Chapter14. 코루틴

Part3

코루틴과 플로우를 활용한 동시성 프로그래밍

- 동시성은 영향력이 큰 주제
- 현대 애플리케이션은 한 번에 하나 이상의 일을 해야할 필요가 있다.
    - 네트워크 요청, io
    - cpu 위주 작업
    - 또는 다른 방식의 다중 처리
- 코루틴 개념에 기반을 둔 독특하고 튼튼한 동시성 프로그래밍 접근 방법 제공
    - 코루틴: 스레드 위에서 동작하는 경량 추상화
    - kotlinx.coroutines
- 플로우
    - 시간이 흐름에 따라 발생할 수 있는 여러 순차적인 값을 다루는 코틀린으로 구축된 방법
    - 이런 값 스트림을 변환할 때 플로우 라이브러리를 쓸 수 있다.
    - 차가운 플로우, 뜨거운 플로우

14장에서 다루는 내용

- 동시성과 병렬성의 개념
- 코틀린에서 동시성 연산을 만드는 빌딩 블록인 일시 중단 함수
- 코틀린에서 코루틴을 활용해 동시성 프로그래밍에 접근하는 방법

현대 프로그램들이 한 번에 한 가지 일만 하느 경우는 드물다. 현대 애플리케이션은 동시에 여러 일을 비동기적으로 할 필요가 있다.

- 이는 개발자에게 다른 여러 연산을 서로 블록시키지 않고 독립적으로 실행하면서 여러 동시 작업을 동기화하고 조화시킬 수 있는 도구가 필요하다는 뜻
- 코루틴

동시성과 병렬성

코루틴 살펴보기 전에 먼저 동시성은 무엇인지, 동시성과 병렬성 개념 사이의 관계를 살펴보자

동시성

- 여러 작업을 동시에 실행하는 것을 말한다.
- 모든 작업을 물리적으로 함께 실행할 필요는 없다.
- 코드의 여러 부분을 돌아가면서 실행하는 것도 동시성 시스템이다.
    - cpu 코어가 하나뿐인 시스템에서 실행되는 애플리케이션까지도 동시성을 사용할 수 있다는 뜻
    - 이런 경우 여러 동시성 태스크를 계속 전환해 가면서 동시성을 달성한다.
    - eg) ui - calca - ui - calc

병렬성

- 코드를 여러 부분으로 나눠서 동시에 수행할 수 있는 능력을 말하는 동시성 대신, 병렬성은 여러 작업을 여러 CPU 코어에서 물리적으로 동시에 실행하는 것을 말한다.
- 병렬 계산은 현대적 멀티코어 하드웨어를 효과적으로 사용할 수 있고, 그 효율을 더 높이는 경우도 많다.
    - 어려운 점은 있다.
    - 14.7.4
    - eg) 1) ui - ui- ui, 2) calc - calc

코루틴을 사용하면 동시성계산과 병렬성 계산을 모두할 수 있다.

> 💡 krill bobrow의 grokkung concurrency는 병렬성과 동시성의 일반적인 개요를 알려줌

코틀린의 동시성 처리 방법: 일시 중단 함수와 코루틴

- 코루틴은 코틀린의 강력한 특징으로 비동기적으로 실행되는 넌블로킹 동시성 코드를 우아하게 작성할 수 있게 해준다.
- 스레드와 같은 전통적 방법과 비교하면 코루틴이 훨씬 더 가볍게 작동한다.
- 구조화된 동시성을 통해 코루틴은 동시성 작업과 그 생명주기를 관리할 수 있는 기능도 제공한다.
- 코틀린에서 코루틴을 사용할 때 기본적인 추상화인 일시 중단(suspending) 함수를 살펴본다.
    - 일시 중단 함수는 스레드를 블록시키는 단점이 없이 순차적 코드처럼 보이는 동시성 코드를 작성할 수 있게 해준다.
    - 코루틴과 콜백, 퓨처, 반응형 스트림 등 다른 동시성 모델을 비교하면서 코루틴 추상화의 단숨함을 살펴보자

스레드와 코루틴 비교

- jvm에서 병렬 프로그래밍과 동시성 프로그래밍을 위한 고전적인 추상화는 스레드를 사용하는 것
- 스레드는 서로 독립적으로 동시에 실행되는 코드 블록을 지정할 수 있게 해준다.
- 자바의 스레드도 코틀린에서 모두 호환된다.
    - 코틀린 표준 라이브러리가 제공하는 편의 함수 사용 가능
    - thread 함수

```
fun main() {
    println("I'm on ${Thread.currentThread().name}")
    thread { // <1>
        println("And I'm on ${Thread.currentThread().name}")
    }
}
```

- 스레드는 애플리케이션을 더 반응성 있게 만들어주고, 멀티코어 cpu의 여러 코어에 작업을 분산시켜 현대적 시스템을 더 효율적 사용할 수 있게 해준다.
- 그러나 스레드를 사용하는 데는 비용이 든다.
    - jvm에서 생성하는 각 세르데는 일반적으로 운영체제가 관리하는 스레드

    - 이러한 시스템 스레드를 생성하고 관리하는 것은 비용이 많이 든다.
        - 보통 한 번에 몇 천개의 스레드만 효과적으로 관리 가능
        - 메모리 할당, 스레드 간 전환은 커널 수준에서 실행 작업
- 게다가 스레다가 어떤 작업이 완료되길 기다리는 동안에는 블록된다.
    - 응답을 기다리는 동안에는 다른 작업을 할 수 없으며, sleep하며 시스템 자원을 차지
    - 따라서 새 스레드를 생성할 때는 매우 신중해야 하며, 짧은 시간 동안 잠깐 사용하는 것은 피하는 것이 좋다.
- 스레드는 기본적으로 독립적인 프로세스로 존재하기 때문에 작업을 관리하고 조정하는 데 어려움이 있을 수 있다.
    - 특히 취소나 예외 처리 같은 개념을 다룰 때 그렇다.
    - 이런 제약은 스레드의 사용성을 제한한다.
        - 생성 제한: 비용과 블로킹
        - 관리 제한: 계층이라는 개념 X
- 코틀린은 스레드에 대한 대안으로 코루틴이라는 추상화를 도입했다.
    - 코루틴은 일시 중단 가능한 계산을 나타낸다.
- 코루틴 장점
    - 코루틴은 초경량 추상화
        - 노트북에서도 십만개 이상 실행 가능
        - 코루틴은 생성하고 관리하는 비용이 저렴하다.
    - 코루틴은 시스템 자원을 블록시키지 않고 실행을 일시 중단할 수 잇으며, 나중에 중단된 지점에서 실행을 재개할 수 있다.
        - 네트워크 요청이나 입출력 작업 같은 비동기 작업을 처리할 때 블로킹 스레드보다 훨씬 효율적
    - 코루틴은 구조화된 동시성이라는 개념을 통해 동시 작업의 구조와 계층을 확립하며, 취소 및 오류 처리를 위한 메커니즘을 제공한다.
        - 동시 계산의 일부가 실패하거나 더 이상 필요하지 않게 됐을 때 구조화된 동시성은 자식으로 시작된 다른 코루틴들도 함께 취소되도록 보장한다.
- 내부적으로 코루틴은 하나 이상의 jvm 스레드에서 실행된다.

- 코루틴을 사용해 작성한 코드도 여전히 기본 스레드 모델이 제공하는 병렬성을 활용할 수 있지만, 운영체제가 부과하는 스레드에 한계에 얽매이지 않는다는 것을 의미한다.

프로젝트 룸(버츄얼 스레드)과 코루틴은 비슷한 문제를 해결하려고 하기 때문에 관계를 살펴보자.

코루틴은 동시성 코드를 작성하기 위한 추상화로 발전

프로젝트 룸의 주요 목표는 기존 i/o 중심 레거시 코드를 가상 스레드로 포팅할 수 있게하는 것

코틀린 코루틴도 룸이 제공하는 새로운 기능을 활용할 수 있을 것이다. 블로킹 api를 사용해 작성된 코드와 코루틴을 더 잘통합하는 등

잠시 멈출 수 있는 함수: 일시 중단 함수

- 코루틴이 스레드, 반응형 스트림, 콜백과 같은 다른 동시성 접근 방식과 다른 핵심 속성으로는 대부분 코드 형태를 크게 변경할 필요가 없다는 점
  - 코드는 여전히 순차적으로 보인다.
- 일시 중단 함수가 어떻게 이런 방식이 가능하게할까?

일시 중단 함수를 사용한 코드는 순차적으로 보인다.

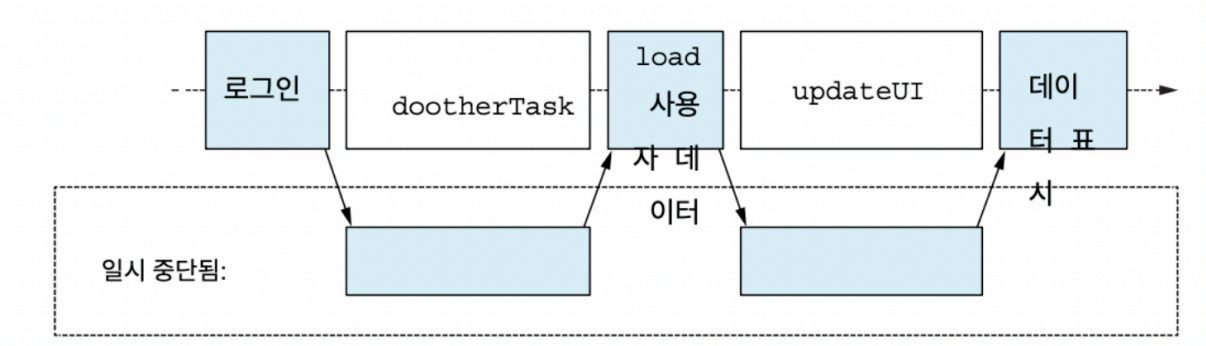
```
fun login(credentials: Credentials): UserID // 블로킹 함수
fun loadUserData(userId: UserID): UserData  // 블로킹 함수
fun showData(data: UserData)

fun showUserInfo(credentials: Credentials) {
    val userId = login(credentials)
    val userData = loadUserData(userId)
    showData(userData)
}
```

- login, loadUserData 함수는 네트워크 요청을 보낸다.
    - (login - block) → (loadUserData - block) → shwoData
- 대부분의 시간을 네트워크 작업 결과를 기다리는데 소모하며, showUserInfo 함수가 실행중인 스레드를 블록시킨다.
    - 블록된 스레드는 자원을 낭비
- 일시 중단 함수는 이 문제를 개선하는 데 도움을 줄 수 있다.

```
suspend fun login(credentials: Credentials): UserID // 논블록
suspend fun loadUserData(userId: UserID): UserData  // 논블록
fun showData(data: UserData)

suspend fun showUserInfo(credentials: Credentials) {
    val userId = login(credentials)
    val userData = loadUserData(userId)
    showData(userData)
}
```



- 코루틴을 사용해 넌블로킹 방식으로 구현
- 코드가 여전히 순차적으로 보인다
    - 실제 차이는 suspend 변경자
- suspend 변경자는 함수가 실행을 잠시 멈출 수도 있다는 뜻이다.
    - 일시 중단은 기저 스레드를 블록시키지 않는다.
    - 대신 함수 실행이 일시 중단되면 다른 코드가 같은 스레드에서 실행 될 수 있다.
- 일시 중단된 사이에 기저 스레드는 다른 작업을 진행할 수 있다.

    - 사용자 인터페이스를 그리거나 등등
- 하지만 기저 라이브러리의 구현(login, loadUserData)도 코틀린 코루틴을 고려돼 작성돼야 한다.
    - 실제로 많은 라이브러리가 코루틴과 함께 작동되는 api를 제공한다.
    - ktor httpclient, retrofit, okhttp 등

코루틴을 다른 접근 방법과 비교

- 3가지 접근 방식을 살펴보자
    - 콜백
    - 반응형 스트림(RxJava)
    - 퓨처

콜백

```
fun loginAsync(credentials: Credentials, callback: (UserID) → Unit)
fun loadUserDataAsync(userId: UserID, callback: (UserData) → Unit)
fun showData(userData: UserData)

fun showUserInfo(credentials: Credentials) {
    loginAsync(credentials) { userId →
        loadUserDataAsync(userId) { userData →
            showData(userData)
        }
    }
}
```

- 함수의 시그니처를 변경해 콜백 파라미터를 제공해야 한다.
    - 콜백 지옥

퓨처

```
fun loginAsync(credentials: Credentials): CompletableFuture<UserID>
fun loadUserDataAsync(userId: UserID): CompletableFuture<UserData>
fun showData(data: UserData)

fun showUserData(credentials: Credentials) {
    loginAsync(credentials)
        .thenCompose { userId → loadUserDataAsync(userId) }
        .thenAccept { userData → showData(userData) }
}
```

- CompletableFuture를 사용하면 콜백 중첩을 피할 수 있지만 새로운 연산자의 의미를 배워야 한다.
    - 반환 타입도 변경해야 한다.

반응형 스트림

```
un login(credentials: Credentails): Single<UserID>
fun loadUserData(userId: UserID): Single<UserData>
fun showData(data: UserData)

fun showUserData(credentials: Credentials) {
    login(credentials)
        .flatMap { userId → loadUserData(userId) }
        .doOnSuccess { userData → showData(userData) }
        .subscribe()
}
```

- 콜백 지옥은 피하지만, 여전히 함수 시그니처 변경해야하며, 반환값을 Single로 감싸야 하고 또 연산자를 사용해야 한다.
- 두 접근 방식 모두 인지적 부가비용, 새로운 연산자를 코드에 도입해야 한다.
    - 코틀린은 suspend 변경자만 추가하면 된다.

일시 중단 함수 호출

- 일시 중단 함수는 실행을 일시 중단할 수 있기 때문에 일반 코드 아무 곳에서나 호출할 수는 없다.
- 일시 중단 함수는 실행을 일시 중단할 수 있는 코드 블록 안에서만 호출할 수 있다.
  - 그런 블록 중 하나가 다른 일시 중단 함수일 수 있다.
  - 이는 '함수가 실행을 일시 중단할 수 있다면 그 함수를 호출하는 함수의 실행도 잠재적으로 일시 중단될 수 있다' 라는 직관과도 잘 들어맞는다.
- 물론 일시 중단 함수의 본문에서 일반 함수도 호출할 수 있다.
- 어떻게 맨 처음에 일시 중단 함수를 호출할 수 있을까?
  - 가장 간단한 답은 main 함수를 suspend
  - 더 범용적이고 강력한 방법은 코루틴 빌더 함수를 사용하는 것
- 코루틴 빌더는 새로운 코루틴을 생성하는 역할을 하며, 일시 중단 함수를 호출하기 위한 일반적인 진입점으로 사용된다.

코루틴의 세계로 들어가기: 코루틴 빌더

- 일시 중단 함수는 실행을 잠깐 멈출 수 있는 함수이며, 다른 일시 중단 함수나 코루틴 안에서만 호출 할 수 있다.
  - 이제는 코루틴에서 일시 중단 함수를 호출하는 것을 알아보자.
- 코루틴은 일시 중단 가능한 계산의 인스턴스다.
  - 이를 다른 코루틴들과 동시에 실행될 수 있는 코드 블록으로 생각할 수 있다.
- 스레드와 비슷하지만 코루틴은 함수 실행을 일시 중단하는 데 필요한 메커니즘을 포함하고 있다.
- 이러한 코루틴을 생성할 때는 코루틴 빌더 함수 중 하나를 사용한다.
  - runBlocking: 블로킹 코드와 일시 중단 함수의 세계을 연결할 때 쓰인다.
  - launch: 값을 반환하지 않는 새로운 코루틴을 시작할 때 쓰인다.
  - async: 비동기적으로 값을 계산할 때 쓰인다.

일반 코드에서 코루틴의 세계로: runBlocking 함수

- 일반 블로킹 코드를 일시 중단 함수의 세계로 연결하려면 runBlocking 코루틴 빌더 함수에게 코루틴 본문을 구성하는 코드 블록을 전달할 수 있다.
- 이렇게 하면 새 코루틴을 생성하고 실행하며, 해당 코루틴이 완료될 때까지 현재 스레드를 블록시킨다.
- 전달된 코드 블록 내에서는 일시 중단 함수를 호출할 수 있다.

```
suspend fun doSomethingSlowly() {
    delay(500.milliseconds)
    println("I'm done")
}

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

- 스레드를 블록시키지 않기 위해서인데 왜 사용하는걸까?
    - 실제로 runBlocking을 사용할 때는 하나의 스레드를 블로킹 한다.
    - 그러나 이 코루틴 안에서는 추가적인 자식 코루틴을 얼마든지 시작할 수 있고, 이 자식 코루틴들은 다른 스레드를 더 이상 블록시키지 않는다.
    - 일시 중단될 때마다 하나의 스레드가 해방돼 다른 코루틴이 코드를 실행할 수 있게 된다.
    - 이런 추가 자식 코루틴을 시작할 때 launch 코루틴 빌더를 사용할 수 있다.

발사 후 망각 코루틴 생성 : launch 함수

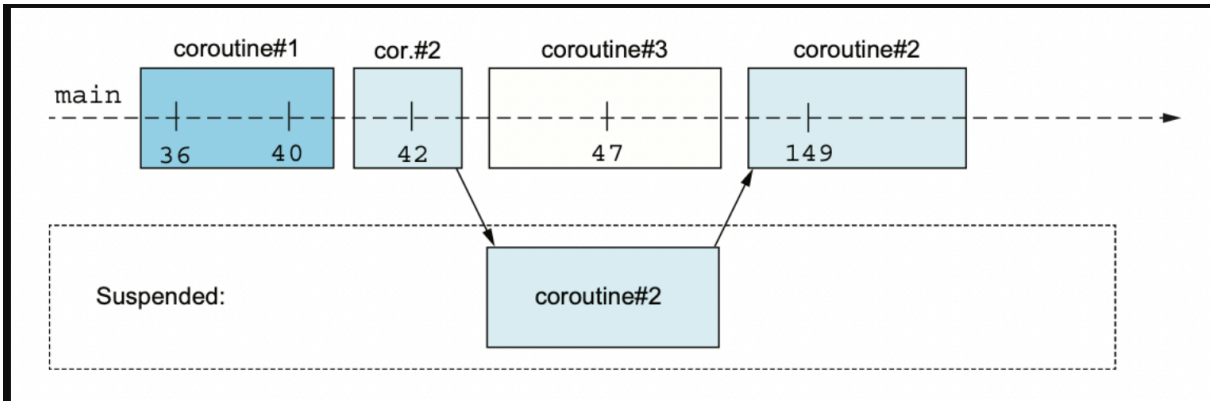
- launch 함수는 새로운 자식 코루틴을 시작하는 데 쓰인다.
- 일반적으로 '발사 후 망각' 시나리오에 사용되며, 어떤 코드를 실행하되 그 결과값을 기다리지 않는 경우에 적합하다.
- runBlocking이 오직 하나의 스레드만 블로킹한다는 주장을 테스트해보자.

```
private var zeroTime = System.currentTimeMillis()
fun log(message: Any?) =
    println(
        "${System.currentTimeMillis() - zeroTime} " +
```

```
            "[${Thread.currentThread().name}] $message"
    )
```

```
fun main() = runBlocking {
    log("The first, parent, coroutine starts")
    launch {
        log("The second coroutine starts and is ready to be suspended")
        delay(100.milliseconds)
        log("The second coroutine is resumed")
    }
    launch {
        log("The third coroutine can run in the meantime")
    }
    log("The first coroutine has launched two more coroutines")
}
```

```
36 [main @coroutine#1] The first, parent, coroutine starts
40 [main @coroutine#1] The first coroutine has launched two more coroutines
42 [main @coroutine#2] The second coroutine starts and is ready to be suspe
47 [main @coroutine#3] The third coroutine can run in the meantime
149 [main @coroutine#2] The second coroutine is resumed
```



- 사각형은 특정 시간에 해당 스레드에서 실행 중인 코루틴을 나타낸다.
- 3개의 코루틴이 시작된다.
    - 첫 번재는 runBlocking에 의해 시작된 부모 코루틴
    - 두번째와 세번째는 2번의 launch 호출에 의한 자식 코루틴

- coroutine#2가 delay 함수를 호출하면 코루틴이 일시 중단된다.
  - 이를 일시 중단 지점이라고 한다.
- 이제 coroutine#2는 지정된 시간동안 일시 중단되고, 메인 스레드는 다룬 코루틴이 실행될 수 있게 해방된다.
  - 그 결과 coroutine#3이 작업을 시작할 수 있다.
  - 100ms 후 coroutine#2 작업을 재개하고 프로그램 전체가 완료된다.
- 이 상황이 병렬성 없이 교차 실행되는 경우라는 것을 알 수 있다.
  - 모두 코루틴이 같은 스레드에서 실행됨
- 코루틴을 여러 스레드에서 병렬로 실행하고 싶다면 코드를 거의 변경하지 않고도 다중 스레드 디스패처를 사용할 수 있다.
  - 14.7
- launch를 사용해 새로운 기본 코루틴을 시작할 수 있지만, 동시 계산을 수행할 수는 있지만 코루틴 내부에서 값을 반환하는 것이 간단치 않다.
  - 값을 반환하는 것보다는 파일이나 데이터베이스에 쓰는 작업처럼 부수 효과를 일으키는 '시작 후 신경 쓰지 않아도 되는' 작업에 더 적합하다.
- launch 함수는 Job 타입의 객체를 반환하는데, 이를 시작된 코루틴에 대한 핸들로 생각할 수 있다.
  - Job 객체를 사용하면 코루틴 실행을 제어할 수 있다.
  - 예를 들어 취소를 촉발시킬 수 있다.
- 계산 결과를 반환하는 경우에는 async 빌더 함수를 쓴다.

일시 중단된 코루틴은 어디로 가는가?

코루틴 작동 주요 작업은 컴파일러가 수행한다. 컴파일러는 코루틴을 일시 중단하고 재개하며, 스케줄링하는 데 필요한 자원 코드를 생성한다. 일시 중단 함수의 코드는 컴파일 시점의 변환되고 실행 시점에 코루틴이 일시 중단될 때 해당 시점의 상태 정보가 메모리에 저장된다. 이 정보를 바탕으로 나중에 실행을 복구하고 재개할 수 있다.

대기 가능한 연산 : async 빌더

- 비동기 계산을 수행할 때 async 빌더 함수를 쓸 수 있다.
- launch와 마찬가지로 async에게도 실행할 코드를 코루틴으로 전달할 수 있다.
- 그러나 반환 타입은 launch와 달리 Deferred<T> 인스턴스다.
- Deferred를 사용해 주로 할 일은 await라는 일시 중단 함수로 그 결과를 기다리는 것이다.

```
suspend fun slowlyAddNumbers(a: Int, b: Int): Int {
    log("Waiting a bit before calculating $a + $b")
    delay(100.milliseconds * a)
    return a + b
}


fun main() = runBlocking {
    log("Starting the async computation")
    val myFirstDeferred = async { slowlyAddNumbers(2, 2) } // <1>
    val mySecondDeferred = async { slowlyAddNumbers(4, 4) } // <1>
    log("Waiting for the deferred value to be available")
    log("The first result: ${myFirstDeferred.await()}") // <2>
    log("The second result: ${mySecondDeferred.await()}")// <2>
}
```

```
0 [main @coroutine#1] Starting the async computation
4 [main @coroutine#1] Waiting for the deferred value to be available
8 [main @coroutine#2] Waiting a bit before calculating 2 + 2
9 [main @coroutine#3] Waiting a bit before calculating 4 + 4
213 [main @coroutine#1] The first result: 4
415 [main @coroutine#1] The second result: 8
```

- 시간을 보면 두 값을 계산하는데 총 약 400ms
    - 가장 오래 걸린 계산 시간과 같다.
- async 호출마다 새 코루틴을 시작함으로써 두 계산이 동시에 일어나게 했다.
- launch와 마찬가지로 async 호출한다고 해서 코루틴이 일시 중단되는 것은 아니다.

- await을 호출하면 그 Deferred에서 결과값이 사용 가능해질 때까지 루트 코루틴이 일시 중단된다.

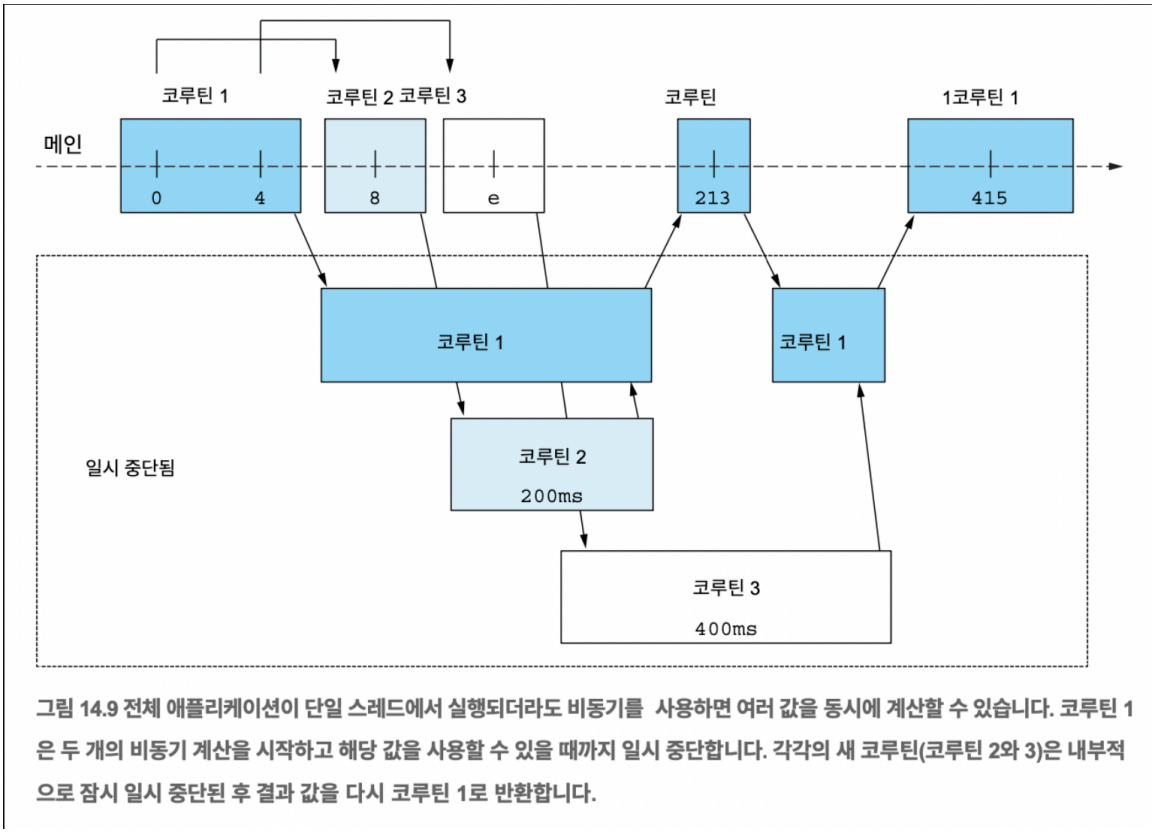


그림 14.9 전체 애플리케이션이 단일 스레드에서 실행되더라도 비동기를 사용하면 여러 값을 동시에 계산할 수 있습니다. 코루틴 1은 두 개의 비동기 계산을 시작하고 해당 값을 사용할 수 있을 때까지 일시 중단합니다. 각각의 새 코루틴(코루틴 2와 3)은 내부적으로 잠시 일시 중단된 후 결과 값을 다시 코루틴 1로 반환합니다.

- Deferred는 Future나 Promise 등과 동일한 개념
- Deferred 객체는 아직 사용할 수 없는 값을 나타낸다.
- 따라서 그 값을 계산하거나 어디서 읽어와야만 한다.
- Deferred 는 미래에 언제가는 값을 알게 될 것이라는 약속, 연기된 계산 결과값을 나타낸다.
- 코틀린에서는 독립적인 작업을 동시에 실행하고 그 결과를 기다릴 때만 async를 사용하면 된다.
    - 여러 작업을 동시에 시작할 필요가 없고 결과를 기다리지 않아도 된다면 async 사용할 필요가 없다.
    - 일반적인 일시 중단 함수 호출이면 충분한다.

빌더와 사용법 요약

빌더	반환값	활용처
`runBlocking`	람다가 계산한 값	블로킹 코드와 넌블로킹 코드 사이를 연결
`launch`	Job	발사 후 망각 코루틴 시작 (부수 효과가 있음)
`async`	Deferred	값을 비동기로 계산 (값을 기다릴 수 있음)

- 코루틴은 스레드 위에 만들어진 추상화
- 그렇다면 코드가 실제로 어떤 스레드에서 실행?
    - runBlocking이라는 특별한 경우 코드는 단순히 함수를 호출한 스레드에서 실행
- 코드가 어떤 스레드에서 실행될지 더 세밀하게 제어하려면 코틀린 코루틴에서 디스패처를 사용한다.

어서 코드를 실행할지 정하기: 디스패처

- 코루틴의 디스패처는 코루틴을 실행할 스레드를 결정한다.
- 디스패처를 선택함으로써 코루틴을 특정 스레드로 제한하거나 스레드 풀에 분산시킬 수 있으며, 코루틴이 한 스레드에서만 실행될지 여러 스레드에서 실행될지 결정할 수 있다.
- 본질적으로 코루틴은 특정 스레드에 고정되지 않는다.
- 코루틴은 한 스레드에서 실행을 일시중단하고 디스패처가 지시하는 대로 다른 스레드에서 실행을 재개할 수 있다.

디스패처 선택

- 코루틴은 기본적으로 부모 코루틴에서 디스패처를 상속받으므로 모든 코루틴에 대해 명시적으로 디스패처를 지정할 필요는 없다.
- 하지만 선택할 수 있는 디스패처들이 있다.
    - Dispatchers.Default
        - 기본 환경에서 실행할 때
    - Dispatchers.Main
        - UI 프레임워크랑 함께 작업할 때
    - Dispatchers.IO
        - 스레드를 블로킹하는 api를 사용할 때

- 명시적 코루틴 실행에 도움을 준다

UI 스레드에서 실행 : Dispathcers.Main

- ui 프레임워크를 사용 할 때는 특정 작업을 ui 스레드나 메인 스레드라고 불리는 특정 스레드에서 실행해야 할 때가 있다.
  - javaFX
  - AWT
  - Swing
  - android
- eg) 사용자 인터페이스 요소를 다시 그리는 작업
- 이런 작업을 안전하게 실행하려면 코루틴을 디스패치 할 때 Main을 사용할 수 있다.
  - Main의 실제 값은 사용하는 프레임워크에 따라 다르다.

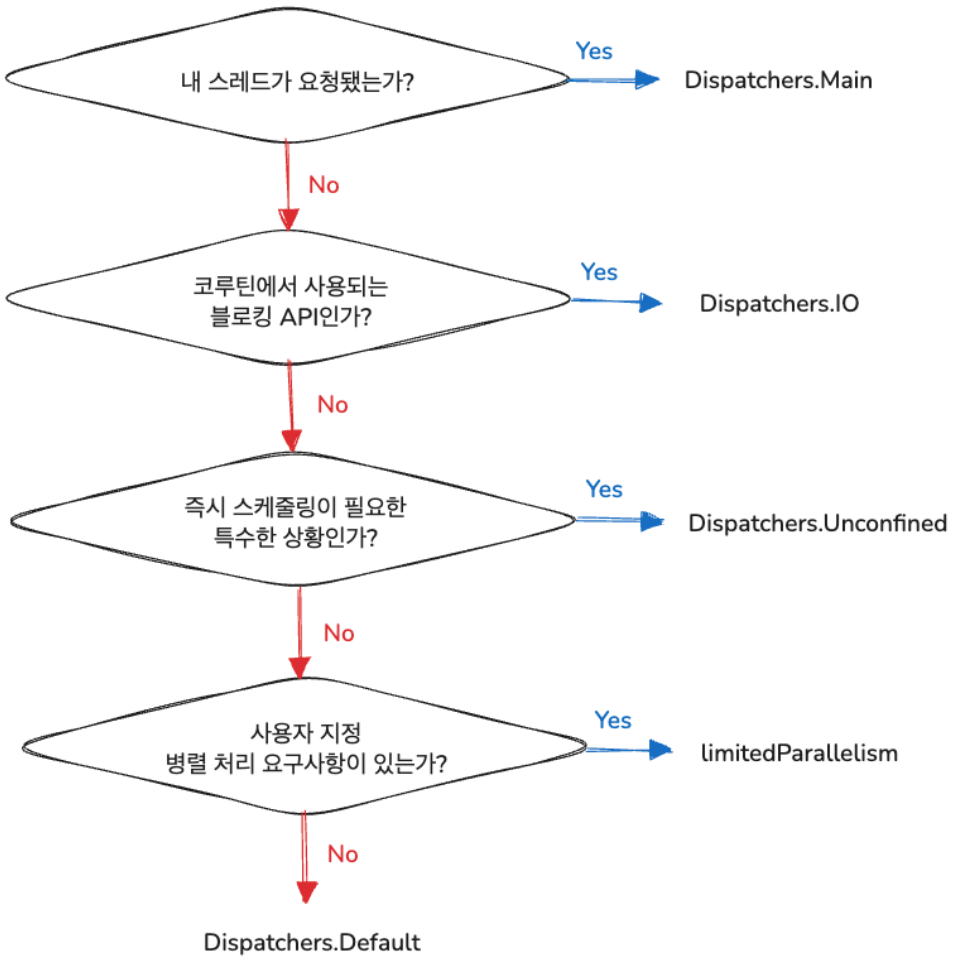
블로킹되는 IO 작업 처리 : Dispatchers.IO

- 서드파티 라이브러리를 사용할 때 코루틴을 염두에 두고 설계된 api를 선택할 수 없는 경우가 있다.
  - eg) db blocking api
- 기본 디스패처에서 이런 블로킹 api를 사용해야 하는 경우 문제가 발생할 수 있다.
  - 기본 디스패처의 스레드 수는 cpu 코어 수와 동일하기 때문에, 기본 스레드 풀이 블로킹 되어 다른 코루틴은 완료될 때까지 실행되지 못하기 때문이다.
  - IO는 이러한 상황을 처리하기 위해 설계되었다.
- 이 디스패처에서 실행된 코루틴은 자동으로 확장되는 스레드 풀에서 실행되며 cpu 집약적이지 않은 작업에 적합하다.

코루틴으로 구축한 동시성 시스템의 성능이 동작에 대해 별도의 요구 사항이 있는 경우 코루틴 라이브러리는 추가 기능을 제공한다. 공식문서 확인해보자

디스패처 비교

디스패처	스레드 개수	활용처
Dispatchers.Default	CPU 코어 수	일반적인 연산, CPU 집약적인 작업
Dispatchers.Main	1	UI 프레임워크의 맥락에서만 ui 작업
Dispatchers.IO	64 + CPU 코어 수 (최대 64개까지만 병렬 실행됨)	블로킹 IO 작업, 네트워크 작업, 파일 작업
Dispatchers.Unconfined	... (아무 스레드나)	즉시 스케줄링해야 하는 특수한 경우
limitedParallelism(n)	커스텀(n)	커스텀 시나리오



- 새 코루틴을 시작할 때 반드시 디스패처를 지정할 필요는 없다.

- 지정하지 않으면 부모 코루틴의 디스패처에서 실행된다는 것

코루틴 빌더에 디스패처 전달

- 코루틴을 특정 디스패처에서 실행하기 위해 코루틴 빌더 함수에게 디스패처를 인자로 전달할 수 있다.
- 모든 코루틴 빌더 함수는 코루틴 디스패처를 명시적으로 지정할 수 있게 한다.

```
fun main() {
    runBlocking {
        log("Doing some work")
        launch(Dispatcher.Default) {
            log("Doing some background work")
        }
    }
}

/*
26 [main @coroutine#1] Doing some work
33 [DefaultDispatcher-worker-1 @coroutine#2] Doing some background work
*/
```

- 첫 번째 로그 호출은 메인 스레드, 두 번째 호출은 디스패처 스레드 풀에 속한 스레드에서 실행된다.
- 코루틴 전체의 디스패처를 바꾸는 대신, 코루틴의 어떤 부분이 어디서 실행될지를 더 세밀하게 제어할 수 있다.
    - withContext 함수

withContext를 사용해 코루틴 안에서 디스패처 바꾸기

- ui 애플리케이션의 고전적 패턴
    - 백그라운드에서 시간이 오래걸리는 연산
    - 결과가 준비되면 ui 스레드로 전환해 사용자 인터페이스 갱신

- 이미 실행 중인 코루틴에서 디스패처를 바꿀 때는 withContext 함수에 다른 디스패처를 전달한다.

```
launch(Dispatchers.Default) {
    val result = performBackgroundOperation()
    withContext(Dispatchcers.Main) {
        updateUI(result)
    }
}
```

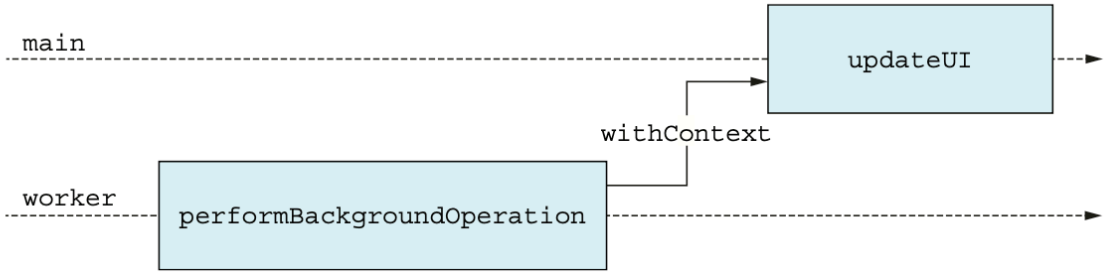


Figure 14.11 The call to `withContext` causes the coroutine, which was originally started on the default dispatcher and running on one of its worker threads, to execute on the specified dispatcher. In this case, this is the main thread to update the UI.

코루틴과 디스패처는 스레드 안정성 문제에 대한 마법 같은 해결책이 아니다

- Default와 IO는 다중 스레드를 사용하는 기본 제공 디스패처다.
- 다중 스레드 디스패처는 코루틴을 여러 스레드에 분산시켜 실행한다.
    - 그렇다면 전형적인 스레드 안전성 문제가 발생할까?
- 한 코루틴은 항상 순차적으로 실행된다.
    - 즉, 어느 단일 코루틴의 어떤 부분도 병렬로 실행되지 않는다.
    - 이는 단일 코루틴에 연관된 데이터가 전형적인 동기화 문제를 일으키지 않는다는 것을 의미한다.
- 하지만 여러 코루틴이 동일한 데이터를 읽거나 변경하는 경우에는 그리 단순하지 않다.
    - 결론적으로는 동시성 문제가 발생한다.

```
fun main() {
    runBlocking {
        launch(Dispatchers.Default) {
            var x = 0
            repeat(10_000) {
                x++
            }
            println(x)

        }
    }
}
// 10,000
```

- 정확한 x의 값

```
fun main() {
    runBlocking {
        var x = 0
        repeat(10_000) {
            launch(Dispatchers.Default) {
                x++
            }
        }
    delay(1.seconds)
        println(x)
    }
}
// 9,916
```

- 카운터 값이 예상보다 낮다.
- 여러 코루틴이 같은 데이터를 수정하고 있기 때문에 다중 스레드 디스패처에서 실행되면 일부 증가 작업이 서로의 결과를 덮어쓰는 상황이 발생할 수 있기 때문이다.

몇가지 해결방법

- 코루틴의 Metex 잠금
  - 이를 통해 임계 영역이 한 번에 하나의 코루틴만 실행되게 보장할 수 있다.

```
fun main() = runBlocking {
  val mutex = Mutex()
    var x = 0
    repeat(10_000) {
        launch(Dispatchers.Default) {
            mutex.withLock {
                        x++
                    }
                }
    }
    delay(1.seconds)
    println(x)

}
// 10000
```

- AtomicInteger 같은 병렬 변경을 위해 설계된 원자적이고 스레드 세이프한 데이터 구조 사용
- 코루틴(또는 withContext 사용해서 임계 영역만)을 단일 스레드 디스패처에서 실행하도록 제한
  - 성능 특성도 고려해야 한다.
- 더 자세한건 공식문서 참고
  - Mutable Shared State and Concurrency

코루틴은 코루틴 콘텍스트에 추가적인 정보를 담고 있다.

- 앞에서 코루틴 빌더 함수와 withContext 함수에 서로 다른 디스패처를 인자로 전달했지만, 사실 파라미터를 살펴보면 실제로는 CoroutineDispatcher가 아니다.
  - 실제로는 CoroutineContext

- 각 코루틴은 추가적인 문맥 정보를 담고 있는데, 이 문맥은 CoroutineContext 라는 형태로 제공된다.
    - CoroutineContext는 여러 요소로 이뤄진 집합이라고 생각할 수 있다.
    - 이 요소 중 하나는 코루틴이 어떤 스레드에서 실행될지를 결정하는 디스패처다.
- CoroutineContext 에는 보통 코루틴의 생명주기와 취소를 관리하는 Job 객체도 포함된다.
- 또한 CoroutineContext에 CoroutineName이나 CoroutineExceptionHandler 같은 추가적인 메타데이터도 있을 수 있다.
- 현재 코루틴의 코루틴 컨텍스트를 확인하려면 어떤 일시 중단 함수 안에서는 CoroutineContext 라는 특별한 속성에 접근하면 된다.
    - 이 속성은 실제로는 코틀린 코드에 정의된 것이 아니라 컴파일러 고유의 기능이다.
    - 즉, 실제 구현은 코틀린 컴파일러에 의해 특별히 처리된다.

```
import kotlin.coroutines.coroutineContext

suspend fun introspect() {
    log(coroutineContext) // coroutineContext 컴파일러 고유 기능에는 코루틴에 대한
}

fun main() {
    runBlocking {
        introspect()
    }
}

// 25 [main @coroutine#1] [CoroutineId(1), "coroutine#1":BlockingCoroutine{A
```

- 코루틴 빌더나 withContext 함수에 인자를 전달하면 자식 코루틴의 컨텍스트에서 해당 요소를 덮어쓴다.
- 여러 파라미터를 한 번에 덮어쓰려면 + 연산자를 사용해 coroutineContext 객체를 결합할 수 있다.

```
fun main() {
    runBlocking(Dispatcher.IO + CoroutineName("CoolRoutine")) {
```

```
        introspect()
    }
}

/**
 * 26) [DefaultDispatcher-worker-1 @Coolroutine#1] [
 *                                     CoroutineName(Coolroutine),
 *                                     CoroutineId(1),
 *                                     "Coolroutine#1":BlockingCoroutine{Active}@
 *                                     Dispatchers.IO
 *                                   ]
 */
```

- 디스패처 및 이름 설정

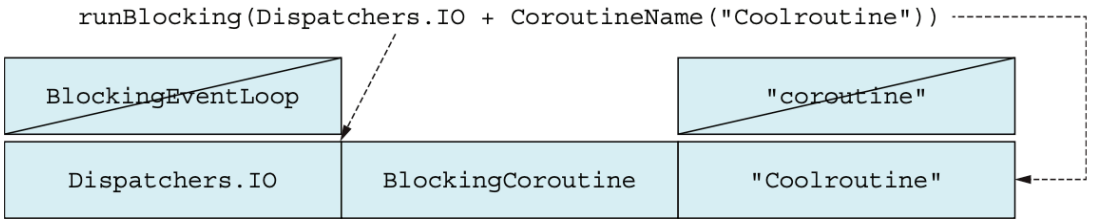


Figure 14.12 The parameters passed to `runBlocking` override the elements in the child's coroutine context: `Dispatchers.IO` replaces the special `BlockingEventLoop` dispatcher from `runBlocking`, and the name of the coroutine is set to `"Coolroutine"`.

- runBlocking에게 전달한 인자는 자식 코루틴의 컨텍스트의 원소를 덮어쓴다.
- IO는 runBlocking의 특별한 디스패처인 BlockingEvnetLoop 디스패처를 대신하며, 코루틴의 이름은 “Coolroutine” 으로 설정된다.

요약

- 동시성은 여러 작업을 동시에 처리하는 것을 의미하며, 여러 작업의 여러 부분이 서로 번갈아 실행되는 방식으로 나타난다.
- 병렬성은 물리적으로 동시에 실행되면서 현대 멀티코어 시스템을 효과적으로 활용하는 것을 말한다.

- 코루틴은 스레드 위에서 동시 실행을 위해 동작하는 경량 추상화다.
- 코틀린의 핵심 동시성 기본 요소는 일시 중단 함수로, 실행을 잠시 멈출 수 있는 함수다.
  - 다른 일시 중단 함수나 코루틴 안에서 일시 중단 함수를 호출할 수 있다.
- 반응형 스트림, 콜백, 퓨처 같은 다른 접근 방식과 달리 일시 중단 함수를 쓸 때는 코드의 모양이 달라지지 않는다.
  - 코드는 여전히 순차적으로 보인다.
- 코루틴은 일시 중단 가능한 계산의 인스턴스다.
- 코루틴은 스레드를 블로킹하는 문제를 피한다.
  - 스레드 블로킹이 문제가 되는 이유는 스레드 생성에 비용이 많이 들고, 시스템 자원이 제한적이기 때문이다.
- 코루틴 빌더를 사용해 새로운 코루틴을 생성할 수 있다.
- 디스패처는 코루틴이 실행될 스레드나 스레드 풀을 결정한다.
- 기본 제공되는 디스패처는 서로 다른 목적을 갖고 있다.
  - Default: 일반적인 용도
  - Main: UI 스레드
  - IO: 블로킹되는 IO 작업을 호출할 때 사용된다.
- Default나 IO 같은 대부분의 디스패처는 다중 스레드 디스패처이기 때문에 여러 코루틴이 병렬로 같은 데이터를 변경할 때 주의가 필요하다.
- 코루틴을 생성할 때 디스패처를 지정하거나 withContext를 사용해 디스패처를 변경할 수 있다.
- 코루틴 컨텍스트에는 코루틴과 연관된 추가 정보가 들어있다.
  - 코루틴 디스패처는 코루틴 컨텍스트의 일부다.