mash-up android -3

코루틴 맛보기

발표 : 김유정

기능

코루틴은의 기능

- Lightweight(경량): 코루틴을 실행 중인 스레드를 차단하지 않는 정지(suspension)를 지원하므로 단일 스레드에서 많은 코루틴을 실행할 수 있다. suspension은 많은 동시 작업을 지원하면서도 메모리를 blocking에 비해 절약합니다.

- Fewer memory leaks(메모리 누수 감소): 구조화된 동시 실행을 사용하여 범위 내에서 작업을 실행한다

- Built-in cancellation support (기본으로 제공되는 취소 지원): 실행 중인 코루틴 계층 구조를 통해 자동으로 취소가 전달

- Jetpack 통합: 많은 Jetpack 라이브러리에 코루틴을 완전히 지원하는 확장 프로그램이 포함되어 있다. 일부 라이브러리는 구조화된 동시 실행에 사용할 수 있는 자체 코루틴 범위도 제공

코루틴 생김새(?)

```
import  kotlinx.coroutines. *

fun  main () {
     GlobalScope .launch { //백그라운드에서 새 코 루틴을 시작
        delay ( 1000L ) // 1초 지연 (기본 시간 단위는 ms)
        println ( " World! " ) // 지연 이후 print
    }
    println ( " Hello, " ) // 코루틴이 지연되는 동안 메인 스레드는 계속된다
    Thread .sleep ( 2000L ) // JVM을 유지하기 위해 2초동안 메인 스레드를 차단
    println("sleep2000")
}
```

코루틴 생김새(?)

```
import  kotlinx.coroutines. *

fun  main () {
     GlobalScope .launch { //백그라운드에서 새 코 루틴을 시작
        delay ( 1000L ) // 1초 지연 (기본 시간 단위는 ms)
        println ( " World! " ) // 지연 이후 print
    }
    println ( " Hello, " ) // 코루틴이 지연되는 동안 메인 스레드는 계속된다
    Thread .sleep ( 2000L ) // JVM을 유지하기 위해 2초동안 메인 스레드를 차단
    println("sleep2000")
}
```

GlobalScope.launch { ... } 와 delay(…)

==

thread { ... } ,Thread.sleep(...)

코루틴 생김새(?)

```
import  kotlinx.coroutines. *

fun  main () {
     GlobalScope .launch { //백그라운드에서 새 코 루틴을 시작
        delay ( 1000L ) // 1초 지연 (기본 시간 단위는 ms)
        println ( " World! " ) // 지연 이후 print
    }
    println ( " Hello, " ) // 코루틴이 지연되는 동안 메인 스레드는 계속된다
    Thread .sleep ( 2000L ) // JVM을 유지하기 위해 2초동안 메인 스레드를 차단
    println("sleep2000")
}
```

결과

```
Hello,
World!
sleep2000
```

주요 핵심 키워드

CoroutineScope (GlobalScope)

CoroutineContext

Dispatcher

launch & async

CoroutineScope (GlobalScope)

- CoroutineScope: 코루틴의 범위, 코루틴 블록을 묶음으로 제어할수 있는 단위
- GlobalScope : CoroutineScope 의 한 종류로 전체 어플리케이션 수명 동안에 작동하고, 취소되지 않는 최상위 수준의 동시 처리를 시작하는데 사용
  응용 프로그램 코드는 일반적으로 응용 프로그램 정의 CoroutineScope를 사용해야하며 , GlobalScope 인스턴스에서 비동기 또는 시작 을 사용하는 것은 매우 권장되지 않습니다.

CoroutineContext

- CoroutineContext 는 코루틴을 어떻게 처리 할것인지 에 대한 여러가지 정보의 집합
- 주요 요소 로는 Job 과 dispatcher 가 있다.

Dispatcher

- **Dispatcher 는 CoroutineContext 의 주요 요소로 해당 Coroutine이 실행을 위해 사용하는 스레드를 정의 해둠**

  - Dispatchers.Default : 디스패처 또는 기타 ContinuationInterceptor 가 컨텍스트에 지정 되지 않은 경우 모든 표준 빌더에서 사용됩니다 . 공유 백그라운드 스레드의 공통 풀을 사용한다.
    이는 **CPU 사용량이 많은 작업**에 적합하다.
  - Dispatchers.IO — 요청시 생성 된 스레드의 공유 풀을 사용하며 IO 작업이나 작업을 차단하는 것에 적합하다.
  - Dispatchers.Main : 응용 프로그램 "Main"또는 "UI"스레드로 제한되고 코루틴이 이미 올바른 컨텍스트에있을 때 즉시 실행하는 디스패처를 반환, 안드로이드의 경우 UI 스레드를 사용
  - Dispatchers.Unconfined — 첫 번째 중단까지 현재 호출 프레임에서 코 루틴 실행을 시작합니다. 이때 코 루틴 빌더 함수가 반환됩니다. 코 루틴은 특정 스레드 또는 풀로 제한하지 않고 해당 일시 중단 함수에서 사용하는 스레드에서 나중에 다시 시작된다. **디스패처는 일반적으로 코드에서 사용할 수 없다** .
  - newSingleThreadContext 는 코루틴이 실행할 스레드를 생성한다. 전용 스레드는 매우 비싼 리소스로 실제 애플리케이션에서는 더 이상 필요하지 않을 때 닫기 함수를 사용하여 해제 하거나 최상위 변수에 저장하고 애플리케이션 전체에서 재사용해야한다.

Dispatcher example

```
launch { // context of the parent, main runBlocking coroutine
    println("main runBlocking : I'm working in thread ${Thread.currentThread().name}")
}
launch(Dispatchers.Unconfined) { // not confined -- will work with main thread
    println("Unconfined : 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(newSingleThreadContext("MyOwnThread")) { // will get its own new thread
    println("newSingleThreadContext: I'm working in thread
${Thread.currentThread().name}")
}
```

Dispatcher example

```
launch { // context of the parent, main runBlocking coroutine
    println("main runBlocking : I'm working in thread ${Thread.currentThread().name}")
}
launch(Dispatchers.Unconfined) { // not confined -- will work with main thread
    println("Unconfined : 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(newSingleThreadContext("MyOwnThread")) { // will get its own new thread
    println("newSingleThreadContext: I'm working in thread
${Thread.currentThread().name}")
}
```

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

Dispatchers.Unconfined example

```
launch(Dispatchers.Unconfined) { // not confined -- will work with main thread
    println("Unconfined      : I'm working in thread ${Thread.currentThread().name}")
    delay(500)
    println("Unconfined      : After delay in thread ${Thread.currentThread().name}")
}
launch { // context of the parent, main runBlocking coroutine
    println("main runBlocking: I'm working in thread ${Thread.currentThread().name}")
    delay(1000)
    println("main runBlocking: After delay in thread ${Thread.currentThread().name}")
}
```

Dispatchers.Unconfined example

```
launch(Dispatchers.Unconfined) { // not confined -- will work with main thread
    println("Unconfined      : I'm working in thread ${Thread.currentThread().name}")
    delay(500)
    println("Unconfined      : After delay in thread ${Thread.currentThread().name}")
}
launch { // context of the parent, main runBlocking coroutine
    println("main runBlocking: I'm working in thread ${Thread.currentThread().name}")
    delay(1000)
    println("main runBlocking: After delay in thread ${Thread.currentThread().name}")
}
```

```
Unconfined      : I'm working in thread main
main runBlocking: I'm working in thread main
Unconfined      : After delay in thread kotlinx.coroutines.DefaultExecutor
main runBlocking: After delay in thread main
```

Dispatchers.Unconfined example

```
launch(Dispatchers.Unconfined) { // not confined -- will work with main thread
    println("Unconfined      : I'm working in thread ${Thread.currentThread().name}")
    delay(500)
    println("Unconfined      : After delay in thread ${Thread.currentThread().name}")
}
launch { // context of the parent, main runBlocking coroutine
    println("main runBlocking: I'm working in thread ${Thread.currentThread().name}")
    delay(1000)
    println("main runBlocking: After delay in thread ${Thread.currentThread().name}")
}
```

```
Unconfined      : I'm working in thread main
main runBlocking: I'm working in thread main
Unconfined      : After delay in thread kotlinx.coroutines.DefaultExecutor
main runBlocking: After delay in thread main
```

- Dispatcher를 Unconfined로 설정하면, 해당 coroutine은 callar thread에서 시작된다.
  단, 이 코루틴이 suspend되었다가 상태가 재시작 되면 적절한 thread에 재할당되어 시작된다

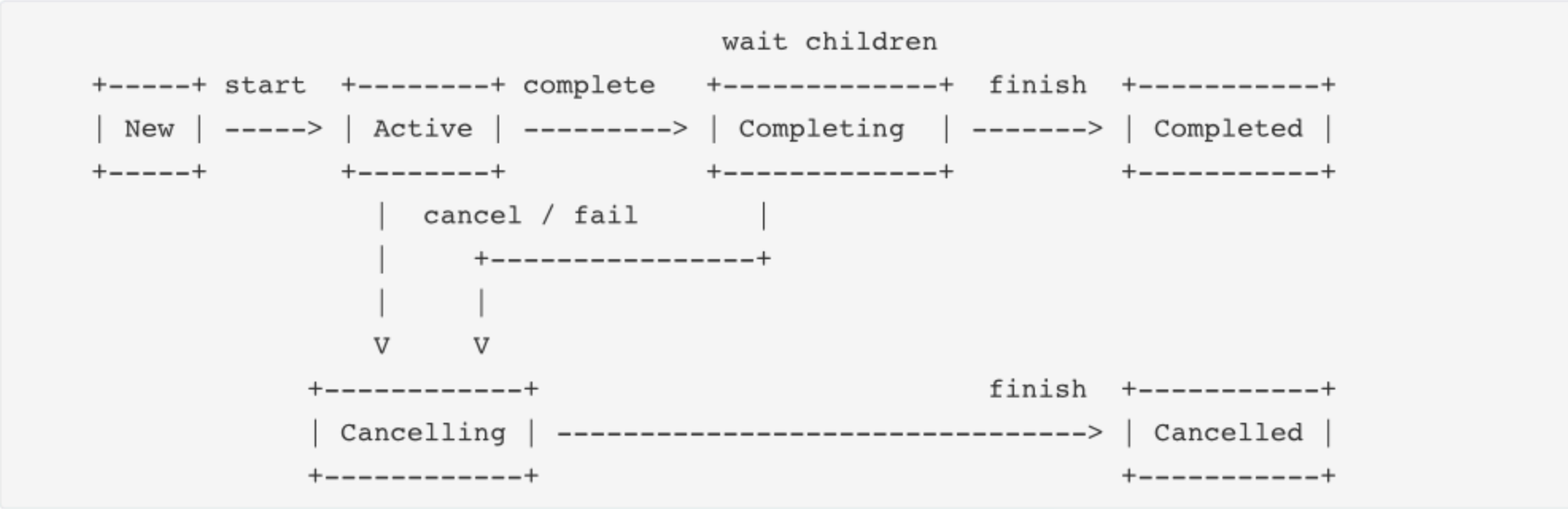
launch & job

launch & job

'launch' 는 현재 스레드를 차단하지 않고 새 코루틴을 시작하며 코루틴에 대한 참조를 Job 으로 반환

```
val job : Job = launch { ... }
```

launch & job - job 의 lifecycle



start : 현재의 coroutine의 동작 상태를 체크하며, 동작 중인 경우 true, 준비 또는 완료 상태이면 false를 return 한다.

join : 현재의 coroutine 동작이 끝날 때까지 대기한다. 다시 말하면 async {} await처럼 사용할 수 있다.

cancel : 현재 coroutine을 즉시 종료하도록 유도만 하고 대기하지 않는다. 다만 타이트하게 동작하는 단순 루프에서는 delay가 없다면 종료하지 못한다.

launch & job - job example

```
//1
val job = GlobalScope.launch { ... }
//2
val job = Job()
CoroutineScope(Dispatchers.Default + job).launch { ... }
```

launch & job - job example

```
        val job1 = launch {
            println("Job1 one scope for start")
            var i = 0
            for (index in 0..5) {
                delay(100)
                println("Job1 one scope  $i ")
            }
        }

        val job2 = launch {
            println("Job2 two scope for start")
            for (index in 0..2) {
                println("Job2 one scope index $index")
                delay(100)
            }
        }
//        job1.join()
//        job2.join()
        joinAll(job1, job2)
```

launch & job - job example

```
        val job1 = launch {
            println("Job1 one scope for start")
            var i = 0
            for (index in 0..5) {
                delay(100)
                println("Job1 one scope  $i ")
            }
        }

        val job2 = launch {
            println("Job2 two scope for start")
            for (index in 0..2) {
                println("Job2 one scope index $index")
                delay(100)
            }
        }
//        job1.join()
//        job2.join()
        joinAll(job1, job2)
```

```
Job1 one scope for start
Job2 two scope for start
Job2 one scope index 0
Job2 one scope index 1
Job1 one scope  0
Job2 one scope index 2
Job1 one scope  1
Job1 one scope  2
Job1 one scope  3
Job1 one scope  4
Job1 one scope  5
```

launch & job - job example

```
val job1 = GlobalScope.launch {
    println("Job1 one scope for start")
    var i = 0
    for (index in 0..5) {
        delay(100)
        println("Job1 one scope  $index ")
    }
}
 launch (job1){
    println("Job2 two scope for start")
    for (index in 0..2) {
        println("Job2 one scope index $index")
        delay(100)
    }
}
job1.join()
```

- 반환받은 Job 객체를 두 번째 launch 의 인자로 사용하면 Job 객체 하나로도 두개의 코드블록을 제어 할 수 있다.

async() — Deferred

async() — Deferred

'async()' 함수로 시작된 코루틴 블록은 Deferred 객체를 반환

```
val deferred : Deferred<T> = async {...}
```

async() — Deferred example

```
val deferred : Deferred<String> = async {
    var i = 0
    for (index in 0..5) {
        println("async index $index")
        delay(100)
    }
    "결과를 반환"
}
val msg = deferred.await()
println(msg) // result 출력
```

async() — Deferred example

```
val deferred : Deferred<String> = async {
    var i = 0
    for (index in 0..5) {
        println("async1 index $index")
        delay(100)
    }
    "결과 1"
}
async(deferred) {
    var i = 0
    for (index in 0..5) {
        println("async2 index $index")
        delay(100)
    }
    "결과 2"
}

val msg = deferred.await()
println(msg) // 결과 1 출력
```

- launch 의 job과 같이 반환받은 deferred 객체를 두 번째 async 의 인자로 사용하면 deferred 객체 하나로도 두개의 코드블록을 제어 할 수 있다.

LAZY— 지연 실행

LAZY— 지연 실행

```
val job = launch (start = CoroutineStart.LAZY) {...}
또는
val deferred = async (start = CoroutineStart.LAZY) {...}
```

코루틴은 launch 나 async 메소드가 실행되면 바로 시작되지만 LAZY 를 사용해 처리시점을 지연시킬 수 있다.

각 코루틴 블록 함수의 start 인자에 CoroutineStart.LAZY 를 사용하면 해당 코루틴 블록은 지연 된다.

LAZY— 지연 실행

- launch 코루틴 블록을 지연 실행 시킬 경우 Job 클래스의 job.start()또는job.join()함수 를 호출하는 시점에 launch 코드 블록이 실행된다.

```
val job = launch (start = CoroutineStart.LAZY) {...}
또는
val deferred = async (start = CoroutineStart.LAZY) {...}
```

LAZY— 지연 실행

```
val job = launch (start = CoroutineStart.LAZY) {...}
또는
val deferred = async (start = CoroutineStart.LAZY) {...}
```

- async 코루틴 블록을 지연 실행 시킬 경우 Deferred 클래스의 deferred.start() 또는deferred.await() 함수를 호출하는 시점에 async 코드 블록이 실행된다.

LAZY— 지연 실행

```
val job = launch (start = CoroutineStart.LAZY) {...}
또는
val deferred = async (start = CoroutineStart.LAZY) {...}
```

- **async 코루틴 블록을 지연 실행 시킬 경우 Deferred 클래스의 deferred.start() 또는deferred.await() 함수를 호출하는 시점에 async 코드 블록이 실행된다.**

- 지연된 **async** 코루틴 블록 의 경우
  start() 함수는 **async** 코루틴 블록을 실행 시키지만 블록의 수행 결과를 반환하지 않는다.
  또한 **await()** 함수와 다르게 코루틴 블록이 완료 되는것을 기다리지도 않는다.

LAZY— 지연 실행

`deferred.await()` 함수를 사용해 async 코드 블록이 실행

```
println("stat")
  val deferred= async (start = CoroutineStart.LAZY){
      var i = 0
      for (index in 0..5) {
          println("async1 index $index")
          delay(100)
      }
  }
  deferred.await()
  println("end")
```

- 지연된 **async** 코루틴 블록 의 경우
  start() 함수는 **async** 코루틴 블록을 실행 시키지만 블록의 수행 결과를 반환하지 않는다.
  또한 **await()** 함수와 다르게 코루틴 블록이 완료 되는것을 기다리지도 않는다.

LAZY— 지연 실행

`deferred.await()` 함수를 사용해 async 코드 블록이 실행

```
println("stat")
  val deferred= async (start = CoroutineStart.LAZY){
      var i = 0
      for (index in 0..5) {
          println("async1 index $index")
          delay(100)
      }
  }
  deferred.await()
  println("end")
```

- result : end 가 실행 후에 호출된다

```
stat
async1 index 0
async1 index 1
async1 index 2
async1 index 3
async1 index 4
async1 index 5
end
```

- 지연된 **async** 코루틴 블록 의 경우
  start() 함수는 **async** 코루틴 블록을 실행 시키지만 블록의 수행 결과를 반환하지 않는다.
  또한 **await()** 함수와 다르게 코루틴 블록이 완료 되는것을 기다리지도 않는다.

LAZY— 지연 실행

- `deferred.start()` 함수를 사용해 async 코드 블록이 실행

```
println("stat")
val deferred= async (start = CoroutineStart.LAZY){
    var i = 0
    for (index in 0..5) {
        println("async1 index $index")
        delay(100)
    }
}
deferred.start()
println("end")
```

- 지연된 **async** 코루틴 블록 의 경우
  start() 함수는 **async** 코루틴 블록을 실행 시키지만 블록의 수행 결과를 반환하지 않는다.
  또한 **await()** 함수와 다르게 코루틴 블록이 완료 되는것을 기다리지도 않는다.

LAZY— 지연 실행

- `deferred.start()` 함수를 사용해 async 코드 블록이 실행

```
println("stat")
val deferred= async (start = CoroutineStart.LAZY){
    var i = 0
    for (index in 0..5) {
        println("async1 index $index")
        delay(100)
    }
}
deferred.start()
println("end")
```

- result : end는 start가 출력되자마자 출력한다.

```
stat
end
async1 index 0
async1 index 1
async1 index 2
async1 index 3
async1 index 4
async1 index 5
```

- 지연된 **async** 코루틴 블록 의 경우
  start() 함수는 **async** 코루틴 블록을 실행 시키지만 블록의 수행 결과를 만완하시 않는나.
  또한 **await()** 함수와 다르게 코루틴 블록이 완료 되는것을 기다리지도 않는다.

runBlocking()

runBlocking()

runBlocking() 함수는 코드 블록이 작업을 완료할 때 까지 기다린다.

```
runBlocking {
    ...
}
```

runBlocking()

runBlocking() 함수는 코드 블록이 작업을 완료할 때 까지 기다린다.

```
runBlocking {
    ...
}
```

단순히 *join* 같은 메소드 기능이 아닌 현재 *thread* 를 *block* 하고 실행되는 코드 이므로 메인 *thread* 사용에 주의해야한다.

안드로이드 의 경우 *runBlocking()* 함수를 메인 *thread (UI thread)* 에서 호출하여 시간이 오래 걸리는 작업을 수행하는 경우 *ANR* 이 발생할 위험이 있으므로 주의해야하며 일반적인 *Blocking* 코드와 *suspend* 스타일로 적힌 라이브러리들을 *bridge*해줄 모적으로 설계된 함수로 메인 함수나 테스트에서 사용하는 것이 바람직하다.