Kotlin Coroutine(코루틴) - 개념과 사용법

만능 개발자 SKaSha 2019. 12. 8. 08:14

프로그램에서 태스크를 수행할 때 운영체제를 사용할 수 있게 하고 특정한 작업에 작업 시간을 할당하는 것을 선점한다라고 합니다.  
선점형 멀티태스킹(Preemptive Multitasking)은 운영체제가 강제로 태스크의 실행을 바꾸는 개념이고 협력형 멀티태스킹은 태스크들이 자발적으로 양보하며 실행을 바꿀 수 있는 개념입니다.

코루틴은 이러한 협력형 멀티태스킹을 이용하여 동시성 프로그래밍을 지원하는데,  
해당 루틴을 일시 중단(Suspended)하는 방식으로 Context-Switching을 없애고,  
최적화된 비동기 함수를 통해 비선점형으로 작동하기 때문에  
복잡한 넌블로킹 코드를 간결하게 해주며 더 나은 성능을 지원합니다.

**launch와 async**

코루틴에서 사용되는 함수는 suspend()로 선언된 지연 함수이어야 코틀린 기능을 사용할 수 있습니다.  
컴파일러는 suspend가 붙은 함수를 자동적으로 추출하여 Continuation 클래스로부터 분리된 루틴을 만듭니다.  
suspend 키워드는 사용자 함수에서도 사용할 수 있는데, 코루틴 블럭 외에서 사용하면 에러가 발생됩니다.

**launch 코루틴 빌더**

suspend fun doWork1(): String {

delay(1000)

return "Work1"}

suspend fun doWork2(): String {

delay(3000)

return "Work2"}

private fun worksInSerial(){ // 순차적 실행

GlobalScope.launch {

valone = doWork1()

valtwo = doWork2()

println("Kotlin One : $one")

println("Kotlin Two : $two") } }

결과값

Kotlin One : Work1 Kotlin Two : Work2

**async 코루틴 빌더**

async는 launch와 다르게 Deferred를 통하여 결과값을 반환하고, 지연된 결과값은 await()를 통해 받을 수 있습니다.  
Deferred는 launch의 리턴값인 Job을 상속받아 구현되었기 때문에 상태제어도 가능합니다.

private fun worksInParallel(){

// 컨텍스트에서 사용할 스레드 수를 정할 수 있다.

val threadPool = Executors.newFixedThreadPool(4)

val MyContext = threadPool.asCoroutineDispatcher()

// Deferred<T> 를 통해 결과값을 반환한다.

val one = GlobalScope.async(MyContext) {

doWork1() }

val two = GlobalScope.async {

doWork2() }

GlobalScope.launch {

val combined = one.await() + "\_"+ two.await()

println("Kotlin Combined : $combined") } }

결과값

Kotlin Combined : Work1\_Work2

**요약**

|  |  |
| --- | --- |
| **키워드** | **리턴값** |
| launch | Job |
| async | Deferred |

자바로 표현하자면 async는 CompletableFuture.supplyAsync()와 같으며, Deferred는 Future와 같은 기능입니다.

**runBlocking**

runBlocking은 새로운 코루틴을 실행하고 완료되기 전까지 현재 스레드를 블로킹합니다.

runBlocking { delay(2000) }

하기와 같이 클래스 내의 멤버 메서드에도 사용할 수 있습니다.

class MyClass { func mySuspendMethod()= runBlocking<Unit> { } }

**join과 cancel**

명시적으로 코루틴의 작업이 완료되는 것을 기다릴때는 join() 함수를,  
작업을 취소할때는 cancel() 함수를 이용할 수 있습니다.

fun main()= runBlocking<Unit> {

val job = launch {

delay(1000L)

println("World!") }

println("Hello")

job.join() // 명시적으로 코루틴이 완료되길 기다림

job.cancel() // 작업을 취소하려면 cancel을 이용

}

**Job 상태 흐름도**

백그라운드에서 실행되는 작업을 Job이라 하며, 부모 작업이 취소되면 하위 자식 작업들도 모두 취소됩니다.

**Job의 상태**

|  |  |  |  |
| --- | --- | --- | --- |
| **상태** | **isActive** | **isCompleted** | **isCancelled** |
| New | false | false | false |
| Active(기본값 상태) | true | false | false |
| Completing | true | false | false |
| Cancelling | false | false | true |
| Cancelled (최종 상태) | false | true | true |
| Completed (최종 상태) | false | true | false |

Job은 상태를 판별하기 위해 isActive, isCompleted, isCancelled 변수를 가지고 있습니다.  
Job이 활성화되면 활성 상태인 Active 상태가 되지만, Job() 팩토리 함수에 인자로 CoroutineStart.LAZY를 설정하면 활성상태가 아닌 New 상태로 생성됩니다.  
New 상태의 job을 Active 상태로 만들기 위해서는 start()나 join() 함수를 호출해야 합니다.

**Job의 상태 흐름도**

wait children

+-----+ start +--------+ complete +-------------+ finish +-----------+

| New | -----> | Active | ---------> | Completing | -------> | Completed |

+-----+ +--------+ +-------------+ +-----------+

| cancel / fail |

| +----------------+

| |

V V

+------------+ finish +-----------+

| Cancelling | --------------------------------> | Cancelled |

+------------+ +-----------+

**많은 양의 코루틴**

repeat(100\_000) { launch { delay(1000L) print("Hello!") } }

repeat() 함수를 이용하여 손 쉽게 많은 양의 코루틴을 생성할 수 있습니다.  
위 코드를 스레드로 바꾸면 Out of Memory Error가 발생하지만 코루틴은 내부적으로 몇 개의 스레드로 수많은 코루틴을 실행할 수 있기 때문에 오류가 발생하지 않으며 메모리나 실행 속도 면에서 큰 장점을 가집니다.

**코루틴과 시퀀스**

Sequence 함수 내부에서 코루틴 지연 함수를 사용하여 최종 형태를 나중에 결정할 수 있는 Lazy한 시퀀스를 만들 수 있습니다.  
이는 시퀀스 요소가 완전히 구성되기 전에 사용 범위와 시점을 결정할 수 있다는 것을 의미합니다.

// 피보나치 수열 생성

val fibonacciSeq = sequence {

var a = 0

var b = 1

yield(1) // (1) 지연 함수가 사용되어 코루틴 생성

while(true) {

yield(a + b) // (2)

val tmp = a + b

a = b

b = tmp } }

fun main() {

println(fibonacciSeq.take(8).toList())

// 다음 요소에 대한 지정

val saved = fibonacciSeq.iterator()

println("${saved.next()}, ${saved.next()}, ${saved.next()}") }

yield() 함수는 표현식을 계속 진행하기 전에 실행을 잠시 멈추고 요소를 반환한뒤 멈춘 시점에서 다시 실행을 재개합니다.  
그러므로 1번과 2번 작업이 일시 중지되었다가 다시 재개되는 부분입니다.

// generateSequence()의 사용

val seq = sequence {

val start = 0

// 단일 값 산출

yield(start)

// 반복 값 산출

yieldAll(1..5step 2)

// 무한한 시퀀스에서 산출

yieldAll(generateSequence(8) { it \* 3}) }

println(seq.take(7).toList()) // [0, 1, 3, 5, 8, 24, 72]

yield() 함수를 이용하는 방법의 경우 generateSequence() 함수를 이용하는것보다 sequence를 더 자유도 있게 생성할 수 있습니다.  
while과 같이 사용하여 특정 규칙에 따라 sequence를 만들 수도 있고 yield(value), yieldAll(list), yieldAll(sequence) 를 이용하여 보다 동적인 sequence를 생성할 수 있습니다.

**참고**

서적 : Do it! 코틀린 프로그래밍