리액티브 프로그래밍
& Spring WebFlux 소개

OS2-1 최지혜

리액티브 프로그래밍 패러다임의 필요성

- 상황의 변화
  - 빅데이터: 페타바이트 단위
  - 다양한 환경: 모바일 디바이스 ~ 클라우드 기반 클러스터
  - 사용패턴
    - 항상 서비스를 사용하길 원함
    - 밀리초 단위 응답시간 기대

- 목표: 리액티브 시스템 (혹은 리액티브 컴포넌트)
  - 고장, 정전 같은 상황에 대처
  - 다양한 네트워크에서 메시지를 교환하고 전달
  - 무거운 작업을 하고 있는 상황에서도 가용성 제공

- 문제점 : 이전의 소프트웨어 아키텍처로는 이런 요구사항을 만족시킬 수 없음

- **해결방법 : 리액티브 프로그래밍**:
  - 다양한 시스템과 소스에서 들어오는 데이터 스트림을 비동기적으로 처리하고 합치기

리액티브 프로그래밍
: 리액티브 스트림을 다루는 프로그래밍

- 리액티브 스트림: 잠재적으로 무한의 비동기 데이터를 순서대로 논블럭킹으로 역압력으로 처리하는 기술
  - 비동기작업 결과 관련객체(퓨처, 콜백 등)를 계속해서 받음
  - 받아서 적용할 작업(=반응)이 있음
  - 그리고 이 작업의 결과도 연속해서 사용자 또는 다른 서비스로 계속해서 전달/응답
  - Java 8 CompletableFuture와 다른점 : 비동기 작업 한 번 결과 받으면 끝

- **그러려면 데이터 주는 객체(=발행자)와 받는 객체(=구독자) 사이 지속되는 연결(구독)**이 필요 (=Pub-Sub)
  - 이 연결을 통해
    - 발행자는 구독자에게 메시지(이벤트)를 전송하고
    - 구독자는 처리가능 속도를 발행자에게 알릴 수 있음 (=역압력)

리액티브 스트림 API

- 인터페이스 : Flow(Java 9)
  - Publisher, Subscriber, Subscription, Processor
- 구현체(리액티브 프레임워크): RxJava, Akka, vert.x Reactor
  - Flow 추상메서드 구현 + a
  - 동작
    - 스레드를 퓨처, 액터, 일련의 콜백을 발생시키는 이벤트 루프 등과 공유
    - 처리할 이벤트를 변환하고 관리함
    - 이러한 기술은 동시성 프로그래밍의 추상 수준을 높여 프로그래머가 멀티스레드를 안정적으로 사용할 수 있도록 함
      - 스레드 풀에서 블록킹 동작 수행
      - 블로킹 작업을 cpu 작업과 i/o 작업 분리해서 스레드 풀 크기 설정

리액티브 프로그래밍에 유용한 기법

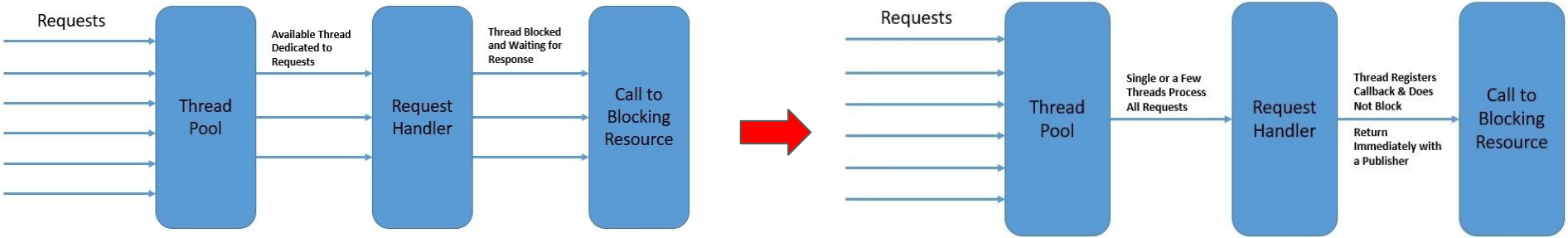
- 리액티브 프레임워크의 오퍼레이터는 순수함수로 이루어져있음
- 순수함수를 이용하는 함수형 프로그래밍
  - 가변공유변수를 사용하는 것을 지양 -> 동시접근, 교착상태 등 문제 X
  - 멀티스레드 프로그래밍에 적합
- 선언형 프로그래밍
  - 이런 API 를 이용하여 선언형 프로그래밍을 할 수 있음

이전의 서버 소프트웨어 아키텍처: 요청당 스레드 모델

특징
- 웹 서버에 대한 사용자 요청들은 서로 다른 연결, 스레드에 의해 처리
- 멀티코어 플랫폼에서 동시성 제공 가능

문제점
- 동시 요청이 너무 많아지면
  - 요청 수 만큼 연결할 수 없음
    - C10k Problem
  - 요청 수 만큼 스레드를 만들 수 없음
    - 스레드가 사용하는 스택 out of memory, 스레드 간 컨텍스트 스위치 오버헤드
- 그래서 가용 스레드 수보다 많은 요청을 받으면 일부 요청은 처리되기까지 오래 기다려야 함

결론: **더 적은 스레드로 더 많은 연결/요청을 처리할 수 있는 방법**이 필요



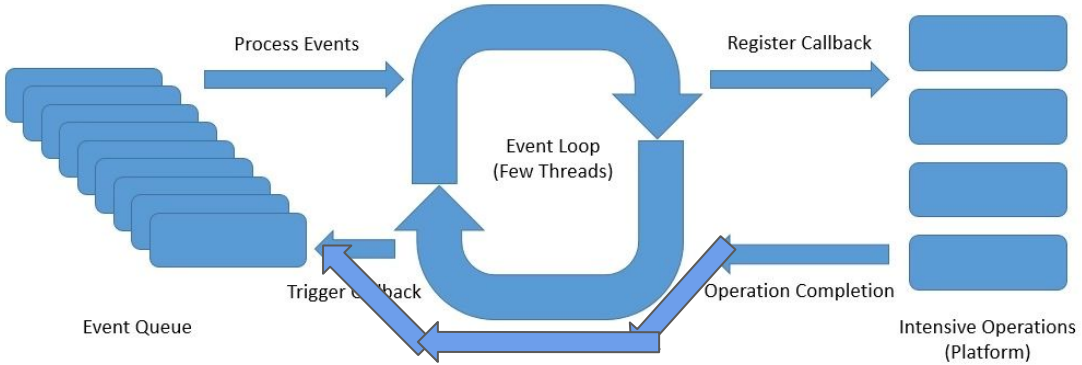
리액티브 프로그래밍을 위한 새로운 서버 스레드 모델 : 이벤트 루프

특징

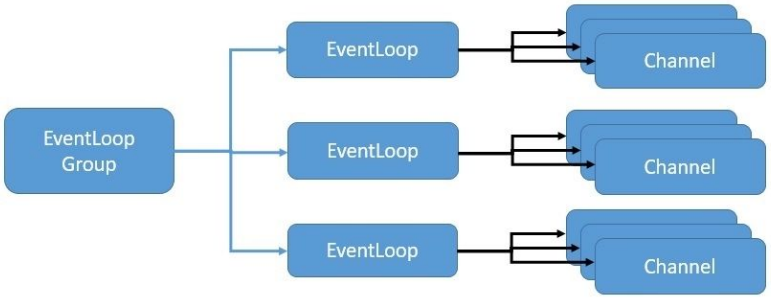
- 이벤트 루프는 단일 스레드에서 실행됨
- Node.js, **Netty** 및 Ngnix를 포함한 여러 플랫폼에서 구현

동작

1. (시간이 오래 걸리는 작업을 하는) 플랫폼에 콜백(이 작업이 완료되면 할 일)을 등록
   a. 데이터베이스 또는 외부 서비스 호출 등
2. 플랫폼은 작업이 완료되면 큐에 콜백을 넣음.
3. 이벤트 루프는 큐에서 순차적으로 콜백을 가져와서 콜스택에 넣어 실행함
4. 결과를 원래 호출자에게 다시 전송함.



Netty



: 역압력을 지원하는 자바 논블럭킹 I/O 라이브러

- Java nio를 추상화함
  - Java nio : 셀렉터클래스를 이용하여 한 스레드에 동시에 여러 연결을 비동기로 처리할 수 있음
- 코어 개념
  - CHANNEL
    - I/O 작업을 위한 커넥션을 의미(소켓의 추상화)
  - CHANNEL FUTURE
    - JAVA5 FUTURE 문제점 개선한 Netty자체 개념
  - 이벤트루프(=리액터), 핸들러
    - EventLoopGroup은 지속적으로 실행되어야 하는 하나 이상의 EventLoop을 관리합니다.
    - 따라서 사용 가능한 코어 수보다 더 많은 이벤트 루프를 생성하지 않는 것이 좋습니다.
    - EventLoop Group은 새로 생성된 각 채널에 EventLoop을 추가로 할당합니다.
  - 인코더, 디코더
    - 메시지를 바이트 시퀀스 <-> 자바 객체

Reactor netty

ReactorNety는 Spring Boot WebFlux 스타터의 기본 내장 서버

스레드의 수와 특성은 우리가 선택하는 실제 반응형 스트림 API 런타임에 따라 달라진다.

서버용 일반 스레드와는 별도로 Netty는 요청 처리를 위한 작업자 스레드를 생성합니다.

Thread Name	State	Type
server	WAITING	Normal
reactor-http-nio-1	RUNNABLE	Daemon
reactor-http-nio-2	RUNNABLE	Daemon
reactor-http-nio-3	RUNNABLE	Daemon

Threads in WebFlux Application on Netty

Spring WebFlux

웹플럭스(WebFlux)는 스프링의 리액티브 스택 웹 프레임워크로, 버전 5.0에 추가되었다.

완전히 논블로킹이며 반응형 스트림 배압을 지원하며 Netty, Underow 및 Servlet 컨테이너와 같은 서버에서 실행됩니다.

Spring의 전통적인 웹 프레임워크를 대체하지는 않습니다.

1. Spring WebFlux는 함수형 라우팅으로 기존의 주석 기반 프로그래밍 모델을 확장한다.
2. 기본 HTTP 런타임을 반응형 스트림 API에 적응시켜 런타임을 상호 운용 가능하게 만든다.
3. 다양한 리액티브 런타임(실행환경)을 지원
    a. Tomcat, **Reactor, Netty** 또는 Underow와 같은 Servlet 3.1+ 컨테이너를 포함
4. HTTP 요청을 위한 리액티브 논블록킹 클라이언트인 WebClient를 포함한다.

더 알아봐야 할 것들

R2DBC(Reactive Relational DB Connection)

Java 8 이후 API

참고

- Baeldung
- Spring Web Flux 공식 문서
- Project Reactor 공식 문서
- 모던 자바 인 액션
- 라인 블로그
  https://engineering.linecorp.com/ko/blog/reactive-streams-with-armeria-1/
- Maria DB 공식 문서
  https://mariadb.com/resources/blog/reactive-programming-with-spring-data-r2dbc-on-mariadb-skysql/
- 얄팍한 코딩사전 리액티브 프로그래밍
  https://www.youtube.com/watch?v=KDiE5qQ3bZI
- 이벤트루프 https://www.youtube.com/watch?v=8aGhZQkoFbQ
- netty https://www.youtube.com/watch?v=DKJ0w30M0vg

병렬성/동시성을 제공하는 Java API - 1

Java 8 Stream API 의 parallel(), parallelStream()

- 적절한 케이스 : I/O 가 포함되지 않은 계산 중심의 동작
    - 스트림의 lazy 함 때문에 I/O 작업이 실제로 언제 처리될지 예측하기 어려움

- (참고) 내부구현 : Fork/Join 프레임워크
    - 재귀적으로(분할/정복) 동작
        - 더 나눌 수 있다면
            - 작은 태스크로 나누어 스레드 풀(ForkJoinPool)의 스레드에 제출, 해당 스레드에서 실행시키기
            - invokeAll() / fork()
            - 스레드풀: ExecutorService 구현체
        - 더 나눌 수 없다면 실행시키기
        - 결과 합치기
    - 실행하려면
        - RecursiveAction (void) 또는 RecursiveTask (반환값 있음) 을 상속하는 클래스를 구현해야함
        - 필수 override 메서드 : compute()

병렬성/동시성을 제공하는 Java API - 2

Java 5 Future, ExecutorService

```
ExecutorService es = Executors.newFixedThreadPool(1);
Future<Integer> f = es.submit(() -> asyncMethod());
System.out.println(f.get());
```

(참고) ExecutorService : 스레드 풀과 태스크를 할당하기 위한 API

- ExecutorService newFixedThreadPool (int num)
  - 동작
    - 워커 스레드라 불리는 nThreads 를 포함하는 ExecutorService를 만들고 이들을 스레드 풀에 저장함
    - 스레드 풀에서 사용하지 않은 스레드로 제출된 태스크를 먼저 온 순서대로 실행
    - 태스크들의 실행이 종료되면 스레드를 풀로 반환
  - 장점
    - 하드웨어에 맞는 수의 태스크를 유지함
    - 동시에 수천 개의 태스크를 스레드 풀에 아무 오버헤드 없이 제출할 수 있음

+) 멀티코어 환경에서 통상적인 JVM 구현에 따르면 새 스레드는 서로 다른 코어에 할당한다고 함

Java 8 CompletableFuture

- Future 조합 기능
- ExecutorService 관련 추상화 API 제공
- 외부에서 Task 종료 가능
- 적절한 케이스: I/O를 기다리는 작업을 병렬로 실행
- 병렬 스트림과 다른점: 스레드풀 커스텀 가능

- 그런데
  - 데이터를 한 번 받고 끝이 아니라 **(여러 번 나온다면) 계속해서** 받고 싶고
  - 받아서 적용할 일(=반응)도 정해놓고 싶음
  - 그리고 이 결과도 연속해서 사용자 또는 다른 서비스로 전달/응답하고 싶음
- 그러려면 데이터 주는 객체(=발행자)와 받는 객체(=구독자) 사이 지속되는 연결이 있어야 함. 또한 이 연결을 통해
  - 발행자는 구독자에게 메시지(이벤트)를 전송하고
  - 구독자는 처리가능 속도를 발행자에게 알림 (=역압력)

-> 리액티브 스트림 API 가 필요

스트림, 리액티브 스트림

스트림

- 연속적인 데이터

Java 8 Stream API

- 스트림 처리
- 반복을 내부구현으로 추상화
- 선언형 -> 읽기쉬움
- 파이프라인 -> 조립 쉬움, 교체쉬움

+) 특히 병렬 프로그래밍에서는 서로 다른 스레드에서 접근/변경 가능한 변수 사용을 지양하는 것이 좋다
(함수형 프로그래밍)

리액티브 스트림

: 논블록킹, 역압력 기능의 비동기 스트림

예) 사용자 요청, 네트워크 응답, 리액티브 DB 응답

Java 9 Flow API

- 리액티브 스트림 처리를 다루는 API 인터페이스
- 구현체 (라이브러리): RxJava 2.0, Akka, **Reactor** 등
- 인터페이스
  - Publisher
  - Subscriber
  - Subscription
  - Processor

CompletableFuture API

팩토리 메서드 supplyAsync

- 두 번째 인수로 **executor** 받았다면 executor의 스레드풀에서 가져오고
  - 아니면 스레드 만들고
- 스레드에 future.complete() 포함한 잡 넘겨주고
- 스레드 start
- future 리턴

executor 커스텀

- 스레드풀의 스레드 수 최적값 찾아서 그만큼 만들 수 있음
- 혹은 필요한 만큼만 쓸 수도 있음
- 가용코어수 * 0/1사이값 갖는 CPU 활용비율 * (1+대기시간/계산시간)
- 가용코어수 : Runtime.getRuntime().avaiableProcessors()