# 13장 동시성

이 장에서는 동시성(concurrent) 코드 작성이라는 중요한 주제를 다룬다. 목표는 코루틴을 이해하는 것이다. 코루틴은 코틀린의 뛰어난 기능 중 하나로 버전 1.1부터 도입됐고 1.3부터 정식으로 릴리스(release) 상태로 격상됐다.

먼저 코틀린 코루틴을 뒷받침하는 일시중단 가능한 함수(suspending function)와 구조적인 복잡성에 대해 살펴보고, 조금씩 동시성 흐름 제어, 코루틴의 생명 주기에 따른 상태 변화, 코루틴 취소나 코루틴의 예외 처리, 동시성 작업이 스레드를 할당받는 방법 등의 주제로 나아간다.

그 후 동시성 작업 사이에 통신을 구현하거나 스레드 안전한(thread-safe) 방식으로 변경 가능한 상태를 공유하고 싶을 때 사용할 수 있는 채널이나 액터와 같은 주제도 살펴본다.

마지막으로 스레드 생성, 동기화(synchronization)와 락(lock) 사용 등 자바 동시성 API를 코틀린에서 사용할 때 도움이 되는 유틸리티들에 대해 논의해보겠다.

## 구조

* 코루틴
* 동시성 통신
* 자바 동시성 사용

## 목표

코루틴 라이브러리가 제공하는 동시성 기본 요소들을 사용하는 방법을 배우고, 이를 통해 규모를 가변적으로 조절할 수 있는(scalable), 반응형(responsive) 코드를 작성한다.

## 코루틴

코틀린 프로그램에서도 자바 동시성 기본 요소를 쉽게 사용해 스레드 안전성을 달성할 수 있다. 하지만 자바 동시성 요소를 사용해도 대부분의 동시성 연산이 블러킹(blocking) 연산이기 때문에 여전히 몇 가지 문제가 남는다. 다른 말로 설명하면 Thread.sleep(), Thread.join(), Object.wait()은 실행이 끝날 때까지 블럭된다. 스레드를 블럭하고 나중에 실행을 재개하려면 시스템 수준에서 계산 비용이 많이 드는 문맥 전환(context switch)을 해야 하기 때문에 프로그램 성능에 부정적인 영향을 미칠 수 있다. 설상가상으로 스레드마다 상당한 양의 시스템 자원을 유지해야 하기 때문에 동시성 스레드를 아주 많이 사용하는 것은 비실용적이거나 (운영체제나 시스템 종류에 따라) 아예 불가능할 수도 있다.

더 효율적인 접근 방법은 비동기(asynchronous) 프로그래밍이다. 동시성 연산에 대해 해당 연산이 완료될 때 호출될 수 있는 람다를 제공할 수 있고, 원래 스레드는 블럭된 상태로 작업 완료를 기다리는 대신 (고객 요청을 처리하거나 UI 이벤트를 처리하는 등의) 다른 유용한 작업을 계속 수행할 수 있다. 이런 접근 방법의 가장 큰 문제는 일반적인 명령형 제어 흐름을 사용할 수 없어서 엄청나게 코드 복잡도가 늘어난다는 점에 있다.

코틀린에서는 두 접근 방법의 장점을 함께 취할 수 있다. 코루틴이라는 강력한 메커니즘 덕분에 우리에게 익숙한 명령형 스타일로 코드를 작성하면 컴파일러가 코드를 자동으로 효율적인 비동기 계산으로 변환해준다. 이런 메커니즘은 실행을 잠시 중단했다가 나중에 중단한 지점부터 실행을 다시 재개할 수 있는 일시중단 가능한 함수라는 개념을 중심으로 이뤄진다.

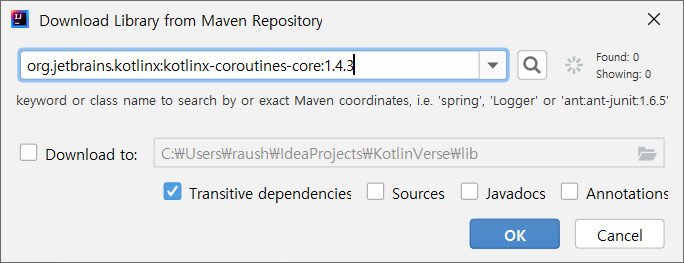
대부분의 코루틴 기능이 별도 라이브러리로 제공되기 때문에 명시적으로 프로젝트 설정에 이를 추가해야 한다. 이 책에서 사용하는 버전은 org.jetbrains.kotlinx:kotlinx-coroutines-core:1.4.3라는 메이븐(maven) 좌표(coordinate)를 통해 사용할 수 있다.

<참고> IDE 팁

메이븐이나 그레이들 같은 빌드 시스템을 채택하지 않고 인텔리J IDEA를 사용하고 있다면 다음과 같은 단계를 거쳐 코루틴 라이브러리를 추가할 수 있다.

1. Porject View 패널의 루트 노드에서 [F4]를 누르거나 마우스 오른쪽 클릭을 해서 Open Module Settings를 선택하라.
2. 왼쪽의 Libraries를 클릭하고 맨 위에 있는 툴바에서 [+] 버튼을 클릭한 다음, From Maven ... 옵션을 선택하라.
3. 라이브러리의 메이븐 아티팩트 좌표를 입력하고(예: org.jetbrains.kotlinx:kotlinx-coroutines-core:1.4.3) OK를 클릭하라(그림 13-1).
4. IDE는 여러분이 지정한 라이브러리와 관련된 의존관계 라이브러리를 함께 다운로드해서 프로젝트 모듈에 추가해준다. **OK**를 눌러서 변경된 내용을 적용하라.

그림 13-1: 코틀린 코루틴 라이브러리 다운로드받기



</참고>

이제부터는 코루틴 라이브러리에서 쓰이는 기본 개념을 설명하고 이런 개념을 동시성 프로그래밍에 어떻게 적용하는지 살펴본다.

## 코루틴과 일시중단 함수

전체 코루틴 라이브러리를 뒷받침하는 기본 요소는 일시중단 함수이다. 이 함수는 일반적인 함수를 더 일반화해서 함수 본문의 원하는 지점에서 함수에 필요한 모든 런타임 문맥을 저장하고 함수 실행을 중단한 다음, 나중에 필요할 때 다시 실행을 계속 진행할 수 있게 한 것이다. 코틀린에서는 이런 함수에 suspend라는 변경자를 붙인다.

<코드>

suspend fun foo() {  
 println("Task started")  
 delay(100)  
 println("Task finished")  
}

</코드>

delay() 함수는 코루틴 라이브러리에 정의된 일시중단 함수다. 이 함수는 Thread.sleep()과 비슷한 일을 한다. 하지만 delay()는 현재 스레드를 블럭시키지 않고 자신을 호출한 함수를 일시중단시키고 스레드를 (다른 일시중단된 함수를 다시 계속 실행하는 등의) 다른 작업을 수행할 수 있게 풀어준다.

일시중단 함수는 일시중단 함수와 일반 함수를 원하는 대로 호출할 수 있다. 일시중단 함수를 호출하면 해당 호출 지점이 일시중단 지점이 된다. 일시중단 지점은 임시로 실행을 중단했다가 나중에 재개할 수 있는 지점을 말한다. 반면 일반 함수 호출은 (지금까지 우리가 다뤄온) 일반 함수처럼 작동해서 함수 실행이 다 끝난 뒤 호출한 함수로 제어가 돌아온다. 반면에 코틀린은 일반 함수가 일시중단 함수를 호출하는 것을 금지한다.

<코드>

fun foo() {  
 println("Task started")  
 delay(100) // Error: delay is a suspend function  
 println("Task finished")  
}

</코드>

<참고> IDE 팁

인텔리J 코틀린 플러그인을 사용하면 일시중단을 호출하는 소스 코드 줄의 왼쪽에 특별한 아이콘이 표시되기 때문에 쉽게 일시중단 함수 호출을 구분할 수 있다(그림 13-1).

그림 13-1: IDE에서 본 일시중단 호출



</참고>

일시중단 함수를 일시중단 함수에서만 호출할 수 있다면, 어떻게 일반 함수에서 일시중단 함수를 호출할 수 있을까? 가장 분명한 방법은 main()을 suspend로 표시하는 것이다.

<코드>

import kotlinx.coroutines.delay  
  
suspend fun main() {  
 println("Task started")  
 delay(100)  
 println("Task finished")  
}

</코드>

이 코드를 실행하면 예상대로 다음 두 문장이 100밀리초 간격으로 표시되는 것을 볼 수 있다.

<코드>

Task started  
Task finished

</코드>

그러나 현실적인 경우 동시성 코드의 동작을 제어하고 싶기 때문에, 공통적인 생명주기(life cycle)와 문맥이 정해진 몇몇 작업(task)이 정의된 구체적인 영역 안에서만 동시성 함수를 호출한다. (이런 구체적 영역을 제공하기 위해) 코루틴을 실행할 때 사용하는 여러 가지 함수를 코루틴 빌더(coroutine builder)라고 부른다. 코루틴 빌더는 CoroutineScope 인스턴스의 확장 함수로 쓰인다. CoroutineScope에 대한 구현 중 가장 기본적인 것으로 GlobalScope 객체가 있다. GlobalScope 객체를 사용하면 독립적인 코루틴을 만들 수 있고, 이 코루틴은 자신만의 작업을 내포할 수 있다. 이제 자주 사용하는 launch(), async(), runBlocking()이라는 코루틴 빌더를 살펴보자.

### 코루틴 빌더

launch() 함수는 코루틴을 시작하고, 코루틴을 실행 중인 작업의 상태를 추적하고 변경할 수 있는 Job 객체를 돌려준다. 이 함수는 CoroutineScope.() -> Unit 타입의 일시중단 람다를 받는다. 이 람다는 새 코루틴의 본문에 해당한다. 간단한 예제를 보자.

<코드>

import kotlinx.coroutines.\*  
import java.lang.System.\*  
  
fun main() {  
 val time = currentTimeMillis()  
   
 GlobalScope.launch {  
 delay(100)  
 println("Task 1 finished in ${currentTimeMillis() - time} ms")  
 }  
   
 GlobalScope.launch {  
 delay(100)  
 println("Task 2 finished in ${currentTimeMillis() - time} ms")  
 }  
   
 Thread.sleep(200)  
}

</코드>

이 코드를 실행하면 다음과 같은 동작을 볼 수 있다.

<코드>

Task 2 finished in 176 ms  
Task 1 finished in 176 ms

</코드>

두 작업이 프로그램을 시작한 시점을 기준으로 거의 동시에 끝났다는 점에서 알 수 있는 것처럼 두 작업이 실제로 병렬적으로 실행됐다는 점에 주목해보자. 다만 실행 순서가 항상 일정하게 보장되지는 않기 때문에 상황에 따라 둘 중 어느 한쪽이 더 먼저 표시될 수 있다. 코루틴 라이브러리는 필요할 때 실행 순서를 강제할 수 있는 도구도 제공한다. 이는 동시성 통신 관련한 절에서 설명한다.

main() 함수 자체는 Thread.sleep()을 통해 메인 스레드 실행을 잠시 중단한다. 이를 통해 코루틴 스레드가 완료될 수 있도록 충분한 시간을 제공한다. 코루틴을 처리하는 스레드는 데몬 모드(daemon mode)로 실행되기 때문에 main() 스레드가 이 스레드보다 빨리 끝나버리면 자동으로 실행이 끝나버린다.

일시중단 함수의 내부에서 sleep()와 같은 스레드를 블럭시키는 함수를 실행할 수도 있지만 그런 식의 코드는 코루틴을 사용하는 목적에 위배된다는 점을 염두에 둬야 한다. 그래서 동시성 작업의 내부에서는 일시중단 함수인 delay()를 사용해야 한다.

<참고> IDE 팁

인텔리J 코틀린 플러그인은 Thread.sleep()이나 Thread.join() 같은 잠재적인 블러킹 함수를 호출하면 경고를 표시해준다.

</참고>

코루틴은 스레드보다 훨씬 가볍다. 특히 코루틴은 유지해야 하는 상태가 더 간단하고 일시중단 및 재개될 때 완전한 문맥 전환을 사용하지 않아도 되기 때문에 엄청난 수의 코루틴을 충분히 동시에 실행할 수 있다.

launch() 빌더는 동시성 작업이 결과를 만들어내지 않는 경우 적합하다. 그래서 이 빌더는 Unit 타입을 반환하는 람다를 인자로 받는다. 하지만 결과가 필요한 경우에는 async()라는 다른 빌더 함수를 사용해야 한다. 이 함수는 Deferred의 인스턴스를 돌려주고, 이 인스턴스는 Job의 하위 타입으로 await() 메서드를 통해 계산 결과에 접근할 수 있게 해준다. await() 메서드를 호출하면 await()은 계산이 완료되거나(따라서 결과가 만들어지거나), 계산 작업이 취소될 때까지 현재 코루틴을 일시중단시킨다. 작업이 취소되는 경우 await()는 예외를 발생시키면서 실패한다. async()를 자바의 퓨처(future)에 해당하는 일시중단 빌더라고 생각할 수 있다. 예제를 살펴보자.

<코드>

import kotlinx.coroutines.\*  
  
suspend fun main() {  
 val message = GlobalScope.async {  
 delay(100)  
 "abc"  
 }  
   
 val count = GlobalScope.async {  
 delay(100)  
 1 + 2  
 }  
   
 delay(200)  
   
 val result = message.await().repeat(count.await())  
 println(result)  
}

</코드>

이 경우에는 main()을 suspend로 표시해서 두 Deferred 작업에 대해 직접 await() 메서드를 호출했다. 출력은 기대한 대로 다음과 같다.

<코드>

abcabcabc

</코드>

launch()와 async() 빌더의 경우 (일시중단 함수 내부에서) 스레드 호출을 블럭시키지는 않지만, 백그라운드 스레드를 공유하는 풀(pool)을 통해 작업을 실행한다. 앞에서 살펴본 launch() 예제에서는 메인 스레드가 처리할 일이 별로 없었기 때문에 sleep()을 통해 백그라운드 스레드에서 실행되는 작업이 완료될 때까지 기다려야 했다. 반대로 runBlocking() 빌더는 디폴트로 현재 스레드에서 실행되는 코루틴을 만들고 코루틴이 완료될 때까지 현재 스레드의 실행을 블럭시킨다. 코루틴이 성공적으로 끝나면 일시중단 람다의 결과가 runBlocking() 호출의 결괏값이 된다. 코루틴이 취소되면 runBlocking()은 예외를 던진다. 반면에 블럭된 스레드가 인터럽트되면 runBlocking()에 의해 시작된 코루틴도 취소된다. 예를 살펴보자.

<코드>

import kotlinx.coroutines.\*  
  
fun main() {  
 GlobalScope.launch {  
 delay(100)  
 println("Background task: ${Thread.currentThread().name}")  
 }  
 runBlocking {  
 println("Primary task: ${Thread.currentThread().name}")  
 delay(200)  
 }  
}

</코드>

이 프로그램을 실행하면 다음과 비슷한 결과를 볼 수 있다.

<코드>

Primary task: main  
Background task: DefaultDispatcher-worker-2

</코드>

runBlocking() 내부의 코루틴은 메인 스레드에서 실행된 반면, launch()로 시작한 코루틴은 공유 풀에서 백그라운드 스레드를 할당받았음을 알 수 있다.

이런 블러킹 동작 때문에 runBlocking()을 다른 코루틴 안에서 사용하면 안 된다. runBlocking()은 블러킹 호출과 넌블러킹 호출 사이의 다리 역할을 하기 위해 고안된 코루틴 빌더이기 때문에, 테스트나 메인 함수에서 최상위 빌더로 사용하는 등의 경우에만 runBlocking()을 써야 한다.

### 코루틴 영역과 구조적 동시성

지금까지 살펴본 예제 코루틴은 전역 영역(global scope)에서 실행됐다. 전역 영역이란 코루틴의 생명주기가 전체 애플리케이션의 생명주기에 의해서만 제약되는 영역이다. 경우에 따라서는 코루틴이 어떤 연산을 수행하는 도중에만 실행되기를 바랄 수도 있다. 동시성 작업 사이의 부모 자식 관계로 인해 이런 실행 시간 제한이 가능하다. 어떤 코루틴을 다른 코루틴의 문맥에서 실행하면 후자가 전자의 부모가 된다. 이 경우 자식의 실행이 모두 끝나야 부모가 끝날 수 있도록 부모와 자식의 생명주기가 연관된다.

이런 기능을 구조적 동시성(structured concurrency)라고 부른다. 지역 변수 영역 안에서 블럭이나 서브루틴을 사용하는 경우와 구조적 동시성을 비교할 수 있다. 예제를 살펴보자.

<코드>

import kotlinx.coroutines.\*  
  
fun main() {  
 runBlocking {  
 println("Parent task started")  
   
 launch {  
 println("Task A started")  
 delay(200)  
 println("Task A finished")  
 }  
   
 launch {  
 println("Task B started")  
 delay(200)  
 println("Task B finished")  
 }  
   
 delay(100)  
 println("Parent task finished")  
 }  
 println("Shutting down...")  
}

</코드>

이 코드는 최상위 코루틴을 시작하고 현재 CoroutineScope 인스턴스 안에서 launch를 호출(영역 객체는 일시중단 람다의 수신객체로 전달된다)해서 두 가지 자식 코루틴을 시작한다. 이 프로그램을 실행하면 다음 결과를 볼 수 있다.

<코드>

Parent task started  
Task A started  
Task B started  
Parent task finished  
Task A finished  
Task B finished  
Shutting down...

</코드>

지연을 100밀리초만 줬기 때문에 runBlocking() 호출의 일시중단 람다로 이뤄진 부모 코루틴의 주 본문이 더 빨리 끝난다. 하지만 부모 코루틴 자체는 이 시점에 실행이 끝나지 않고 일시중단 상태로 두 자식이 모두 끝날 때까지 기다린다. runBlocking()이 메인 스레드를 블럭하고 있었기 때문에 부모 스레드가 끝나야 메인 스레드의 블럭이 풀리고 마지막 메시지가 출력된다.

coroutineScope() 호출로 코드 블럭을 감싸면 커스텀 영역을 도입할 수도 있다. runBlocking()과 마찬가지로 coroutineScope() 호출은 람다의 결과를 반환하고, 자식들이 완료되기 전까지 실행이 완료되지 않는다. coroutineScope()와 runBlocking()의 가장 큰 차이는 coroutineScope()가 일시중단 함수라 현재 스레드를 블럭시키지 않는다는 점이다.

<코드>

import kotlinx.coroutines.\*  
  
fun main() {  
 runBlocking {  
 println("Custom scope start")  
   
 coroutineScope {  
 launch {  
 delay(100)  
 println("Task 1 finished")  
 }  
   
 launch {  
 delay(100)  
 println("Task 2 finished")  
 }  
 }  
   
 println("Custom scope end")  
 }  
}

</코드>

앞의 coroutineScope() 호출이 자식 코루틴이 실행이 끝날 때까지 일시중단되기 때문에 Custom scope end 메시지가 마지막에 표시된다.

일반적으로 부모 자식 관계는 예외 처리와 취소 요청을 공유하는 영역을 정의하는 더 복잡한 코루틴 계층 구조를 만들어낼 수 있다. 이 주제에 대해서는 코루틴 잡(job)과 취소에 대해 설명할 때 다시 다루겠다.

## 코루틴 문맥

코루틴마다 CoroutineContext 인터페이스로 표현되는 문맥이 연관되어 있으며, 코루틴을 감싸는 변수 영역의 coroutineContext 프로퍼티를 통해 이 문맥에 접근할 수 있다. 문맥은 키-값 쌍으로 이뤄진 불변 컬렉션이며, 코루틴에서 사용할 수 있는 여러 가지 데이터가 들어있다. 이 데이터 중 일부는 코루틴 장치에 있어 특별한 의미를 가지며, 런타임에 코루틴이 실행되는 방식에 영향을 끼친다. 두 가지 요소에 주목해보자.

* 코루틴이 실행 중인 취소 가능한 작업을 표현하는 잡(job)
* 코루틴과 스레드의 연관을 제어하는 디스패처(dispatcher)

일반적으로 문맥은 CoroutineContext.Element를 구현하는 아무 데이터나 저장할 수 있다. 특정 원소에 접근하려면 get() 메서드나 인덱스 연산자에 키를 넘겨야 한다.

<코드>

GlobalScope.launch {  
 // 현재 잡을 얻고 “Task is active: true”를 출력  
 println("Task is active: ${coroutineContext[Job.Key]!!.isActive}")  
}

</코드>

기본적으로 launch(), async() 등의 표준 코루틴 빌더에 의해 만들어지는 코루틴은 현재 문맥을 이어받는다. 필요하면 빌더 함수에 context 파라미터를 지정해서 새 문맥을 넘길 수도 있다. 새 문맥을 만들려면 두 문맥의 데이터를 합쳐주는 plus() 함수/+ 연산자를 사용하거나, 주어진 키에 해당하는 원소를 문맥에서 제거해주는 minusKey() 함수를 사용하면 된다.

<코드>

import kotlinx.coroutines.\*  
  
private fun CoroutineScope.showName() {  
 println("Current coroutine: ${coroutineContext[CoroutineName]?.name}")  
}  
  
fun main() {  
 runBlocking {  
 showName() // Current coroutine: null  
 launch(coroutineContext + CoroutineName("Worker")) {  
 showName() // Current coroutine: Worker  
 }  
 }  
}

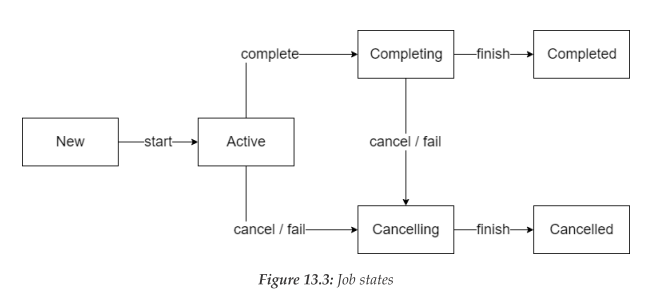
</코드>

코루틴을 실행하는 중간에 withContext()에 새 문맥과 일시중단 람다를 넘겨서 문맥을 전환시킬 수도 있다. 예를 들어 어떤 코드 블록을 다른 스레드에서 실행하고 싶을 때 이런 기능이 유용하다. 스레드 건너뛰기 기능은 코루틴 디스패처에 대해 다룰 때 예제를 살펴보겠다.

## 코루틴 흐름 제어와 잡 생명주기

잡은 동시성 작업의 생명주기를 표현하는 객체다. 잡을 사용하면 작업 상태를 추적하고 필요할 때 작업을 취소할 수 있다. 그림 13-3을 보면 잡이 취할 수 있는 상태에 대해 알 수 있다. 그림에서 각 상태의 의미와 잡의 상태가 한 상태에서 다른 상태로 어떻게 전이되는지에 대해 자세히 살펴보자.

그림 13-3: 잡 상태



[그림번역]

신규(New)

start - 시작

활성화(Active)

complete - 완료

finish - 끝

완료 중(Completing)

완료됨(Completed)

cancel/fail - 취소/실패

취소 중(Cancelling)

취소됨(Cancelled)

[번역끝]

</그림>

활성화 상태는 작업이 시작됐고 아직 완료나 취소로 끝나지 않았다는 뜻이다. 이 상태가 보통 디폴트 상태다. 즉, 잡은 생성되자마자 활성화 상태가 된다. launch()나 async()는 CoroutineStart 타입의 인자를 지정해서 잡의 초기 상태를 선택하는 기능을 제공하기도 한다.

* CoroutineStart.DEFAULT는 디폴트 동작이며, 잡을 즉시 시작한다.
* CoroutineStart.LAZY는 잡을 자동으로 시작하지 말라는 뜻이다. 이 경우에는 잡이 신규 상태가 되고 시작을 기다리게 된다.

신규 상태의 잡에 대해 start()나 join() 메서드를 호출하면 잡이 시작되면서 활성화 상태가 된다. 다음 예제를 보자.

<코드>

import kotlinx.coroutines.\*  
  
fun main() {  
 runBlocking {  
 val job = launch(start = CoroutineStart.LAZY) {  
 println("Job started")  
 }  
   
 delay(100)  
   
 println("Preparing to start...")  
 job.start()  
 }  
}

</코드>

이 예제는 자식 코루틴의 시작을 부모 코루틴이 메시지를 호출한 뒤로 미룬다. 출력은 다음과 같다.

<코드>

Preparing to start...  
Job started

</코드>

활성화 상태에서는 코루틴 장치가 잡을 반복적으로 일시중단 및 재개시킨다. 잡이 다른 잡을 시작할 수도 있는데, 이 경우 새 잡은 기존 잡의 자식이 된다. 따라서 잡의 부모 자식관계는 동시성 계산 사이에 트리 형태의 의존 구조를 만든다. children 프로퍼티를 통해 완료되지 않은 자식 잡들을 얻을 수 있다. 다음 코드를 보자.

<코드>

import kotlinx.coroutines.\*  
  
fun main() {  
 runBlocking {  
 val job = coroutineContext[Job.Key]!!  
   
 launch { println("This is task A") }  
 launch { println("This is task B") }  
   
 // 2 children running  
 println("${job.children.count()} children running")  
 }  
}

</코드>

코루틴이 일시중단 람다 블록의 실행을 끝내면 잡의 상태는 완료 중 상태로 바뀐다. 이 상태는 기본적으로 자식들의 완료를 기다리는 상태다. 잡은 모든 자식이 완료될 때까지 이 상태를 유지하고, 모든 자식이 완료되면 잡의 상태가 완료 중에서 완료됨으로 바뀐다.

잡의 join() 메서드를 사용하면 조인 대상 잡이 완료될 때까지 현재 코루틴을 일시중단시킬 수 있다. 다음 프로그램은 루트 코루틴의 메시지가 두 자식 메시지의 실행이 끝난 후에 출력되도록 보장해준다.

<코드>

import kotlinx.coroutines.\*  
  
fun main() {  
 runBlocking {  
 val job = coroutineContext[Job.Key]!!  
 val jobA = launch { println("This is task A") }  
 val jobB = launch { println("This is task B") }  
   
 jobA.join()  
 jobB.join()  
   
 println("${job.children.count()} children running")  
 }  
}

</코드>

결과는 다음과 같다.

<코드>

This is task A  
This is task B  
0 children running

</코드>

예상한 대로 job.children.count()를 평가하는 시점에는 활성화된 자식이 없다.

현재 잡 상태를 잡의 isActive, isCancelled, isComplete 프로퍼티로부터 추적할 수 있다. 이들의 의미를 다음 표에 정리했다. 이에 대한 정보는 Job 인터페이스 문서에서 볼 수 있다.

<표 13-1> Determining the current state by Job properties

|  |  |  |  |
| --- | --- | --- | --- |
| 잡 상태 | isActive | isCompleted | isCancelled |
| 신규 | false | false | false |
| 활성화 | true | false | false |
| 완료 중 | true | false | false |
| 취소 중 | false | false | true |
| 취소됨 | false | true | true |
| 완료됨 | false | true | false |

완료됨, 취소됨인 상태의 잡의 isCompleted가 true라는 점에 유의하라. isCancelled 프로퍼티를 검사하면 이들을 구분할 수 있다. 반면 잡 외부에서 활성화된 작업과 완료 중인 작업을 구분할 수는 없다.

## 취소

잡의 cancel() 메서드를 호출하면 잡을 취소할 수 있다. 이 메서드는 더 이상 필요 없는 계산을 중단시킬 수 있는 표준적인 방법을 제공한다. 취소에는 협력해야 한다. 즉, 취소 가능한 코루틴이 스스로 취소가 요청됐는지 검사해서 적절히 반응을 해줘야 한다. 다음 프로그램을 살펴보자.

<코드>

import kotlinx.coroutines.\*  
  
suspend fun main() {  
 val squarePrinter = GlobalScope.launch(Dispatchers.Default) {  
 var i = 1  
 while (true) {  
 println(i++)  
 }  
 }  
   
 delay(100) // 자식 잡이 어느 정도 실행될 시간을 준다  
 squarePrinter.cancel()  
}

</코드>

이 코드는 정수를 계속 출력하는 코루틴을 시작한다. 이 코루틴이 100밀리초 실행된 다음에 중단되게 했는데, 프로그램을 실행하면 squarePrinter가 계속 실행되는 모습을 볼 수 있다. 이유는 이 코루틴이 취소를 위해 협력하지 않기 때문이다. 이런 문제를 해결하는 방법으로, 다음 작업을 수행하기 전에 코루틴이 취소됐는지 검사하는 방법이 있다.

<코드>

import kotlinx.coroutines.\*  
  
suspend fun main() {  
 val squarePrinter = GlobalScope.launch(Dispatchers.Default) {  
 var i = 1  
 while (isActive) {  
 println(i++)  
 }  
 }  
   
 delay(100) // 자식 잡이 어느 정도 실행될 시간을 준다  
 squarePrinter.cancel()  
}

</코드>

isActive 확장 프로퍼티는 현재 잡이 활성화된 상태인지 검사한다. CoroutineScope(일시중단 람다의 수신 객체가 이 타입임)의 이 프로퍼티를 살펴보면 단순히 현재 잡의 isActive에게 위임하고 있다. 이제 부모 코루틴이 cancel() 메서드를 호출하면 squarePrinter의 상태가 취소 중으로 바뀌고, 그 다음 isActive 검사를 통해 루프를 종료시킬 수 있다. 코루틴이 실행을 끝내면 상태가 취소됨으로 바뀐다. 이 코드를 실행해보면 대략 100밀리초 이후 코루틴이 중단되는 모습을 볼 수 있다.

다른 방법은 상태를 검사하는 대신에, CancellatinoExcption을 발생시키면서 취소에 반응할 수 있게 일시중단 함수를 호출하는 것이다. 이는 잡을 취소하는 과정이 진행 중이라는 사실을 전달하는 토큰 역할을 하기 위해 코루틴 라이브러리 내부에서 쓰이는 예외다. 코루틴 라이브러리에 정의된 delay()나 join() 등의 모든 일시중단 함수가 이 예외를 발생시켜 준다. 한가지 예를 추가하면, yield()를 들 수 있다. 이 함수는 실행 중인 잡을 일시중단시켜서 자신을 실행 중인 스레드를 다른 코루틴에게 양보한다(마치 Thread.yield()를 호출하면 현재 스레드를 일시중단시키고 다른 스레드가 실행될 수 있는 기회를 부여하는 것과 비슷하다).

<코드>

import kotlinx.coroutines.\*  
  
suspend fun main() {  
 val squarePrinter = GlobalScope.launch(Dispatchers.Default) {  
 var i = 1  
 while (true) {  
 yield()  
 println(i++)  
 }  
 }  
 delay(100) // 자식 잡이 어느 정도 실행될 시간을 준다  
 squarePrinter.cancel()  
}

</코드>

부모 코루틴이 취소되면 자동으로 모든 자식의 실행을 취소한다. 이 과정은 부모에게 속한 모든 잡 계층이 취소될 때까지 계속된다. 다음 예제를 살펴보자.

<코드>

import kotlinx.coroutines.\*  
  
fun main() {  
 runBlocking {  
 val parentJob = launch {  
 println("Parent started")  
   
 launch {  
 println("Child 1 started")  
 delay(500)  
 println("Child 1 completed")  
 }  
   
 launch {  
 println("Child 2 started")  
 delay(500)  
 println("Child 2 completed")  
 }  
   
 delay(500)  
 println("Parent completed")  
 }  
   
 delay(100)  
 parentJob.cancel()  
 }  
}

</코드>

이 프로그램은 한 쌍의 자식을 시작하는 코투틴을 시작한다. 이렇게 만들어진 세 가지 작업은 모두 완료 메시지를 표시하기 전에 500밀리초를 기다린다. 하지만 부모 잡이 100밀리초만에 취소된다. 그 결과 세 잡 중 어느 하나도 완료 상태에 도달하지 못하며, 프로그램은 다음과 같이 출력된다.

<코드>

Parent started  
Child 1 started  
Child 2 started

</코드>

## 타임아웃

경우에 따라 작업이 완료되기를 무작정 기다릴 수 없어서 타임아웃을 설정해야 할 때가 있다. 코루틴 라이브러리는 정확히 이럴 때 사용할 수 있는 withTimeout()이라는 함수를 제공한다. 다음 예제는 파일을 읽는 도중에 타임아웃이 걸린 코루틴을 시작한다.

<코드>

import kotlinx.coroutines.\*  
import java.io.File  
  
fun main() {  
 runBlocking {  
 val asyncData = async { File("data.txt").readText() }  
 try {  
 val text = withTimeout(50) { asyncData.await() }  
 println("Data loaded: $text")  
 } catch (e: Exception) {  
 println("Timeout exceeded")  
 }  
 }  
}

</코드>

파일을 50밀리초 안에 읽을 수 있다면 withTimeout()은 결과를 돌려주기만 한다. 하지만 50밀리초 안에 읽을 수 없으면 TimeoutCancellationException을 던지기 때문에 파일을 읽는 코루틴이 취소된다.

비슷한 함수로 withTimeoutOrNull()이라는 것도 있다. 이 함수는 타임아웃이 발생하면 예외를 던지는 대신 널 값을 돌려준다.

## 코루틴 디스패치하기

코루틴은 스레드와 무관하게 일시중단 가능한 계산을 구현할 수 있게 해주지만, 코루틴을 실행하려면 여전히 스레드와 연관을 시켜야 한다. 코루틴 라이브러리에는 특정 코루틴을 실행할 때 사용할 스레드를 제어하는 작업을 담당하는 특별한 컴포넌트가 있다. 이 컴포넌트를 코루틴 디스패처(dispatcher)라고 부른다.

디스패처는 코루틴 문맥의 일부다. 따라서 launch()나 runBlocking() 등의 코루틴 빌더 함수에서 이를 지정할 수 있다. 디스패처는 그 자체로 원소가 하나뿐인 문맥이기도 하므로, 코루틴 빌더에게 디스패처를 넘길 수 있다.

<코드>

import kotlinx.coroutines.\*  
  
fun main() {  
 runBlocking {  
 // 전역 스레드 풀 디스패처를 사용해 코루틴을 실행한다  
 launch(Dispatchers.Default) {  
 println(Thread.currentThread().name) // DefaultDispatcher-worker-1  
 }  
 }  
}

</코드>

코루틴 디스패처는 병렬 작업 사이에 스레드를 배분해주는 자바 실행기(executor)와 비슷하다. 실제로 asCoroutineDispatcher() 확장 함수를 사용하면 기존 실행기 구현을 그에 상응하는 코루틴 디스패처로 쉽게 바꿀 수 있다. 다음 예제는 실행하는 스레드에 WorkerThread1, WorkerThread2 등의 이름을 부여하는 커스텀 스레드 팩토리를 사용하는 풀 기반의 실행기 서비스를 정의한다. 명시적으로 작업자 스레드를 데몬 스레드로 지정해서 코루틴 실행이 끝난 후에 프로그램이 종료되지 못하는 일이 없게 했다는 점에 유의하라.

<코드>

import kotlinx.coroutines.\*  
import java.util.concurrent.ScheduledThreadPoolExecutor  
import java.util.concurrent.atomic.AtomicInteger  
  
fun main() {  
 val id = AtomicInteger(0)  
   
 val executor = ScheduledThreadPoolExecutor(5) { runnable ->  
 Thread(  
 runnable,  
 "WorkerThread-${id.incrementAndGet()}"  
 ).also { it.isDaemon = true }  
 }  
   
 executor.asCoroutineDispatcher().use { dispatcher ->  
 runBlocking {  
 for (i in 1..3) {  
 launch(dispatcher) {  
 println(Thread.currentThread().name)  
 delay(1000)  
 }  
 }  
 }  
 }  
}

</코드>

delay()는 실행기가 별도의 스레드를 만들게 한다. 따라서 이 코드는 다음과 같은 결과를 출력한다.

<코드>

WorkerThread-1  
WorkerThread-2  
WorkerThread-3

</코드>

하지만 구체적인 스레드 순서는 다를 수 있다.

ExecutorService의 인스턴스에 대해 asCoroutineDispatcher() 호출하면 ExecutorCoroutineDispatcher를 반환하는데, 이 디스패처는 Closeable 인스턴스도 구현한다. 이 디스패처의 바탕이 되는 실행기 서비스를 종료하고 스레드를 유지하기 위해 할당했던 시스템 자원을 해제하려면 close() 함수를 직접 호출하거나, 방금 본 코드처럼 use() 함수 블록 안에서 디스패처를 사용해야 한다.

코루틴 라이브러리에는 기본적으로 몇 가지 디스패처 구현을 제공한다. 그 중 일부를 Dispatchers 객체를 통해 사용할 수 있다.

* Dispatchers.Default : 공유 스레드 풀로, 풀 크기는 디폴트로 사용 가능한 CPU 코어 수이거나 2(둘 중 큰 값)다. 이 구현은 일반적으로 작업 성능이 주로 CPU 속도에 의해 결정되는 CPU 위주의 작업에 적합하다.
* Dispatchers.IO : 스레드 풀 기반이며 디폴트 구현과 비슷하지만, 파일을 읽고 쓰는 것처럼 잠재적으로 블러킹될 수 있는 I/O를 많이 사용하는 작업에 최적화되어 있다. 이 디스패처는 스레드 풀을 디폴트 구현과 함께 공유하지만, 필요에 따라 스레드를 추가하거나 종료시켜준다.
* Dispatchers.Main : 사용자 입력이 처리되는 UI 스레드에서만 배타적으로 작동하는 디스패처다.

newFixedThreadPoolContext()나 newSingleThreadPoolContext()를 사용하면 직접 만든 스레드 풀을 사용하거나 심지어는 스레드를 하나만 사용하는 디스패처도 만들 수 있다. 예를 들어 Executor를 기반으로 예제를 다시 작성할 수 있다.

<코드>

import kotlinx.coroutines.\*  
  
@Suppress("EXPERIMENTAL\_API\_USAGE")  
fun main() {  
 newFixedThreadPoolContext(5, "WorkerThread").use { dispatcher ->  
 runBlocking {  
 for (i in 1..3) {  
 launch(dispatcher) {  
 println(Thread.currentThread().name)  
 delay(1000)  
 }  
 }  
 }  
 }  
}

</코드>

newFixedThreadPoolContext()나 newSingleThreadPoolContext()가 실험적인 기능으로 표시되어 있고, 향후 공유 스레드 풀을 기반으로 하는 더 새로운 함수로 대치될 예정이기 때문에, 경고를 피하기 위해 @Suppress 애너테이션을 사용했다.

(이 장 앞의 예제에서 했던 것처럼) 디스패처를 명시적으로 지정하지 않으면 여러분이 코루틴을 시작한 영역으로부터 디스패처가 자동으로 상속된다. 다음 예제를 생각해 보자.

<코드>

import kotlinx.coroutines.\*  
  
fun main() {  
 runBlocking {  
 println("Root: ${Thread.currentThread().name}")  
   
 launch {  
 println("Nested, inherited: ${Thread.currentThread().name}")  
 }  
   
 launch(Dispatchers.Default) {  
 println("Nested, explicit: ${Thread.currentThread().name}")  
 }  
 }  
}

</코드>

이 코드는 메인 스레드에서 실행되는 최상위 코루틴을 시작한다. 이 코루틴은 내부에 두 가지 코루틴을 포함한다. 첫 번째 코루틴은 문맥(그리고 그에 따라 코루틴 디스패처도)을 부모 코루틴으로부터 물려받고, 두 번째 코루틴은 명시적으로 문맥을 전달받는다. 따라서 이 코드는 다음과 같은 출력을 내놓는다.

<코드>

Root: main  
Nested, explicit: DefaultDispatcher-worker-1  
Nested, inherited: main

</코드>

부모 코루틴이 없으면 암시적으로 Dispatchers.Default로 디스패처를 가정한다. 다만 runBlocking() 빌더는 현재 스레드를 사용한다.

코루틴이 생명주기 내내 같은 디스패처를 사용할 필요도 없다. 디스패처가 코루틴 문맥의 일부이기 때문에, withContext() 함수를 사용해 디스패처를 오버라이드할 수 있다.

<코드>

import kotlinx.coroutines.\*  
  
@Suppress("EXPERIMENTAL\_API\_USAGE")  
fun main() {  
 newSingleThreadContext("Worker").use { worker ->  
 runBlocking {  
 println(Thread.currentThread().name) // main  
 withContext(worker) {  
 println(Thread.currentThread().name) // Worker  
 }  
 println(Thread.currentThread().name) // main  
 }  
 }  
}

</코드>

이 기법은 중단가능 루틴의 일부를 한 스레드에서만 실행하고 싶을 때 유용하다.

## 예외 처리

예외 처리의 경우, 코루틴 빌더들은 두 가지 기본 전략 중 하나를 따른다. 첫 번째는 launch() 같은 빌더가 선택한 전략으로, 예외를 부모 코루틴에게 전달하는 전략이다. 이 경우 예외는 다음과 같이 전파된다.

* 부모 코루틴이 (자식에게서 발생한 오류와) 똑같은 오류로 취소된다. 이로 인해 부모의 나머지 자식도 모두 취소된다.
* 자식들이 모두 취소되고 나면 부모는 예외를 코루틴 트리의 윗부분으로 전달한다.

전역 영역에 있는 코루틴에 도달할 때까지 이 과정이 반복된다. 그 후 예외가 CoroutineExceptionHandler.Consider에 의해 처리된다. 예를 들어 다음 코드를 보자.

<코드>

import kotlinx.coroutines.\*  
  
fun main() {  
 runBlocking {  
 launch {  
 throw Exception("Error in task A")  
 println("Task A completed")  
 }  
   
 launch {  
 delay(1000)  
 println("Task B completed")  
 }  
   
 println("Root")  
 }  
}

</코드>

최상위 코루틴은 한 쌍의 내부 작업을 시작한다. 그 중 첫 번째 코루틴은 예외를 던진다. 이로 인해 최상위 작업이 취소되고, 최상위의 자식인 두 작업도 취소된다. 그리고 최상위에서 아무 커스텀 핸들러도 지정하지 않았기 때문에 프로그램은 Thread.uncaughtExceptionHandler에 등록된 디폴트 동작을 실행한다. 그에 따라 다음과 같은 메시지가 표시되고 그 뒤에 예외에 대한 스텍 트레이스(trace)가 출력된다.

<코드>

Root  
Exception in thread "main" java.lang.Exception: Error in task A

</코드>

CoroutineExceptionHandler는 현재 코루틴 문맥과 던져진 예외를 인자로 전달받는다.

<코드>

fun handleException(context: CoroutineContext, exception: Throwable)

</코드>

핸들러를 만드는 가장 간단한 방법은 인자가 두 개인 람다를 받는 CoroutineExceptionHandler() 함수를 쓰는 것이다.

<코드>

val handler = CoroutineExceptionHandler{ \_, exception ->  
 println("Caught $exception")  
}

</코드>

이 핸들러의 인스턴스가 예외를 처리하도록 지정하려면 코루틴 문맥에 인스턴스를 넣어야 한다. 핸들러 자체는 뻔한 문맥이기 때문에, 그냥 코루틴 빌더의 context 인자로 핸들러를 넘길 수 있다.

<코드>

import kotlinx.coroutines.\*  
  
suspend fun main() {  
 val handler = CoroutineExceptionHandler{ \_, exception ->  
 println("Caught $exception")  
 }  
   
 GlobalScope.launch(handler) {  
 launch {  
 throw Exception("Error in task A")  
 println("Task A completed")  
 }  
   
 launch {  
 delay(1000)  
 println("Task B completed")  
 }  
   
 println("Root")  
 }.join()  
}

</코드>

이 프로그램이 다음 메시지를 출력한다. 이 말은 프로그램이 예외에 대한 디폴트 동작을 오버라이드했다는 뜻이다.

<코드>

Root  
Caught java.lang.Exception: Error in task A

</코드>

문맥에 핸들러 인스턴스 정의가 없는 경우, 코루틴 라이브러리는 JVM ServiceLoader 장치를 통해 설정된 모든 전역 핸들러를 호출하고 현재 스레드에 대해서는 uncaughtExceptionHandler를 발생시킨다.

CoroutineExceptionHandler는 전역 영역에서 실행된 코루틴에 대해서만 정의할 수 있고, CoroutineExceptionHandler이 정의된 코루틴의 자식에 대해서만 적용된다. 그래서 runBlocking() 코드를 GlobalScope.launch()로 변경하고 main() 함수를 suspend로 표시하고 join() 호출을 사용해야 한다. 원래 예제에서 runBlocking()을 그대로 사용하는 대신 핸들러만 추가한다고 해도, 코루틴이 전역 영역에서 실행되지 않기 때문에 프로그램은 여전히 디폴트 핸들러를 사용하게 된다.

<코드>

import kotlinx.coroutines.\*  
  
fun main() {  
 val handler = ...  
 runBlocking(handler) {  
 ...  
 }  
}

</코드>

예외를 처리하는 다른 방법은 async() 빌더에서 사용하는 방법으로, 던져진 예외를 저장했다가 예외가 발생한 계산에 대한 await() 호출을 받았을 때 다시 던지는 것이다. 앞의 예제를 조금 바꿔보자.

<코드>

import kotlinx.coroutines.\*  
  
fun main() {  
 runBlocking {  
 val deferredA = async {  
 throw Exception("Error in task A")  
 println("Task A completed")  
 }  
 val deferredB = async {  
 println("Task B completed")  
 }  
 deferredA.await()  
 deferredB.await()  
 println("Root")  
 }  
}

</코드>

출력은 다음과 같다.

<코드>

Exception in thread "main" java.lang.Exception: Error in task A

</코드>

deferredA.await()에서 예외가 다시 던져지므로 프로그램이 println("Root") 문장을 실행하지 못하기 때문이다.

코루틴 데이터에 접근할 때 예외를 다시 던지는 방식을 채용하는 async와 유사한 빌더들의 경우 CoroutineExceptionHandler를 사용하지 않는다. 따라서 코루틴 문맥에 CoroutineExceptionHandler 인스턴스를 설정했더라도 아무 효과가 없다. 그냥 전역 디폴트 핸들러가 호출된다.

내포된 코루틴에서 발생한 예외를 전역 핸들러를 통하지 않고 부모 수준에서 처리하고 싶으면 어떻게 해야 할까? try-catch 블록으로 예외를 처리하려고 시도하면 어떤 일이 벌어지는지 살펴보자.

<코드>

import kotlinx.coroutines.\*  
  
fun main() {  
 runBlocking {  
 val deferredA = async {  
 throw Exception("Error in task A")  
 println("Task A completed")  
 }  
   
 val deferredB = async {  
 println("Task B completed")  
 }  
   
 try {  
 deferredA.await()  
 deferredB.await()  
 } catch (e: Exception) {  
 println("Caught $e")  
 }  
 println("Root")  
 }  
}

</코드>

이 코드를 실행하면 핸들러가 실제 호출되는 모습을 볼 수 있다. 하지만 프로그램은 여전히 예외와 함께 중단된다.

<코드>

Caught java.lang.Exception: Error in task A  
Root  
Exception in thread "main" java.lang.Exception: Error in task A

</코드>

이유는 자식(여기서는 deeferredA)이 실패한 경우에는 부모를 취소시키기 위해 자동으로 예외를 다시 던지기 때문이다. 이 동작을 변경하려면 수퍼바이저(supervisor) 잡을 사용해야 한다.

수버파이저 잡이 있으면 취소가 아래 방향으로만 전달된다. 여러분이 수퍼바이저를 취소하면 수퍼바이저 잡은 자동으로 자신의 모든 자식을 취소한다. 하지만 수퍼바이저가 아니라 자식이 취소된 경우, 수퍼바이저나 수퍼바이저의 다른 자식들은 아무 영향을 받지 않는다.

부모 코루틴을 수퍼바이저로 변환하려면 coroutineScope() 대신 supervisorScope() 함수를 사용해 새로운 영역을 정의해야 한다. 앞의 예제를 변경해보자.

<코드>

import kotlinx.coroutines.\*  
  
fun main() {  
 runBlocking {  
 supervisorScope {  
 val deferredA = async {  
 throw Exception("Error in task A")  
 println("Task A completed")  
 }  
   
 val deferredB = async {  
 println("Task B completed")  
 }  
   
 try {  
 deferredA.await()  
 } catch (e: Exception) {  
 println("Caught $e")  
 }  
 deferredB.await()  
 println("Root")  
 }  
 }  
}

</코드>

이제 예외가 발생하더라도 B 작업과 루트 코루틴이 완료된다.

<코드>

Task B completed  
Caught java.lang.Exception: Error in task A  
Root

</코드>

수퍼바이저의 동작은 일반적인 취소에도 적용된다는 점에 유의하라. 수퍼바이저의 자식 중 하나에 cancel()을 호출해도 해당 코루틴의 형제자매나 수퍼바이저 자신에는 아무 영향이 없다.

## 동시성 통신

이 절에서는 스레드 안전성을 유지하면서 여러 동시성 작업 사이에 효율적으로 데이터를 공유할 수 있게 해주는 코루틴 라이브러리의 고급 기능에 대해 이야기한다. 정확히 말해 여기서는 코루틴과 액터 사이에서 동기화나 락을 사용하지 않고도 변경 가능한 상태를 안전하게 공유할 수 있는 데이터 스트림을 제공하는 메커니즘인 채널(channel)에 대해 다룰 것이다.

### 채널

채널은 임의의 데이터 스트림을 코루틴 사이에 공유할 수 있는 편리한 방법이다. Channel 인터페이스가 제공하는 채널에 대한 기본 연산은 데이터를 보내는 send() 메서드와 데이터를 받는 receive() 메서드다. 이 메서드는 자신의 작업을 완료할 수 없다. 예를 들어 채널 내부 버퍼가 꽉 찼는데 데이터를 채널에 보내려고 하면 채널은 현재 코루틴을 일시중단시키고 나중에 처리가 가능할 때 재개한다. 이 부분이 자바의 동시성 API에서 채널과 비슷한 역할을 하는 블러킹 큐(queue)와 채널의 가장 큰 차이다. 블러킹 큐는 스레드를 블럭시킨다.

제네릭 Channel() 함수를 사용해 채널을 만들 수 있다. 이 함수는 채널의 용량을 지정하는 정수 값을 받는다. 채널 기본 구현은 크기가 정해진 내부 버퍼를 사용한다. 버퍼가 꽉 차면 최소 하나 이상의 채널 원소가 상대방에 의해 수신될 때까지 send() 호출이 일시중단된다. 비슷하게 버퍼가 비어있으면 누군가 최소 하나 이상의 원소를 채널로 송신할 때까지 receive() 호출이 일시중단된다. 예제를 살펴보자.

<코드>

import kotlinx.coroutines.channels.Channel  
import kotlinx.coroutines.\*  
import kotlin.random.Random  
  
fun main() {  
 runBlocking {  
 val streamSize = 5  
 val channel = Channel<Int>(3)  
   
 launch {  
 for (n in 1..streamSize) {  
 delay(Random.nextLong(100))  
 val square = n\*n  
 println("Sending: $square")  
 channel.send(square)  
 }  
 }  
   
 launch {  
 for (i in 1..streamSize) {  
 delay(Random.nextLong(100))  
 val n = channel.receive()  
 println("Receiving: $n")  
 }  
 }  
 }  
}

</코드>

첫 번째 코루틴은 정수 제곱 값의 스트림을 만들어내고, 원소를 세 개 저장할 수 있는 채널에 이 스트림을 전송한다. 이와 동시에 두 번째 코루틴이 생성된 수를 수신한다. 두 코루틴 중 어느 한 쪽이 뒤쳐져서 채널 버퍼가 꽉 차거나 비는 경우가 생겨서 일시 중단이 벌어질 수 있게 일부러 지연 시간을 난수로 지정했다. 다양한 출력이 생길 수 있는데, 그 중 하나는 다음과 같다.

<코드>

Sending: 1  
Receiving: 1  
Sending: 4  
Receiving: 4  
Sending: 9  
Sending: 16  
Receiving: 9  
Sending: 25  
Receiving: 16  
Receiving: 25

</코드>

출력이 실제 지연 시간 값이나 다른 환경에 의해 달라질 수 있지만, 채널은 모든 값이 송신된 순서 그대로 수신되도록 보장한다.

Channel() 함수는 채널의 동작을 바꿀 수 있는 여러 특별한 값을 받을 수 있다. 이러한 값은 Channel 인터페이스의 동반 객체에 상수로 정의되어 있다.

* Channel.UNLIMITED (= Int.MAX\_VALUE): 이 경우 채널의 용량이 제한이 없고, 내부 버퍼는 필요에 따라 증가한다. 이런 채널은 send()시에 결코 일시중단되는 일이 없다. 다만 receive()를 하는 경우 버퍼가 비어있으면 일시중단될 수 있다.
* Channel.RENDEZVOUS (= 0): 이 경우 채널은 아무 내부 버퍼가 없는 랑데뷰 채널이 된다. send() 호출은 다른 어떤 코루틴이 receive()를 호출할 때까지 항상 일시중단된다. 마찬가지로 receive() 호출은 다른 어떤 코루틴이 send()를 호출할 때까지 일시중단된다. 채널 생성 시 capacity를 지정하지 않으면 이 방식의 채널이 생성된다.
* Channel.CONFLATED (= -1): 이 경우에는 송신된 값이 합쳐지는 채널(conflated channel)이다. 이 말은 send()로 보낸 원소를 최대 하나만 버퍼에 저장하고 이 값이 누군가에 의해 수신되기 전에 다른 send() 요청이 오면 기존에 값을 덮어쓴다는 말이다. 따라서 수신되지 못한 원소 값은 소실된다. 이 채널의 경우 send() 메서드는 결코 일시중단되지 않는다.
* Channel.UNLIMITED보다 작은 임의의 양수를 지정하면 버퍼 크기가 일정하게 제한된 채널이 생긴다.

랑데뷰 채널은 생산자와 소비자 코루틴이 교대로 활성화되도록 보장한다. 예를 들어 앞 예제의 채널 크기를 0으로 바꾸면 딜레이 시간과 관계 없이 안정적인 동작 순서를 볼 수 있다.

<코드>

Sending: 1  
Receiving: 1  
Sending: 4  
Receiving: 4  
Sending: 9  
Receiving: 9  
Sending: 16  
Receiving: 16  
Sending: 25  
Receiving: 25

</코드>

값이 합쳐지는 채널은 스트림에 기록한 모든 원소가 도착할 필요가 없고 소비자 코루틴이 뒤쳐지는 경우 생산자가 만들어낸 값 중 일부를 버려도 되는 경우 쓰인다. 첫 번째 예제의 소비자 지연 시간을 생산자 지연 시간의 2배로 설정해보자.

<코드>

import kotlinx.coroutines.channels.Channel  
import kotlinx.coroutines.\*  
  
fun main() {  
 runBlocking {  
 val streamSize = 5  
 val channel = Channel<Int>(Channel.CONFLATED)  
   
 launch {  
 for (n in 1..streamSize) {  
 delay(100)  
 val square = n\*n  
 println("Sending: $square")  
 channel.send(square)  
 }  
 }  
   
 launch {  
 for (i in 1..streamSize) {  
 delay(200)  
 val n = channel.receive()  
 println("Receiving: $n")  
 }  
 }  
 }  
}

</코드>

실행해본 결과, 생산된 값 중 대략 절반만 수신되어 처리되며, 다음과 같은 출력이 나올 수 있다.

<코드>

Sending: 1  
Receiving: 1  
Sending: 4  
Sending: 9  
Receiving: 9  
Sending: 16  
Sending: 25  
Receiving: 25

</코드>

앞의 프로그램을 실행하면 마지막 줄을 출력한 다음에도 프로그램이 끝나지 않는다. 이유는 수신자 코루틴이 1부터 streamSize까지 이터레이션하므로 수신하리라 기대하는 원소 개수가 5개이기 때문이다. 하지만 실제 수신되는 원소는 streamSize/2 근처이기 때문에 이 기대는 결코 충족될 수가 없다. 이 상황에서 필요한 것은 채널이 닫혀서 더 이상 데이터를 보내지 않는다는 사실을 알려주는 일종의 신호다. Channel API는 생산자 쪽에서 close() 메서드를 사용해 이런 신호를 보낼 수 있게 해준다. 소비자 쪽에서는 반복 횟수를 고정하는 대신 채널에서 들어오는 데이터에 대해 이터레이션을 할 수 있다.

<코드>

import kotlinx.coroutines.channels.Channel  
import kotlinx.coroutines.\*  
  
fun main() {  
 runBlocking {  
 val streamSize = 5  
 val channel = Channel<Int>(Channel.CONFLATED)  
 launch {  
 for (n in 1..streamSize) {  
 delay(100)  
 val square = n\*n  
 println("Sending: $square")  
 channel.send(square)  
 }  
 channel.close()  
 }  
 launch {  
 for (n in channel) {  
 println("Receiving: $n")  
 delay(200)  
 }  
 }  
 }  
}

</코드>

이제 데이터 교환이 완료된 후 프로그램이 제대로 끝난다.

소비자 쪽에서는 명시적인 이터레이션을 사용하지 않고 consumeEach() 함수를 통해 모든 채널 콘텐트를 얻어서 사용할 수 있다.

<코드>

channel.consumeEach {  
 println("Receiving: $n")  
 delay(200)  
}

</코드>

채널이 닫힌 후 send()를 호출하면 ClosedSendChannelException 예외가 발생하며 실패한다. 채널이 닫힌 후 receive()를 호출하면 버퍼에 있는 원소가 소진될 때까지 정상적으로 원소가 반환되지만, 그 다음에는 마찬가지로 ClosedSendChannelException 예외가 발생한다.

채널 통신에 참여하는 생산자와 소비자가 꼭 하나씩일 필요는 없다. 예를 들어 한 채널을 여러 코루틴이 동시에 읽을 수도 있다. 이런 경우를 팬 아웃(fan out)이라고 한다.

<코드>

import kotlinx.coroutines.channels.Channel  
import kotlinx.coroutines.\*  
import kotlin.random.Random  
  
fun main() {  
 runBlocking {  
 val streamSize = 5  
 val channel = Channel<Int>(2)  
   
 launch {  
 for (n in 1..streamSize) {  
 val square = n\*n  
 println("Sending: $square")  
 channel.send(square)  
 }  
 channel.close()  
 }  
   
 for (i in 1..3) {  
 launch {  
 for (n in channel) {  
 println("Receiving by consumer #$i: $n")  
 delay(Random.nextLong(100))  
 }  
 }  
 }  
 }  
}

</코드>

생산자 코루틴이 생성한 데이터 스트림을 세 소비자 코루틴이 나눠 받는다. 따라서 다음과 같은 출력이 나올 수 있다.

<코드>

Sending: 1  
Sending: 4  
Sending: 9  
Receiving by consumer #1: 1  
Receiving by consumer #2: 4  
Receiving by consumer #3: 9  
Sending: 16  
Sending: 25  
Receiving by consumer #3: 16  
Receiving by consumer #1: 25

</코드>

마찬가지로 여러 생산자 코루틴이 한 채널에 써 넣은 데이터를 한 소비자 코루틴이 읽는 팬 인(fan in)도 있다. 좀 더 일반적인 경우라면 여러 생산자와 여러 소비자가 여러 채널을 공유할 수도 있다. 일반적으로 채널의 동작은 공정(fair)하다. 즉, 어떤 채널에 대해 receive()를 맨 처음 호출한 코루틴이 다음 원소를 읽게 된다.

### 생산자

앞에서 컬렉션 API에 대해 설명할 때 보여준 sequence() 함수와 비슷하게 동시성 데이터 스트림을 생성할 수 있는 producer()라는 특별한 코루틴 빌더가 있다. 이 빌더는 채널과 비슷한 send() 메서드를 제공하는 ProducerScope 영역을 도입해준다.

<코드>

import kotlinx.coroutines.channels.\*  
import kotlinx.coroutines.\*  
  
fun main() {  
 runBlocking {  
 val channel = produce {  
 for (n in 1..5) {  
 val square = n\*n  
 println("Sending: $square")  
 send(square)  
 }  
 }  
   
 launch {  
 channel.consumeEach { println("Receiving: $it") }  
 }  
 }  
}

</코드>

이 경우 채널을 명시적으로 닫을 필요가 없다. 코루틴이 종료되면 producer() 빌더가 채널을 자동으로 닫아준다.

예외 처리 관점에서 볼 때, producer()는 async()/await()의 정책을 따른다. producer() 안에서 예외가 발생하면 예외를 저장했다가 해당 채널에 대해 receive()를 가장 처음 호출한 코루틴 쪽에 예외가 다시 던져진다.

### 티커

coroutines 라이브러리에는 티커(ticker)라고 하는 특별한 랑데뷰 채널이 있다. 이 채널은 Unit 값을 계속 발생시키되 한 원소와 다음 원소의 발생 시점이 주어진 지연 시간만큼 떨어져있는 스트림을 만든다. 이 채널을 만들려면 ticker() 함수를 사용해야 한다. 이 함수를 호출할 때 다음을 지정할 수 있다.

* delayMillis: 티커 원소의 발생 시간 간격을 밀리초 단위로 지정한다
* initialDelayMillis: 티커 생성 시점부터 최초 발생하는 원소까지의 시간 간격이다. 디폴트 값은 delayMillis와 같다.
* context: 티커를 실행할 코루틴 문맥이다(디폴트는 빈 문맥이다).
* mode: 티커의 행동을 결정하는 TickerMode 이넘이다.
  + TickerMode.FIXED\_PERIOD : 생성되는 원소 사이의 시간 간격을 지정된 지연 시간에 최대한 맞추기 위해 실제 지연 시간을 조정한다.
  + TickerMode.FIXED\_RATE : 실제 흘러간 시간과 관계 없이 delayMillis로 지정한 지연 시간만큼 시간을 지연시킨 후 다음 원소를 송신한다.

티커 모드의 차이를 알아보기 위해 다음 코드를 살펴보자.

<코드>

import kotlinx.coroutines.\*  
import kotlinx.coroutines.channels.\*  
  
fun main() = runBlocking {  
 val ticker = ticker(100)  
 println(withTimeoutOrNull(50) { ticker.receive() })  
 println(withTimeoutOrNull(60) { ticker.receive() })  
 delay(250)  
 println(withTimeoutOrNull(1) { ticker.receive() })  
 println(withTimeoutOrNull(60) { ticker.receive() })  
 println(withTimeoutOrNull(60) { ticker.receive() })  
}

</코드>

실행하면 다음 출력을 볼 수 있다.

<코드>

null  
kotlin.Unit  
kotlin.Unit  
kotlin.Unit  
null

</코드>

이 코드의 실행 과정을 단계별로 살펴보자.

1. 50밀리초 간격으로 티커 신호를 받으려고 시도한다. 티커 지연 시간이 100밀리초이므로 withTimeoutOrNull()은 신호를 받지 못하고 타임아웃이 걸려서 널을 반환한다.
2. 그 후 다음 60밀리초 안에 신호를 받으려고 시도한다. 이번에는 티커가 시작한 지 100밀리초가 확실히 지날 것이기 때문에 널이 아닌 확실히 결과를 얻는다. receive()가 호출되면 티커가 재개된다.
3. 그 후 소비자 코루틴이 약 250밀리초 동안 일시중단된다. 100밀리초 후에 티커는 다른 신호를 보내고 신호가 수신될 때까지 일시중단된다. 이후 소비자와 티커 코루틴 모두 150밀리초동안 일시중단 상태로 남는다.
4. 소비자 코루틴이 재개되고 신호를 요청하려고 시도한다. 신호가 이미 보내졌기 때문에 receive()는 즉시 결과를 반환한다(그래서 타임아웃을 1밀리초로 작게 잡았다). 이제 티커는 마지막 신호를 보낸 이후 얼마나 시간이 지났는지 검사하고, 250밀리초가 지났다는 사실을 알게 된다. 이는 두 번의 완전한 주기(200밀리초)와 50밀리초의 나머지에 해당한다. 티커는 자신의 대기 시간을 조정해서 다음 신호를 보낼 때까지의 지연 시간을 100 - 50 = 50밀리초로 줄인다. 이렇게 줄이는 이유는 전체 지연 시간(100밀리초)을 지난 후 새 신호를 보내기 위해서다.
5. 다음으로 신호를 받으려는 receive() 호출은 거의 즉시 응답을 받는다. 따라서 티커는 전체 지연 시간(100밀리초)을 다시 기다린다. 그 결과 마지막 receive() 호출은 60밀리초 타임아웃 안에 티커로부터 신호를 받지 못하기 때문에 널을 받는다.

티커 모드를 FIXED\_RATE로 고정하면 결과가 다음과 같이 바뀐다.

<코드>

null  
kotlin.Unit  
kotlin.Unit  
null  
kotlin.Unit

</코드>

초반부는 앞의 예제(디폴트 티커 모드 FIXED\_PERIOD)와 거의 비슷하게 진행된다. 하지만 250밀리초의 긴 지연 이후 소비자 코루틴이 재개될 때부터는 동작이 달라진다. 티커가 250밀리초 지연 사이에 이미 신호를 보냈기 때문에 세 번째 receive() 호출은 즉시 반환된다. 하지만 이제 티커는 실제 흐른 시간을 고려하지 않고 그냥 100밀리초를 기다린다. 따라서 네 번째 receive()는 60밀리초 안에 신호를 받지 못하고 널을 반환한다. 반대로 다섯 번째 호출의 경우에는 중간에 신호가 발생하기 때문에 신호를 정상적으로 받는다.

티커 관련 API는 현재 실험 단계이며 미래의 코루틴 라이브러리 버전에서는 언제든 다른 내용으로 바뀔 수 있다는 점에 유의하라.

## 액터

가변 상태를 스레드 안전하게 공유하는 방법을 구현하는 일반적인 방법으로 액터 모델이 있다. 액터(actor)는 내부 상태와 다른 액터에게 메시지를 보내서 동시성 통신을 진행할 수 있는 수단을 제공하는 객체다. 액터는 자신에게 들어오는 메시지를 리슨(listen)하고, 자신의 상태를 바꾸면서 메시지에 응답할 수 있으며, 다른 메시지를 (자기 자신이나 다른 액터에게) 보낼 수 있고, 새로운 액터를 시작할 수 있다. 액터의 상태는 액터 내부에 감춰져 있기 때문에 다른 액터가 직접 이 상태에 접근할 수 없다. 다른 액터는 단지 메시지를 보내고 응답을 받아서 상태를 알 수 있을 뿐이다. 따라서 액터 모델은 락 기반의 동기화와 관련한 여러 가지 문제로부터 자유로울 수 있다.

코틀린 코루틴 라이브러리에서는 actor() 코루틴 빌더를 사용해 액터를 만들 수 있다. 액터는 특별한 영역(ActorScope)을 만들며 이 영역은 기본 코루틴 영역에 자신에게 들어오는 메시지에 접근할 수 있는 수신자 채널이 추가된 것이다. actor() 빌더는 결과를 생성해내는 것이 목적이 아닌 잡을 시작한다는 점에서는 launch()와 비슷하지만 CoroutineExceptionHandler에 의존하는 launch()와 같은 예외 처리 정책을 따른다.

액터 API의 기본적인 사용법을 알아보겠다. 다음 예제를 통해 은행 계좌 잔고를 유지하고 어떤 금액을 저축하거나 인출할 수 있는 액터를 살펴보자. 첫째로 메시지를 표현하는 클래스를 몇 가지 정의해야 한다.

<코드>

sealed class AccountMessage  
  
class GetBalance(  
 val amount: CompletableDeferred<Long>  
) : AccountMessage()  
  
class Deposit(val amount: Long) : AccountMessage()  
  
class Withdraw(  
 val amount: Long,  
 val isPermitted: CompletableDeferred<Boolean>  
) : AccountMessage()

</코드>

봉인된 클래스를 사용하면 AccountMessage 타입을 처리하는 when 식에서 else를 쓰지 않아도 된다.

GetBalance 인스턴스에는 CompletableDeferred라는 타입의 프로퍼티가 있다. 액터는 이 프로퍼티를 사용해서 GetBalance 메시지를 보낸 코루틴에게 현재 잔고를 돌려준다. Withdraw 클래스에도 인출에 성공하면 true를 돌려주고 그렇지 않으면 flase를 돌려주기 위한 isPermitted이라는 비슷한 프로퍼티가 있다.

이제 계좌 잔고를 유지하는 액터를 구현할 수 있다. 기본 로직은 단순하다. 메시지가 들어오는 채널을 계속 폴링(polling)하면서 수신한 메시지의 종류에 따라 적절한 동작을 수행한다.

<코드>

fun CoroutineScope.accountManager(  
 initialBalance: Long  
) = actor<AccountMessage> {  
  
 var balance = initialBalance  
   
 for (message in channel) {  
 when (message) {  
 is GetBalance ->message.amount.complete(balance)  
   
 is Deposit -> {  
 balance += message.amount  
 println("Deposited ${message.amount}")  
 }  
   
 is Withdraw -> {  
 val canWithdraw = balance >= message.amount  
 if (canWithdraw) {  
 balance -= message.amount  
 println("Withdrawn ${message.amount}")  
 }  
 message.isPermitted.complete(canWithdraw)  
 }  
 }  
 }  
}

</코드>

actor() 빌더는 produce()에 대응한다고 할 수 있다. 두 빌더 모두 통신에 채널을 사용하지만, 액터는 데이터를 받기 위해 채널을 사용하고 생산자는 소비자에게 데이터를 보내기 위해 채널을 생성한다. 액터는 기본적으로 랑데뷰 채널을 사용한다. 하지만 acotr() 함수를 호출하면서 capacity를 변경하면 채널의 성격을 바꿀 수 있다.

CompletableDeferred에서 complete() 메서드를 사용하는 부분에 유의하라. 액터 클라이언트에게 요청 결과를 돌려줄 때 이 방법을 사용한다.

이제 이 액터와 통신하는 코루틴을 한 쌍 만들자.

<코드>

private suspend fun SendChannel<AccountMessage>.deposit(  
 name: String,  
 amount: Long  
) {  
 send(Deposit(amount))  
 println("$name: deposit $amount")  
}  
  
private suspend fun SendChannel<AccountMessage>.tryWithdraw(  
 name: String,  
 amount: Long  
) {  
 val status = CompletableDeferred<Boolean>().let {  
 send(Withdraw(amount, it))  
 if (it.await()) "OK" else "DENIED"  
 }  
 println("$name: withdraw $amount ($status)")  
}  
  
private suspend fun SendChannel<AccountMessage>.printBalance(  
 name: String  
) {  
 val balance = CompletableDeferred<Long>().let {  
 send(GetBalance(it))  
 it.await()  
 }  
 println("$name: balance is $balance")  
}  
  
fun main() {  
 runBlocking {  
 val manager = accountManager(100)  
 withContext(Dispatchers.Default) {  
 launch {  
 manager.deposit("Client #1", 50)  
 manager.printBalance("Client #1")  
 }  
   
 launch {  
 manager.tryWithdraw("Client #2", 100)  
 manager.printBalance("Client #2")  
 }  
 }  
   
 manager.tryWithdraw("Client #0", 1000)  
 manager.printBalance("Client #0")  
 manager.close()  
 }  
}

</코드>

액터에게 메시지를 보내기 위해서는 액터가 사용하는 채널에 대해 send() 메서드를 호출해야 한다. 가능한 출력의 예는 다음과 같다.

<코드>

Client #1: deposit 50  
Deposited 50  
Withdrawn 100  
Client #2: withdraw 100 (OK)  
Client #2: balance is 50  
Client #1: balance is 50  
Client #0: withdraw 1000 (DENIED)  
Client #0: balance is 50

</코드>

연산 순서는 달라질 수 있지만(특히 병렬 처리가 관련되면 순서가 다를 수 있다), 결과는 일관성이 있다. 공개적으로 접근 가능한 가변 상태가 없기 때문에 락이나 임계 영역 같은 동기화 요소를 사용하지 않아도 된다.

한 가지 더, 액터 빌더도 현재 실험적인 API이기 때문에 향후 변경될 여지가 있다는 점을 언급해둔다.

## 자바 동시성 사용하기

코틀린에서만 사용할 수 있는 코루틴 라이브러리 외에도 JVM 플랫폼에서는 JDK가 제공하는 동기화 요소를 활용할 수 있다. 이 절에서는 스레드 생성이나 동기화 등과 관련한 작업을 좀 더 편하게 할 수 있도록 코틀린 표준 라이브러리가 제공하는 여러 가지 도우미 함수에 대해 살펴보자.

### 스레드 시작하기

범용 스레드를 시작하려면 스레드에서 실행하려는 실행가능(Runnable) 객체에 대응하는 람다와 스레드 프로퍼티들을 지정해서 thread() 함수를 사용하면 된다.

* start: 스레드를 생성하자마자 시작할지 여부(디폴트는 true)
* isDaemon: 스레드를 데몬 모드로 시작할지 여부(디폴트는 false). 데몬 스레드는 JVM의 종료를 방해하지 않고 메인 스레드가 종료될 때 자동으로 함께 종료된다.
* contextClassLoader: 스레드 코드가 클래스와 자원을 적재할 때 사용할 클래스 로더(디폴트는 널)
* name: 커스텀 스레드 이름. 디폴트는 널인데, 이는 JVM이 이름을 자동으로 지정한다는 뜻이다(Thread-1, Thread-2 등으로 정해진다).
* priority: Thread.MIN\_PRIORITY(=1)부터 Thread.MAX\_PRIORITY(=10) 사이의 값으로 정해지는 우선순위로, 어떤 스레드가 다른 스레드에 비해 얼마나 많은 CPU 시간을 배정받는지 결정한다. 디폴트 값은 -1이며 이 값은 자동으로 우선순위를 정하라는 뜻이다.
* block: () -> Unit 타입의 함수값으로 새 스레드가 생성되면 실행할 코드다.

예를 들어 다음 프로그램은 매 150밀리초마다 메시지를 출력하는 스레드를 시작한다.

<코드>

import kotlin.concurrent.thread  
  
fun main() {  
 println("Starting a thread...")  
   
 thread(name = "Worker", isDaemon = true) {  
 for (i in 1..5) {  
 println("${Thread.currentThread().name}: $i")  
 Thread.sleep(150)  
 }  
 }  
   
 Thread.sleep(500)  
 println("Shutting down...")  
}

</코드>

새 스레드가 데몬 모드로 시작했으므로 메인 스레드가 500밀리초 슬립한 다음 실행을 끝낼 때 이 스레드도 함께 끝나기 때문에 메시지가 4개만 출력된다. 이 프로그램을 실행한 결과는 다음과 같다.

<코드>

Starting a thread...  
Worker: 1  
Worker: 2  
Worker: 3  
Worker: 4  
Shutting down...

</코드>

다른 함수로는 어떤 지정한 시간 간격으로 동작을 수행하는 자바 타이머 관련 함수가 있다. timer() 함수는 어떤 작업을 이전 작업이 끝난 시점을 기준으로 고정된 시간 간격으로 실행하는 타이머를 설정한다. 그 결과 어떤 작업이 시간이 오래 걸리면 이후의 모든 실행이 연기된다. 따라서 이 타이머는 FIXED\_RATE 모드로 작동하는 코틀린 티커에 비유할 수 있다. 타이머를 timer() 호출로 설정할 때 다음 옵션을 지정할 수 있다.

* name: 타이머 스레드의 이름(디폴트는 널)
* daemon: 타이머 스레드를 데몬 스레드로 할지 여부(디폴트는 false)
* startAt: 최초로 타이머 이벤트가 발생하는 시간을 나타내는 Date 객체
* period: 연속된 타이머 이벤트 사이의 시간 간격(밀리초 단위)
* action: 타이머 이벤트가 발생할 때마다 실행될 TimeTask.() -> Unit 타입의 람다

또는 이 방법을 사용하는 대신, 최초 이벤트가 몇 밀리초 뒤에 시작할지를 나타내는 initalDelay 파라미터가 있는(이 값의 디폴트는 0이다) 오버로드된 다른 timer()를 사용할 수도 있다.

앞의 예제를 타이머를 써서 다시 작성해보자.

<코드>

import kotlin.concurrent.timer  
  
fun main() {  
 println("Starting a thread...")  
 var counter = 0  
   
 timer(period = 150, name = "Worker", daemon = true) {  
 println("${Thread.currentThread().name}: ${++counter}")  
 }  
   
 Thread.sleep(500)  
 println("Shutting down...")  
}

</코드>

또한, 두 타이머 이벤트 사이의 시간 간격을 최대한 일정하게 맞춰주는 fixedRateTimer() 함수들도 있다. 이 함수들은 장기적으로 타이머 이벤트 사이의 시간 간격을 일정하게 유지하기 위해 지연 시간을 조정해 준다는 점에서 FIXED\_PERIOD 모드의 티커에 비유할 수 있다.

### 동기화와 락

동기화는 특정 코드 조각이 한 스레드에서만 실행되도록 보장하기 위한 공통적인 기본 요소다. 이런 코드 조각을 다른 스레드가 실행하고 있다면 해당 코드에 진입하려고 시도하는 다른 스레드들은 모두 대기해야 한다. 자바에서는 코드에 동기화를 도입하는 방법이 두 가지 있다. 첫 번째, 락(lock)으로 사용하려는 어떤 객체를 지정하는 특별한 동기화 블록을 사용해 동기화해야 하는 코드를 감쌀 수 있다. 코틀린에서도 이런 동기화 구문은 자바 구문과 상당히 비슷하다. 다만 언어에 내장된 구조를 사용하는 대신, 표준 라이브러리 함수를 사용해야 한다.

<코드>

import kotlin.concurrent.thread  
  
fun main() {  
 var counter = 0  
 val lock = Any()  
   
 for (i in 1..5) {  
 thread(isDaemon = false) {  
 synchronized(lock) {  
 counter += i  
 println(counter)  
 }  
 }  
 }  
}

</코드>

개별 덧셈의 결과는 달라질 수 있어서 중간 결과가 바뀔 수 있지만, 동기화로 인해 전체 합계는 항상 15가 된다. 다음은 출력의 예다.

<코드>

1  
4  
8  
13  
15

</코드>

일반적으로 synchronized() 함수는 람다의 반환값을 반환한다. 예를 들어 호출되는 시점의 중간 카운터 값을 읽을 때 synchronized()를 사용할 수도 있다.

<코드>

import kotlin.concurrent.thread  
  
fun main() {  
 var counter = 0  
 val lock = Any()  
   
 for (i in 1..5) {...} // 앞의 예제에서 본 스레드를 생성하는 부분  
   
 val currentCounter = synchronized(lock) { counter }  
 println("Current counter: $currentCounter")  
}

</코드>

출력되는 중간 결과는 다를 수 있지만, 이 모든 값은 다섯 개의 덧셈 스레드 중 하나가 만들어낸 값에 해당한다.

자바에서 동기화에 사용하는 다른 방법은 메서드에 synchronized 변경자를 붙이는 것이다. 이럴 경우 메서드 본문 전체가 현재의 클래스 인스턴스(메서드가 인스턴스 메서드인 경우)나 Class 인스턴스 자체(메서드가 정적 메서드인 경우)에 의해 동기화된다. 코틀린에서는 @Synchronized 애너테이션을 통해 같은 목적을 달성할 수 있다.

<코드>

import kotlin.concurrent.thread  
  
class Counter {  
 private var value = 0  
 @Synchronized fun addAndPrint(value: Int) {  
 value += value  
 println(value)  
 }  
}  
  
fun main() {  
 val counter = Counter()  
 for (i in 1..5) {  
 thread(isDaemon = false) { counter.addAndPrint(i) }  
 }  
}

</코드>

표준 라이브러리에는 동기화 블록과 비슷한 어떤 Lock 객체(java.util.concuirrent.locks 패키지에 있음)를 사용해 주어진 람다를 실행하게 해주는 withLock() 함수도 있다. withLock()을 사용하면 함수가 알아서 락을 풀어주므로, 예외가 발생할 때 락을 푸는 것에 신경 쓰지 않아도 된다. 예를 들어 Counter 클래스에 이 함수를 적용해보자.

<코드>

class Counter {  
 private var value = 0  
 private val lock = ReentrantLock()  
   
 fun addAndPrint(value: Int) {  
 lock.withLock {  
 value += value  
 println(value)  
 }  
 }  
}

</코드>

그 외에도 ReentrantReadWriteLock의 읽기와 쓰기 락을 사용해 주어진 작업을 수행하는 read()와 write() 함수도 있다. write() 함수는 기존 읽기 락을 쓰기 락으로 자동 승격시켜줌으로써 재진입 가능(reentrant[[1]](#footnote-1))한 락의 의미를 유지한다.

**자바와 코틀린의 차이**: 자바의 Object 클래스에 정의된 wait(), notify(), notifyAll() 메서드는 코틀린 Any에 없다. 필요하다면 명시적으로 객체를 java.lang.Object 값으로 캐스팅해서 이런 함수를 쓸 수 있다.

<코드>

(obj as Object).wait()

</코드>

다른 블러킹 메서드와 마찬가지로 wait()도 일시중단 함수 안에서 호출하지 말아야 한다는 점에 유의하라.

## 결론

이 장에서는 코틀린 코루틴 기반 동시성의 기초를 다뤘다. 일시중단 함수를 통해 동시성 코드를 작성할 수 있는 이유를 살펴보고, 코루틴 빌더와 영역과 문맥을 사용해 코루틴의 생명 주기를 관리하는 방법에 대해 설명했다. 여러 동시성 작업 사이에 데이터를 효율적으로 공유하기 위해 채널과 액터 기반의 통신을 사용하는 방법도 살펴봤다.

추가로 JVM 플랫폼에서 사용할 수 있는 동시성 API를 코틀린에서 활용할 때 도움이 되는 코틀린 표준 라이브러리 함수에 대해 살펴봤다.

다음 장에서는 테스트에 대해 중점적으로 살펴본다. 코틀린 테스트 코드를 처리할 수 있는 테스트 프레임워크를 몇 가지 소개하고, 코틀린 언어의 특징과 DSL이 여러 가지 테스트케이스를 작성할 때 어떻게 유용한지에 대해 살펴보겠다.

## 질문

1. 일시중단 함수란 무엇인가? 일시중단 함수의 동작과 일반적인 함수의 동작이 어떻게 다른가?
2. launch()와 async() 빌더로 코루틴을 만드는 방법은 무엇인가? launch()와 async()의 차이는 무엇인가?
3. runBlocking() 빌더의 목적에 대해 설명하라.
4. 구조적 동시성이란 무엇인가?
5. 동시성 잡의 생명주기를 설명하라. 코루틴 트리에서 잡 취소가 어떻게 전파되는지 설명하라.
6. 코루틴 디스패처란 무엇인가? 코루틴 라이브러리가 제공하는 공통적인 디스패처 구현에 대해 설명하라.
7. 코루틴 내부에서 디스패처를 바꾸는 방법은 무엇인가?
8. 코루틴 라이브러리가 사용하는 예외 처리 메커니즘에 대해 설명하라. CoroutineExceptionHandler의 역할은 무엇인가?
9. 수퍼바이저 잡이란 무엇인가? 내포된 코루틴에서 발생한 예외를 처리할 때 어떻게 수퍼바이저 잡을 활용할 수 있는가?
10. 채널이란 무엇인가? 코루틴 라이브러리가 지원하는 채널에는 어떤 종류가 있는가?
11. produce() 함수를 사용해 채널을 만드는 방법은 무엇인가?
12. 티커 채널의 동작에 대해 설명하라.
13. 액터 모델의 기본 개념을 설명하라. 액터를 코루틴 동시성 라이브러리에서 활용하는 방법은 무엇인가?
14. 코틀린 표준 라이브러리가 제공하는 스레드 생성 유틸리티 함수에 대해 설명하라.
15. 코틀린 코드에서 스레드 간의 동기화와 락을 어떻게 사용하는지 설명하라.

1. 옮긴이 - 락을 이미 획득한 스레드가 다시 같은 락을 요청해도 문제없이 작동할 때 이런 락을 재진입가능하다고 말한다. [↑](#footnote-ref-1)