```
Started in main
Changed to DefaultDispatcher-worker-1
Back to main
```

CoroutineScope

- Là phạm vi mà các coroutine hoạt động, gắn liền với lifecycle của đối tượng như activity, fragment.
- Khi lifecycle của các đối tượng này kết thúc thì các coroutines đang chạy trong phạm vi tương ứng cũng bị kết thúc.
- Các kiểu phạm vi:
  - CoroutineScope – interface cơ bản, tạo scope với context tùy chọn
  - GlobalScope – được sử dụng để khởi tạo các coroutines bậc cao hoạt động suốt vòng đời của ứng dụng
  - MainScope – tạo scope với context là Main thread
  - lifecycleScope – scope gắn với Activity, Fragment
  - viewModelScope – scope gắn với ViewModel

Ví dụ

```
val scope = CoroutineScope(Dispatchers.Default)
runBlocking(Dispatchers.Default) {
    scope.launch {
        repeat(10) {
            delay(1000L)
            println("Job1: ${it + 1}")
        }
    }
    scope.launch {
        repeat(10) {
            delay(800L)
            println("Job2: ${it + 1}")
        }
    }

    delay(5000L)
    scope.cancel()
}
```