Coroutine

What is Coroutine

> Coroutines are computer program components that generalize subroutines for non-preemptive multitasking, by allowing execution to be suspended and resumed.

실행이 일시 중지되고 재개 될 수 있도록하여 비선 점형 멀티 태스킹을 위해 서브 루틴을 일반화하는 컴퓨터 프로그램 구성 요소입니다.

- Wikipedia
- difference-between-subroutine-co-routine-function-and-thread
- coroutine vs thread

Coroutine : Only in Kotlin?

NO!

- coroutine은 하나의 개념이다. 개념 자체는 1958년에 나왔다.
- 여러가지 언어들(C++, Kotlin, Javscript, ...)로 구현되어있다. 복잡한 연산이 많은 Unity에서도 사용되고 있다고 함
- Kotlin의 경우 언어에서 native하게 지원을 해준다고 보면 된다.

Coroutine 특징

Coroutine ~= Light-weight Thread

- 경량화 스레드라고 불리는 이유는 스레드의 라이프 사이클과 비슷하고, 스레드에 의해 코루틴이 실행되지만 스레드에 종속적이지 않기 때문에 스레드를 신경쓸 필요 없이 보여서 코루틴만 신경쓰면 되므로 스레드처럼 보이는게 있어서 인것 같다.
- 스레드는 native 스레드 (OS단)에 직접 매핑되어 관리하는 반면, 코루틴은 실제 사용하는 유저가 관리하기 때문에 오버헤드가 적다. (컨텍스트 스위칭 비용 low)
- 코루틴은 스레드의 생성보다 훨씬 빠르고 저렴한 비용이 드는 것이 특징.

동시성 프로그래밍 지원

Kotlin의 경우 `suspend` 키워드를 사용하여 동시성 프로그래밍을 지원한다. (병렬 프로그래밍도 지원함)

suspend fun의 의미는 현재 동작중인 스레드를 차단하지 않고 코루틴 실행을 cancel, resume 할 수 있다는 것을 의미한다.

> 동시성 프로그래밍 vs 병렬 프로그래밍(여담)
>
> > 동시성 : 한 사람이 두개의 큐브를 왔다갔다 하면서 맞추는것
> > 병렬 : 두 사람이 동시에 두개의 큐브를 왔다갔다 하면서 맞추는것

Coroutine Examples(주요 개념들)

runBlocking, launch

- runBlocking은 블록 안의 코드들이 마무리 될 때 까지 쓰레드를 블록한다. 따라서 End function 이 마지막에 찍힘
- launch block은 백그라운드로 job을 돌리도록 도와주는 dsl이다.

```
fun main() {
    runBlocking<Unit> {
        launch {
            delay(1000L)
            println("World")
        }
        println("Hello")
        delay(2000L)
    }
    println("End function")
}
// 출력
// Hello
// World
// End function
```

coroutine은 내부 coroutine들이 완료되어야 종료

```
fun main() {
    runBlocking {
        val jobs = List(10) {
            launch {
                delay(1000L)
                println("aaa")
            }
        }
        // join을 하고 안하고에 따라 End runBlock이 먼저 찍힐지 끝나고 찍힐지가 결정된다
        // jobs.forEach { it.join() }

        println("End runBlock ")
    }
    println("End function")
}

// End runBlock
// aaa
// aaa
// ...
// End Function
```

Coroutine 코드 함수화

suspend fun 안에서는 coroutine api(delay 등)를 사용할 수 있습니다.

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

// this is your first suspending function
suspend fun doWorld() {
    delay(1000L)
    println("World!")
}
```

Coroutine이 light-weight라는 증거

Thread로 아래와 같은 작업을 하면 OOM 나고 죽는다.

```
fun main() = runBlocking {
    repeat(100_000) { // launch a lot of coroutines
        launch {
            delay(1000L)
            print(".")
        }
    }
}
```

Global coroutines ~= daemon

GlobalScope은 process가 죽으면 같이 종료된다. 데몬같은 개념이라고 보면 된다.

```
fun main() = runBlocking {
    GlobalScope.launch {
        repeat(1000) { i ->
            println("I'm sleeping $i ...")
            delay(500L)
        }
    }
    delay(1300L) // just quit after delay
}
// I'm sleeping 1 ...
// I'm sleeping 2 ...
// I'm sleeping 3 ...
// !== process exit
```

Coroutine Examples(asynchronous codes)

앞으로 사용할 두 suspend 함수들

```
suspend fun doSomethingUsefulOne(): Int {
    delay(1000L) // pretend we are doing something useful here
    return 1
}

suspend fun doSomethingUsefulTwo(): Int {
    delay(1000L) // pretend we are doing something useful here, too
    return 2
}
```

Without async block

```
fun main() = runBlocking {
    val time = measureTimeMillis {
        val one = doSomethingUsefulOne()
        val two = doSomethingUsefulOne()
        println("The answer is ${one + two}")
    }

    println("Completed in $time ms")
}

// The answer is 3
// The answer is 2017 ms
```

With async block

```
fun main() = runBlocking {
    val time = measureTimeMillis {
        val one = async { doSomethingUsefulOne() }
        val two = async { doSomethingUsefulOne() }
        println("The answer is ${one.await() + two.await()}")
    }

    println("Completed in $time ms")
}

// The answer is 3
// The answer is 1017 ms
```

Lazy async evaluation

```
fun main() = runBlocking {
    val time = measureTimeMillis {
        val one = async(start = CoroutineStart.LAZY) { doSomethingUsefulOne() }
        val two = async (start = CoroutineStart.LAZY) { doSomethingUsefulTwo() }
        // some computation
        one.start() // start the first one
        two.start() // start the second one

        println("The answer is ${one.await() + two.await()}")
    }

    println("Completed in $time ms")
}

// The answer is 3
// The answer is 1017 ms
```

Async-style functions

GlobalScope은 coroutine에서 권장하지 않는 문법입니다. ...OneAsync() 함수에서 exception이 날 경우 try catch로 exception handling은 할 수 있지만 비동기 job은 유지된채 남습니다.

coroutine은 javascript의 async 함수와는 다르게 실행하는 곳에서 async 여부를 결정합니다.

```
fun somethingUsefulOneAsync() = GlobalScope.async { doSomethingUsefulOne() }
fun somethingUsefulTwoAsync() = GlobalScope.async { doSomethingUsefulTwo() }

fun main() {
    val time = measureTimeMillis {
        val one = doSomethingUsefulOneAsync()
        val two = doSomethingUsefulTwoAsync()

        runBlocking {
            println("The answer is ${one.await() + two.await()}")
        }
    }

    println("Completed in $time ms")
}
```

Coroutine Examples(with thread blocking code)

Coroutine Additional Concepts

CoroutineScope

- 코루틴의 범위, 코루틴 블록을 묶음으로 제어할 수 있는 단위
- 예제로 봤던 GlobalScope(async 예제)도 CoroutineScope의 한 종류이다. 이 경우에는 프로그램 전반에 걸쳐 백그라운드로 동작하는 scope을 가진다

CoroutineContext

- Coroutine을 어떻게 처리할 것인지에 대한 여러가지 정보의 집합
- 주요 요소로는 Job과 Dispatcher가 있다. Job은 하나하나의 코루틴 블록을 의미한다.

Coroutine Additional Concepts

Dispatcher

- CoroutineContext 의 주요 요소. Thread를 어떻게 관리할지 정의하는 곳이다
- 파라미터 없이 launch를 사용한다면 부모 CoroutineScope의 context와 dispatcher를 그대로 상속받음

```
fun main() = runBlocking {
    // main runBlocking : main
    launch { println("main : ${Thread.currentThread().name}") }

    // Unconfined : main
    launch(Dispatchers.Unconfined) { println("Unconfined : ${Thread.currentThread().name}") }

    // Default : DefaultDispatcher-worker-1 -> GlobalScope에서 launch 한 것과 동일
    launch(Dispatchers.Default) { println("Default : ${Thread.currentThread().name}") }

    // newSingleThreadContext : MyOwnThread
    launch(newSingleThreadContext("MyOwnThread")) {
        println("newSingleThreadContext : ${Thread.currentThread().name}")
    }
}
```

Asynchronous code with thread blocking call

```
class UserRepository {
    fun findById(id: Long): User? {
        Thread.sleep(1_000) // thread blocking code
        return User(id = id, name = UUID.randomUUID().toString())
    }
}
```

Asynchronous code with thread blocking call

```
class UserService(
    private val userRepository: UserRepository
) {
    fun findUsersSync() {
        val time = measureTimeMillis {
            val firstUser = userRepository.findById(1L)
            val secondUser = userRepository.findById(2L)
            println("$firstUser, $secondUser")
        }

        // 2010ms
        println("time = ${time}ms)
    }
}
```

Asynchronous code with thread blocking call

```
class UserService(
    private val userRepository: UserRepository
) {
    suspend fun findUsersAsyncInWrongWay() {
        val time = measureTimeMillis {
            val firstUser = async { userRepository.findById(1L) }
            val secondUser = async { userRepository.findById(2L) }
            println("${firstUser.await()}, ${secondUser.await()}")
        }

        // 2009ms
        println("time = ${time}ms)
    }
}
```

Asynchronous code with thread blocking call

```
class UserService(
    private val userRepository: UserRepository
) {
    suspend fun findUsersAsyncInCorrectWay() {
        val time = measureTimeMillis {
            val firstUser = async(Dispatchers.IO) { userRepository.findById(1L) }
            val secondUser = async(Dispatchers.IO) { userRepository.findById(2L) }
            println("${firstUser.await()}, ${secondUser.await()}")
        }

        // 1010ms
        // 다른 쓰레드에서 코드가 실행되기에 병렬적으로 동작을 하긴 했지만,
        // IO dispatcher가 JDBC blocking call들로 병목을 잡고 있을 수 있다.
        // 따라서 이런 경우에는 DB에 접근하는 call들만 따로 관리하는 Dispatcher를 만들어야 한다.
        println("time = ${time}ms")
    }
}
```

Asynchronous code with thread blocking call

```
package co.lopun.coroutines

import kotlinx.coroutines.CoroutineDispatcher
import kotlinx.coroutines.reactor.asCoroutineDispatcher
import java.util.concurrent.Executors
// Scheduler == Dispatcher in Reactor(Webflux)
import reactor.core.scheduler.Schedulers


// 필요에 따라서 Dispatcher들을 프로젝트별로 나눠도 되고 IO, COMPUTE 외의 다른 Dispatcher를 만들어도 된다.
enum class Dispatchers(val dispatcher: CoroutineDispatcher) {
  DB_SCHEDULER(Schedulers.newBoundedElastic(100, 100_000, "DB").asCoroutineDispatcher()),
  DB_THREAD_POOL(Executors.newFixedThreadPool(threads).asCoroutineDispatcher()),
  COMPUTE(Schedulers.parallel().asCoroutineDispatcher())
}

fun main() = runBlocking {
    async(Dispatchers.DB_SCHEDULER.dispatcher) { /* blocking call */ }
    async(Dispatchers.DB_THREAD_POOL.dispatcher) { /* blocking call */ }
}
```

References

- coroutine 강좌 시리즈(1~5)
- coroutine 개념 익히기
- KotlinConf 2017 - Introduction to Coroutines
- KotlinConf 2017 - Deep Dive into Coroutines on JVM
- marp - markdown ppt slide generator