2~3장

Tags 동시성

코루틴 인 액션

코루틴을 시작하는 두가지 방법

- 코루틴을 시작하는 방법은 아래 두가지가 존재한다.

1. **Async 코루틴**
2. **launch 코루틴**

Async 코루틴

> 예외 미처리

```
fun main() = runBlocking {
    val task = GlobalScope.async {
        doSomething()
    }
    task.join()
    println("Completed")
}
fun doSomething() {
    throw UnsupportedOperationException("Can't do")
}
//출력 결과
예외 스택은 존재하지 않고 단지 Completed만 출력
```

- async()는 Deffered<T>를 반환
  - 취소 불가능한 넌블로킹 퓨처를 의마하며 T는 그 결과의 유형을 나타냄
- `GlobalScope` : 애플리케이션의 전체 생명 주기에 걸쳐 동작

예외 처리

```
@OptIn(InternalCoroutinesApi::class)
fun main() = runBlocking {
    val task = GlobalScope.async {
        doSomething()
    }
    task.join()
    if (task.isCancelled) {
        val exception = task.getCancellationException()
        println("Error with message: ${exception.cause}")
    } else {
        println("Success")
    }
}
//출력 결과
위에서 예외 미처리를 한 값과는 다르게 Error를 catch하는 것이 가능하다.
```

await 이용

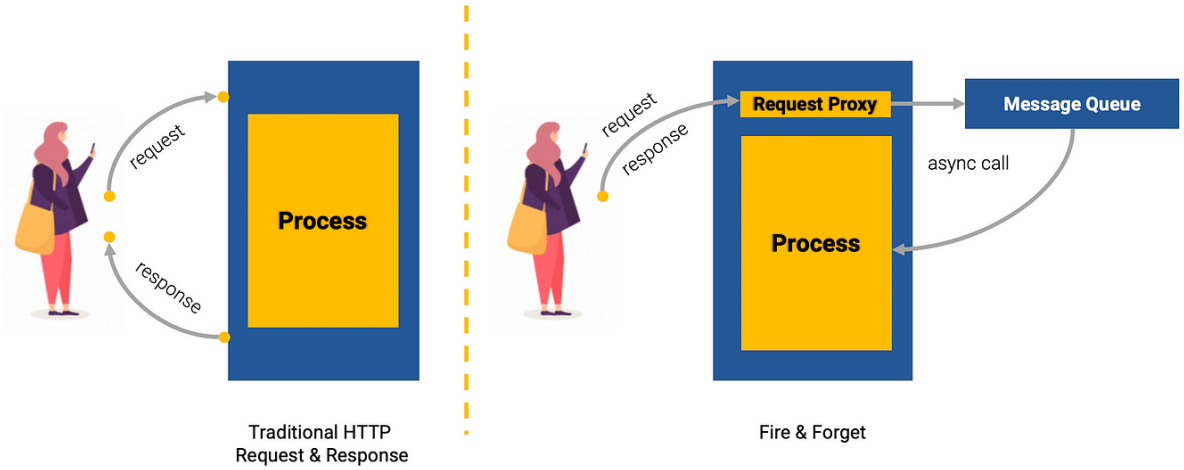
```
fun main() = runBlocking {
    val task = GlobalScope.async {
        doSomething()
    }
    task.await()
    println("Completed")
}
```

- await()를 호출해서 중단되는데 이 경우가 예외를 감싸지 않고 전파하는 `unwrapping deferred`다.

launch 코루틴

- 결과를 반환하지 않는 코루틴을 시작하려면 launch()를 사용해야 한다.
- 연산이 실패한 경우에만 통보받을수 있으면 필요할 때 취소할 수 있는 함수도 함께 제공된다.

fire-and-forget pattern



- 메시징에서 주로 사용되면 요청을 한 후 신경 쓸 필요 없는 경우와 같은 것을 말한다!

```
fun main() = runBlocking {
    val task = GlobalScope.launch {
        doSomething()
    }
    task.join()
    println("completed")
}
```

- 스택에 예외가 출력이 되긴 하지만 애플리케이션은 실행 완료가 되는 것을 알 수 있다. (completed가 출력됨)

코루틴 시작할 때 특정 디스패처 사용

Default Dispatcher 이용

```
fun main() = runBlocking {
    val task = launch {
        printCurrentThread()
    }
    task.join()
}

fun printCurrentThread() {
    println("Running in thread [${Thread.currentThread().name}]")
}
//main 출력
```

- 현재 스레드의 이름을 출력

Default Dispatcher 변경

```
fun main() = runBlocking {
    val dispatcher = newSingleThreadContext(name = "ServiceCall")
    val task = GlobalScope.launch(dispatcher) {
        printCurrentThread()
    }
    task.join()
}

fun printCurrentThread() {
    println("Running in thread [${Thread.currentThread().name}]")
}
//ServiceCall 출력
```

서비스 호출을 위한 코루틴 생성

```
private fun fetchRssHeadlines(): List<String> {
    val builder = factory.newDocumentBuilder()
    val xml = builder.parse("https://www.npr.org/rss/rss.php?id=1001")
    val news = xml.getElementsByTagName("channel").item(0)
    return (0 until news.childNodes.length)
        .asSequence()
        .map { news.childNodes.item(it) }
        .filter { Node.ELEMENT_NODE == it.nodeType }
        .map { it as Element }
        .filter { "item" == it.tagName }
        .map {
            it.getElementsByTagName("title").item(0).textContent
        }
        .toList()
}
```

```
//이렇게라도 출력해보자!
fun main() = runBlocking {
    val result = GlobalScope.async {
        fetchRssHeadlines()
    }
    val list = result.await()
    println(list)
}
```

- https://www.npr.org/rss/rss.php?id=1001 여기 파싱 안됨…
    - tasooning.tistory.com/rss 로 해보시길…

라이프 사이클과 에러 핸들링

- 잡과 디퍼드

잡과 디퍼드

- **결과가 없는 비동기 함수**
    - 일반적인 시나로으는 로그에 기록하고 분석 데이터를 전송하는 것과 같은 백그라운드 작업
    - 완료 여부를 모니터링할 수 있지만 결과를 갖지 않는 백그라운드 작업
- **결과를 반환하는 비동기 함수**
    - 비동기 함수가 웹 서비스에서 정보를 가져올 때 거의 대부분 해당 함수를 사용해 정보를 반환

잡

- 파이어-앤-포켓 작업
- 한번 시작된 작업은 예외가 발생하지 않는한 대기하지 않는다.

```
fun main() = runBlocking {
    val job = GlobalScope.launch {
```

```
        // Do background task here
    }
}
```

예외 처리

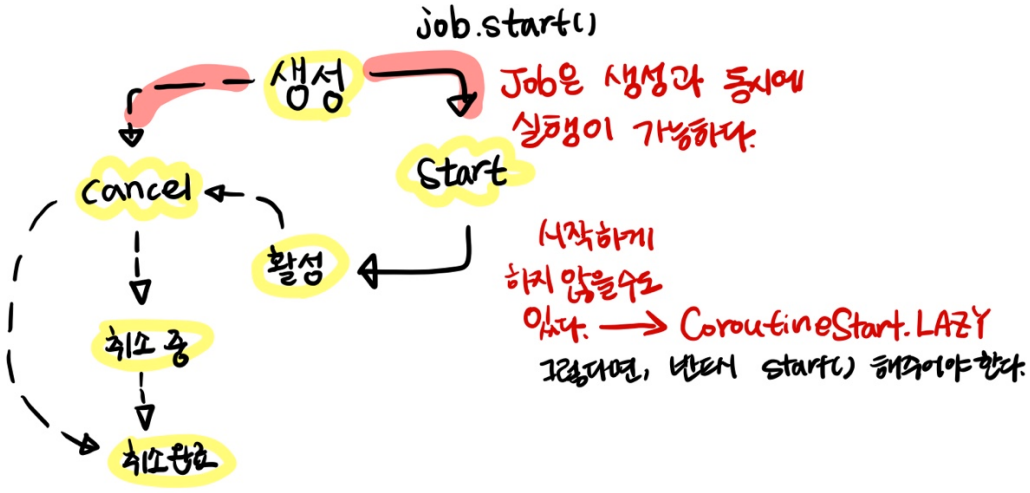
- 잡 내부에서 발생하는 예외는 잡을 생성한 곳까지 전파된다.
- 잡이 완료되기를 기다리지 않아다 발생한다.

```
fun main() = runBlocking {
    val job = GlobalScope.launch {
        TODO("Not Implemented!")
    }

    delay(500)
}
```

- 오류가 출력된다!
- 만약 delay가 없는 경우 먼저 Main 쓰레드가 종료되어 오류가 출력되지 않을 수 있다.

라이프사이클



상태	설명
New(생성)	존재하지만 아직 실행되지 않는 잡
Active(활성)	실행 중인 잡. 일시 중단된 잡도 활성으로 간주된다
Completed(완료 됨)	잡이 더 이상 실행되지 않는 경우
Canceling(취소 중)	실행 중인 잡에서 cancel()이 호출되면 취소가 완료될 때까지 시간이 걸리기도 한다. 이것은 활성과 취소 사이의 중간 상태다.
Cancelled(취소 됨)	취소로 인해 실행이 완료된 잡. 취소된 잡도 완료로 간주될 수 있다

생성

- 잡은 기본적으로 launch()나 job()을 사용해 생성될 때 자동으로 시작
  - 자동으로 시작되지 않게 하려면 `CoroutineStart.LAZY`

```
fun main() = runBlocking{
  GlobalScope.launch(start = CoroutineStart.LAZY){
    TODO("Not implemented yet!")
  }
  delay(500)
}
```

- 코드를 실행하면 오류 출력되지 않는다. 작업이 생성됐지만 시작된 적이 없어서!

활성

- 생성 상태에 있는 잡은 다양한 방법으로 실행 가능한데 일반적으로 start()나 join()을 호출해서 실행
  - 둘의 차이점은 전자는 잡이 완료될 때까지 기다리지 않고 잡을 시작하는 반면 후자는 잡이 완료될 때까지 실행을 일시 중단한다는 점

start

```
fun main() {
    val job = GlobalScope.launch(start = CoroutineStart.LAZY){
        delay(3000)
    }
    job.start()
}
```

- `start()` 는 실행을 일시 중단하지 않으므로 일시 중단 함수(suspending function)나 코루틴에서 호출할 필요가 없다. 애플리케이션의 어느 부분에서도 호출할 수 있다.

join

```
fun main() = runBlocking{
    val job = GlobalScope.launch(start = CoroutineStart.LAZY){
        delay(3000)
    }
    job.join()
}
```

- join()은 실행을 일시 중단할 수 있으므로 코루틴 또는 일시 중단 함수에서 호출해야 한다. 이를 위해 runBlocking()이 사용되고 있다.

취소중

- 취소 요청을 받은 활성 잡은 취소중이라고 하는 스테이징 상태로 들어갈 수 있다.
- cancel()을 이용해 호출

```
job.cancel()


job.cancel(cause = Exception("Tiemout!")) //이거 안됨...
job.cancel(cause = CancellationException("Time out!!")) //이거써야될듯?
```

취소됨

- 취소 또는 처리되지 않은 예외로 인해 실행이 종료된 잡은 취소됨으로 간주
- 잡이 취소되면 `getCacellationException()` 함수를 통해 취소해 대한 정보를 얻을 수 있다.

```
@OptIn(InternalCoroutinesApi::class)
fun main() = runBlocking{
    val job = GlobalScope.launch(start = CoroutineStart.LAZY){
        delay(5000)
    }

    delay(2000)

    job.cancel(cause = CancellationException("Time out!!"))

    val cancellation = job.getCancellationException()
    println(cancellation.message)
}
```

- 취소된 잡과 예외로 실패한 잡을 구별하기 힘들다.

CoroutineExceptionHandler 사용

```
fun main() = runBlocking {
    val exceptionHandler = CoroutineExceptionHandler { _: CoroutineContext, throwable ->
        println("Job cancelled due to ${throwable.message}")
    }

    GlobalScope.launch(exceptionHandler) {
        TODO("Not implemented yet!")
    }
```

```
    delay(2000)
}

//출력
Job cancelled due to An operation is not implemented: Not implemented yet!
```

invokeOnComletion() 사용

```
fun main() = runBlocking {
    GlobalScope.launch {
        TODO("Not implemented yet!")
    }.invokeOnCompletion { cause->
        cause?.let {
            println("Job cancelled due to ${it.message}")
        }
    }
}
```

- 개인적으로 이게 깔끔해보인다…

완료됨

- 실행이 중지된 잡은 `완료됨(completed)`로 간주
- 실행이 정상적으로 종료됐거나 취소됐는지 또는 예외 때문에 종료됐는지 관계없이 적용

잡의 현재 상태 확인

isActive : 잡이 활성 상태인지 여부, 잡이 일시 중지인 경우도 true 반환

isCompleted : 잡이 실행을 완료했는지 여부

isCancelled : 잡 취소 여부, 취소가 요청되면 즉시 true

상태	isActive	isCompleted	isCancelled
생성됨(Created)	false	false	false

상태	isActive	isCompleted	isCancelled
활성(Active)	true	false	false
취소중(Cancelling)	false	false	true
취소됨(Cancelled)	false	true	true
완료됨(Completed)	false	true	false

디퍼드

- 디퍼드는 결과를 갖는 비동기 작업을 수행하기 위해 작을 확장한다.
- 일반적으로 Future, Promise와 비슷한 개념이다.

```
fun main() = runBlocking{
  val headlines = GlobalScope.async {
    getHeadlines()
  }
  headlinesTask.await()
}
```

- 디퍼드는 Futuredhk ehddmldjek.

```
val ary =  CompletableDeferred<List<Any>>()
```

- 이런식으로도 표현 가능

예외처리

- 순수한 잡과 달리 디퍼드는 처리되지 않은 예외를 자동으로 전파하지 않는다.
  - 디퍼드의 결과를 대기할 것으로 예상하기 때문이다!

- 예외가 발생하게 하려면 await()를 해야 한다.

```
fun main(): Unit = runBlocking {
    val deferred = GlobalScope.async {
```

```
        TODO("Not implemented yet!")
    }

    kotlin.runCatching {
        deferred.await()
    }.onFailure {
        println(it.message)
    }
}
```

- 이처럼 예외처리 하는 것도 가능하다.

상태는 한 방향으로만 이동

- 잡이 특정 상태에 도달하면 이전 상태로 돌아가지 않는다.
- 만약 job이 complete가 되었을 때 다시 start를 해도 해당 잡은 실행되지 않는다.

```
일부 잡의 상태는 최종 상태로 간주(최종 상태는 잡을 옮길수 없는 상태)
ex) 취소됨,완료됨
```

RSS- 여러 피드에서 동시에 읽기

피드 목록 지원

```
private fun asyncFetchRssHeadlines(feed: String, dispatcher: CoroutineDispatcher) = GlobalScope.async(dispatcher) {
    val builder = factory.newDocumentBuilder()
    val xml = builder.parse(feed)
```

```
    val news = xml.getElementsByTagName("channel").item(0)
    (0 until news.childNodes.length)
        .asSequence()
        .map { news.childNodes.item(it) }
        .filter { Node.ELEMENT_NODE == it.nodeType }
        .map { it as Element }
        .filter { "item" == it.tagName }
        .map {
            it.getElementsByTagName("title").item(0).textContent
        }
        .toList()
}

fun main() {
    val feeds = listOf(
        "https://tasooning.tistory.com/rss",
        "https://reviewjju.tistory.com/rss",
        "https://shlee0882.tistory.com/rss"
    )
}
```

스레드 풀 만들기

```
private val dispatcher = newFixedThreadPoolContext(2,"IO")
```

- `asyncFetchRssHeadlines` 는 서버에서 정보를 가져올 뿐 아니라 파싱도 하기 때문에 풀의 크기를 늘린다.

⇒ 이렇게 하면 단일 스레드에서 처리하는게 아니라 다른 스레드들도 사용?

데이터 동시에 가져오기

```
val requests = mutableListOf<Deferred<List<String>>>()

feeds.mapTo(requests) {
    asyncFetchRssHeadlines(it, dispatcher)
}

requests.forEach {
    it.await() // 각 코드가 완료될 때까지 대기
}
```

응답 병합 / 예외처리

```
private val dispatcher = newFixedThreadPoolContext(2,"IO")
private val factory = DocumentBuilderFactory.newInstance()
val feeds = listOf(
    "https://tasooning.tistory.com/rss",
    "https://reviewjju.tistory.com/rss",
    "https://shlee0882.tistory.com/rss"
)


private fun asyncFetchRssHeadlines(feed: String, dispatcher: CoroutineDispatcher) = GlobalScope.async(dispatcher) {
    val builder = factory.newDocumentBuilder()
    val xml = builder.parse(feed)
    val news = xml.getElementsByTagName("channel").item(0)
    (0 until news.childNodes.length)
        .asSequence()
        .map { news.childNodes.item(it) }
        .filter { Node.ELEMENT_NODE == it.nodeType }
        .map { it as Element }
        .filter { "item" == it.tagName }
        .map {
            it.getElementsByTagName("title").item(0).textContent
        }
        .toList()
}

@OptIn(ExperimentalCoroutinesApi::class, DelicateCoroutinesApi::class)
private fun asyncLoadNews() = GlobalScope.launch {
    val requests = mutableListOf<Deferred<List<String>>>()

    feeds.mapTo(requests) {

        asyncFetchRssHeadlines(it, dispatcher)
    }

    requests.forEach {
        it.join() //await() 대신 join()을 사용하면 예외가 전파되지 않는다.
    }
    val headlines = requests
        .filter { !it.isCancelled }
        .flatMap { it.getCompleted() }

    println(headlines) //헤드라인 출력

    val failed = requests
        .filter { it.isCancelled }
        .size

}

suspend fun main() {
    asyncLoadNews()
    delay(1000) //delay를 주어야 출력 가능 (아니면 바로 메인 쓰레드가 끝나버림)
}
```

요약

- 잡은 아무것도 반환하지 않는 백그라운드 작업에 사용
- 디퍼드는 백그라운드 작업이 수신하려는 것을 반환할 때 사용
- 디퍼드는 잡을 확장해 무언가를 반환할 가능성을 높인다.
- 디퍼드가 가질 수 있는 상태는 잡의 상태와 같다.