

01 Hello Concurrent World

① 생성일	@2023년 3월 12일 오후 2:44
: 태그	

동시성이란.

코틀린이 동시성 문제를 어떻게 다루는지 소개

프로세스, 스레드, 코루틴 및 이들의 관계

운영체제 → 프로세스 생성 → 여기에 스레드 연결

그리고 메인 스레드를 실행함

프로세스

프로세스는 실행중인 애플리케이션의 인스턴스.

- 상태값을 가짐
 - 핸들, 프로세스id, 데이터, 네트워크 연결 등은 프로세스 상태의 일부
- 해당 프로세스 내부의 스레드가 액세스 가능

애플리케이션은 여러 프로세스로 구성될 수 있음. (인터넷 브라우저도 여러 프로세스로 구성됨)

그러나, 다중 프로세스 애플리케이션을 구현하는 데는 다양한 문제가 있음.

여기서는 단일 프로세스 안에서 하나 이상의 스레드를 실행하는 애플리케이션의 구현에만 한정해서 다룸

스레드

실행 스레드는 프로세스가 실행할 일련의 명령을 포함.

- 프로세스는 최소 하나의 스레드를 포함함. 그리고 이 스레드는 애플리케이션의 entry point를 실행하기 위해 생성됨 → 보통 `main()` 함수 aka. main thread
- 스레드가 끝나면 프로세스의 다른 스레드와 상관 없이 프로세스가 종료됨

```
fun main(args: Array<String> {  
    dowork()  
}
```

- 애플리케이션이 실행되면 `main()` 함수의 명령집합이 포함된 메인 스레드가 생성됨. 그리고 여기 예시의 `dowork()` 가 메인스레드에서 실행됨. → 해당 `dowork()` 함수 종료시 애플리케이션의 실행이 종료됨
- 스레드 안에서 명령은 한번에 하나씩 실행됨
 - 스레드가 블록되면 블록이 끝날때까지 같은 스레드에서 다른 명령을 실행할 수 없음
→ 블로킹 작업을 별도의 전용 스레드에 할당하는것이 UX에 덜 부정적인 영향을 미침
 - 많은 스레드가 같은 프로세스에서 생성될 수 있으며 서로 통신 가능
- 스레드는 프로세스의 리소스를 액세스하고 수정도 가능하며, local storage 라는 자체 저장소도 가짐

UI 스레드

GUI 애플리케이션에 있는 UI 스레드. 사용자 인터페이스를 업데이트하고 사용자와 애플리케이션 간의 상호작용을 리스닝함

- GUI 애플리케이션은 UI 스레드를 블록하지 않음. (애플리케이션의 응답성을 유지하기 위해서임. 스레드를 블록하면 애플리케이션이 UI를 업데이트하거나 사용자로부터 상호작용을 수신하지 못하기에...)
 - android 3.0 이상부터 불가능 하다고 함
 - android에서는 기본적으로 main thread가 UI thread임

코틀린이 동시성을 구현한 방식에서는 개발자가 직접 스레드를 시작하거나 중지할 필요가 없다

→ 코틀린이 특정 스레드나 스레드 풀을 생성해서 코루틴을 실행하도록 지시하기만 하면됨.
나머지 처리는 프레임워크가 할 것.

코루틴

경량 스레드로 정의됨.

스레드와 비슷한 라이프사이클을 갖고 있고, 스레드처럼 코루틴이 프로세서가 실행할 명령어 집합의 실행을 정의하기 때문.

코루틴과 스레드 간의 관계

- 코루틴은 스레드 안에서 실행됨 (그러나 종속된 관계가 아님)
- 스레드 하나에 많은 코루틴이 있을 수 있음
 - 하지만 2개 이상의 코루틴이 한 시점에 실행되지 않음

코루틴과 스레드의 차이점

- 코루틴을 스레드보다 빠르고 적은 비용으로 생성할 수 있다
 - 수천개의 스레드 생성하는것보다 수천개의 코루틴을 더 빠르고 쉽게 생성하고, 리소스도 더 적게 사용
 - 코틀린은 고정된 크기의 스레드 풀을 사용하고 코루틴을 스레드들에 배포하기 때문에 실행 시간이 매우 적게 증가함.
- 코루틴이 일시 중단되는 동안, 실행중인 스레드는 다른 코루틴을 실행하는데 사용됨



Thread 클래스의 `activeCount()` 를 사용하면 활성화된 스레드 수를 알 수 있음.

[테스트] 10,000개의 코루틴 생성시 4개의 스레드 생성됨. ↔ 코루틴 1개 생성시 두개의 스레드만 생성되었음

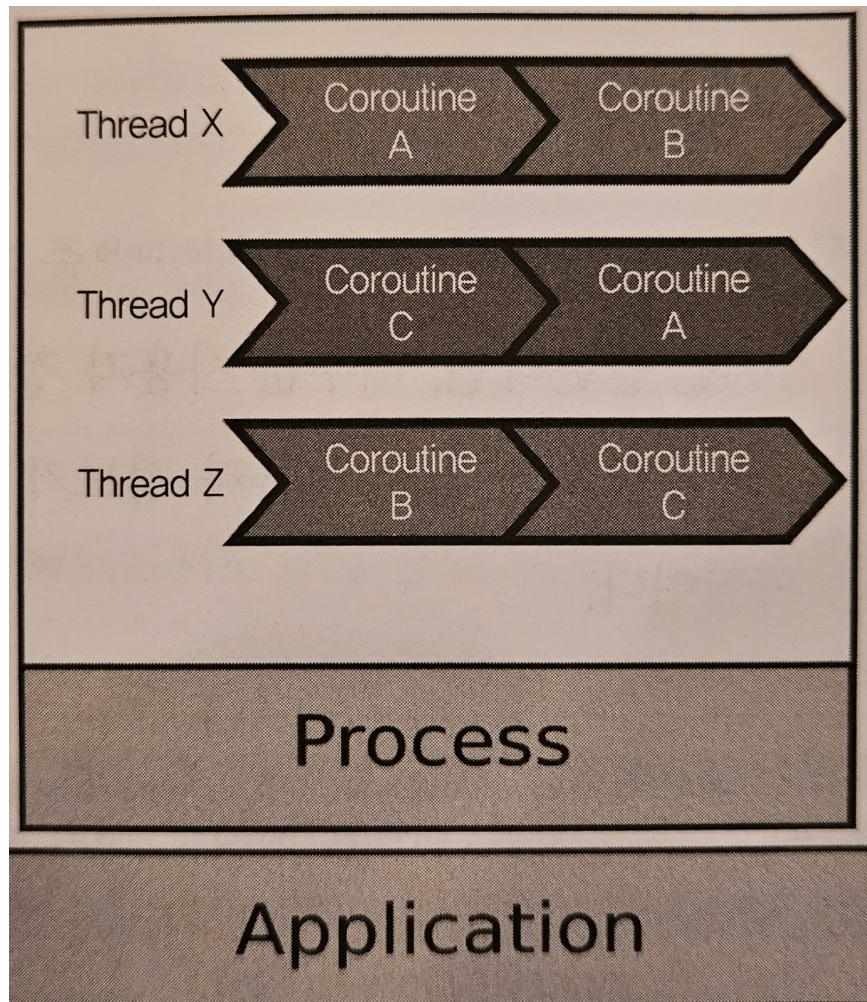
- 코루틴은 스레드 안에서 실행될 뿐, 종속된 관계가 아니기에 코루틴 일부를 특정 스레드에서 실행하고, 실행을 중지한 다음 나중에 다른 스레드에서 계속 실행하는 것이 가능함.

예시

```
suspend fun createCoroutines(amount: Int) {  
    val jobs = ArrayList<Job>()  
    for (i in 1..amount) {  
        jobs += launch {  
            println("started $i in ${Thread.currentThread().name}")  
            delay(1000)  
            println("finished $i in ${Thread.currentThread().name}")  
        }  
    }  
    jobs.forEach {  
        it.join()  
    }  
}
```

→ 다른 스레드에서 다시 시작하는 경우가 많음.

- 스레드는 한 번에 하나의 코루틴만 실행할 수 있기에, 프레임워크가 필요에 따라 코루틴을 스레드들 사이에 옮길 수 있다. 또 개발자가 코루틴을 실행할 스레드를 지정하거나, 코루틴을 해당 스레드에서만 실행되게 제한할지 여부를 정할 수 있음
 - (유연성 대박적)



동시성에 대해

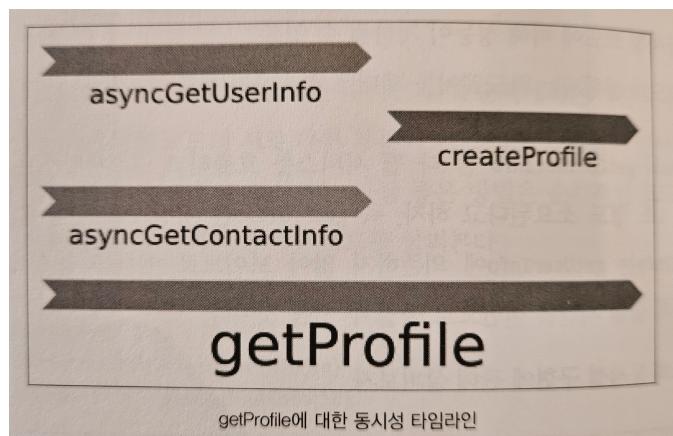
특정 입력이 들어오면 언제나 똑같은 과정을 거쳐서 항상 똑같은 결과를 내려놓아야 올바른 동시성코드. 그렇지만 실행 순서에서는 약간의 가변성을 허용함.

- 코드가 서로 독립성이 필요하다.

순차 코드의 문제점

1. 동시성 코드에 비해 성능이 저하될 수 있음
2. 코드가 실행되는 하드웨어를 제대로 활용 못할 수 있음

동시성 코드를 구현한다면?



- 2배로 시간이 걸릴 수도 있었던 UserInfo, ContactInfo 데이터를 동시에 수행함으로써 시간을 단축
- 하지만 두 정보중 어떤 정보가 먼저 도착할지 모름
- 그래도 두 정보가 모두 도착해야 `createProfile`이 수행됨

동시성이 까다로운 이유 = 순서에 관계없이 결과가 결정적이어야함을 보장해야함

동시성(concurrency)은 병렬성(parallelism)이 아니닷

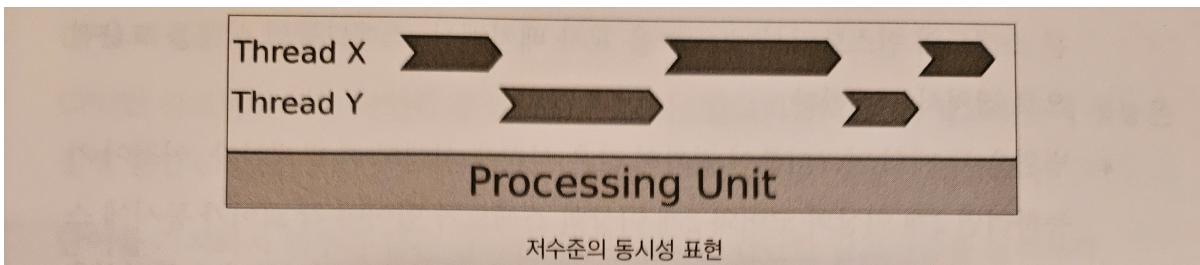
둘 다 두개의 코드가 동시에 실행된다는 점에서 둘 다 상당히 비슷한 점이 있긴 함

동시성과 병렬성의 차이점

같은 프로세스 안에서 서로 다른 명령 집합의 타임라인이 겹칠때 동시성이 발생한다는 점이다.

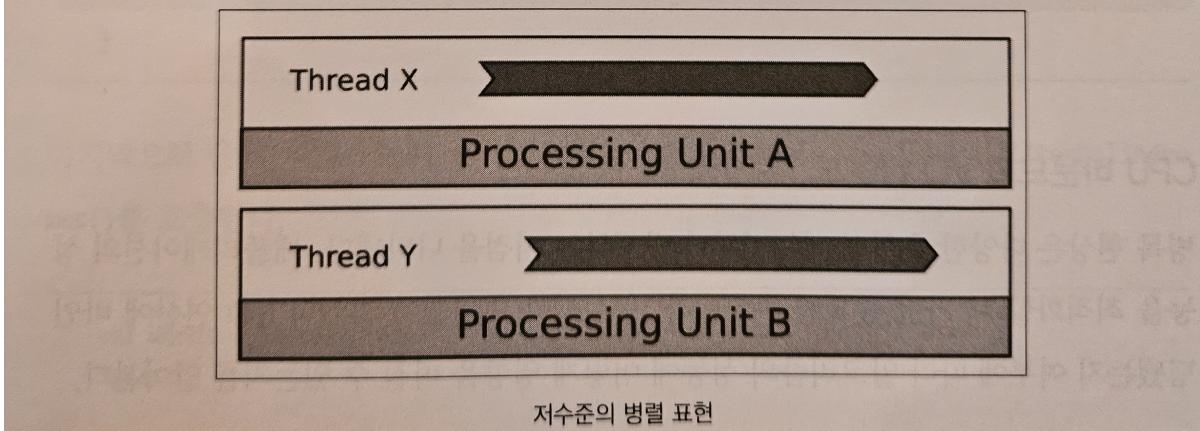
동시성은 정확히 같은 시점에 실행되는지 여부와 상관X. 하지만 병렬성은 상관이 있음

두 다이어그램 중 첫번째는 단일 코어, 두번째는 두개 코어라고 가정



반면에 병렬 실행parallel execution은 두 스레드가 정확히 같은 시점에 실행될 때만 발생한다. 두 개의 코어가 있는 컴퓨터에서 `getProfile()`이 실행되고 있는 경우 코어 하나는 `asyncGetUserInfo()`의 명령을 실행하고 다른 하나의 코어에서 `asyncGetContactInfo()`의 명령을 실행한다.

다음 다이어그램은 두 개의 프로세스 유닛을 사용해 각각 독립적인 스레드를 실행하는 동시성 코드를 병렬로 실행한 것이다. 이때 스레드 X와 Y의 타임라인이 겹칠 뿐만 아니라, 정확히 같은 시점에 실행되고 있다.



동시성

두 개 이상의 알고리즘의 실행 시간이 겹쳐질 때 발생함. 중첩이 발생하려면 두 개 이상의 실행 스레드가 필요함. 만약에 이 두 실행스레드가 단일코어에서 실행되면 병렬이 아니라 동시에 실행된다. 위쪽의 다이어그램처럼 코어가 인스트럭션을 교차배치해서 실행함

병렬성

두 개의 알고리즘이 정확히 같은 시점에 실행될 때 발생함. 2개 이상의 코어와 2개 이상의 스레드가 있어야 각 코어가 동시에 스레드의 인스트럭션을 실행할 수 있음

CPU 바운드와 I/O 바운드

동시성/병렬성이 CPU나 I/O 연산에 바인딩 되었는지 여부에 따라 성능에 영향이 갈 수 있음

CPU 바운드

- CPU만 완료하면 되는 작업을 중심으로 구현되는 알고리즘이 많음
- 알고리즘의 성능은 실행중인 CPU의 성능에 좌우되며, CPU만 업그레이드로 성능 향상됨

CPU 바운드 알고리즘에서 동시성과 병렬성

- 다중코어에서 병렬성을 활용하면 성능을 향상시킬 수 있음
- 단일 코어에서 동시성을 구현하면 성능이 저하되기도 함
 - context switching 비용이 더 많이 들기 때문.
 - isPalindrome()이라는 회문판별알고리즘을 1000개씩 끊은 3개의 스레드에서 3000개의 단어를 판별한다면.. 현재 스레드의 상태를 저장한 뒤 다음 스레드의 상태를 적재해야하기 때문에, 전체 프로세스 관점에서 오버헤드가 발생할 수 있음
 - 순차적 구현에서는 단일코어가 모든 작업을 수행해서 컨텍스트 스위칭 비용 발생 안해서 더 빨리 수행될 수 있다.
- 병렬 수행의 경우 코어의 개수에 비례해서 수행 시간이 줄여질 것이다.

CPU 바운드 알고리즘을 위해서는 현재 사용 중인 장치의 코어 수를 기준으로 적절한 스레드 수를 생성하도록 고려해야한다.

(CPU 바운드 알고리즘을 실행하기 위해 생성된 스레드 풀인 Kotlin의 CommonPool 활용가능)

I/O 바운드

- I/O 바운드는 입출력 장치에 의존하는 알고리즘
- 시스템에서 파일을 읽거나, 네트워크 통신등의 작업이 포함되어있는 알고리즘
- I/O 작업을 기준으로 성능에 대한 병목이 생겨서 최적화가 외부 시스템이나 장치에 의존적..
 - DB 같은 스토리지 성능.. 최적화.. 인터넷속도 등

I/O 바운드 알고리즘에서 동시성/병렬성

- 한 스레드가 입출력 대기 중에 다른 유용한 작업에서 타 스레드를 사용할 수 있도록 만든다
 - 병렬/단일코어에 상관없이 유사하게 수행될 것.
- I/O 작업은 늘 동시성으로 실행하는 편이 좋다.

reason why Concurrency is so hard..

레이스 컨디션(= 경합 조건)

코드를 동시성으로 작성했지만 순차적 코드처럼 동작할 것이라고 예상할때 발생
동시성 코드가 항상 특정한 순서로 실행될 것이라고 가정하면 안됨.

원자성 위반

원자성 작업: 작업이 사용하는 데이터를 간섭 없이 접근할 수 있는 것

단일 스레드 애플리케이션에서는 모든 코드가 순차 실행 \Rightarrow 모든 작업이 다 atomic 할 것이다.

동시에 수정될 수 있을 때 원자성이 위반되지 않게 보장되어야 한다.

교착 상태

동시성 코드가 올바르게 동기화 되려면 다른 스레드에서 작업이 완료되는 동안 실행을 일시 중단하거나 차단할 필요가 있음. 하지만 이 차단이 순환적 의존관계를 갖고 있다면 교착상태가 발생해 전체 앱이 다운되는 상황이 일어날 수도..

→ race condition과 자주 같이 발생한다.

라이브 락

교착상태와 비슷하지만, 앱이 다운되지는 않고, 상태가 계속 변하지만 정상 실행상태로 돌아 오지 못하는 상황이다.

코틀린에서의 동시성

NonBlocking

코틀린은 중단 가능한 연산(=Suspendable Computation)이라는 기능을 제공함

- 스레드 실행을 블로킹하지 않고, 실행만 잠시 중단하는 것이다.
 - 스레드 Y에서 작업이 끝나길 기다리며 스레드X를 블로킹하지 않고, 대기해야 하는 코드를 일시중단(suspend)시키고 그동안 스레드X를 다른 연산작업에 사용한다.

코틀린은 channel, actor, mutual exclusions(상호배제)와 같은 primitives를 제공해 스레드를 블록하지 않고도 통신 및 동기화를 통해 동시성 코드를 짤 수 있게 기능을 제공한다.

명시적인 선언

연산이 동시에 실행돼야 하는 시점을 명시적으로 만드는 것이 중요하다.

suspendable computations (=suspend 함수)들은 기본적으로 순차 실행된다.

```
fun main(args: Array<String>) = runBlocking {
    val time = measureMillis {
        val name = async { getName() }
        val lastName = async { getLastName() }

        println("Hello, ${name.await()} ${lastName.await()}")
    }
    println("Execution took $time ms")
}
```

async {} 함수를 호출해 두 함수를 동시에 호출했고, println시 await()를 호출해, 결과를 반환 받을때까지 main함수가 블록될 수 있게 runBlocking 을 선언함

가독성

코틀린은 관용구적인 동시성 코드를 허용함

```
suspend fun getProfile(id: Int) {
    val basicUserInfo = asyncGetUserInfo(id)
    val contactInfo = asyncGetContactInfo(id)

    createProfile(basicUserInfo.await(), contactInfo.await())
}
```

함수명이 `async`로 시작하거나 `Async`로 끝나도록 이름을 지으면 `async` 함수로 인식되어서 `await`를 걸 수 있음

`suspend` 메소드는 ⇒ 백그라운드 스레드에서 실행될 두 메소드를 호출하고 정보를 처리하기 전에 완료를 기다림.

기본형 활용

- `newSingleThreadContext()` 함수에 스레드 이름을 파라미터로 호출하면 스레드가 생성된다
 - 생성되면 필요한 만큼 많은 코루틴을 수행하는 데 사용할 수 있음
- `newFixedThreadPoolContext()` 함수에 스레드풀 크기와 이름 파라미터를 넣어 호출하면 스레드 풀이 생성된다
- `CommonPool`은 CPU 바운드 작업에 최적인 스레드 풀이다. 최적크기는 시스템 코어에서 1을 뺀 값임
- 코루틴을 다른 스레드로 이동시키는 역할은 런타임이 담당
- 채널, 뮤텍스, 스레드 한정 등의 기능이 코루틴 통신/동기화를 위해 제공됨

유연성

- Channel: 코루틴 간에 데이터를 안전하게 보내고 받는데 사용할 수 있는 파이프
- Worker Pool : 다양한 스레드에서 연산 집합 처리를 나눌 수 있는 코루틴 풀
- Actors: 현재 사용하는 상태로 채널과 코루틴을 감싼 래퍼로 여러 스레드에서 상태를 안전하게 수정하는 매커니즘 제공
- Mutex: 크리티컬 존 영역을 정의해 한번에 하나의 스레드만 실행할 수 있도록 하는 동기화 매커니즘. 이전 코루틴이 크리티컬 존을 빠져나올때까지 현재 액세스하려는 코루틴을 일시정지한다
- 스레드 한정(Thread Confinement): 코루틴의 실행을 제한해서 지정된 스레드에서만 실행하도록 하는 기능
- 생성자(반복자 및 시퀀스): 필요에 따라 정보를 생성할 수 있고 새로운 정보가 필요하지 않을때 중단될 수 있는 데이터 소스

코틀린 동시성 관련 개념과 용어

일시중단 연산(suspending computations)

= 스레드를 블로킹하지 않고 실행을 일시 중지할 수 있는 연산

- 일시 중단 연산을 통해 스레드를 다시 시작해야 할 때까지 해당 스레드를 다른 연산(코루틴)에서 사용할 수 있음

일시중단 함수

함수 형식의 일시 중단 연산. suspend 제어자가 붙은 function

람다 일시 중단

일시 중단 람다는 익명의 로컬 함수 (=일반적인 람다와 마찬가지)

- 일시 중단 람다는 다른 일시 중단 함수를 호출함으로써 자신의 실행을 중단할 수 있다는 점이 있다.

코루틴 디스패처

코루틴을 시작하거나 재개할 스레드를 결정하기 위해 코루틴 디스패처가 사용됨

모든 코루틴 디스패처는 CoroutineDispatcher 인터페이스를 구현해야 함

- DefaultDispatcher: 현재 CommonPool과 같음
- CommonPool: 공유된 백그라운드 스레드 풀에서 코루틴을 실행하고 다시 재개함. 기본 크기는 CPU바운드 작업에서 사용하기 적합함.
- Unconfined: 현재 스레드에서 코루틴을 시작하지만 어떠한 스레드에서도 코루틴이 다시 재개될 수 있. 디스패처에서는 스레드 정책을 사용하지 않음

Coroutines do not have a default dispatcher as it depends on the context in which they are launched. When you create a coroutine, you can specify the dispatcher to use, or if you don't specify one, the coroutine will inherit the dispatcher from the coroutine that launches it.

In Kotlin Coroutines, the `CommonPool` dispatcher was previously used as the default dispatcher, but it has been deprecated since Kotlin 1.3.0 and removed in Kotlin 1.5.0. Now, if you don't specify a dispatcher, the coroutine will use the ``Dispatchers.Default`` dispatcher, which is backed by a shared pool of threads and is suitable for CPU-bound tasks that do not consume blocking I/O.

Other dispatchers available in Kotlin Coroutines include:

- ``Dispatchers.IO``: for I/O-bound tasks that may block, such as file I/O and network I/O.
- ``Dispatchers.Main``: for executing tasks on the main thread of an Android application.
- ``Dispatchers.Unconfined``: for running a coroutine without any specific dispatcher, which can be useful for performance optimization in certain cases, but can also lead to unexpected behavior if used incorrectly.

이제는 coroutine dispatcher를 명시하지 않을 경우 `Dispatchers.Default` 디스패처를 사용하며, 이는 CPU bound task(I/O 블로킹을 소비하지 않는)에 적합한 스레드 공유풀에서 코루틴을 실행함.

디스패처와 함께 필요에 따라 스레드 정책을 정의할 수 있음

[스레드 풀]

- `newSingleThreadContext()`: 단일 스레드로 디스패처를 생성함. 여기에서 실행되는 코루틴은 항상 같은 스레드에서 시작되고 재개된다. 함수에 스레드 이름을 파라미터로 호출하면 스레드가 생성된다
 - 생성되면 필요한 만큼 많은 코루틴을 수행하는 데 사용할 수 있음
- `newFixedThreadPoolContext()`: 지정된 크기의 스레드 풀이 있는 디스패처를 만든다. 런타임은 디스패처에서 실행된 코루틴을 시작하고 재개할 스레드를 결정한다. 함수에 스레드풀 `크기` 와 `이름` 파라미터를 넣어 호출하면 스레드 풀이 생성된다

코루틴 빌더

| 일시 중단 람다를 받아 그것을 실행시키는 코루틴을 생성하는 함수다.

- `async()`: 결과가 예상되는 코루틴을 시작할때 사용. 결과/예외를 반환한다
(`=Deferred<T>`)
- `launch()`: 결과를 반환하지 않는 코루틴을 시작할때 사용. 본인 또는 자식 코루틴의 실행을 취소하기 위해 사용할 수 있는 Job을 반환함
- `runBlocking()`: 블로킹 코드를 suspendable code로 연결하기 위해 작성됨. 보통 main 메소드와 유닛테스트에서 사용함. `runBlocking()`은 코루틴의 실행이 끝날때까지 현재 스레드를 차단함