[[Kotlin] 코틀린 - 코루틴#4 - context와 dispatchers](https://tourspace.tistory.com/153)

2년 전 *by* 뚜덜이~



*이 글은 아래 링크의 내용을 기반으로 하여 설명합니다.*

*<https://github.com/Kotlin/kotlinx.coroutines/blob/master/coroutines-guide.md>  
또한 예제에서 로그 print시 println과 안드로이드의 Log.e()를 혼용합니다.*

coroutine은 항상 kotlin standard library에 정의된 CoroutineContext로 대표되는 어떤 context에서 실행됩니다.

corouitne의 context는 어려 요소의 set으로 구성되며, main 요소는 **Job**과 **dispatcher** 입니다.

Dispatchers and threads

coroutine context는 어떤 쓰레드에서 해당 coroutine을 실행할지에 대한 dispatcher 정보를 담고 있습니다.

이 dispatcher에 따라서 실행될 특정 thread를 지정하거나, thread pool을 지정하거나, 또는 지정없이 사용할 수 있습니다.

launch 와 async같은 coroutine builder는 CoroutineContext값을 optional로 지정할 수 있고 이를 통해 dispatcher를 지정할 수 있습니다.

fun main()= runBlocking<Unit>{

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}") } }

위 코드 실행시 로그 print 순서를 다를수 있지만 아래와 같이 출력됩니다.

*Unconfined : I'm working in thread main*

*Default : I'm working in thread DefaultDispatcher-worker-1*

*newSingleThreadContext: I'm working in thread MyOwnThread*

*main runBlocking : I'm working in thread main*

**파라미터 없이 launch를 사용한다면 부모 CoroutineScope의 context와 dispatcher를 그대로 상속받습니다.**

여기서는 runBlocking이 부모이니 해당 context와 main thread에서 실행됩니다.

**Dispatchers.Unconfined**는 main에서 실행되었지만 좀 다른 mechanism을 같습니다. (이거 아래에서 따로 설명 합니다.)

**Dispatchers.Default**는 **GlobalScope에서 launch를 시킨것과 동일합니다.**

따라서 이 coroutine은 공유해서 사용하는 thread Pool을 이용합니다. (아마 ForkJoinPool을 이용하겠죠?)

따라서 아래 두개는 같은 코드 입니다.

 launch(Dispathers.Default){...}

 GlobalScope.launch{...}

하지만 취소시에는 GloblaScope을 쓴것과 dispacther.default인것과는 동작이 다릅니다.

이건 아래서 따로 설명합니다.

**newSingleThreadContext**는 코루틴을 실행시킬 새로운 thread를 생성합니다.

하지만 이렇게 singleThreadPool을 만들어 사용하는건 비싼 resource를 사용합니다.

(coroutine의 의도와 다르게 thread를 새로 띄우는 비용이 듭니다.)

이는 사용이 끝나면 반드시 release해야 하므로 use keyword와 같이 사용하든지, top level에 변수로 지정해서 재활용하는 형태로 써야 합니다.

Unconfined vs confined dispatcher

Dispatcher를 Unconfined로 설정하면, **해당 coroutine은 callar thread에서 시작됩니다.**

단, 이 코루틴이 **suspend되었다가 상태가 재시작 되면 적절한 thread에 재할당되어 시작됩니다.**

따라서 Unconfined는 CPU time을 소비하지 않는 작업이나, 공유되는 데이터에 접근하지 않아야 하는 작업들에서 이용하는게 적절합니다. (특정 스레드로 지정되어 처리되어야 하는경우에는 사용하지 않는다.)

즉 UI 처럼 main thread에서만 수행되어야 하는 작업들은 unconfined로 지정하면 안됩니다.

**dispatcher의 기본값은 외부 CoroutienScope의 값을 상속 받습니다.**

특히 runBlocking의 기본 dispatcher는 이를 시작한 **thread 입니다.**

따라서 내부에서 코루틴이 시작되는 경우 해당 코루틴들은 FIFO로 scheduling될꺼라는걸 예상해 볼 수 있습니다.

fun main()= runBlocking<Unit>{

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*

첫번째 launch는 unconfined로 시작했기 때문에 runBlocking을 시작된 main thread에서 수행됩니다.

두번재 launch역시 다른 인자 없이 사용했기에 runBlocking의 dispatcher를 사용합니다.

세번째 로그는 unconfined로 설정된 탓에 다른 thread에서 수행됩니다.

네번재 로그 역시 runBlocking의 dispatcher를 사용하기 때문에 main thread에서 수행됩니다.

Debugging coroutines and threads

코루틴은 suspend와 resume이 반복되면서 다른 thread로 변경될 수 있습니다. 따라서 하나의 thread만 사용하는 dispatcher라고 하더라도 현재 어떤 coroutine이 수행중인지는 확인하기 힘듭니다.

이를 좀더 쉽게 해주기 위해서 -Dkotlinx.coroutines.debug를 JVM option에 추가하면 coroutine에 대한 정보도 로그로 남길 수 있습니다.

fun log(msg: String) = println("[${Thread.currentThread().name}] $msg")

fun main()= runBlocking{

val a = async {

log("I'm computing a piece of the answer")

6}

val b = async {

log("I'm computing another piece of the answer")

7}

log("The answer is ${a.await() \* b.await()}") }

여기에는 runBlocking과 async 두개로 총 세개의 coroutine이 존재 합니다.

[main @coroutine#2] I'm computing a piece of the answer

[main @coroutine#3] I'm computing another piece of the answer

[main @coroutine#1] The answer is 42

Jumping between threads

아래 코드는 하나의 coroutine이 어떻게 다른 thread에서 분리되어 실행되는지를 보여줍니다.

fun log(msg: String)= println("[${Thread.currentThread().name}] $msg")

fun main() {

newSingleThreadContext("Ctx1").use { ctx1 ->

newSingleThreadContext("Ctx2").use { ctx2 ->

runBlocking(ctx1) {

log("Started in ctx1")

withContext(ctx2) {

log("Working in ctx2") }

log("Back to ctx1") } } } }

*[Ctx1 @coroutine#1] Started in ctx1*

*[Ctx2 @coroutine#1] Working in ctx2*

*[Ctx1 @coroutine#1] Back to ctx1*

여기서 주목해야할 두가지는 아래와 같습니다.

 **runBlocking에 인자로 context를 넣어줄 수 있다.**

 **withContext를 통해 sequential한 code flow는 유지하지만 필요한 부분만 다른 context로 전환시킬 수 있다.**

Job in the context

Job 역시 context의 일부 입니다. 따라서 **coroutineContext[Job]**을 통해 job을 코드안에서도 꺼내 올 수 있습니다.

fun main()= runBlocking{

println("My job is ${coroutineContext[Job]}") }

*My job is "coroutine#1":BlockingCoroutine{Active}@6d311334*

runBlocking안에서 쓰이는 isActive는 사실 **coroutineContext[Job].isActive == true**와 같습니다.

Children of a coroutine

CoroutineScope에서 다른 coroutine을 launch 시키면 해당 코루틴은 CoroutineScope.coroutineContext를 상속받고 이때 같이 새로 생성되는 Job 역시 부모 coroutine Job의 자식이 됩니다.

부모 코루틴이 취소되면 자식들 역시 recursive하게 전부 취소됩니다.

fun main()= runBlocking{

// launch a coroutine to process some kind of incoming request

valrequest = launch {

// it spawns two other jobs, one with GlobalScope

GlobalScope.launch {

println("job1: I run in GlobalScope and execute independently!")

delay(1000)

println("job1: I am not affected by cancellation of the request") }

// and the other inherits the parent context

launch {

delay(100)

println("job2: I am a child of the request coroutine")

delay(1000)

println("job2: I will not execute this line if my parent request is cancelled") } }

delay(500)

request.cancel() // cancel processing of the request

delay(1000) // delay a second to see what happens

println("main: Who has survived request cancellation?") }

job1: I run in GlobalScope and execute independently!

job2: I am a child of the request coroutine

job1: I am not affected by cancellation of the request

main: Who has survived request cancellation?

위에서 job1은 GlobalScope에서 띄운 launch이기 대문에 request의 자식이 아닙니다.

따라서 request Job을 cancel하더라도 job1은 취소되지 않습니다.

맨 처음 예제를 다시 보겠습니다.

해당 예제의 각 launch coroutine 내부에 delay(1000L)을 주고 로그를 하나씩 더 찍도록 했습니다.

fun test3\_4() {

runBlocking {

val job = launch(newSingleThreadContext("Parent Thread")) {

launch {

// context of the parent, main runBlocking coroutine

Log.e(TAG, "main runBlocking#1 :my thread ${Thread.currentThread().name}")

delay(1000L)

Log.e(TAG, "main runBlocking#2 : my thread ${Thread.currentThread().name}") }

launch(Dispatchers.Unconfined) {

// not confined -- will work with main thread

Log.e(TAG, "Unconfined#1 : my thread ${Thread.currentThread().name}")

delay(1000L)

Log.e(TAG, "Unconfined#2 : my thread ${Thread.currentThread().name}") }

launch(Dispatchers.Default) {

// will get dispatched to DefaultDispatcher

Log.e(TAG, "Default#1 : my thread ${Thread.currentThread().name}")

delay(1000L)

Log.e(TAG, "Default#2 : my thread ${Thread.currentThread().name}") } launch(newSingleThreadContext("MyOwnThread")) {

// will get its own new thread

Log.e(TAG, "newSingleThreadContext#1: my thread ${Thread.currentThread().name}")

delay(1000L)

Log.e(TAG, "newSingleThreadContext#2: my thread ${Thread.currentThread().name}") } }

delay(500L)

job?.cancelAndJoin() } }

그리고 500ms 이후에 부모를 취소합니다.

이렇게 한다면 각각 다른 dispatcher를 갖는 자식들은 취소될까요?  
dispatcher는 다르지만 job은 부모 job의 자식이므로 전부 취소되면서 #2가 표기된 로그들은 전부 찍히지 않습니다.

*Unconfined#1 : my thread Parent Thread*

*Default#1 : my in thread DefaultDispatcher-worker-1*

*main runBlocking#1 : my thread Parent Thread*

*newSingleThreadContext#1: my thread MyOwnThread*

Parental Responsibilities

부모 coroutine은 자식 coroutine이 끝날때까지 항상 대기합니다.

fun main()= runBlocking{

// launch a coroutine to process some kind of incoming request

val request = launch {

repeat(3) { i -> // launch a few children jobs

launch {

delay((i + 1) \* 200L) // variable delay 200ms, 400ms, 600ms

println("Coroutine $iis done") } }

println("request: I'm done and I don't explicitly join my children that are still active") }

request.join() // wait for completion of the request, including all its children

println("Now processing of the request is complete") }

*request: I'm done and I don't explicitly join my children that are still active*

*Coroutine 0 is done*

*Coroutine 1 is done*

*Coroutine 2 is done*

*Now processing of the request is complete*

여기서 쓰인 join()은 로그 순서를 보장하기위해 쓰였습니다. join()을 제거하더라도 모든 로그를 찍고 main()함수가 종료됩니다.

(단, "Now Proecessing..."이 찍히고 난 다음에 "Coroutine 0... Coroutine 1... Coroutine 2..."가 찍히겠죠?

Naming coroutines for debugging

Thread에 이름을 주듯 coroutine에도 이름을 줄 수 있습니다.

fun log(msg: String)= println("[${Thread.currentThread().name}] $msg")

fun main()= runBlocking(CoroutineName("main")) {

log("Started main coroutine")

// run two background value computations

val v1 = async(CoroutineName("v1coroutine")) {

delay(500)

log("Computing v1")

252}

val v2 = async(CoroutineName("v2coroutine")) {

delay(1000)

log("Computing v2")

6}

log("The answer for v1 / v2 = ${v1.await() / v2.await()}") }

*[main @main#1] Started main coroutine*

*[main @v1coroutine#2] Computing v1*

*[main @v2coroutine#3] Computing v2*

*[main @main#1] The answer for v1 / v2 = 42*

Combining context elements

context는 여러 요소를 가질수 있습니다. 복합적인 요소를 갖도록 하기 위해서는 **+** 연산자를 이용합니다.

fun main()= runBlocking{

launch(Dispatchers.Default + CoroutineName("test")) {

println("I'm working in thread ${Thread.currentThread().name}") } }

Cancellation via explicit job

안드로이드의 activity는 생명주기를 같는 object 입니다.

이처럼 해당 object에 종속적으로 동작해야 하는경우 Job을 직접 생성해서 사용할 수 있습니다.

class Activity: CoroutineScope {

lateinit varjob: Job

fun create(){

job = Job() }

fun destroy(){

job.cancel() }

// to be continued ...

// class Activity continues

override val coroutineContext: CoroutineContext

get() = Dispatchers.Default + job

// to be continued ...

// class Activity continues

fun doSomething(){

// launch ten coroutines for a demo, each working for a different time

repeat(10) { i ->

launch {

delay((i + 1) \* 200L) // variable delay 200ms, 400ms, ... etc

println("Coroutine $iis done") } } }

} // class Activity ends

fun main()= runBlocking{

val activity = Activity()

activity.create() // create an activity

activity.doSomething() // run test function

println("Launched coroutines")

delay(500L) // delay for half a second

println("Destroying activity!") activity.destroy() // cancels all coroutines

delay(1000) // visually confirm that they don't work

}  
  
먼저 Job 객체를 하나 생성합니다.

그리고 activity의 생성주기에 맞춰 create()와 destroy()에 Job의 할당과 cancel()을 추가합니다.

CoroutineScope를 상속받아 사용하므로 해당 activity 내에서는 코루틴을 함수들을 쉽게 사용할 수 있습니다.

이때 coroutineContext를 override하여 멤버변수로 생성해 놓은 Job을 conroutine context로 사용하도록 합니다.

따라서 activity 내부에서 시작되는 모든 corouitne들은 같은 context를 사용하고, activity가 종료시 전부 취소될 수 있습니다.

*Launched coroutines*

*Coroutine 0 is done*

*Coroutine 1 is done*

*Destroying activity!*

혹시나 하는 마음에 예제를 하나더 추가해 봅니다.

화면에서 DB의 정보를 읽어 화면을 갱신해야 하는경우 아래와 같이 구성할 수 있습니다.

class CoroutineActivity: CoroutineScope {

lateinit var job: Job

override val coroutineContext: CoroutineContext = job + Dispatchers.Main

fun onCreate(){

job = SupervisorJob() }

fun onDestroy(){

job.cancel() }

fun updateScreen(){

init()

launch(Dispatchers.Default) {

val listData = getDataFromRoom()

withContext(coroutineContext) {

//UI thread에서 화면 갱신

updateListUI(listData) } }

//기타 다른 작업 수행..

}

private fun init(){

//…

}

private suspend fun updateListUI(data: List<String>){

setAdatper(data) }

// Background에서 수행해야 하는 작업

private suspend fun getDataFromRoom():List<String> {

delay(1500)

return mutableListOf<String>("apple", "banana", "tomato") }

// List adpater에 data를 넣는다.

private suspend fun setAdatper(data: List<String>){

delay(100) } }

Thread-local data

Thread를 시작할때 해당 Thread 내부에서만 사용하기 위한 값으로 local data를 넘겨줘야 하는 경우가 있습니다.

다만 coroutine은 고정된 Thread에서만 수행되지 않고, 지속적으로 다른 thread로 변경되면서 수행될수 있기 때문에 이런 값을 공유하여 사용하기는 어렵습니다.

유사하게 coroutine의 경우 특정 코루틴 내부에서만 사용되는 local data를 지정할 수 있습니다.

이 값은 context에 추가적인 요소로 들어가면서 코루틴의 context가 동일하다면 get()하여 값을 얻어갈 수 있고, context가 switch되면 해당 context의 값으로 재설정 됩니다.

context에 값을 저장하기 위해서는 확장함수인 **ThreadLocal**과 **asContentElement** 가 있습니다.

val threadLocal = ThreadLocal<String?>() // declare thread-local variable

fun main()= runBlocking{

threadLocal.set("main")

println("Pre-main, current thread: ${Thread.currentThread()}, thread local value: '${threadLocal.get()}'")

val job = launch(Dispatchers.Default + threadLocal.asContextElement(value = "launch")) {

println("Launch start, current thread: ${Thread.currentThread()}, thread local value: '${threadLocal.get()}'")

yield()

println("After yield, current thread: ${Thread.currentThread()}, thread local value: '${threadLocal.get()}'") }

job.join()

println("Post-main, current thread: ${Thread.currentThread()}, thread local value: '${threadLocal.get()}'") }

위 예제에서 Dispatchers.Default로 코루틴을 launch 시킵니다.

따라서 이 코루틴은 **CommonPool**의 threads에 의해서 실행됩니다.

이때 **threadLocal.asContextElement**로 "*launch*"값을 설정했기 때문에 어떤 thread에서 해당 coroutine이 수행되든 ThreadLocal값은 "*launch*"를 유지합니다.

Pre-main, current thread: Thread[main @coroutine#1,5,main], thread local value: 'main'

Launch start, current thread: Thread[DefaultDispatcher-worker-1 @coroutine#2,5,main], thread local value: 'launch'

After yield, current thread: Thread[DefaultDispatcher-worker-2 @coroutine#2,5,main], thread local value: 'launch'

Post-main, current thread: Thread[main @coroutine#1,5,main], thread local value: 'main'

valmyThreadLocal = ThreadLocal<String?>()

...

println(myThreadLocal.get()) // Prints "null"

launch(Dispatchers.Default + myThreadLocal.asContextElement(value = "foo")) {

println(myThreadLocal.get()) // Prints "foo"

withContext(UI) {

println(myThreadLocal.get()) // Prints "foo", but it's on UI thread} }

println(myThreadLocal.get()) // Prints "null"

위 예제 역시ThreadLocal()을 사용하는 방법에 대한 예시 입니다.

ThreadLocal()에는 primitive type을 넣을수 있습니다.

다만, ThreadLocal()값을 변경하더라도 해당 context는 이를 적용하지 않습니다. (바뀔때마다 ThreadLocal()값을 tracking 할 수 없으므로)

myThreadLocal.set("main")

withContext(UI) {

println(myThreadLocal.get()) // Prints "main"

myThreadLocal.set("UI") }

println(myThreadLocal.get()) // Prints "main", not "UI"

만약 threadLocal값을 수정하고자 한다면 withContext를 사용해야 합니다.

withContext(myThreadLocal.asContextElement("foo")) {

println(myThreadLocal.get()) // Prints "foo"}

출처: <https://tourspace.tistory.com/153> [투덜이의 리얼 블로그]