코틀린 코루틴

Kotlin Coroutines

: Deep Dive

Kotlin Coroutines: Deep Dive
by Marcin Moskała

코틀린 코루틴: 안드로이드 및 백엔드 개발자를 위한 비동기 프로그래밍

전자책 1쇄 발행 2023년 11월 8일 **지은이** 마르친 모스카와 **옮긴이** 신성열 **펴낸이** 한기성 **펴낸곳** (주)도서출판인사이트 **편집** 백주옥 **등록번호** 제2002-000049호 **등록일자** 2002년 2월 19일 **주소** 서울특별시 마포구 연남로5길 19-5 **전화** 02-322-5143 **팩스** 02-3143-5579 **블로그** https://blog.insightbook.co.kr **이메일** insight@insightbook.co.kr **ISBN** 978-89-6626-427-8

프로그래밍 인사이트

코틀린 코루틴

안드로이드 및 백엔드 개발자를 위한 비동기 프로그래밍

마르친 모스카와 지음 | 신성열 옮김

인사이트

차례

옮긴이의 글

제가 코틀린을 처음 접하게 된 건 회사에 입사한 지 얼마 안 된 주니어 개발자 시절이었습니다. 그 당시 팀 내에 있는 많은 시니어 개발자들이 자바와 스프링을 사용해 서버를 뚝딱뚝딱 만들어 내는 것을 보고 역시 자바와 스프링이 가장 좋은 언어와 프레임워크라는 생각을 갖고 있었습니다. 그러던 중, 한 개발자가 테스트용 클라이언트를 코틀린으로 만들었는데, 당시에는 자바도 충분히 편리한 언어인데 왜 코틀린의 익숙하지 않은 문법을 배우면서 고생할까 하는 의구심이 들었습니다.

하지만 몇 년이 지난 지금, 안드로이드는 코틀린을 정식 언어로 선언하여 사용하고 있으며, 서버를 개발하고 운영하는 조직에서도 코틀린을 서서히 도입하고 있는 상황입니다. 또한 많은 팀에서 코틀린으로 새로운 프로젝트를 개발하고 있으며, 운영하고 있는 서버 또한 기존 언어에서 코틀린으로 포팅하고 있습니다. 저도 코틀린을 처음 접했을 때 가졌던 어리석은 생각과 달리, 현재는 코틀린의 다양한 기능을 활용하고 있으며, 대부분 프로젝트를 코틀린으로 진행하고 있습니다.

코틀린은 정말 편리한 언어입니다. 자바에서 가장 문제가 되었던 널(null) 가능성을 컴파일 타임에 검증하여 런타임에 발생할 수 있는 예외를 사전에 차단할 수 있으며, 자바에서 보일러플레이트 코드였던 세터와 게터는 데이터 클래스를 선언함으로써 대체할 수 있습니다. 이 외에도 기존 언어 대신 코틀린을 사용해야 할 이유는 정말 많습니다.

코틀린은 비동기 프로그래밍을 언어적으로 지원하기 위해 코루틴이라는 개념도 도입했습니다. 이 책에도 나오는 내용이지만, 자바에서 비동기 프로그래밍을 구현하려면 RxJava 라이브러리를 사용해야 하는 등 제약이 많습니다. 코

틀린 코루틴도 사용하려면 의존성이 필요하지만, 코틀린 언어를 만든 젯브레인에서 개발한 라이브러리를 사용하는 것이기 때문에 사실상 언어적으로 지원하는 것이나 마찬가지입니다.

코틀린을 처음 접할 때는 코루틴이 정식 버전이 아니었지만, 정식으로 도입된 지금은 많은 개발자가 코루틴을 사용하고 있을 정도로 인기가 많습니다. 하지만 코루틴을 제대로 배우고 싶어도 국내에 출판되어 있는 책이 없어 어려움이 많았습니다. 때마침 코틀린을 사랑해 마지않는 마르친 모스카와가 쓴 《Kotlin Coroutines》을 번역해 달라는 요청이 들어와, 코루틴을 자세히 공부할 수도 있고, 코틀린의 장점 중 하나인 비동기 프로그래밍을 쉽게 처리할 수 있다는 점을 소개할 수 있겠다 싶어 흔쾌히 수락했습니다. 번역을 하면서 최대한 원문의 의미를 살리려고 노력했으나, 여전히 미흡한 점이 있음을 너그럽게 이해해 주시길 바랍니다.

많은 사람이 코틀린이라는 언어에 반했던 것처럼, 이 책을 읽는 여러분들 또한 코루틴의 기능을 배우고 현업에서 적용하면서 비동기 프로그래밍까지 지원하는 코틀린의 매력에 푹 빠져보기를 바랍니다.

2023년 9월

신성열

지은이의 글

"왜 코틀린 코루틴을 배우고 싶은가요?" 제 워크숍에 참여한 사람들에게 이런 질문을 하면 보통 "새로운 기술이니까요." 또는 "모든 사람이 코루틴에 대해 이야기하던데요." 같은 대답을 합니다. 좀더 기술적으로 질문을 하면 사람들은 "코루틴은 가벼운 스레드니까요.", "RxJava보다 다루기 쉬우니까요." 또는 "명령형으로 코드를 짜더라도 동시성을 제공하니까요."와 같은 말을 합니다. 이 답변이 틀린 것은 아니지만 코루틴은 사람들이 생각하는 것보다 훨씬 많은 걸 제공합니다. 코루틴은 동시성을 다루는 가장 좋은 도구라 할 수 있습니다. 코루틴은 컴퓨터 과학 분야에서 1960년대부터 다뤄온 개념이지만, 현업에서는 제한적인 형태로만 사용되어 왔습니다(async/await처럼). 코루틴의 사용성이 변화하기 시작한 건 Golang이 코루틴을 범용적으로 사용하면서부터였습니다. 코틀린 또한 코루틴을 도입하였으며, 코루틴을 가장 강력하고 사용하기 쉽게 구현했다고 생각합니다.

동시성을 다루는 건 점점 중요해지고 있지만, 기존의 기술은 이에 대한 충분한 대책이 되지 못했습니다. 현재 추세로는 코루틴이 IT 산업이 지향하는 방향이며, 코틀린 코루틴은 이를 실현하고 있는 대표적인 예라고 할 수 있습니다. 실제로 어떻게 사용되는지 예제를 통해 이를 보여 주고자 합니다. 이 책을 재밌게 읽을 수 있기를 바랍니다.

이 책의 대상 독자

백엔드와 안드로이드 개발 모두를 접해본 사람으로서 저는 이 책을 두 가지 관점에 중점을 두고 쓰려고 노력했습니다. 현재 코틀린이 가장 많이 사용되는 분야는 백엔드와 안드로이드이며, 코루틴은 해당 분야에서 활용하기에 적합하도록 설계되었습니다.[1] 따라서 이 책이 안드로이드와 백엔드 개발자들을 위한 책

1 구글의 안드로이드 팀이 우리가 이 책에서 설명하는 특징을 설계하고 구현했습니다.

이라고 말할 수 있지만, 코틀린을 사용하는 다른 분야의 개발자들에게도 유용할 거라 생각합니다.

이 책은 독자들이 코틀린에 대해 잘 알고 있다고 생각하고 쓰였습니다. 만약 코틀린에 익숙하지 않다면, 드미트리 제메로프(Dmitry Jemerov)와 스베트라나 이사코바(Svetlana Isakova)가 쓴 《Kotlin in Action》(에이콘, 2017)을 먼저 읽는 걸 추천합니다.

이 책의 구성

이 책은 다음과 같이 네 개의 부로 구성되어 있습니다.

- 1부: 코틀린 코루틴 이해하기 — 코틀린 코루틴이란 무엇인지 그리고 어떻게 작동하는지에 초점이 맞춰져 있습니다.
- 2부: 코틀린 코루틴 라이브러리 — kotlinx.coroutines 라이브러리에서 가장 중요한 개념과 이를 잘 사용하는 방법을 설명합니다.
- 3부: 채널과 플로우 — kotlinx.coroutines 라이브러리의 채널과 플로우를 다룹니다.
- 4부: 코틀린 코루틴 적용하기 — 코틀린 코루틴의 일반적인 사용 예제와 코틀린 코루틴을 사용한 가장 중요한 모범 사례를 다룹니다.

이 책에서 다루는 내용

이 책은 제가 진행한 워크숍을 기초로 쓰였습니다. 워크숍을 진행하면서 참가자들이 어떤 부분에 관심이 있고 없는지를 파악할 수 있었습니다. 참가자들이 가장 많이 했던 질문은 다음과 같습니다.

- 코루틴은 실제로 어떻게 작동하나요? (1부)
- 코루틴을 현업에서 어떻게 사용할 수 있나요? (2부, 3부, 특히 4부)
- 코루틴을 사용하기에 가장 좋은 분야는 무엇인가요? (2부, 3부, 특히 4부)
- 코틀린 코루틴은 어떻게 테스트할 수 있나요? (2부의 '코틀린 코루틴 테스트하기'와 3부의 '플로우 테스트하기')
- 플로우란 무엇이며, 어떻게 작동하나요? (3부)

'개발자를 위한 코틀린' 시리즈

이 책은 '개발자를 위한 코틀린' 시리즈 중 하나입니다. '개발자를 위한 코틀린' 시리즈는 다음 책들로 구성되어 있습니다.

- 《Kotlin Essentials》: 코틀린의 기본 특징을 다룹니다.
- 《Functional Kotlin》: 함수 타입, 람다 표현식, 컬렉션 처리, 도메인 전용 언어(DSL), 스코프 함수를 포함한 함수형 코틀린의 특징을 다룹니다.
- 《Kotlin Coroutines(코틀린 코루틴)》: 코루틴을 사용하고 테스트하는 방법, 플로우 사용법, 코루틴의 모범 사례, 코루틴을 사용할 때 저지르는 가장 흔한 실수들과 같은 코루틴의 모든 특징을 다룹니다.
- 《Advanced Kotlin》: 제네릭 가변성 수식어, 위임(delegation), 멀티플랫폼 프로그래밍, 어노테이션 처리, 코틀린 심벌 처리(KSP), 컴파일러 플러그인과 같은 코틀린의 고급 기능을 다룹니다.
- 《Effective Kotlin(이펙티브 코틀린)》: 코틀린 프로그래밍의 모범 사례를 다룹니다.

이 책들은 훌륭한 코틀린 개발자로 성장하는 데 필요한 모든 것을 다루고 있습니다.

이 책의 표기법

구체적인 코드를 예로 들 때는 고정폭 서체를 사용합니다. 개념을 표기할 때는 앞 글자를 대문자로 표기합니다. 특정 타입에 속한 요소를 표기할 때는 소문자를 사용합니다. 예를 들면,

- `Flow`는 타입 또는 인터페이스이므로 고정폭 서체로 표기합니다(예: `Flow`를 반환하는 함수가 중단 함수가 되어서는 안 됩니다).
- Flow는 개념을 나타내므로 대문자로 시작합니다(예: 채널(Channel)과 플로우(Flow)의 핵심적인 차이를 말해 줍니다).
- flow는 리스트나 집합과 같은 인스턴스이므로 소문자로 표기합니다(예: 모든 플로우(flow)는 몇 가지 요소로 구성됩니다).

또 다른 예를 들면, List는 인터페이스 또는 타입을 의미합니다(i의 타입은 List입니다.). List는 개념(자료 구조)을 나타내지만, list는 수많은 리스트 중 하나를 뜻합니다(list 변수는 리스트를 담고 있습니다).

코루틴 상태 중 'Cancelled'가 있어 '취소(cancellation)/취소된(cancelled)'을 제외하면 미국식 영어를 사용했습니다.

이 책의 코드 표기법

이 책에 소개된 대부분의 코드는 import 구문 없이 실행 가능합니다. 이 책의 일부분은 코틀린 아카데미(Kt. Academy) 홈페이지에서 실행 가능한 코드가 포함된 기사로 소개되었으며, 독자들은 이 코드를 실제로 실행할 수 있습니다. 모든 코드는 다음 깃허브(GitHub) 저장소에서 찾아볼 수 있습니다.

https://github.com/MarcinMoskala/coroutines_sources

코드 결과는 println 함수를 사용해 확인합니다. 결과는 코드의 끝에 주석으로 표시됩니다. 결과로 표시되는 줄 사이에 지연이 있다면, 주석 내 괄호로 표시합니다. 다음은 그 예입니다.

```
suspend fun main(): Unit = coroutineScope  {
    launch {
        delay(1000L)
        println("World!")
    }
    println("Hello,")
}
// Hello,
// (1초 후)
// World!
```

가끔 코드 일부나 결과가 ...로 생략되어 표시되는 경우도 있습니다. 이때는 '코드가 더 있지만 예제와는 무관하다'고 생각하면 됩니다.

```
launch(CoroutineName("Name1")) { ... }
launch(CoroutineName("Name2") + Job()) {  ... }
```

몇몇 예제에서는 주석을 출력 또는 대기하는 코드 바로 옆에 배치했습니다. 순서가 확실히 정해졌을 때입니다.

```
suspend fun main(): Unit = coroutineScope  {
    println("Hello,") // Hello,
    delay(1000L)      // (1초 후)
    println("World!") // World!
}
```

코드 일부에서는 각 줄의 끝에 숫자를 넣어 코드가 실행되는 걸 쉽게 설명하려고 했습니다. 다음과 같은 예를 들 수 있습니다.

```
suspend fun  main(): Unit = coroutineScope  {
    println("Hello,") // 1
    delay(1000L)      // 2
    println("World!") // 3
}
```

1에서는 "Hello"를 출력하고 delay가 포함된 2에서 1초를 기다린 뒤, 3에서 "World!"를 출력합니다.

감사의 말

이 책은 리뷰어들의 제안과 의견이 없었다면 그리 좋은 책이 될 수 없었을 것입니다. 리뷰어들 모두에게 감사의 말을 전합니다. 리뷰어들을 참여도순으로 소개하면 다음과 같습니다.

니콜라 코티(Nicola Coti)

코틀린 분야의 GDE(Google Developer Expert)입니다. 버전 1.0 이전부터 코틀린을 사용해 왔으며, 모바일 개발자들이 주로 사용하는 Detekt, Chucker, AppIntro와 같은 오픈소스 라이브러리와 도구를 관리하고 있습니다. 지금은 메타(Meta)의 리액트 네이티브(React Native) 코어 팀에서 일하고 있으며, 모바일 분야에서 가장 인기 있는 크로스 플랫폼 프레임워크를 개발하고 있습니다.

개발자 모임에도 활발하게 참여하고 있습니다. 국제 컨퍼런스에서 연사로 강연을 하거나 CFP의 위원으로 활동하기도 하며 유럽의 개발자 모임을 지원하는 등 다양한 활동을 하고 있습니다. 여가 시간에는 빵을 굽거나, 팟캐스트를 하거나, 달리기를 합니다.

가리마 자인(Garima Jain)

인도 출신의 안드로이드 GDE(Google Developer Expert)입니다. 커뮤니티에서는 @ragdroid로 유명합니다. 가리마는 고대디(GoDaddy)의 안드로이드 핵심 기술자입니다. 국제적인 연사이며 기술 블로거로서 활발하게 활동하고 있습니다. 커뮤니티에서 다른 사람들과 소통하고 생각을 공유하는 것을 좋아합니다. 쉴 때는 TV를 보거나, 탁구나 농구하는 걸 좋아합니다. 코딩뿐 아니라 소설도 정말 좋아해 프로그래밍 기술을 접목시킨 소설에 대해 다른 사람들과 이야기하거나 블로그를 통해 공유하고 있습니다.

일미르 우스마노프(Ilmir Usmanov)

젯브레인(JetBrains)의 소프트웨어 개발자로, 2017년부터 코틀린 컴파일러에서 코루틴 지원을 담당하고 있습니다. 코루틴 설계에서 안정화와 구현을 책임졌습니다. 이후 인라인 클래스를 비롯한 다른 기능을 개발했습니다. 현재 코루틴과 관련된 일은 버그 수정과 최적화만 수행하고 있는데, 코루틴은 이미 구현이 완료되어 크게 수정할 일이 많지 않기 때문입니다.

션 맥퀼란(Sean Mcquillan)

구글의 개발자 애드보케이트(Developer Advocate)[2]입니다. 10년 동안 트윌리오(Twilio)와 샌프란시스코의 스타트업에서 일했으며, 애플리케이션을 적정 규모로 빌드할 수

2 (옮긴이) 개발자들을 대상으로 회사가 보유하고 있는 기술과 문화를 전파해 개발자들을 모으고 회사가 보유한 기술 플랫폼을 확산시키는 사람

있도록 좋은 도구를 적용하는 방법에 대해 관심이 많습니다. 션은 높은 품질의 애플리케이션을 빠르게 빌드하는 좋은 방법을 연구하고 있습니다. 안드로이드 관련 일을 하지 않을 때는 피아노를 연주하거나 모자를 짭니다.

이고르 워즈다(Igor Wojda)

10년 이상의 소프트웨어 개발 경력이 있는 열정적인 개발자입니다. 가장 관심 있어 하는 분야는 안드로이드 애플리케이션의 구조와 코틀린 언어이며, 오픈소스 커뮤니티에도 활발하게 참여하고 있습니다. 이고르는 컨퍼런스의 연사이자 《Kotlin in Action》을 기술적으로 검토하였고, 《Android Development with Kotlin》을 집필하기도 했습니다. 이고르는 다른 개발자들과 열정적인 코딩에 대해 이야기하는 걸 좋아합니다.

자나 자롤리모바(Jana Jarolimova)

아바스트(Avast)의 안드로이드 개발자입니다. 프라하 대학교(Prague City University)에서 자바에 대해 가르친 뒤, 모바일 쪽 개발을 시작했으며 그 후 코틀린이란 언어에 대해 애정을 갖게 되었습니다.

리차드 슈엘렉(Richard Schielek)

실력 있는 개발자로 코틀린과 코루틴이 안정화되기 전부터 현업에서 사용해 왔습니다. 유럽의 우주 산업 분야에서도 몇 년 동안 일했습니다.

브제보로드 톨스토퍄토프(Vsevolod Tolstopyatov)

코틀린 라이브러리 팀을 이끌고 있습니다. 젯브레인에서 일했으며, API 설계, 동시성, JVM 내부 구조, 성능 튜닝과 방법론에 대해 관심이 많습니다.

루카스 레흐너(Lukas Lechner), 이브라힘 일마즈(Ibrahim Yilmaz), 딘 제르마노비치(Dean Djermanovic), 단 오닐(Dan O'Neill)

이 책의 어휘와 문법을 검토하고 교정해 준 마이클 팀버레이크(Michael Timberlake)에게도 감사하다는 말을 전하고 싶습니다.

1부

코틀린 코루틴 이해하기

코틀린 코루틴 라이브러리에 대해 살펴보기 전에 몇 가지 기본적인 개념부터 살펴보겠습니다. 코루틴이란 무엇인가? 중단(suspension)은 어떻게 작동하는가? 코틀린은 코루틴을 어떤 식으로 구현하는가? 몇 가지 유용한 도구와 사용 예제를 보면서 이 질문에 대한 답을 찾아봅시다.

1장

Kotlin Coroutines

코틀린 코루틴을 배워야 하는 이유

왜 우리는 코틀린 코루틴을 배워야 할까요? 이미 널리 사용되고 있는 RxJava나 Reactor와 같은 JVM 계열 라이브러리가 있는데도 말이죠. 게다가 자바 또한 언어 자체적으로 멀티스레드를 지원하고 있고, 스레드 대신 낡은 방식인 콜백 함수를 활용하고 있는 개발자도 많습니다. 우리는 이미 비동기적 연산을 수행하기 위한 다양한 방법을 알고 있습니다.

코틀린 코루틴은 기존의 방식보다 훨씬 많은 것을 제공합니다. 코루틴은 1963년에 처음 제시되었지만,[1] 코루틴이란 개념이 실제 현업에서 사용될 수 있도록 구현되기까지 수십 년이 걸렸습니다.[2] 코틀린 코루틴은 반세기 전의 논문에서 소개된 강력한 기능을 실생활에서도 유용하게 사용할 수 있게 설계된 라이브러리로 만들어졌습니다. 게다가 코틀린 코루틴은 멀티플랫폼에서 작동시킬 수 있기 때문에 코틀린을 사용하는 모든 플랫폼(JVM, JS, iOS 또는 다른 모든 모듈들)을 넘나들며 사용할 수 있습니다. 무엇보다 코틀린 코루틴을 도입한다고 해서 기존 코드 구조를 광범위하게 뜯어고칠 필요도 없습니다. 약간의

1 Conway, Melvin E. (July 1963). "Design of a Separable Transition-Diagram Compiler". Communications of the ACM. ACM. 6 (7): 396-408. doi:10.1145/366663.366704. ISSN 0001-0782. S2CID 10559786

2 Go는 2009년에 코루틴을 가장 먼저 상용화하여 도입하였습니다. 코루틴은 Lisp와 같이 오래된 언어에서도 구현되었으나 별다른 인기를 끌지 못했습니다. 그 이유는 코루틴이 구현된 방식이 실제로 사용되기에 적합하게 설계되지 않았기 때문입니다. Lisp(Haskell과 같이)는 현업 개발자들을 위한 언어가 아닌 컴퓨터 과학자들의 놀이터로 여겨진 탓이 큽니다.

수고로도 코틀린 코루틴이 제공하는 대부분의 기능을 사용할 수 있으며, 이는 RxJava나 콜백에서는 기대할 수 없던 것이었습니다. 이는 코틀린 코루틴이 초보 개발자들이 사용하는 데도 무리가 없음을 의미합니다.[3]

코루틴을 적용한 예를 확인해 봅시다. 코루틴을 사용한 예제와 잘 알려진 기존의 접근 방식을 사용한 예제를 비교하면 두 방법이 어떻게 다른지 알 수 있을 것입니다. 두 가지 범용적인 예제인 안드로이드와 백엔드 비즈니스 로직을 구현한 것을 비교해 봅시다. 우선 안드로이드 예제부터 보겠습니다.

안드로이드(그리고 다른 프론트엔드 플랫폼)에서의 코루틴 사용

프론트엔드 단에서 애플리케이션 로직을 구현할 때 가장 흔하게 사용하는 방법은 다음과 같습니다.

1. 하나 또는 다양한 소스(API, 뷰 구성요소, 데이터베이스, 설정, 다른 애플리케이션)로부터 데이터를 얻어온다.
2. 데이터를 처리한다.
3. 가공된 데이터로 무엇인가를 한다(뷰를 통해 보여 주기, 데이터베이스에 저장하기, API로 전송하기 등).

코루틴을 비롯한 비동기 연산을 지원하는 라이브러리가 어떻게 사용되는지 확인하기 위해 안드로이드 앱을 개발하고 있다고 해 봅시다. API로부터 뉴스를 가지고 와서 정렬한 다음, 스크린에 띄우는 로직을 구현하는 경우를 생각해 볼 수 있습니다. 다음은 해당 로직을 간단하게 구현한 예제입니다.

```
fun onCreate() {
    val news = getNewsFromApi()
    val sortedNews = news
        .sortedByDescending { it.publishedAt }
    view.showNews(sortedNews)
}
```

3 물론 코루틴을 잘 사용하려면 코루틴 개념을 잘 이해하고 있어야 한다는 사실에는 변함이 없습니다.

안타깝게도 안드로이드에서는 앞의 예제처럼 간단하게 구현할 수 없습니다. 안드로이드에서는 하나의 앱에서 뷰를 다루는 스레드가 단 하나만 존재합니다. 이 스레드는 앱에서 가장 중요한 스레드라 블로킹되면 안 되기 때문에, 이런 방법으로 구현할 수 없습니다. `onCreate` 함수가 메인 스레드에서 실행된다면 `getNewsFromApi` 함수가 스레드를 블로킹할 것이고, 애플리케이션에 앱 크래시(비정상 종료)가 발생할 것입니다. `getNewsFromApi` 함수를 다른 스레드에서 실행하더라도 `showNews`를 호출할 때 정보가 없으므로 메인 스레드에서도 마찬가지로 크래시가 발생합니다.

스레드 전환

스레드 전환이 위에서 말한 문제를 푸는 가장 직관적인 방법입니다. 블로킹이 가능한 스레드를 먼저 사용하고, 이후에 메인 스레드로 전환하면 됩니다.

```
fun onCreate() {
    thread {
        val news = getNewsFromApi()
        val sortedNews = news
            .sortedByDescending { it.publishedAt }
        runOnUiThread {
            view.showNews(sortedNews)
        }
    }
}
```

몇몇 애플리케이션에서는 위와 같은 스레드 전환 방식을 아직도 사용하고 있지만, 다음과 같은 문제가 있습니다.

- 스레드가 실행되었을 때 멈출 수 있는 방법이 없어 메모리 누수로 이어질 수 있습니다.
- 스레드를 많이 생성하면 비용이 많이 듭니다.
- 스레드를 자주 전환하면 복잡도를 증가시키며 관리하기도 어렵습니다.
- 코드가 쓸데없이 길어지고 이해하기 어려워집니다.

다음과 같이 생각하면 위의 문제를 이해하는 데 도움이 됩니다. 뷰를 열었다가

재빨리 닫았다고 생각해 봅시다. 뷰가 열려 있는 동안, 데이터를 가져와 처리하는 스레드가 다수 생성됩니다. 생성된 스레드들을 제거하지 않으면, 스레드는 주어진 작업을 계속 수행한 후 더 이상 존재하지 않는 뷰를 수정하려고 시도할 것입니다. 이것은 디바이스에 불필요한 작업이며, 백그라운드에서 예외(exceptions)를 유발하거나 예상하기 어려운 결과가 발생할 수 있습니다.

위에서 말한 문제점들을 떠올리며, 더 나은 방법을 생각해 봅시다.

콜백

콜백(callback)은 앞에서 주어진 문제를 해결하는 또 다른 패턴입니다. 콜백의 기본적인 방법은 함수를 논블로킹(non-blocking)으로 만들고, 함수의 작업이 끝났을 때 호출될 콜백 함수를 넘겨주는 것입니다. 콜백 패턴을 쓰는 함수는 다음과 같습니다.

```
fun onCreate() {
    getNewsFromApi { news ->
        val sortedNews = news
            .sortedByDescending { it.publishedAt }
        view.showNews(sortedNews)
    }
}
```

위와 같이 콜백을 이용해 구현한 방식 또한 중간에 작업을 취소할 수 없습니다. 취소할 수 있는 콜백 함수를 만들 수도 있지만 쉬운 일은 아닙니다. 콜백 함수 각각에 대해 취소할 수 있도록 구현해야 할 뿐 아니라, 취소하기 위해선 모든 객체를 분리해서 모아야 합니다.

```
fun onCreate() {
    startedCallbacks += getNewsFromApi { news ->
        val sortedNews = news
            .sortedByDescending { it.publishedAt }
        view.showNews(sortedNews)
    }
}
```

콜백 구조는 이 문제를 간단하게 풀 수 있지만 단점이 많습니다. 세 군데서 데이터를 얻어오는 다음과 같은 예제를 살펴봅시다.

```
fun showNews() {
    getConfigFromApi { config ->
        getNewsFromApi(config) { news ->
            getUserFromApi { user ->
                view.showNews(user, news)
            }
        }
    }
}
```

위 코드는 다음과 같은 이유 때문에 완벽한 해결책이 될 수 없습니다.

- 뉴스를 얻어오는 작업과 사용자 데이터를 얻어오는 작업은 병렬로 처리할 수 있지만, 현재의 콜백 구조로는 두 작업을 동시에 처리할 수 없습니다(이를 콜백으로 해결하기란 매우 어렵습니다).
- 아까 지적했던 것처럼 취소할 수 있도록 구현하려면 많은 노력이 필요합니다.
- 들여쓰기가 많아질수록 코드는 읽기 어려워집니다(콜백을 많이 사용한 코드가 읽기 어렵다고 생각되는 이유입니다). 이런 상황을 '콜백 지옥(callback hell)'이라 부르며, 오래된 Node.JS 프로젝트에서 쉽게 찾을 수 있습니다.

```
    describe('.totalValue', function(){
      it('should calculate the total value of items in a space', function(done){
      var table = new Item('table', 'dining room','07/23/2014', '1','3000');
      var chair = new Item('chair', 'living room','07/23/2014', '3','300');
      var couch = new Item('couch', 'living room','07/23/2014', '2','1100');
      var chair2 = new Item('chair', 'dining room','07/23/2014', '4','500');
      var bed = new Item('bed', 'bed room','07/23/2014', '1','2000');

      table.save(function(){
        chair.save(function(){
          couch.save(function(){
            chair2.save(function(){
              bed.save(function(){
                Item.totalValue({room: 'dining room'}, function(totalValue){
                  expect(totalValue).to.equal(5000);
                  done();
                  });
                });
              });
            });
          });
        });
      });
    });
});
```

- 콜백을 사용하면 작업의 순서를 다루기 힘들어집니다. 다음과 같이 프로그레스 바를 보여 주는 방법은 작동하지 않습니다.

```
fun onCreate() {
   showProgressBar()
   showNews()
   hideProgressBar() // 잘못된 방식입니다.
}
```

프로그레스 바는 뉴스를 보여 주는 작업을 시작하고 곧바로 사라집니다. 프로그레스 바가 제대로 작동하게 하려면 showNews에도 콜백 함수를 만들어야 합니다.

```
fun onCreate() {
    showProgressBar()
    showNews {
        hideProgressBar()
    }
}
```

지금까지 콜백 구조가 적절한 해결책이 아니라는 걸 살펴보았습니다. 이제 또 다른 접근법인 RxJava와 리액티브 스트림(reactive stream)에 대해 알아보겠습니다.

RxJava와 리액티브 스트림

자바 진영(안드로이드와 백엔드 모두)에서 인기 있는 또 다른 해결책은 RxJava나 그 뒤를 이은 Reactor와 같은 리액티브 스트림(또는 리액티브 확장 라이브러리)을 사용하는 것입니다. 이 방법을 사용하면 데이터 스트림 내에서 일어나는 모든 연산을 시작, 처리, 관찰할 수 있습니다. 리액티브 스트림은 스레드 전환과 동시성 처리를 지원하기 때문에 애플리케이션 내의 연산을 병렬 처리하는 데 사용됩니다.

다음은 RxJava를 사용해 문제를 해결한 예제입니다.

```
fun onCreate() {
    disposables += getNewsFromApi()
        .subscribeOn(Schedulers.io())
        .observeOn(AndroidSchedulers.mainThread())
        .map { news ->
            news.sortedByDescending { it.publishedAt }
        }
        .subscribe { sortedNews ->
            view.showNews(sortedNews)
        }
}
```

위 예제에서 disposables는 사용자가 스크린을 빠져나갈 경우 스트림을 취소하기 위해 필요합니다.

RxJava를 사용한 방법이 콜백을 사용한 것보다 훨씬 더 좋은 방법입니다. 메모리 누수도 없고, 취소가 가능하며, 스레드를 적절하게 사용하고 있습니다. 하지만 RxJava는 구현하기에 아주 복잡하다는 단점이 있습니다. 이 방식을 처음 시작할 때 보여 준 '이상적인' 코드와 비교하면(아래에도 나와 있는), 완전히 다른 형태의 코드라고 볼 수 있습니다.

```
fun onCreate() {
    val news = getNewsFromApi()
    val sortedNews = news
        .sortedByDescending { it.publishedAt }
    view.showNews(sortedNews)
}
```

subscribeOn, observeOn, map, subscribe와 같은 함수들을 RxJava를 사용하기 위해 배워야 합니다. 취소하는 작업 또한 명시적으로 표시해야 합니다. 객체를 반환하는 함수들은 Observable이나 Single 클래스로 래핑(wrapping)해야 합니다. 실제로 RxJava를 도입하려면 수많은 코드를 바꿔야 합니다.

```
fun getNewsFromApi(): Single<List<News>>
```

데이터를 보여 주기 전에 세 개의 엔드포인트(endpoint, 서비스에서 다른 서비

스에 요청을 보내는 지점)를 호출해야 한다는 것도 또 다른 문제입니다. 이 또한 RxJava를 사용해 해결할 수 있지만 훨씬 복잡합니다.

```
fun showNews() {
    disposables += Observable.zip(
        getConfigFromApi().flatMap { getNewsFromApi(it) },
        getUserFromApi(),
        Function2 { news: List<News>, config: Config ->
            Pair(news, config)
        })
        .subscribeOn(Schedulers.io())
        .observeOn(AndroidSchedulers.mainThread())
        .subscribe { (news, config) ->
            view.showNews(news, config)
        }
}
```

위 코드는 동시성 처리도 되어 있으며 메모리 누수도 없지만, `zip`, `flatMap`과 같은 RxJava 함수를 사용해야 하고 값을 `Pair`로 묶고 분리도 해야 합니다. 적절한 구현 방법이긴 하지만 너무 복잡하다는 단점이 있습니다. 이제 코루틴이 어떻게 문제를 해결하는지 살펴봅시다.

코틀린 코루틴의 사용

코틀린 코루틴이 도입한 핵심 기능은 코루틴을 특정 지점에서 멈추고 이후에 재개할 수 있다는 것입니다. 코루틴을 사용하면 우리가 짠 코드를 메인 스레드에서 실행하고 API에서 데이터를 얻어올 때 잠깐 중단시킬 수도 있습니다. 코루틴을 중단시켰을 때 스레드는 블로킹되지 않으며 뷰를 바꾸거나 다른 코루틴을 실행하는 등의 또 다른 작업이 가능합니다. 데이터가 준비되면 코루틴은 메인 스레드에서 대기하고 있다가(코루틴이 대기하는 건 스레드를 기다리고 있는 코루틴이 쌓여있는 경우와 같이 극히 드문 상황에서 일어납니다) 메인 스레드가 준비되면 멈춘 지점에서 다시 작업을 수행합니다.