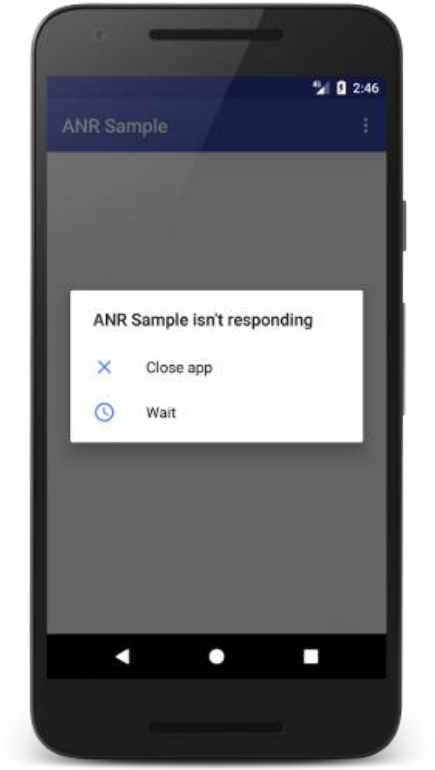
Kotlin coroutine과 Flow를 사용한 비동기 처리

2016010873 박정욱

목차

- 왜 비동기 처리를 해야 하는가?
- Kotlin의 비동기 처리 기술, coroutine
- 구조적 동시성(Structured Concurrency)이란?
- 비동기적 데이터 스트림 Flow
- CoroutineDispatcher를 사용하여 작업 형식에 따라 coroutine 분배하기
- 실제 적용 사례

왜 비동기 처리를 해야 하는가?



- Android 애플리케이션은 싱글 스레드로 시작
- 기본 스레드는 UI 이벤트 처리 책임이 있음
- 이외 실행 시간이 오래 걸리는 작업들의 경우, 기본 스레드에서 실행 시 **응답 없음** 오류 발생
- 이러한 작업은 별도 스레드에서, 비동기적으로 처리해야 함

Kotlin의 비동기 처리 기술, coroutine

AsyncTask

```
public abstract class AsyncTask
extends Object
```

java.lang.Object
↳ android.os.AsyncTask<Params, Progress, Result>

This class was deprecated in API level 30.
Use the standard `java.util.concurrent` or Kotlin concurrency utilities instead.

- 기존의 AsyncTask는 더 이상 권장되지 않는 방식!
- Java의 경우, RxJava 사용
- Kotlin의 경우, coroutine!

Kotlin의 비동기 처리 기술, coroutine

RxJava

Kotlin
Coroutines

- 기존의 AsyncTask는 더 이상 권장되지 않는 방식!
- Java의 경우, RxJava 사용
- Kotlin의 경우, coroutine!

Kotlin의 비동기 처리 기술, coroutine

- 비선점형 멀티태스킹을 위해 일반화된 서브루틴
- **실행을 중단**할 수 있고, 중단된 다음 장소부터 **다시 실행 가능**함
- 일반적으로 선점형으로 구현되는 Thread와는 비슷하지만 서로 다름
- 동시성을 제공하지만, 병렬성은 제공하지 않음

구조적 동시성(Structured Concurrency)이란?

```
suspend fun concurrentSum(): Int = coroutineScope {
    val one = async { doSomethingUsefulOne() }
    val two = async { doSomethingUsefulTwo() }
    one.await() + two.await()
}
```

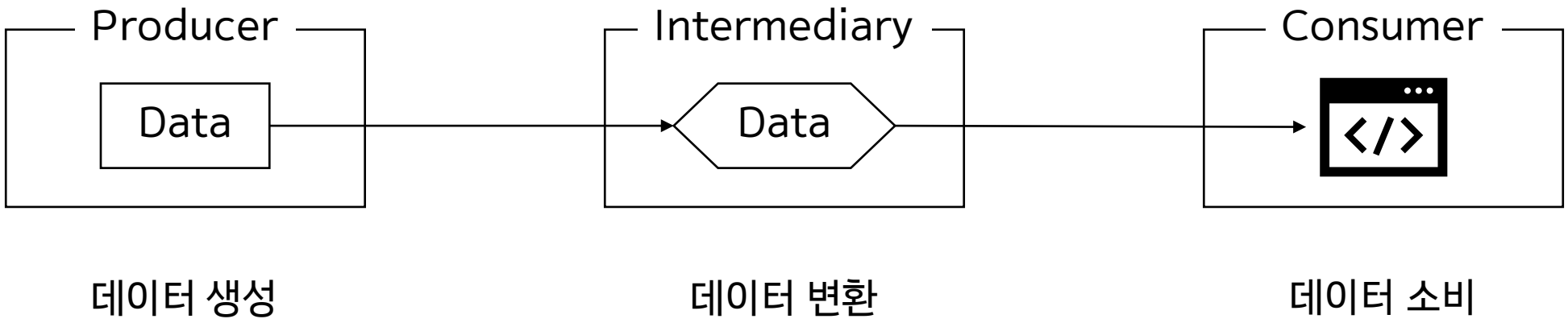
- coroutineScope 내에서 일어나는 일은 **All-or-nothing**
  - DB의 Transaction과도 유사
- 하위에서 예외가 발생하여 취소 시, 상위 범위로 취소 전파
- 상위에서 실행 취소 시, 하위 범위로 취소 전파

CoroutineDispatcher를 사용하여 작업 형식에 따라 coroutine 분배하기

```kotlin
@Module
@InstallIn(SingletonComponent::class)
object DispatcherModule {

    @DefaultDispatcher
    @Provides
    fun provideDefaultDispatcher(): CoroutineDispatcher = Dispatchers.Default

    @IoDispatcher
    @Provides
    fun provideIoDispatcher(): CoroutineDispatcher = Dispatchers.IO

    @MainDispatcher
    @Provides
    fun provideMainDispatcher(): CoroutineDispatcher = Dispatchers.Main
}

@Retention(AnnotationRetention.BINARY)
@Qualifier
annotation class DefaultDispatcher

@Retention(AnnotationRetention.BINARY)
@Qualifier
annotation class IoDispatcher

@Retention(AnnotationRetention.BINARY)
@Qualifier
annotation class MainDispatcher
```

- 실행 시간이 오래 걸리는 작업은 별도 스레드에서!
- 각 작업에 특화된 스레드가 있음
- 특정 작업이 실행되는 스레드를 정해줄 수 있음

비동기적 데이터 스트림 Flow



- Coroutine을 이용한 리액티브 프로그래밍을 지원하기 위한 요소
- 비동기적으로 **여러** 값을 계산하거나 얻기 위해 사용

실제 적용 사례

```
override suspend fun insert(routine: RoutineWithTodo) {
    dao.insert(routine)
}

override suspend fun update(routine: RoutineWithTodo) {
    dao.update(routine)
}

override suspend fun delete(routine: RoutineWithTodo) {
    dao.delete(routine)
}
```

```
override suspend fun insert(routine: Routine) = withContext(ioDispatcher) {  this: CoroutineScope
    localDataSource.insert(RoutineMapper.fromRoutineToRoutineWithTodo(routine))
}

override suspend fun update(routine: Routine) {
    localDataSource.update(RoutineMapper.fromRoutineToRoutineWithTodo(routine))
}

override suspend fun deleteInLocal(routine: Routine) = withContext(ioDispatcher) {  this: CoroutineScope
    localDataSource.delete(RoutineMapper.fromRoutineToRoutineWithTodo(routine))
}
```

실제 적용 사례

```
@Transaction
@Query("SELECT * FROM routine_table")
fun getRoutinesWithTodos(): Flow<List<RoutineWithTodo>>
```

```
override fun getRoutineList(): Flow<List<RoutineWithTodo>> {
    return dao.getRoutinesWithTodos()
}
```

```
override fun getAllRoutinesFromLocal(): Flow<List<Routine>> {
    return localDataSource.getRoutineList().map { list ->
        list.map { it: RoutineWithTodo
            RoutineMapper.fromRoutineWithTodoToRoutine(it)
        }
    }
}
```

정리

- Android는 UI 이벤트를 관리하는 기본 스레드에서 실행 시간이 긴 작업을 하면 안 된다!
- Kotlin에서는 coroutine을 이용해 비동기 처리가 가능하다!
- 데이터의 변화에 따라 값을 여러 개 얻고 싶을 때는 Flow!
- 특정 작업이 실행될 스레드를 각각 지정해주는 방법이 있다!