3주차

코루틴 개념

코루틴(Coroutine)은 비동기 프로그래밍을 효율적으로 수행하기 위한 프로그램 구성 요소입니다. 일반적인 함수 호출과 달리, 코루틴은 실행 중인 상태를 유지한 채로 중단할 수 있으며, 필요 시 중단된 지점에서 다시 실행을 이어나갈 수 있습니다.

`코루틴은 이전에 자신이 실행이 마지막으로 중단되었던 지점 다음의 장소에서 실행을 재개한다`

멜빈 콘웨이가 1958년에 코루틴이라는 용어를 만들었으며, 어셈블리 프로그램에 적용했다.
코루틴은 협력잡업 예외 이벤트루프 반복자 무한목록 및 파이프와 같은 프로그램 구성요소를 구현하는데 적합하다

- **장점**: 메모리 효율적이고, 비동기 작업을 쉽게 관리할 수 있습니다.
- **주요 특징**:
  - **비동기성**: 시간이 오래 걸리는 작업(I/O, 네트워크 호출 등)을 블로킹하지 않고 처리 가능.
  - **경량성**: 코루틴은 스레드보다 가볍고, 메모리 오버헤드가 적습니다.
  - **컨커런시(Concurrency)**: 여러 작업을 병렬로 실행하는 것처럼 보이지만, 실제로는 단일 스레드에서 이루어질 수도 있습니다.

코루틴을 왜 사용하는가?

1. **효율적인 비동기 작업 처리**: I/O 작업이 많은 애플리케이션에서 자원을 효율적으로 사용하도록 돕습니다.
2. **스레드 블로킹 제거**: 스레드를 블로킹하지 않기 때문에 더 많은 작업을 단일 스레드에서 실행할 수 있습니다.
3. **가독성 높은 코드 작성**: 콜백 지옥(Callback Hell)을 제거하여 동기 코드처럼 읽히는 비동기 코드를 작성할 수 있습니다.
4. **비용 절감**: 스레드 기반 동시성보다 더 적은 메모리와 CPU 리소스를 소모합니다.

코루틴 동작 원리

1. **이벤트 루프 기반**:
   코루틴은 이벤트 루프에서 실행됩니다. 이벤트 루프는 작업을 큐에 추가하고, 각 작업의 상태를 관리합

니다.

2. **중단점(yield)**:
   코루틴은 특정 작업(I/O 작업 등)을 수행하는 동안 중단되고, 다른 코루틴이나 작업이 실행됩니다.
3. **비교 - 스레드와의 차이점**:

코루틴	스레드
스레드 위에서 동작 (JVM의 가상 스레드)	실제 OS 스레드에서 실행됩니다.
가볍고, 메모리와 컨텍스트 전환 비용 적음	상대적으로 무겁고 컨텍스트 전환 비용이 큼
이벤트 루프와 협력적 멀티태스킹 사용	OS의 선점형 스케줄링에 의존
블로킹 없이 실행	특정 작업에서 스레드가 블로킹될 수 있음

Global Scope

코루틴은 비동기 프로그래밍을 쉽게 구현하기 위한 경량 스레드이다

Coroutine Scope - 코루틴의 생명주기를 관리하는 역할

Global Scope / runBlocking

`GlobalScope` 는 코루틴을 애플리케이션 전체의 수명 동안 실행하도록 하는 범위입니다. GlobalScope에서 시작된 코루틴은 명시적으로 취소되지 않는 한 계속 실행됩니다. 이는 주로 백그라운드 작업이나 애플리케이션의 수명 동안 지속되어야 하는 작업에 사용됩니다.

```
import kotlinx.coroutines.*

fun main() {
    GlobalScope.launch {
        // 백그라운드에서 실행될 코루틴
        delay(1000L)
        println("GlobalScope: 코루틴 실행 완료")
    }

    println("메인 함수 실행 중")
    // 메인 함수가 종료되지 않도록 잠시 대기
    Thread.sleep(2000L)
}
```

수명 주기: GlobalScope에서 시작된 코루틴은 애플리케이션의 전체 수명 동안 실행됩니다.
취소: 명시적으로 취소하지 않으면 계속 실행됩니다.
컨텍스트: 기본적으로 Dispatchers.Default를 사용하여 백그라운드 스레드에서 실행됩니다.

runBlocking

runBlocking은 현재 스레드를 블록하면서 코루틴을 실행합니다. 주로 테스트나 메인 함수에서 비동기 코드를 동기적으로 호출할 때 사용됩니다. runBlocking 내에서 실행된 모든 코루틴은 해당 블록이 완료될 때까지 실행됩니다.

```
import kotlinx.coroutines.*

fun main() {
    runBlocking {
        // runBlocking 내부에서 실행될 코루틴
        launch {
            delay(1000L)
            println("runBlocking: 코루틴 실행 완료")
        }

        println("runBlocking: 메인 블록 실행 중")
    }
    println("메인 함수 종료")
}
```

Coroutine vs Thread

코루틴을 사용하는 이유와 스레드와의 차이점을 설명할게.

1. 경량성

- **코루틴**: 코루틴은 매우 경량이야. 수천 개의 코루틴을 생성해도 메모리와 CPU 자원을 비교적 적게 사용해. 코루틴이 스택을 사용하지 않고, 필요할 때만 실행되기 때문이지.
- **스레드**: 스레드는 운영체제 수준에서 관리되며, 생성과 전환 비용이 높아. 많은 수의 스레드를 생성하면 메모리와 CPU 자원을 많이 소모하게 돼.

2. 쉬운 비동기 코드 작성

- **코루틴**: 코루틴은 비동기 코드를 동기 코드처럼 작성할 수 있게 해줘. `suspend` 함수와 같은 기능을 통해 비동기 작업을 쉽게 표현할 수 있어.
- **스레드**: 스레드를 사용한 비동기 프로그래밍은 코드가 복잡해지고, 동기화 문제를 해결하기 위해 추가적인 코드가 필요해.

3. 구조화된 동시성

- **코루틴**: 코루틴은 구조화된 동시성을 제공해. 이는 코루틴의 수명 주기가 명확하게 정의되어 있어, 부모 코루틴이 끝나면 자식 코루틴도 자동으로 취소되는 등의 관리가 용이해.
- **스레드**: 스레드는 구조화된 동시성을 제공하지 않기 때문에, 스레드의 수명 주기를 관리하는 것이 더 어려워.

4. 비동기 흐름 제어

- **코루틴**: 코루틴은 `launch`, `async`, `withContext` 등의 빌더를 제공하여 비동기 흐름을 제어하는 것이 쉬워. 특히 `async`와 `await`를 사용하면 비동기 작업의 결과를 쉽게 얻을 수 있어.
- **스레드**: 스레드를 사용한 비동기 흐름 제어는 복잡하며, 콜백이나 Future/Promise 패턴을 사용해야 해.

5. 쉬운 취소와 타임아웃

- **코루틴**: 코루틴은 취소와 타임아웃을 쉽게 처리할 수 있어. `withTimeout`이나 `withTimeoutOrNull` 같은 함수로 간단하게 타임아웃을 설정할 수 있어.
- **스레드**: 스레드의 취소와 타임아웃 처리는 복잡하고, 안전하게 구현하기 어려워.

6. 더 나은 에러 처리

- **코루틴**: 코루틴은 `try-catch` 블록을 사용하여 에러를 처리할 수 있으며, 코루틴 스코프 내에서 발생한 예외를 쉽게 관리할 수 있어.
- **스레드**: 스레드의 에러 처리는 더 복잡하며, 예외가 발생한 스레드를 추적하고 관리하는 것이 어려워.

예제 코드 비교

코루틴 예제

```
import kotlinx.coroutines.*

fun main() = runBlocking {
    launch {
        delay(1000L)
        println("코루틴 1 완료")
    }
    launch {
        delay(500L)
        println("코루틴 2 완료")
    }
```

```
    println("메인 함수 실행 중")
}
```

스레드 예제

```
fun main() {
    val thread1 = Thread {
        Thread.sleep(1000L)
        println("스레드 1 완료")
    }
    val thread2 = Thread {
        Thread.sleep(500L)
        println("스레드 2 완료")
    }
    thread1.start()
    thread2.start()
    println("메인 함수 실행 중")
    thread1.join()
    thread2.join()
}
```

Suspend

`suspend` 함수는 Kotlin 코루틴에서 비동기 작업을 수행하기 위해 사용되는 함수야. `suspend` 키워드는 함수가 일시 중단(suspend)될 수 있으며, 나중에 다시 재개(resume)될 수 있음을 나타내. `suspend` 함수는 코루틴 내에서만 호출될 수 있어. 비동기 작업을 더 쉽게 작성하고 관리할 수 있도록 도와주지.

왜 `suspend` 함수를 사용해야 할까?

`suspend` 함수를 사용하면 다음과 같은 장점이 있어:

1. **비동기 작업을 동기 코드처럼 작성할 수 있어**:
    - `suspend` 함수를 사용하면 비동기 작업을 마치 동기 작업처럼 순차적으로 작성할 수 있어. 이는 코드의 가독성과 유지보수성을 높여줘.
2. **코루틴 내에서 일시 중단과 재개를 쉽게 처리할 수 있어**:
    - `suspend` 함수는 코루틴이 일시 중단될 수 있는 지점을 명확하게 정의해줘. 이를 통해 코루틴이 비동기 작업을 수행하는 동안 다른 작업을 처리할 수 있어.
3. **비동기 흐름 제어가 쉬워**:
    - `suspend` 함수는 다른 `suspend` 함수와 결합하여 복잡한 비동기 흐름을 쉽게 제어할 수 있어. 예를 들어, 여러 비동기 작업을 순차적으로 실행하거나 병렬로 실행할 수 있어.

`suspend` 함수의 예제

다음은 `suspend` 함수를 사용하는 예제야:

```
import kotlinx.coroutines.*

suspend fun fetchData(): String {
    delay(1000L) // 1초 동안 일시 중단
    return "데이터 가져오기 완료"
}

fun main() = runBlocking {
    println("비동기 작업 시작")
    val result = fetchData() // suspend 함수 호출
    println(result) // "데이터 가져오기 완료" 출력
    println("비동기 작업 종료")
}
```

설명

- `fetchData` 함수는 `suspend` 키워드로 정의되어 있어. 이 함수는 1초 동안 일시 중단된 후 문자열을 반환해.
- `runBlocking` 블록 내에서 `fetchData` 함수를 호출해. `runBlocking`은 현재 스레드를 블록하면서 코루틴을 실행하는 함수야.
- `fetchData` 함수가 호출되면 1초 동안 일시 중단되고, 이후 "데이터 가져오기 완료"를 반환해.
- 반환된 결과를 출력하면 "비동기 작업 종료"가 출력돼.

`suspend` 함수의 특성

1. **코루틴 내에서만 호출 가능**:
   - `suspend` 함수는 코루틴 내에서만 호출할 수 있어. `runBlocking`, `launch`, `async` 등 코루틴 빌더 내에서 호출해야 해.
2. **일시 중단과 재개**:
   - `suspend` 함수는 `delay`, `withContext`, `await` 등 다른 `suspend` 함수와 함께 사용되어 코루틴을 일시 중단하고 나중에 재개할 수 있어.
3. **비동기 작업의 효율적 관리**:
   - `suspend` 함수를 사용하면 비동기 작업을 효율적으로 관리할 수 있어. 예를 들어, 네트워크 요청, 파일 I/O, 데이터베이스 쿼리 등을 `suspend` 함수로 구현할 수 있어.

1. 코루틴을 이용한 I/O 작업 병렬 처리

아래는 코루틴을 사용하여 3개의 I/O 작업을 병렬로 처리한 뒤 결과를 반환하는 간단한 예제입니다.

```
import kotlinx.coroutines.*

fun main() = runBlocking {
    val start = System.currentTimeMillis()

    // 3개의 비동기 작업을 병렬 실행
    val result1 = async { performIO("Task1", 1000) }
    val result2 = async { performIO("Task2", 2000) }
    val result3 = async { performIO("Task3", 1500) }

    // 모든 작업 완료 대기 및 결과 결합
    println("Results: ${result1.await()}, ${result2.await()},
${result3.await()}")
    val end = System.currentTimeMillis()
    println("Execution time: ${end - start} ms")
}

suspend fun performIO(taskName: String, delayTime: Long): String {
    delay(delayTime)  // 실제 I/O 작업 대신 delay로 대체
    println("$taskName completed!")
    return "$taskName Result"
}
```

2. 스레드 기반 구현

```
import java.util.concurrent.Executors
import java.util.concurrent.Future

fun main() {
    val executor = Executors.newFixedThreadPool(3)
    val start = System.currentTimeMillis()

    val future1: Future<String> = executor.submit<String> {
performIOThread("Task1", 1000) }
    val future2: Future<String> = executor.submit<String> {
performIOThread("Task2", 2000) }
    val future3: Future<String> = executor.submit<String> {
performIOThread("Task3", 1500) }

    val results = listOf(future1.get(), future2.get(), future3.get())
    println("Results: $results")

    val end = System.currentTimeMillis()
    println("Execution time: ${end - start} ms")
    executor.shutdown()
```

```
}

fun performIOThread(taskName: String, delayTime: Long): String {
    Thread.sleep(delayTime)  // 실제 I/O 작업 대신 sleep으로 대체
    println("$taskName completed!")
    return "$taskName Result"
}
```

실습코드

https://youtu.be/Vs34wiuJMYk?si=vagZYLX9wO9XfHCc (새차원 코틀린 코루틴)

```
import kotlinx.coroutines.*
import kotlin.concurrent.thread


fun main(){

//    startCoroutine()
//    learnRunBlocking()
//    learnRunBlocking2()
//    learnJob1()
//    structuredConcurrency()
//    suspendFuntionSub()
//    coroutineIsLight()
    threadIsHeavy()
}

fun threadIsHeavy() = runBlocking {

    repeat(1000){
        thread { Thread.sleep(1000L)
            print("*")
        }
    }}

fun coroutineIsLight() = runBlocking {

    repeat(100_000){
        launch { delay(1000L)
        print("*")}
    }}
```

```
fun suspendFuntionSub() = runBlocking {
    launch{
        suspendFunction()
    }
    println("Hello,")

}


suspend fun suspendFunction(){
    delay(1000L)
    print("World")


}


fun structuredConcurrency() = runBlocking{
    this.launch {
        delay(1000L)
        println("World!")
    }
    this.launch {
        delay(1100L)
        println("World!")
    }
    this.launch {
        delay(1200L)
        println("World!")
    }
    this.launch {
        delay(1300L)
        println("World!")
    }
    this.launch {
        delay(1400L)
        println("World!")
    }

    println("Hello!")

}


fun learnJob1() =runBlocking {
```

```
    //launch 를 하면 job이 반환됨
    val job = GlobalScope.launch {
        delay(3000L)
        println("World")
    }

    println("Hello,")
    job.join() // join은 항상 Coroutine 함수 안에서 실행되어야함
// join은 모든 코루틴 job들이 끝나기를 기다려줌
}


fun learnRunBlocking2() = runBlocking{
    GlobalScope.launch {
        delay(1000L)
        println("World")
    }

    println("Hello,")
    delay(999L)


}

fun learnRunBlocking(){

    GlobalScope.launch{ // 코루틴을 반환 launch를 하려면 Scope가 필요

        delay(1000L)
        println("World!")
    }

    println("Hello,")

    runBlocking { delay(2000L) } // 이것또한 코루틴을 만들어 blockingCoroutine을 리
턴함


}

fun startCoroutine(){

    GlobalScope.launch{ // 코루틴을 반환 launch를 하려면 Scope가 필요

        delay(1000L)
        println("World!")
```

```
    }

    println("Hello,")

    runBlocking { delay(2000L) }

}
```