리액티브 스트림과 Operation 체인

1. Sink

리액티브 스트림즈의 Signal을 프로그래밍 방식으로 푸시할 수 있는 방법에는 다음과 같은 종류가 있다.

전통적인 방식 : generate( ), create( ) 등을 사용하는 것. 싱글 스레드 기반에서 Signal을 전송하는 데 사용

Sink를 이용하는 방식 : 멀티 스레드 방식으로 Signal을 전송해도 스레드 안전성을 보장

싱글스레드가 아닌 여러 개의 스레드에서 각각의 전혀 다른 작업들을 처리한 후 처리 결과를 반환해야 하는 상황에서 안전하게 사용할 수 있는 방식

그렇다면 Sink의 종류에는 어떤 것이 있을까? 크게 emit하는 데이터의 건수에 따라 One과 Many로 구분할 수 있다.

Sinks.One: 한 건의 데이터를 전송하는 방법을 정의해둔 기능 명세

- Mono 방식으로 Subscriber가 데이터를 소비할 수 있도록 Sinks의 스펙 또는 사양이라고 볼 수 있다.
- EmitFailureHandler 객체를 통해 emit 도중 에러가 발생했을 때 재시도를 하지 않고 즉시 실패 처리

  → 결과적으로 스레드 안전성을 보장하기 위함
- 처음 emit한 데이터는 정상적으로 emit되지만 나머지 데이터들은 Drop

Sinks.Many : 여러 건의 데이터를 다양한 방식으로 전송하는 방법을 정의해둔 기능 명세

emit한 데이터를 어떤 subscriber가 처리할지 ManySpec을 통해 설정할 수 있다.

- UnicastSpec : 단 하나의 Subscriber에게만 데이터를 emit

  ※ 2번째 Subscriber가 데이터를 받으려고 하는 경우 IllegalStateException 발생
- MulticastSpec: 여러 개의 Subscriber에게 데이터를 emit
- MulticastReplaySpec: emit된 데이터를 다시 replay해서 구독 전에 이미 emit된 데이터라도 Subscriber가 전달받을 수 있도록 함
  - limit: 가장 나중에 emit된 데이터부터 Subscriber에게 전달하는 기능

Sinks가 Publisher로 동작하는 경우 기본적으로 Hot Publisher로 동작. 특히 `onBackpressureBuffer()` 메서드의 경우,
Warm up 특징을 가지는 Hot Sequence로 동작하며 첫 구독이 발생한 시점에 DownStream으로 데이터가 전달한다.

Reactor의 Hot Sequence 분류

Reactor에서는 Warm up, Hot 방식으로 세분화되어 구분할 수 있고 아래의 특징을 가진다.

Warm up : 최초 구독이 발생하기 전까지 데이터를 emit 하지 않음

Hot : 구독 여부와 상관 없이 데이터를 emit

2. Scheduler

운영체제 레벨의 Scheduler는 실행되는 프로그램 프로세스를 선택하고 실행하는 과정에서 프로세스의 라이프 사이클을 관리한다.

Reactor에서 이야기하는 Scheduler는 Reactor Sequence에서 사용되는 스레드를 관리하게 된다. 즉, 비동기 프로그래밍에서 스레드를 관리하는 것을 의미한다.

스레드의 구분

- 물리적 스레드 : 하드웨어와 관련된 스레드 (병렬성)
- 논리적 스레드 : 소프트웨어적으로 생성되는 스레드 (동시성)

스케줄러 Operator

Reactor에서는 Operator를 통해 Scheduler를 사용할 수 있다.

subscribeOn()

구독이 발생한 직후 실행될 스레드를 지정하는 Operator. 구독 시점 직후에 원본 Publisher의 동작을 수행하기 위한 스레드라고 볼 수 있다.

doOneNext : 원본 Flux에서 데이터가 emit될 때 동작한다.

doOnSubscribe : 구독이 발생한 시점에 추가적인 어떤 처리가 필요한 경우 해당 처리 동작을 추가할 수 있다.

현재 실행되는 스레드에서 실행하게 된다 (=. 구독이 발생한 시점에 실행되는 스레드)

publishOn()

DownStream으로 Signal을 전송할 때 실행되는 스레드를 제어하는 역할을 하는 Operator.

코드 상에서 `publishOn()` 기준으로 아래쪽 DownStream의 실행 스레드를 변경한다.

Parallel()

라운드 로빈 방식으로 CPU 코어 개수만큼 스레드를 병렬로 실행하여 물리적인 스레드를 할당한다. 이 때, 어떤 작업을 처리하기 위해 물리적인 스레드 전부를

사용하지 않아도 되는 경우 스레드 개수를 설정해 병렬로 처리할 수 있다.

※ Reactor에서 R.R 방식으로 CPU의 논리적인 코어 수에 맞게 데이터를 그룹화 한 것을 **rail** 이라고 표현

parallel : emit되는 데이터를 CPU의 논리적인 코어 수에 맞게 사전에 골고루 분배하는 역할

runOn : 실제로 병렬 작업을 수행할 스레드의 할당을 담당

PublisherOn과 Subscriber 동작

원본 publisher와 나머지 동작을 역할에 맞춰 스레드를 분리하기 위해 SubscribeOn과 PulbishOn을 함께 사용하여 스레드를 관리한다.

다음 예제를 살펴보자.

```
@Slf4j
public class MixInSubscrbeOnPublishOn {


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

        Flux.fromArray(new Integer[]{1, 3, 5, 7})

                .subscribeOn(Schedulers.boundedElastic()) // 원본 Flux 데이터가 emit될 때 boundedElastic 방식의 스레
```

```
            .doOnNext(data -> log.info("# doOnNext fromArray: {}", data))
            .filter(data -> data > 3)
            .doOnNext(data -> log.info("# doOnNext filter: {}", data))

            .publishOn(Schedulers.parallel())  // DownStream의 데이터 처리를 parallel() 방식의 스레드를 사용해 처리
            .map(data -> data * 10)
            .doOnNext(data -> log.info("# doOnNext map: {}", data))
            .subscribe(data -> log.info("# onNext: {}", data));

        Thread.sleep(500L);
    }
}
```

예제 코드를 살펴보았을 때 실행 스레드를 목적에 맞춰 적절하게 분리해낼 수 있음을 알 수 있었다.

Scheduler 종류

- immediate( ) : 별도의 스레드를 추가적으로 생성하지 않고, 현재 스레드에서 작업을 처리할 때에 사용
  - publishOn을 사용하며 immediate( )를 사용하는 이유는?

    → 특정 메서드를 호출하 때 파라미터로 Scheduler를 전달하는 시점에서 메서드 내부 Operator 체인 작업에서 파라미터로 받은 Scheduler의 스레드 방식이나 스레드를 실행하도록

    하고 싶은 경우, 혹은 스레드를 추가 할당하고 싶지 않은 경우 고려해볼 수 있다.
- single( ): 스레드 하나만 생성해 Scheduler가 제거되기 전까지 재사용하는 방식
  - Operator 체인에서 스레드가 생성된 이후 같은 체인을 다시 실행하 때 생성된 스레드가 존재하게 되면 이를 활용하는 방법이다.
  - 하나의 스레드를 재사용하면서 다수의 작업을 처리할 수 있는데, 지연 시간이 짧은 작업을 처리할 때 효과적이다.
- newSingle( ): 호출할 때마다 매번 새로운 스레드 하나를 생성
  - 스레드의 이름을 설정하고, 데몬 스레드 여부를 결정할 수 있다.
- boundedElastic( ) : ExecutorService 기반의 스레드 풀을 생성한 후, 정해진 수만큼의 스레드를 사용하여 작업을 처리하고 종료된 스레드는 반납 후 재사용하는 방식
  - Blocking I/O 작업을 효과적으로 처리하기 위한 방식이다.
  - 다른 Non-Blocking 처리에 영향을 주지 않도록 전용 스레드를 할당해서 Blocking I/O 작업을 처리하므로 처리 시간을 효율적으로 단축할 수 있다.
- parallel( ) : ExecutorService 기반의 스레드 풀을 생성한 후, 정해진 수만큼의 스레드를 사용하여 작업을 처리하고 종료된 스레드는 반납 후 재사용하는 방식
  - Non-Blocking I/O에 최적화되어 있는 스케줄러로서 CPU 코어 수만큼 스레드 생성한다.

3. Context

Reactor의 Context는 Operator와 같은 Reactor 구성요소 간에 전파되는 Key/Value 저장소

Operator 체인상의 각 Operator가 Context 정보를 동일하게 이용할 수 있도록 DownStream에서 UpStream으로 Context가 전파되는 특징을 가지고 있다.

특징

- 각각의 실행 스레드와 매핑되는 ThreadLocal과 달리 Subscriber와 매핑

- DownStream에서 UpStream으로 전파. 이후 각 Operator가 해당 Context 정보를 동일하게 이용 가능
  - 이런 특징으로 ContextWrite( )를 사용할 때에는 Operator 체인의 가장 마지막에 위치
- 동일한 키에 대한 값을 중복해서 저장하면 Operator 체인에서 가장 위쪽에 위치한 contextWrite( )가 저장한 값으로 덮음
- 구독이 발생할 때마다 하나의 Context가 해당 구독에 연결
- Iner Sequence 내부에서는 외부 Context 접근 가능. Inner Sequence 외부에서 내부 Context로는 접근 불가

Context에 쓰인 데이터를 읽는 방식은 크게 두 가지로 구분할 수 있다.

1. 원본 데이터 소스 레벨에서 읽는 방식

   deferContextual( ) : defer( )와 같은 원리로 동작. Context에 저장된 데이터와 원본 데이터 소스의 처리를 지연시키는 역할
2. Operator 체인의 중간에서 읽는 방식

   transformDeferredContextual( )

여기서 Context의 데이터를 읽어야 하는 경우 받는 파라미터로 정의된 람다 표현식을 확인해보면 ContextView 타입의 객체라는 것을 확인할 수 있다.

살펴볼 점은, Reactor에서는 Operator 체인 상의 서로 다른 스레드들이 Context에 저장된 데이터에 손쉽게 접근이 가능하다.

이러한 기능으로 인해 스레드 안정성이 보장되어야 하는데 Context의 데이터 저장의 Flow를 살펴보면 다음과 같다.

Context.put( ) → 불변 객체 → contextWrite( )로 전달함