Chapter5 - async와 Deferred

launch 코루틴 빌더를 통해 생성되는 코루틴은 기본적으로 작업 실행 후 결과를 반환하지 않는다.
하지만 우리가 코루틴을 다룰 때는 코루틴으로부터 결과를 수신해야 하는 경우가 빈번하다. ex) 네트워크 통신을 실행 후 응답받아 처리

launch 코루틴 빌더는 결과값이 없는 코루틴 객체인 Job이 반환.
async 코루틴 빌더는 결과값이 있는 코루틴 객체인 Deferred를 반환.

5.1. async 사용해 결과값 수신하기

```
public fun CoroutineScope.launch(
    context: CoroutineContext = EmptyCoroutineContext,
    start: CoroutineStart = CoroutineStart.DEFAULT,
    block: suspend CoroutineScope.() -> Unit
): Job { ... }

public fun <T> CoroutineScope.async(
    context: CoroutineContext = EmptyCoroutineContext,
    start: CoroutineStart = CoroutineStart.DEFAULT,
    block: suspend CoroutineScope.() -> T
): Deferred<T>
```

launch는 코루틴에서 결과값이 반환되지 않기 때문에 Job 객체를 반환하는데, async는 코루틴에서 결과값을 담아 반환하기 위해 Deferred<\T> 타입의 객체를 반환한다.

Deferred 객체는 미래의 어느 시점에 결과값이 반환될 수 있음을 표현하는 코루틴 객체이다. 코루틴이 실행 완료될 때 결과값이 반환되므로 언제 결과값이 반환될지 정확히 알 수 없고, 만약 결과값이 필요하다면 결과값이 수신될 때까지 대기해야 한다.

Deferred 객체는 결과값 수신의 대기를 위해 await 함수를 제공한다.

await의 대상이 된 Deferred 코루틴이 실행 완료될 때까지 await 함수를 호출한 코루틴을 일시 중단하며, Deferred 코루틴이 실행 완료되면 결과값을 반환하고 호출부의 코루틴을 재개한다. 이러한 점에서 Job 객체의 join 함수와 유사하게 동작한다.

```
fun main() = runBlocking<Unit> {
    val networkDeferred: Deferred<String> = async(Dispatchers.IO) {
        delay(1000L) // 네트워크 요청
        return@async "Dummy Response" // 결과값 반환
    }
```

```
    val result = networkDeferred.await() // networkDeferred로부터 결과값이 반환될
때까지 runBlocking 일시 중단
    println(result) // Dummy Response 출력
}
```

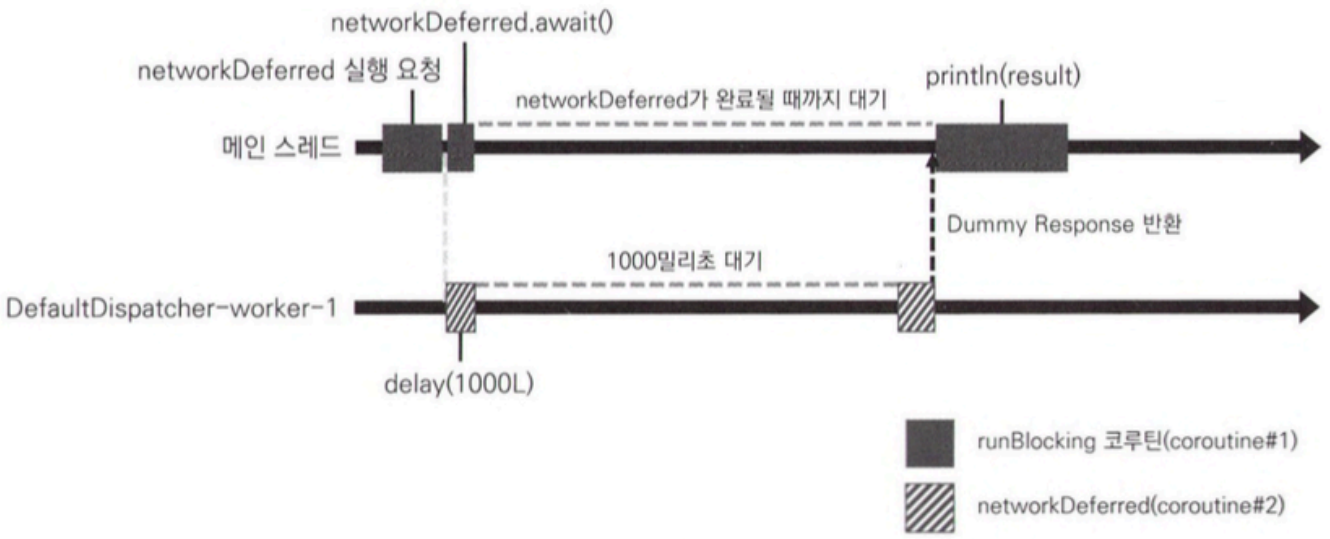


그림 5-1 await를 사용한 결과 수신

만약 await를 하지 않으면 어떻게 될까?

```
fun main() = runBlocking<Unit> {
    val networkDeferred: Deferred<String> = async(Dispatchers.IO) {
        delay(1000L) // 네트워크 요청
        return@async "Dummy Response" // 결과값 반환
    }
}
/*
// 결과:
(1초 후)
Process finished with exit code 0
/*
```

await 함수가 결과값 수신을 위해 호출부인 runBlocking 코루틴을 일시중단하는 역할을 하는데, 위 코드에서는 await 함수가 호출되지 않으므로 그대로 종료된다.

5.2. Deferred는 특수한 형태의 Job이다.

```
public interface Deferred<out T> : Job {
    public suspend fun await(): T
```

```
    ...
}
```

Deferred 인터페이스는 Job 인터페이스의 서브타입으로 몇 가지 기능이 추가되었을 뿐 여전히 Job 객체의 일종이다.
그래서 join 함수도 사용가능하고, isActive, isCancelled, isCompleted 같은 프로퍼티도 사용 가능하다.

5.3. 복수의 코루틴으로부터 결과값 수신하기

```
fun main() = runBlocking<Unit> {
    val startTime = System.currentTimeMillis() // 1. 시작 시간 기록
    val participantDeferred1: Deferred<Array<String>> =
async(Dispatchers.IO) { // 2. 플랫폼1에서 등록한 관람객 목록을 가져오는 코루틴
        delay(1000L)
        return@async arrayOf("James","Jason")
    }
    val participants1 = participantDeferred1.await() // 3. 결과가 수신 될 때까지
대기

    val participantDeferred2: Deferred<Array<String>> =
async(Dispatchers.IO) { // 4. 플랫폼2에서 등록한 관람객 목록을 가져오는 코루틴
        delay(1000L)
        return@async arrayOf("Jenny")
    }
    val participants2 = participantDeferred2.await() // 5. 결과가 수신 될 때까지
대기

    println("[${getElapsedTime(startTime)}] 참여자 목록:
${listOf(*participants1, *participants2)}") // 6. 지난 시간 표시 및 참여자 목록을 병
합해 출력
}
/*
// 결과:
[지난 시간: 2018ms] 참여자 목록: [James, Jason, Jenny]
*/
```

서로 간에 독립적인 작업을 처리할 때 await를 위와 같이 사용하게 되면, 순차적으로 처리하기 때문에 매우 비효율적이다. 독립적인 코루틴을 동시에 처리할 수 있는데도 코루틴의 이점을 버리는 것과 마찬가지다.

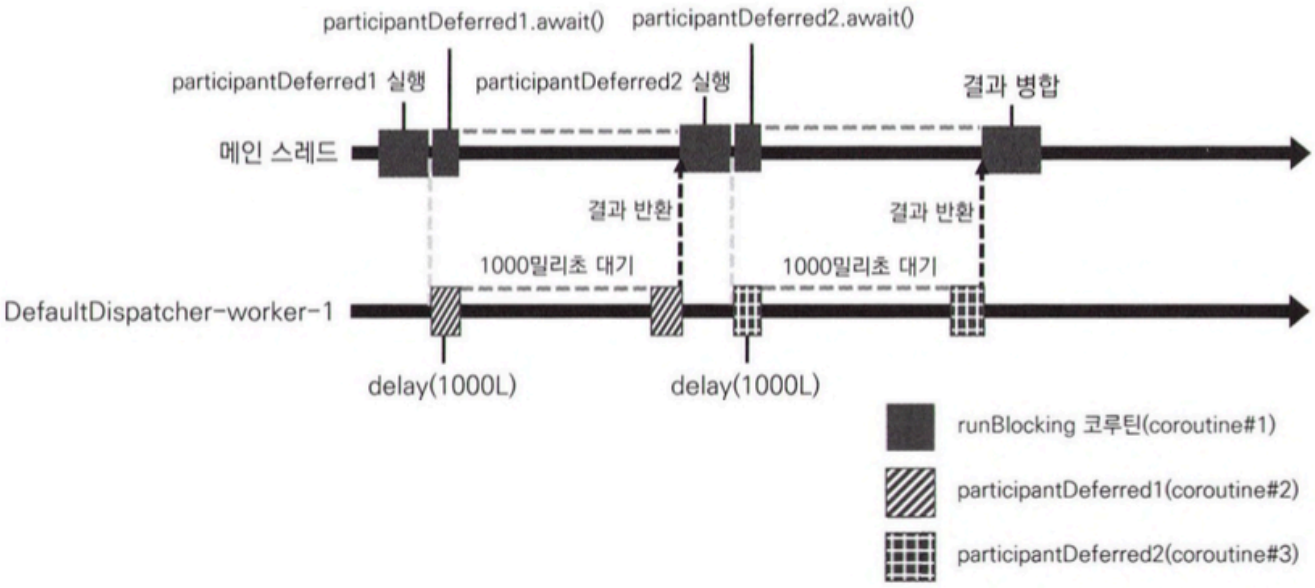


그림 5-2 순차 처리되는 await

```
fun main() = runBlocking<Unit> {
  val startTime = System.currentTimeMillis() // 1. 시작 시간 기록
  val participantDeferred1: Deferred<Array<String>> = async(Dispatchers.IO)
{ // 2. 플랫폼1에서 등록한 관람객 목록을 가져오는 코루틴
    delay(1000L)
    return@async arrayOf("James","Jason")
  }

  val participantDeferred2: Deferred<Array<String>> = async(Dispatchers.IO)
{ // 3. 플랫폼2에서 등록한 관람객 목록을 가져오는 코루틴
    delay(1000L)
    return@async arrayOf("Jenny")
  }

  val participants1 = participantDeferred1.await() // 4. 결과가 수신 될 때까지 대
기
  val participants2 = participantDeferred2.await() // 5. 결과가 수신 될 때까지 대
기

  println("[${getElapsedTime(startTime)}] 참여자 목록: ${listOf(*participants1,
*participants2)}") // 6. 지난 시간 기록 및 참여자 목록 병합
}
/*
// 결과:
[지난 시간: 1018ms] 참여자 목록: [James, Jason, Jenny]
*/
```

이러한 문제를 해결하기 위해서 await를 호출하는 위치를 모든 코루틴이 실행된 이후로 만들어야 한다.

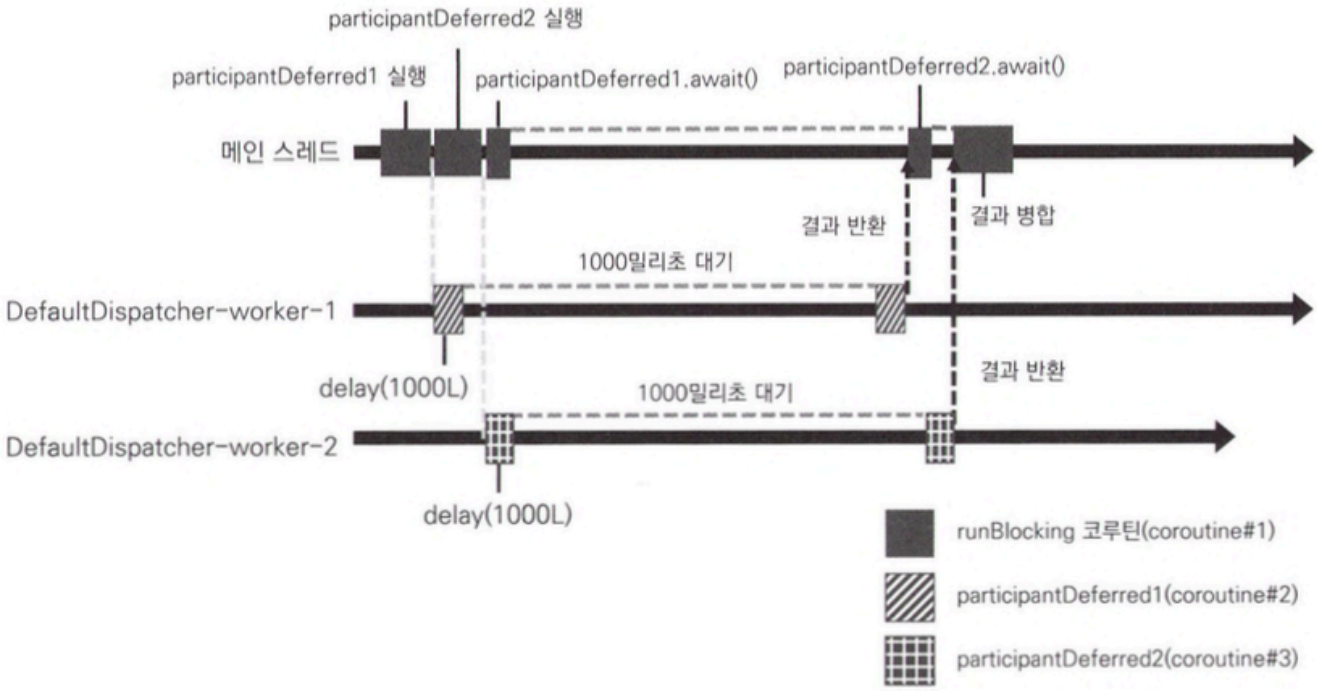


그림 5-3 함께 처리되는 async

```
// 여러 Deferred<T> 타입의 인자를 받을 수 있다.
public suspend fun <T> awaitAll(vararg deferreds: Deferred<T>): List<T> =
    if (deferreds.isEmpty()) emptyList() else AwaitAll(deferreds).await()

// Collection의 확장 함수로 선언된 awaitAll()
public suspend fun <T> Collection<Deferred<T>>.awaitAll(): List<T> =
    if (isEmpty()) emptyList() else AwaitAll(toTypedArray()).await()
```

독립적인 작업의 개수가 많다면 awaitAll() 함수를 통해서 한꺼번에 await 처리하여 코드를 좀더 간단하게 작성할 수 있다.

5.4. withContext

```
public suspend fun <T> withContext(
    context: CoroutineContext,
    block: suspend CoroutineScope.() -> T
): T { ... }
```

withContext 함수는 인자로 주어진 새로운 CoroutineContext 객체를 사용하여 block 람다식을 실행하고, 람다식을 벗어나면 다시 기존의 CoroutineContext 객체를 사용하여 코루틴을 재개하는 함수이다. 즉 block 안에서는 새로운 실행 환경이 적용되지만 block을 벗어나면 기존의 실행 환경으로 되돌아온다.

```
fun main() = runBlocking<Unit> {
  val networkDeferred: Deferred<String> = async(Dispatchers.IO) {
    delay(1000L) // 네트워크 요청
    return@async "Dummy Response" // 문자열 반환
  }
  val result = networkDeferred.await() // networkDeferred로부터 결과값이 반환될 때
까지 대기
  println(result)
}

fun main() = runBlocking<Unit> {
  val result: String = withContext(Dispatchers.IO) {
    delay(1000L) // 네트워크 요청
    return@withContext "Dummy Response" // 문자열 반환
  }
  println(result)
}
```

위와 같이 async-await 쌍을 withContext 함수로 대체가 가능하다.

하지만 실제 내부 동작은 async-await 쌍과 다르다.

```
fun main() = runBlocking<Unit> {
  println("[${Thread.currentThread().name}] runBlocking 블록 실행")
  async(Dispatchers.IO) {
    println("[${Thread.currentThread().name}] async 블록 실행")
  }.await()
}
/*
// 결과:
[main @coroutine#1] runBlocking 블록 실행
[DefaultDispatcher-worker-1 @coroutine#2] async 블록 실행
*/
```

async-await 쌍을 사용하면 runBlocking으로 생성된 코루틴과 async로 생성된 코루틴이 서로 다른 코루틴인 것을 확인할 수 있다. (coroutine#1, coroutine#2)

```
fun main() = runBlocking<Unit> {
  println("[${Thread.currentThread().name}] runBlocking 블록 실행")
  withContext(Dispatchers.IO) {
    println("[${Thread.currentThread().name}] withContext 블록 실행")
  }
}
/*
```

```
// 결과:
[main @coroutine#1] runBlocking 블록 실행
[DefaultDispatcher-worker-1 @coroutine#1] withContext 블록 실행
*/
```

하지만 withContext를 사용하면 실행하는 스레드는 달라도 실행하는 코루틴은 동일한 것을 확인할 수 있다. (coroutine#1)

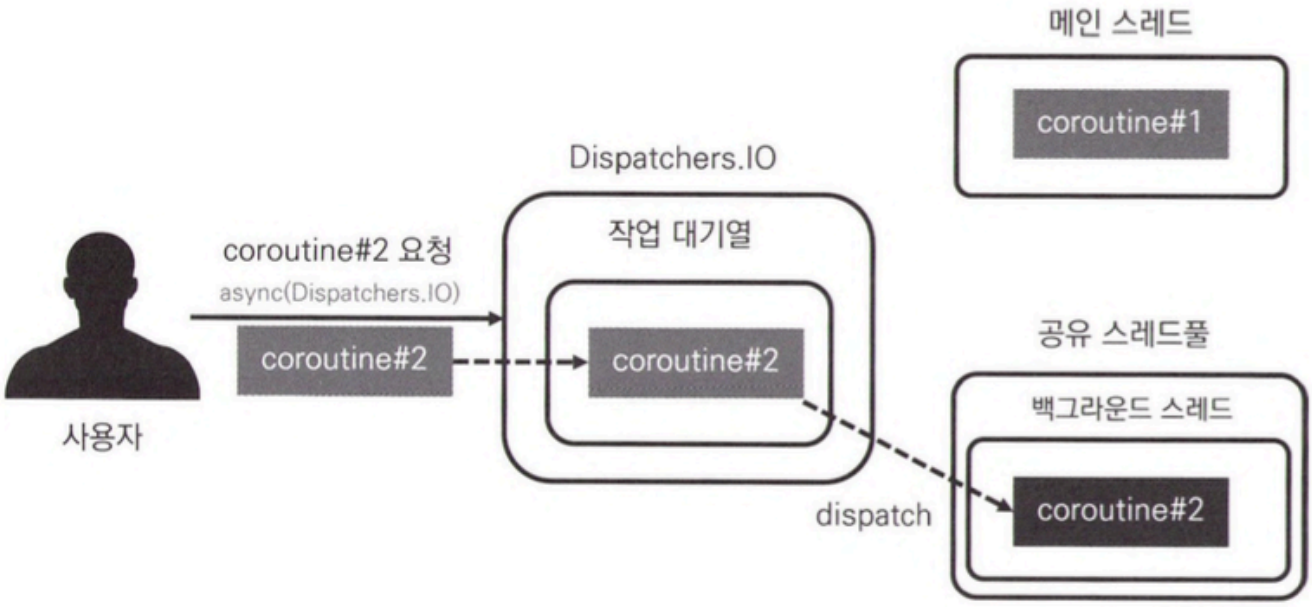
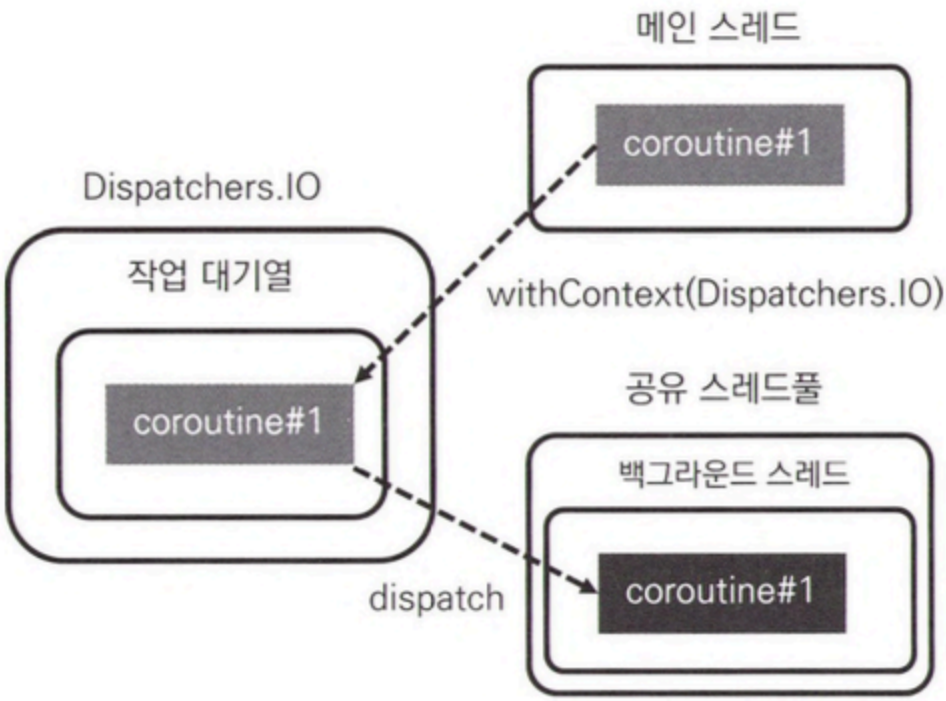


그림 5-6 async의 동작



* 백그라운드 스레드: DefaultDispatcher-worker-1

그림 5-5 withContext의 동작

async-await 쌍을 사용하면 새로운 코루틴 만들기 때문에 총 2개의 코루틴이 생성된다. 하지만 await 함수를 통해 순차 처리가 되어 동기적으로 실행되는 것이다. 반면에 withContext를 호출하면 코루틴이 유지된 채로 코루틴의 실행환경만 변경되기 때문에 동기적으로 실행되는 것이다.

추가적으로 블로그에 withContext 함수가 코루틴 빌더 함수라고 소개된 곳이 꽤 있는데, withContext 함수는 새로운 코루틴을 생성하는 함수가 아니기 때문에 코루틴 빌더 함수가 아니라고 생각한다.

withContext 사용시 주의점

```
fun main() = runBlocking<Unit> {
  val startTime = System.currentTimeMillis()
  val helloString = withContext(Dispatchers.IO) {
    delay(1000L)
    return@withContext "Hello"
  }

  val worldString = withContext(Dispatchers.IO) {
    delay(1000L)
    return@withContext "World"
  }

  println("[${getElapsedTime(startTime)}] ${helloString} ${worldString}")
}
/*
// 결과:
[지난 시간: 2018ms] Hello World
*/
```

withContext는 동기적으로 실행되기 때문에 복수의 독립적인 작업이 병렬로 실행돼야 하는 상황에서 withContext를 사용할 경우 비효율적이다. 아까 await 함수의 위치를 중간중간에 넣었을 때와 마찬가지인 문제이다.

그래서 이럴 때는 withContext를 사용하지 말고 async를 사용 후, await 함수 호출을 모든 코루틴이 실행된 뒤에 해야 한다. 혹은 awaitAll 함수를 호출하여 처리해야 한다.