**Project Reactor**

목차

[1. Project Reactor 개요 3](#_Toc55911730)

[1.1 Project Reactor 3](#_Toc55911731)

[1.2 Reactive Programming 3](#_Toc55911732)

[1.3 Reactive Streams 3](#_Toc55911733)

[1) 스트리밍 처리 4](#_Toc55911734)

[2) 비동기 방식 5](#_Toc55911735)

[3) 백 프레셔 6](#_Toc55911736)

[4) 리액티브 스트림즈 API 8](#_Toc55911737)

[2. Project Reactor 사용법 11](#_Toc55911738)

[2.1 구조 11](#_Toc55911739)

[2.2 환경 구성 12](#_Toc55911740)

[2.3 시퀀스 생성 14](#_Toc55911741)

[1) just 14](#_Toc55911742)

[2) range 16](#_Toc55911743)

[3) fromArray 16](#_Toc55911744)

[2.4시퀀스 변환 18](#_Toc55911745)

[1) map 18](#_Toc55911746)

[2) flat map 19](#_Toc55911747)

[3) filter 20](#_Toc55911748)

[4) take 21](#_Toc55911749)

[5) skip 22](#_Toc55911750)

[2.5 에러 처리 22](#_Toc55911751)

[1) 에러 신호 처리 22](#_Toc55911752)

[2) 에러 발생하면 기본 값 사용 23](#_Toc55911753)

[2.6 쓰레드 스케쥴링 24](#_Toc55911754)

[2.7 병렬 실행 24](#_Toc55911755)

# Project Reactor 개요

## 1.1 Project Reactor

Reactor는 Pivotal의 오픈소스 프로젝트로, JVM 위에서 동작하는 논블럭킹 애플리케이션을 만들기 위한 라이브러리이다. Reactor는 Reactive Streams의 구현체이기도 하고, Spring Framework 5부터 Reactive Programming 위해 지원되는 라이브러리이다.

여기서 ‘Reactive Programming’, ‘Reactive Streams’ 란 무엇인지 다음 장에서 설명하겠다.

## 1.2 Reactive Programming

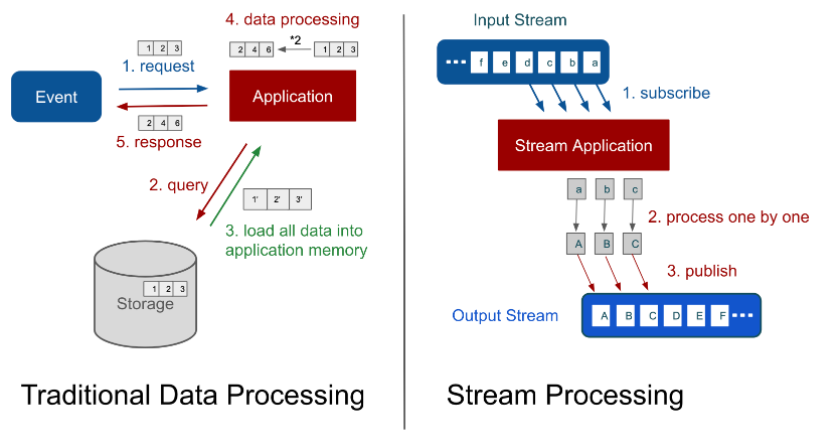
Reactive Programming은 데이터 흐름을 먼저 정의하고 데이터가 변경되었을 때 연관되는 함수나 수식이 업데이트되는 방식이다.

즉, 비동기 데이터 Stream으로 Non-Blocking 어플리케이션을 구현하는 프로그래밍을 말한다. Stream으로 프로그래밍 한다는 것은 함수형 처리가 가능해 진다는 것이다. 다시 말해 filter 하거나 map 할 수도 있고 여러 형태로 편하게 사용가능 해 진다는 말이 된다.

## 1.3 Reactive Streams

Reactive Streams (http://www.reactive-streams.org/)는 JVM용 스트림 지향 라이브러리의 표준 및 스펙이다. 홈페이지에선 논블로킹(Non-blocking) 백 프레셔(back pressure)를 이용한 비동기 데이터 처리의 표준이라고 말하고 있다. 다음으로 ‘스트리밍 처리’, ‘비동기(asynchronous) 방식’, ‘백 프레셔’라는 각각의 단어가 의미하는 바를 설명하겠다.

### 1) 스트리밍 처리



#### 전통적인 데이터 처리 방식

* 특징

데이터 처리 요청이 오면 페이로드(payload)를 모두 애플리케이션의 메모리에 저장한 후에 다음 처리를 해야 한다. 추가로 필요한 데이터도 저장소에서 조회하여 메모리에 적재해야 한다.

* 문제점

전달된 데이터는 물론 저장소에서 조회한 데이터까지 모든 데이터가 애플리케이션의 메모리에 적재되어야만 응답 메시지를 만들 수 있다는 것이다. 만약 필요한 데이터의 크기가 메모리 용량보다 크다면 ‘out of memory’ 에러가 발생하게 된다.

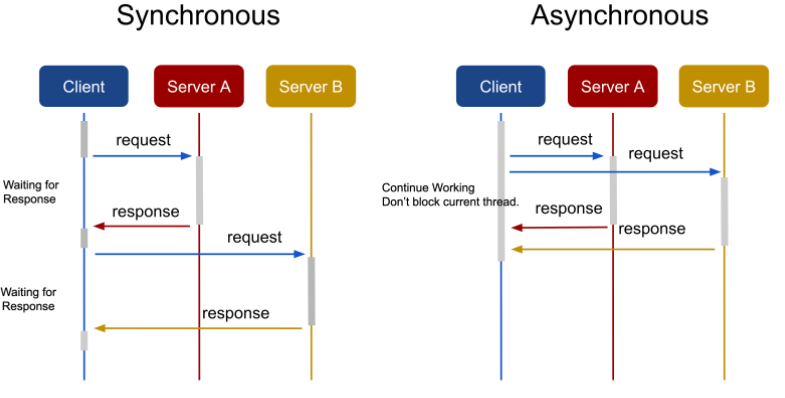
#### 스트림 처리 방식

* 특징

스트림 처리 방식은 크기가 작은 시스템 메모리로도 많은 양의 데이터를 처리할 수 있다. 입력 데이터에 대한 파이프 라인을 만들어 데이터가 들어오는 대로 물 흐르듯이 구독(subscribe)하고, 처리한 뒤, 발행(publish)까지 한 번에 연결하여 처리할 수 있다. 이렇게 하면 서버는 많은 양의 데이터도 탄력적으로 처리할 수 있다.

### 2) 비동기 방식

비동기 방식은 동기(synchronous) 방식과 비교하며 살펴보겠다. 아래 그림은 동기 방식과 비동기 방식의 처리 과정을 나타낸 그림이다.



#### 동기 방식

* 특징

클라이언트가 서버에 요청을 보내면 응답을 받기 전까지 블로킹(blocking)된다. 블로킹된다는 것은 현재 스레드(thread)가 다른 일을 하지 못하고 기다린다는 것을 의미한다.

* 문제점

두 개의 요청을 A와 B 서버로 보내면, A의 응답이 끝나고 나서야 B로 요청을 보낼 수 있다.

#### 비동기 방식

* 특징

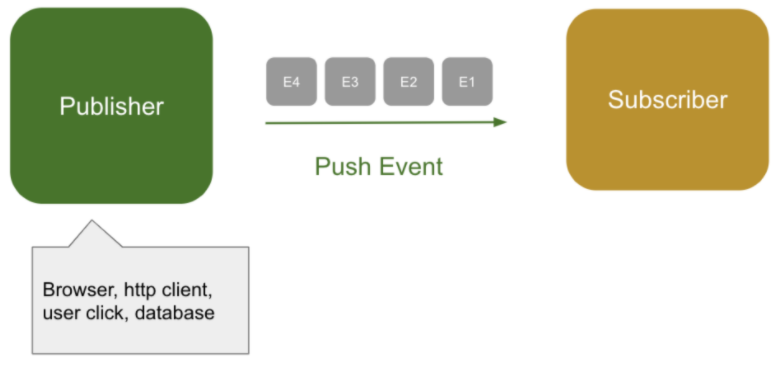
현재 스레드가 블로킹되지 않기 때문에 다른 일을 계속할 수 있다. A에게 요청을 보낸 뒤 다른 일을 처리할 수도 있고, 혹은 B에게 또 다른 요청을 보낼 수도 있다. 동기 방식과 비교하여 비동기 방식의 장점을 정리하면 아래와 같다.

1. 빠른 속도 : 두 개의 요청을 동시에 보내기 때문에 더 빠른 응답 속도를 보여줄 것이다.
2. 적은 리소스 사용 : 현재 스레드가 블로킹되지 않고 다른 업무를 처리할 수 있어서 더 적은 수의 스레드로 더 많은 양의 요청을 처리할 수 있다.

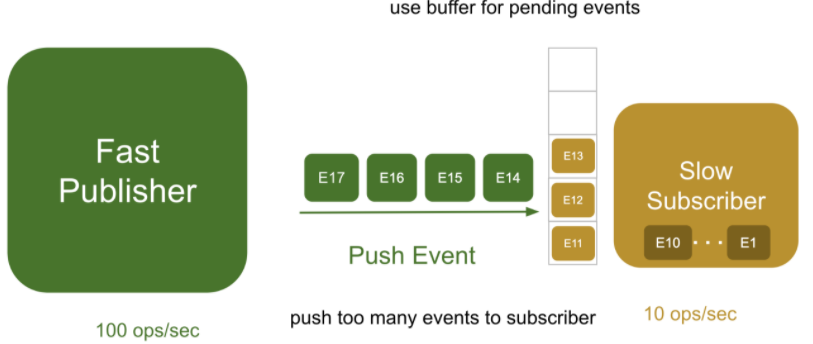
### 3) 백 프레셔

백 프레셔에 대해 알아보기 전 푸시(push) 방식, 그리고 풀(pull) 방식에 대해서 알아보겠다.

#### 푸시 방식



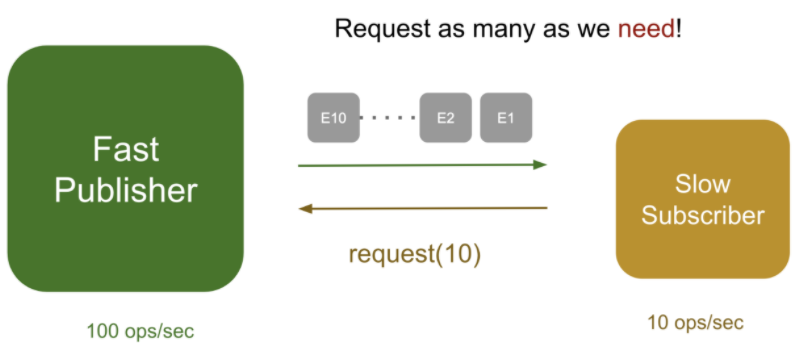
Project Reactor는 OOP에서 Observer 디자인 패턴에서 발전한 것이다. Observer 패턴에서 데이터를 전달할 때, 발행자(publisher)가 구독자(subscriber)에게 데이터를 밀어 넣는데 발행자는 구독자의 상태를 고려하지 않는다. 만약 발행자가 1초 동안 100개의 메시지를 보내는데 구독자는 1초에 10개밖에 처리하지 못한다면 큐(queue)를 이용해서 대기 중인 이벤트를 저장해야 한다.



서버가 가용할 수 있는 메모리는 한정되어 있다. 만약 초당 100개의 메모리를 계속 푸시한다면 버퍼를 다 사용해버려서 오버플로(overflow)가 발생한다. 이를 해결하려면 발행자가 데이터를 전달할 때 구독자가 필요한 만큼만 전달해야 한다. (백 프레셔의 기본 원리).

#### 풀 방식

풀 방식에선 구독자가 10개를 처리할 수 있다면 발행자에게 10개만 요청한다. 발행자는 요청받은 만큼만 전달하면 되고, 구독자는 더 이상 ‘out of memory’ 에러를 걱정하지 않아도 된다.



여기서 좀 더 탄력적으로 적용하여, 구독자가 이미 8개의 일을 처리하고 있다면 추가로 2개만 더 요청하여 자신이 현재 처리 가능한 범위 내에서만 메시지를 받게 할 수 있다. 풀 방식에선 이렇게 전달되는 모든 데이터의 크기를 구독자가 결정한다. 이런 다이나믹 풀 방식의 데이터 요청을 통해서 구독자가 수용할 수 있는 만큼만 데이터를 요청하는 방식이 백 프레셔이다.

### 4) 리액티브 스트림즈 API

#### 구성요소

Reactive Streams는 그럴듯한 단어로 설명되어 뭔가 복잡할 것 같지만, 실제 내부는 Publisher, Subscriber, Subscription, Processor 네 가지 간단한 인터페이스들의 조합으로 구성되어 있다.

* Publisher

Publisher(생산자)는 Subscriber(구독자)를 받아들이는 메서드를 가진다.

|  |
| --- |
| public interface Publisher<T> {  public void subscribe(Subscriber<? super T> s);  } |

* Subscriber

Subscriber(구독자)는 Subscription을 등록하고 Subscription에서 오는 신호에 따라서 동작한다. 구독자에게 오는 신호로는 onNext, onError, onComplete가 있다.

|  |
| --- |
| public interface Subscriber<T> {  public void onSubscribe(Subscription s);  public void onNext(T t);  public void onError(Throwable t);  public void onComplete();  } |

각 메서드는 다음과 같다.

1. onSubscribe(Subscription s): 구독을 하면 Publisher와 연동된 Subscription을 받는다. 전달받은 Subscription을 이용해서 Publisher에 데이터를 요청한다.
2. onNext(T t): Publisher가 next 신호를 보내면 호출된다.
3. onError(Throwable t): Publisher가 error 신호를 보내면 호출된다.
4. onComplete(): Publisher가 complete 신호를 보내면 호출된다.

* Subscription

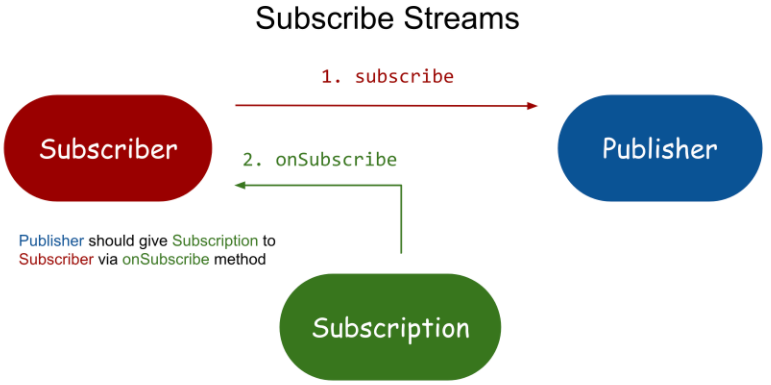
Subscription은 Publisher와 Subscriber 사이에서 중계하는 역할을 한다. request 메서드는 Subscriber가 Publisher에게 데이터를 요청하는 갯수이며 cancel은 구독을 취소하겠다는 의미이다

|  |
| --- |
| public interface Subscription {  public void request(long n);  public void cancel();  } |

* Processor

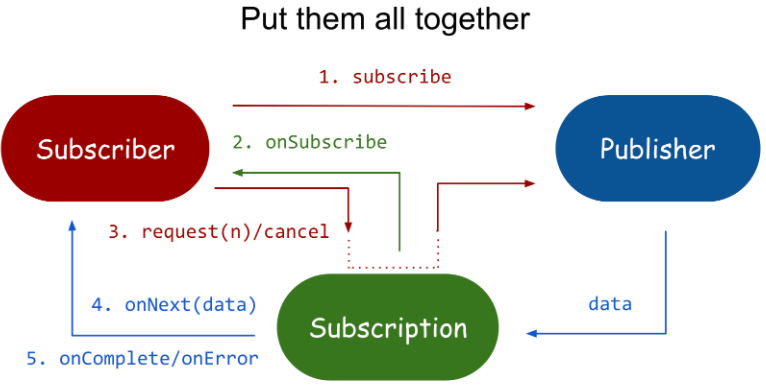
|  |
| --- |
| public interface Processor<T, R> extends Subscriber<T>, Publisher<R> {} |

#### Reactive Streams API 사용 흐름



1. Subscriber가 subscribe 함수를 사용해 Publisher에게 구독을 요청한다.

2. Publisher는 onSubscribe 함수를 사용해 Subscriber에게 Subscription을 전달한다.



3. 이제 Subscription은 Subscriber와 Publisher 간 통신의 매개체가 된다. Subscriber는 Publisher에게 직접 데이터 요청을 하지 않는다. Subscription의 request 함수를 통해 Publisher에게 전달한다.

4. Publisher는 Subscription을 통해 Subscriber의 onNext에 데이터를 전달하고, 작업이 완료되면 onComplete, 에러가 발생하면 onError 시그널을 전달한다.

5. Subscriber와 Publisher, Subscription이 서로 유기적으로 연결되어 통신을 주고받으면서 subscribe부터 onComplete까지 연결되고, 이를 통해 백 프레셔가 완성된다.

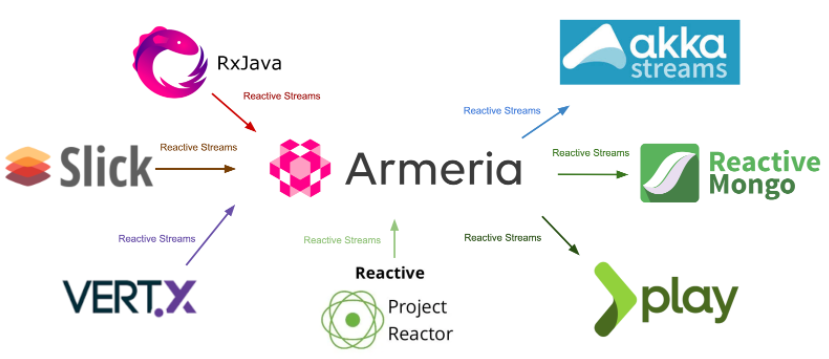
#### Reactive Stream 구현체 종류

Reactive Streams API는 위에서 설명한 인터페이스가 전부이고 따로 구현체가 없다. 그렇다면 직접 만들어서 사용해야 할까? 단순한 인터페이스와는 달리 구현 시 따라야 하는 규칙이 복잡하기 때문에 직접 구현하는 것보다는 이미 잘 만들어져서 검증까지 받은 구현체를 사용하는 게 좋다.

Reactive Streams에는 다양한 구현체가 존재한다. 각각의 구현체는 서로 특성이 조금씩 다르기 때문에 상황에 따라, 필요에 맞게 골라서 사용할 수 있다.

* 순수하게 스트림 연산 처리만 필요하다면 RxJava나 Reactor Core, Akka Streams 등을 사용하면 된다.
* 저장소의 데이터를 Reactive Streams로 조회하고 싶다면 ReactiveMongo나 Slick 등을 사용하면 된다.
* 웹 프로그래밍과 연결된 Reactive Streams가 필요하다면 Armeria와 Vert.x, Play Framework, Spring WebFlux를 활용할 수 있다.

이렇게 각각 특성이 다른 구현체들은 모두 Reactive Streams를 통해서 서로 통신할 수 있다.



# Project Reactor 사용법

## 2.1 구조

#### Flux, Mono

* 특징

Flux, Mono는 Project Reactor에서 제공하는 Reactive Stream의 Publisher 인터페이스 구현체이다. 그리고 둘 다 Reactor에서 제공하는 풍부한 연산자들(operators)의 조합을 통해 스트림을 표현할 수 있다.

* 차이점

Mono는 0-1개의 결과만을 처리하기 위한 객체이고, Flux는 0-N개인 여러 개의 결과를 처리하는 객체이다.

Reactive Stream은 Publisher를 이용해서 스트림을 정의하며 Subscriber를 이용해서 발생한 신호를 처리한다. Subscriber가 Publisher로부터 신호를 받는 것을 구독이라고 한다. 다음 코드는 Project Reactor에서 Flux를 구독하는 예제이다.

|  |
| --- |
| Flux<Integer> seq = Flux.just(1, 2, 3); // Integer 값을 발생하는 Flux 생성  seq.subscribe(value -> System.out.println("데이터 : " + value)); // 구독 |

실제 값 발생은 구독(subscription) 시점에 이뤄진다. 위 코드는 Flux.subscribe(Consumer) 메서드를 이용해서 구독한다. 이 경우 Flux가 발생한 신호를 Consumer가 받아서 처리한다. 위 코드는 수신한 데이터를 콘솔에 출력하므로 위 코드를 실행하면 다음과 같은 결과가 출력된다.

|  |
| --- |
| 데이터 : 1  데이터 : 2  데이터 : 3 |

## 2.2 환경 구성

#### 프로젝트 리액터 사용을 위한 메이븐 설정

리액터는 reactor-core, reactor-netty, reactor-extra, reactor-adapter 등 다양한 모듈로 구성되어 있다. 각 모듈의 버전을 맞추기 위해 리액터는 메이븐 BOM(Bill Of Materials)을 제공한다. 이 글에서는 reactor-core 의존만 사용하긴 하지만 BOM을 포함한 의존 설정을 사용해보자. 메이븐 의존 설정은 다음과 같다.

|  |
| --- |
| <dependencyManagement>      <dependencies>          <dependency>              <groupId>io.projectreactor</groupId>              <artifactId>reactor-bom</artifactId>              <version>Bismuth-SR9</version>              <type>pom</type>              <scope>import</scope>          </dependency>      </dependencies>  </dependencyManagement>  <dependencies>      <dependency>          <groupId>io.projectreactor</groupId>          <artifactId>reactor-core</artifactId>      </dependency>      <dependency>          <groupId>org.slf4j</groupId>          <artifactId>slf4j-api</artifactId>          <version>1.7.12</version>      </dependency>      <dependency>          <groupId>ch.qos.logback</groupId>          <artifactId>logback-classic</artifactId>          <version>1.2.3</version>      </dependency>  </dependencies> |

#### gradle

|  |
| --- |
| repositories {  mavenCentral()  }  dependencies {  compile "io.projectreactor:reactor-core:3.4.0"  testCompile "io.projectreactor:reactor-test:3.4.0"  } |

## 2.3 시퀀스 생성

### 1) just

시퀀스를 생성하는 가장 쉬운 방법은 just()를 사용하는 것이다. just() 메서드는 시퀀스로 사용할 데이터가 이미 존재할 때 사용한다. Flux, Mono 모두 just() 메서드를 사용가능하다.

* 사용법
* Flux

|  |
| --- |
| // 1, 2, 3 데이터를 차례대로 발생하고 complete 신호를 발생한다.  Flux<Integer> seq = Flux.just(1, 2, 3);  // 발생할 데이터를 주지 않으면 complete 신호만 발생  Flux<Integer> seq = Flux.just(); |

* Mono

|  |
| --- |
| // Mono는 1개 값만 생성하므로 데이터를 1개만 받음  Mono<Integer> seq = Mono.just(1);  // null을 값으로 받으면 값이 없는 Mono  Mono<Integer> seq1 = Mono.justOrEmpty(null); // complete 신호  Mono<Integer> seq2 = Mono.justOrEmpty(1); // next(1) 신호- complete 신호  // Optional을 값으로 받음  Mono<Integer> seq3 = Mono.justOrEmpty(Optional.empty()); // complete 신호  Mono<Integer> seq4 = Mono.justOrEmpty(Optional.of(1)); // next(1) 신호 - complete 신호 |

* Flux 예제

String 데이터를 전달하는 Flux 를 생성

|  |
| --- |
| @Test  public void test\_flux\_just\_consumer(){  List<String> names = new ArrayList<>();  Flux<String> flux = Flux.just("AAA", "BBB").log();  flux.subscribe(s->{  System.out.println("시퀀스 수신:"+s);  names.add(s);  });  Assert.assertEquals(names, Arrays.asList("AAA", "BBB"));  } |

* Flux는 Subscribe가 실행하기 전까지는 어떤 일도 발생하지 않는다. Publisher (발행자)는 구독이 되었을 경우에만 데이터를 Subscriber(구독자)에게 전달한다. Subscriber(구독자)가 Publisher 에 구독을 하는 과정은, Publisher(발행자)에 정의된 subscribe() 메서드를 사용한다.
* 이때, 매개변수로 Consumer 함수를 전달할 수 있는데, Consumer 함수는 데이터 전달을 해서 Subscriber의 onNext 이벤트가 발생을 했을때 실행되는 함수이다.
* 이 예제는 Consumer함수에 데이터를 하나씩 받을 때마다 System.out.print 를 실행해서 데이터 수신을 확인하고, 전달 받은 데이터를 List<String> 에 add하는 로직이다.

(설명) 시퀀스 입력 순서대로 subscribe 된다. subscribe() 메서드를 실행하면 처음으로 onSubscribe() 메서드가 실행된다. onSubscribe()에 publisher에서 생성한 subscription이 매개변수로 전달되고 subscription은 request()메서드를 사용해서 publisher에 데이터 1개 전달을 요청한다. 요청받은 publisher가 데이터를 1개 보내면 Subscriber에서 onNext()메서드를 실행한다. onNext() 메서드는 다음데이터를 요청한다. 최초 구독 이후에 데이터를 요청하려면 onNext()메서드가 필요하다.

* Mono 예제

|  |
| --- |
| @Test  public void test\_mono\_just(){  List<Signal<Integer>> signals = new ArrayList<>(4);  final Integer[] result = new Integer[1];  Mono<Integer> mono = Mono.just(1).log()  .doOnEach(integerSignal -> {  signals.add(integerSignal);  System.out.println("Signal... : " + integerSignal);  });  mono.subscribe(integer -> result[0] = integer);  Assert.assertEquals(signals.size(), 2);  Assert.assertEquals(signals.get(0).getType().name(),"ON\_NEXT");  Assert.assertEquals(signals.get(1).getType().name(),"ON\_COMPLETE");  Assert.assertEquals(result[0].intValue(), 1);  } |

### 2) range

Flux.range() 메서드를 사용하면 순차적으로 증가하는 Integer를 생성하는 Flux를 생성할 수 있다.

* 사용법

|  |
| --- |
| Flux<Integer> seq = Flux.range(11, 5); |

* Flux 예제

|  |
| --- |
| @Test  public void test\_flux\_range(){  List<Integer> list = new ArrayList<>();  Flux<Integer> flux = Flux.range(1,5).log();  flux.subscribe(list::add);  Assert.assertEquals(5,list.size());  Assert.assertEquals(1,(int)list.get(0));  Assert.assertNotEquals(5,(int)list.get(0));  } |

### 3) fromArray

fromArray 메서드를 사용하면 Array를 사용해서 Flux 를 생성할 수 있다.

* Flux 예제

|  |
| --- |
| @Test  public void test\_flux\_fromArray(){  List<String> names = new ArrayList<>();  Flux<String> flux = Flux.fromArray(new String[]{"A", "B", "C"}).log();  flux.subscribe(names::add);  Assert.assertEquals(names, Arrays.asList("A", "B", "C"));  } |

4) empty

empty 메서드를 사용하면 아무값도 전달하지 않는, Empty Flux를 만들 수 있다.

* Flux 예제

|  |
| --- |
| @Test  public void test\_flux\_empty(){  List<String> names = new ArrayList<>();  Flux<String> flux = Flux.empty();  flux.subscribe(names::add);  Assert.assertEquals(names.size(), 0);  } |

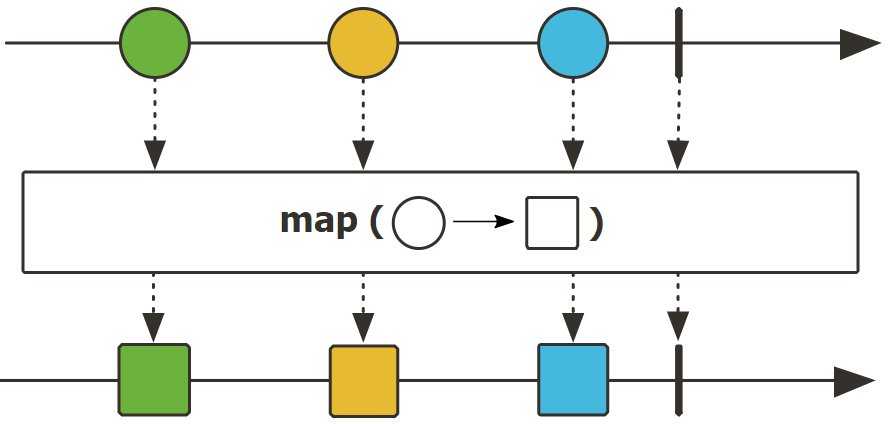
* Mono 예제

|  |
| --- |
| @Test  public void test\_mono\_empty(){  List<Signal<Integer>> signals = new ArrayList<>(4);  Mono<Integer> mono = Mono.<Integer>empty()  .doOnEach(signals::add);  mono.subscribe();  Assert.assertEquals(signals.size(), 1);  Assert.assertTrue(signals.get(0).isOnComplete());  } |

## 2.4시퀀스 변환

### 1) map

각각의 데이터를 one-to-one 으로 변환

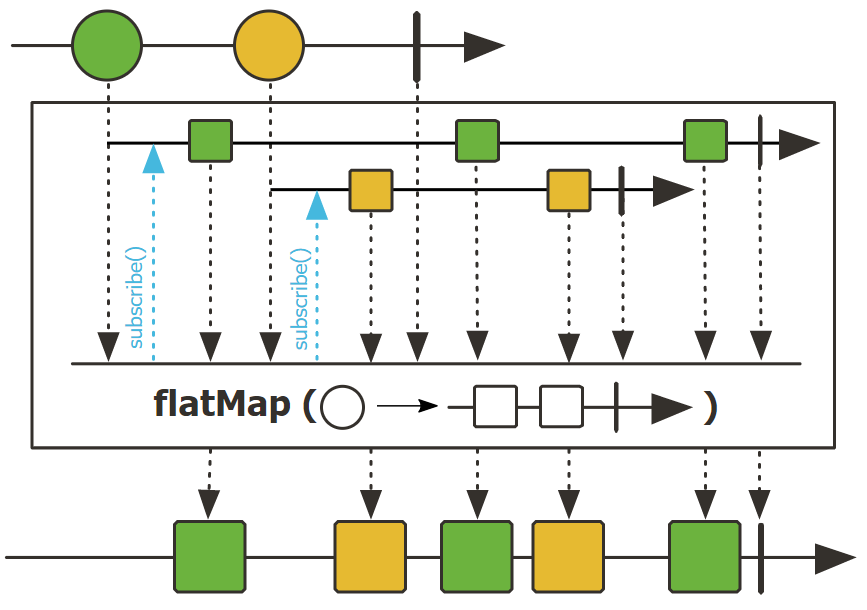


* 예제

|  |
| --- |
| @Test  public void flux\_map(){  List<String> names = new ArrayList<>();  Flux<String> flux = Flux.just("EDDY", "IRENE").log();  flux.map(String::toLowerCase)  .subscribe(names::add);  Assert.assertEquals(2,names.size());  Assert.assertEquals("eddy",names.get(0));  Assert.assertEquals("irene",names.get(1));  Assert.assertNotEquals("EDDY",names.get(0));  } |

### 2) flat map

각각의 데이터를 one-to-n 으로 변환

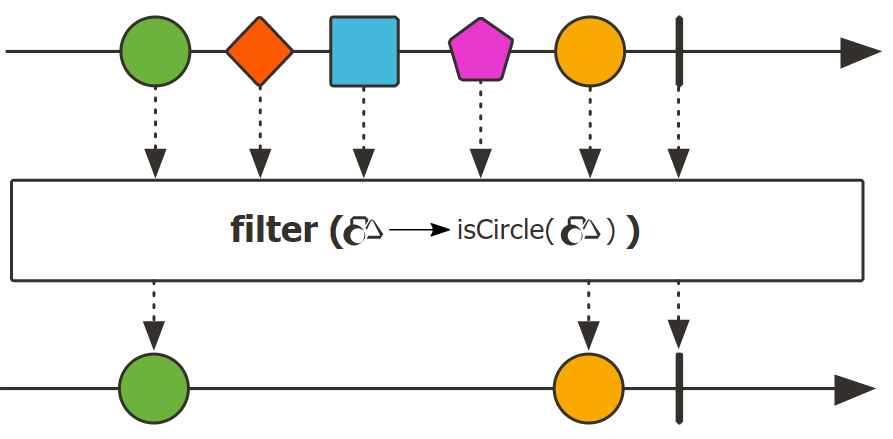


* 예제

|  |
| --- |
| @Test  public void flux\_flatmap\_integer(){  List<Integer> integerList = new ArrayList<>();  Flux<Integer> integerFlux = Flux.just(1, 2, 3).log();  integerFlux.flatMap(i -> Flux.range(0, i))  .subscribe(integerList::add);  // expected : 0, 0, 1, 0, 1, 2  List<Integer> expected = Arrays.asList(0,0,1,0,1,2);  Assert.assertEquals(6,integerList.size());  //TODO:assertArrays..  Assert.assertEquals(expected, integerList);  } |

### 3) filter

filter()를 이용해서 시퀀스가 생성한 데이터를 걸러낼 수 있다. filter()에 전달한 함수의 결과가 true인 데이터만 전달하고 false인 데이터는 발생하지 않는다.



* 사용법

다음은 1부터 10 사이의 숫자 중에서 2로 나눠 나머지가 0인 (즉 짝수인) 숫자만 걸러내는 예를 보여준다.

|  |
| --- |
| Flux.range(1, 10)  .filter(num -> num % 2 == 0)  .subscribe(x -> System.out.print(x + " -> ")); |

실행 결과 : 2 -> 4 -> 6 -> 8 -> 10 →

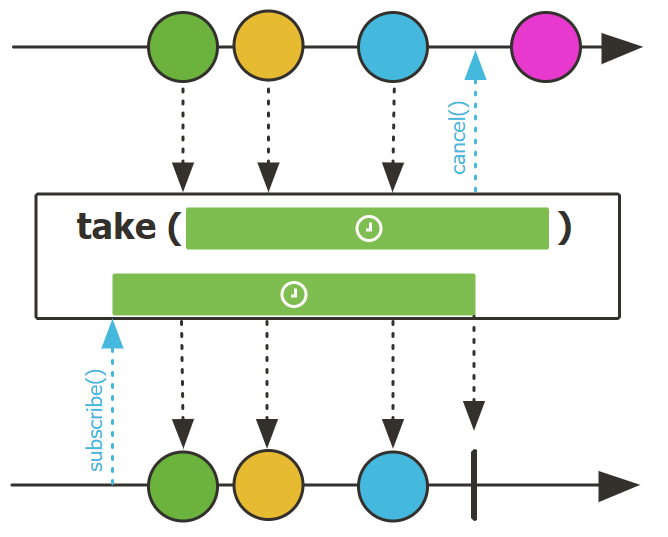
* 예제

Flux<String>을 생성해서 subscribe를 한다. 이 때, "blue"가 아닌 데이터만 전달하도록 filter를 추가했다.

|  |
| --- |
| @Test  public void flux\_filter(){  List<String> colors = new ArrayList<>();  Flux<String> flux = Flux.just("blue", "green", "orange", "purple").log();  flux.filter(color->!color.equals("blue"))  .subscribe(colors::add);  Assert.assertEquals(3, colors.size());  Assert.assertTrue(colors.contains("green"));  Assert.assertFalse(colors.contains("blue"));  } |

### 4) take

시퀀스 순서대로 특정 개수의 데이터만 전달



* 예제

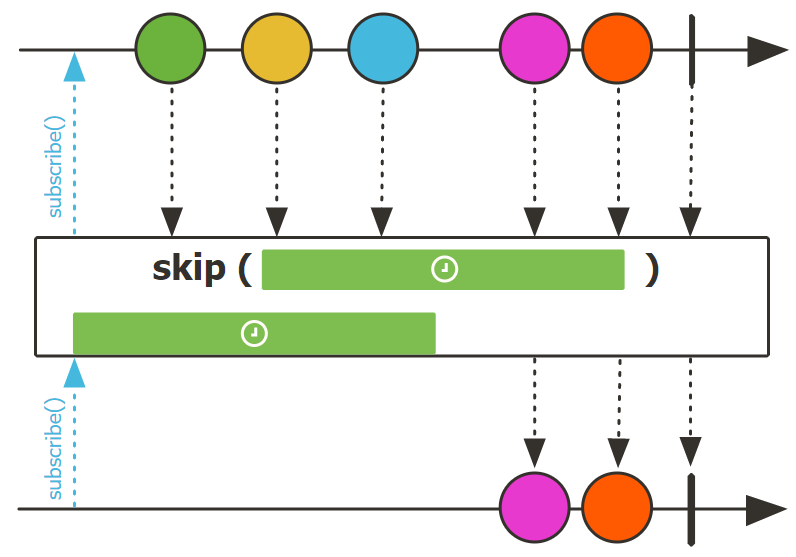
Flux<String> 시퀀스에서 2개의 데이터만 subscribe 한다.

* 예제

|  |
| --- |
| @Test  public void flux\_take(){  List<String> colors = new ArrayList<>();  Flux<String> flux = Flux.just("blue", "green", "orange", "purple").log();  flux.take(2).subscribe(colors::add);  Assert.assertEquals(2, colors.size());  Assert.assertTrue(colors.contains("green"));  Assert.assertFalse(colors.contains("orange"));  } |

### 5) skip

시퀀스 순서대로 특정 개수의 데이터를 건너뛴다.



* 예제

|  |
| --- |
| @Test  public void flux\_skip(){  List<String> colors = new ArrayList<>();  Flux<String> flux = Flux.just("blue", "green", "orange", "purple").log();  flux.skip(3).subscribe(colors::add);  Assert.assertEquals(1, colors.size());  Assert.assertTrue(colors.contains("purple"));  Assert.assertFalse(colors.contains("blue"));  } |

## 2.5 에러 처리

시퀀스는 데이터를 발생하는 과정에서 에러를 발생할 수 있다. 리액터는 에러를 처리하는 여러 방법을 제공한다. 참고로 에러 신호는 종료 신호이다. 따라서 에러 신호가 발생하면 시퀀스는 종료되고 더 이상 데이터를 발생하지 않는다.

### 1) 에러 신호 처리

에러 신호가 발생하면 Subscriber의 onError 메서드가 호출된다. 이 메서드를 구현한 Subscriber를 이용해서 구독을 하면 에러 신호를 알맞게 처리할 수 있다. 또한 에러 처리를 위한 Consumer를 파라미터로 갖는 subscribe() 메서드를 사용해서 익셉션을 처리할 수 있다. 다음 코드는 에러 처리를 위한 Consumer를 파라미터로 갖는 subscribe() 메서드의 사용 예를 보여준다.

* 예제

아래 예제는 1부터 10개의 값을 발생하는데 값이 5이면 익셉션을 발생하는 시퀀스를 생성한다. subscribe() 메서드는 3개의 인자를 갖는데 차례대로 next, error, complete 신호를 처리한다. 실행 결과는 다음과 같다.

|  |
| --- |
| @Test  public void test\_error01(){  Flux.range(1, 10)  .map(x -> {  if (x == 5) throw new RuntimeException("exception"); // 에러 발생  else return x;  })  .subscribe(  i -> System.out.println(i), // next 신호 처리  ex -> System.err.println(ex.getMessage()), // error 신호 처리  () -> System.out.println("complete") // complete 신호 처리  );  } |

실행결과

에러 신호는 종료 신호이므로 익셉션 발생 이후에 더 이상 next 신호가 발생하지 않는 것을 확인할 수 있다.

|  |
| --- |
| 1  2  3  4  exception |

### 2) 에러 발생하면 기본 값 사용

에러가 발생할 때 에러 대신에 특정 값을 발생하고 싶다면 onErrorReturn() 메서드를 사용한다. 이 메서드의 사용 예는 다음과 같다.

* 예제

|  |
| --- |
| @Test  public void test\_error02(){  Flux<Integer> seq = Flux.range(1, 10)  .map(x -> {  if (x == 5) throw new RuntimeException("exception");  else return x;  })  .onErrorReturn(-1);  seq.subscribe(System.out::println);  } |

## 2.6 쓰레드 스케쥴링

리액터는 비동기 실행을 강제하지 않는다.

* 예제

|  |
| --- |
| @Test  public void test01(){  Logger logger = LoggerFactory.getLogger(ThreadSchedulingTests.class);  Flux.range(1, 3)  .map(i -> {  logger.info("map {} to {}", i, i + 2);  return i + 2;  })  .flatMap(i -> {  logger.info("flatMap {} to Flux.range({}, {})", i, 1, i);  return Flux.range(1, i);  })  .subscribe(i -> logger.info("next " + i));  } |

* 실행 결과

위 코드에서 logger는 쓰레드 이름을 남기도록 설정한 로거다. 위 코드를 실행하면 다음과 같은 결과를 출력한다.

|  |
| --- |
| 15:08:03.428 [Test worker] INFO com.example.demo.ThreadSchedulingTests - map 1 to 3  15:08:03.428 [Test worker] INFO com.example.demo.ThreadSchedulingTests - flatMap 3 to Flux.range(1, 3)  15:08:03.428 [Test worker] INFO com.example.demo.ThreadSchedulingTests - next 1  15:08:03.428 [Test worker] INFO com.example.demo.ThreadSchedulingTests - next 2  15:08:03.428 [Test worker] INFO com.example.demo.ThreadSchedulingTests - next 3  15:08:03.428 [Test worker] INFO com.example.demo.ThreadSchedulingTests - map 2 to 4  15:08:03.428 [Test worker] INFO com.example.demo.ThreadSchedulingTests - flatMap 4 to Flux.range(1, 4)  15:08:03.428 [Test worker] INFO com.example.demo.ThreadSchedulingTests - next 1  15:08:03.428 [Test worker] INFO com.example.demo.ThreadSchedulingTests - next 2  15:08:03.428 [Test worker] INFO com.example.demo.ThreadSchedulingTests - next 3  15:08:03.428 [Test worker] INFO com.example.demo.ThreadSchedulingTests - next 4  15:08:03.428 [Test worker] INFO com.example.demo.ThreadSchedulingTests - map 3 to 5  15:08:03.428 [Test worker] INFO com.example.demo.ThreadSchedulingTests - flatMap 5 to Flux.range(1, 5)  15:08:03.428 [Test worker] INFO com.example.demo.ThreadSchedulingTests - next 1  15:08:03.428 [Test worker] INFO com.example.demo.ThreadSchedulingTests - next 2  15:08:03.428 [Test worker] INFO com.example.demo.ThreadSchedulingTests - next 3  15:08:03.428 [Test worker] INFO com.example.demo.ThreadSchedulingTests - next 4  15:08:03.428 [Test worker] INFO com.example.demo.ThreadSchedulingTests - next 5  BUILD SUCCESSFUL in 3s  5 actionable tasks: 3 executed, 2 up-to-date  오후 3:08:03: Tasks execution finished ':cleanTest :test --tests "com.example.demo.ThreadScheduling.test01"'. |

실행 결과를 보면 map(), flatMap(), subscribe()에 전달한 코드가 모두 Test worker 쓰레드에서 실행된 것을 알 수 있다. 즉 map 연산, flatMap 연산뿐만 아니라 subscribe를 이용한 구독까지 모두 Test worker 쓰레드가 실행한다.

#### publishOn을 이용한 신호 처리 쓰레드 스케줄링

publishOn() 메서드를 이용하면 next, complete, error신호를 별도 쓰레드로 처리할 수 있다. map(), flatMap() 등의 변환도 publishOn()이 지정한 쓰레드를 이용해서 처리한다. 다음 코드를 보자.

* 예제

publishOn()은 두 개의 인자를 받는다. 이 코드에서 첫 번째 인자인 Schedulers.newElastic("PUB")은 비동기로 신호를 처리할 스케줄러이다. 다양한 스케줄러가 존재하는데 이에 대해서는 뒤에서 다시 살펴본다. 일단 지금은 스케줄러가 별도 쓰레드를 이용해서 신호를 처리한다고 생각하면 된다.

두 번째 인자인 2는 스케줄러가 신호를 처리하기 전에 미리 가져올 (prefetch) 데이터 개수이다. 이는 스케줄러가 생성하는 비동기 경계 시점에 보관할 수 있는 데이터의 개수로 일종의 버퍼 크기가 된다.

위 코드를 실제로 실행하면 어떤 일이 벌어지는지 보자. 다음은 결과이다.

|  |
| --- |
| @Test  public void test02() throws InterruptedException {  Logger logger = LoggerFactory.getLogger(ThreadSchedulingTests.class);  CountDownLatch latch = new CountDownLatch(1);  Flux.range(1, 6)  .map(i -> {  logger.info("map 1: {} + 10", i);  return i + 10;  })  .publishOn(Schedulers.newElastic("PUB"), 2)  .map(i -> { // publishOn에서 지정한 PUB 스케줄러가 실행  logger.info("map 2: {} + 10", i);  return i + 10;  })  .subscribe(new BaseSubscriber<Integer>() {    @Override  protected void hookOnSubscribe(Subscription subscription) {  logger.info("hookOnSubscribe");  requestUnbounded();  }  @Override  protected void hookOnNext(Integer value) {  logger.info("hookOnNext: " + value); // publishOn에서 지정한 스케줄러가 실행  }  @Override  protected void hookOnComplete() {  logger.info("hookOnComplete"); // publishOn에서 지정한 스케줄러가 실행  latch.countDown();  }  });    latch.await();  } |

* 실행 결과

|  |
| --- |
| 15:16:22.280 [Test worker] INFO com.example.demo.ThreadSchedulingTests - hookOnSubscribe  15:16:22.280 [Test worker] INFO com.example.demo.ThreadSchedulingTests - map 1: 1 + 10  15:16:22.280 [Test worker] INFO com.example.demo.ThreadSchedulingTests - map 1: 2 + 10  15:16:22.280 [PUB-2] INFO com.example.demo.ThreadSchedulingTests - map 2: 11 + 10  15:16:22.280 [PUB-2] INFO com.example.demo.ThreadSchedulingTests - hookOnNext: 21  15:16:22.280 [PUB-2] INFO com.example.demo.ThreadSchedulingTests - map 2: 12 + 10  15:16:22.280 [PUB-2] INFO com.example.demo.ThreadSchedulingTests - hookOnNext: 22  15:16:22.280 [PUB-2] INFO com.example.demo.ThreadSchedulingTests - map 1: 3 + 10  15:16:22.280 [PUB-2] INFO com.example.demo.ThreadSchedulingTests - map 1: 4 + 10  15:16:22.280 [PUB-2] INFO com.example.demo.ThreadSchedulingTests - map 2: 13 + 10  15:16:22.280 [PUB-2] INFO com.example.demo.ThreadSchedulingTests - hookOnNext: 23  15:16:22.280 [PUB-2] INFO com.example.demo.ThreadSchedulingTests - map 2: 14 + 10  15:16:22.280 [PUB-2] INFO com.example.demo.ThreadSchedulingTests - hookOnNext: 24  15:16:22.280 [PUB-2] INFO com.example.demo.ThreadSchedulingTests - map 1: 5 + 10  15:16:22.280 [PUB-2] INFO com.example.demo.ThreadSchedulingTests - map 1: 6 + 10  15:16:22.280 [PUB-2] INFO com.example.demo.ThreadSchedulingTests - map 2: 15 + 10  15:16:22.280 [PUB-2] INFO com.example.demo.ThreadSchedulingTests - hookOnNext: 25  15:16:22.280 [PUB-2] INFO com.example.demo.ThreadSchedulingTests - map 2: 16 + 10  15:16:22.280 [PUB-2] INFO com.example.demo.ThreadSchedulingTests - hookOnNext: 26  15:16:22.280 [PUB-2] INFO com.example.demo.ThreadSchedulingTests - hookOnComplete  BUILD SUCCESSFUL in 3s  5 actionable tasks: 3 executed, 2 up-to-date  오후 3:16:22: Tasks execution finished ':cleanTest :test --tests "com.example.demo.ThreadScheduling.test02"'. |

최초에 2개를 미리 가져올 때를 제외하면 나머지는 모두 publishOn()으로 전달한 스케줄러의 쓰레드(쓰레드 이름이 "PUB"로 시작)가 처리하는 것을 알 수 있다.

## 2.7 병렬 실행

예제