Chapter 3. CoroutineDispatcher

발표자 : 홍성덕

CoroutineDispatcher란?

- Dispatcher는 dispatch(보내다)와 -er의 합성어
- CoroutineDispatcher는 코루틴을 보내는 주체
  - 코루틴을 스레드로 보내 실행시키는 역할

CoroutineDispatcher의 동작 살펴보기

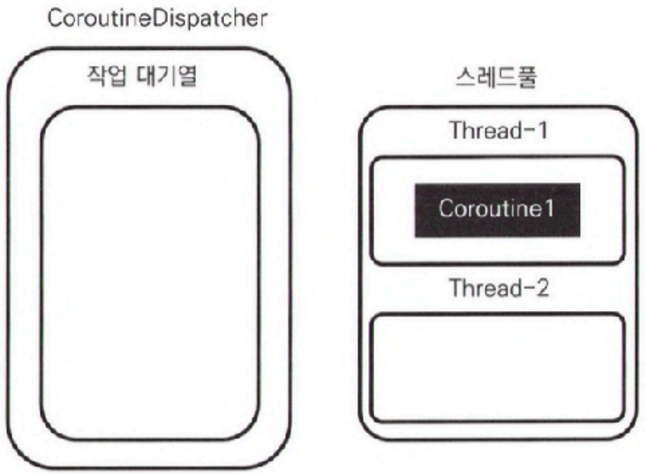


그림 3-1 CoroutineDispatcher 동작 살펴보기

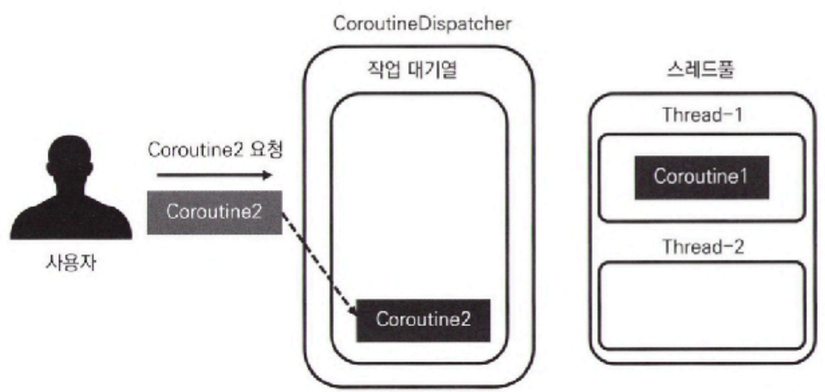


그림 3-2 실행 요청받은 코루틴을 적재하는 CoroutineDispatcher

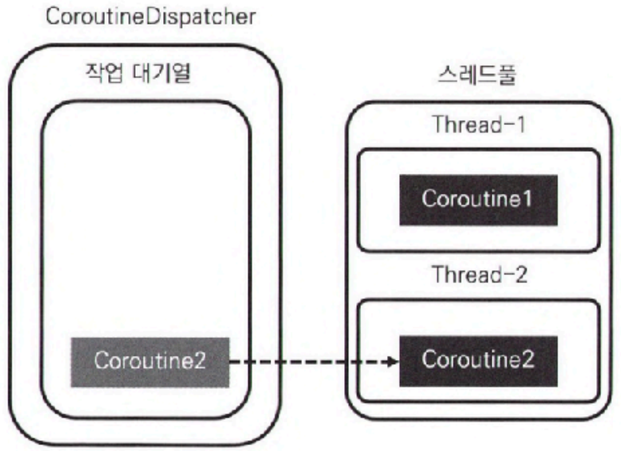


그림 3-3 스레드로 보내지는 코루틴

1. 두 개의 스레드로 구성된 스레드풀을 사용할 수 있는 CoroutineDispatcher가 존재하고, 하나의 스레드에서 이미 Coroutine1이 실행 중인 상황을 가정

2. 디스패처 객체에 코루틴의 실행이 요청되면, 디스패처는 코루틴을 작업 대기열에 적재
(작업 대기열 타입은 Queue)

3. 디스패처 객체는 사용 가능 스레드가 있는지 확인 후, 적재된 코루틴을 해당 스레드로 보내 실행

CoroutineDispatcher의 동작 살펴보기

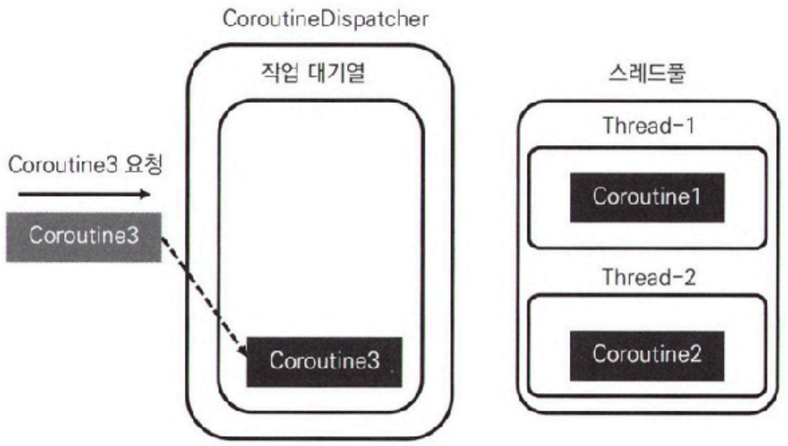


그림 3-4 추가로 요청된 Coroutine3 코루틴

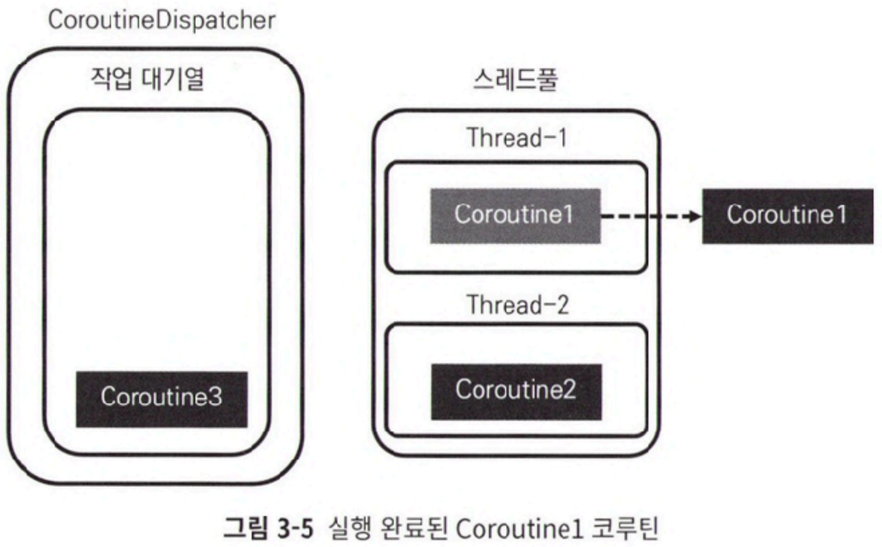


그림 3-5 실행 완료된 Coroutine1 코루틴

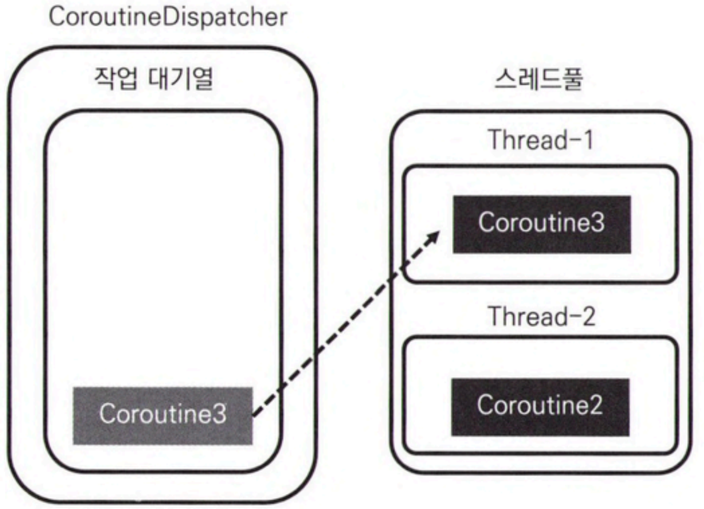


그림 3-6 쉬고 있는 스레드에 작업을 할당하는 CoroutineDispatcher

1. 스레드를 코루틴이 모두 점유하고 있는 상황에서 Coroutine3을 추가로 실행 요청
   → 디스패처는 코루틴을 작업 대기열에 적재하고 대기하도록 만든다.

2. Coroutine1이 완료되어 스레드가 사용가능한 상황

3. 대기하고 있던 Coroutine3이 사용 가능한 스레드에 보내져서 실행

제한된 디스패처 vs 무제한 디스패처

- CoroutineDispatcher에는 두 가지 종류가 존재
  - 제한된(Confined) 디스패처와 무제한(Unconfined) 디스패처
- 제한된 디스패처는 사용할 수 있는 스레드나 스레드 풀이 제한된 디스패처
- 무제한 디스패처는 사용할 수 있는 스레드나 스레드 풀이 제한되지 않은 디스패처
- 제한 : 스레드의 수를 제한한다는 의미가 아니고, 사용할 수 있는 스레드를 특정 스레드 혹은 특정 스레드 풀로 제한한다는 의미
  → 무제한 디스패처는 실행되는 스레드가 매번 달라질 수 있고 특정 스레드로 제한되어 있지 않다.

제한된 디스패처 생성

- 단일 스레드 디스패처 (newSingleThreadContext( ) 를 통해 생성)

```
fun main() = runBlocking<Unit> { this: CoroutineScope
    val dispatcher = newSingleThreadContext(name = "SingleThread")
    launch(context = dispatcher) { this: CoroutineScope
        println("[${Thread.currentThread().name}] 실행")
    }
}
/*
// 결과:
[SingleThread @coroutine#2] 실행
*/
```

제한된 디스패처 생성

- 멀티 스레드 디스패처 (newFixedThreadPoolContext( ) 를 통해 생성)

```kotlin
fun main() = runBlocking<Unit> { this: CoroutineScope
    val multiThreadDispatcher = newFixedThreadPoolContext(
        nThreads = 2,
        name = "MultiThread"
    )
    launch(context = multiThreadDispatcher) { this: CoroutineScope
        println("[${Thread.currentThread().name}] 실행")
    }
    launch(context = multiThreadDispatcher) { this: CoroutineScope
        println("[${Thread.currentThread().name}] 실행")
    }
}
/*
// 결과:
[MultiThread-1 @coroutine#2] 실행
[MultiThread-2 @coroutine#3] 실행
*/
```

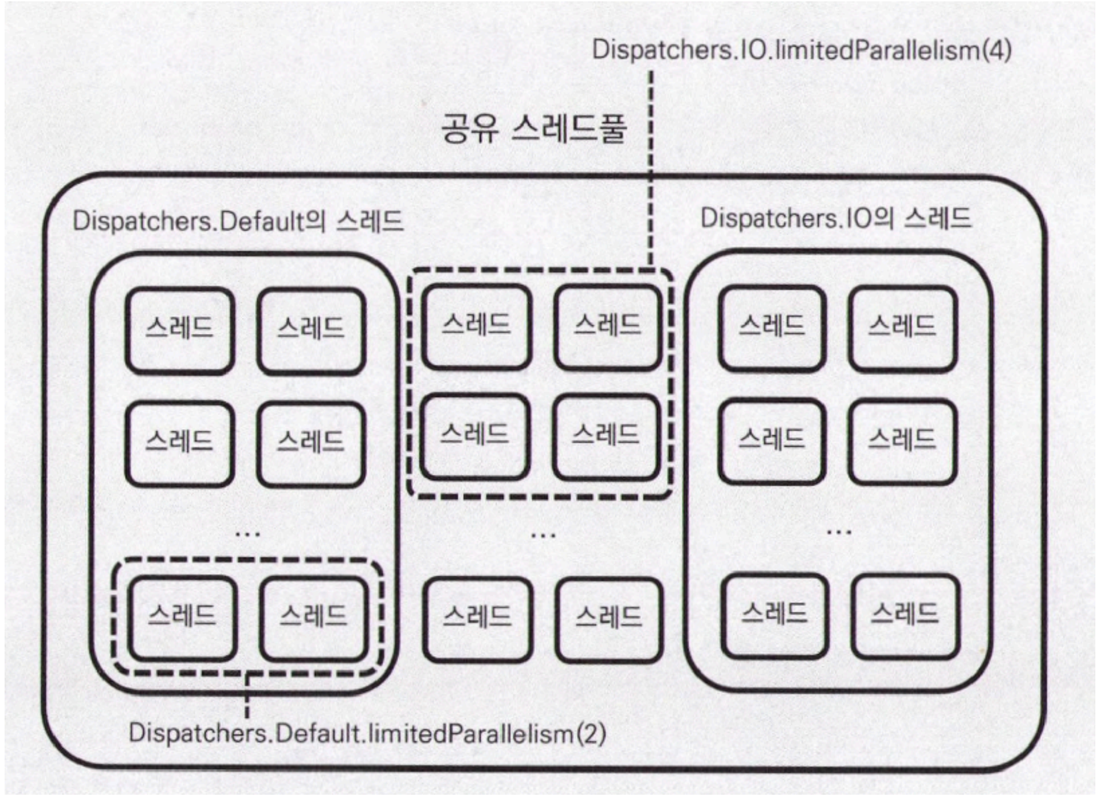
미리 정의된 CoroutineDispatcher

- newSingleThreadContext( ), newFixedThreadPoolContext( ) 함수를 사용하여 CoroutineDispatcher 객체를 만드는 것을 권장하지 않는다.
- 특정 디스패처 객체에서만 사용되는 스레드풀이 생성되어 비효율적
- 여러 개발자가 함께 개발할 경우 특정 용도의 디스패처 객체가 존재하고 있다는 것을 몰라서 다른 디스패처 객체를 만들 수 있다 : 리소스 낭비
- 그래서 코루틴 라이브러리는 이러한 일을 방지하고자 미리 정의된 디스패처 목록을 제공

미리 정의된 CoroutineDispatcher

- Dipsatchers.IO : 네트워크 요청이나 파일 입출력 등의 I/O 작업을 위한 CoroutineDispatcher
- Dispatchers.Default : CPU를 많이 사용하는 연산 작업을 위한 CoroutineDispatcher
- Dispatchers.Main : 메인 스레드를 사용하기 위한 CoroutineDispatcher
- Dispatchers.Main.immediate : 작업이 이미 메인스레드에 존재한다면 디스패치 과정을 거치지 않고 바로 실행

공유 스레드풀을 사용하는 Dispatchers.IO와 Dispatchers.Default



- 코루틴 라이브러리는 스레드의 생성과 관리를 효율적으로 할 수 있도록 애플리케이션 레벨의 공유 스레드풀을 제공
- Dispatchers.IO와 Dispatchers.Default는 코루틴 라이브러리의 공유 스레드풀을 사용
- 공유 스레드풀에서는 스레드를 무제한으로 생성 가능하고, 코루틴 라이브러리는 공유 스레드풀에 스레드를 생성하고 사용할 수 있도록 하는 API를 제공
- 스레드 풀 내에서 Dispatchers.IO와 Dispatchers.Default가 사용하는 스레드는 구분된다.

limitedParallelism 사용해서 스레드 사용 제한

```kotlin
fun main() = runBlocking<Unit> { this: CoroutineScope
    launch(Dispatchers.Default.limitedParallelism( parallelism: 2)){ this: CoroutineScope
        repeat( times: 10) { it: Int
            launch { this: CoroutineScope
                println("[${Thread.currentThread().name}] 코루틴 실행")
            }
        }
    }
}
/*
// 결과:
[DefaultDispatcher-worker-2 @coroutine#3] 코루틴 실행
[DefaultDispatcher-worker-1 @coroutine#4] 코루틴 실행
[DefaultDispatcher-worker-2 @coroutine#5] 코루틴 실행
...
[DefaultDispatcher-worker-1 @coroutine#10] 코루틴 실행
[DefaultDispatcher-worker-2 @coroutine#11] 코루틴 실행
[DefaultDispatcher-worker-2 @coroutine#12] 코루틴 실행
*/
```

- Dispatchers.Default를 사용해서 무겁고 오래 걸리는 연산을 처리하면 특정 연산을 위해서 Dispatchers.Default의 모든 스레드가 사용될 수 있다.
- 이를 방지하기 위해 Dispatchers.Default의 일부 스레드만 사용하는 limitedParallelism 함수를 지원

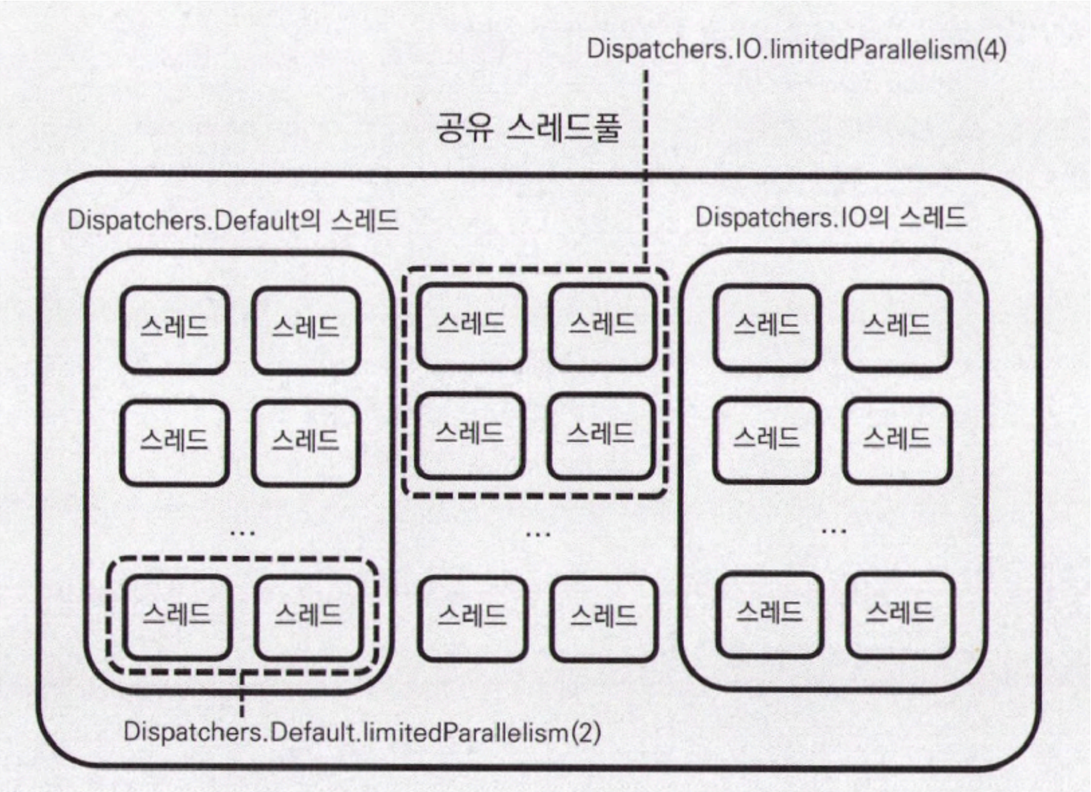
Dispatchers.IO의 limitedParallelism

```kotlin
fun main() = runBlocking<Unit> { this: CoroutineScope
    launch(Dispatchers.IO.limitedParallelism( parallelism: 100)) { this: CoroutineScope
        repeat( times: 200) { it: Int
            launch { this: CoroutineScope
                Thread.sleep( millis: 1000L)
                println("[${Thread.currentThread().name}] 코루틴 실행")
            }
        }
    }
}
/*
// 결과: 사용된 스레드의 순서는 매번 다르게 나온다.
[DefaultDispatcher-worker-60] 코루틴 실행
[DefaultDispatcher-worker-74] 코루틴 실행
[DefaultDispatcher-worker-68] 코루틴 실행
...
[DefaultDispatcher-worker-85] 코루틴 실행
[DefaultDispatcher-worker-49] 코루틴 실행
[DefaultDispatcher-worker-98] 코루틴 실행
*/
```

- Dispatchers.IO의 limitedParallelism 함수는 공유 스레드 풀의 스레드로 구성된 새로운 스레드 풀을 만들어내며, 스레드의 수를 제한 없이 만들어낼 수 있다.

Dispatchers.IO의 limitedParallelism

```kotlin
fun main() = runBlocking<Unit> { this: CoroutineScope
    launch(Dispatchers.IO.limitedParallelism( parallelism: 100)) { this: CoroutineScope
        repeat( times: 200) { it: Int
            launch { this: CoroutineScope
                Thread.sleep( millis: 1000L)
                println("[${Thread.currentThread().name}] 코루틴 실행")
            }
        }
    }
}
/*
// 결과: 사용된 스레드의 순서는 매번 다르게 나온다.
[DefaultDispatcher-worker-60] 코루틴 실행
[DefaultDispatcher-worker-74] 코루틴 실행
[DefaultDispatcher-worker-68] 코루틴 실행
...
[DefaultDispatcher-worker-85] 코루틴 실행
[DefaultDispatcher-worker-49] 코루틴 실행
[DefaultDispatcher-worker-98] 코루틴 실행
*/
```



- 공유 스레드풀 상에서 Dispatchers.IO나 Dispatchers.Default와 관계 없는 스레드로 구성된 스레드풀을 만들어낸다.