디버깅 및 테스트

1. Debugging
2. Testing
TestPublisher 테스팅

1. Debugging

프로그래밍 방식에 따라 문제가 발생한 원인을 단계적으로 찾아가기 위해 아래와 같은 방법을 생각해볼 수 있다.

- 동기식 / 명령형 프로그래밍 방식 : Exception이 발생했을 때, 스택트레이스를 확인하거나 예외 발생이 예상되는 코드에 BP를 걸어 단계적으로 찾아감. (⇒ 상대적 디버깅이 쉬움)
- 선언형 프로그래밍 방식 : Reactor에서 처리되는 작업들이 대부분 비동기적으로 실행 (⇒ 디버깅이 어려움)

Debug Mode를 사용한 디버깅

Reactor에서의 디버그 모드 활성화는 `Hooks.onPeratorDebug()` 를 통해 이루어진다.

원래 출력되는 에러 메시지와 스택 트레이스 사이에 Operator 체인 상에서 에러가 발생한 지점을 정확히 가리키고, 에러가 시작된 지점부터 전파 상태를 표시

한계점

- 애플리케이션 내에 있는 모든 Operator의 스택트레이스를 캡처

  → 애플리케이션 내에서 비용이 많이 드는 동작 과정
- 에러가 발생하면 캡처한 정보를 기반으로 에러가 발생한 Assembly 스택트레이스를 원본 스택트레이스 중간에 끼워 넣음

  ※ Assembly : Reactor의 Operator들은 Mono / Flux를 리턴. Operator 체인에서 리턴하는 새로운 Mono / Flux가 선언된 지점

따라서 처음부터 디버그 모드를 활성화 하는 것은 권장되지 않는다.

프로덕션 환경의 경우 모든 Operator 체인의 스택트레이스 캡처 비용을 절약하고 디버깅 정보를 추가할 수 있도록 ReactorDebugAgent를 이용해볼 수 있다.

checkpoint( ) Operator를 사용한 디버깅

특정 Operator 체인 내의 스택트레이스만 캡처

사용 방법

1. Traceback을 출력하는 방법

   실제 에러가 발생한 assembly 지점 또는 에러가 전파된 assembly 지점의 traceback이 추가

```
public static void main(String[] args) {
    Flux
            .just(2, 4, 6, 8)
            .zipWith(Flux.just(1, 2, 3, 0), (x, y) -> x / y)
            .map(num -> num + 2)
            .checkpoint()
            .subscribe(
                    data -> log.info("# onNext: {}", data),
                    error -> log.error("# onError: ", error)
            );

}
```

예제 코드를 살펴볼 경우, `checkpoint()` 지점까지는 에러가 전파되었음을 확인할 수 있지만 정확하게 어느 지점에서 발생한 에러인지 파악하기는 어렵다.

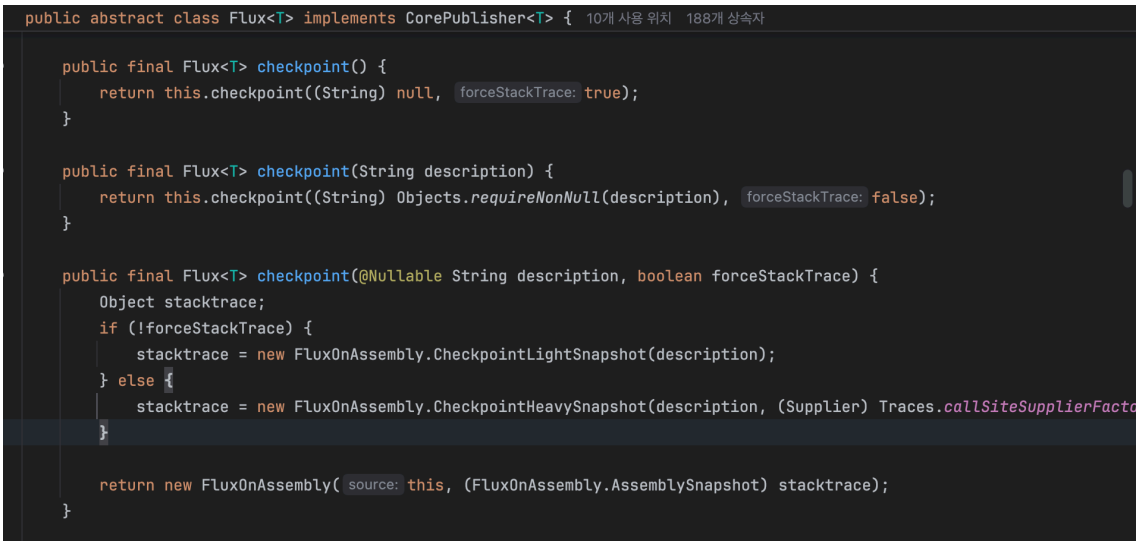
만약 정확한 에러 발생 지점을 찾아야 한다면 현재 `checkpoint()`를 기준으로 Upstream의 Assembly 지점에 `checkpoint()`를 다르게 설정해 Operator 체인 상 어느 지점에서 에러가 발생했는지 구분해야 한다.

2. Traceback 출력 없이 식별자를 포함한 Description을 출력해서 에러 발생 지점을 예상

   `checkpoint()` 에는 파라미터로 다음과 같은 인자를 받을 수 있다.

   description (Param 1) : 설명

   forceStackTrace (Param 2) : traceback 출력 여부

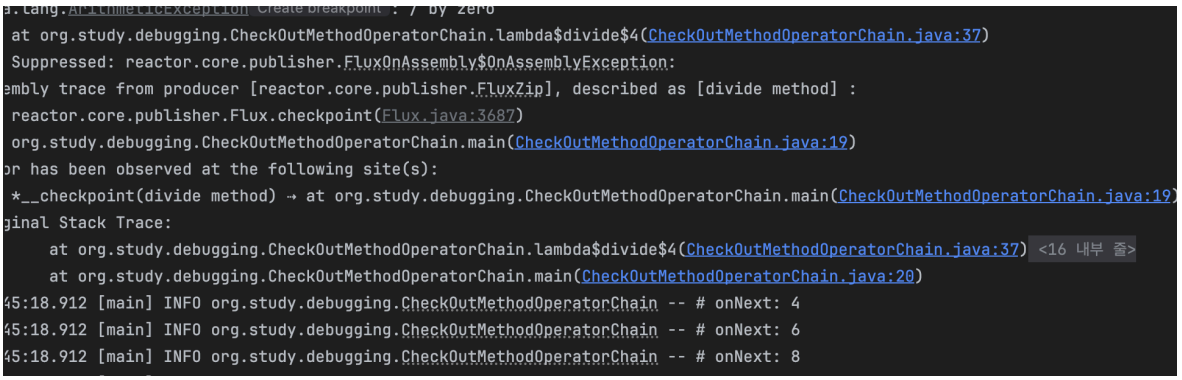


```
public abstract class Flux<T> implements CorePublisher<T> {  10개 사용 위치  188개 상속자

    public final Flux<T> checkpoint() {
        return this.checkpoint((String) null,  forceStackTrace: true);
    }

    public final Flux<T> checkpoint(String description) {
        return this.checkpoint((String) Objects.requireNonNull(description),  forceStackTrace: false);
    }

    public final Flux<T> checkpoint(@Nullable String description, boolean forceStackTrace) {
        Object stacktrace;
        if (!forceStackTrace) {
            stacktrace = new FluxOnAssembly.CheckpointLightSnapshot(description);
        } else {
            stacktrace = new FluxOnAssembly.CheckpointHeavySnapshot(description, (Supplier) Traces.callSiteSupplierFacto
        }

        return new FluxOnAssembly( source: this, (FluxOnAssembly.AssemblySnapshot) stacktrace);
    }
```

Operator 체인이 복잡해지는 경우 에러 발생 지점을 찾는 것이 쉽지 않다. 이 경우 description을 이용해 에러 발생 지점을 유추할 수 있도록 구분할 수 있다.

→ 메서드 별로 고유한 기능에 대해 Reactor Sequence를 실행하고 Operator 체인을 반환하는 경우라면 반환 값이 `checkpoint()` 의 description을 활용하여 메서드에서 에러가 발생할 경우를 대비해 명시적으로 설명을 추가할 수 있다.

```java
@Slf4j
public class CheckOutMethodOperatorChain {
    public static void main(String[] args) {

        Flux<Integer> source = reactor.core.publisher.Flux.just(2, 4, 6, 8);
        Flux<Integer> other = reactor.core.publisher.Flux.just(1, 2, 3, 0);

        divide(source, other)
                .checkpoint("divide method", true)
                .subscribe(
                        data -> log.info("# onNext: {}", data),
                        error -> log.error("# onError: ", error)
                );

        plus(source)
                .checkpoint("plus method", true)
                .subscribe(
                        data -> log.info("# onNext: {}", data),
                        error -> log.error("# onError: ", error)
                );

    }


    private static Flux<Integer> divide(Flux<Integer> source, Flux<Integer> other) {
        return source
                .zipWith(other, (x, y) -> x / y);
    }

    private static Flux<Integer> plus(Flux<Integer> source) {
        return source
                .map(num -> num + 2);
    }
}
```



이후 실행시켜 에러를 확인해보면 어떤 checkpoint( )까지 에러가 전파되었는지 확인해볼 수 있다.

3. log( ) Operator

   Reactor Sequence의 동작을 로그로 출력. 이 때에 `onSubscribe()`, `request()`, `onNext()` 같은 Signal이 출력

log() Operator를 사용하면 에러가 발생한 지점에 단계적으로 접근할 수 있고, 사용 개수에 제한이 없으므로 다른 Operator 뒤에 추가해 Reactor Sequence의 내부 동작을 좀 더 상세하게

분석하면서 디버깅할 수 있음

※ Traceback : 디버그 모드를 활성화하면 Operator의 Assembly 정보를 캡처하는데 이중에서 에러가 발생한 Operator의 스택트레이스를 캡처한 Assembly 정보

2. Testing

Reactor에서 가장 일반적인 테스트 방식은 Flux / Mono를 이용한 Reactor Sequence에서 구독 시점에 Operator 체인이 예상대로 동작하는지를 테스트하는 것

항목

아래와 같은 항목들에 대해 단계적으로 테스트를 진행할 수 있다.

- Reactor Sequence에서 다음에 발생할 Signal이 무엇인지?
- 기대하던 데이터들이 emit 되었는지?
- 특정 시간 동안 emit된 데이터가 있는지?

이렇듯, Operator 체인의 다양한 동작을 테스트하기 위해 Reactor에서는 StepVerifier를 제공한다. 테스트 방법에는 테스트하고자 하는 항목에 따라 테스트 코드를 구분할 수 있고, 사용되는 API가 다르다.

Signal 이벤트 테스트

Reactor Sequence에서 발생하는 Signal 이벤트를 테스트하는 것. 세부 항목을 살펴보면 다음과 같이 Reactor Sequence에서 발생하는 Signal을 대상으로 테스트가 진행되는 것을 확인할 수 있다.

항목

- 구독이 이루어지는지?
- onNext Signal을 통해 전달되는 값이 파라미터의 값과 같은지?
- onComplete / onError Singal이 전송 되는지?
- 구독 시점에서 이후에 emit될 데이터의 개수가 적절한지?
- 구독 시점 이후에 Context가 전파되었는지?

```java
@Test
public void takeNumberTest() {
    Flux<Integer> source = Flux.range(0, 1000);
    StepVerifier
            .create(ReactorSignalTest.takenumber(source, 500),
                    StepVerifierOptions.create().scenarioName("Verify from 0 to 499"))
            .expectSubscription()
            .expectNext(0)
            .expectNextCount(498)
            .expectNext(500)
            .expectComplete()
            .verify();
}

public static Flux<Integer> takenumber(Flux<Integer> source, long n) {
    return source
            .take(n);
}
```

테스트 코드를 면밀히 살펴보면 아래의 내용과 같다.

- 구독이 이루어지고 있는지 검증
- 구독 시점에 다음 emit할 데이터의 값이 0인지?
- 그 이후 시점에 데이터 emit될 데이터의 양이 498개가 될 수 있는지?
- onComplete Signal이 전송되었는지?

시간 기반(Time-based) 테스트

가상의 시간을 이용해 미래에 실행되는 Reactor Sequence의 시간을 앞당겨 테스트할 수 있는 기능을 지원

이 때 사용되는 메서드에서는 가상 스케줄러의 제어를 받아 특정 시간에 메서드가 호출될 수 있느지 테스트를 진행할 수 있다.

- withVirtualTime( ) 메서드 : VirtualTimeScheduler라는 가상의 스케줄러의 제어를 받아 then( ) 메서드를 사용해 후속 작업을 할 수 있도록 함.

```java
Java
복사 캡션 ...
public class TimeBasedTest {

    @Test
    public void getExecutableTest() {

        StepVerifier
                .withVirtualTime(() -> TimeBasedTest.executable(
                        Flux.interval(Duration.ofHours(1)).take(1)
                ))
                .expectSubscription()
                .then(() -> VirtualTimeScheduler
                        .get()
                        .advanceTimeBy(Duration.ofHours(1)))
                .expectNextCount(1)
                .expectComplete()
                .verify();

    }

    public static Flux<Tuple2<Long, String>> executable(Flux<Long> source) {
        return source
                .flatMap(not -> Flux.just(
                        Tuples.of(not, "fixture")
                ));
    }
}
```

외에도 verify( ) 메서드를 이용해 테스트를 수행하는데 걸리는 시간을 제한해, 지정한 시간 내로 테스트 메서드가 종료되는지 확인할 수 있다.

지정한 시간을 초과해 테스트 메서드가 종료되는 경우 AssertionError가 발생해 테스트가 실패하게 된다.

- exepctNoEvent( ) 메서드: 지정한 시간 동안 어떤 Signal 이벤트도 발생하지 않았음을 기대. 이와 동시에 지정한 시간만큼 시간을 앞당기게 된다.

Backpressure 테스트

원본 Flux에서 Publisher에 의해 데이터가 emit되는 속도보다 Subscriber의 데이터 처리 속도보다 빠르기 때문에 Backpressure를 설정하여 시스템에 문제가 발생할 수 있는 부분을 방지한다.

이런 Backpressure 역시 테스트 대상에 포함될 수 있고, Backpressure 방법에 따라 사용되는 에러나 Drop에 대해서도 테스트가 가능하다.

메서드 참고

thenConsumeWhile( ) : Comsumer가 처리하는 데이터가 람다식으로 주어지는 파라미터의 조건을 만족하는지 검증

verifyThenAssertThat( ) : 검증을 트리거 한 뒤 추가적인 Assertions을 사용 가능

hasDroppedElements( ) : Drop된 데이터가 있음을 검증

Context 테스트

Reactor Sequence에서 Context 역시 테스트가 가능하다.

메서드 참고

expectAccessibleContext( ) : 구독 이후, Context가 전파됨을 기대

hasKey( ) : 키에 해당하는 값이 있음을 기대

then( ) : Sequence 다음 Signal 이벤트 기댓값을 평가

Record 기반 테스트

Reactor Sequence를 테스트하다 보면 emit된 데이터에 대해 구체적인 조건으로 검증해야 하는 경우가 있다. 이 경우 recordWith( )을 사용해볼 수 있다.

→ recordWith( ) : 파라미터로 전달한 Java의 컬렉션에 emit도니 데이터를 추가하는 것으로 세션을 시작

메서드 참고

recordWith( ) : emit된 데이터에 대한 기록을 시작

consumeRecordedWith( ) : 컬렉션에 기록된 데이터를 소비

TestPublisher 테스팅

TestPublisher를 사용하면, 프로그래밍 방식으로 Signal을 발생시키면서 원하는 상황을 미세하게 재연하며 테스트를 진행할 수 있다.

정상 동작 TestPublisher 의미 (Well-behaved)

emit하는 데이터가 Null인지, 요청하는 개수보다 더 많은 데이터를 emit하는지 등의 리액티브 스트림즈 사양 위반 여부를 체크한다는 의미

오동작 TestPublisher 의미 (Misbehaving)

리액티브 스트림즈 사양 위반 여부를 사전에 체크하지 않는다는 의미. 따라서 리액티브 스트림즈 사양에 위반되더라도 TestPublisher 데이터는 emit 가능

TestPublisher를 통해 복잡한 로직이 포함된 대상 메서드를 테스트하거나 조건에 따라서 Signal을 변경해야 하는 등 특정 상황에 대해 테스트하기가 용이하다.

Signal 발생

next( ) : 1개 이상의 onNext Signal 발생

emit( ) : 1개 이상의 onNext Signal을 발생한 뒤, onComplete Signal 발생

complete( ) / error( ) : onComplete / error Signal 발생

오동작하는 TestPublisher를 생성하기 위한 위반 조건

- ALLOW_NULL : 전송한 데이터가 null 이더라도 NullPointerException을 발생시키지 않고 다음 호출을 진행
- CLEANUP_ON_TERMINATE : onComplete, onError, emit 같은 Terminal Signal을 연달아 여러 번 보낼 수 있도록 함
- DEFER_CANCELLATION : cancel Signal을 무시하고 계속해서 Signal을 emit
- REQUEST_OVERFLOW : 요청 개수보다 더 많은 Signal이 발생하더라도 다음 호출을 진행

PublisherProbe를 사용한 테스팅

PublisherProbe는 Sequence의 실행 경로를 테스트할 수 있다.

주로 주어진 조건에 따라 Reactor Sequence 상에서 Sequence가 분리되는 경우가 존재하는데, 실행 경로를 추적해서 정상적으로 실행되었는지 테스트하기 위함이다.

```
@Test
public void publisherProbeTest() {
    PublisherProbe<String> probe = PublisherProbe.of(PublisherProbeTest.supplyStandbyPower());

    StepVerifier
            .create(PublisherProbeTest.processTask(
                    PublisherProbeTest.supplyMainPower(),
                    probe.mono()
            ))
            .expectNextCount(1)
            .verifyComplete();

    probe.assertWasSubscribed();
    probe.assertWasRequested();
    probe.assertWasNotCancelled();
}
```

processTask( ) 메서드는 2개의 파라미터에 대해 조건을 기준으로 특정 Sequence를 반환하는 메서드이다.

이 경우 검증하고자 하는 Reactor Sequence가 담긴 Mono / Flux 객체를 PublisherProbe에 담아서 이를 활용할 수 있다.

메서드 참고

assertWasSubscribed( ) : 기대하는 Publisher가 구독을 했는지?

assertWasRequested( ) : 기대하는 Publisher가 요청을 했는지?

assertWasNotCancelled( ) : 기대하는 Publisher가 중간에 최소가 되지 않았는지?