Chapter 4. 코루틴 빌더와 Job

발표자 : 홍성덕

코루틴 빌더와 Job

- 코루틴을 생성하는 데 사용하는 함수가 코루틴 빌더 함수 (ex: runBlocking, launch)
- 코루틴 빌더 함수는 코루틴을 생성하고 코루틴을 추상화한 Job 객체를 생성
- 코루틴 빌더 함수를 통해 반환된 Job 객체는 코루틴의 상태를 추적하고 제어하는 데 사용

```
fun main() = runBlocking<Unit> { this: CoroutineScope
    val job: Job = launch(Dispatchers.IO) { this: CoroutineScope // Job 객체 반환
        println("[${Thread.currentThread().name}] 실행")
    }
}
```

join을 사용한 코루틴 순차 처리

```kotlin
fun main() = runBlocking<Unit> { this: CoroutineScope
    val updateTokenJob = launch(Dispatchers.IO) { this: CoroutineScope
        println("[${Thread.currentThread().name}] 토큰 업데이트 시작")
        delay( timeMillis: 100L)
        println("[${Thread.currentThread().name}] 토큰 업데이트 완료")
    }
    val networkCallJob = launch(Dispatchers.IO) { this: CoroutineScope
        println("[${Thread.currentThread().name}] 네트워크 요청")
    }
}
/*
// 결과:
[DefaultDispatcher-worker-1 @coroutine#2] 토큰 업데이트 시작
[DefaultDispatcher-worker-3 @coroutine#3] 네트워크 요청
[DefaultDispatcher-worker-1 @coroutine#2] 토큰 업데이트 완료
*/
```

- 인증 토큰을 업데이트 한 후 네트워크 요청이 실행되어야 정상적으로 처리되는 상황을 가정
- 순차 처리가 되지 않는다면 토큰 업데이트를 완료하기 전에 네트워크를 요청하는 문제가 발생

join을 사용한 코루틴 순차 처리

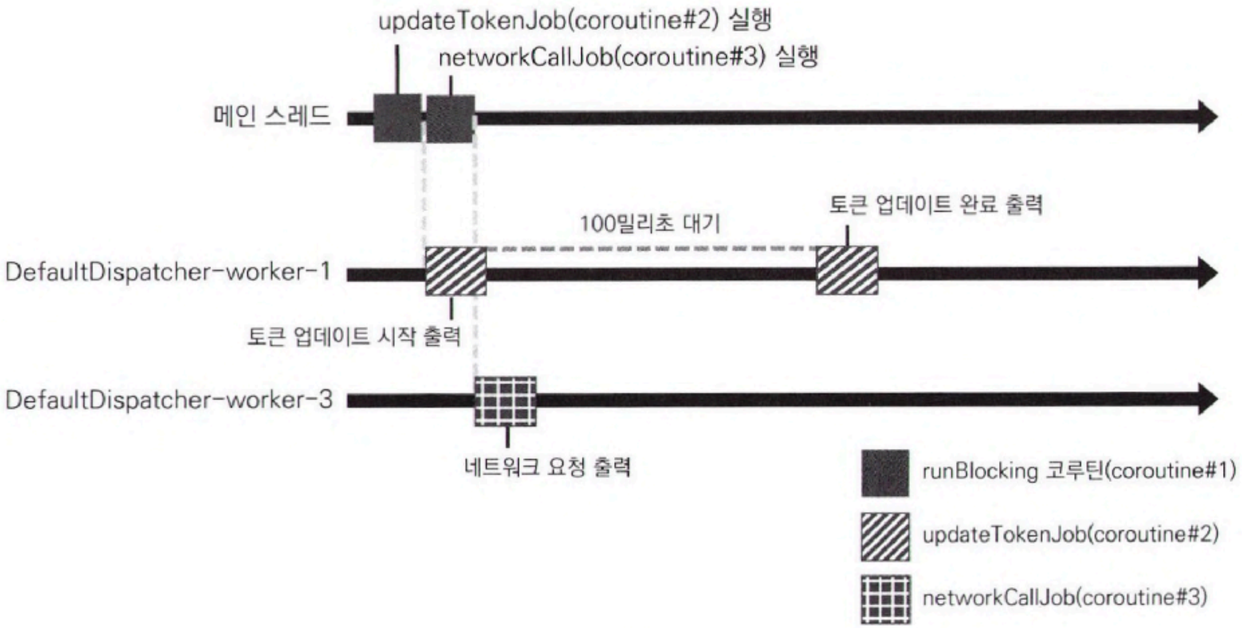

그림 4-1 순차 처리 안 된 코루틴

- 인증 토큰을 업데이트 한 후 네트워크 요청이 실행되어야 정상적으로 처리되는 상황을 가정
- 순차 처리가 되지 않는다면 토큰 업데이트를 완료하기 전에 네트워크를 요청하는 문제가 발생

join을 사용한 코루틴 순차 처리

```kotlin
fun main() = runBlocking<Unit> { this: CoroutineScope
    val updateTokenJob = launch(Dispatchers.IO) { this: CoroutineScope
        println("[${Thread.currentThread().name}] 토큰 업데이트 시작")
        delay( timeMillis: 100L)
        println("[${Thread.currentThread().name}] 토큰 업데이트 완료")
    }
    updateTokenJob.join() // updateTokenJob이 완료될 때까지 일시 중단
    val networkCallJob = launch(Dispatchers.IO) { this: CoroutineScope
        println("[${Thread.currentThread().name}] 네트워크 요청")
    }
}
/*
// 결과:
[DefaultDispatcher-worker-1 @coroutine#2] 토큰 업데이트 시작
[DefaultDispatcher-worker-1 @coroutine#2] 토큰 업데이트 완료
[DefaultDispatcher-worker-1 @coroutine#3] 네트워크 요청
*/
```

- Job 객체의 join 함수를 사용하면 코루틴 간에 순차 처리가 가능
- join의 대상이 된 코루틴 (updateTokenJob)의 작업이 완료될 때까지 join을 호출한 코루틴 (runBlocking 코루틴)이 일시 중단
- updateTokenJob 내부의 코드가 모두 실행되면 runBlocking 코루틴이 재개되어 networkCallJob을 실행

join을 사용한 코루틴 순차 처리

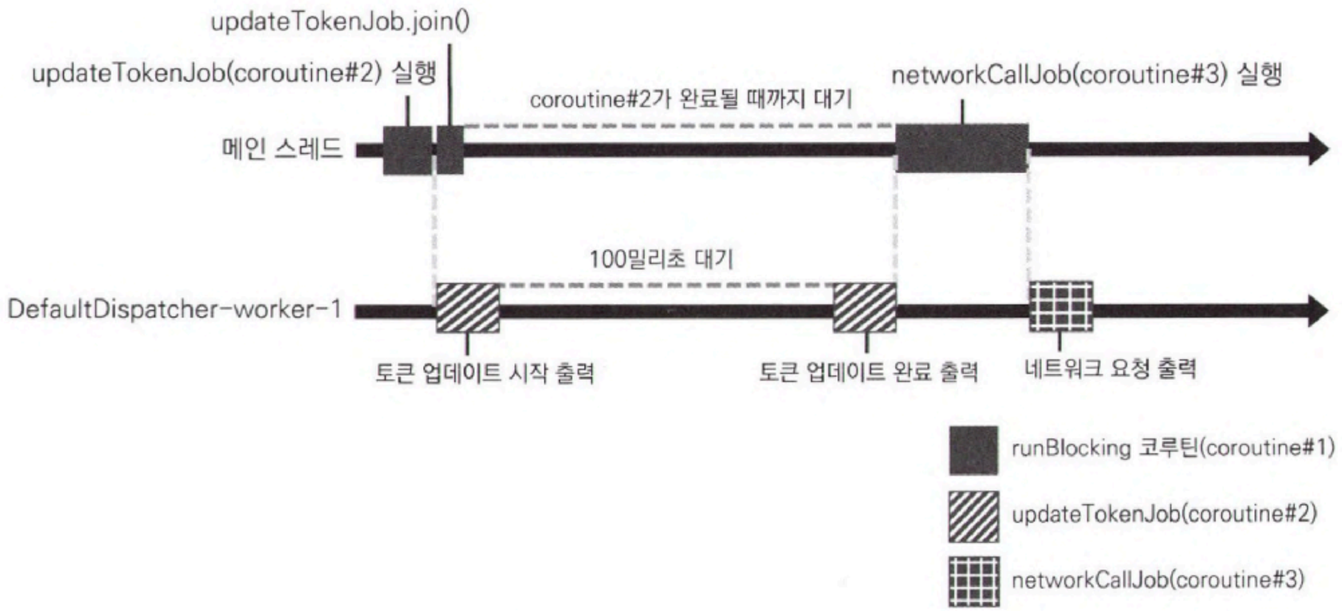

그림 4-2 순차 처리된 코루틴

- Job 객체의 join 함수를 사용하면 코루틴 간에 순차 처리가 가능
- join의 대상이 된 코루틴 (updateTokenJob)의 작업이 완료될 때까지 join을 호출한 코루틴 (runBlocking 코루틴)이 일시 중단
- updateTokenJob 내부의 코드가 모두 실행되면 runBlocking 코루틴이 재개되어 networkCallJob을 실행

joinAll을 사용한 코루틴 순차 처리

```kotlin
fun main() = runBlocking<Unit> { this: CoroutineScope
    val convertImageJob1: Job = launch(Dispatchers.Default) { this: CoroutineScope
        Thread.sleep( millis: 1000L) // 이미지 변환 작업 실행 시간
        println("[${Thread.currentThread().name}] 이미지1 변환 완료")
    }
    val convertImageJob2: Job = launch(Dispatchers.Default) { this: CoroutineScope
        Thread.sleep( millis: 1000L) // 이미지 변환 작업 실행 시간
        println("[${Thread.currentThread().name}] 이미지2 변환 완료")
    }

    joinAll(convertImageJob1, convertImageJob2) // 이미지1과 이미지2가 변환될 때까지 대기

    val uploadImageJob: Job = launch(Dispatchers.IO) { this: CoroutineScope
        println("[${Thread.currentThread().name}] 이미지1,2 업로드")
    }
}
/*
// 결과:
[DefaultDispatcher-worker-1 @coroutine#2] 이미지1 변환 완료
[DefaultDispatcher-worker-2 @coroutine#3] 이미지2 변환 완료
[DefaultDispatcher-worker-1 @coroutine#4] 이미지1,2 업로드
*/
```

- 실제 개발 시에는 서로 독립적인 여러 코루틴을 병렬로 실행한 후, 모두 끝날 때까지 기다리는 것이 효율적
- joinAll의 대상이 된 코루틴들 (convertImageJob1, convertImageJob2)의 실행이 모두 끝날 때까지 호출부의 코루틴 (runBlocking 코루틴)을 일시 중단
- joinAll은 내부적으로 전달받은 job 객체들을 forEach 문을 통해 join 함수를 호출한다.

joinAll을 사용한 코루틴 순차 처리

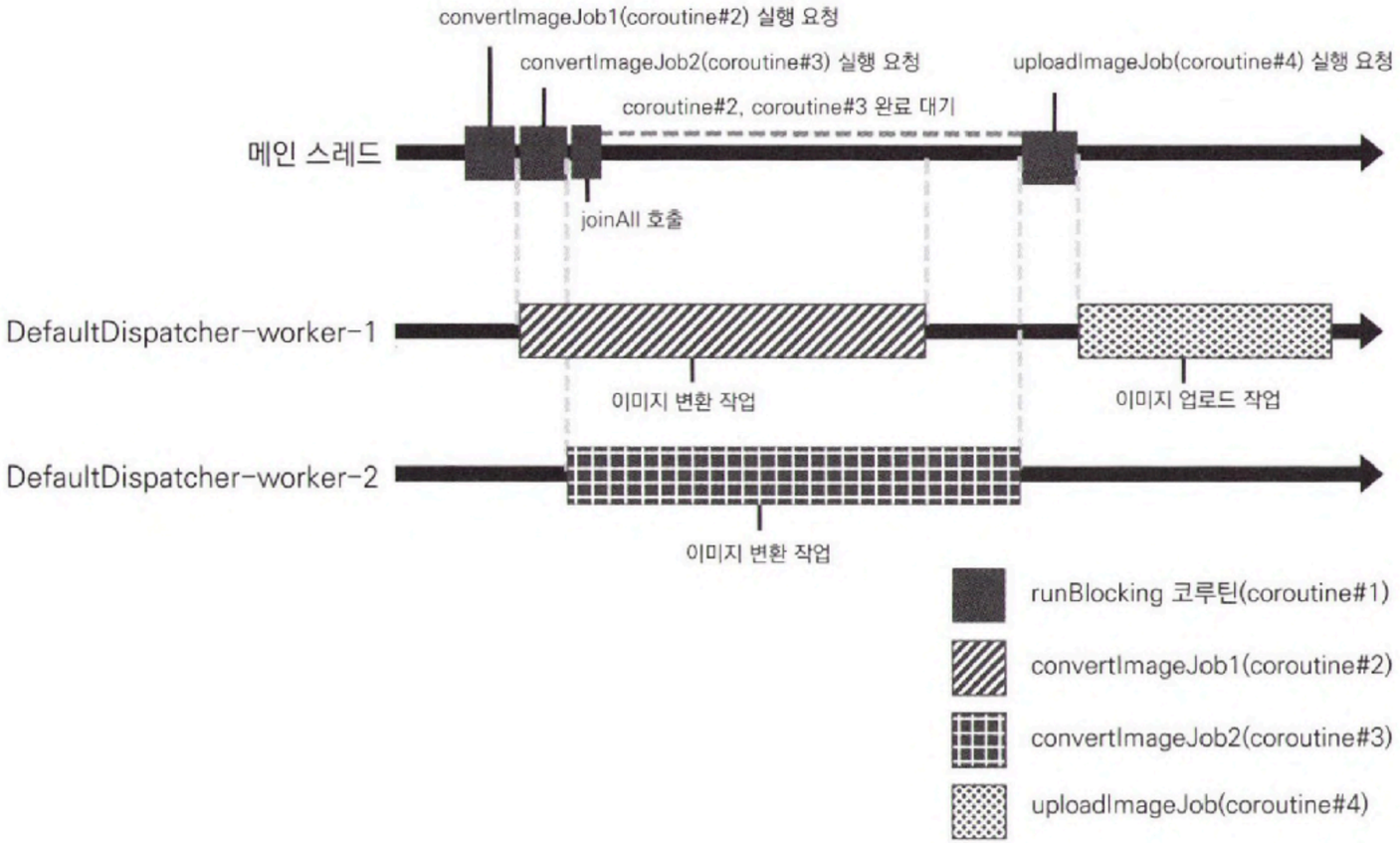

그림 4-4 joinAll을 사용한 순차 처리

CoroutineStart.LAZY 사용해 코루틴 지연 시작

```kotlin
fun main() = runBlocking<Unit> { this: CoroutineScope
    val startTime = System.currentTimeMillis()
    val lazyJob: Job = launch(start = CoroutineStart.LAZY) { this: CoroutineScope
        println("[${Thread.currentThread().name}][${getElapsedTime(startTime)}] 지연 실행")
    }
    delay( timeMillis: 1000L) // 1초간 대기
    lazyJob.start() // 코루틴 실행
}
/*
// 결과:
[main @coroutine#2][지난 시간: 1014ms] 지연 실행
*/
```

- launch 함수 호출 시 생성되는 코루틴은 실행 가능한 스레드가 있다면 곧바로 실행
- CoroutineStart.LAZY를 전달하면 코루틴은 생성 후 대기 상태에 놓이고, 실행을 요청해야 비로소 시작된다.

코루틴 취소하기

```
fun main() = runBlocking<Unit> { this: CoroutineScope
    val startTime = System.currentTimeMillis()
    val longJob: Job = launch(Dispatchers.Default) { this: CoroutineScope
        repeat( times: 10) { repeatTime →
            delay( timeMillis: 1000L) // 1000밀리초 대기
            println("[${getElapsedTime(startTime)}] 반복횟수 ${repeatTime}")
        }
    }
    delay( timeMillis: 3500L) // 3500밀리초(3.5초)간 대기
    longJob.cancel() // 코루틴 취소
}
/*
// 결과:
[지난 시간: 1016ms] 반복횟수 0
[지난 시간: 2021ms] 반복횟수 1
[지난 시간: 3027ms] 반복횟수 2
*/
```

- cancel( )
- cancel( ) 함수를 호출한다고 해서 코루틴은 즉시 취소되는 것이 아니다.
- 취소가 요청된 상태일 뿐이고 미래의 어느 시점에 취소가 요청되었는지 체크 후 취소

코루틴 취소하기

```
fun main() = runBlocking<Unit> { this: CoroutineScope
    val longJob: Job = launch(Dispatchers.Default) { this: CoroutineScope
        // 작업 실행
    }
    longJob.cancelAndJoin() // longJob이 취소될 때까지 runBlocking 코루틴 일시 중단
    executeAfterJobCancelled()
}

fun executeAfterJobCancelled() {
    // 작업 실행
}
```

- cancelAndJoin( ) 을 호출하여 코루틴 취소가 완료될 때까지 호출부의 코루틴을 일시중단
- 취소가 완료된 후 runBlocking 코루틴이 재개되어 이후 작업 실행

코루틴의 취소 확인

- 코루틴이 취소를 확인하는 시점은 일시 중단 지점이나 코루틴이 실행을 대기하는 시점
- delay, yield 같은 일시중단 함수를 사용한 취소 확인
- CoroutineScope.isActive를 사용한 취소 확인

코루틴의 상태

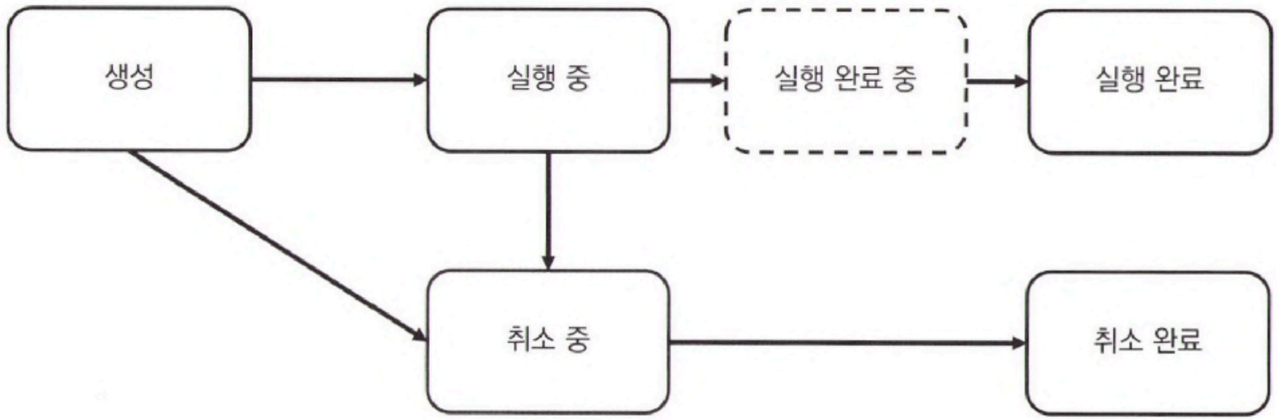

그림 4-5 코루틴의 상태

- 생성 (New) : 코루틴을 생성하면 기본적으로 생성 상태에 놓이고 자동으로 실행 중 상태로 넘어간다. 만약 자동으로 실행 중 상태로 변경하지 않으려면 CoroutineStart.LAZY 전달
- 실행 중 (Active) : 실행된 후 일시 중단되었을 때도 실행 중 상태
- 실행 완료 (Completed) : 모든 코드가 실행 완료된 상태
- 취소 중 (Cancelling) : 코루틴에 취소가 요청된 상태
- 취소 완료 (Cancelled) : 코루틴의 취소 확인 시점에 취소가 확인된 경우 취소 완료 상태가 된다.

Job의 상태 변수

- Job 객체는 코루틴이 어떤 상태에 있는지 나타내는 상태 변수들을 외부로 공개
- isActive : 코루틴이 활성화되어 있는지의 여부
- isCancelled : 코루틴이 취소 요청됐는지의 여부
- isCompleted : 코루틴 실행이 완료되었는지의 여부

Job의 상태 변수

코루틴 상태	isActive	isCancelled	isCompleted
생성(New)	false	false	false
실행 중(Active)	true	false	false
실행 완료(Completed)	false	false	true
취소 중(Cancelling)	false	true	false
취소 완료(Cancelled)	false	true	true

표 4-1 코루틴 상태별 Job 상태표

- 취소 중 상태는 취소가 요청된 상태일 뿐, 취소가 완료된 상태는 아니기 때문에 isCancelled가 true, isCompleted가 false
- 취소 완료 상태는 isCancelled와 isCompleted가 둘다 true