스프링캠프 2024

동시성의 미래
코루틴과 버추얼 스레드

이상훈

강연자 소개

- 12년차 백엔드 엔지니어
- 법률 인공지능 스타트업 **엘박스**에서 근무 중
- 경력의 절반이 코틀린 사용
- 페이스북 그룹 **Kotlin 한국 유저 모임** 운영자
- 코틀린 & 백엔드 주제로 온라인 강의 경험
- digimon.1740@gmail.com

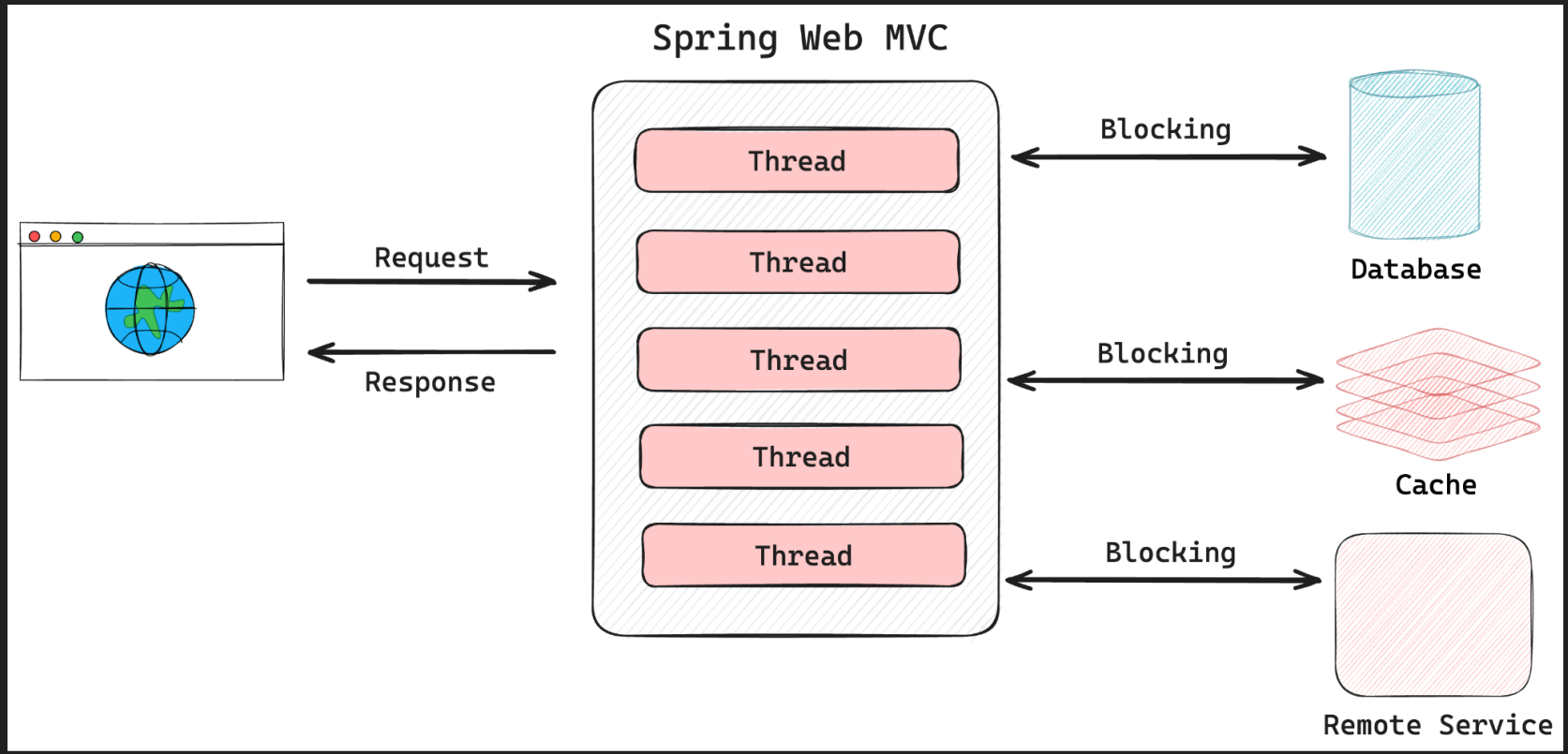
발표 순서

- 전통적 웹 방식
- 리액티브 프로그래밍
- 코틀린 코루틴
- 버추얼 스레드
- 코루틴과 버추얼 스레드의 통합

전통적 웹 방식

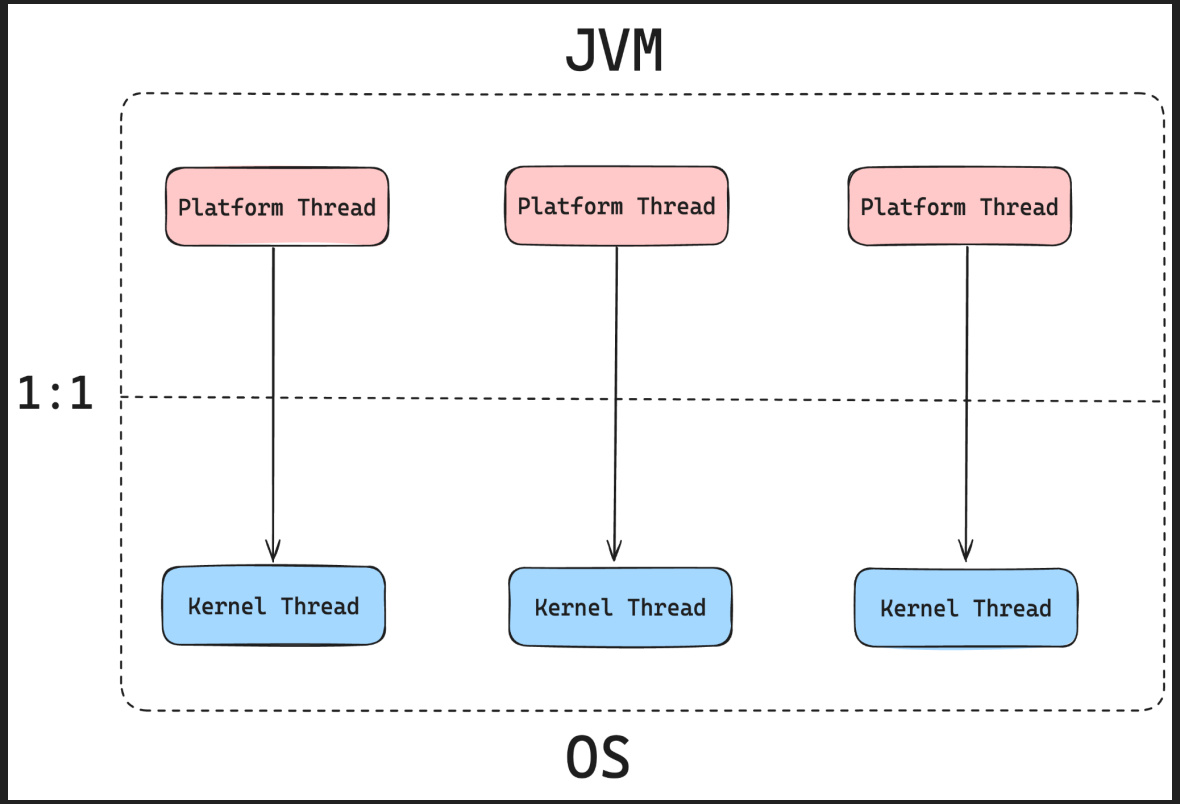
전통적 웹 방식

전통적 웹 방식은 1개 요청당 1개의 스레드를 사용하는 **Thread Per Request** 모델



전통적 웹 방식

플랫폼 스레드



플랫폼 스레드와 커널 스레드가 1:1 매핑되는 구조이다

자원이 한정되므로 무한정 스레드를 늘릴 수없음

정리하면

- 전통적인 방식은 IO 처리에 대한 결과를 받기 전까지 스레드 블로킹 발생
- 커널 스레드를 사용하기 때문에 무한정 생성 할 수 없어 스레드 풀을 사용
- 웹 서버의 최대 처리량은 스레드 풀에 생성된 스레드의 수로 제한
- 비동기-논블로킹 방식을 사용하면 **더 적은 스레드로 더 많은 일**을 할 수 있지만 가독성과 유지보수성 저하

리액티브 프로그래밍

리액티브 프로그래밍

- 비동기-논블로킹의 문제가 되는 **콜백헬**을 함수형 프로그래밍 관점으로 해결
- 대표적 표준은 **리액티브 스트림즈**이다
- **비동기 데이터 스트림**과 **논블로킹-백프레셔**에 대한 사양 제공
- Project Reactor, RxJava, Akka Streams
- 스프링에서는 Spring WebFlux를 사용해 리액티브 웹 스택 적용 가능

리액티브 프로그래밍

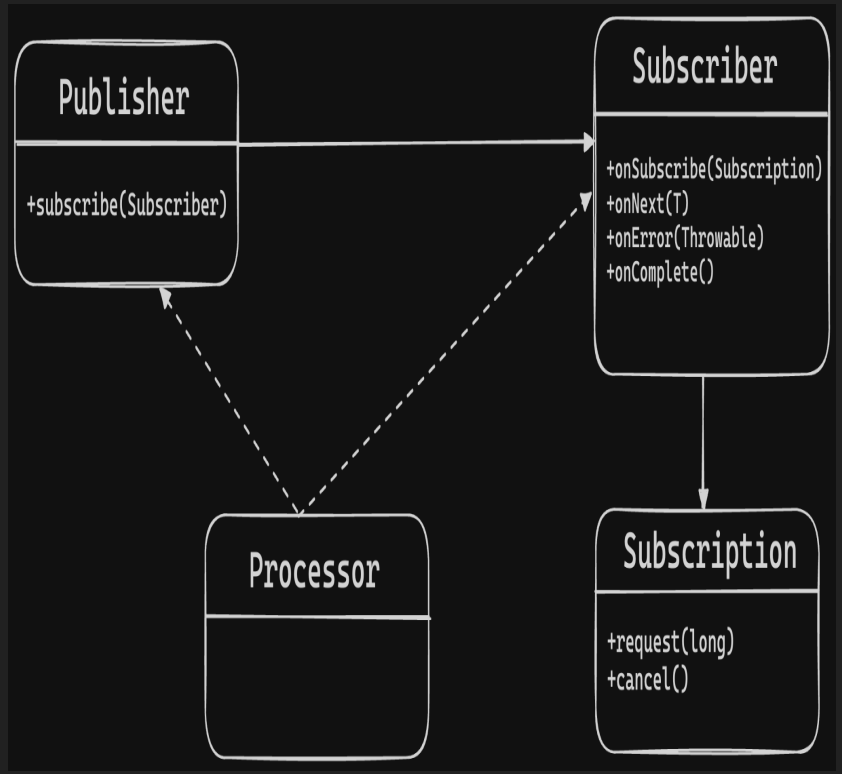
```kotlin
@GetMapping("/users/{userId}")
fun getUserResponse(
    @PathVariable("userId") userId: Long,
): Mono<UserCompositeResponse> =
    Mono.zip(
        getUser(userId),
        getFollower(userId),
    ).map { (user, follower) ->
        UserCompositeResponse(
            user = user,
            follower = follower,
        )
    }


fun getUser(userId: Long): Mono<UserResponse> =
    webClient
        .get()
        .uri("http://example.dev/api/users/${userId}")
        .retrieve()
        .bodyToMono(UserResponse::class.java)


fun getFollower(userId: Long): Mono<FollowerResponse> =
    webClient
        .get()
        .uri("http://example.dev/api/users/${userId}/follower")
        .retrieve()
        .bodyToMono(FollowerResponse::class.java)
```

여러 비동기 작업을 병렬로 수행한 후 하나의 결과로 합침

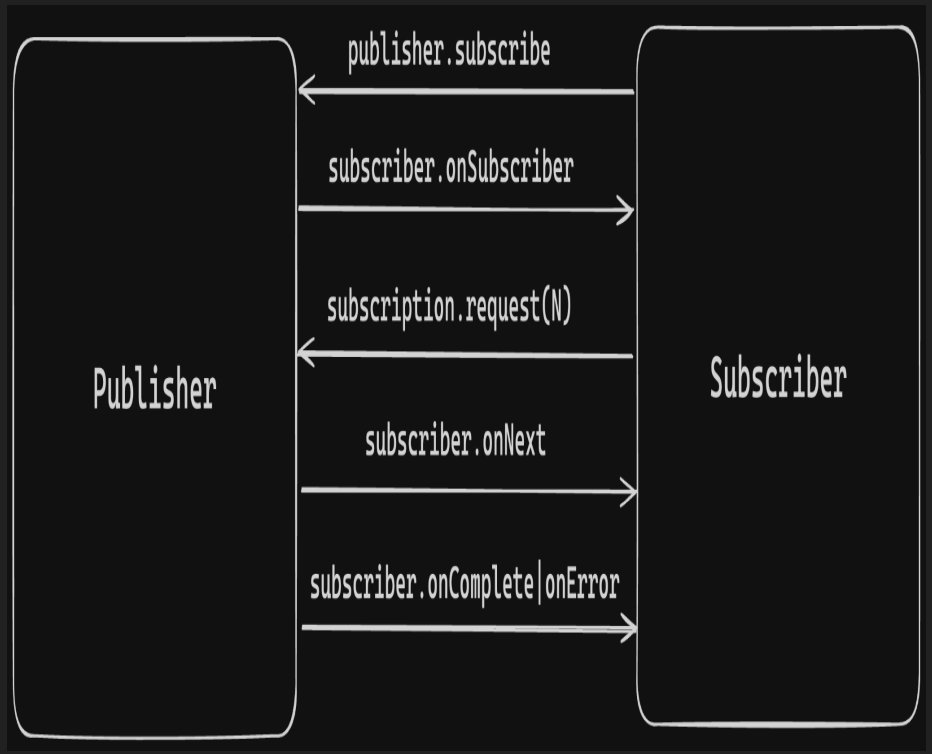
리액티브 프로그래밍 ...
정말 좋지만 제대로 쓰려면 공부할게 많다

리액티브 프로그래밍

리액티브 스트림즈 핵심 인터페이스

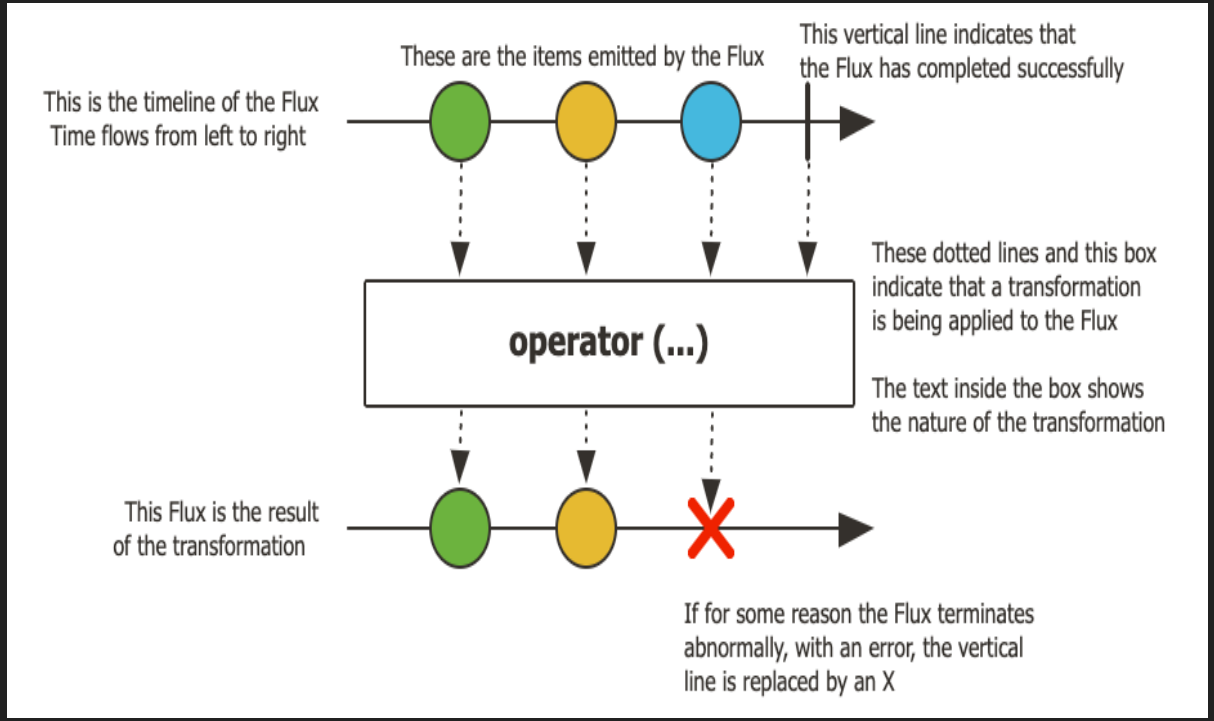


리액티브 스트림즈 프로토콜



리액티브 프로그래밍

마블 다이어그램



다양한 연산자의 동작 원리를 이해하려면 마블 다이어그램 이해가 필수

리액티브 프로그래밍

WebFlux에서 블로킹 API가 호출된다면..?

```
@Service
class UserService(
    private val userJpaRepository: UserJpaRepository,
) {

    fun getUserBlocking(userId: Long): Mono<User> {
        return userJpaRepository.findById(userId)
            .toMono()
    }

}
```

블로킹 API

```
@Service
class UserService(
    private val userJpaRepository: UserJpaRepository,
) {

    fun getUser(userId: Long): Mono<User> {
        return Mono.fromCallable {
            userJpaRepository.findById(userId)
        }.subscribeOn(Schedulers.boundedElastic())
    }

}
```

블로킹 API를 별도의 스레드 풀 위에서 실행

정리하면

- 리액티브 프로그래밍은 비동기-논블로킹 방식의 여러 문제를 해결해 준다
- Spring WebFlux를 사용해 리액티브 웹 스택으로 개발할 수 있다
- 충분한 학습 없이 도입하면 많은 문제가 발생할 수 있다
- 대용량 데이터에 대한 스트리밍 처리 등에서는 다양한 연산자 제공을 통해 여전히 매력적임

코틀린 코루틴

코틀린 코루틴

코루틴은 코틀린에서 비동기 프로그래밍을 손쉽게 작성할 수 있게 도와주는 **확장 라이브러리**이다

코틀린 코루틴

Using in your projects

Maven

Add dependencies (you can also add other modules that you need):

```
<dependency>
    <groupId>org.jetbrains.kotlinx</groupId>
    <artifactId>kotlinx-coroutines-core</artifactId>
    <version>1.8.1-Beta</version>
</dependency>
```

And make sure that you use the latest Kotlin version:

```
<properties>
    <kotlin.version>1.9.21</kotlin.version>
</properties>
```

Gradle

Add dependencies (you can also add other modules that you need):

```
dependencies {
    implementation("org.jetbrains.kotlinx:kotlinx-coroutines-core:1.8.1-Beta")
}
```

And make sure that you use the latest Kotlin version:

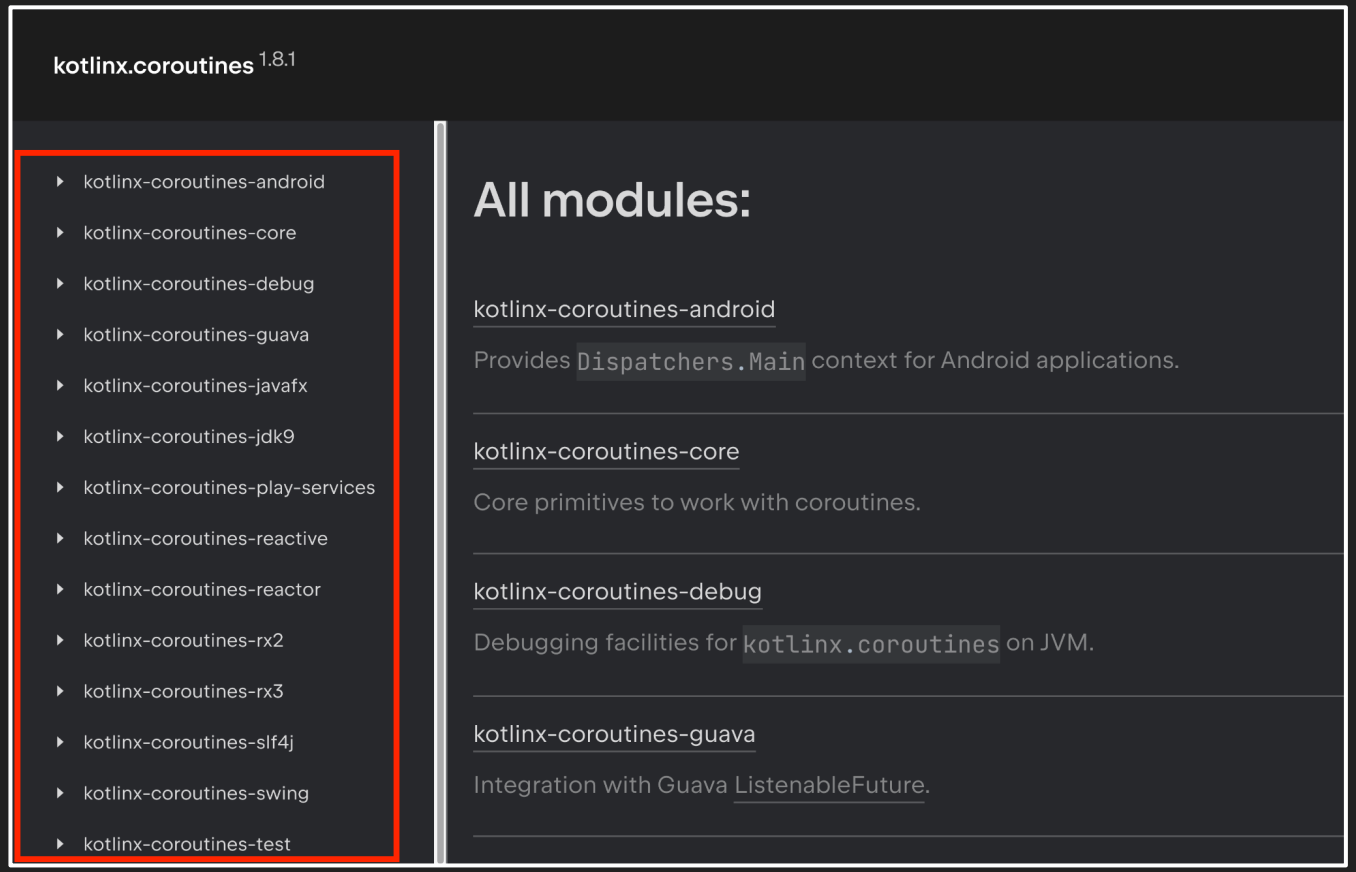
```
plugins {
    // For build.gradle.kts (Kotlin DSL)
    kotlin("jvm") version "1.9.21"

    // For build.gradle (Groovy DSL)
    id "org.jetbrains.kotlin.jvm" version "1.9.21"
}
```

Make sure that you have `mavenCentral()` in the list of repositories:

```
repositories {
    mavenCentral()
}
```

코틀린 코루틴

코틀린 코루틴의 라이브러리 상호 운용성



코틀린 코루틴

리액티브 코드를 코루틴으로 리팩토링

```kotlin
@GetMapping("/users/{userId}")
fun getUserResponse(
    @PathVariable("userId") userId: Long,
): Mono<UserCompositeResponse> =
    Mono.zip(
        getUser(userId),
        getFollower(userId),
    ).map { (user, follower) ->
        UserCompositeResponse(
            user = user,
            follower = follower,
        )
    }

fun getUser(userId: Long): Mono<UserResponse> =
    webClient
        .get()
        .uri("http://localhost:9090/api/users/${userId}")
        .retrieve()
        .bodyToMono(UserResponse::class.java)


fun getFollower(userId: Long): Mono<FollowerResponse> =
    webClient
        .get()
        .uri("http://localhost:9090/api/users/${userId}/follower")
        .retrieve()
        .bodyToMono(FollowerResponse::class.java)
```

```kotlin
@GetMapping("/users/{userId}")
suspend fun getUserResponseAwait(
    @PathVariable("userId") userId: Long,
): UserCompositeResponse = coroutineScope {

    val user = async { getUser(userId) }
    val follower = async { getFollower(userId) }

    UserCompositeResponse(user.await(), follower.await())
}

suspend fun getUser(userId: Long): UserResponse =
    webClient
        .get()
        .uri("http://localhost:9090/api/users/${userId}")
        .retrieve()
        .awaitBody()


suspend fun getFollower(userId: Long): FollowerResponse =
    webClient
        .get()
        .uri("http://localhost:9090/api/users/${userId}/follower")
        .retrieve()
        .awaitBody()
```

코틀린 코루틴

리액티브 코드를 코루틴으로 리팩토링

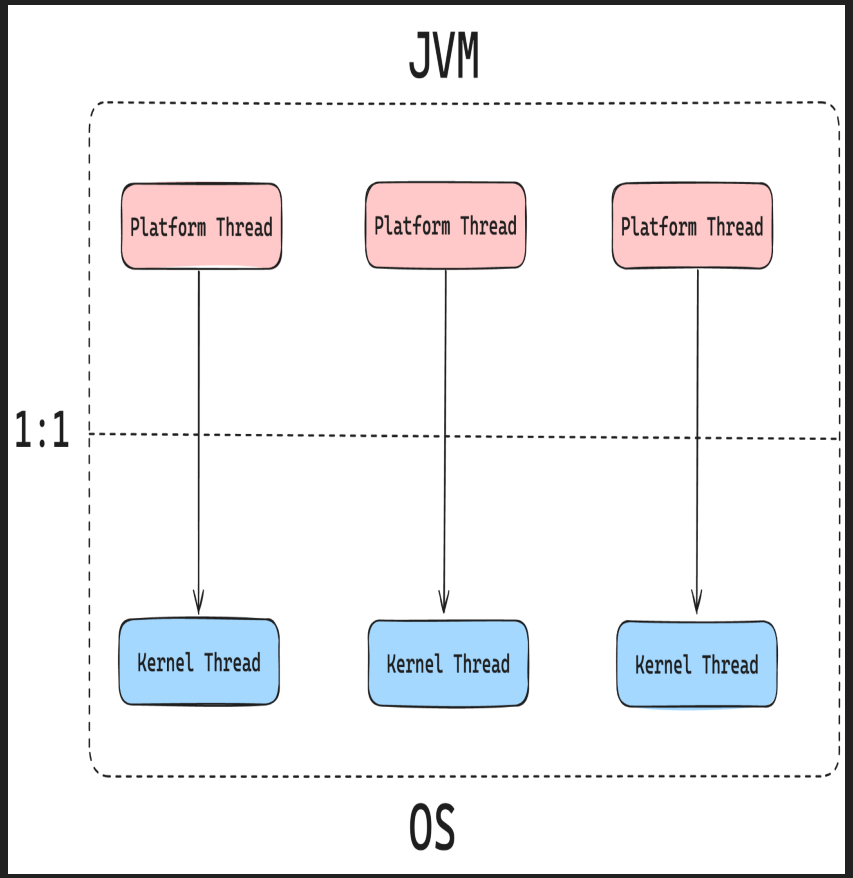
```kotlin
@GetMapping("/users/{userId}")
fun getUserResponse(
    @PathVariable("userId") userId: Long,
): Mono<UserCompositeResponse> =
    Mono.zip(
        getUser(userId),
        getFollower(userId),
    ).map { (user, follower) ->
        UserCompositeResponse(
            user = user,
            follower = follower,
        )
    }

fun getUser(userId: Long): Mono<UserResponse> =
    webClient
        .get()
        .uri("http://localhost:9090/api/users/${userId}")
        .retrieve()
        .bodyToMono(UserResponse::class.java)


fun getFollower(userId: Long): Mono<FollowerResponse> =
    webClient
        .get()
        .uri("http://localhost:9090/api/users/${userId}/follower")
        .retrieve()
        .bodyToMono(FollowerResponse::class.java)
```

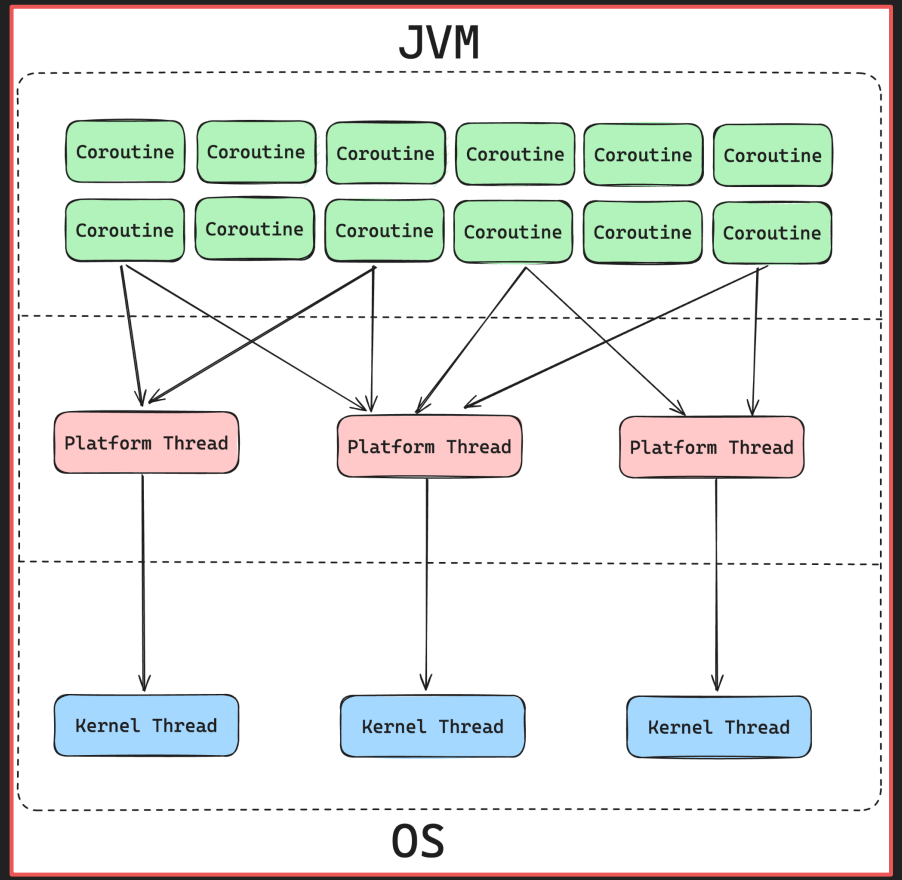
suspend

```kotlin
@GetMapping("/users/{userId}")
fun getUserResponseAwait(
thVariable("userId") userId: Long,
CompositeResponse = coroutineScope {

    val user = async { getUser(userId) }
    val follower = async { getFollower(userId) }

    UserCompositeResponse(user.await(), follower.await())
}

suspend fun getUser(userId: Long): UserResponse =
    webClient
        .get()
        .uri("http://localhost:9090/api/users/${userId}")
        .retrieve()
        .awaitBody()


suspend fun getFollower(userId: Long): FollowerResponse =
    webClient
        .get()
        .uri("http://localhost:9090/api/users/${userId}/follower")
        .retrieve()
        .awaitBody()
```

코루틴은 매우 가볍다

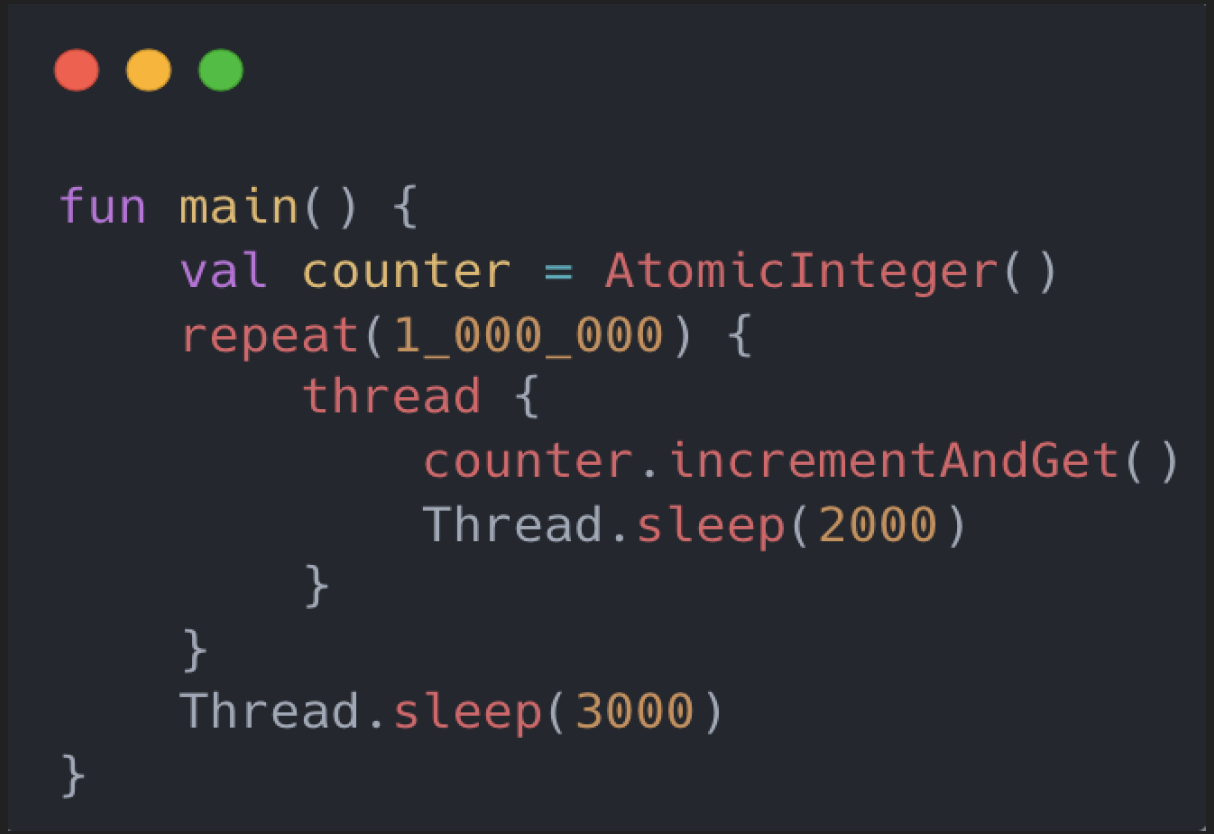
코틀린 코루틴


JVM
Platform Thread
Platform Thread
Platform Thread
1:1
Kernel Thread
Kernel Thread
Kernel Thread
OS


JVM
Coroutine
Coroutine
Coroutine
Coroutine
Coroutine
Coroutine
Coroutine
Coroutine
Coroutine
Coroutine
Coroutine
Coroutine
Platform Thread
Platform Thread
Platform Thread
Kernel Thread
Kernel Thread
Kernel Thread
OS

코틀린 코루틴

스레드 100만개를 생성하는 예제



```
fun main() {
    val counter = AtomicInteger()
    repeat(1_000_000) {
        thread {
            counter.incrementAndGet()
            Thread.sleep(2000)
        }
    }
    Thread.sleep(3000)
}
```

코틀린 코루틴

네이티브 스레드 생성 제한에 도달해 **OutOfMemoryError** 발생

```
[0.309s][warning][os,thread] Failed to start thread "Unknown thread" - pthread_create failed (EAGAIN) for attributes: stacksize: 2048k, guardsize: 16k, detached.
[0.309s][warning][os,thread] Failed to start the native thread for java.lang.Thread "Thread-4074"
Exception in thread "main" java.lang.OutOfMemoryError: unable to create native thread: possibly out of memory or process/resource limits reached
    at java.base/java.lang.Thread.start0(Native Method)
    at java.base/java.lang.Thread.start(Thread.java:1526)
    at kotlin.concurrent.ThreadsKt.thread(Thread.kt:42)
    at kotlin.concurrent.ThreadsKt.thread$default(Thread.kt:20)
    at Example1Kt.main(example1.kt:34)
    at Example1Kt.main(example1.kt)
```

코틀린 코루틴

코루틴 100만개를 생성하는 예제

```kotlin
fun main() = runBlocking {
    val counter = AtomicInteger()
    repeat( times: 1_000_000) {
        launch {
            counter.incrementAndGet()
            delay( timeMillis: 2000)
        }
    }
    delay( timeMillis: 3000)
    println("totalCoroutines : ${counter.get()}")

}
```

Example1Kt

```
/Users/sanghoon/Library/Java/JavaVirtualMachines/temurin-21.0.1/Contents/Home/bin/java ...
totalCoroutines : 1000000

Process finished with exit code 0
```

코틀린 코루틴

```kotlin
fun main() = runBlocking {
    val counter = AtomicInteger()
    repeat( times: 1_000_000) {
        launch {
            counter.incrementAndGet()
            delay( timeMillis: 2000)
        }
    }
    delay( timeMillis: 3000)
    println("totalCoroutines : ${counter.get()}")

}
```

Example1Kt

```
/Users/sanghoon/Library/Java/JavaVirtualMachines/temurin-21.0.1/Contents/Home/bin/java ...
totalCoroutines : 1000000

Process finished with exit code 0
```

delay 함수는 지정된 시간 만큼 코루틴을 대기시킨다.

Thread.sleep과 차이점은 스레드가 블로킹되지 않는다.

코틀린 코루틴

```
fun main() = runBlocking {
    val counter = AtomicInteger()
    repeat( times: 1_000_000) {
        launch {
            counter.incrementAndGet()
            Thread.sleep( millis: 2000)
        }
    }
    delay( timeMillis: 3000)
    println("totalCoroutines : ${counter.get()}")
}
```

delay가 아닌 **Thread.sleep**를 사용하면 ?

스레드가 블로킹되는 환경에선 코루틴도 성능이 떨어질 수 밖에 없다

코틀린 코루틴은 비동기-논블로킹 환경에서 사용하는 것이 적절

코루틴 빌더 :
코루틴을 생성하는 함수

코틀린 코루틴

```
fun main() {

    runBlocking {
        println("Hello")
    }
    println("World")
}

// Hello
// World
```

runBlocking에 감싸진 코드는 모든 수행이 끝날 때까지 스레드가 블로킹된다

테스트 코드, 스프링 배치 등 코루틴을 지원하지 않거나 블로킹 스택과 브릿지하는 경우에 사용

코틀린 코루틴

```
fun main() =  runBlocking {
    launch {
        delay(500L)
        println("World!")
    }
    println("Hello")
}

// Hello
// World!
```

launch는 스레드 차단 없이 새로운 코루틴을 시작하며 결과를 만들어내지 않는 비동기 작업에 적합

코틀린 코루틴

```
fun sum(a: Int, b: Int) = a + b

fun main() = runBlocking<Unit> {

    val result1: Deferred<Int> = async {
        delay(100)
        sum(1, 3)
    }

    println("result1 : ${result1.await()}")

    val result2: Deferred<Int> = async {
        delay(100)
        sum(2, 5)
    }

    println("result2 : ${result2.await()}")
}

// result1 : 4
// result2 : 7
```

async 빌더는 비동기 작업을 통해 결과를 만들어내는 경우 적합하다

비동기 작업의 결과로 **Deferred**를 반환하는데 **await** 함수를 통해 async로 수행한 비동기 작업의 결과를 받아올 수 있다

구조적 동시성 :
코루틴의 생명주기와 예측 가능성 향상

코틀린 코루틴

```
suspend fun main() {
    doSomething()
}

private suspend fun doSomething() = coroutineScope {

    launch {
        delay(200)
        println("world!")
    }

    launch {
        println("hello")
    }

}

// hello
// world!
```

스레드 블로킹 없이 각 동시 작업의 수행이 완료된 후 함수가 종료된다

코틀린 코루틴

구조적 동시성에서 예외를 다루는 방법 : 모든 코루틴 취소

```
suspend fun main() = coroutineScope<Unit> {

    launch {
        delay(200)
        println("world!")

    }

    launch {
        throw RuntimeException()
    }

}

Exception in thread "main" java.lang.RuntimeException
    at structuredconcurrency.TryCatchKt$main$2$2.invokeSuspend(tryCatch.kt:14)
    at kotlin.coroutines.jvm.internal.BaseContinuationImpl.resumeWith(ContinuationImpl.kt:33)
    at kotlinx.coroutines.DispatchedTask.run(DispatchedTask.kt:106)
    at kotlinx.coroutines.scheduling.CoroutineScheduler.runSafely(CoroutineScheduler.kt:570)
    at kotlinx.coroutines.scheduling.CoroutineScheduler$Worker.executeTask(CoroutineScheduler.kt:750)
    at kotlinx.coroutines.scheduling.CoroutineScheduler$Worker.runWorker(CoroutineScheduler.kt:677)
    at kotlinx.coroutines.scheduling.CoroutineScheduler$Worker.run(CoroutineScheduler.kt:664)
```

coroutineScope 내부의 자식 코루틴에서 에러가 발생하면 모든 코루틴이 종료된다

코틀린 코루틴

구조적 동시성에서 예외를 다루는 방법 : 빌더 내부에서 **try-catch**로 핸들링

```
suspend fun main() = coroutineScope<Unit> {

    launch {
        delay(200)
        println("world!")

    }

    launch {
        try {
            throw RuntimeException()
        } catch (e: Exception) {
            println("hello")
        }
    }
}

// hello
// world!
```

코틀린 코루틴

구조적 동시성에서 예외를 다루는 방법 : **SupervisorScope**

```
suspend fun main() = coroutineScope<Unit> {

    launch {
        delay(200)
        println("예외가 발생해도 출력!")

    }

    supervisorScope {
        launch {
            throw RuntimeException()
        }
    }
}
```

supervisorScope를 사용해 예외를 부모 코루틴으로 전파하지 않는 방법

```
Exception in thread "DefaultDispatcher-worker-2" java.lang.RuntimeException
    at structuredconcurrency.CoroutineScopeKt$main$2$2$1.invokeSuspend(coroutineScope.kt:20)
    at kotlin.coroutines.jvm.internal.BaseContinuationImpl.resumeWith(ContinuationImpl.kt:33)
    at kotlinx.coroutines.DispatchedTask.run(DispatchedTask.kt:106)
    at kotlinx.coroutines.scheduling.CoroutineScheduler.runSafely(CoroutineScheduler.kt:570)
    at kotlinx.coroutines.scheduling.CoroutineScheduler$Worker.executeTask(CoroutineScheduler.kt:750)
    at kotlinx.coroutines.scheduling.CoroutineScheduler$Worker.runWorker(CoroutineScheduler.kt:677)
    at kotlinx.coroutines.scheduling.CoroutineScheduler$Worker.run(CoroutineScheduler.kt:664)
    Suppressed: kotlinx.coroutines.DiagnosticCoroutineContextException:
[StandaloneCoroutine{Cancelling}@7b8a66f9, Dispatchers.Default]

예외가 발생해도 출력!
```

코틀린 코루틴

구조적 동시성에서 예외를 다루는 방법 : **NonCancellable**

```
suspend fun main() = coroutineScope<Unit> {

    withContext(NonCancellable) {
        launch {
            delay(200)
            println("예외가 발생해도 출력!")
        }
    }

    launch {
        throw RuntimeException()
    }

}
```

다른 코루틴에서 예외가 발생해도 작업이 수행된다

```
예외가 발생해도 출력!
Exception in thread "main" java.lang.RuntimeException
    at structuredconcurrency.TryCatchKt$main$2$2.invokeSuspend(tryCatch.kt:15)
    at kotlin.coroutines.jvm.internal.BaseContinuationImpl.resumeWith(ContinuationImpl.kt:33)
    at kotlinx.coroutines.DispatchedTask.run(DispatchedTask.kt:106)
    at kotlinx.coroutines.scheduling.CoroutineScheduler.runSafely(CoroutineScheduler.kt:570)
    at kotlinx.coroutines.scheduling.CoroutineScheduler$Worker.executeTask(CoroutineScheduler.kt:750)
    at kotlinx.coroutines.scheduling.CoroutineScheduler$Worker.runWorker(CoroutineScheduler.kt:677)
    at kotlinx.coroutines.scheduling.CoroutineScheduler$Worker.run(CoroutineScheduler.kt:664)
```

코루틴 컨텍스트 :
코루틴이 실행되는 환경을 제어한다

코틀린 코루틴

코루틴이 동작하는 스레드를 결정하는 방법 : **Dispatchers**

```
fun main() = runBlocking<Unit> {

    launch(Dispatchers.IO) {
        println("launch1 Thread-name : ${Thread.currentThread().name}")
    }

    launch {
        println("launch2 Thread-name : ${Thread.currentThread().name}")
    }

}
```

```
launch1 Thread-name : DefaultDispatcher-worker-1 @coroutine#2
launch2 Thread-name : main @coroutine#3
```

코틀린 코루틴

코루틴이 동작하는 스레드를 결정하는 방법 : **Dispatchers**

```
public actual object Dispatchers {

    @JvmStatic
    public actual val Default: CoroutineDispatcher = DefaultScheduler

    @JvmStatic
    public actual val Main: MainCoroutineDispatcher get() = MainDispatcherLoader.dispatcher

    @JvmStatic
    public actual val Unconfined: CoroutineDispatcher = kotlinx.coroutines.Unconfined

    @JvmStatic
    public val IO: CoroutineDispatcher = DefaultIoScheduler

}
```

코틀린 코루틴

ThreadLocal의 데이터를 코루틴에 전파하는 방법

```
@GetMapping("/users/{userId}/order")
suspend fun getUserAndOrder(
    @PathVariable("userId") userId: Long,
) {
    MDC.put("requestId", UUID.randomUUID().toString())

    logger.info { "call remote api" }

    coroutineScope {
        val user = async(Dispatchers.IO) {
            logger.info { "call user api" }
            getUser(userId)
        }

        val order = async(Dispatchers.IO) {
            logger.info { "call order api" }
            getOrder(userId)
        }

        UserOrderResponse(
            user = user.await(),
            order = order.await()
        )
    }

}
```

```
[reactor-http-nio-2]         INFO  [requestId=e2a75398-3b39-48f3-b5a2-57aeb9ec53a5] call remote api
[DefaultDispatcher-worker-1] INFO  [requestId=] call user api
[DefaultDispatcher-worker-1] INFO  [requestId=] call order api
```

코틀린 코루틴

ThreadLocal의 데이터를 코루틴에 전파하는 방법

```kotlin
@GetMapping("/users/{userId}/order")
suspend fun getUserAndOrder(
    @PathVariable("userId") userId: Long,
) {
    MDC.put("requestId", UUID.randomUUID().toString())

    logger.info { "call remote api" }

    withContext(MDCContext()) {

        val user = async(Dispatchers.IO) {
            logger.info { "call user api" }
            getUser(userId)
        }

        val order = async(Dispatchers.IO) {
            logger.info { "call order api" }
            getOrder(userId)
        }

        UserOrderResponse(
            user = user.await(),
            order = order.await()
        )
    }
}
```

ThreadContextElement를 사용해 ThreadLocal의 데이터를 코루틴에 전파할 수 있다

```
[reactor-http-nio-2]         INFO  [requestId=bc5cb224-5d0b-4643-bf05-10625605155f] call remote api
[DefaultDispatcher-worker-1] INFO  [requestId=bc5cb224-5d0b-4643-bf05-10625605155f] call user api
[DefaultDispatcher-worker-1] INFO  [requestId=bc5cb224-5d0b-4643-bf05-10625605155f] call order api
```

정리하면

- 비동기 작업을 동기식 방식 처럼 순차적인 코드를 작성할 수 있다
- **경량 스레드**로 적은 리소스로 대규모 동시성 작업을 효율적으로 지원
- **구조적 동시성**으로 계층 구조를 명확히 정의하고 생명 주기 관리가 쉽다
- **코루틴 컨텍스트**를 사용해 상황에 따라 실행 환경을 제어할 수 있다
- 다만, 블로킹 환경에서 사용하면 성능 향상이 높지 않다
- 코루틴을 사용하는 측에서도 **suspend** 를 선언해야한다

버추얼 스레드

버추얼 스레드

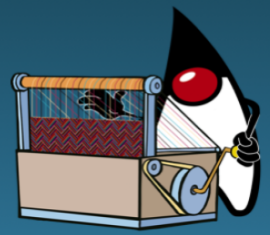
https://wiki.openjdk.org/display/loom

Main

Created by Iris Clark, last modified by Ron Pressler on Aug 01, 2022

Project Loom

Fibers and Continuations



Project Loom is to intended to explore, incubate and deliver Java VM features and APIs built on top of them for the purpose of supporting easy-to-use, high-throughput lightweight concurrency and new programming models on the Java platform.

This OpenJDK project is sponsored by the HotSpot Group.

Source Code
https://github.com/openjdk/loom

Early Access Binaries
http://jdk.java.net/loom/

버추얼 스레드

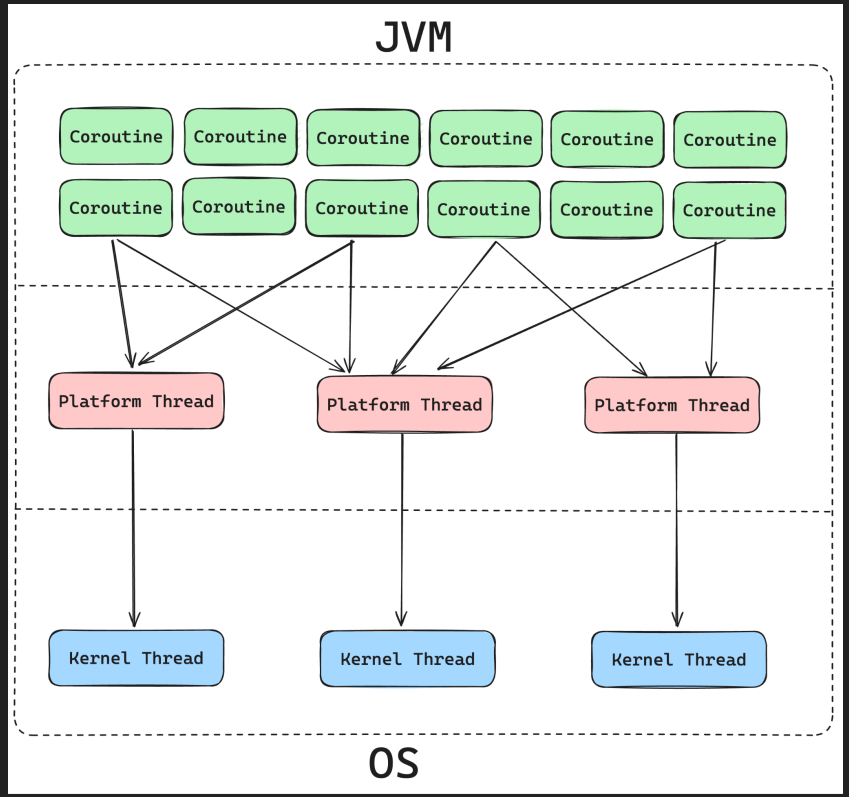
Project Loom은 Java 플랫폼 위에서 경량으로 동작하며 높은 처리량을 달성하면서 사용하기 쉬운 새로운 프로그래밍 모델을 지원하기 위해 만들어졌습니다.

Project Loom is to intended to explore, incubate and deliver Java VM features and APIs built on top of them for the purpose of supporting easy-to-use, high-throughput lightweight concurrency and new programming models on the Java platform.

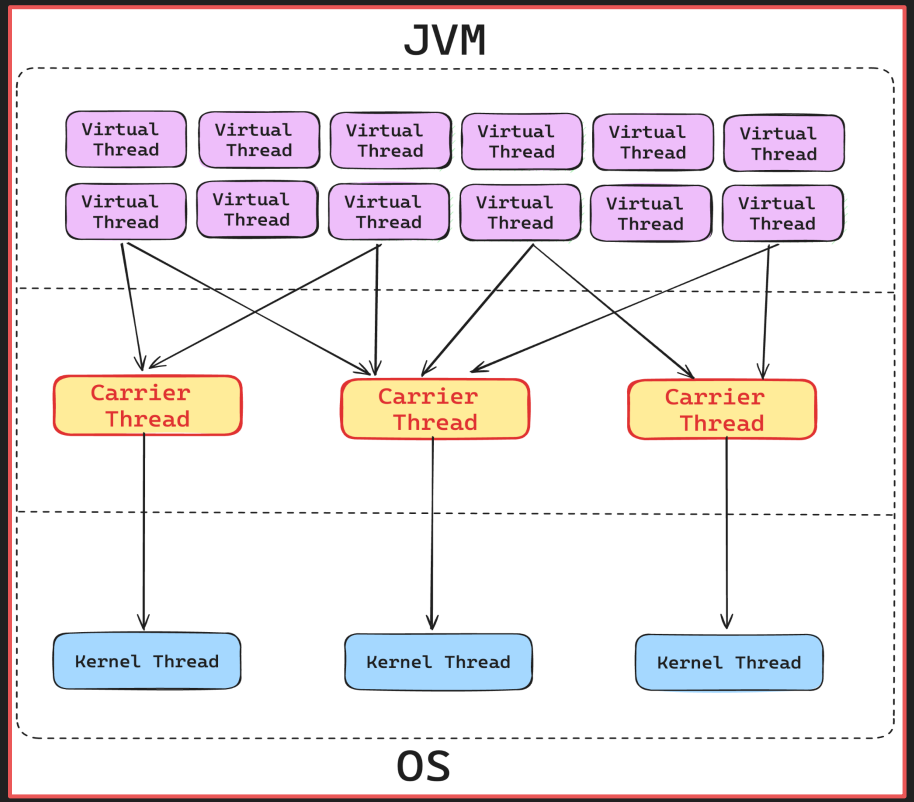
This OpenJDK project is sponsored by the HotSpot Group.

버추얼 스레드

코틀린 코루틴



버추얼 스레드



버추얼 스레드

버추얼 스레드를 생성하는 방법

```
fun main() {

    Thread.startVirtualThread {
        println("Thread.startVirtualThread started!!")
    }

    Thread.ofVirtual().start {
        println("Thread.ofVirtual started!!")
    }

    Executors.newVirtualThreadPerTaskExecutor().execute {
        println("Executors.newVirtualThreadPerTaskExecutor started!!")
    }

    Thread.sleep(10)
}


Thread.startVirtualThread started!!
Thread.ofVirtual started!!
Executors.newVirtualThreadPerTaskExecutor started!!
```

버추얼 스레드

버추얼 스레드 100만개를 생성하는 예제

```kotlin
fun main() {
    val counter = AtomicInteger()
    repeat(times: 1_000_000) {
        Thread.startVirtualThread {
            counter.incrementAndGet()
            Thread.sleep(millis: 2000)
        }
    }
    Thread.sleep(millis: 3000)
    println("totalVirtualThreads : ${counter.get()}")
}
```

Example1Kt

```
/Users/sanghoon/Library/Java/JavaVirtualMachines/temurin-21.0.1/Contents/Home/bin/java
totalVirtualThreads : 1000000
```

버추얼 스레드

버추얼 스레드 스프링 부트 통합 : **Spring Boot 3.2 이전**

```kotlin
@EnableAsync
@Configuration
class VirtualThreadConfig {

    @Bean(TaskExecutionAutoConfiguration.APPLICATION_TASK_EXECUTOR_BEAN_NAME)
    fun asyncTaskExecutor(): AsyncTaskExecutor {
        return TaskExecutorAdapter(Executors.newVirtualThreadPerTaskExecutor())
    }

    @Bean
    fun protocolHandlerVirtualThreadExecutorCustomizer(): TomcatProtocolHandlerCustomizer<*> {
        return TomcatProtocolHandlerCustomizer { protocolHandler: ProtocolHandler ->
            protocolHandler.executor = Executors.newVirtualThreadPerTaskExecutor()
        }
    }
}
```

버추얼 스레드

버추얼 스레드 스프링 부트 통합 : **Spring Boot 3.2 이후**

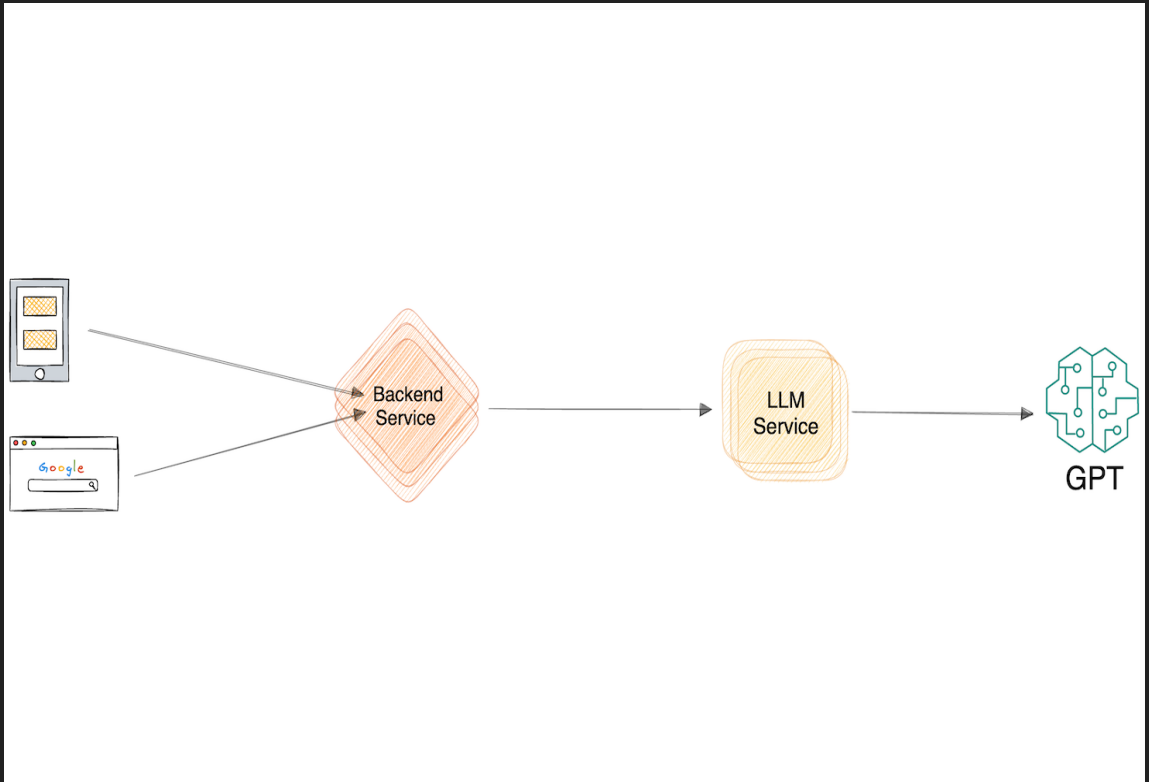
```yaml
# application.yml
spring:
  threads:
    virtual:
      enabled: true
```

```properties
# application.properties
spring.thread.virtual.enabled=true
```

버추얼 스레드 성능 테스트

버추얼 스레드

BE에서 LLM을 서빙하는 서비스를 호출하는 시나리오



기술 스택

- Spring WebMVC 3.2+
- RestClient

LLM API

- **응답 시간 : 20s**

측정 조건

- 테스트 수행 시간 : 10s
- 요청 수 : 100/s
- 총 요청 수 : 1000개

버추얼 스레드

성능 테스트 : 플랫폼 스레드 (스레드 200개)

```
@Component
class AiClient(
    private val aiRestClient: RestClient,
) {

    fun ask(request: AskRequest): AskResponse {
        val response = aiRestClient.post()
            .uri("/gpt/ask")
            .body(request)
            .retrieve()
            .body(AskResponse::class.java)

        return response ?: AskResponse(answer = "No answer")
    }

}
```

버추얼 스레드

성능 테스트 : 플랫폼 스레드 (스레드 200개)

```
echo 'GET http://localhost:8080/ask' | vegeta attack -duration=10s -rate=100/s | vegeta report

Requests      [total, rate, throughput]         1000, 100.10, 5.00
Duration      [total, attack, wait]             39.991s, 9.99s, 30.001s
Latencies     [min, mean, 50, 90, 95, 99, max]  20.003s, 28.004s, 30.001s, 30.001s, 30.001s, 30.001s, 30.006s
Bytes In      [total, mean]                     54000, 54.00
Bytes Out     [total, mean]                     0, 0.00
Success       [ratio]                           20.00%
Status Codes  [code:count]                      0:800  200:200
Error Set:
Get "http://localhost:8080/ask": context deadline exceeded (Client.Timeout exceeded while awaiting headers)
```

측정 결과

- 평균 소요시간 : 28.29초
- 성공 응답 수 : 200개
- 오류 응답 수 : 800개
- 성공률 : 20%

버추얼 스레드

성능 테스트 : 코루틴 + IO 디스패쳐 (스레드 64개)

```kotlin
@Component
class AiClient(
    private val aiRestClient: RestClient,
) {

    suspend fun ask(request: AskRequest) = withContext(Dispatchers.IO) {
        val response = aiRestClient.post()
            .uri("/gpt/ask")
            .body(request)
            .retrieve()
            .body(AskResponse::class.java)

        response ?: AskResponse(answer = "No answer")
    }

}
```

버추얼 스레드

성능 테스트 : 코루틴 + IO 디스패쳐 (스레드 64개)

```
echo 'GET http://localhost:8080/ask' | vegeta attack -duration=10s -rate=100/s | vegeta report

Requests      [total, rate, throughput]         1000, 100.10, 1.60
Duration      [total, attack, wait]             39.991s, 9.99s, 30.001s
Latencies     [min, mean, 50, 90, 95, 99, max]  20.005s, 29.361s, 30.001s, 30.001s, 30.001s, 30.001s, 30.002s
Bytes In      [total, mean]                     1280, 17.28
Bytes Out     [total, mean]                     0, 0.00
Success       [ratio]                           6.40%
Status Codes  [code:count]                      0:936  200:64
Error Set:
Get "http://localhost:8080/ask": context deadline exceeded (Client.Timeout exceeded while awaiting headers)
```

측정 결과

- 평균 소요시간 : 28.48초
- 성공 응답 수 : 64개
- 오류 응답 수 : 936개
- 성공률 : 6.40%

버추얼 스레드

성능 테스트 : 코루틴 + IO 디스패쳐 (스레드 200개)

```kotlin
@Component
class AiClient(
    private val aiRestClient: RestClient,
) {

    val limitedPool = Dispatchers.IO.limitedParallelism(200)

    suspend fun ask(request: AskRequest) = withContext(limitedPool) {
        val response = aiRestClient.post()
            .uri("/gpt/ask")
            .body(request)
            .retrieve()
            .body(AskResponse::class.java)

        response ?: AskResponse(answer = "No answer")
    }

}
```

버추얼 스레드

성능 테스트 : 코루틴 + IO 디스패쳐 (스레드 200개)

```
Requests      [total, rate, throughput]         1000, 100.10, 5.00
Duration      [total, attack, wait]             39.991s, 9.99s, 30.001s
Latencies     [min, mean, 50, 90, 95, 99, max]  20.002s, 28.007s, 30.001s, 30.001s, 30.001s, 30.002s, 30.003s
Bytes In      [total, mean]                     54000, 54.00
Bytes Out     [total, mean]                     0, 0.00
Success       [ratio]                           20.00%
Status Codes  [code:count]                      0:800  200:200
Error Set:
Get "http://localhost:8080/ask": context deadline exceeded (Client.Timeout exceeded while awaiting headers)
```

측정 결과

- 평균 소요시간 : 28.28초
- 성공 응답 수 : 200개
- 오류 응답 수 : 800개
- 성공률 : 20%

버추얼 스레드

성능 테스트 : 버추얼 스레드

```
@Component
class AiClient(
    private val aiRestClient: RestClient,
) {

    fun ask(request: AskRequest): AskResponse {
        val response = aiRestClient.post()
            .uri("/gpt/ask")
            .body(request)
            .retrieve()
            .body(AskResponse::class.java)

        return response ?: AskResponse(answer = "No answer")
    }

}
```

```
spring:
  threads:
    virtual:
      enabled: true
```

버추얼 스레드

성능 테스트 : 버추얼 스레드

```
echo 'GET http://localhost:8080/ask' | vegeta attack -duration=10s -rate=100/s | vegeta report

Requests      [total, rate, throughput]         1000, 100.09, 33.34
Duration      [total, attack, wait]             29.997s, 9.991s, 20.006s
Latencies     [min, mean, 50, 90, 95, 99, max]  20.002s, 20.009s, 20.006s, 20.008s, 20.009s, 20.131s, 20.222s
Bytes In      [total, mean]                     270000, 270.00
Bytes Out     [total, mean]                     0, 0.00
Success       [ratio]                           100.00%
Status Codes  [code:count]                      200:1000
Error Set:
```

측정 결과

- 평균 소요시간 : 20.06초
- 성공 응답 수 : 1000개
- 오류 응답 수 : 0개
- 성공률 : 100%

버추얼 스레드

성능 테스트 : Spring WebFlux + 코루틴

```kotlin
@Component
class AiClient(
    private val aiWebClient: WebClient,
) {

    suspend fun ask(request: AskRequest): AskResponse {
        val response = aiWebClient.post()
            .uri("/gpt/ask")
            .bodyValue(request)
            .retrieve()
            .awaitBodyOrNull<AskResponse>()

        return response ?: AskResponse(answer = "No answer")
    }
}
```

버추얼 스레드

성능 테스트 : Spring WebFlux + 코루틴

```
echo 'GET http://localhost:8081/ask' | vegeta attack -duration=10s -rate=100/s | vegeta report

Requests      [total, rate, throughput]         1000, 100.10, 33.34
Duration      [total, attack, wait]             29.996s, 9.99s, 20.006s
Latencies     [min, mean, 50, 90, 95, 99, max]  20.003s, 20.011s, 20.006s, 20.008s, 20.009s, 20.212s, 20.306s
Bytes In      [total, mean]                     270000, 270.00
Bytes Out     [total, mean]                     0, 0.00
Success       [ratio]                           100.00%
Status Codes  [code:count]                      200:1000
Error Set:
```

측정 결과

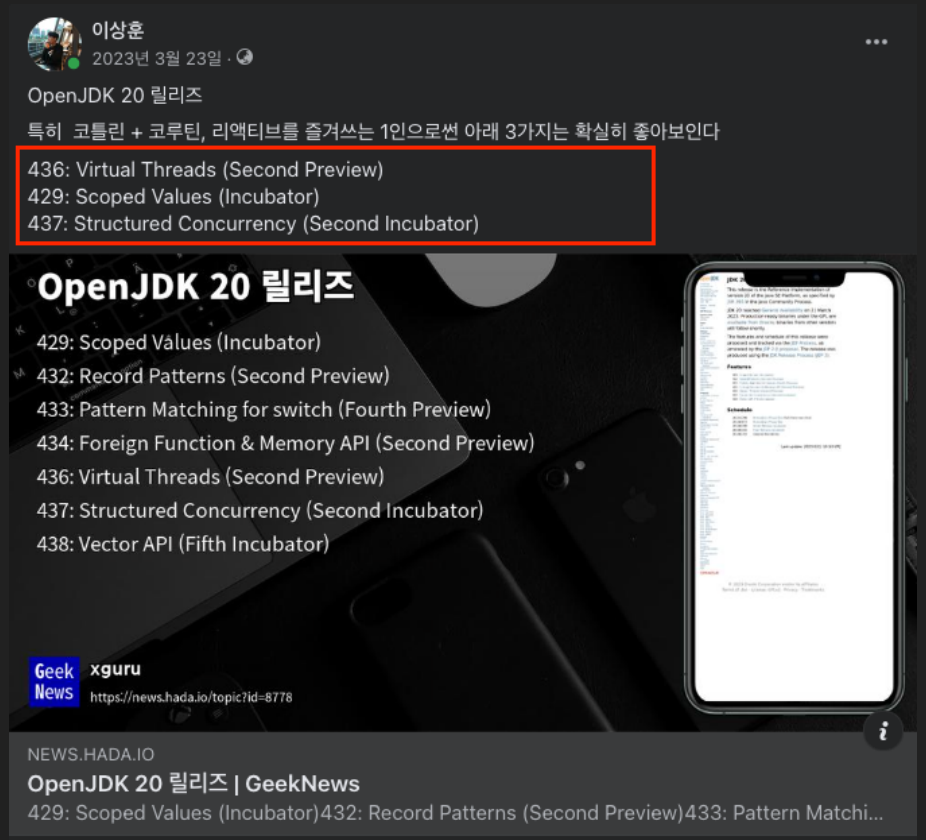
- 평균 소요시간 : 20.08초
- 성공 응답 수 : 1000개
- 오류 응답 수 : 0개
- 성공률 : 100%

정리하면

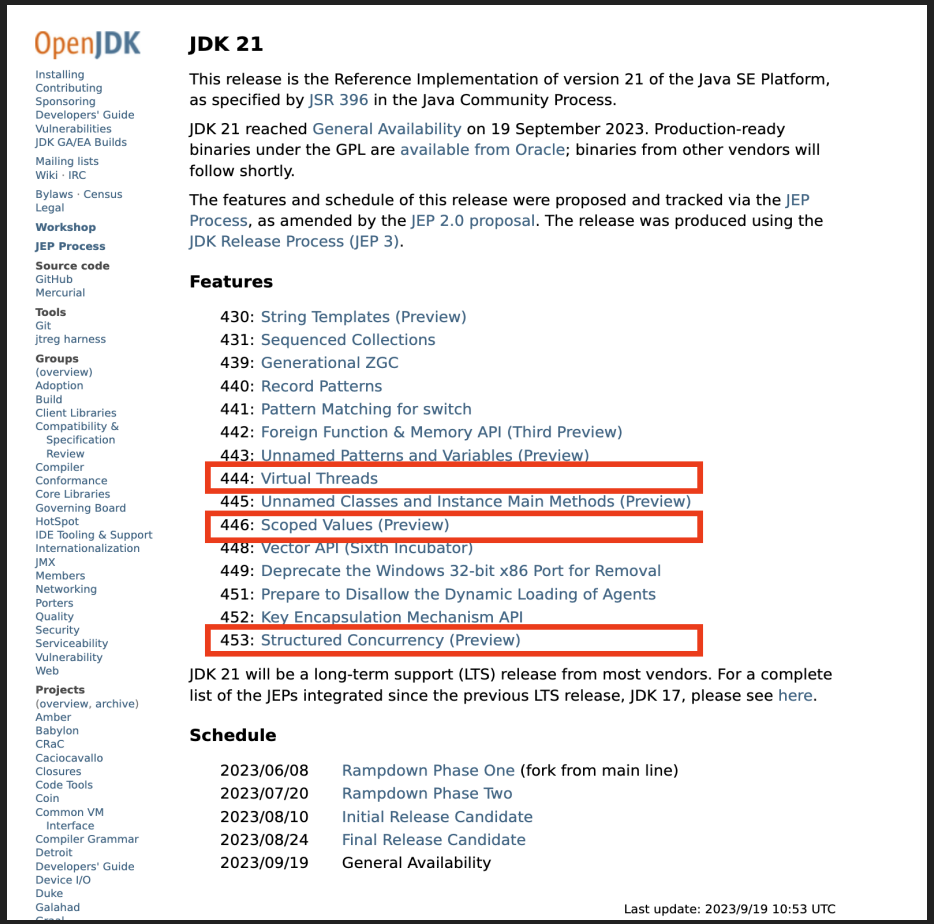
- 전통적인 동기식 코드 그대로 작성 가능
- JVM 레벨의 경량 스레드로 적은 리소스로 대규모 동시성 작업을 효율적으로 지원
- 버추얼 스레드는 기존 스레드 API를 그대로 활용할 수 있도록 설계
- Spring Web MVC + 버추얼 스레드는 Spring WebFlux + 코루틴과 거의 비슷한 성능을 보여준다
- Pinned 이슈 등 아직 성능을 100% 내기 어려운 환경이 있으므로 리서치 필요

코루틴과 버추얼 스레드의 통합

코루틴과 버추얼 스레드의 통합



코루틴과 버추얼 스레드의 통합



OpenJDK

Installing
Contributing
Sponsoring
Developers' Guide
Vulnerabilities
JDK GA/EA Builds

Mailing lists
Wiki · IRC

Bylaws · Census
Legal

Workshop

JEP Process

Source code
GitHub
Mercurial

Tools
Git
jtreg harness

Groups
(overview)
Adoption
Build
Client Libraries
Compatibility & Specification Review
Compiler
Conformance
Core Libraries
Governing Board
HotSpot
IDE Tooling & Support
Internationalization
JMX
Members
Networking
Porters
Quality
Security
Serviceability
Vulnerability
Web

Projects
(overview, archive)
Amber
Babylon
CRaC
Caciocavallo
Closures
Code Tools
Coin
Common VM Interface
Compiler Grammar
Detroit
Developers' Guide
Device I/O
Duke
Galahad

JDK 21

This release is the Reference Implementation of version 21 of the Java SE Platform, as specified by JSR 396 in the Java Community Process.

JDK 21 reached General Availability on 19 September 2023. Production-ready binaries under the GPL are available from Oracle; binaries from other vendors will follow shortly.

The features and schedule of this release were proposed and tracked via the JEP Process, as amended by the JEP 2.0 proposal. The release was produced using the JDK Release Process (JEP 3).

Features

430: String Templates (Preview)
431: Sequenced Collections
439: Generational ZGC
440: Record Patterns
441: Pattern Matching for switch
442: Foreign Function & Memory API (Third Preview)
443: Unnamed Patterns and Variables (Preview)
444: Virtual Threads
445: Unnamed Classes and Instance Main Methods (Preview)
446: Scoped Values (Preview)
448: Vector API (Sixth Incubator)
449: Deprecate the Windows 32-bit x86 Port for Removal
451: Prepare to Disallow the Dynamic Loading of Agents
452: Key Encapsulation Mechanism API
453: Structured Concurrency (Preview)

JDK 21 will be a long-term support (LTS) release from most vendors. For a complete list of the JEPs integrated since the previous LTS release, JDK 17, please see here.

Schedule

2023/06/08	Rampdown Phase One (fork from main line)
2023/07/20	Rampdown Phase Two
2023/08/10	Initial Release Candidate
2023/08/24	Final Release Candidate
2023/09/19	General Availability

Last update: 2023/9/19 10:53 UTC

- Virtual Threads
- Structured Concurrency
- Scoped Values

구조적 동시성 : **StructuredTaskScope (Second Preview)**

Module java.base
Package java.util.concurrent

Class StructuredTaskScope<T>

java.lang.Object
    java.util.concurrent.StructuredTaskScope<T>

Type Parameters:

T - the result type of tasks executed in the task scope

All Implemented Interfaces:

AutoCloseable

Direct Known Subclasses:

StructuredTaskScope.ShutdownOnFailure[PREVIEW], StructuredTaskScope.ShutdownOnSuccess[PREVIEW]

```
public class StructuredTaskScope<T>
extends Object
implements AutoCloseable
```

StructuredTaskScope is a preview API of the Java platform.
Programs can only use StructuredTaskScope when preview features are enabled.
Preview features may be removed in a future release, or upgraded to permanent features of the Java platform.

A basic API for *structured concurrency*. StructuredTaskScope supports cases where a task splits into several concurrent subtasks, and where the subtasks must complete before the main task continues. A StructuredTaskScope can be used to ensure that the lifetime of a concurrent operation is confined by a *syntax block*, just like that of a sequential operation in structured programming.

구조적 동시성 : 동시 작업의 결과가 필요하지 않은 경우 (Java)

```java
public class StructuredTaskScopeExample {

    public static void main(String[] args) {
        var example = new StructuredTaskScopeExample();
        example.sendMessage();
    }

    public void sendMessage() {
        try (var scope = new StructuredTaskScope.ShutdownOnFailure()) {
            scope.fork((Callable<Void>) () -> {
                sendEmail();
                return null;
            });

            scope.fork((Callable<Void>) () -> {
                sendSms();
                return null;
            });

            scope.join();
            scope.throwIfFailed((e) -> new RuntimeException("Failed to send"));

        } catch (InterruptedException e) {
            throw new RuntimeException(e);
        }
    }

    public static void sendEmail() {
        System.out.println("이메일 발송 완료");
    }

    public static void sendSms() {
        System.out.println("SMS 발송 완료");
    }
}
```

구조적 동시성 : 동시 작업의 결과가 필요하지 않은 경우 (Kotlin)

```java
public class StructuredTaskScopeExample {

    public static void main(String[] args) {
        var example = new StructuredTaskScopeExample();
        example.sendMessage();
    }

    public void sendMessage() {
        try (var scope = new StructuredTaskScope.ShutdownOnFailure()) {
            scope.fork((Callable<Void>) () -> {
                sendEmail();
                return null;
            });

            scope.fork((Callable<Void>) () -> {
                sendSms();
                return null;
            });

            scope.join();
            scope.throwIfFailed((e) -> new RuntimeException("Failed to send"));

        } catch (InterruptedException e) {
            throw new RuntimeException(e);
        }
    }

    public static void sendEmail() {
        System.out.println("이메일 발송 완료");
    }

    public static void sendSms() {
        System.out.println("SMS 발송 완료");
    }
}
```

```kotlin
fun main() {
    sendMessage()
}

fun sendMessage() {
    StructuredTaskScope.ShutdownOnFailure().use { scope ->
        scope.fork { sendEmail() }
        scope.fork { sendSms() }

        scope.join()
        scope.throwIfFailed { e ->
            RuntimeException("Failed to send ", e)
        }
    }
}


fun sendEmail() {
    // .. 이메일 발송 구현
    println("이메일 발송 완료")
}

fun sendSms() {
    // .. SMS 발송 구현
    println("SMS 발송 완료")
}
```

구조적 동시성 : 동시 작업의 결과가 필요하지 않은 경우 (Coroutines)

```
fun main() {
    sendMessage()
}

fun sendMessage() {
    StructuredTaskScope.ShutdownOnFailure().use { scope ->
        scope.fork { sendEmail() }
        scope.fork { sendSms() }

        scope.join()
        scope.throwIfFailed { e ->
            RuntimeException("Failed to send ", e)
        }
    }
}


fun sendEmail() {
    // .. 이메일 발송 구현
    println("이메일 발송 완료")
}

fun sendSms() {
    // .. SMS 발송 구현
    println("SMS 발송 완료")
}
```

```
이메일 발송 완료
SMS 발송 완료
```

```
suspend fun sendMessage() = coroutineScope {
    launch { sendEmail() }

    launch { sendSms() }
}
```

```
fun sendMessage() = runBlocking {
    launch { sendEmail() }

    launch { sendSms() }
}
```

구조적 동시성 : 동시 작업의 결과를 만들어내야 하는 경우 (Java)

```java
public class StructuredTaskScopeExample2 {

    public static void main(String[] args) {
        var example = new StructuredTaskScopeExample2();
        example.getUsers();
    }

    public List<UserResponse> getUsers() {
        try (var scope = new StructuredTaskScope.ShutdownOnFailure()) {

            var fork1 = scope.fork(() -> callApi(1L));

            var fork2 = scope.fork(() -> callApi(2L));

            scope.join();
            scope.throwIfFailed((e) -> new RuntimeException("Failed to get user"));

            return List.of(fork1.get(), fork2.get());
        } catch (InterruptedException e) {
            throw new RuntimeException(e);
        }
    }

}
```

구조적 동시성 : 동시 작업의 결과를 만들어내야 하는 경우 (Coroutines)

```java
public List<UserResponse> getUsers() {
    try (var scope = new StructuredTaskScope.ShutdownOnFailure()) {

        var fork1 = scope.fork(() -> callApi(1L));

        var fork2 = scope.fork(() -> callApi(2L));

        scope.join();
        scope.throwIfFailed((e) -> new RuntimeException("Failed to get user"));

        return List.of(fork1.get(), fork2.get());
    } catch (InterruptedException e) {
        throw new RuntimeException(e);
    }
}
```

```kotlin
suspend fun getUsers(): List<UserResponse> = coroutineScope {
    val user1 = async { callApi(1) }
    val user2 = async { callApi(2) }

    listOf(user1.await(), user2.await())
}
```

```kotlin
fun getUsers() = runBlocking {
    val user1 = async { callApi(1) }
    val user2 = async { callApi(2) }

    listOf(user1.await(), user2.await())
}
```

ScopedValue : 스레드 간에 데이터를 저장하고 공유하는 방법 (java)

```java
public void sendMessage() {
    ScopedValue<String> requestId = ScopedValue.newInstance();

    ScopedValue.where(requestId, UUID.randomUUID().toString()).run(() -> {
        logger.info("[requestId: {}] call api", requestId.get() );

        try (var scope = new StructuredTaskScope.ShutdownOnFailure()) {
            scope.fork((Callable<Void>) () -> {
                logger.info("[requestId: {}] sendEmail", requestId.get() );
                sendEmail();
                return null;
            });

            scope.fork((Callable<Void>) () -> {
                logger.info("[requestId: {}] sendSms", requestId.get());
                sendSms();
                return null;
            });

            scope.join();
            scope.throwIfFailed((e) -> new RuntimeException("Failed to send"));

        } catch (InterruptedException e) {
            throw new RuntimeException(e);
        }

    });
}
```

ScopedValue : 스레드 간에 데이터를 저장하고 공유하는 방법 (coroutines)

```java
public void sendMessage() {
    ScopedValue<String> requestId = ScopedValue.newInstance();

    ScopedValue.where(requestId, UUID.randomUUID().toString()).run(() -> {
        logger.info("[requestId: {}] call api", requestId.get() );

        try (var scope = new StructuredTaskScope.ShutdownOnFailure()) {
            scope.fork((Callable<Void>) () -> {
                logger.info("[requestId: {}] sendEmail", requestId.get() );
                sendEmail();
                return null;
            });

            scope.fork((Callable<Void>) () -> {
                logger.info("[requestId: {}] sendSms", requestId.get());
                sendSms();
                return null;
            });

            scope.join();
            scope.throwIfFailed((e) -> new RuntimeException("Failed to send"));

        } catch (InterruptedException e) {
            throw new RuntimeException(e);
        }

    });
}
```

```kotlin
val logger = KotlinLogging.logger { }

fun main() = runBlocking<Unit> {

    MDC.put("requestId", UUID.randomUUID().toString())
    logger.info { "[requestId: ${MDC.get("requestId")}] call api" }

    withContext(MDCContext()) {
        launch {
            logger.info { "[requestId: ${MDC.get("requestId")}] sendEmail" }
            sendEmail()
        }

        launch {
            logger.info { "[requestId: ${MDC.get("requestId")}] sendEmail" }
            sendSms()
        }
    }
}
```

버추얼 스레드를 코루틴 디스패쳐로 변환

```
val Dispatchers.VIRTUAL_THREAD: ExecutorCoroutineDispatcher
    get() = Executors.newVirtualThreadPerTaskExecutor().asCoroutineDispatcher()

suspend fun sendMessage() = withContext(Dispatchers.VIRTUAL_THREAD) {

    launch { sendEmail() }

    launch { sendSms() }
}
```

코루틴으로 작성된 라이브러리를 그대로 사용 (AWS SDK for Kotlin)

```kotlin
suspend fun getObjectBytes(bucketName: String, keyName: String, path: String) {
    val request = GetObjectRequest {
        key = keyName
        bucket = bucketName
    }

    S3Client { region = "us-east-1" }.use { s3 ->
        s3.getObject(request) { resp ->
            val myFile = File(path)
            resp.body?.writeToFile(myFile)
            println("Successfully read $keyName from $bucketName")
        }
    }
}
```

코루틴으로 작성된 라이브러리를 그대로 사용 (AWS SDK for Kotlin)

```kotlin
fun getObjectBytes(bucketName: String, keyName: String, path: String) = runBlocking {
    val request = GetObjectRequest {
        key = keyName
        bucket = bucketName
    }

    S3Client { region = "us-east-1" }.use { s3 ->
        s3.getObject(request) { resp ->
            val myFile = File(path)
            resp.body?.writeToFile(myFile)
            println("Successfully read $keyName from $bucketName")
        }
    }
}
```

SSE로 실시간 주식 시세 제공

```
data:{"symbol":"AAPL","price":156.7}

data:{"symbol":"AAPL","price":167.35}

data:{"symbol":"AAPL","price":155.27}

data:{"symbol":"AAPL","price":150.03}

data:{"symbol":"AAPL","price":157.56}

data:{"symbol":"AAPL","price":168.15}
```

SSE로 시세를 제공하기 위해 코루틴의 Flow를 사용해 데이터 스트리밍 API 구현

```kotlin
@GetMapping("/price/{symbol}/stream", produces = [MediaType.TEXT_EVENT_STREAM_VALUE])
fun findPrice(
    @PathVariable symbol: String,
): Flow<StockPriceResponse> {
    return flow {
        while (true) {
            val response = stockPriceClient.findPrice(symbol)
            emit(response)
            delay(500)
        }
    }
}
```

Network 블로킹 IO 발생

블로킹 IO가 발생해도 문제 없음

```kotlin
@GetMapping("/price/{symbol}/stream", produces = [MediaType.TEXT_EVENT_STREAM_VALUE])
fun findPrice(
    @PathVariable symbol: String,
): Flow<StockPriceResponse> {
    return flow {
        while (true) {
            val response = stockPriceClient.findPrice(symbol)
            emit(response)
            delay(500)
        }
    }.flowOn(Dispatchers.VIRTUAL_THREAD)
}
```

버추얼 스레드 디스패쳐에서 수행

블로킹 IO가 발생해도 문제 없음

```kotlin
@GetMapping("/price/{symbol}/stream", produces = [MediaType.TEXT_EVENT_STREAM_VALUE])
fun findPrice(
    @PathVariable symbol: String,
): Flow<StockPriceResponse> {
    return flow {
        while (true) {
            val response = stockPriceClient.findPrice(symbol)
            emit(response)
            delay(500)
        }
    }
}
```

```yaml
spring:
  threads:
    virtual:
      enabled: true
```

정리하면

- 코루틴의 단점인 블로킹 상황의 성능 하락 이슈를 버추얼 스레드가 해결
- 코루틴과 버추얼 스레드의 상호 운용을 통해 코루틴의 잘 만들어진 구조적 동시성과 코루틴 컨텍스트를 그대로 활용할 수 있다
- 버추얼 스레드에서 지원하지 않는 기능(스트리밍, 백프레셔 등)은 코루틴의 고급 라이브러리를 활용해 구현할 수 있다
- 기존에 코루틴을 사용했던 애플리케이션에서 JDK 버전을 올려 버추얼 스레드를 혼합 사용 가능

Q & A

참고 자료

- https://d2.naver.com/helloworld/1203723
- https://youtu.be/bOLChQ3fFQo?si=1GoiWcUnjPekf7pv
- https://docs.oracle.com/en/java/javase/21/core/virtual-threads.html#GUID-BEC799E0-00E9-4386-B220-8839EA6B4F5C
- https://www.youtube.com/watch?v=vQP6Rs-ywlQ
- https://www.youtube.com/watch?v=1qezCNVWpHc
- https://www.baeldung.com/java-20-scoped-values
- https://quarkus.io/blog/virtual-thread-1/