Chapter15. 구조화된 동시성

15장에서 다루는 내용

- 구조화된 동시성을 통해 코루틴 간의 계층을 설정하는 방법
- 구조화된 동시성을 통해 코드 실행과 취소를 세밀하게 제어하고, 코루틴 계층 전반에 걸쳐 자동으로 취소를 전파하는 방법
- 코루틴 컨텍스트와 구조화된 동시성 간의 관계
- 취소 시에도 올바르게 동작하는 코드를 작성하는 방법

실제 애플리케이션에서 많은 코루틴을 관리하게 될 가능성이 크다. 여러 동시 작업을 처리할 때 큰 도전은 실행 중인 개별 작업을 추적하고, 더 이상 필요하지 않을 때 이를 취소하며, 오류를 제대로 처리하는 것이다.

코루틴을 추적하지 않으면 리소스 누수와 불필요한 작업을 하게 될 위험이 있다. 예를 들어 사용자가 네트워크 리소스를 요청한 후 즉시 다른 화면으로 이동한다고 가정해보자. 네트워크 요청과 수신된 정보의 후처리를 담당하는 코루틴을 추적할 방법이 없다면 결과가 결국 버려지더라도 모든 코루틴이 끝날 때까지 실행되게 놔두는 수밖에 없다.

다행히도 구조화된 동시성, 즉 애플리케이션 안에서 코루틴과 그 생애주기의 계층을 관리하고 추적할 수 있는 기능이 코틀린 코루틴의 핵심에 내장돼있다. 구조화된 동시성은 수동으로 시작된 각 코루틴을 일일이 추적하지 않아도 기본적으로 작동한다. 애플리케이션 전반에 구조화된 동시성을 사용하면 계획보다 오래 실행되거나 잊혀진 코루틴은 발생하지 않는다.

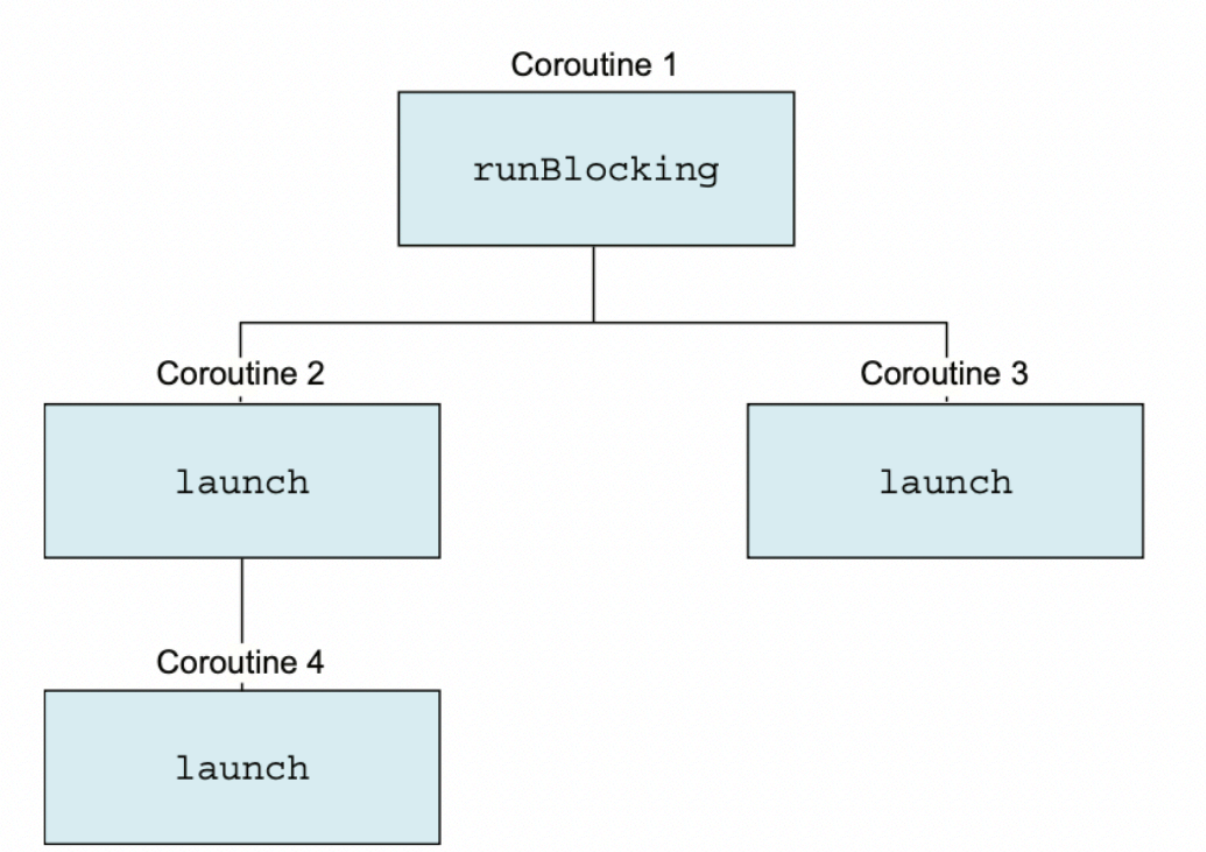
코루틴 스코프가 코루틴 간의 구조를 확립한다.

- 구조화된 동시성을 통해 각 코루틴은 코루틴 스코프에 속하게 된다.
- 코루틴 스코프는 코루틴 간의 부모-자식 관계를 확립하는데 도움을 준다.
    - launch와 async 코루틴 빌더 함수들은 CoroutineScope 인터페이스의 확장 함수
    - 즉, 다른 코루틴 빌더의 본문에서 launch나 async를 사용해 새로운 코루틴을 만들면 자동으로 해당 코루틴의 자식이 된다.

```
fun main() {
    runBlocking { // this: CoroutineScope
        launch { // this: CoroutineScope, runBlocking 코루틴의 자식
            delay(1.seconds)
            launch {
                delay(250.milliseconds)
                log("Grandchild done")
            }
            log("Child 1 done!")
        }
        launch {
            delay(500.milliseconds)
            log("Child 2 done!")
        }
        log("Parent done!")
```

```
    }
}

// 29 [main @coroutine#1] Parent done!
// 539 [main @coroutine#3] Child 2 done!
// 1036 [main @coroutine#2] Child 1 done!
// 1293 [main @coroutine#4] Grandchild done
```



- runBlocking 함수 본문이 거의 즉시 실행을 마쳤음에도 모든 자식 코루틴이 완료될 때까지 프로그램이 종료되지 않는 것을 알 수 있다.
    - 이는 구조화된 동시성 덕분
- 코루틴 간에는 부모-자식 관계(Job 객체들 간의 관계)가 있으므로 runBlocking은 여전히 어떤 자식 코루틴이 작업 중인지 알고, 모든 작업이 완료될 때까지 기다린다.
    - 구조화된 동시성 덕분에 코루틴은 계층 구조 안에 존재한다.
    - 명시적으로 지정하지 않았음에도 각 코루틴은 자식이나 부모를 알고 있다.

- 실행한 코루틴이나 그 자손을 수동으로 추적할 필요가 없으며, 수동으로 await를 호출할 필요도 없다.
    - 부모 코루틴이 취소되면 자식 코루틴도 자동으로 취소되는 기능을 가능하게 한다.
    - 이 기능은 예외 처리에서도 큰 도움이 된다. (18장)

코루틴 스코프 생성: coroutineScope 함수

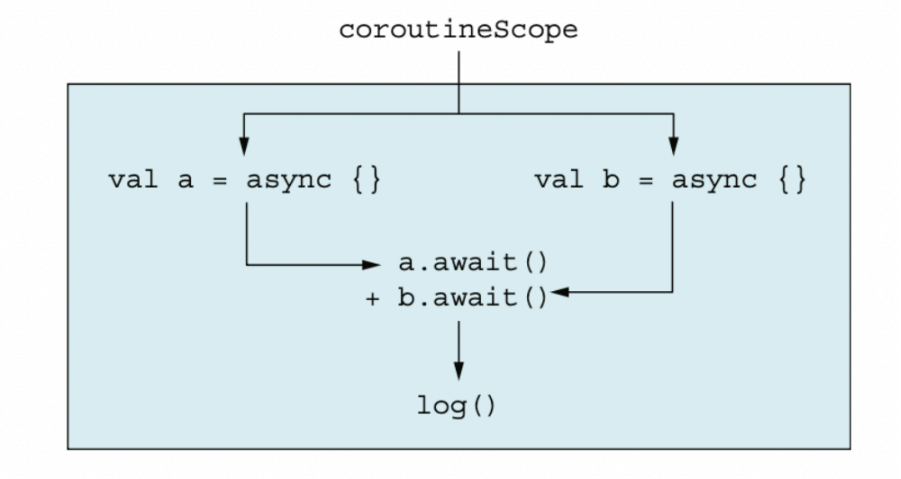
- 코루틴 빌더를 사용해 새로운 코루틴을 만들면 이 코루틴은 자체적인 CoroutineScope를 생성한다.
- 하지만 새로운 코루틴을 만들지 않고도 코루틴 스코프를 그룹화할 수 있는데, 이때 coroutineScope 함수를 사용할 수 있다.
    - 해당 함수는 일시 중단 함수로, 새로운 코루틴 스코프를 생성하고 해당 영역 안의 모든 자식 코루틴이 완료될 때까지 기다린다.
- coroutineScope 함수의 전형적인 사용 사례는 동시적 작업 분해, 즉 여러 코루틴을 활용해 계산을 수행하는 것이다.

```
suspend fun generateValue(): Int {
    delay(500.milliseconds)
    return Random.nextInt(0, 10)
}

suspend fun computeSum() { // 일시 중단 함수
    log("Computing a sum...")
    val sum = coroutineScope { // coroutineScope 함수를 사용해 새로운 코루틴 스코
        val a = async { generateValue() }
        val b = async { generateValue() }
        a.await() + b.await() // coroutineScope는 결과를 돌려주기 전에 모든 자식 코루
    }
    log("Sum is $sum")
}

fun main() = runBlocking {
    computeSum()
}
```

```
// 0 [main @coroutine#1] Computing a sum...
// 432 [main @coroutine#1] Sum is 10
```



- coroutineScope를 동시적 작업 분해에 사용한다.
- 두 코루틴이 병렬로 실행되며, 그 결과를 사용해 값을 계산해 로그에 남긴다.

코루틴 스코프를 컴포넌트와 연관시키기: CoroutineScope

- coroutineScope 함수가 작업을 분해하는 데 사용되는 반면 구체적 생명주기를 정의하고, 동시 처리나 코루틴의 시작과 종료를 관리하는 클래스를 만들고 싶을 때도 있다.
- 이런 시나리오에서는 CoroutineScope 생성자 함수를 사용해 새로운 독자적인 코루틴 스코프를 생성할 수 있다.
- coroutineScope와는 달리 이 함수는 실행을 일시 중단하지 않으며 , 단순히 새로운 코루틴을 시작할 때 쓸수 있는 새로운 코루틴 스코프를 생성하기만 한다.
- CoroutineScope는 하나의 파라미터를 받는다.
  - 해당 코루틴 스코프와 연관된 코루틴 컨텍스트다.
  - 예를 들어 해당 범위에서 시작된 코루틴이 사용할 디스패처를 지정할 수 있다.
- 기본적으로 CoroutineScope를 디스패처만으로 호출하면 새로운 Job이 자동으로 생성된다.
  - 하지만 대부분의 실무에서는 CoroutineScope와 함께 SupervisorJob을 사용하는 것이 좋다.

- SupervisorJob은 동일한 영역과 관련된 다른 코루틴을 취소하지 않고, 처리되지 않은 예외를 전파하지 않게 해주는 특수한 Job이다.

```
class ComponentWithScope(dispatcher: CoroutineDispatcher = Dispatchers.D

    private val scope = CoroutineScope(dispatcher + SupervisorJob())

    fun start() {
        log("Starting!")
        scope.launch {
            while (true) {
                delay(500.milliseconds)
                log("Component working!")
            }
        }
        scope.launch {
            log("Doing a one-off task...")
            delay(500.milliseconds)
            log("Task done!")
        }
    }

    fun stop() {
        log("Stopping!")
        scope.cancel()
    }
}
```

- 자체 생명주기를 따르며 코루틴을 시작하고 관리할 수 있는 클래스를 만들었다.
- 이 클래스는 생성자 인자로 코루틴 디스패처를 받고, CoroutineScope 함수를 사용해 클래스와 연관된 새로운 코루틴 스코프를 생성한다.
- start 함수는 계속 실행되는 코루틴 하나와 작업을 수행하는 코루틴 스코프를 생성한다.
- stop 함수는 클래스와 연관된 범위를 취소하며, 이로 인해 이전에 시작된 코루틴들도 함께 취소된다.

```
fun main() {
    val c = ComponentWithScope()
    c.start()
    Thread.sleep(2000)
    c.stop()
}


// 22 [main] Starting!
// 37 [DefaultDispatcher-worker-2 @coroutine#2] Doing a one-off task...
// 544 [DefaultDispatcher-worker-1 @coroutine#1] Component working!
// 544 [DefaultDispatcher-worker-2 @coroutine#2] Task done!
// 1050 [DefaultDispatcher-worker-2 @coroutine#1] Component working!
// 1555 [DefaultDispatcher-worker-2 @coroutine#1] Component working!
// 2039 [main] Stopping!
```

- ComponentWithScope 클래스의 인스턴스를 생성하고 start를 호출하면 컴포넌트 내부에서 코루틴이 시작된다.
- 그 후 stop을 호출하면 컴포넌트의 생명주기가 종료된다.
- 생명주기를 관리해야 하는 컴포넌트를 다루는 프레임워크에서는 내부적으로 CoroutineScope 함수를 많이 사용한다.

💡 coroutineScope vs CoroutineScope

coroutineScope

- 작업을 동시성으로 실행하기 위해 분해할 때 사용
- 여러 코루틴을 시작하고, 모두 완료될 때까지 기다리며, 결과를 계산할 수도 있다.
- 자식들이 모두 완료될 때까지 기다리기 때문에 일시 중단 함수다.

CoroutineScope

- 코루틴을 클래스의 생명주기와 연관시키는 영역을 생성할 때 쓰인다.
- 이 함수는 영역을 생성하지만 추가 작업을 기다리지 않고 즉각 반환된다.
- 반환된 코루틴 스코프를 나중에 취소할 수 있다.

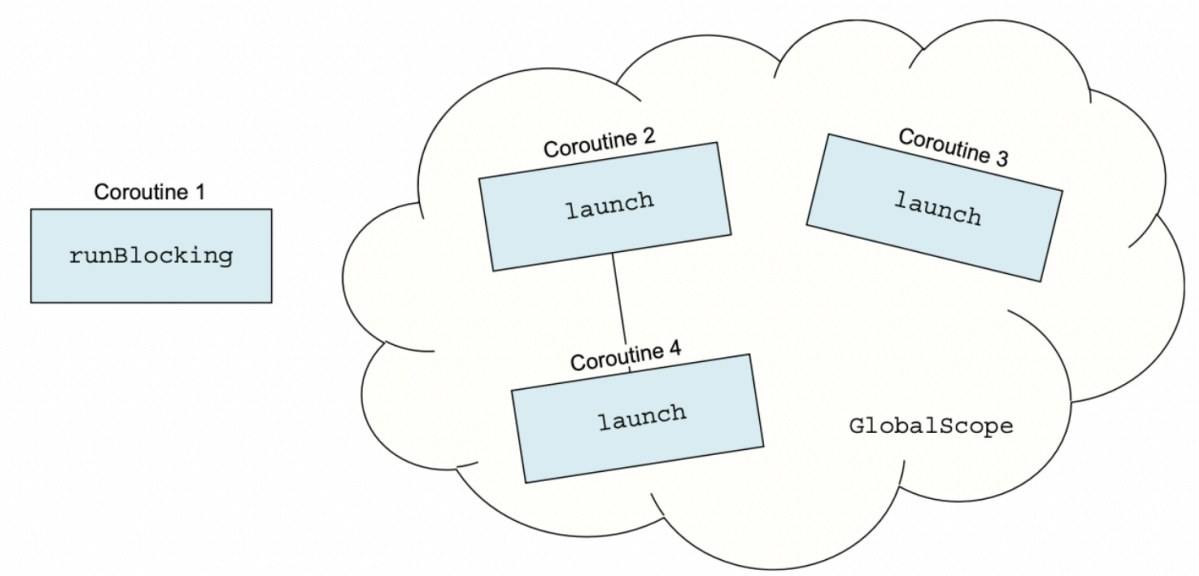
GlobalScope의 위험성

- 특수한 코루틴 스코프 인스터스인 GlobalScope
- 전역 수준에 존재하는 코루틴 스코프
- 편해보이지만 단점 존재
  - GlobalScope를 사용하면 구조화된 동시성이 제공하는 모든 이점을 포기해야 한다.
  - 전역 범위에서 시작된 코루틴은 자동으로 취소되지 않으며, 생명주기에 대한 개념도 없다.
  - 리소스 누수가 발생하거나 불필요한 작업을 계속 수행하면서 계산 자원을 낭비하게 될 가능성이 크다.

```
fun main() {
    runBlocking {
        GlobalScope.launch {
```

```
            delay(1000.milliseconds)
            launch {
                delay(250.milliseconds)
                log("Grandchild done")
            }
            log("Child 1 done!")
        }
        GlobalScope.launch {
            delay(500.milliseconds)
            log("Child 2 done!")
        }
        log("Parent done!")
    }
}

// 28 [main @coroutine#1] Parent done!
```



- GlobalScope를 사용함으로써 구조화된 동시성에서 자동으로 설정되는 계층 구조가 깨져서 즉시 종료된다.
- 코루틴 2부터 4는 runBlocking과 연관된 코루틴1과의 부모 관계에서 벗어나 있다.
- 따라서 부모 코루틴이 없으므로 프로그램은 자식들이 완료되기 전에 종료된다.
- 이러한 이유로 GlobalScope는 특수한 주석과 함께 선언된다.
    - 경고성

- 일반 애플리케이션에서 GlobalScope를 선택해야 하는 경우는 매우 드물다.
    - 보통은 코루틴 빌더나 coroutineScope 함수를 사용해 더 적합한 영역을 찾아 시작 하는 것이 좋다.

코루틴 컨텍스트와 구조화된 동시성

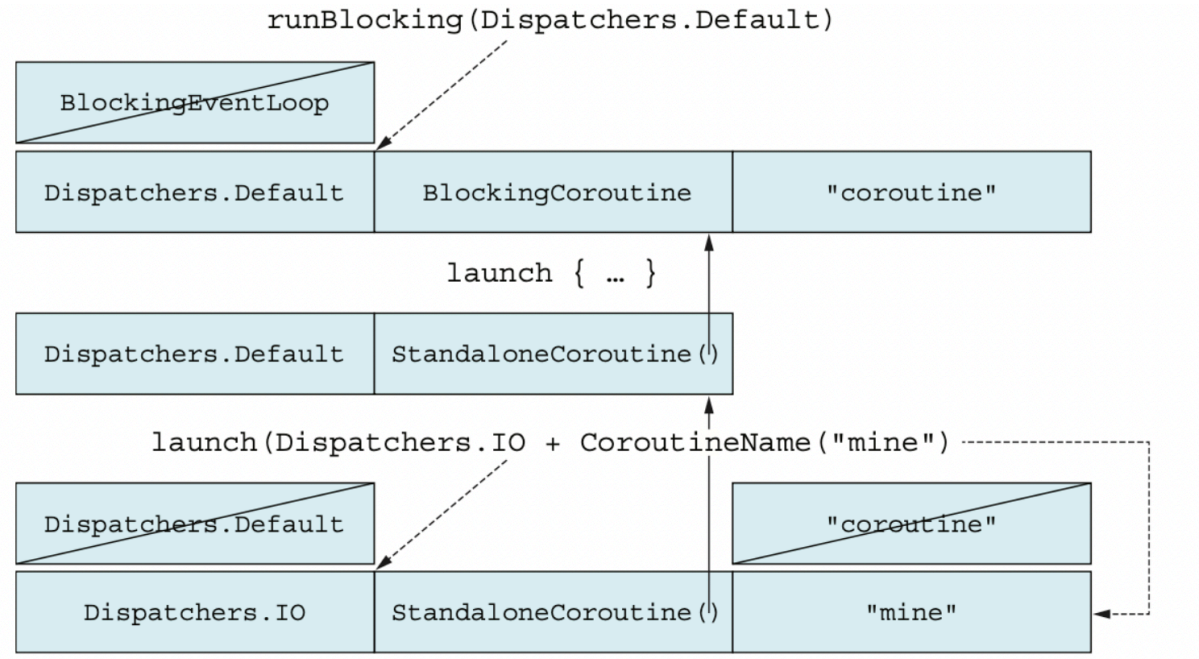
- 구조화된 동시성 개략적인 내용 파악했으므로 코루틴 컨텍스트에 대해 다시 돌아가자.
- 코루틴 컨텍스트는 구조화된 동시성 개념과 밀접한 관련이 있으며, 이는 코루틴 간의 부모 - 자식 계층을 따라 상속된다.
- 새로운 코루틴을 시작할 때 코루틴 컨텍스트에 어떤 일이 일어날까?
    - 먼저 자식 코루틴은 부모의 컨텍스트를 상속받는다.
    - 그런 다음 새로운 코루틴은 부모-자식 관계를 설정하는 역할을 하는 새 Job 객체를 생성한다.
    - 마지막으로 코루틴 컨텍스트에 전달된 인자가 적용된다.
    - 이 인자들은 상속받은 값을 덮어쓸 수 있다.

```
fun main() {
    runBlocking(Dispatchers.Default) {
        log(coroutineContext)
        launch {
            log(coroutineContext)
            launch(Dispatchers.IO + CoroutineName("mine")) {
                log(coroutineContext)
            }
        }
    }
}

// 0 [DefaultDispatcher-worker-1 @coroutine#1] [CoroutineId(1), "coroutine#1"
// 1 [DefaultDispatcher-worker-2 @coroutine#2] [CoroutineId(2), "coroutine#2
// 2 [DefaultDispatcher-worker-3 @mine#3] [CoroutineName(mine), Coroutine
```

- 이제 코루틴 컨텍스트가 코루틴 계층 구조에 따라 어떻게 전달되는지 알았기 때문에 “디스패처를 지정하지 않고 새로운 코루틴을 시작하면 어떤 디스패처에서 실행될까?” 라는 질문의 답은 더 쉽다.

- 답은 Default가 아니고 부모 코루틴의 디스패처에서 실행된다.



- runBlocking은 특수한 디스패처인 BlockingEvnetLoop로 시작되며, 인자로 받은 값에 의해 Dispatchers.Default로 덮어 써진다.
- 코루틴은 BlockingCoroutine 이라는 Job 객체를 생성하고, 기본값인 “coroutine”으로 이름을 초기화 한다.
- launch는 기본 디스패처를 상속받고 자신의 Job 객체로 StandaloneCoroutine을 생성하며 부모 Job과의 관계를 설정한다.
  - 이름은 변경되지 않는다.
- 두번째 launch 호출도 디스패처를 상속받고 새로운 Job을 생성하며, 코루틴 이름도 함께 설정된다.
  - launch로 전달된 파라미터는 디스패처를 IO로 변경하고 코루틴 이름을 mine으로 지정한다.

```
fun main() = runBlocking(CoroutineName("A")) {
    log("A's job: ${coroutineContext.job}")
    launch(CoroutineName("B")) {
        log("B's job: ${coroutineContext.job}")
        log("B's parent: ${coroutineContext.job.parent}")
```

```
    }
    log("A's children: ${coroutineContext.job.children.toList()}")
}

// 0 [main @A#1] A's job: "A#1":BlockingCoroutine{Active}@41
// 17 [main @A#1] A's children: ["B#2":StandaloneCoroutine{Active}@24]
// 25 [main @B#2] B's job: "B#2":StandaloneCoroutine{Active}@24
// 25 [main @B#2] B's parent: "A#1":BlockingCoroutine{Completing}@41
```

- 코드에서 코루틴 간의 부모 - 자식 관계, 더 정확히 코루틴과 연관된 Job간의 관계를 실제로 확인할 수 있다.
- 각 코루틴의 코루틴 컨텍스트에서 job, job.parent, job.children 속성을 확인하면 이를 볼 수 있다.

```
fun main() = runBlocking<Unit> { // coroutine#1
    log("A's job: ${coroutineContext.job}")
    coroutineScope {
        log("B's parent: ${coroutineContext.job.parent}") // A
        log("B's job: ${coroutineContext.job}") // C
        launch { //coroutine#2
            log("C's parent: ${coroutineContext.job.parent}") // B
        }
    }
}

// 0 [main @coroutine#1] A's job: "coroutine#1":BlockingCoroutine{Active}@3
// 4 [main @coroutine#1] B's parent: "coroutine#1":BlockingCoroutine{Active}
// 4 [main @coroutine#1] B's job: "coroutine#1":ScopeCoroutine{Active}@6c4
// 8 [main @coroutine#2] C's parent: "coroutine#1":ScopeCoroutine{Completi
```

- launch, async 코루틴 빌더 함수로 시작된 코루틴과 마찬가지로, coroutineScope 함수도 자체 Job 객체를 갖고 부모 - 자식 계층 구조에 참여한다.
- 구조화된 동시성에 의해 설정된 이 부모 - 자식 관계는 취소와도 연관이 있다.

취소

- 취소는 코드가 완료되기 전에 실행을 중단하는 것을 의미한다.
- 계산 작업을 취소할 수 있어야 견고하고 효율적이다.
  - 취소는 불필요한 작업을 막아준다.
    - 취소할 수 없으면 완료될 때까지 게산 처리하거나 네트워크를 통해 전체 응답 다운로드 및 결과를 버려야한다.
    - 서버 측 처리량에 긍정적
  - 메모리나 리소스 누수를 방지하는 데도 도움을 준다.
    - 코루틴이 리소스를 계속 차지하거나 메모래에서 데이터 구조체 대한 참조 유지로 인한 가비지 컬렉터 메모리 해제가 안 될 수 있다.
  - 오류 처리에도 중요한 역할
    - 18장
    - 여러 코루틴이 함께 결과를 계산하는 경우
    - 그 중 한 계산이 실패하면 더 이상 결과를 합리적으로 계산할 수 없는 상황인 경우, 모든 진행중인 요청이 완료될 때까지 기다릴 이유가 없다.
  - 이렇기 때문에 코틀린 코루틴은 취소를 처리하는 내장 메커니즘을 제공한다.

취소 촉발

- 여러 코루틴 빌더 함수의 반환값을 취소를 촉발하는 핸들로 사용할 수 있다.
- launch 코루틴 빌더는 Job을 반환하고, async 코루틴 빌더는 Deferred를 반환한다.
- 둘 다 cancel을 호출해 해당 코루틴의 취소를 촉발할 수 있다.

```
fun main() {
    runBlocking {
        val launchedJob = launch { // Job을 반환
            log("I'm launched!")
            delay(1000.milliseconds)
            log("I'm done!")
        }
        val asyncDeferred = async { // Deferred를 반환
```

```
        log("I'm async")
            delay(1000.milliseconds)
            log("I'm done!")
        }
        delay(200.milliseconds)
         // 2가지 다 취소 가능
        launchedJob.cancel()
        asyncDeferred.cancel()
    }
}

// 0 [main @coroutine#1] I'm launched!
// 7 [main @coroutine#1] I'm async
```

- 15.1.4 에서 본 것 처럼 각 코루틴 스코프의 코루틴 콘텍스트에도 Job이 포함돼 있으며, 이를 사용해 영역을 취소할 수 있다.
- 코루틴을 수동으로 취소하는 것 외에도 라이브러리가 특정 조건에서 자동으로 코루틴을 취소하게 할 수 있다.

시간제한이 초과된 후 자동으로 취소 호출

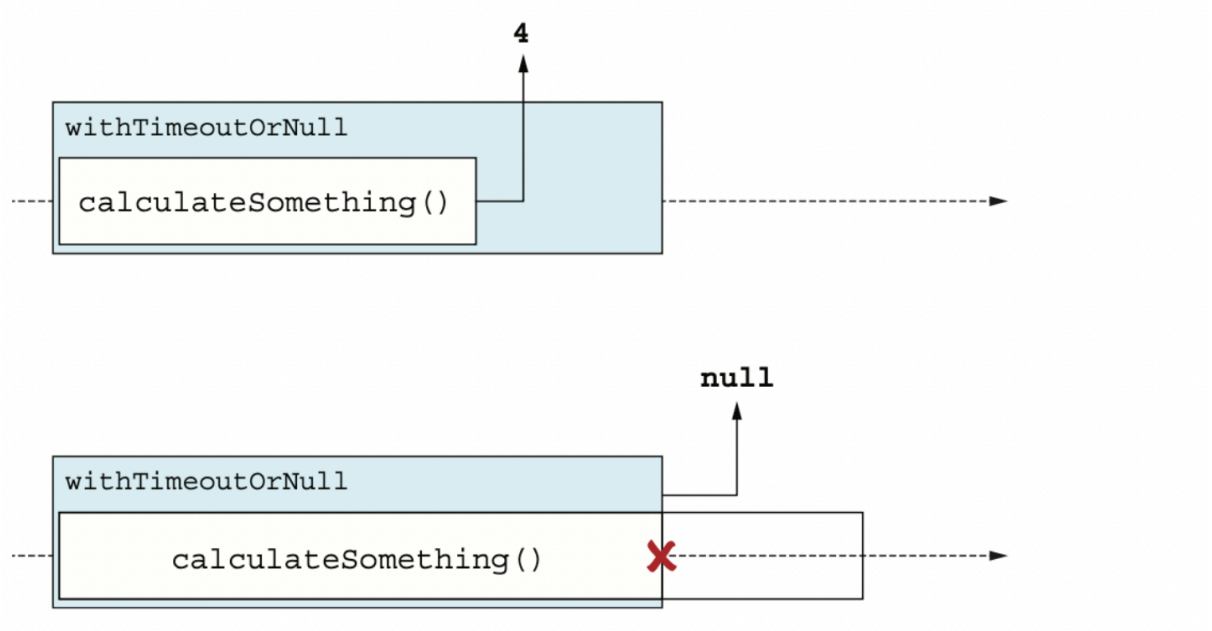
- 코틀린 코루틴 라이브러리는 코루틴의 취소를 자동으로 촉발할 수 있는 몇 가지 편리한 함수도 제공한다.
    - withTimeout
    - withTimeoutOrNull
- withTimeout
    - 타임아웃이 되면 예외를 발생시킨다.
        - TimeoutCancellationException
    - 타임아웃을 처리하려면 withTimeout 호출을 try 블록으로 감싸고 발생한 예외를 잡아내야 한다.
- withTimeoutOrNull
    - 비슷하게 타임아웃이 발생하면 null을 반환한다.

💡 TimeoutCancellationException 예외는 잊지 말고 캐치해야 한다.

- TimeoutCancellationException의 상위 타입인 CancellationException은 코루틴을 취소하기 위한 특별한 표식으로 사용된다.
- TimeoutCancellationException를 잡지 않으면 호출한 코루틴이 의도와 다르게 취소될 수 있다.
- 이 문제를 피하기 위해 withTimeoutOrNull 사용하는 편이 좋다.

```
suspend fun calculateSomething(): Int {
    delay(3.seconds)
    return 2 + 2
}

fun main() = runBlocking {
    val quickResult = withTimeoutOrNull(500.milliseconds) {
        calculateSomething()
    }
    println(quickResult)
    // null
    val slowResult = withTimeoutOrNull(5.seconds) {
        calculateSomething()
    }
    println(slowResult)
    // 4
}
```



- withTimeoutOrNull을 사용하면 일시 중단 함수의 실행 시간을 제한할 수 있다.
- 함수가 주어진 시간 내에 값을 반환하면 그 즉시 값을 반환하고, 시간이 초과되면 함수는 취소되고 null이 반환된다.

취소는 모든 자식 코루틴에게 전파된다.

- 코루틴을 취소하면 해당 코루틴의 모든 자식 코루틴도 자동으로 취소된다.
  - 구조화된 동시성의 강력한 기능
- 각 코루틴은 자신이 시작한 다른 코루틴을 알고 있기 때문에 취소할 때 스스로 자식들을 정리할 수 있으며, 불필요한 코루틴이 남지 않는다.

```
fun main() = runBlocking {
    val job = launch {
        launch {
            launch {
                launch { // 취소된 잡의 손자의 손자
                    log("I'm started")
                    delay(500.milliseconds)
                    log("I'm done!")
                }
            }
```

```
        }
    }
    delay(200.milliseconds)
    job.cancel()
}

// 0 [main @coroutine#5] I'm started
```

- 코루틴이 중첩돼 있는 경우에도 가장 바깥쪽 코루틴을 취소하면 손자의 손자 코루틴까지도 모두 적절히 취소된다.
- 그렇다면 코루틴은 어디에서 어떻게 취소될 수 있을까?

취소된 코루틴은 특별한 지점에서 CancellationException을 던진다.

- 취소 메커니즘은 CancellationException 이라는 특수한 예외를 특별한 지점에서 던지는 방식으로 작동한다.
    - 우선적으로 일시 중단 지점이 이런 지점이다.
- 취소된 코루틴은 일시 중단 지점에서 CancellationException을 던진다.
    - 일시 중단 지점은 코루틴의 실행을 일시 중단할 수 있는 지점이다.
- 일반적으로 코루틴 라이브러리 안의 모든 일시 중단 함수는 CancellationException 던져질 수 있는 지점을 도입한다.

```
coroutineScope {
    log("A")
    delay(500.milliseconds) // 이 지점에서 함수가 취소될 수 있다.
    log("B")
    log("C")
}
```

- 해당 코드에서는 영역이 취소됐는지 여부에 따라 'A'나 'ABC' 가 출력되며, 'AB'는 절대 출력되지 않는다.
    - 취소 지점이 없기 때문

- 코루틴은 예외를 사용해 코루틴 계층에서 취소를 전파하기 때문에 이 예외를 실수로 삼켜버리거나 직접 처리하지 않도록 주의해야 한다.

```
suspend fun doWork() {
    delay(500.milliseconds) // 예외를 던지지만
    throw UnsupportedOperationException("Didn't work!")
}

fun main() {
    runBlocking {
        withTimeoutOrNull(2.seconds) {
            while (true) {
                try {
                    doWork()
                } catch (e: Exception) { // 예외를 삼켜버려서 취소를 막는다.
                    println("Oops: ${e.message}")
                }
            }
        }
    }
}

// Oops: Didn't work!
// Oops: Didn't work!
// Oops: Didn't work!
// Oops: Timed out waiting for 2000 ms
// ... (does not terminate)
```

- 2초 후 withTimeoutOrNull 함수는 자식 코루틴 스코프의 취소를 요청한다.
- 이로 인해 다음 delay 호출이 CancellationException을 던지게 된다.
- 하지만 catch 구문에서 모든 종류의 예외를 잡기 때문에 이 코드는 무한히 반복된다.
- 문제를 해결하려면 CancellationException 는 다시 예외를 던지거나, UnsupportedOperationException 예외만 잡아야 한다.
    - 그러면 예상한대로 취소된다.

취소는 협력적이다

- 코틀린 코루틴에 기본적으로 포함된 모든 함수는 이미 취소 가능하다.
- 하지만 직접 작성한 코드에서는 직접 코루틴을 취소 가능하게 만들어야 한다.

```
suspend fun doCpuHeavyWork(): Int {
    log("I'm doing work!")
    var counter = 0
    val startTime = System.currentTimeMillis()
    while (System.currentTimeMillis() < startTime + 500) {
        counter++ // 500ms 동안 증가시켜 cpu 집약적인 연산
    }
    return counter
}

fun main() {
    runBlocking {
        val myJob = launch {
            repeat(5) {
                doCpuHeavyWork()
            }
        }
        delay(600.milliseconds)
        myJob.cancel()
    }
}

// 30 [main @coroutine#2] I'm doing work!
// 535 [main @coroutine#2] I'm doing work!
// 1036 [main @coroutine#2] I'm doing work!
// 1537 [main @coroutine#2] I'm doing work!
// 2042 [main @coroutine#2] I'm doing work!
```

- 이 코드가 “I'm doing work!” 메시지를 2번 출력한 후 취소되리라 예상할 수도 있지만 실제로는 5번 모두 완료된다.

- 왜 그럴까?
    - 취소는 함수 안의 일시 중단 지점에서 CancellationException을 던지는 방식으로 작동한다는 사실을 기억하자
    - doCpuHeavyWork 함수는 suspend 변경자로 표시돼 있음에도 실제로는 일시 중단 지점을 포함하지 않는다.
    - 함수는 log를 호출하고 그 후에 500ms 동안 카운터를 증가시키는 cpu 집약 작업을 수행한다.
- 이것이 바로 코틀린 코루틴의 취소가 협력적 이라고 하는 이유
    - 일시 중단 함수는 스스로 취소 가능하게 로직을 제공해야 한다.
    - 코드가 취소 가능한 다른 함수를 호출할 때는 자동으로 취소 가능 지점이 도입된다.
- 예를 들어 doWork 함수 본문에 delay 호출을 추가하면 해당 함수는 취소 가능한 지점을 갖게 된다.

```
suspend fun doCpuHeavyWork(): Int {
    log("I'm doing work!")
    var counter = 0
    val startTime = System.currentTimeMillis()
    while (System.currentTimeMillis() < startTime + 500) {
        counter++
      delay(100.milliseconds) // 이 함수 호출은 doCpuHeavyWork 함수를 취소할 수
    }
    return counter
}
```

- 인위적으로 지연시키는 것은 원하지 않을 것이다.
- 대신 코틀린 코루틴에는 코드를 취소 가능하게 만드는 유틸리티 함수들이 있다.
    - ensureActive
    - yield 함수
    - isActive 속성

코루틴이 취소됐는지 확인

- 코루틴이 취소됐는지 확인할 때는 CoroutineScope의 isActive 속성을 확인한다.
- 이 값이 false 라면 코루틴은 더 이상 활성 상태가 아니다.
- 이 경우 현재 작업을 완료하고, 획득한 리소스를 닫은 후 반환할 수 있다.

```
val myJob = launch {
    repeat(5) {
        doCpuHeavyWork()
        if(!isActive) return@launch
    }

}
```

- 예를 들어 현재 코루틴 스코프가 취소 됐는지 확인하도록 루프를 다시 작성할 수 있다.

```
val myJob = launch {
    repeat(5) {
        doCpuHeavyWork()
        ensureActive()
    }

}
```

- isActive를 확인해서 명시적으로 반환하는 대신, 편의 함수로 ensureActive를 제공한다.
- 해당 함수는 코루틴이 더 이상 활성 상태가 아닐 경우 CancellationException을 던진다.

다른 코루틴에게 기회를 주기 : yield 함수

- 이와 관련해 yield라는 함수도 제공한다.
- 이 함수는 코드 안에서 취소 가능 지점을 제공할 뿐만 아니라 현재 점유된 디스패처에서 다른 코루틴이 작업할 수 있게 해준다.

```
suspend fun doCpuHeavyWork(): Int {
    var counter = 0
    val startTime = System.currentTimeMillis()
    while (System.currentTimeMillis() < startTime + 500) {
        counter++
    }
    return counter
}

fun main() {
    runBlocking {
        launch {
            repeat(3) {
                doCpuHeavyWork()
            }
        }
        launch {
            repeat(3) {
                doCpuHeavyWork()
            }
        }
    }
}

// 29 [main @coroutine#2] I'm doing work!
// 533 [main @coroutine#2] I'm doing work!
// 1036 [main @coroutine#2] I'm doing work!
// 1537 [main @coroutine#3] I'm doing work!
// 2042 [main @coroutine#3] I'm doing work!
// 2543 [main @coroutine#3] I'm doing work!
```

- 작업을 수행하는 두 코루틴을 실행하는 코드
- doCpuHeavyWork 함수가 일시 중단 지점을 포함하지 않으면 첫 번째로 실행된 코루틴이 완료될 때까지 두 번째 코루틴은 실행되지 않음을 알 수 있다.
- 이유는 무엇일까?

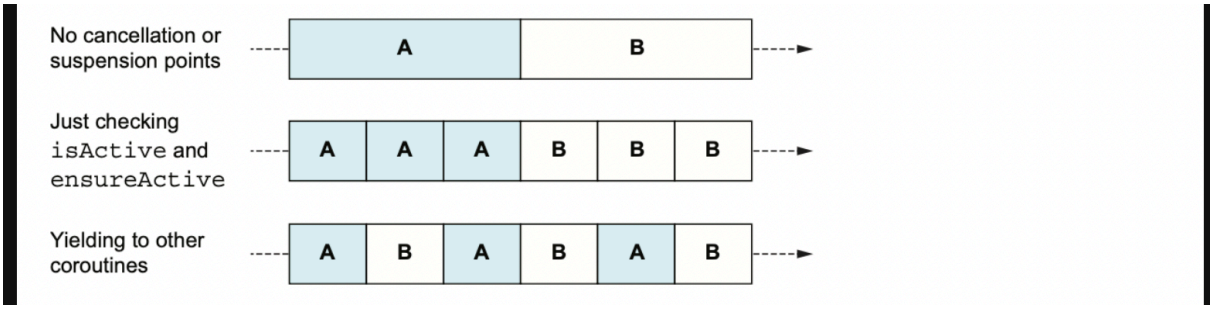
    - 코루틴 본문에 일시 중단 지점이 없으면 첫 번째 코루틴의 실행이 일시 중단될 기회가 없어서 두 번째 코루틴이 실행되지 못한다.
    - isActive를 확인하거나 ensureActive를 호출해도 동일하다.
    - 이 함수들은 취소 여부만 확인할 뿐 실제로 코루틴을 일시 중단시키지는 않는다.
- 여기서 yield 함수가 유용하다
    - 해당 함수는 CancellationException을 던질 수 있는 지점을 제공할 뿐만 아니라, 대기 중인 다른 코루틴이 있으면 디스패처가 제어를 다른 코루틴에게 넘길 수 있게 해준다.

```
suspend fun doCpuHeavyWork(): Int {
    var counter = 0
    val startTime = System.currentTimeMillis()
    while (System.currentTimeMillis() < startTime + 500) {
        counter++
        yield()
    }
    return counter
}

fun main() {
    runBlocking {
        launch {
            repeat(3) {
                doCpuHeavyWork()
            }
        }
        launch {
            repeat(3) {
                doCpuHeavyWork()
            }
        }
    }
}
```

```
0 [main @coroutine#2] I'm doing work!
559 [main @coroutine#3] I'm doing work!
1062 [main @coroutine#2] I'm doing work!
1634 [main @coroutine#3] I'm doing work!
2208 [main @coroutine#2] I'm doing work!
2734 [main @coroutine#3] I'm doing work!
```

- yield를 호출하면 서로 다른 코루틴들이 교차 실행되며 코루틴2와 코루틴3이 번갈아가며 작업을 처리할 수 있게 된다.



- 일시 중단 지점과 취소 지점이 없는 경우 isActive나 ensureActive를 확인하는 것과 yield를 호출하는 것의 차이를 보여준다.
- 일시 중단 지점이 없으면 여러 코루틴이 완전히 완료될 때까지 교차 실행되지 않으며(단일 스레드 디스패처의 경우), 취소 지점에서 isActive를 확인하거나 ensureActive를 호출해도 작업이 조기에 취소될 수 있다.
- yield를 사용해 다른 코루틴이 기저 스레드를 사용하면 코루틴이 교차 실행될 수 있다.

함수	설명
isActive	취소가 요청됐는지 확인한다.(작업을 중단하기 전에 정리 작업을 수행하기 위함)
ensureActive	'취소 지점' 도입. 취소 시 `CancellationException` 을 던져 즉시 작업을 중단한다.
yield	CPU 집약적인 작업이 기저 스레드(또는 스레드 풀)를 소모하는 것을 방지하기 위해 계산 자원을 양도한다

리소르를 얻을 때 취소를 염두에 두기

- 실제 코드는 종종 db 연결, io 등과 같은 리소르를 사용해 작업해야 하며, 사용 후 이를 명시적으로 닫아야 적절하게 해제된다.

- 취소는 다른 예외와 마찬가지로 코드의 조기 반환을 유발할 수 있으므로, 코루틴이 취소된 후에도 리소르를 계속 소유하지 않게 주의해야 한다.

```
class DatabaseConnection : AutoCloseable {
    fun write(s: String) = println("writing $s!")
    override fun close() {
        println("Closing!")
    }
}

fun main() {
    runBlocking {
        val dbTask = launch {
            val db = DatabaseConnection()
            delay(500.milliseconds)
            db.write("I love coroutines!")
            db.close()
        }
        delay(200.milliseconds)
        dbTask.cancel()
    }
    println("I leaked a resource!")
}
```

- 문자열을 저장하기 위해 데이터베이스 커넥션 객체를 사용하는데, 이 요청과 관련된 코루틴을 db 연결을 닫기 전에 의도적으로 취소해보자
- 그 결과 close 함수가 호출되지 않고 리소스가 누수됐다.
- 코루틴 기반 코드는 항상 취소 시에도 견고하게 작동하도록 설계돼야 한다.

```
val dbTask = launch {
    val db = DatabaseConnection()
    try {
```

```
        delay(500.milliseconds)
        db.write("I love coroutines!")
    } finally {
        db.close()
    }
}
```

- 취소가 발생하면 `CancellationException`이 발생하므로, 코루틴 컨텍스트상에서도 일반 함수에서 예외를 처리하는 것과 동일한 메커니즘, finally를 사용할 수 있다.

```
val dbTask = launch {
    DatabaseConnection().use {
        delay(500.milliseconds)
        it.write("I love coroutines!")
    }
}
```

- 리소스가 AutoClosable 인터페이스를 구현하는 경우 use 함수를 사용해 같은 동작 가능하다.

프레임워크가 여러분 대신 취소할 수 있다.

- 많은 실제 애플리케이션에서는 프레임워크가 코루틴 스코프를 제공하고, 취소를 자동으로 처리한다.
- 이런 경우 사용자는 적절한 코루틴 스코프를 선택하고, 작성한 코드가 실제로 취소될 수 있도록 설계할 의무가 있다.

```
class MyViewModel: ViewModel() {
    init {
        viewModelScope.launch { // 스코프
            while (true) {
                println("Tick!")
                delay(1000.milliseconds)
            }
```

```
        }
    }
}
```

- android
    - ViewModel이 표시된 화면에서 벗어날 때 viewModelScope가 취소되며, 스코프 안에서 모든 코루틴도 함께 취소된다.

```
routing {
    get("/") { // this: PiplelineContext
        launch { // 스코프
            println("I'm doing some background work!")
            deplay(5000.milliseconds)
            println("I'm done")
        }
    }
}
```

- ktor
- PiplelineContext : CoroutineScope 상속

```
routing {
    get("/") {
        call.application.launch {
            println("I'm doing some background work!")
            delay(5000.milliseconds)
            println("I'm done")
        }
    }
}
```

- 클라이언트와 상관없이 비동기적으로 계속 작업을 수행해야하는 경우에는 다른 스코프
- Application
    - 코루틴 스코프로 작동

    - 애플리케이션 생명주기가 같다.
    - call 변수를 통해 범우에 접근 할 수 있다.
- http 취소하더라도 코루틴은 취소되지 않는다.
    - 요청 수준의 코루틴 스코프가 아니기 때문
    - 애플리케이션 수준

요약

- 구조화된 동시성은 코루틴의 작업을 제어할 수 있게 해주며, '제멋대로인' 코루틴이 취소되지 않고 계속 실행되는 것을 방지한다.
- 일시 중단 함수인 coroutineScope 도우미 함수와 CoroutineScope 생성자 함수를 사용해 새로운 코루틴 스코프를 생성할 수 있다. 이름은 비슷하지만 이들의 목적은 다르다.
    - coroutineScope는 작업을 병렬로 분해하기 위한 함수로, 여러 코루틴을 시작 하고 결과를 계산한 후 그 결과를 반환한다.
    - Coroutinescope는 클래스의 생명주기와 코루틴을 연관시키는 스코프를 생성 하며, 일반적으로 SupervisorJob과 함께 사용된다.
- Globalscope는 특별한 코루틴 스코프로, 예제 코드에서 자주 볼 수 있지만 구조화된 동시성을 깨뜨리기 때문에 애플리케이션 코드에서는 사용하지 말아야 한다.
- 코루틴 콘텍스트는 개별 코루틴이 어떻게 실행되는지 관리하며, 코루틴 계층을 따라 상속된다.
- 코루틴과 코루틴 스코프 간의 부모-자식 계층 구조는 코루틴 콘텍스트에 있는 Job객체를 통해 설정된다.
- 일시 중단 지점은 코루틴이 일시 중단될 수 있고, 다른 코루틴이 작업을 시작할 수 있는 지점이다.
- 취소는 일시 중단 지점에서 CancellationException을 던지는 방식으로 구현된다.
- 취소 예외는 절대 무시(잡아내고 처리하지 않음)돼서는 안 된다. 예외를 다시 던지던지 아니면 아예 잡아내지 않는 것이 좋다.
- 취소는 정상적인 상황이므로 코드는 이를 처리할 수 있게 설계해야 한다.
- cancel이나 withTimoutOrNull 같은 함수를 사용해 직접 취소를 호출할 수 있다.
    - 기존의 여러 프레임워크도 코루틴을 자동으로 취소할 수 있다.
- 함수에 suspend 변경자를 추가하는 것만으로는 취소를 지원할 수 없다.

    - 하지만 코틀린 코루틴은 취소 가능한 일시 중단 함수를 작성하는 데 필요한 메커니즘(예: ensureActive나 yield 함수나 isActive 속성)을 제공한다.
- 프레임워크는 코루틴 스코프를 사용해 코루틴을 애플리케이션의 생명주기와 연결하는 데 도움을 준다(예: 화면에 ViewModel이 표시되는 동안이나 요청 핸들러가 실행되는 동안).