Reactor 요약

1 Background

Microservice Inner Architecture 항목의 Gateway Server를 비롯하여 일부 Aggregation Server는 많은 요청을 처리할 수 있어야 하며, Domain Server로의 요청을 직렬 및 병렬 등 여러 방법으로 중개해야 한다. 또한 이 과정에서 발생할 수 있는 실패에 대한 대비를 해야 하며, 이를 잘 제어할 필요가 있다.

Spring Projects 중 이와 같이 다량의 요청에 대응할 수 있도록 만들어진 것은 Spring Framework 중 WebFlux이며, WebFlux는 기본적으로 Reactor를 이용하여 비동기 프로그래밍을 할 수 있도록 설계가 되어 있다.

2 Goals

- Reactor의 사용 배경에 대한 이해
- Reactor의 기초적인 사용법 학습
    - 비동기 시퀀스 제어
    - 테스트

3 Targets

- TR
- TL
- 이 외에 Aggregation Server나 Gateway Server를 다룰 멤버

4 Materials

- Reactor Core (https://godekdls.github.io/Reactor%20Core/contents/)
    - Reactor 공식 문서에 대한 한글 번역
    - 버전은 최신이 아니지만 프로젝트에 필요한 핵심은 크게 바뀌지 않았음
- Hands-on
    - https://github.com/reactor/lite-rx-api-hands-on

5 Summary - Reactor Core

5.1 Introducing Reactor

Reactor를 간단 요약하면,

- JVM 위에서 동작
- 완전한 논블로킹 리액티브 프로그래밍을 위한 기반 라이브러리
- 배압(backpressure)를 관리하는 방식으로 요구(demand)를 효율적으로 관리

- Java 8의 함수형 API와 직접 통합
- reactor-netty 프로젝트의 프로세스와 논블로킹 방식으로 통신 가능

5.2 Prerequisites

- Java 8 이상
- 전의 의존성 존재
    - org.reactivestreams:reactive-streams
        - Reactor v3.5.4 기준으로 v1.0.3

5.3 Blocking Can Be Wasteful

비동기 프로그래밍 패러다임

- 데이터 스트림과 변경 사항 전파에 초점
    - 정적 혹은 동적 데이터 스트림을 손쉽게 원하는 프로그래밍 언어로 표현할 수 있음
- Reactor는 리액티브 프로그래밍 패러다임의 구현체

Reactvie Extension(Rx) 라이브러리

- 마이크로소프트가 닷넷(.NET) 생태계에 만든 라이브러리
- 이후 RxJava라는 이름으로 JVM 위에서 실행되는 라이브러리로 구현됨

Reactive Streams

- RxJava를 비롯하여 비동기 프로그래밍 패러다임의 표준화
- 이를 구현한 Java 9의 기본 구현체의 이름은 Flow

기존 객체 지향 언어의 디자인 패턴과의 관계

- 리액티브 프로그래밍 패러다임은 종종 옵저버 디자인 패턴의 확장으로 사용되곤 했음
- 메인 리액티브 스트림 패턴을 이터레이터 디자인 패턴과 비교 가능
    - 이터레이터는 pull 기반
        - Iterable-Iterator 쌍의 성격과 유사
        - 명령형 프로그래밍 패턴
        - next()를 호출하는 것은 개발자의 책임
    - 리액티브 스트림은 push 기반
        - Publisher-Subscriber 쌍이 대신
        - 새로운 데이터가 있음을 Publisher가 Subscriber에게 통지
        - 명령형이 아닌 선언형으로 표현

5.3.1 Blocking Can Be Wasteful

프로그램 성능을 끌어올리는 방법 두 가지

- 병렬 처리
- 리소스를 효율적으로 사용

자바 개발자는 보통 블로킹 코드로 프로그램을 작성한다.

- 성능에 변경이 생기지만 않으면 괜찮을 수 있다.
- 하지만 리소스를 더 사용하는 쪽으로 확장되며,
- 경합이나 동시성 이슈가 발생한다.
- 또한 I/O 대기 시간 동안 리소스를 낭비하게 된다.
    - 병렬 처리는 만능 해결책이 아니다.

5.4 Asynchronicity to the Rescue?

리소스를 효율적으로 사용하기 위해 비동기, 논블로킹 코드를 작성하는 방법을 사용할 수 있다.

JVM 위에서 동작하는 비동기 코드를 만드는 방법

- Callbacks
- Futures

하지만 이런 기존 방식도 제약이 있다. 콜백은 조합이 까다로우며 유지 보수하기 어려운 코드를 만들어낼 수 있다. (일명 콜백 지옥)

```
userService.getFavorites(userId, new Callback<List<String>>() { // (1)
  public void onSuccess(List<String> list) { // (2)
    if (list.isEmpty()) { // (3)
      suggestionService.getSuggestions(new Callback<List<Favorite>>() {
        public void onSuccess(List<Favorite> list) { // (4)
          UiUtils.submitOnUiThread(() -> { // (5)
            list.stream()
                .limit(5)
                .forEach(uiList::show); // (6)
            });
        }

        public void onError(Throwable error) { // (7)
          UiUtils.errorPopup(error);
        }
      });
    } else {
      list.stream() // (8)
          .limit(5)
          .forEach(favId -> favoriteService.getDetails(favId, // (9)
            new Callback<Favorite>() {
              public void onSuccess(Favorite details) {
                UiUtils.submitOnUiThread(() -> uiList.show(details));
              }

              public void onError(Throwable error) {
                UiUtils.errorPopup(error);
              }
            }
          ));
    }
  }

  public void onError(Throwable error) {
    UiUtils.errorPopup(error);
  }
});
```

```
userService.getFavorites(userId) // (1)
           .flatMap(favoriteService::getDetails) // (2)
           .switchIfEmpty(suggestionService.getSuggestions()) // (3)
           .take(5) // (4)
           .publishOn(UiUtils.uiThreadScheduler()) // (5)
           .subscribe(uiList::show, UiUtils::errorPopup); // (6)
```

Future 객체는 상황이 조금 더 낫지만 여전히 조합해서 사용하는 것이 어려운 편이다.

- get() 메소드를 호출하면 결국 블로킹된다.
- 지연 연산(lazy computation)을 지원하지 않는다.
- 멀티 밸류에 대한 지원이 부족하며, 에러 처리를 커스텀하기 어렵다.

5.5 From Imperative to Reactive Programming

Reactor의 목적은 JVM의 고전적인 비동기 접근법의 문제를 해결하는 것

- 쉽게 구성할 수 있고, 가독성 있다.
- 데이터는 풍부한 연산자로 조작할 수 있는 플로우로 표현한다.
- 구독하기 전까진 아무 일도 일어나지 않는다.
- 배압(backpressure), 즉, 컨슈머가 프로듀서에 데이터 생산 속도가 너무 빠르다는 신호를 보낼 수 있다.
- 고수준이면서 동시성에 구애받지 않을 정도의 높은 수준으로 추상화한다.

5.5.1 Hot vs Cold

- Cold 시퀀스는 각 Subscriber마다 데이터 소스를 포함해서 새로 시작한다.
    - 예를 들어, 데이터 소스가 HTTP 호출을 래핑하고 있다면, 구독할 때마다 HTTP 요청을 새로 만든다.
- Hot 시퀀스는 Subscriber마다 매번 처음부터 만들지 않는다.
    - 나중에 구독한 구독자는 구독 이후 생산한 신호만 받는다.
    - 단, 캐싱을 통해 전체 혹은 일부를 재사용할 수 있다.

- hot 시퀀스는 구독자가 없을 때도 발생할 수 있다. (이때는 예외적으로 “구독하기 전에 아무 일도 일어나지 않는다”는 규칙이 적용되지 않는다.)

5.6 Flux

Flux<T>는 0개부터 N개까지의 아이템을 생산하는 비동기 시퀀스를 나타내는 표준 Publisher<T>이다.

종료 이벤트를 포함한 모든 이벤트는 선택 사항이다.

- onComplete 항목을 비우면 무한히 동작할 수 있게 된다.
- 예를 들면, Flux.interval(Duration)은 일정한 시간 값을 방출하는 무한한 Flux<Long>을 생산한다.

5.7 Mono

최대 1개의 아이템 생산에 특화된 Publisher<T>이다.

Flux가 지원하는 모든 연산자를 다 제공하지는 않는다.

하지만 일부 연산자를 통해 Flux로 전환이 가능하다.

5.7.1 Simple Ways to Create a Flux or Mono and Subscribe to It

```
Flux<String> seq1 = Flux.just("foo", "bar", "foobar");
```

```
List<String> iterable = Arrays.asList("foo", "bar", "foobar");
Flux<String> seq2 = Flux.fromIterable(iterable);
subscribe(); // (1)

subscribe(Consumer<? super T> consumer); // (2)

subscribe(Consumer<? super T> consumer,
          Consumer<? super Throwable> errorConsumer); // (3)

subscribe(Consumer<? super T> consumer,
          Consumer<? super Throwable> errorConsumer,
          Runnable completeConsumer); // (4)

subscribe(Consumer<? super T> consumer,
          Consumer<? super Throwable> errorConsumer,
          Runnable completeConsumer,
          Consumer<? super Subscription> subscriptionConsumer); // (5)
```

Flux와 Mono를 구독할 때는 Java 8의 람다를 사용하며, .subscribe() 메소드의 시그니처도 다양하다.

```
subscribe(); // (1)

subscribe(Consumer<? super T> consumer); // (2)

subscribe(Consumer<? super T> consumer,
          Consumer<? super Throwable> errorConsumer); // (3)

subscribe(Consumer<? super T> consumer,
          Consumer<? super Throwable> errorConsumer,
          Runnable completeConsumer); // (4)

subscribe(Consumer<? super T> consumer,
          Consumer<? super Throwable> errorConsumer,
          Runnable completeConsumer,
          Consumer<? super Subscription> subscriptionConsumer); // (5)
```

예시: 1부터 3까지의 Integer를 생산하고 간단하게 구독

```
Flux<Integer> ints = Flux.range(1, 3); // (1)
ints.subscribe(i -> System.out.println(i)); // (2)
```

```
1
2
3
```

5.8 Threading and Schedulers

리액터는 RxJava처럼 동시성에 구애받지 않는다.

- 특정 동시성 모델을 강요하지 않는다.
- 오히려 개발자가 통제할 수 있다.

대부분의 연산자가 이전 연산자를 실행했던 Thread에서 작업을 이어나간다.

만약 스레드를 지정한 적이 없다면 subscribe()를 호출한 Thread에서 실행한다.

```
public static void main(String[] args) throws InterruptedException {
  final Mono<String> mono = Mono.just("hello "); // (1)

  Thread t = new Thread(() -> mono
      .map(msg -> msg + "thread ")
      .subscribe(v -> // (2)
          System.out.println(v + Thread.currentThread().getName()) // (3)
      )
  )
  t.start();
  t.join();

}
```

```
hello thread Thread-0
```

5.8.1 Scheduler

- Reactor에서 코드 실행 모델과 실행 위치를 추상화한 것
- ExecutorService와 유사하게 스케쥴링을 담당하지만 자체 추상화가 좀 더 되어 있음
    - 테스트를 위한 가상 시간
    - trampolining
    - 즉각적인 스케쥴링 등

5.8.2 Schedulers

- Schedulers.immediate()
    - 현재 Thread에서 수행
- Schedulers.single()
    - 단일 Thread를 가지는 Scheduler를 생성
- Schedulers.boundedElastic()
    - 새 워커 풀을 만들고 여유 있는 스레드가 있으면 재사용
    - 너무 오랫동안 놀고 있는 스레드는 폐기
    - 한도에 도달하면 제출된 태스크는 100,000개까지 큐에 담고 여유가 생긴 스레드가 있으면 다시 스케줄링
    - 블로킹 I/O를 감쌀 때 활용
- Schedulers.parallel()
    - 병렬 작업을 튜닝할 수 있는 워커의 고정 풀
    - CPU 코어 수만큼 워크를 생성한다.
- Schedulers.fromExecutorService(ExecutorService)

5.9 Handling Errors

에러를 특정 값으로 대체하는 경우

```
Flux.just(1, 2, 0)
    .map(i -> "100 / " + i + " = " + (100 / i)) //this triggers an error with 0
    .onErrorReturn("Divided by zero :("); // error handling example
```

Predicate를 적용하여 복구 여부를 결정하는 경우

```
Flux.just(10)
    .map(this::doSomethingDangerous)
    .onErrorReturn(e -> e.getMessage().equals("boom10"), "recovered10");
```

에러를 다른 Publisher로 복구하는 경우

```
Flux.just("timeout1", "unknown", "key2")
    .flatMap(k -> callExternalService(k)
        .onErrorResume(error -> { // (1)
            if (error instanceof TimeoutException) // (2)
                return getFromCache(k);
            else if (error instanceof UnknownKeyException)  // (3)
                return registerNewEntry(k, "DEFAULT");
            else
                return Flux.error(error); // (4)
        })
    );
```

에러를 감싸 다시 던지는 경우

```
Flux.just("timeout1")
    .flatMap(k -> callExternalService(k))
    .onErrorMap(original -> new BusinessException("oops, SLA exceeded", original));
```

```
Flux.just("timeout1")
    .flatMap(k -> callExternalService(k))
    .onErrorResume(original -> Flux.error(
            new BusinessException("oops, SLA exceeded", original))
    );
```

로깅과 같이 에러에 대해 작업을 수행하지만 에러 자체는 다시 던지고 싶은 경우

```
LongAdder failureStat = new LongAdder();
Flux<String> flux =
Flux.just("unknown")
    .flatMap(k -> callExternalService(k) // (1)
        .doOnError(e -> {
            failureStat.increment();
            log("uh oh, falling back, service failed for key " + k); // (2)
        })
        // (3)
    );
```

에러 여부와 관계 없이 항상 마지막에 실행하고 싶은 경우

```
Stats stats = new Stats();
LongAdder statsCancel = new LongAdder();

Flux<String> flux =
Flux.just("foo", "bar")
    .doOnSubscribe(s -> stats.startTimer())
    .doFinally(type -> { // (1)
        stats.stopTimerAndRecordTiming(); // (2)
        if (type == SignalType.CANCEL) // (3)
          statsCancel.increment();
    })
    .take(1); // (4)
```

재시도가 필요한 경우

```
Flux.interval(Duration.ofMillis(250))
    .map(input -> {
        if (input < 3) return "tick " + input;
        throw new RuntimeException("boom");
    })
    .retry(1)
    .subscribe(System.out::println, System.err::println);

Thread.sleep(2100);
```

```
tick 0
tick 1
tick 2
tick 0
tick 1
tick 2
java.lang.RuntimeException: boom
```

재시도를 조금 더 상세히 제어하고 싶은 경우

```
Flux<String> flux = Flux
    .<String>error(new IllegalArgumentException())
    .doOnError(System.out::println)
    .retryWhen(Retry.from(companion ->
        companion.take(3)));
```

5.10 Test

5.10.1 Testing a Scenario with StepVerifier

```
public <T> Flux<T> appendBoomError(Flux<T> source) {
  return source.concatWith(Mono.error(new IllegalArgumentException("boom")));
}
```

```
@Test
public void testAppendBoomError() {
  Flux<String> source = Flux.just("thing1", "thing2");

  StepVerifier.create(
    appendBoomError(source))
    .expectNext("thing1")
    .expectNext("thing2")
    .expectErrorMessage("boom")
    .verify();
}
```

verify()를 꼭 호출 해야 하는 것에 주의하자. 그래야 subscription이 일어나고, 올바르게 종료되었는지 확인할 수 있다.

어떻게 종료되는지도 테스트 가능하다.

- verifyComplete()
- verifyError()
- verifyErrorMessage()

verify()는 타임아웃이 없음에 주의하자.

- verify(Duration)
- StepVerifier.setDefaultTimeout(Duration)

5.10.2 Manipulating Time

```
StepVerifier.withVirtualTime(() -> Mono.delay(Duration.ofDays(1)))
    .expectSubscription()
    .expectNoEvent(Duration.ofDays(1))
    .expectNext(0L)
    .verifyComplete();
```

StepVerifier와 테스트용 Scheduler의 소통으로 VirtualTimer가 하루치 시간을 감아 빠르게 동작한다.

5.10.3 PublisherProbe

```
private Mono<String> executeCommand(String command) {
    return Mono.just(command + " DONE");
}

public Mono<Void> processOrFallback(Mono<String> commandSource, Mono<Void> doWhenEmpty) {
    return commandSource
            .flatMap(command -> executeCommand(command).then()) // (1)
            .switchIfEmpty(doWhenEmpty); // (2)
}
```

```
@Test
public void testCommandEmptyPathIsUsed() {
    PublisherProbe<Void> probe = PublisherProbe.empty();

    StepVerifier.create(processOrFallback(Mono.empty(), probe.mono()))
                .verifyComplete();

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

5.11 Appendix

꼭 한 번씩 훑어 봐야 하는 내용

- https://godekdls.github.io/Reactor%20Core/appendixawhichoperatordoineed/
    - 시퀀스 생성법
    - 시퀀스 변경법
    - 시퀀스에서 일부를 필터링하는 방법
    - 에러를 핸들링하는 방법
    - 시간과 관련된 제어가 필요한 경우
    - Flux를 분할하는 방법
    - 비동기로 돌아가야 하는 경우
- https://godekdls.github.io/Reactor%20Core/appendixbfaqbestpracticesandhowdoi/
    - 동기 문맥을 비동기로 호출하는 방법
    - Chaining 없이 사용할 때 생길 수 있는 오류
    - zipWhen이나 zipWith가 동작하지 않는 경우
    - retry를 retryWhen으로 구현하는 방법
    - Backoff와 함께 retry하는 방법
    - MDC와 Context

6 추가 자료

- reactor-extra 시작하기
- 마블 다이어그램