Operators

1. Sequence 생성을 위한 Operator

defer( )

선언한 시점에 데이터를 emit하는 것이 아니라 구독하는 시점에 데이터를 emit하는 Flux / Mono를 생성. 데이터 emit을 지연시키기 때문에 꼭 필요한 시점에 데이터를 emit하여

불필요한 프로세스를 줄일 수 있다.

just( )와 차이

- just( )의 경우 Hot Publisher 형태로 동작하므로 Subscriber의 구독 여부와는 상관 없이 데이터를 emit
- 이후 구독이 발생하면 emit된 데이터를 replay를 통해 Subscriber에게 전달

주의할 점

Java 복사 캡션

```java
@Slf4j
public class DeferOperatorExample {

    public static void main(String[] args) throws InterruptedException {

        Mono.just("Hello")
                .delayElement(Duration.ofSeconds(3))
                .switchIfEmpty(say())   // 메서드가 불필요하게 호출 됨.
//                .switchIfEmpty(Mono.defer(() -> Mono.just("Hi"))) // 이러한 문제를 해결하기 위해 defer로 감싸기
                .subscribe(data -> log.info("# onNext: {}", data));


        Thread.sleep(3500L);
    }

    private static Mono<String> say() {
        log.info("# Say hello.");
        return Mono.just("Hi");
    }
}
```

```
> Task :org.study.operation.DeferOperatorExample.main()
21:10:28.113 [main] INFO org.study.operation.DeferOperatorExample -- # Say hello.
21:10:31.124 [parallel-1] INFO org.study.operation.DeferOperatorExample -- # onNext: Hello

BUILD SUCCESSFUL in 3s
2 actionable tasks: 2 executed
오후 9:10:31: 실행이 완료되었습니다 ':org.study.operation.DeferOperatorExample.main()'.
```

자바 프로그래밍: evaluation

이러한 문제는 자바 언어의 특징을 살펴보아야 한다. 자바의 경우 eager / lazy evaluation이라는 개념이 존재한다.

lazy evaluaiton: 논리식의 OR 조건일 때 선행 조건이 참일 경우 후행 조건을 판단하지 않음

eager evaluation: 메서드에 들어오는 매개변수의 값에 따라 메서드의 실행 흐름을 결정할 수 있음

eager evaluation을 직관적으로 코드를 통해 표현해보면 다음과 같은 흐름을 파악할 수 있다.

```
public String routine(boolean bool) {
  return bool ? "true" : "false";

public String sub_routine(boolean bool) {
  return bool;
}

// call:  routine(sub_routine(true);
// 호출 순서: sub_routine -> routine
```

매개변수의 형태로 특정 메서드의 반환 값을 파라미터로 받아 메서드를 호출할 때, 파라미터로 들어가는 매개변수의 값을 평가하기 위해 `sub_routine` 메서드가 우선 호출된다.

다시 표현하면, `routine` 메서드는 들어오는 매개변수의 값에 따라 매개변수의 실행흐름을 결정할 수 있다는 이야기이다. 명령형 자바 프로그래밍에서는 이와 같은 특징을 가지고 있다.

그렇다면 코드에서 표현되어지는 lazy evaluation은 어떻게 작성되어야 할까?

```
public String routine(Supplier<Boolean> f) {
  return f.get() ? "true" : "false";

public String sub_routine(boolean bool) {
  return bool;
}

// call: routine(() -> subroutine(true));
// 호출 순서: routine -> sub_routine
```

실행 순서가 바뀐 것을 확인할 수 있다. 이는 람다식이 평가되는 시점에 `sub_routine`을 호출하게 되므로 매개변수의 값을 실제 필요한 시점에 맞춰 지연시킬 수 있다는 장점을 가지고 있다.

Reactor Operator의 swithIfEmpty( ) 역시 eager evaluation 특징을 가지고 있는 메서드이다.

switchIfEmpty( ) 에서 불필요한 메서드 호출을 해결하자

상황

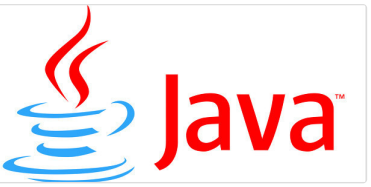
- 파라미터로 전달되어 반환 값을 얻기 위해 실행되는 메서드의 호출이 불필요하거나 실행 비용이 높은 경우
- Nono empty가 반환되지 않는 경우

이러한 상황에서는 lazy 전략을 통해 실행 시점을 뒤로 미뤄보는 방법을 생각해볼 수 있다.

→ Mono.defer( ), Mono.fromSupplier( )

참고자료

- Lazy evaluation



generate( ) / create( )

프로그래밍 방식으로 Signal 이벤트를 발생시킬 경우 사용되는 Operator들이다.

generate( ): 동기적으로 한 번에 한 건씩 emit

create( ): 한 번에 여러 건의 데이터를 비동기적으로 emit

publisher가 데이터를 emit하는 방식

pull 방식: Subscriber가 request( ) 메서드를 통해 요청을 보내면 Publisher가 해당 요청 개수만큼의 데이터를 emit하는 방법

push 방식: Subscriber의 요청과 상관없이 비동기적으로 데이터를 emit하는 방식

이렇듯 generate와 create 고유한 특징에 맞춰 모두 유연하게 상황에 맞춰 pub 과정을 재작성할 수 있으며, 대표적인 예로 create( ) a게 서드의 경우 pull 방식의 구현과 push 방식의 구현 방법이 존재하는 등

Sink를 재정의하는데 크게 제약을 받지 않는다.

2. Sequence 필터링 Operator

filter( ) / filterWhen( )

일치하는 조건에 맞춰 DownStream으로 데이터를 emit하는 Operator이다. filterWhen( )의 경우 비동기적으로 실행되어 필터링을 수행할 수 있다. 대표적으로 Inner Sequence를 이용할 수 있는데 다음의 코드를 살펴보자.

```
@Slf4j
public class FilterOperatorExample {

    static Map<String, Tuple2<String, Integer>> vaccineMap = Map.of(
            "A vaccine", Tuples.of("A vaccine", 3_000_000),
            "B vaccine", Tuples.of("B vaccine", 2_000_000),
            "C vaccine", Tuples.of("C vaccine", 4_000_000),
            "D vaccine", Tuples.of("C vaccine", 1_000_000),
            "E vaccine", Tuples.of("C vaccine", 7_000_000)
    );

    public static void main(String[] args) throws InterruptedException {

        Flux
                .fromIterable(Flux.just("A vaccine", "B vaccine", "C vaccine").toIterable())
                .filterWhen(vaccine -> Mono.just(vaccineMap.get(vaccine).getT2() >= 3_000_000)
                        .publishOn(Schedulers.parallel())
                )
                .subscribe(data -> log.info("# onNext: {}", data));

        Thread.sleep(1000);
    }
}
```

이처럼 Inner Sequnece에서 emit된 데이터를 Subscriber가 이용할 경우 Reactor의 Scheduler 기능을 이용해 filterWhen 메서드에서 비동기적으로 작업을 처리할 수 있다.

takeUntil( ) / takeWhile( )

take( ) Operator의 경우 Upstream에서 emit되는 데이터 중 특정 개수만큼 DownStream으로 데이터를 emit 한다.

추가적으로 살펴볼만한 Operator에는

takeUntil의 경우 파라미터로 입력받은 람다 표현식이 true임을 만족할 때까지 DownStream으로 데이터를 emit하게 되고,

takeWhile에서는 파라미터로 입력 받은 람다 표현식이 true임을 만족하는 동안 DownStream으로 데이터를 emit한다는 Operator이다.

차이점

takeUntil( ) : emit된 데이터는 Predicate를 평가할 때 사용한 데이터가 포함

takeWhile( ): emit된 데이터는 Predicate를 평가할 때 사용한 데이터가 포함되지 않음

3. Sequence 변환 Operator

concat( ) / merge( )

concat( ): 파라미터로 입력되는 Publisher의 Sequence를 연결해 데이터를 순차적으로 emit.

먼저 입력된 Publisher의 Sequence가 종료될 때까지 나머지 Sequence들은 subscirbe 되지 않고 대기

merge( ): Seqeunce 들에서 emit 되는 데이터를 인터리빙 방식(=. emit된 시간 순서대로)으로 병합

모든 Publisher의 Sequence가 즉시 subscribe

4. 다수의 Subscriber에게 Flux를 멀티캐스팅하기 위한 Operator

Subscriber가 구독할 경우 UpStream에서 emit된 데이터가 구독 중인 모든 Subscriber에게 멀티캐스팅 될 수 있도록 기능을 제공하는 메서드가 존재.

이러한 메서드는 기존의 Reactor Sequence의 Cold Sequence 특징을 Hot Sequence로 동작할 수 있도록 변환한다. publish( ) 메서드와 같이 사용하는 아래의 메서드를 살펴보자.

autoConnect( )

publish( )를 사용하게 되면 구독이 발생하더라도 connect( )를 직접 호출하기 전까지 데이터를 emit 하지 않는다.

autoConnect( )의 경우 지정하는 숫자 만큼의 구독이 발생하는 시점에 UpStream 소스에 자동으로 연결되므로 별도의 connect( ) 호출이 필요하지 않는다.

refCount( )

지정된 숫자만큼 구독이 발생하는 시점에 UpStream 소스에 연결된다는 특징은 autoConnect( )와 비슷하지만, 모든 구독이 취소되거나 UpStream의 데이터 emit이 종료될 경우 UpStream과 연결이 해제된다.

주로 무한 스트림 상황에서 모든 구독이 취소될 경우 연결을 해제하는데 사용해볼 수 있다.

차이점

refCount( )의 경우 모든 구독이 취소되어 연결이 해제된 이후, 다른 Subscriber가 구독을 이어가는 경우라면 UpStream에서 emit하는 데이터가 처음부터 replay 되는 특징을 가지고 있다.

반면, autoConnect( )의 경우 UpStream 연결을 해제하지 않으므로 UpStream의 데이터를 이어서 전달받을 수 있다.