Chapter18. 오류 처리와 테스트

다루는 내용

- 오류와 예외 발생 시 코드의 동작 제어
- 오류 처리를 구조적 동시성 개념과 연결하는 방법
- 시스템 일부가 실패해도 정상적으로 작동하는 코드를 작성하는 방법
- 동시성 코드를 위한 단위 테스트 작성법
- 테스트 실행 속도를 높이고 세밀한 동시성 제약 조건을 테스트하는 방법
- 터빈 라이브러리를 사용한 플로우 테스트

코틀린 코루틴을 사용할 때 오류처리는 15장에서 다뤘던 구조적 동시성과 깊게 얽혀 있다. 미처 잡아내지 않은 예외가 구조적 동시성의 계층을 따라 어떻게 처리되고 전파되는지, 플로우에서 오류를 어떻게 처리하는지, 이들의 동작을 제어하기 위해 사용할 수 있는 도구들이 무엇인지 다룬다.

애플리케이션의 강건성을 높이기 위한 또 다른 중요한 측면은 테스트다. 코루틴을 사용하는 테스트 코드를 작성하는 방법을 살펴보고, 가상 시간을 사용해 테스트를 실행하는 특별한 기능과 터빈 라이브러리를 사용해 플로우를 편리하게 테스트하는 방법을 살펴본다.

코루틴 내부에서 던져진 오류 처리

- 일시 중단 함수나 코루틴 빌더 안에 작성한 코드도 예외를 발생시킬 수 있다.
- 이런 예외를 처리하기 위해 launch나 async 호출을 try-catch로 감싸고 싶을 수 있다.
    - 하지만 그렇게 해도 효과가 없다.
    - 코루틴 빌더 함수이기 때문이다.
- 코루틴 빌더는 실행할 새로운 코루틴을 생성하는데, 이 새로운 코루틴에서 발생한 예외는 코루틴 빌더를 감싸고 있는 catch 블록에 의해 잡히지 않는다.

    - 마치 새로 생성된 스레드에서 발생한 예외가 스레드를 만든 코드에서 잡히지 않는 것과 같다.

```
fun main(): Unit = runBlocking {
    try {
        launch {
            throw UnsupportedOperationException("Ouch!")
        }
    } catch (u: UnsupportedOperationException) {
        println("Handled $u") // 실행되지 않음
    }
}
```

```
Exception in thread "main" java.lang.UnsupportedOperationException: Ouch!
    at ch18._1_NoThrowAcrossBoundariesKt$main$1$1.invokeSuspend(1_NoThro
...
```

- launch 빌더안에서 exception이 발생하는 경우 예외가 잡히지 않는다.

```
fun main(): Unit = runBlocking {
    launch {
        try {
            throw UnsupportedOperationException("Ouch!")
        } catch (u: UnsupportedOperationException) {
            println("Handled $u")
        }
    }
}

// Handled java.lang.UnsupportedOperationException: Ouch!
```

- 이 예외를 올바르게 처리하는 한 가지 방법은 launch에 전달되는 람다 블록 안에 try-catch 블록을 넣는 것이다.

- 이 경우 예외가 코루틴 경계를 넘지 않기 때문에 코루틴이 없었을 때처럼 처리될 수 있다.

```
fun main(): Unit = runBlocking {
    val myDeferredInt: Deferred<Int> = async {
        throw UnsupportedOperationException("Ouch!")
    }
    try {
        val i: Int = myDeferredInt.await()
        println(i)
    } catch (u: UnsupportedOperationException) {
        println("Handled: $u")
    }
}
```

```
Handled: java.lang.UnsupportedOperationException: Ouch!
Exception in thread "main" java.lang.UnsupportedOperationException: Ouch!
    at ch18._3_AsyncThrowsKt$main$1$myDeferredInt$1.invokeSuspend(3_Asy
...
```

- async로 생성된 코루틴이 예외를 던진다면 그 결과에 대해 await를 호출할 때 이 예외가 다시 발생한다.
- await가 원하는 타입의 의미 있는 값을 돌려줄 수 없기 때문에 예외를 던져야만 한다.
- await()를 try-catch로 감싸면 이 예외를 처리할 수 있다.
    - catch에서 예외를 잡는 것을 확인할 수 있지만, 동시에 콘솔에도 예외가 출력된다.
- 이는 await가 예외를 다시 던지지만 원래의 예외도 여전히 관찰되기 때문이다.
    - 이 예에서 async는 예외를 부모 코루틴인 runBlocking에 전파하고, 프로그램은 종료된다.
- 자식 코루틴은 잡히지 않은 예외를 항상 부모 코루틴에 전파한다.
    - 이는 부모 코루틴이 이 예외를 처리해야 할 책임을 가진다는 의미다.

코틀린 코루틴에서의 오류 전파

- 15장에서 구조적 동시성을 처음 접할 때 구조적 동시성의 주된 책임이 취소 처리 이외에 오류 처리라는 점을 간단히 언급한 적이 있다.
- 구조적 동시성 패러다임은 자식 코루틴에서 발생한 잡히지 않은 예외가 부모 코루틴에 의해 어떻게 처리되는지에 영향을 준다.
- 자식에게 작업을 나누는 방식에 따라 자식의 오류를 처리하는 방식도 달라진다.
- 자식 중 하나의 실패가 부모의 실패로 이어질 것인지 여부에 따라 2가지 방식으로 나눌 수 있다.
    - 코루틴이 작업을 동시적으로 분해해 처리하는 경우 자식 중 하나의 실패는 더 이상 최종 결과를 얻을 수 없다는 점을 의미한다.
        - 이런 경우 부모 코루틴도 예외로 완료돼야 하며, 여전히 작업 중인 다른 자식은 더 이상 필요가 없는 결과를 생성하는 것을 피하기 위해 취소된다.
        - 한 자식의 실패가 부모의 실패로 이어진다.
    - 하나의 자식이 실패해도 전체 실패로는 이어지지 않을 때
        - 자식들에게 벌어진 실패를 부모가 처리해야 하지만 자식의 실패로 인해 시스템 전체가 실패하면서 멈추지는 말아야 하는 경우를, 자식이 부모의 실행을 감독한다고 말한다.
        - 이러한 감독 코루틴은 일반적으로 코루틴 계층의 최상위에 위치한다.
- 코루틴에서 자식 코루틴을 부모가 어떻게 처리할지는 부모 코루틴의 컨텍스트에 Job과 SupervisorJob 중 어느 것이 있는지에 따라 달라진다.

자식이 실패하면 모든 자식을 취소하는 코루틴

- 14.8절에서 코루틴 컨텍스트를 설명할 때 코루틴 간의 부모-자식 계층이 Job 객체를 통해 구축된다는 점을 배웠다.
- 따라서 코루틴이 SupervisorJob 없이 생성된 경우 자식 코루틴에서 발생한 잡히지 않은 예외는 부모 코루틴을 예외로 완료시키는 방식으로 처리된다.
- 실패한 자식 코루틴은 자신의 실패를 부모에게 전파한다. 그러면 부모는 다음을 수행한다.
    - 불필요한 작업을 막기 위해 다른 모든 자식을 취소한다.
    - 같은 예외를 발생시키면서 자신의 실행을 완료시킨다.
    - 자신의 상위 계층으로 예외를 전파한다.

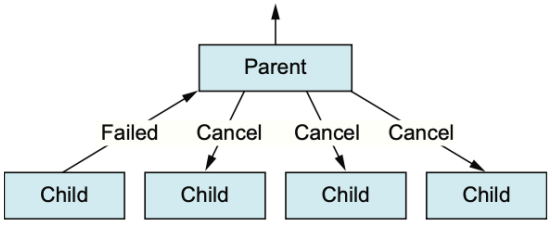

Figure 18.1 When child coroutines fail with an uncaught exception, they notify their parent. The parent, in turn, cancels all the sibling coroutines, and propagates the exception further up the coroutine hierarchy.

- 자식 코루틴이 잡히지 않는 예외로 실패하면 부모에게 통지한다.
- 부모는 형제 코루틴들을 모두 취소하고 예외를 코루틴 계층의 상위로 전달한다.
- 이런 동작은 같은 스코프 안에서 동시성 계산을 함께 수행하고 공통의 결과를 반환하는 코루틴 그룹에게 아주 유용하다.
- 이런 스코프의 코루틴 중 하나가 잡을 수 없는 예외로 인해 실패한다는 말은 그 코루틴이 스스로 해결할 수 없는 문제가 발생했다는 의미이며, 이런일이 벌어질 때 공통의 결과를 합리적으로 계산할 방법이 더 이상 없다는 가정이 필요하다.
- 따라서 이런 상황에 다른 형제 코루틴이 이제는 불필요해진 작업을 계속 수행하거나 자원을 계속 잡고 있는 것을 막기 위해 이들은 취소한다.

```
fun main(): Unit = runBlocking {
    launch {
        try {
            while (true) {
                println("Heartbeat!")
                delay(500.milliseconds)
            }
        } catch (e: Exception) {
            println("Heartbeat terminated: $e")
            throw e
        }
    }
    launch {
        delay(1.seconds)
        throw UnsupportedOperationException("Ow!")
    }
}
```

```
Heartbeat!
Heartbeat!
Heartbeat terminated: kotlinx.coroutines.JobCancellationException: Parent job
Exception in thread "main" java.lang.UnsupportedOperationException: Ow!
```

- 첫번째 코루틴 하트비트 역할
  - 루프 돌면서 메시지 출력
  - 예외가 발생하면 이 코루틴은 예외를 출력한 다음에 예외를 다시 던지면서 끝난다.
- 두번째 코루틴은 1초가 지난 후 예외를 던지는데, 이 예외를 잡아내지는 않는다.
- 결과를 보면 형제 코루틴 중 하나가 예외를 던지면 하트비트 코루틴도 취소된다.
- 기본적으로 이 예제와 같은 runBlocking을 포함한 모든 코루틴 빌더는 일반적인 감독이 아닌 코루틴을 생성한다.
  - 그렇기 대문에 한 코루틴이 잡히지 않은 예외로 종료되면 다른 자식 코루틴도 취소된다.
- 이런 오류 전파 동작은 launch로 시작된 코루틴뿐만 아니라 모든 코루틴에게도 적용된다.
  - 예를 들어 자식 코루틴을 async로 시작해도 같은 동작을 볼 수 있다.

구조적 동시성은 코루틴 스코프를 넘는 예외에만 영향을 미친다.

- 형제 코루틴을 취소하고 예외를 코루틴 계층 상위로 전파하는 이 동작은 코루틴 스코프를 넘는 처리되지 않은 예외에만 영향을 미친다.
- 따라서 이 동작을 피하는 가장 쉬운 방법은 처음부터 스코프를 넘는 예외를 던지지 않는 것이다.
- 한 코루틴 안에만 속해 있는 try-catch 블록은 예상대로 동작한다.

```
fun main(): Unit = runBlocking {
    launch {
        try {
            while (true) {
                println("Heartbeat!")
                delay(500.milliseconds)
```

```
            }
        } catch (e: Exception) {
            println("Heartbeat terminated: $e")
            throw e
        }
    }
    launch {
        try {
            delay(1.seconds)
            throw UnsupportedOperationException("Ow!")
        } catch (u: UnsupportedOperationException) {
            println("Caught $u")
        }
    }
}
```

```
Heartbeat!
Heartbeat!
Caught java.lang.UnsupportedOperationException: Ow!
Heartbeat!
Heartbeat!
Heartbeat!
```

- UnsupportedOperationException 를 직접 잡음으로써 예외가 발생한 다음에도 하트비트 코루틴이 계속 텍스트를 출력하는 것을 볼 수 있다.
- 처리되지 않은 예외를 코루틴 계층 위쪽으로 전파하고 형제 코루틴을 취소하는 것은 애플리케이션에서 구조적 동시성 패러다임을 강제하는 데 도움이 된다. 하지만 처리되지 않은 예외 하나가 전체 애플리케이션을 무너뜨려서는 안된다.
    - 대신 이 오류 전파에 대한 경계를 정의할 수 있어야 한다.
- 코루틴에서는 슈퍼바이저 코루틴을 사용해 경계를 정의할 수 있다.

💡 15장에서 살펴본 것처럼 코루틴에서 예외를 잡을 때는 취소 예외(CancellationException)와 그 모든 하위 타입을 주의해야한다. 취소는 코루틴 생명주기의 자연스러운 일부분이기 때문에 이런 예외를 코드가 삼켜서는 안된다. 대신 애초에 예외를 잡지 않거나, 다시 던져야 한다.

슈퍼바이저는 부모와 형제가 취소되지 않게 한다.

- 슈퍼바이저는 자식이 실패하더라도 생존한다.
- 일반 job을 사용하는 스코프와 달리 슈퍼바이저는 일부 자식이 실패를 보고하더라도 실패하지 않는다.
- 슈퍼바이저는 다른 자식 코루틴을 취소하지 않으며 예외를 구조적 동시성 계층 상위로 전파하지 않는다.
- 이 때문에 종종 코루틴 계층의 최상위 코루틴으로서 슈퍼바이저가 쓰인다.

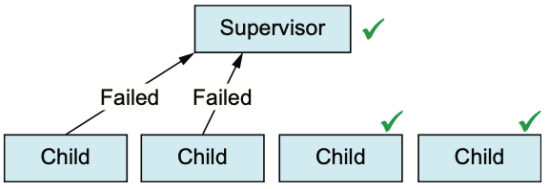

Figure 18.2 When one or more child coroutines of a supervisor fail, the sibling and parent coroutines all keep on working. The exception is no longer propagated further.

- 코루틴이 자식 코루틴의 슈퍼바이저가 되려면 그 코루틴에 연관된 job이 일반적이 job이 아니라 SupervisorJob이어야 한다.
- SupervisorJob은 Job과 마찬가지로 역할을 하지만 예외를 부모에게 전파하지 않으며, 다른 자식 작업이 실패해도 취소되지 않게 한다.
- 물론 SupervisorJob도 구조적 동시성에 참여해 여전히 취소될 수 있고 취소 예외를 올바르게 전파한다.

```
fun main(): Unit = runBlocking {
    supervisorScope {
        launch {
            try {
                while (true) {
```

```
                println("Heartbeat!")
                delay(500.milliseconds)
            }
        } catch (e: Exception) {
            println("Heartbeat terminated: $e")
            throw e
        }
    }
    launch {
        delay(1.seconds)
        throw UnsupportedOperationException("Ow!")
    }
  }
}
```

```
Heartbeat!
Heartbeat!
Exception in thread "main" java.lang.UnsupportedOperationException: Ow!
...
Heartbeat!
Heartbeat!
...
```

- 슈퍼바이저의 동작을 직접 확인하려면 supervisorScope 함수를 사용해 스코프를 만들 수 있다.
    - 15.1.1 절에서 살펴본 coroutineScope 함수와 비슷하지만 중요한 차이점이 있다.
    - 자식 코루틴 중 하나가 실패해도 형제 코루틴이 종료되지 않고 처리되지 않은 예외는 더 이상 전파되지 않는다.
    - 대신 부모 코루틴과 형제 코루틴은 모두 계속 작동한다.
- 따라서 하트비트 코루틴이 계속 실행되도록 리스트를 수정하려면 launch 호출을 supervisorScope로 감싸면 된다.
    - 예외가 발생한 후에도 하트비트 코루틴이 계속 작동하는 것을 확인할 수 있다.
    - 자식 코루틴이 부모 코루틴을 취소하지 못하게 슈퍼바이저가 막은 것
- 그렇다면 왜 예외가 출력됐는데도 애플리케이션이 계속 실행될 수 있었을까?

    - SupervisorJob이 launch 빌더로 시작된 자식 코루틴에 대해 CoroutineExceptionHandler를 호출하기 때문이다.
    - 18.3절에서 자세히 보자
- 코루틴을 지원하는 프레임워크는 종종 슈퍼바이저 역할을 하는 코루틴 스코프를 기본적으로 제공한다.
    - 예를 들어 ktor의 Application 스코프를 개별 요청 핸들러의 수명보다 더 오래 실행되는 코루틴을 시작할 때 쓸 수 있다.
        - 이런 코루틴은 전체 케이토 애플리케이션이 실행되는 동안 계속 살아 있을 수 있다.
        - Application 스코프는 슈퍼바이저 역할도 하며, 어느 한 코루틴에서 처리되지 않은 예외가 발생해도 전체 애플리케이션을 중단시키지 않는다.
    - 반면 케이토의 PipelineContext는 한 요청 핸들러와 같은 수명을 가진 코루틴을 책임진다.
        - 여기서 가정은 PipelineContext 내에서 여러 코루틴이 함께 작업하면서 요청에 대한 응답을 계산하다는 것이다.
        - 따라서 어느 한 코루틴이 처리되지 않은 예외로 실패하면 더 이상 결과를 계산할 합리적인 방법이 없기 때문에 해당 요청과 관련된 다른 코루틴도 함께 취소된다.
    - 슈퍼바이저는 애플리케이션에서 코루틴 계층의 위쪽에 위치하는 경우가 많다.

CoroutineExceptionHandler: 예외 처리를 위한 마지막 수단

- 자식 코루틴은 처리되지 않은 예외를 부모 코루틴에 전파한다.
- 이 때 예외가 슈퍼바이저에 도달하거나 계층의 최상위로 가서 부모가 없는 루트 코루틴에 도달하면 예외는 더 이상 전파되지 않는다.
- 이 시점에서 처리되지 않은 예외는 CoroutineExceptionHandler 라는 특별한 핸들러에게 전달된다.
    - 이 핸들러는 코루틴 컨텍스트의 일부다
- 코루틴 컨텍스트에 예외 핸들러가 없다면 처리되지 않은 예외는 시스템 전역 예외 핸들러로 이동한다.

- CoroutineExceptionHandler를 코루틴 컨텍스트에 제공하면 처리되지 않은 예외를 처리하는 동작을 커스텀화할 수 있다.
- 코틀린 프레임워크는 자체적으로 코루틴 예외 핸들러를 제공할 수 있다.
    - 예를 들어 케이토는 CoroutineExceptionHandler 를 사용해 처리되지 않은 예외의 문자열 표현을 로그 프로바이더에 보낸다.
    - 반면 안드로이드 ViewModel은 viewModelScope와 연관된 SupervisorJob의 컨텍스트에서 CoroutineExceptionHandler 를 지정하지 않는다.
    - 따라서 viewModelScope에서 시작된 코루틴에서 launch를 사용해 처리되지 않은 예외가 발생하면 앱이 종료된다.
- 핸들러 정의는 간단하다. 이 핸들러는 코루틴 컨텍스트와 처리되지 않은 예외를 람다의 파라미터로 받는다.
- CoroutineExceptionHandler를 코루틴 컨텍스트의 원소로 추가할 수 있다.

```
class ComponentWithScope(dispatcher: CoroutineDispatcher = Dispatchers.D
    private val exceptionHandler = CoroutineExceptionHandler { _, e →
        println("[ERROR] ${e.message}")
    }

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

    fun action() = scope.launch {
        throw UnsupportedOperationException("Ouch!")
    }
}


fun main() = runBlocking {
    val supervisor = ComponentWithScope()
    supervisor.action()
    delay(1.seconds)
}
```

```
// [ERROR] Ouch!
```

- 이 컴포넌트의 스코프에서 SupervisorJob을 정의해 스코프를 슈퍼바이저로 만들고, 사용자 정의 예외 핸들러를 코루틴 컨텍스트의 요소로 지정했다.
- 출력 결과는 예외가 커스텀 예외 핸들러에 의해 처리됐음을 보여준다.

내부적으로는 슈퍼바이저의 직접적인 자식들이 컨텍스트의 커스텀 코루틴 예외 핸들러(이런 핸들러가 정의된 경우)에 예외를 전달하거나 디폴트 핸들러에게 예외를 전달하는 방식으로 예외를 처리한다. 단순화를 위해 이런 식으로 처리가 돼도 슈퍼바이저가 자식들의 예외를 처리하는 것으로 생각할 수 있다. 또한 코루틴 예외 핸들러는 코루틴 계층에서 최상위 코루틴이 launch 빌더로 시작된 경우에만 호출된다는 점도 기억할 필요가 있다.

- 자식 코루틴, 즉 코루틴 스코프에서 시작된 코루틴이나 다른 코루틴에서 시작된 코루틴은 처리되지 않은 예외의 처리를 부모에게 위임한다는 점을 다시 한 번 기억하자.
- 부모는 다시 이 처리를 자신의 부모에게 위임하며 계층의 최상위에 이를 때까지 이런 위임이 계속된다.
- 따라서 중간에 있는 CoroutineExceptionHandler 라는 것은 존재하지 않는다.
- 루트 코루틴이 아닌 코루틴의 컨텍스트에 설치된 핸들러는 결코 사용되지 않는다.

```
private val topLevelHandler = CoroutineExceptionHandler { _, e →
    println("[TOP] ${e.message}")
}

private val intermediateHandler = CoroutineExceptionHandler { _, e →
    println("[INTERMEDIATE] ${e.message}")
}

@OptIn(DelicateCoroutinesApi::class) // 미묘한 API를 명시적으로 사용하게 한다.
fun main() {
    GlobalScope.launch(topLevelHandler) {
```

```
        launch(intermediateHandler) {
            throw UnsupportedOperationException("Ouch!")
        }
    }
    Thread.sleep(1000)
}

// [TOP] Ouch!
```

- GlobalScope.launch api를 사용해 루트 코루틴을 생성하고 커스텀 코루틴 예외 핸들러를 그 컨텍스트의 일부로 제공한다.
- 또한 중간 예외 핸들러를 launch 코루틴에게 제공한다.
- 이 코드를 실행하면 계층의 최상위에 있는 코루틴 예외 핸들러만 실행되며 중간에 있는 핸들러는 쓰이지 않는다는 사실을 알 수 있다.
- 이는 예외가 여전히 부모 코루틴에게 전파될 수 있기 때문이다.

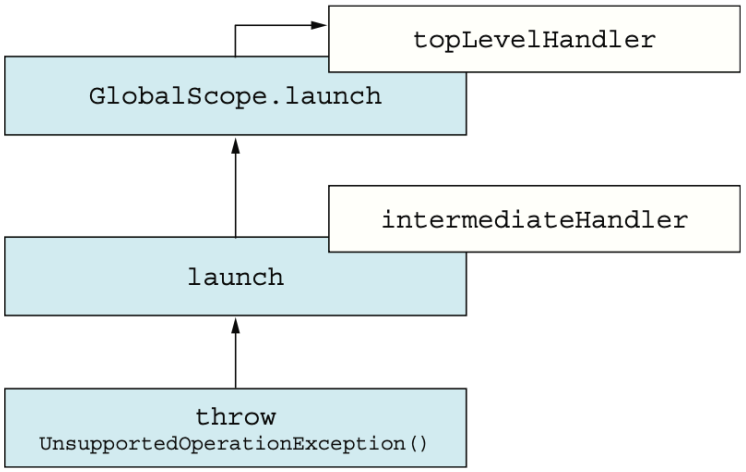

Figure 18.3 Even though the intermediate call to `launch` has a coroutine exception handler in its coroutine context, it's not a root coroutine. Therefore, the exception continues to be propagated along the hierarchy of coroutines. Only the exception handler at the root coroutine (here, `GlobalScope.launch`) is invoked.

- 중간에 launch 호출에 코루틴 예외 핸들러가 있음에도 루트 코루틴이 아니기 때문에 예외가 계속해서 코루틴 계층을 따라 전파한다.
- 그 결과 최상위 코루틴인 GlobalScope.launch의 예외 핸들러만 호출된다.

CoroutineExceptionHandler를 launch와 async에 적용할 때의 차이점

- CoroutineExceptionHandler를 살펴볼 때 예외 핸들러는 계층의 최상위 코루틴이 laucnh로 생성된 경우에만 호출된다.
- 최상위 코루틴이 async로 생성된 경우에는 CoroutineExceptionHandler가 호출되지 않는다는 뜻이다.

```
class ComponentWithScope(dispatcher: CoroutineDispatcher = Dispatchers.D
    private val exceptionHandler = CoroutineExceptionHandler { _, e →
        println("[ERROR] ${e.message}")
    }

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

    fun action() = scope.launch { // launch - async
        async {
            throw UnsupportedOperationException("Ouch!")
        }
    }
}

fun main() = runBlocking {
    val supervisor = ComponentWithScope()
    supervisor.action()
    delay(1.seconds)
}

// [ERROR] Ouch!
```

- launch로 코루틴을 시작하고 내부에서 async를 사용해 코루틴을 추가로 사작한다.
- 슈퍼바이저 밑의 최상위 코루틴이 launch로 시작된 경우에는 CoroutineExceptionHandler가 호출된다.

```
class ComponentWithScope(dispatcher: CoroutineDispatcher = Dispatchers.D
    private val exceptionHandler = CoroutineExceptionHandler { _, e →
```

```
        println("[ERROR] ${e.message}")
    }


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


    fun action() = scope.async { // async - launch
        launch {
            throw UnsupportedOperationException("Ouch!")
        }
    }
}


fun main() = runBlocking {
    val supervisor = ComponentWithScope()
    supervisor.action()
    delay(1.seconds)
}

// 출력에 아무것도 표시되지 않음
```

- 이제 최상위 코루틴을 async로 변경
- 코루틴 예외 핸들러가 호출되지 않는 것을 확인할 수 있다.
- 최상위 코루틴이 async로 시작되면 이 예외를 처리하는 책임은 await()을 호출하는 Deferred의 소비자에게 있다.
    - 따라서 코루틴 예외 핸들러는 이 예외를 무시할 수 있다.
- 그리고 소비자 코드는 await 호출을 try-catch 블록으로 감싸는 방식으로 예외를 처리할 수 있다.
    - 이 경우에는 try-catch가 코루틴 취소에 영향을 끼치지 못한다.
    - 이 예제의 scope에 SupervisorJob이 없었다면 처리되지 않은 예외가 여전히 스코프의 다른 자식 코루틴들을 모두 취소시켰을 것이다.

플로우에서 예외 처리(다시 읽기)

- 플로우도 예외를 던질 수 있다.

```
class UnhappyFlowException : Exception()

val exceptionalFlow = flow {
    repeat(5) { number →
        emit(number)
    }
    throw UnhappyFlowException()
}
```

- 예제의 플로우를 수집하면 5개의 원소가 배출된 다음에 UnhappyFlowException라는 커스텀 예외가 발생한다.
- 일반적으로 플로우의 일부분(플로우가 생성되거나 변환되거나 수집되는 중에)에서 예외가 발생하면 collect에서 예외가 던져진다.
    - 이는 collect 호출을 try-catch 블록으로 감싸면 예상대로 동작한다는 뜻이다.
- 이 때 플로우가 중간 연산자가 적용됐는지 여부와는 관계 없다.

```
fun main() = runBlocking {
    val transformedFlow = exceptionalFlow.map {
        it * 2
    }
    try {
        transformedFlow.collect {
            print("$it ")
        }
    } catch (u: UnhappyFlowException) {
        println("\nHandled: $u")
    }
    // 0 2 4 6 8
    // Handled: UnhappyFlowException
}
```

- map 함수를 통해 exceptionalFlow를 transformedFlow로 변환한 후 collect 호출을 try-catch 감싼 모습
- 출력을 보면 예외가 제대로 처리됐음을 확인할 수 있다.
    - 하지만 더 복잡하고 긴 플로우 파이프라인을 구축할 때는 catch 연산자를 사용하는 쪽이 더 편리하다.

catch 연산자로 업스트림 예외 처리

- catch 는 플로우에서 발생한 예외를 처리할 수 있는 중간 연산자다.
- 이 함수에 연결된 람다 안에서 플로우에 발생한 예외에 접근할 수 있다.
    - 이 때 예외는 람다의 파라미터로 전달된다.
- 이 연산자는 취소 예외를 자동으로 인식하기 때문에 취소가 발생한 경우에는 catch 블록이 호출되지 않는다.
    - 게다가 catch는 스스로 값을 방출할 수도 있기 때문에 예외를 오류 값으로 변환해 다운스트림 플로우에서 소비할 수도 있다.

```
fun main() = runBlocking {
    exceptionalFlow
        .catch { cause →
            println("\nHandled: $cause")
            emit(-1)
        }
        .collect {
            print("$it ")
        }
}

// 0 1 2 3 4
// Handled: ch18.UnhappyFlowException
// -1
```

- exceptionalFlow 에서 발생한 예외를 잡아 로그에 기록하고 수집자에게 오류 값으로 -1을 방출

- 여기서 catch 연산자가 오직 업스트림에 대해서만 작동하며, 플로우 처리 파이프라인의 앞쪽에서 발생한 예외들만 잡아낸다는 사실을 기억하자.

```
fun main() = runBlocking {
    exceptionalFlow
        .map {
            it + 1
        }
        .catch { cause →
            println("\nHandled $cause")
        }
        .onEach {
            throw UnhappyFlowException()
        }
        .collect()
}

// Exception in thread "main" ch18.UnhappyFlowException
...
```

- catch 호출 다음에 위치한 onEach 람다에서 발생한 예외는 잡히지 않는다.

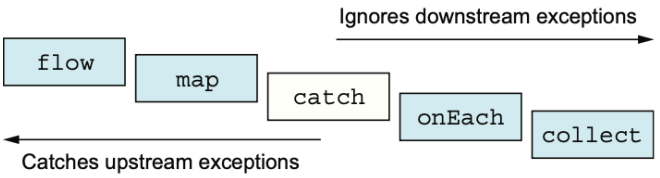

Figure 18.4 The `catch` operator only catches exceptions that occur upstream. Exceptions occurring downstream are unaffected by the `catch` operator.

- catch 연산자는 업스트림에서 발생한 예외만 처리
- 다운스트림에서 발생한 예외는 영향을 받지 않는다.

💡 collect 람다 안에서 발생한 예외를 처리하려면 collect 호출을 try-catch 블록으로 감싸면 된다. 그 대신 onEach, catch, collect 사슬을 람다 없이 사용해 논리를 재작성할 수도 있다. 여기서 중요한 점은 예외가 발생할 수 있는 지점 다음에 catch 연산자가 위치해야 한다는 것이다. catch 연산자는 업스트림에서 발생한 예외만 처리하므로 예외를 catch 블록에서 다시 던져서 다운스트림에 있는 다른 catch 연산자에서 처리하게 하는 것도 완전히 올바른 코드다.

술어가 참일 때 플로우의 수집 재시도: retry 연산자

- 플로우를 처리하는 도중에 예외가 발생했을 때 단순히 오류 메시지와 함께 종료하는 대신 작업을 재시도하고 싶을 수 있다.
- 내장된 retry 연산자는 이를 매우 편리하게 만들어준다.
- catch 와 마찬가지로 retry는 업스트림의 예외를 잡는다.
    - 예외를 처리하고 Boolean 값을 반환하는 람다를 사용할 수 있다.
- 람다가 true를 반환하면 재시도가 시작되며, 재시도 동안 업스트림의 플로우가 처음부터 다시 수집되면서 모든 중간 연산이 다시 실행된다.

```
class CommunicationException : Exception("Communication failed!")

val unreliableFlow = flow {
    println("Starting the flow!")
    repeat(10) { number →
        if (Random.nextDouble() < 0.1) throw CommunicationException()
        emit(number)
    }
}

fun main() = runBlocking {
    unreliableFlow
        .retry(5) { cause →
            println("\nHandled: $cause")
            cause is CommunicationException
```

```
        }
        .collect { number ->
            print("$number ")
        }
}
```

- 10개의 원소를 배출하려고 하는 불안정한 플로우를 시뮬레이션
- 각각의 배출은 10% 확률로 실패할 수 있으며, retry 연산자를 사용해 플로우를 다시 수집한다.

```
Starting the flow!
0 1 2 3 4 5 6
Handled: ch18.CommunicationException: Communication failed!
Starting the flow!
0 1 2 3
Handled: ch18.CommunicationException: Communication failed!
Starting the flow!
0 1 2 3 4 5
Handled: ch18.CommunicationException: Communication failed!
Starting the flow!
0 1
Handled: ch18.CommunicationException: Communication failed!
Starting the flow!
0 1 2 3 4 5 6 7
Handled: ch18.CommunicationException: Communication failed!
Starting the flow!
0 1

Exception in thread "main" ch18.CommunicationException: Communication fai
```

- 재시도할 때는 업스트림의 연산자가 모두 다시 실행된다는 점을 기억해야 한다.
- 업스트림의 플로우가 부수 효과를 일으키는 작업을 수행하는 경우 이런 작업이 여러번 실행되는 것을 볼 수 있다.
    - 이 경우 작업이 멱등성을 갖거나 반복 실행이 다른 방식으로 올바르게 처리되는지 확인해야 한다.

코루틴과 플로우 테스트

- 테스트 메서드에서 코루틴을 사용하려면 runTest 코루틴 빌더를 사용하면 된다.
- 이런 별도의 코루틴 빌더가 필요한 이유와 왜 runBlocking으로는 충분하지 않을까?
  - 14.6.1절에서 살펴본 것처럼 runBlocking 빌더 함수는 일반 코틀린 코드와 동시성 코틀린 코드 사이에 다리를 놓는 역할을 하기 때문에 일시 중단 함수나 코루틴, 플로우를 사용하는 코드를 테스트할 때도 이를 쓸 수 있다.
    - 그렇지만 단점이 있다.
  - runBlocking을 사용하면 테스트가 실시간으로 실행된다.
    - 이는 코드에 delay가 지정된 경우에 결과가 계산되기 전에 시간 지연이 전부 실행된다는 뜻이다.
    - 테스트 스위트가 커지면 이런 불필요한 긴 실행 시간이 누적되면서 테스트 속도가 느려질 수 있다.
  - 코루틴은 이에 대한 해결책으로 가상 시간을 사용한 테스트 실행을 제공한다.

코루틴을 사용하는 테스트를 빠르게 만들기: 가상 시간과 테스트 디스패처

- 모든 테스트를 실시간으로 실행해서 지연을 기다리느라 테스트를 느리게 실행하는 대신, 코루틴은 가상 시간을 사용해 테스트 실행을 빠르게 진행할 수 있게 해준다.
- 가상 시간을 사용할 때는 지연이 자동으로 빠르게 진행되기 때문에 앞에서 언급한 문제를 해결할 수 있다.

```
class PlaygroundTest {
    @Test
    fun testDelay() = runTest {
        val startTime = System.currentTimeMillis()
        delay(20.seconds)
        println(System.currentTimeMillis() - startTime)
        // 11
```

```
    }

}
```

- 20초의 delay를 선언했음에도 실질적으로 즉시 실행되며, 몇 밀리초 만에 완료된다.
- runTest는 속도를 높이기 위해 특별한 테스트 디스패처와 스케줄러를 사용한다.
- 따라서 실제로 코루틴의 지연시간을 기다리지 않고 빠르게 진행시킨다.

인위적으로 지연 시간이 자동으로 빠르게 진행되기 때문에 runTest는 기본적으로 타임아웃을 실제 시간으로 60초로 지정한다. 때로는 더 많은 시간이 필요할 수도 있다. 그런 경우 runTest를 호출할 때 timeout 파라미터를 지정할 수 있다.

- runBlocking과 마찬가지로 runTest의 디스패처는 단일 스레드다.
- 따라서 기본적으로 모든 자식 코루틴은 동시에 실행되며 테스트 코드와 병렬로 실행되지 않는다.
    - 다중 스레드 디스패처를 명시적으로 지정한 자식 코루틴은 예외
- 15.2.4 취소를 살펴봤을 때처럼 단일 스레드 디스패처를 공유하는 경우 다른 코루틴이 코드를 실행하려면 코드가 일시 중단 지점을 제공해야 하며, runTest도 예외는 아니다.
    - 테스트 단언문을 작성할 때 특히 이를 감안해야 한다.

```
@Test
fun testDelay() = runTest {
    var x = 0
    launch {
        x++
    }

    launch {
        x++
    }
```

```
}

assertEquals(2, x)
```

- runTest 본문에 일시 중단 지점이 없기 때문에 다음 테스트의 단언문은 실패한다.
- delay, yield 등 다른 일시 중단 함수 호출을 추가해 일시 중단 지점을 추가하면 테스트가 통과한다.
- 또한 테스트 디스패처에서는 TestCoroutineScheduler를 통해 가상 시간을 더 세밀하게 제어할 수 있다.
    - 이 스케줄러는 코루틴 컨텍스트의 일부다.
- runTest 빌더 함수의 블록안에서는 TestScope라는 특수한 스코프에 접근할 수 있으며, 이 스코프는 TestCoroutineScheduler 기능을 사용할 수 있게 해준다.
- 이 스케줄러의 핵심 함수는 다음과 같다.
    - runCurrent: 현재 실행하게 예약된 모든 코루틴을 실행한다.
    - advanceUntilIdle: 예약된 모든 코루틴을 실행한다.

```
@OptIn(ExperimentalCoroutinesApi::class)
@Test
fun testDelay() = runTest {
    var x = 0
    launch {
        delay(500.milliseconds)
        x++
    }
    launch {
        delay(1000.milliseconds)
        x++
    }
    println(currentTime) // 0
    delay(600.milliseconds)
    assertEquals(1, x)
    println(currentTime) // 600

  delay(500.milliseconds)
```

```
    assertEquals(2, x)
    println(currentTime) // 1100
}
```

- 가상 디스패처의 시계를 단순히 앞으로 이동시키려면 delay를 사용할 수 있다.
- 가상시계이므로 이 지연은 즉시 완료되며, 지연이 끝나는 가상 시간보다 이전에 실행하도록 예약된 코드도 실행된다.
- 가상 디스패처의 현재 시간이 궁금하다면 currentTime 속성을 사용할 수 있다.

```
@OptIn(ExperimentalCoroutinesApi::class)
@Test
fun testDelay() = runTest {
    var x = 0
    launch {
        x++
        launch {
            x++
        }
    }
    launch {
        delay(1000.milliseconds)
        x++
    }
    runCurrent()
    assertEquals(2, x)
    advanceUntilIdle()
    assertEquals(3, x)


}
```

- 현재 실행되도록 스케줄이 돼 있는 모든 코루틴을 실행할 때는 runCurrent 함수를 사용할 수 있다.
- 이 과정에서 즉시 실행할 새 코루틴이 예약되면 그런 코루틴도 직접 실행된다.

- 미래의 어느 시점에 실행하도록 예약된 코루틴까지 실행하려면 advanceUntilIdle 함수를 사용할 수 있다.

터빈으로 플로우 테스트

```
val myFlow = flow {
    emit(1)
    emit(2)
    emit(3)
}

@Test
fun doTest() = runTest {
    val results = myFlow.toList()
    assertEquals(3, result.size)

}
```

- 플로우를 사용하는 코드를 테스트하는 것도 runTest를 사용하는 다른 일시 중단 코드 테스트와 본질적으로 다르지 않다.
- 예를 들어 toList를 호출해서 유한한 풀로우의 모든 원소를 먼저 컬렉션에 수집한 다음, 기대한 모든 원소가 실제로 결과 컬렉션에 있는지 확인할 수 있다.

```
@Test
fun doTest() = runTest {
    val results = myFlow.test {
        assertEquals(1, awaitItme())
        assertEquals(2, awaitItme())
        assertEquals(3, awaitItme())
        awaitComplete()
    }

}
```

- 무한한 플로우나 더 까다로운 불변성을 다뤄야할 수도 있다.
    - 플로우 테스트 작성에 도움을 주는 터빈 라이브러리르 통해 이런 경우를 지원할 수 있다.
- 터빈의 핵심 기능은 플로우의 확장 함수인 test 함수
    - test 함수는 새코루틴을 실행하며 내부적으로 플로우를 수집한다.
    - test의 람다에서 awaitItem, awaitComplete, awaitError 함수를 테스트 프레임워크의 일반 단언문과 함께 사용해서 플로우에 대한 불변 조건을 지정하고 검증할 수 있다.
    - 또한 플로우가 방출한 모든 원소가 테스트에 의해 적절히 소비되도록 보장한다.
- 또한 터빈은 여러 플로우를 결합해 테스트하거나 테스트를 위해 시스템 일부를 대체할 수 있는 독립적인 터빈 객체를 만드는 기능도 제공한다.

요약

- 한 코루틴에만 국한된 예외는 코루틴이 아닌 일반적인 코드와 마찬가지로 처리할 수 있다.
    - 코루틴 경계를 넘는 예외는 좀 더 주의를 기울여야 한다.
- 기본적으로 코루틴에서 처리되지 않은 예외가 발생하면 부모 코루틴과 모든 형제 코루틴이 취소된다.
    - 이를 통해 구조적 동시섬 개념이 강제로 적용된다.
- supervisorScope나 SupervisorJob을 사용하는 다른 코루틴 스코프에서 사용되는 슈퍼바이저는 자식 코루틴 중 하나가 실패해도 다른 자식 코루틴을 취소하지 않는다.
    - 또한 처리하지 않은 예외를 코루틴 계층의 위로 전파하지도 않는다.
- await는 async 코루틴에서 발생한 예외를 다시 던진다.
- 슈퍼바이저 애플리케이션에서 오랫동안 실행되는 부분에 자주 사용된다.
    - 종종 케이토의 Application처럼 프레임워크에 내장된 부품으로 제공되는 경우도 있다.
- 처리되지 않은 예외는 슈퍼바이저를 만나거나 코루틴 계층의 최상단에 도달할 때까지 전파된다.
    - 이 시점에서 처리되지 않은 예외는 코루틴 컨텍스트의 일부인 CoroutineExceptionHandler에게 전달된다.

    - 컨텍스트에 코루틴 예외 핸들러가 없으면 시스템의 전역 예외 핸들러에게 전달된다.
- jvm과 안드로이드에서 기본 시스템 예외 핸들러가 다르다.
    - jvm에서는 스택트레이스를 오류 콘솔에 기록한다. 안드로이드는 오류를 발생시키면서 애플리케이션 중단시킨다.
- CoroutineExceptionHandler는 예외를 처리하는 마지막 수단으로 예외를 잡을 수는 없지만, 예외가 기록되는 방식을 사용자 정의할 수 있다.
    - CoroutineExceptionHandler는 계층의 최상단에 있는 루트 코루틴의 컨텍스트에 위치한다.
- 최상단 코루틴을 launch 빌더를 시작한 경우에만 CoroutineExceptionHandler가 호출된다.
    - async 빌더로 시작한 경우에는 이 핸들러가 호출되지 않으며, Defeered를 기다리는 코드가 예외를 처리해야 한다.
- 플로우를 오류 처리는 collect를 try-catch 문으로 감싸거나 전용 catch 연산자를 사용한다.
- catch 연산자는 업스트림에서 발생한 예외만 처리하며, 다운스트림 예외는 무시한다.
    - 심지어 예외를 다시 던져 다운스트림에서 처리하게 하기 위해 catch를 사용할 수도 있다.
- retry를 사용해 예외가 발생했을 때 플로우 수집을 처음부터 다시 시작할 수 있다.
    - 이를 통해 코드가 오류를 복구할 수 있는 기회를 가질 수 있다.
- runTest의 가상 시간을 활용하면 코투린 코드 테스트 속도를 높일 수 있다.
    - 모든 지연 시간이 자동으로 빠르게 진행된다.
- TestCoroutineScheduler는 runTest가 노출시키는 TestScope의 일부로, 현재 가상 시간을 추적하며 runCurrent와 advanceUntilIdle 같은 함수로 테스트 실행을 세밀하게 제어할 수 있다.
- 테스트 디스패처는 단일 스레드로 작동한다.
    - 이에 따라 테스트 단언문을 호출하기 전에 새로 시작한 코루틴들이 실행될 수 있는 시간을 수동으로 보장해줘야만 한다.
- 터빈 라이브러리는 플로우 기반의 코드를 간편하게 테스트하게 해준다.
    - 이 라이브러의 핵심 api는 test 확장 함수로, 플로우에서 원소를 수집하고 awaitItem과 같은 함수를 사용해 테스트 중인 플로우의 원소 배출을 확인할 수 있

다.