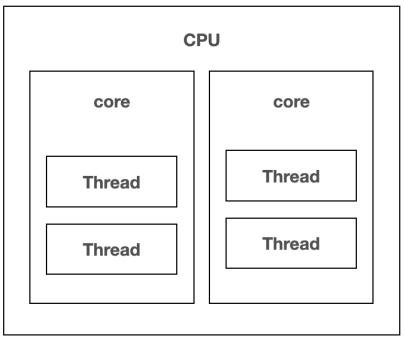
4~6일차 - Chapter 10 ~ 13

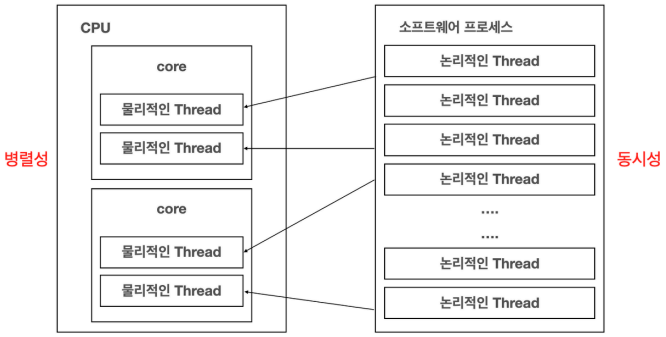
Chapter 10. Scheduler

10.1 스레드(Thread)의 개념 이해

- **물리적인 스레드(Pysical Thread)** : CPU 코어를 논리적으로 나눈 것
  - 듀얼코어 4 스레드 (작업관리자 [성능]에서 확인 가능)



- **논리적인 스레드(Logical Thread)** : 프로세스 내에서 실행되는 세부 작업의 단위
  - 이론적으로는 제한이 없지만 실제로는 물리적인 스레드의 가용 범위내에서 생성 가능



💡

- 병렬성(Parallelism) : 물리적인 스레드가 실제로 동시에 실행됨
- 동시성(Concurrency) : 동시에 실행되는 것처럼 보이지만 실제로는 아님
  - 무수히 많은 논리적인 스레드가 물리적인 스레드를 아주 빠른 속도로 번갈아 가면서 사용

10.2 Scheduler란?

- 어떤 스레드에서 무엇을 처리할지 제어(경쟁조건 등 고려)를 Scheduler가 개발자 대신 수행
  - 코드가 간결해짐
  - 개발자 부담 적어짐

10.3 Scheduler를 위한 전용 Operator

- subscribeOn()
  - 구독 발생 직후 실행될 스레드 지정
  - 원본 Publisher 동작 수행을 위한 스레드

```
public static void main(String[] args) throws InterruptedException {
    Flux.fromArray(new Integer[] {1, 3, 5, 7})
            .subscribeOn(Schedulers.boundedElastic())
            .doOnNext(data -> log.info("# doOnNext: {}", data))
            .doOnSubscribe(subscription -> log.info("# doOnSubscribe"))
            .subscribe(data -> log.info("# onNext: {}", data));

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

```
08:35:03.363 [main] INFO - # doOnSubscribe // 최초 실행 스레드 : main
08:35:03.371 [boundedElastic-1] INFO - # doOnNext: 1 // 구독 발생 직후 스레드 : boundedElastic
08:35:03.373 [boundedElastic-1] INFO - # onNext: 1
08:35:03.373 [boundedElastic-1] INFO - # doOnNext: 3
08:35:03.373 [boundedElastic-1] INFO - # onNext: 3
08:35:03.373 [boundedElastic-1] INFO - # doOnNext: 5
08:35:03.373 [boundedElastic-1] INFO - # onNext: 5
08:35:03.373 [boundedElastic-1] INFO - # doOnNext: 7
08:35:03.373 [boundedElastic-1] INFO - # onNext: 7
```

- publishOn()
  - Downstream으로 Signal을 전송할 때 실행되는 스레드 제어

```
public static void main(String[] args) throws InterruptedException {
    Flux.fromArray(new Integer[] {1, 3, 5, 7})
            .doOnNext(data -> log.info("# doOnNext: {}", data))
            .doOnSubscribe(subscription -> log.info("# doOnSubscribe"))
            .publishOn(Schedulers.parallel())
            .subscribe(data -> log.info("# onNext: {}", data));

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

```
08:40:35.020 [main] INFO - # doOnSubscribe
08:40:35.024 [main] INFO - # doOnNext: 1
08:40:35.027 [main] INFO - # doOnNext: 3
08:40:35.027 [parallel-1] INFO - # onNext: 1 // Downstream 실행 스레드 변경
08:40:35.027 [main] INFO - # doOnNext: 5
08:40:35.027 [main] INFO - # doOnNext: 7
08:40:35.027 [parallel-1] INFO - # onNext: 3
08:40:35.027 [parallel-1] INFO - # onNext: 5
08:40:35.027 [parallel-1] INFO - # onNext: 7
```

- parallel()

- 병렬성을 가지는 물리적인 스레드
  - cf. `subscribeOn()`, `publishOn()`은 동시성을 가지는 논리적인 스레드
- 라운드 로빈 방식으로 논리적인 코어(물리적인 스레드) 개수만큼의 스레드를 병렬로 실행
- Example10_3

> `subscribeOn()`, `publishOn()`를 활용하여 Publisher에서 데이터를 emit하는 스레드와 emit된 데이터를 가공 처리하는 스레드를 적절하게 분리할 수 있음

Scheduler의 종류

- `Schedulers.immediate()` : 별도의 스레드를 추가적으로 생성하지 않고, 현재 스레드에서 작업 처리
- `Schedulers.single()` : 스레드 하나만 생성해서 Scheduler가 제거되기 전까지 재사용
- `Schedulers.newSingle()` : 매번 새로운 스레드 하나 생성
- `Schedulers.boundedElastic()` : ExecutorService 기반의 스레드 풀 생성한 후, 정해진 수만큼의 스레드를 사용하여 작업을 처리하고 작업 종료된 스레드는 반납하여 재사용
  - 기본적으로 CPU 코어 수 x 10만큼의 스레드 생성
  - 최대 100,000개의 작업 큐에서 대기 가능
  - Blocking I/O 작업을 효과적으로 처리하기 위한 방식
  - 다른 Blocking I/O 작업이 Non-Blocking 처리에 영향을 주지 않도록 전용 스레드 할당
- `Schedulers.parallel()` : Non-Blocking I/O에 최적화되어 있는 Scheduler로서 CPU 코어 수만큼의 스레드 생성
- `Schedulers.fromExecutorService()` : 기존에 이미 사용하고 있는 ExecutorService가 있다면 이 ExecutorService로부터 Scheduler를 생성하는 방식
- `Schedulers.newXXXX()` : 스레드 이름, 생성 가능한 디폴트 스레드의 개수, 스레드의 유효 시간, 데몬 스레드로의 동작 여부 등 직접 지정해서 커스텀 스레드 풀 생성

Chapter 11. Context

11.1 Context란?

- 어떠한 상황에서 그 상황을 처리하기 위해 필요한 정보
- Reactor에서 Context의 정의

> Operator 같은 Reactor 구성요소 간에 전파되는 key/value 형태의 저장소

  - **전파** : Downstream에서 Upstream 체인상의 각 Operator가 해당 Context 정보를 동일하게 이용할 수 있음
- Reactor의 Context는 Subscriber와 매핑됨
  - cf. ThreadLocal의 경우 실행 스레드와 매핑됨
- 즉, 구독이 발생할 때마다 해당 구독과 연결된 하나의 Context가 생김

```
public static void main(String[] args) throws InterruptedException {
    Mono
        .deferContextual(ctx →
            Mono
                .just("Hello" + " " + ctx.get("firstName"))
```

```
                .doOnNext(data -> log.info("# just doOnNext : {}", data))
        )
        .subscribeOn(Schedulers.boundedElastic())
        .publishOn(Schedulers.parallel())
        .transformDeferredContextual(
                (mono, ctx) -> mono.map(data -> data + " " + ctx.get("lastName"))
        )
        .contextWrite(context -> context.put("lastName", "Jobs"))
        .contextWrite(context -> context.put("firstName", "Steve"))
        .subscribe(data -> log.info("# onNext: {}", data));

    Thread.sleep(100L);
}
```

- `contextWrite()` Operator로 Context에 데이터 쓰기 작업을 할 수 있다.
- `Context.put()`으로 Context에 데이터를 쓸 수 있다.
- `deferContextual()` Operator로 Context에 데이터 읽기 작업을 할 수 있다.
- `Context.get()`으로 Context에서 데이터를 읽을 수 있다.
- `transformDeferredContextual()` Operator로 Operator 중간에서 Context에 데이터 읽기 작업을 수 있다.

11.2 자주 사용되는 Context 관련 API

- Context API (쓰기)
  - put(key, value) : key/value 형태로 Context에 값을 쓴다.
  - of(key1, value1, key2, value2, ...) : key/value 형태로 Context에 여러 개의 값을 쓴다.
  - pulAll(ContextView) : 현재 Context와 파라미터로 입력된 ContextView를 merge한다.
  - delete(key) : Context에서 key에 해당하는 value를 삭제한다.
- ContextView API (읽기)
  - get(key) : ContextView에서 key에 해당하는 value를 반환한다.
  - getOrEmpty(key) : ContextView에서 key에 해당하는 value를 Optional로 래핑해서 반환한다.
  - getOrDefault(key, default value) : ContextView에서 key에 해당하는 value를 가져온다. key에 해당하는 value가 없으면 default value를 가져온다.
  - hasKey(key) : ContextView에서 특정 key가 존재하는지를 확인한다.
  - isEmpty() : Context가 비어 있는지 확인한다.
  - size() : Context 내에 있는 key/value의 개수를 반환한다.

11.3 Context의 특징

- Context는 구독이 발생할 때마다 하나의 Context가 해당 구독에 연결된다.
- Context는 Operator 체인의 아래에서 위로 전파된다.
  - 모든 Operator에서 Context에 저장된 데이터를 읽어올 수 있으려면 `contextWrite()`을 Operator 체인의 맨 마지막에 둬야 한다.
  - 동일한 키에 대한 값을 중복해서 저장하면 Operator 체인에서 가장 위쪽에 위치한 `contextWrite()`이 저장한 값으로 덮어쓴다.
- Inner Sequence 내부에서는 외부 Context에 저장된 데이터를 읽을 수 있다.
- Inner Sequence 외부에서는 Inner Sequence 내부 Context에 저장된 데이터를 읽을 수 없다.

- 인증 정보 같은 직교성(독립성)을 가지는 정보를 전송하는 데 적합하다.

Chapter 12 Debugging

12.1 Reactor에서의 디버깅 방법

- 비동기, 선언형 프로그래밍 방식의 Reactor는 디버깅이 쉽지 않음

12.1.1 Debug Mode를 사용한 디버깅

- `Hooks.onOperatorDebug()`를 통해 디버그 모드 활성화 (Example12_1)
- 체인상에서 에러가 시작된 지점부터 에러 전파 상태를 표시
- 비용이 많이 드는 동작
  - 스택트레이스 캡처
  - 에러가 발생한 Assembly의 스택트레이스를 원본 스택트레이스 중간에 끼워 넣음
  - 운영 환경에서는 `reactor.tools.agent.ReactorDebugAgent` 이용
- IntelliJ에서는 [Enable Reactor Debug mode]에서 `Hooks.onOperatorDebug()` 또는 `ReactorDebugAgent.init()` 중 택1 하여 활성화 가능

12.1.2 checkpoint( ) Operator를 사용한 디버깅

- `checkpoint()` Operator를 사용하여 특정 Operator 체인 내의 스택트레이스를 캡처
  - 각 Operator 체인에 `checkpoint()`를 추가한 후, 범위를 좁혀 가면서 단계적으로 에러 발생 지점을 찾는다.
- Traceback 출력 (Example12_2)

```
.checkpoint()
```

- Traceback 출력 없이 식별자를 포함한 Description 출력 (Example12_4)

```
.checkpoint("Example12_4.zipWith.checkpoint")
```

- Traceback과 Description 모두 출력 (Example12_5)
  - forceStackTrace=true로 지정

```
.checkpoint("Example12_4.zipWith.checkpoint", true)
```

12.1.3 log( ) Operator를 사용한 디버깅

- `log()` Operator를 추가하여 Reactor Signal 출력 (Example12_7)

Chapter 13. Testing

- StepVerifier - Operator 체인의 다양한 동작 방식 테스트
  - Flux 또는 Mono를 Reactor Sequence로 정의한 후, 구독 시점에 해당 Operator 체인이 사니리오대로 동자갛는지 테스트

    - Signal 이벤트 테스트
        - create() 통해 Sequence 생성
        - expectXXXX()를 통해 예상되는 Signal 기대값 평가
        - verify()로 검증
- TestPublisher - Signal을 발생시키며 원하는 상황을 미세하게 재연하며 테스트 진행
- PublisherProbe - Sequence의 실행 경로를 테스트