Giới thiệu Reactor Core

1. Tổng Quan

Reactor Core là một thư viện Java 8 implement mô hình **Reactive Programming**. Nó được xây dựng dựa trên **Reactive Streams Specification** - một tiêu chuẩn để xây dựng ứng dụng Reactive.

Trong bài viết này, chúng ta sẽ đi từng bước nhỏ thông qua Reactor cho đến khi có cái nhìn toàn cảnh cũng như cách thực thi của **Reactor core**.

Maven Dependencies

Đây là thư viện của Reactor, chúng ta có thể lấy thư viện mới nhất tại đây

```
<dependency>
    <groupId>io.projectreactor</groupId>
    <artifactId>reactor-core</artifactId>
    <version>3.2.8.RELEASE</version>
</dependency>
```

2. Tạo ra một luồng dữ liệu

Để có một ứng dụng phản ứng (reactive), điều đầu tiên chúng ta cần phải làm là tạo ra một luồng dữ liệu. Không có dữ liệu này chúng ta sẽ không có bất cứ điều gì để phản ứng, đó là lý do tại sao đây là bước đầu tiên.

Reactor core cung cấp 2 loại dữ liệu cho phép chúng ta thực hiện điều này.

a) Flux

Cách đầu tiên đó là dùng **Flux. Flux** là một luồng có thể phát ra **0..n** phần tử. Ví dụ tạo đơn giản:

```
Flux<Integer> just = Flux.just(1,2,3,4);
```

b) Mono

Cách thứ hai đó là **Mono**. **Mono** là một luồng có thể phát ra **0..1** phần tử. Nó hoạt động gần giống hệ như Flux, chỉ là bị giới hạn không quá một phần tử. Ví dụ:

```
Mono<String> just = Mono.just("atomPtit");
```

Điều lưu ý rằng cả **Flux** và **Mono** đề được triển khai từ interface **Publisher**. Cả hai đều tuân thủ tiêu chuẩn **Reactive**, chúng ta có thể sử dụng interface như sau:

```
Publisher<String> just = Mono.just("foo");
```

c) Subscribe()

Hãy luôn ghi nhớ rằng: Không có gì xảy ra cho đến khi **subscribe()** .

Trong reactor, khi bạn viết một **Publisher**, dữ liệu không bắt đầu được bơm vào theo mặc định. Thay vào đó, bạn tạo một mô tả trừu tượng về quy định không đồng bộ của bạn(hỗ trợ tái sử dụng).

Để hiểu rõ luồng hoạt động hãy theo dõi qua ví dụ đơn giản sau.

```
<dependency>
    <groupId>io.projectreactor</groupId>
    <artifactId>reactor-core</artifactId>
    <version>3.2.8.RELEASE</version>
</dependency>
<dependency>
    <groupId>ch.qos.logback</groupId>
    <artifactId>logback-classic</artifactId>
    <version>1.1.3</version>
</dependency>
```

Chúng ta thêm thư viện logback. Điều này sẽ giúp chúng ta ghi nhật ký đầu ra của quá trình hoạt động reactor từ đó hiểu rõ hơn về luồng dữ liệu.

```
public class ReactorCode {
    public static void main(String[] args) {
        List<Integer> elements = new ArrayList<>();
        Flux.just(1, 2, 3, 4)
                .log()
                .subscribe(elements::add);
    }
}

/* OUTPUT:
23:02:16.996 [main] DEBUG reactor.util.Loggers$LoggerFactory - Using Slf4j
logging framework
23:02:17.014 [main] INFO  reactor.Flux.Array.1 - | onSubscribe([Synchronous
Fuseable] FluxArray.ArraySubscription)
23:02:17.017 [main] INFO  reactor.Flux.Array.1 - | request(unbounded)
23:02:17.018 [main] INFO  reactor.Flux.Array.1 - | onNext(1)
23:02:17.018 [main] INFO  reactor.Flux.Array.1 - | onNext(2)
23:02:17.018 [main] INFO  reactor.Flux.Array.1 - | onNext(3)
23:02:17.018 [main] INFO  reactor.Flux.Array.1 - | onNext(4)
23:02:17.019 [main] INFO  reactor.Flux.Array.1 - | onComplete()
*/
```

Hãy nhìn vào phần output, mọi thứ đều chạy trên main thread. Bây giờ chúng ta đi xem rõ từng dòng thực thi:

1. **onSubscribe**() - Điều này được gọi thi chúng ra đăng ký (subscriber()) luồng
2. **request(unbounded)** - Khi chúng ta gọi đăng ký, thì hàm này được chạy ngầm nhằm ý nghĩa tạo đăng ký. Trong trường hợp này chạy mặc định là unbounded (không giới hạn), nghĩa là nó yêu cầu mọi phần tử có sẵn.
3. **onNext()** - Hàm này được gọi cho mọi phần tử đơn.
4. **onComplete()** - Hàm này được gọi sau cùng sau khi nhận được phần tử cuối cùng. Trong thực có thể xảy ra các hàm khác như onError(), cái mà có thể được gọi khi xảy ra một exception.

3. So sánh với Streams Java 8

Có vẻ nhiều người vẫn đang nghĩ sự tương đồng với Stream trong Java 8:

```
List<Integer> collected = Stream.of(1, 2, 3, 4)
  .collect(toList());
```

Sự khác biết cốt lõi là Reactive là một hình **push** (đẩy) , trong khi Stream Java 8 là mô hình **pull** (kéo)

Streams Java 8 là terminal - kéo tất cả dữ liệu và trả về một kết quả. Với ***Reactive***, chúng ta có một luồng vô hạn đến từ một nguồi tài nguyên bên ngoài, với nhiều người **subscribe()**. Chúng ta cũng có thể làm những việc như kết hợp các luồng, điều tiết luồng và backpressure.

a) Backpressure

Trong ví dụ trên, người đăng ký nói với Publisher đẩy từng phần tử một. Điều này có thể trở nên quá tải cho người đăng ký phải tiêu thụ hết tất cả tài nguyên của nó.

Backpressure đơn giản chỉ là bảo với Publisher gửi cho nó ít dữ liệu hơn để ngăn chặn nó bị quá tải.

Ví dụ dưới đây, chúng ta sẽ yêu cầu chỉ gửi 2 phần từ cùng một lúc bằng cách sử dụng request ():

```
Flux.just(1, 2, 3, 4)
  .log()
  .subscribe(new Subscriber<Integer>() {
    private Subscription s;
    int onNextAmount;

    @Override
    public void onSubscribe(Subscription s) {
        this.s = s;
```

```
        s.request(2);
    }

    @Override
    public void onNext(Integer integer) {
        elements.add(integer);
        onNextAmount++;
        if (onNextAmount % 2 == 0) {
            s.request(2);
        }
    }

    @Override
    public void onError(Throwable t) {}

    @Override
    public void onComplete() {}
});

//OUTPUT
/*
23:31:15.395 [main] INFO  reactor.Flux.Array.1 - | onSubscribe([Synchronous
Fuseable] FluxArray.ArraySubscription)
23:31:15.397 [main] INFO  reactor.Flux.Array.1 - | request(2)
23:31:15.397 [main] INFO  reactor.Flux.Array.1 - | onNext(1)
23:31:15.398 [main] INFO  reactor.Flux.Array.1 - | onNext(2)
23:31:15.398 [main] INFO  reactor.Flux.Array.1 - | request(2)
23:31:15.398 [main] INFO  reactor.Flux.Array.1 - | onNext(3)
23:31:15.398 [main] INFO  reactor.Flux.Array.1 - | onNext(4)
23:31:15.398 [main] INFO  reactor.Flux.Array.1 - | request(2)
23:31:15.398 [main] INFO  reactor.Flux.Array.1 - | onComplete()
*/
```

Bây giờ chúng ta nhìn thấy hàm request() được gọi trước, tiếp theo đó là 2 hàm onNext() thực hiện, sau đó lại là request().

b) Concurrency

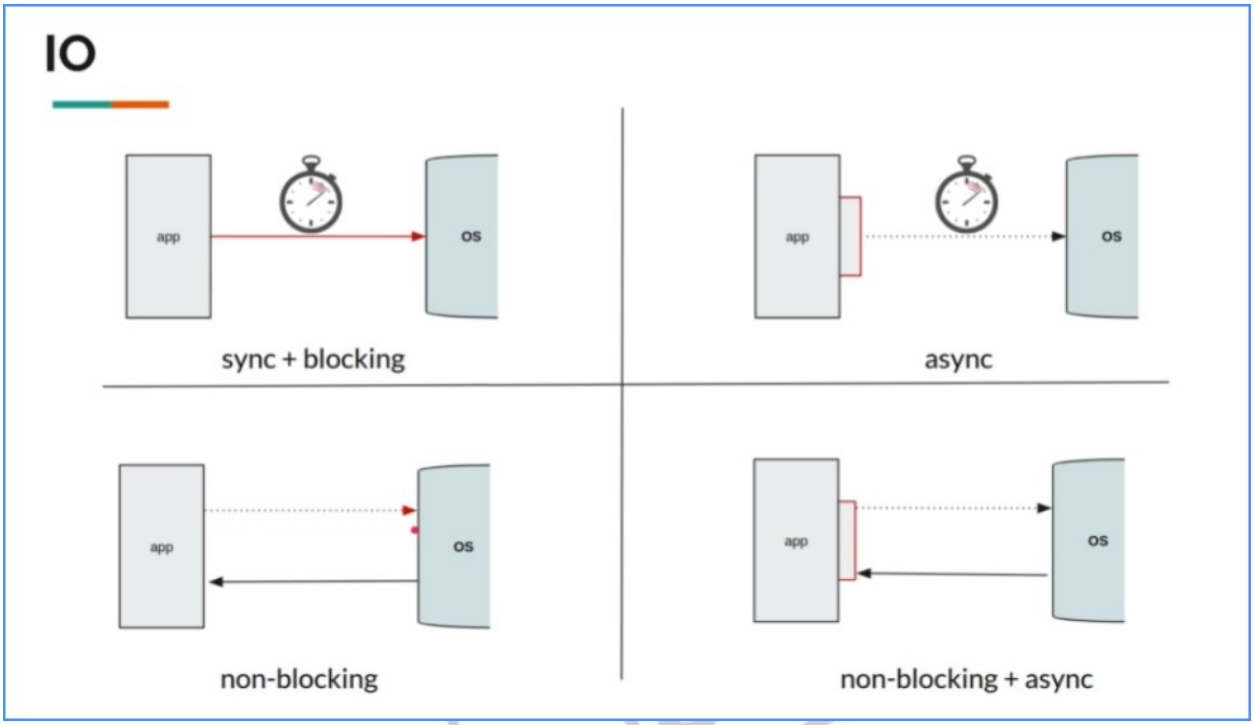
Tất cả các ví dụ trên chúng ta đều đang chạy trên một luồng chính. Tuy nhiên, chúng ta có thể kiểm soát luồng nào mà code của chúng ta chạy nếu chúng ta muốn. Các inteface **Scheduler** cung cấp một sự trừu tượng với asynchronous.

```
public class ReactorCode {
    public static void main(String[] args) {
        ExecutorService service = Executors.newFixedThreadPool(10);
        Flux.just(1, 2, 3, 4)
                .log()
                .subscribeOn(Schedulers.fromExecutorService(service))
                .subscribe();

        Flux.just(5, 6, 7, 8)
                .log()
                .subscribeOn(Schedulers.fromExecutorService(service))
                .subscribe();
    }
}

//OUTPUT
/*
23:48:02.972 [main] DEBUG reactor.util.Loggers$LoggerFactory - Using Slf4j
logging framework
23:48:02.996 [pool-1-thread-2] INFO  reactor.Flux.Array.2 - |
onSubscribe([Synchronous Fuseable] FluxArray.ArraySubscription)
23:48:02.996 [pool-1-thread-1] INFO  reactor.Flux.Array.1 - |
onSubscribe([Synchronous Fuseable] FluxArray.ArraySubscription)
23:48:03.000 [pool-1-thread-2] INFO  reactor.Flux.Array.2 - | request(unbounded)
23:48:03.000 [pool-1-thread-1] INFO  reactor.Flux.Array.1 - | request(unbounded)
23:48:03.001 [pool-1-thread-1] INFO  reactor.Flux.Array.1 - | onNext(1)
23:48:03.001 [pool-1-thread-2] INFO  reactor.Flux.Array.2 - | onNext(5)
23:48:03.001 [pool-1-thread-1] INFO  reactor.Flux.Array.1 - | onNext(2)
23:48:03.001 [pool-1-thread-2] INFO  reactor.Flux.Array.2 - | onNext(6)
23:48:03.001 [pool-1-thread-1] INFO  reactor.Flux.Array.1 - | onNext(3)
23:48:03.001 [pool-1-thread-2] INFO  reactor.Flux.Array.2 - | onNext(7)
23:48:03.001 [pool-1-thread-1] INFO  reactor.Flux.Array.1 - | onNext(4)
23:48:03.001 [pool-1-thread-2] INFO  reactor.Flux.Array.2 - | onNext(8)
23:48:03.002 [pool-1-thread-1] INFO  reactor.Flux.Array.1 - | onComplete()
23:48:03.002 [pool-1-thread-2] INFO  reactor.Flux.Array.2 - | onComplete()
*/
```

Ở đây chúng ta dùng [ExecutorService] link-ExecutorService, 2 luồng code thực hiện song song trên 2 thread khác nhau, điều mà đã chứng minh bằng output.

Chapter 1: Giới thiệu Webflux

1. I/O Model



- **Sync + Blocking:** Đây là cách thông thường mà hầu hết các chương trình hoạt động. Khi một tác vụ được gọi, chương trình chờ cho đến khi tác vụ đó hoàn thành trước khi di chuyển đến các tác vụ khác. Trong quá trình chờ đợi, chương trình bị block, không thể thực hiện bất kỳ tác vụ nào khác.
- **Async:** Trái ngược với cách tiếp cận đồng bộ, các tác vụ không chờ đợi lẫn nhau. Thay vào đó, chúng được gửi đi và chương trình tiếp tục thực hiện các tác vụ khác trong khi chờ đợi kết quả. Khi kết quả sẵn sàng, chương trình sẽ xử lý nó.
- **Non-blocking:** Đây là khi một tác vụ có thể được gọi mà không làm chương trình bị chặn. Ngược lại, chương trình tiếp tục thực hiện các tác vụ khác mà không cần phải đợi tác vụ đó hoàn thành. Chương trình sẽ kiểm tra lại sau để xem liệu tác vụ đã hoàn thành hay chưa.
- **Non-blocking + Async:** Kết hợp giữa non-blocking và async. Trong trường hợp này, chương trình tiếp tục thực hiện các tác vụ mà không cần chờ đợi tác vụ đang chạy, và đồng thời cũng không bị chặn. Khi tác vụ hoàn thành, nó sẽ được xử lý mà không làm chương trình bị chặn.

2. Reactive Streams trong Java

Reactive Streams là một khái niệm định nghĩa cơ chế để xử lý stream một cách bất đồng bộ (asynchronous) với non-blocking back pressure. Back pressure ở đây chúng ta có thể hiểu nôm

na có quá nhiều công việc phải xử lý cùng một lúc nên dẫn tới quá tải. Non-blocking back pressure có nghĩa là để tránh cái việc mà cùng một lúc quá nhiều công việc ập tới, trong 1 thời điểm chúng ta chỉ xử lý một số công việc nào đó thôi, khi nào xử lý xong những công việc đó thì mới nhận tiếp những công việc mới…Trong bài viết này, mình sẽ trình bày về khái niệm Reactive Streams trong Java các bạn nhé!

Ý tưởng chính của Reactive Streams là:

- Chúng ta có một Publisher để phát ra các thông tin.
- Chúng ta có một hoặc nhiều Subscribers tiếp nhận các thông tin mà Publisher phát ra.
- Và một Subscription làm cầu nối giữa Publisher và Subscribers.

Giữa Publisher và Subscribers kết nối với nhau theo nguyên tắc:

- Publisher sẽ add tất cả Subscribers mà nó cần notify.
- Subscribers sẽ nhận tất cả các thông tin được thêm vào Publisher.
- Subscribers sẽ request để yêu cầu và xử lý một hoặc nhiều thông tin từ Publisher theo kiểu bất đồng bộ thông qua đối tượng Subscription.
- Khi một Publisher có một thông tin để publish, thông tin này sẽ được gửi tới tất cả các Subscribers đang yêu cầu.

3. Observer Pattern là gì?

Observer Pattern là một trong những Pattern thuộc nhóm hành vi (Behavior Pattern). Nó định nghĩa mối phụ thuộc **một** – **nhiều** giữa các đối tượng để khi mà một đối tượng có sự thay đổi trạng thái, tất các thành phần phụ thuộc của nó sẽ được thông báo và cập nhật một cách tự động.

Observer có thể đăng ký với hệ thống. Khi hệ thống có sự thay đổi, hệ thống sẽ thông báo cho Observer biết. Khi không cần nữa, mẫu Observer sẽ được gỡ khỏi hệ thống.

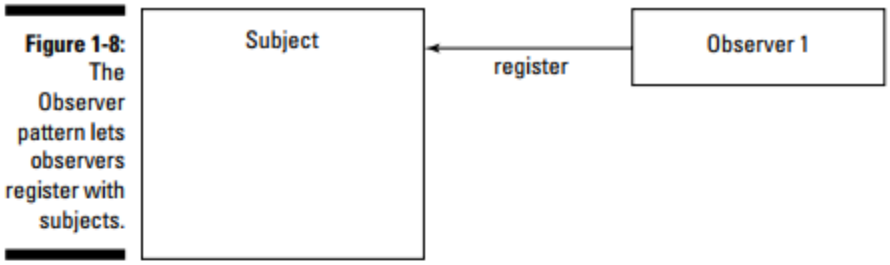


Figure 1-8: The Observer pattern lets observers register with subjects.

And another observer, Observer 2, can register itself as well, as shown in Figure 1-9.

Figure 1-9: More than one observer can register with a subject.

Now the subject is keeping track of two observers. When an event occurs, the subject notifies both observers. (See Figure 1-10.)

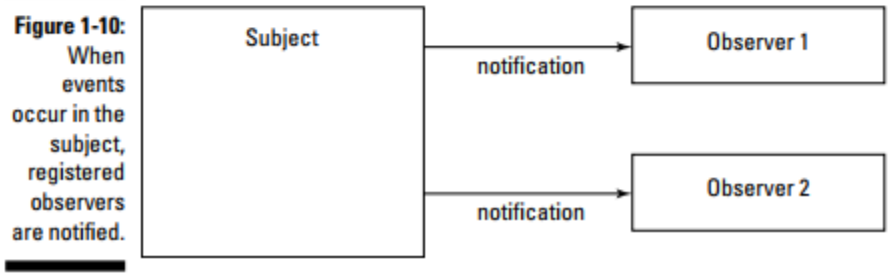


Figure 1-10: When events occur in the subject, registered observers are notified.

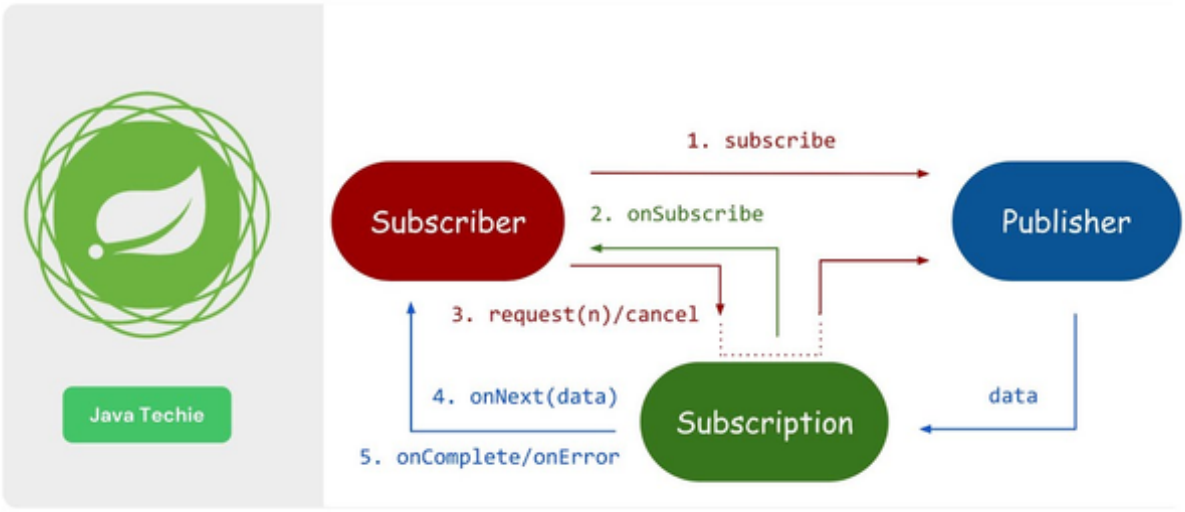
- Hình 1-8, cho phép observer thứ 1 đăng ký với hệ thống.
- Hình 1-9, cho phép observer thứ 2 đăng ký với hệ thống.
- Hiện tại hệ thống đang liên lạc với 2 observer: Observer 1 và Observer 2. Khi hệ thống phát sinh một sự kiện cụ thể nào đó, nó sẽ thông báo (notification) với cả 2 observer như hình số 1-10.

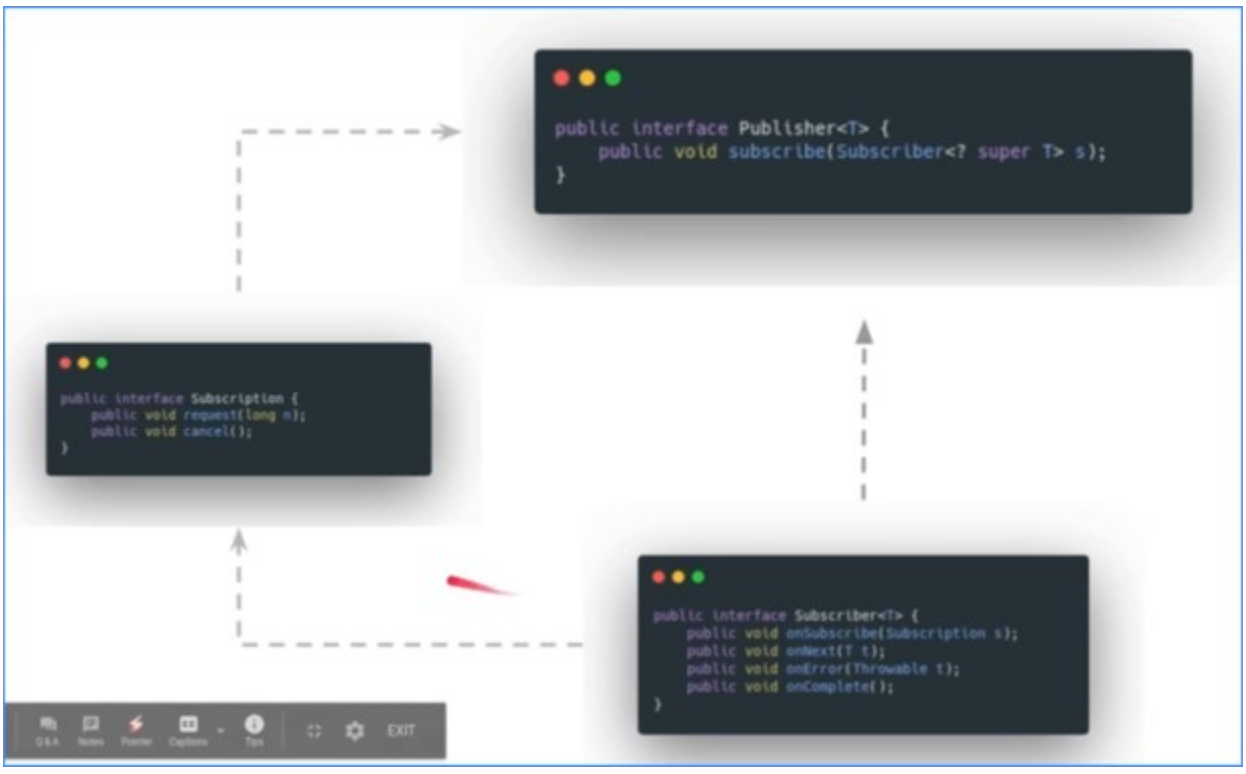
Observer Pattern còn gọi là Dependents, Publish/Subcribe hoặc Source/Listener.

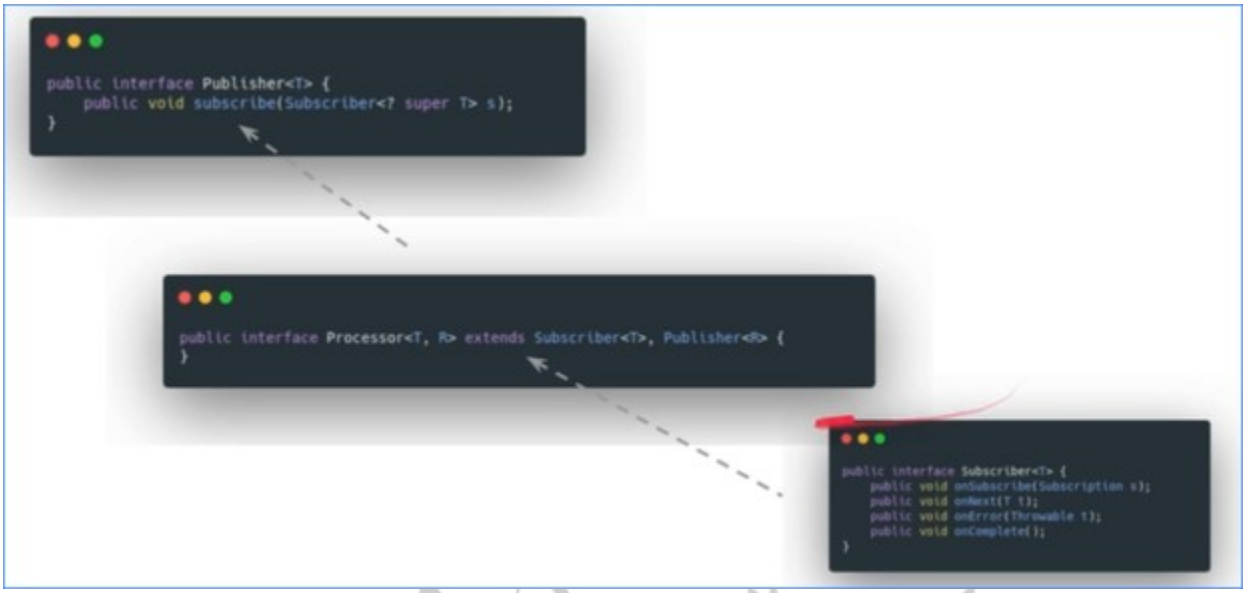
Reactive Programming

Specification

1 Publisher 2 Subscriber 3 Subscription







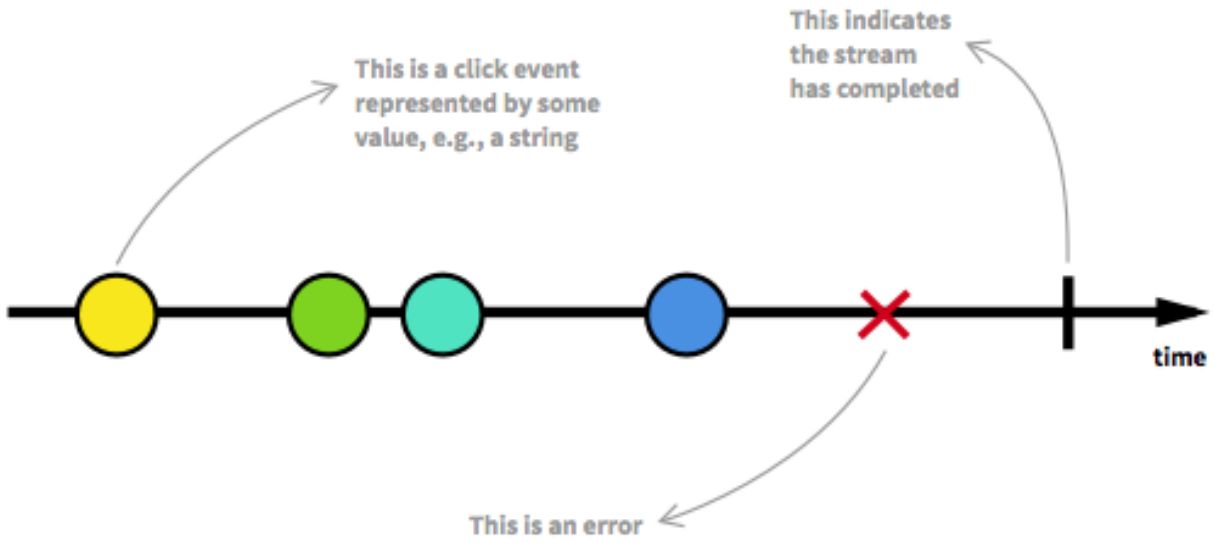
4. Reactive programming là gì?

Reactive programming is programming with asynchronous data streams

Reactive programming là lập trình với các luồng dữ liệu không đồng bộ

- Ở khái niệm trên chúng ta cần chú ý đến 2 điểm quan trọng **Stream** & **Ansynchronous**
  - **Stream :** Khi thực hiện 1 task bất kỳ thường chúng ta chỉ quan tâm đến 3 yếu tố :
    - Giá trị trả về từ task đó (Data)
    - Thông báo lỗi (Error nếu có)
    - Thời điểm task finish (Completed)

Khi lập trình đồng bộ (synchronous) việc xác định 3 yếu tố trên không khó khăn, nhưng khi lập trình bất đồng bộ (asynchronous) việc xác định 3 yếu tố này là không hề dễ dàng. Như vậy để giải quyết vấn đề này ta cần có 1 cơ chế giúp xác định được 3 yếu tố trên cả khi lập trình đồng bộ & bất đồng bộ. Funtion Reactive Programming giải quyết vấn đề này bằng cách sử dụng stream để truyền tải dữ liệu: nó có thể sẽ emit ra 3 thứ : 1 value, 1 error, 1 completed (tín hiệu kết thúc 1 task) theo 1 trình tự thời gian từ nơi phát ra (Producer) tới nơi lắng nghe (Subscriber).

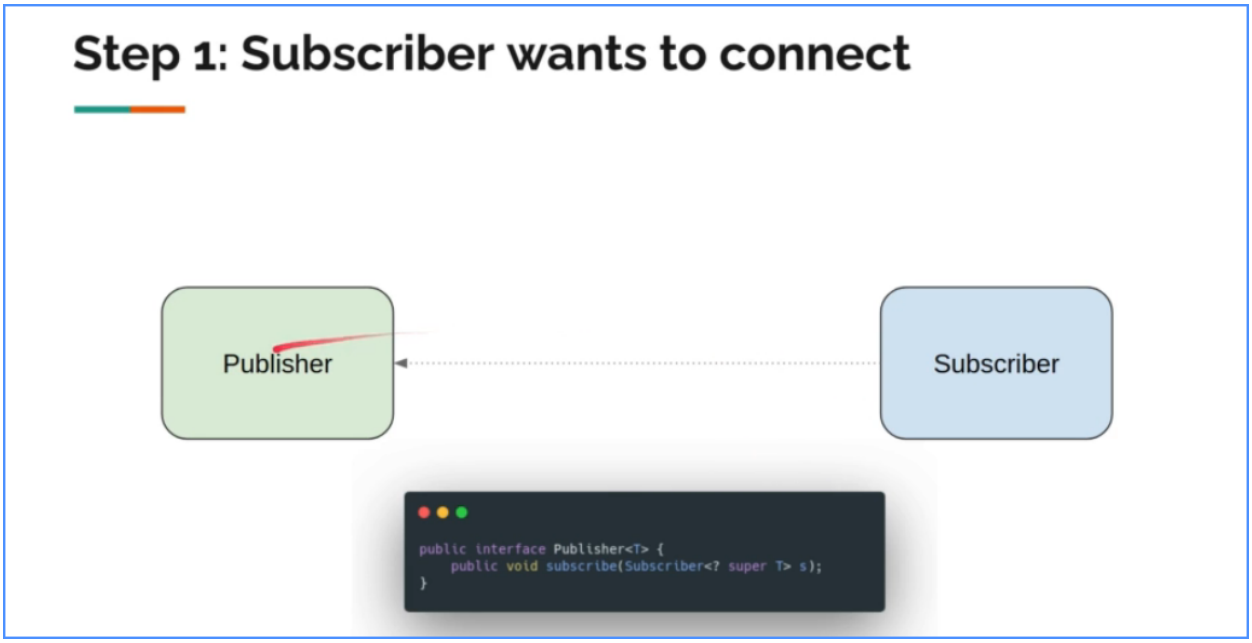


Tại sao nên dùng nó?

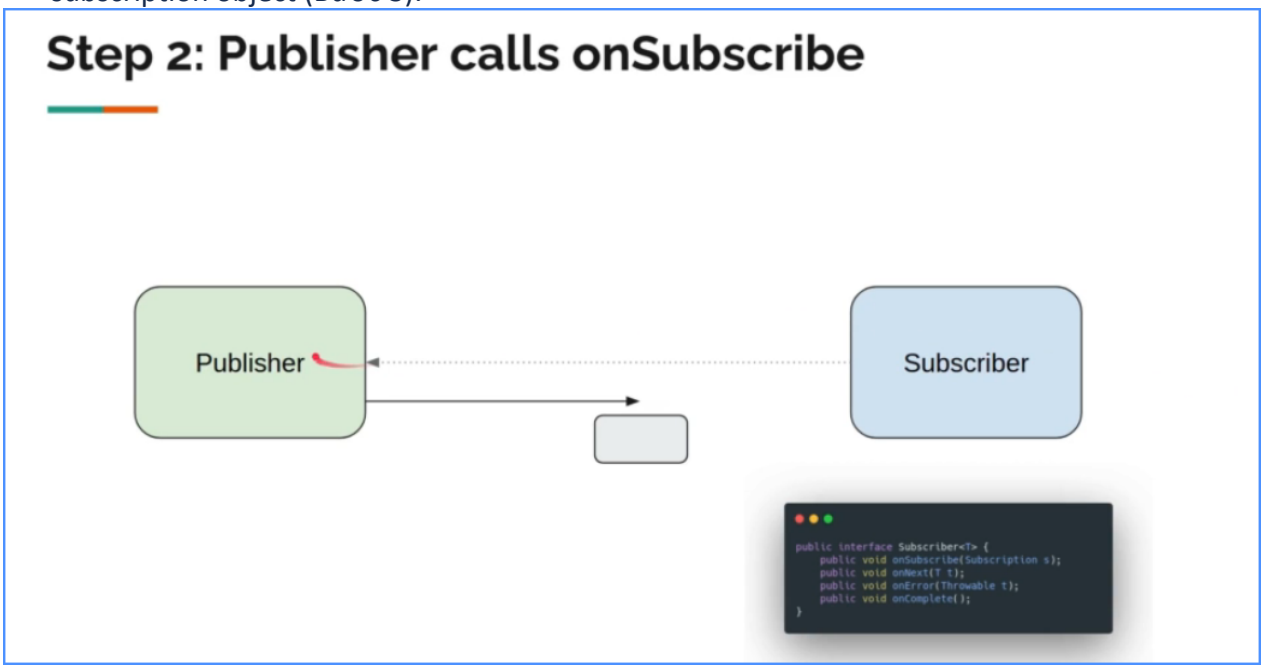
- Nếu chỉ dừng lại ở đó thì FRP cũng không có gì đặc biệt, điều tạo nên sức mạnh của FRP là việc áp dụng functional programming cho phép filter (filter, take, scan, …), chuyển đổi từ stream này qua stream khác (map, flatMap, reduce), hoặc merge nhiều stream thành một stream mới (combine, merge, zip, …) khá dễ dàng mà không làm thay đổi trạng thái của stream ban đầu.
- Việc sử dụng FRP sẽ cải thiện được trải nghiệm người dùng, khi chúng ta muốn ứng dụng phản hồi nhanh hơn.Lợi ích tiếp theo là giúp hạn chế lưu trữ, quản lý các state trung gian. Trong ví dụ clickStream trên, nếu như sử dụng cách lập trình thông thường, thì phải khai báo rất nhiều biến (state) để lưu trữ các bước trung gian. Ví dụ: timer, click count collection, … Trong FRP, các bước này là không cần thiết nhờ khả năng chuyển đổi stream (map, flatMap, reduce, ….).
- Một điểm mạnh khác của RP là giúp cho việc xử lý lỗi trong lập trình bất đồng bộ nhẹ nhàng hơn rất nhiều. Nếu bạn nào từng handle error khi lập trình bất đồng bộ, multiple thread, thì sẽ thấy việc này không hề dễ dàng. RP giúp tách biệt việc xử lý lỗi với logic. Việc này giúp cho code trong sáng hơn rất nhiều.

5. Publisher & Subscriber Communication

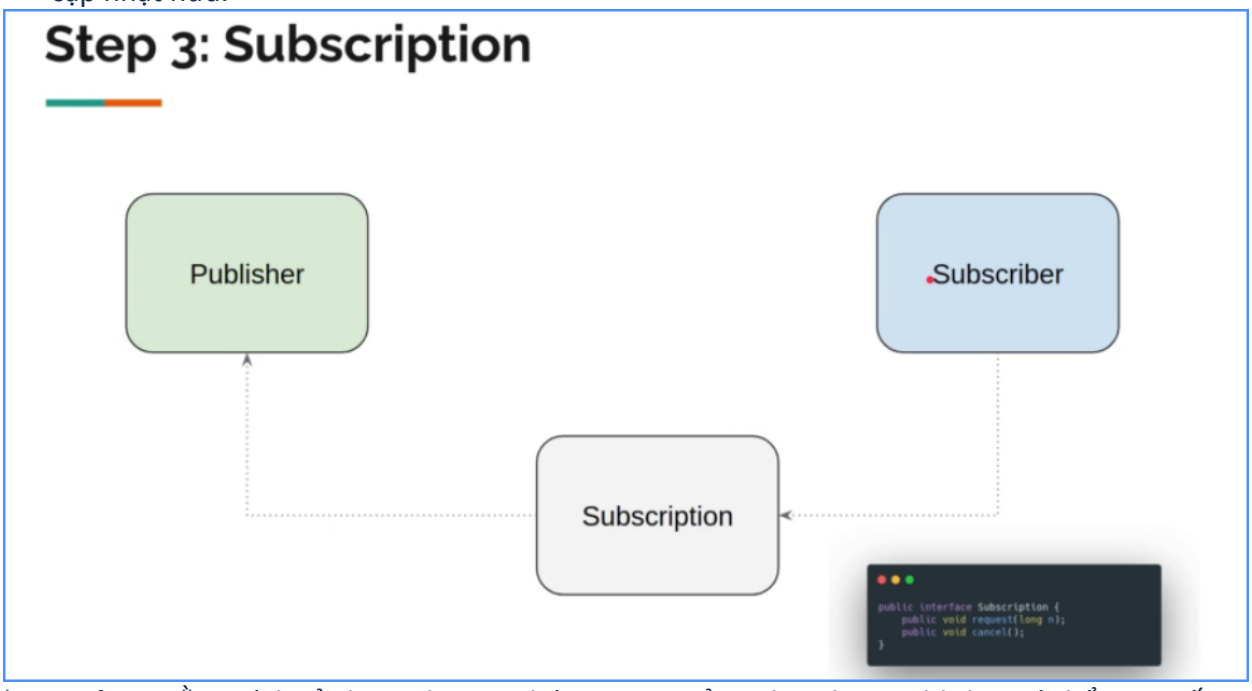
- **Bước một:** Subscriber muốn nhận thông tin cập nhật từ Publisher. Publisher interface có phương thức subscribe thông qua đó instance Subscriber được chuyển đến Publisher, để có thể tạo đối tượng Subscriber để nhận thông tin cập nhật từ Publisher..



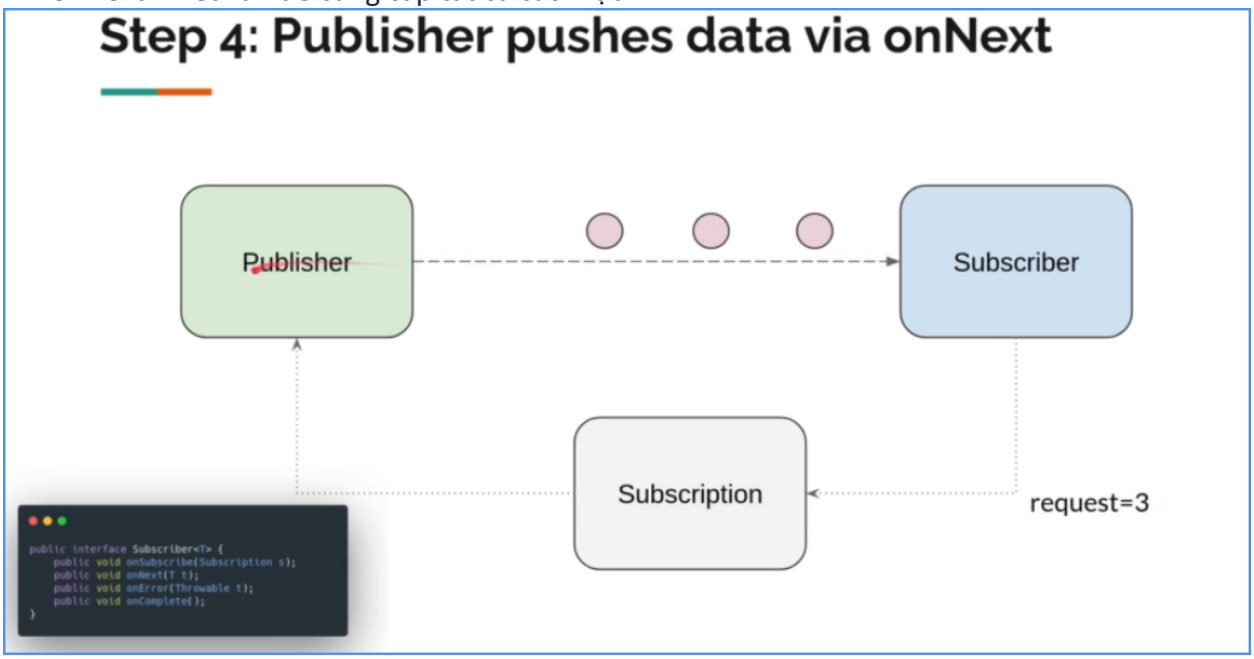
- **Bước hai:** Khi Publisher chấp nhận yêu cầu của Subscriber, nó sẽ chuyển subscription object cho Subscriber. Điều này được thực hiện như một phần của phương thức onSubscribe của Subscriber, sử dụng phương thức này cho phép Subscriber liên lạc với Publisher thông qua subscription object (Bước 3).



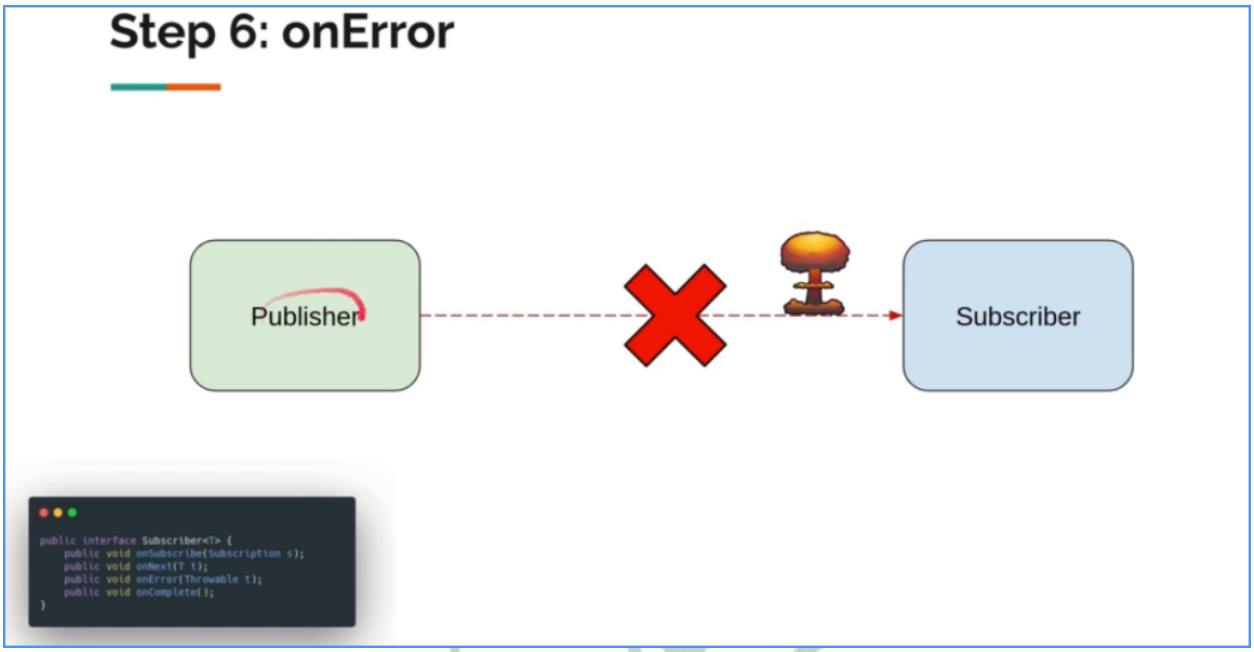
- **Bước 3:** Subscriber có thể yêu cầu dữ liệu hoặc hủy đăng ký nếu không muốn nhận thông tin cập nhật nữa.



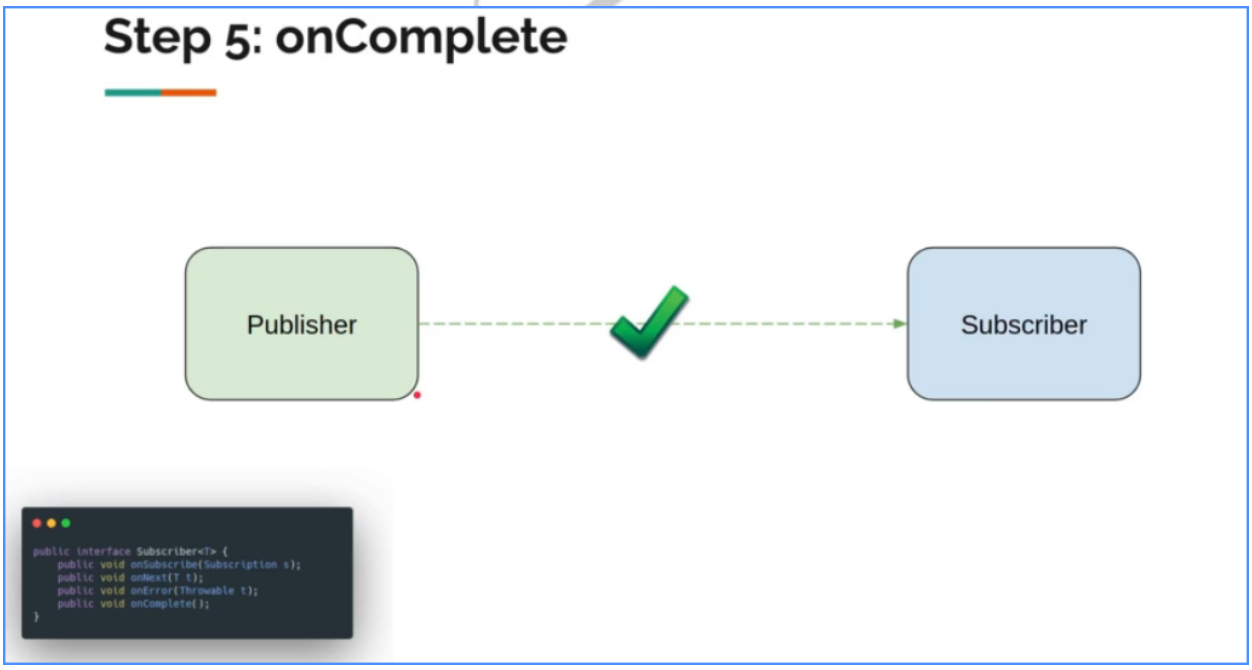
- **Bước 4:** Bằng cách sử dụng phương thức onNext của Subscriber, Publisher có thể cung cấp dữ liệu cho Subscriber. Nếu Publisher có nhiều mục cần phát ra, nó sẽ gọi phương thức onNext nhiều lần để cung cấp tất cả các mục.



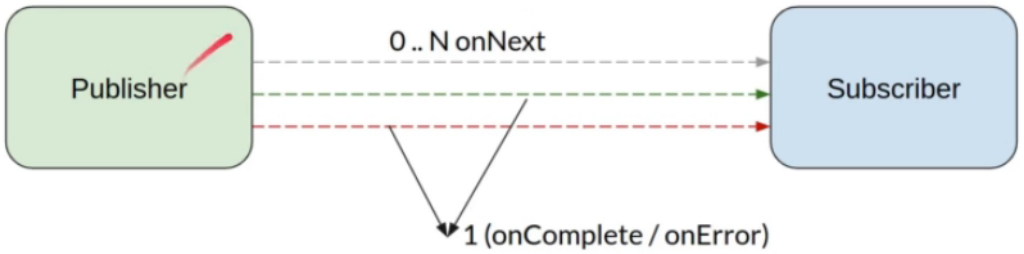
- **Bước 5:** Khi Publisher không còn mục nào để phát ra hoặc đã phát tất cả các mục cho Subscriber, nó có thể gọi phương thức onComplete. Điều này thông báo cho Subscriber rằng công việc của Publisher đã hoàn thành và Publisher sẽ không phát thêm bất kỳ mục nào cho Subscriber.



- **Bước 6:** Nếu Publisher gặp bất kỳ sự cố không mong muốn nào trong khi xử lý yêu cầu của Subscriber, Publisher có thể chuyển chi tiết lỗi đến Subscriber thông qua phương thức onError. Trong trường hợp này, Publisher sẽ không phát thêm bất kỳ mục nào cho Subscriber.



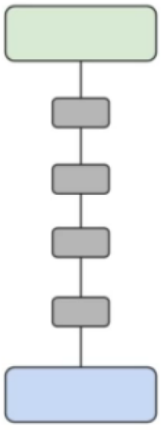
onNext / onComplete / onError



6. Một số thuật ngữ:

Terminologies

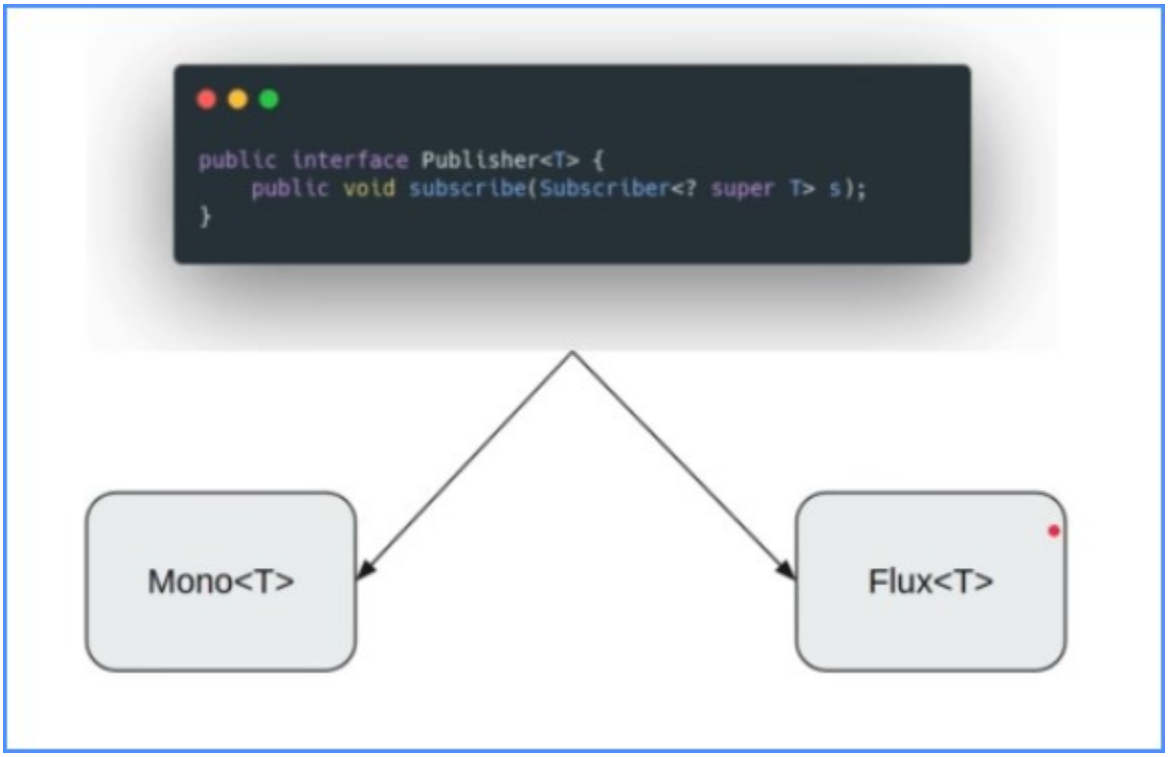
- Publisher
  - Source
  - Observable
  - Upstream
  - Producer
- Subscriber
  - Sink
  - Observer
  - Downstream
  - Consumer



Chapter 2: Reactor

Reactor là một nền tảng để ta triển khai việc lập trình theo phong cách **reactive programming**. Nó được tích hợp trực tiếp với Java 8 funcion APIs như CompletableFuture, Stream, Duration.

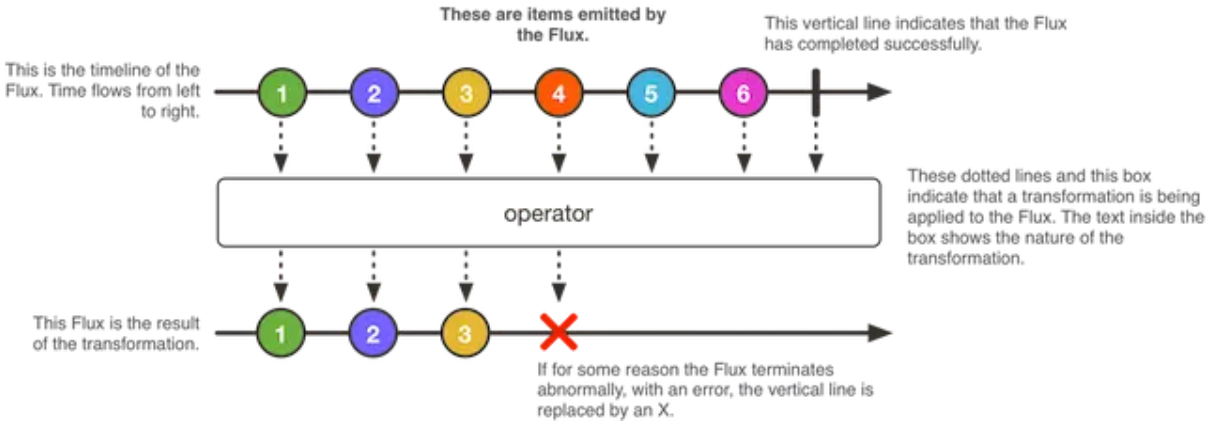
Reactor cung cấp 2 loại về **Publisher** :



Flux: là một steam phát ra từ 0...n phần tử, có thể hình dung nó là một List dữ liệu. Ví dụ tạo đơn giản:

```
Flux<Integer> just = Flux.just(1,2,3,4);
```

Và cũng giống như khái niệm về Reactive, có 3 tín hiệu mà Flux emit ra để Subscribe có thể nhận được đó là **onNext()** để hứng return data, **onComplete()** để nhận tín hiệu Stream hoàn thành và **onError()** để nhận giá trị lỗi trả về.

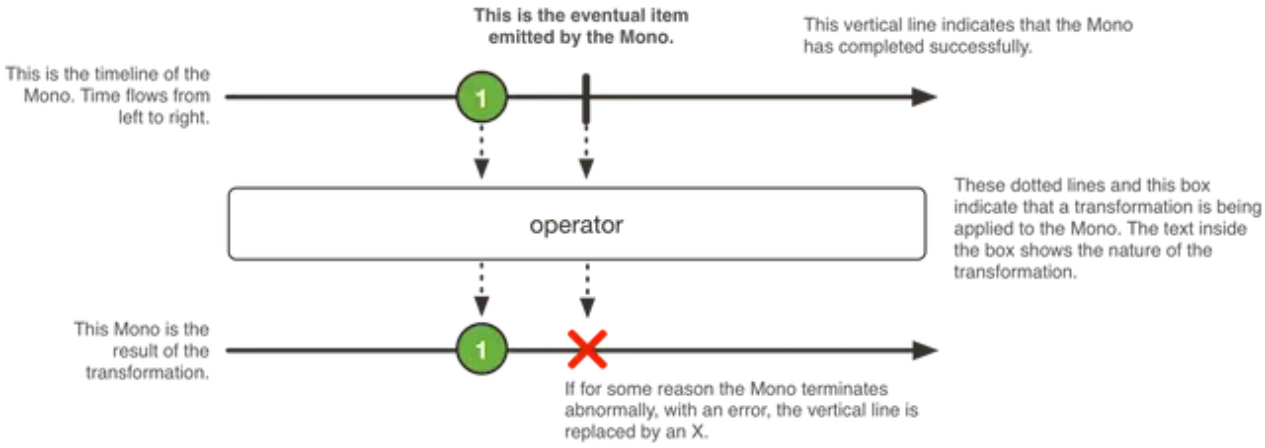


Mono: là một steam phát ra từ 0...1 phần tử. Nó hoạt động gần giống như Flux, chỉ là bị giới hạn không quá một phần tử hoặc không có phần tử nào (rỗng) . Ví dụ:

```
Mono<String> just = Mono.just("ABC"); // Mono với 1 phần tử
Mono<Void> just = Mono.empty(); // Mono với 0 phần tử (rỗng)
```

Cũng giống như Flux Mono cung cấp 3 function onNext(), onComplet() và onError() để Subscribe thao tác với dữ liệu được trả về.

Mono cũng có thể truyền đổi thành một Flux, ví dụ 2 hoặc nhiều Mono có thể gộp thành (combine) một Flux bằng cách sử dụng function **concatWith(),** ví dụ Mono#concatWith(Publisher) sẽ trả về một Flux. Hay sử dụng Mono#then(Mono) để trả về một Mono khác với mục đích kết thúc một Stream mà không quan tâm tới dữ liệu của Mono gốc. Điểm khác nhau giữa Mono#then và Mono#map đó là then hoạt động dựa trên tính hiệu onComplete mặc dù Mono gốc có thể empty, trong khi map hay flatmap chỉ hoạt động dựa trên tín hiệu onNext , có nghĩa là chỉ hoạt động khi Mono gốc có dữ liệu trả về (not empty).



Chapter 3: Mono

1. Mono & Flux

Khi có flux, nhu cầu mono là gì?. Sẽ rất tiện lợi khi biết chắc chắn rằng chỉ mong đợi một item từ publisher => có thể sử dụng mono chẳng hạn.

Khi lấy dữ liệu trong cơ sở dữ liệu có bao nhiêu bản ghi?

- Chỉ muốn có một thông tin tổng hợp duy nhất. Trong trường hợp đó, sẽ sử dụng mono.
- Nếu bạn muốn tất cả các bản ghi từ bảng thì có thể sử dụng flex. Bạn biết chắc chắn rằng có thể mong đợi không có bản ghi hoặc một bản ghi hoặc N bản ghi.

2. [Mono]- Just

Hàm Mono.just trong Reactor Framework được sử dụng để tạo một Mono từ giá trị đã cho. Nó chuyển đổi giá trị đầu vào thành một Mono publisher, phát ra giá trị duy nhất đó và hoàn thành.

Ví dụ, bạn có thể sử dụng Mono.just để tạo một Mono phát ra một chuỗi đơn giản:

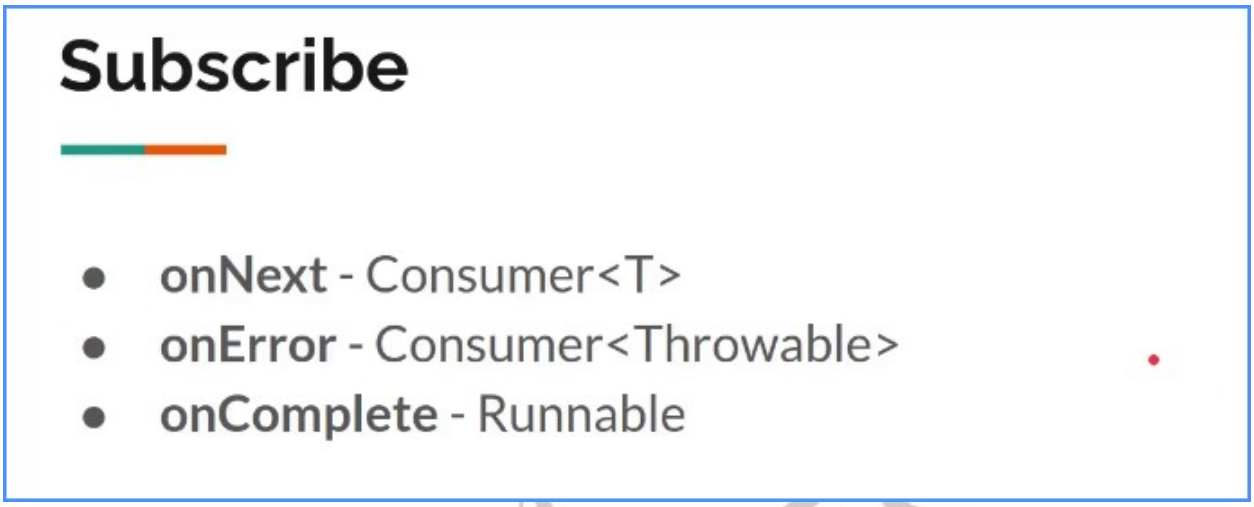
```
Mono<String> mono = Mono.just("Hello, world!");
```

Lưu ý

- Giống khi sử dụng Stream, stream sẽ là lazy khi không thực hiện các Terminal Operations
- Quy tắc số một trong Reactive programming: Không có gì xảy ra cho đến khi bạn subscribe.

```
import reactor.core.publisher.Mono;

public class Lec02MonoJust {

    public static void main(String[] args) {

        // publisher
        Mono<Integer> mono = Mono.just(1);

        System.out.println(mono);

        mono.subscribe(i -> System.out.println("Received : " + i));

    }

}
```

3. [Mono]- Subscribe



Subscribe: Như đã nói bên trên rằng "**không có gì xảy ra cho đến khi subscribe"**, các Stream như Mono hay Flux sẽ không hành động gì cả cho tới khi nó được **Observer** hay **Subcriber** (lắng nghe). Do vậy trong Reactor có cung cấp một function subscribe() để thực hiện lắng nghe Stream.

Ví dụ để subcribe một Flux với basic method không có đối số (arguments)

onNext event: khi cung cấp consumer kiểu T, tham số đầu tiên là cho cuộc gọi onNext. Okay. Vì vậy, khi cung cấp consumer, publisher sẽ gọi phương thức onNext. Vì vậy, bất kỳ hành vi nào cung cấp làm phần của consumer sẽ được gọi. Nếu bạn muốn in nó, nó sẽ được in ra. Đây là tham số đầu tiên rất quan trọng.

```
Flux<Integer> ints = Flux.range(1, 3); //Tạo một Flux với 3 phần tử từ 1->3
ints.subscribe(); // Thực hiện lắng nghe trên Flux vừa tạo
```

Với ví dụ trên thì sẽ không có out-put nào tạo ra, để có thể bắt (catch) được các out-put thì ta sẽ truyền một đối số là`Consumer` vào `subscribe()` ví dụ:

```
Flux<Integer> ints = Flux.range(1, 3);
ints.subscribe(i -> System.out.println(i)); // subcribe Flux và in ra dữ liệu trả
về của nó
/*Output:
1
2
3
```

```
*/
```

Error Event: Một lỗi có thể được sử lý ngay trong subcribe ((error handler) như ví dụ sau:

```
Flux<Integer> ints = Flux.range(1, 6) //(1)
      .map(i -> { // (2)
        if (i <= 3) {
           return i;
        }
        throw new RuntimeException("Got to 4");
      });
ints.subscribe(i -> System.out.println(i), //(3)
      error -> System.err.println("Error: " + error)); //(4)
```

1. Tạo một Stream Flux có 4 phần tử từ 1-> 6
2. Map lại Stream hiện tại ra một Steam mới mà chỉ được phép có 3 phần tử từ 1->3 nếu lớn hơn sẽ throw ra một Exception
3. Print ra dữ liệu output của Stream mới được tạo
4. sử dụng consumer là error để kết thúc Stream và out-put ra lỗi nếu có

Output:

```
/*
1
2
3
*/
```

Error: java.lang.RuntimeException: Got to 4

Completed Event: Nếu có một error được throw ra thì Stream sẽ dừng lại (completed) ngay lập tức. Nếu không có lỗi xảy ra thì ta có thể tạo một event completed khi Stream kết thúc như ví dụ:

```
Flux<Integer> ints = Flux.range(1, 4); //(1)
ints.subscribe(i -> System.out.println(i), //(2)
    error -> System.err.println("Error " + error), (3)
    () -> System.out.println("Done")); (4)
```

1. Tạo một Stream Flux có 4 phần tử từ 1-> 4

2. Print ra dữ liệu output của Stream
3. Sử dụng consumer là error để completed Stream và out-put ra lỗi nếu có
4. Sử dụng consumer là () để completed Stream và out-put ra event complete

Output:

```
1
2
3
4
Done
```

Subscribe có thể yêu cầu một hành động nào đó, ví dụ như yêu cầu số lượng dữ liệu được emit ra trước khi Steam được complete bằng cách sử dụng sub , ví dụ

```
Flux<Integer> ints = Flux.range(1, 4);
ints.subscribe(i -> System.out.println(i),
    error -> System.err.println("Error " + error),
    () -> System.out.println("Done"),
    sub -> sub.request(10));
```

Stream trên sẽ bị treo (hangs) vĩnh viễn (không bao giờ completed) trừ khi Stream được cancel, bởi vì Subscribe đã yêu cầu chỉ completed cho tới khi nhận được đủ 10 phần tử.

Output: Event “Done” sẽ không bao giờ được output ra.

```
1
2
3
4
```

Cancel Event: function Subscribe trả về một kiểu dữ liệu là Disposable và Disposable Interface có cung cấp một method là dispose() để giúp một Stream có thể bị hủy bỏ (cancel) ví dụ:

```
Flux.just(1,2,3).subscribe().dispose();
```

Stream sẽ ngay lập tức bị cancel ngay sau khi nó được Subscribe

5. [Mono]- Emitting Empty / Error

Mono là một loại dữ liệu tuần tự có thể phát ra một giá trị duy nhất hoặc không có giá trị (empty) hoặc phát ra một lỗi.

Để phát ra một Mono empty, bạn có thể sử dụng phương thức Mono.empty() như sau:

```
Mono.empty();
```

Để phát ra một Mono với lỗi, bạn có thể sử dụng phương thức Mono.error() và truyền vào đối tượng Exception hoặc Throwable tương ứng, ví dụ:

```
Mono.error(new RuntimeException("Lỗi xảy ra"));
```

Để thực hiện việc phát ra Mono empty hoặc Mono với lỗi trong các hàm xử lý của Webflux, bạn có thể sử dụng các phương thức như flatMap, switchIfEmpty, onErrorResume, onErrorReturn, v.v.

Ví dụ sử dung empty và error

```
public static void main(String[] args) {
    userRepository(20)
            .subscribe(
                    Util.onNext(),
                    Util.onError(),
                    Util.onComplete()
            );

}

private static Mono<String> userRepository(int userId){
    // 1
    if(userId == 1){
        return Mono.just(Util.faker().name().firstName());
    }else if(userId == 2){
        return Mono.empty(); // null
    }else
        return Mono.error(new RuntimeException("Not in the allowed range"));
}
```

Ví dụ, trong một HandlerFunction của Webflux, bạn có thể sử dụng switchIfEmpty để xử lý trường hợp Mono rỗng:

```
public Mono<ServerResponse> handleRequest(ServerRequest request) {
    return someService.getData()
            .flatMap(data -> ServerResponse.ok().bodyValue(data))
            .switchIfEmpty(ServerResponse.notFound().build());
}
```

Trong ví dụ trên, someService.getData() trả về một Mono. Nếu Mono đó rỗng, switchIfEmpty sẽ được kích hoạt và trả về một ServerResponse với mã trạng thái 404 (Not Found).

Tương tự, bạn có thể sử dụng onErrorResume để xử lý trường hợp Mono gặp lỗi:

```
public Mono<ServerResponse> handleRequest(ServerRequest request) {
    return someService.getData()
            .flatMap(data -> ServerResponse.ok().bodyValue(data))
            .onErrorResume(e ->
ServerResponse.status(HttpStatus.INTERNAL_SERVER_ERROR).bodyValue("Lỗi xảy ra"));
}
```

Trong ví dụ trên, nếu Mono gặp lỗi, onErrorResume sẽ được kích hoạt và trả về một ServerResponse với mã trạng thái 500 (Internal Server Error) và thông báo lỗi tương ứng.

Hy vọng những thông tin này giúp bạn hiểu cách sử dụng Mono để phát ra giá trị empty hoặc lỗi trong Webflux.

6. [Mono]- From Supplier

Mono.fromSupplier là một phương thức để tạo một Mono từ một Supplier. Supplier là một functional interface trong Java, nó chỉ định một hàm không có tham số và trả về một giá trị.

Khi sử dụng Mono.fromSupplier, bạn cung cấp một Supplier và Mono sẽ sử dụng nó để tạo một Mono emitting giá trị được cung cấp bởi Supplier đó.

Dưới đây là một ví dụ minh họa của cách sử dụng Mono.fromSupplier trong Java:

```
import reactor.core.publisher.Mono;

public class Example {
    public static void main(String[] args) {
        Supplier<String> supplier = () -> "Hello, world!";

        Mono<String> mono = Mono.fromSupplier(supplier);

        mono.subscribe(System.out::println); // Output: Hello, world!
    }
}
```

Trong ví dụ trên, chúng ta tạo một Supplier để cung cấp chuỗi "Hello, world!". Sau đó, chúng ta sử dụng Mono.fromSupplier để tạo một Mono từ Supplier đó. Khi chúng ta subscribe vào Mono này và in ra giá trị, chúng ta sẽ nhận được đầu ra là "Hello, world!".

Mono.fromSupplier thường được sử dụng khi bạn muốn tạo một Mono emitting một giá trị được tính toán hoặc cung cấp bởi một Supplier. Nó cung cấp một cách thuận tiện để tạo Mono từ một function không có tham số.

7. [Mono]- From Callable

Mono.fromCallable là một phương thức để tạo một Mono từ một Callable. Callable là một functional interface trong Java, nó chỉ định một hàm có thể trả về một giá trị và có thể ném một ngoại lệ (Exception).

Khi sử dụng Mono.fromCallable, bạn cung cấp một Callable và Mono sẽ sử dụng nó để tạo một Mono emitting giá trị hoặc ngoại lệ được trả về bởi Callable đó.

Dưới đây là một ví dụ minh họa về cách sử dụng Mono.fromCallable trong Java:

```
import reactor.core.publisher.Mono;

import java.util.concurrent.Callable;

public class Example {
    public static void main(String[] args) {
        Callable<String> callable = () -> {
            // Các công việc khác nhau có thể được thực hiện ở đây
```

```
            return "Hello, world!";
        };

        Mono<String> mono = Mono.fromCallable(callable);

        mono.subscribe(System.out::println); // Output: Hello, world!
    }
}
```

Trong ví dụ trên, chúng ta tạo một Callable để thực hiện một công việc và trả về chuỗi "Hello, world!". Sau đó, chúng ta sử dụng Mono.fromCallable để tạo một Mono từ Callable đó. Khi chúng ta subscribe vào Mono này và in ra giá trị, chúng ta sẽ nhận được đầu ra là "Hello, world!".

Mono.fromCallable thường được sử dụng khi bạn muốn tạo một Mono emitting một giá trị hoặc ngoại lệ được tính toán hoặc trả về bởi một Callable. Nó cung cấp một cách thuận tiện để tạo Mono từ một function có thể ném ngoại lệ.

8. [Mono]- Pipeline Build vs Execution

Trong Reactor, khi làm việc với Mono, chúng ta có hai giai đoạn quan trọng: Pipeline Build và Pipeline Execution.

1. Pipeline Build (Xây dựng đường ống):
- Trong giai đoạn này, chúng ta xây dựng một chuỗi các phép biến đổi (operators) trên Mono để định nghĩa các bước xử lý dữ liệu.
- Các phép biến đổi này có thể là map, flatMap, filter, retry, timeout, và nhiều phép biến đổi khác để biến đổi, lọc, hoặc xử lý dữ liệu trên Mono.
- Pipeline Build thường được thực hiện bằng cách gọi các phương thức của Mono và các operators trên Mono để xây dựng chuỗi xử lý dữ liệu theo yêu cầu của bạn.
2. Pipeline Execution (Thực thi đường ống):
- Sau khi xây dựng xong chuỗi xử lý dữ liệu, chúng ta thực hiện việc thực thi chuỗi đó bằng cách gọi phương thức subscribe() trên Mono.
- Khi Mono được subscribe, chuỗi xử lý dữ liệu sẽ được kích hoạt và bắt đầu thực thi.
- Mono sẽ lắng nghe sự kiện và tương tác với dữ liệu theo chuỗi phép biến đổi đã định nghĩa trong Pipeline Build.
- Khi có dữ liệu được phát ra (emitted), Mono sẽ truyền dữ liệu đó qua các phép biến đổi và operators trong chuỗi xử lý dữ liệu.
- Kết quả cuối cùng được trả về qua các subscriber hoặc được xử lý tiếp trong chuỗi xử lý dữ liệu.

Tóm lại, Pipeline Build (Xây dựng đường ống) được sử dụng để thiết lập chuỗi các phép biến đổi trên Mono, định nghĩa các bước xử lý dữ liệu. Trong khi đó, Pipeline Execution (Thực thi đường ống) là giai đoạn thực sự thực hiện chuỗi xử lý dữ liệu, khi Mono được subscribe và các phép biến đổi được kích hoạt để xử lý dữ liệu và trả về kết quả.

```
public static void main(String[] args) {

    getName();
    getName();

    Util.sleepSeconds(4);
}

private static Mono<String> getName(){
    System.out.println("entered getName method");
    return Mono.fromSupplier(() -> {
        System.out.println("Generating name..");
        Util.sleepSeconds(3);
        return Util.faker().name().fullName();
    }).map(String::toUpperCase);
}
```

9. [Mono]- Where is Async?

```
  public static void main(String[] args) {

    String name = getName()
            .subscribeOn(Schedulers.boundedElastic())
            .subscribe(Util.onNext()).toString();
    System.out.println(name);
    getName();

    Util.sleepSeconds(4);
}

private static Mono<String> getName(){
    System.out.println("entered getName method");
    return Mono.fromSupplier(() -> {
        System.out.println("Generating name..");
        Util.sleepSeconds(3);
        return Util.faker().name().fullName();
    }).map(String::toUpperCase);
}
```

=> Output

```
entered getName method
reactor.core.publisher.LambdaMonoSubscriber@5a61f5df
entered getName method
Generating name..
Received : ELISHA KEELING
```

Sử dụng subscribeOn(Schedulers.boundedElastic()) để thực hiện việc đăng ký và thực thi Mono trên một luồng không chặn (boundedElastic). Điều này cho phép Mono được thực thi bất đồng bộ.

Tiếp theo, sử dụng subscribe(Util.onNext()) để đăng ký một người nghe (subscriber) cho Mono và sử dụng toString() để chuyển đổi giá trị phát ra thành chuỗi. Cuối cùng, in ra giá trị name và gọi getName() một lần nữa.

Trong phương thức getName(), bạn sử dụng Mono.fromSupplier() để tạo một Mono từ một supplier bất đồng bộ. Trong trường hợp này, supplier sinh ra một tên giả định sau đó dùng map để chuyển đổi tên thành chữ hoa.

Do sử dụng subscribeOn(Schedulers.boundedElastic()) và Util.sleepSeconds(3) để tạo một độ trễ, các hoạt động trong getName() được thực thi bất đồng bộ.

10. [Mono]- Block

Trong Reactive Webflux, phương thức block() được sử dụng trên đối tượng Mono để chuyển từ mô hình lập trình bất đồng bộ sang mô hình lập trình đồng bộ. Khi bạn gọi phương thức block() trên một Mono, nó sẽ chặn luồng thực thi cho đến khi Mono phát ra một giá trị hoặc kết thúc.

Việc sử dụng block() được thực hiện trong một số trường hợp nhất định khi bạn cần lấy kết quả từ một Mono và sử dụng nó trong một phần của mã đồng bộ, ví dụ như trong phương thức main của ứng dụng hoặc trong các phương thức không hỗ trợ lập trình bất đồng bộ.

Tuy nhiên, cần lưu ý rằng việc sử dụng block() trong môi trường Reactive Webflux không được khuyến nghị. Điều này có thể gây chặn luồng thực thi và làm gián đoạn tính năng không chặn của ứng dụng, làm giảm hiệu suất và khả năng mở rộng của hệ thống

```
public static void main(String[] args) {

    getName();
```

```
    String name = getName()
            .subscribeOn(Schedulers.boundedElastic())
            .block();
    System.out.println(name);
    getName();

    Util.sleepSeconds(4);
}

private static Mono<String> getName(){
    System.out.println("entered getName method");
    return Mono.fromSupplier(() -> {
        System.out.println("Generating name..");
        Util.sleepSeconds(3);
        return Util.faker().name().fullName();
    }).map(String::toUpperCase);
}
```

=> Output

```
entered getName method
Generating name..
MR. NORBERTO ONDRICKA
entered getName method
```

11. [Mono]- From Future

Phương thức fromFuture(Future) trong lớp Mono được sử dụng để chuyển đổi một Future thành một Mono. Future là một cơ chế trong Java để đại diện cho một giá trị hoặc kết quả của một phép tính bất đồng bộ.

Khi sử dụng phương thức fromFuture, bạn có thể chuyển đổi một Future thành một Mono tương ứng, cho phép bạn tích hợp các phép tính bất đồng bộ dựa trên Future vào luồng reactive.

Ví dụ, nếu bạn có một Future đại diện cho kết quả của một phép tính bất đồng bộ, bạn có thể sử dụng Mono.fromFuture(future) để chuyển đổi Future thành một Mono. Sau đó, bạn có thể xử lý kết quả của Mono bằng cách sử dụng các toán tử reactive như map, flatMap, v.v.

```
Future<String> futureResult = ... // Một Future đại diện cho kết quả bất đồng bộ

Mono<String> monoResult = Mono.fromFuture(futureResult);
monoResult.subscribe(result -> {
    // Xử lý kết quả
    System.out.println(result);
```

```
});
```

```
public class Lec07MonoFromFuture {

    public static void main(String[] args) {

        Mono.fromFuture(getName())
                .subscribe(Util.onNext());
        Util.sleepSeconds(1);
    }

    private static CompletableFuture<String> getName(){
        return CompletableFuture.supplyAsync(() ->
Util.faker().name().fullName());
    }
}
```

Trong ví dụ trên, Mono.fromFuture(futureResult) chuyển đổi futureResult thành một Mono<String>, và sau đó bạn có thể đăng ký một người nghe (subscriber) để xử lý kết quả khi Mono phát ra giá trị.

Tuy nhiên, cần lưu ý rằng việc sử dụng Future và fromFuture trong môi trường Reactive Webflux không được khuyến nghị. Thay vào đó, nếu có thể, hãy sử dụng các phương thức reactive như Mono.fromCallable hoặc Mono.fromCompletionStage để thực hiện các phép tính bất đồng bộ trong môi trường Reactive Webflux.

12. [Mono]- From Runnable

Phương thức fromRunnable(Runnable) trong lớp Mono được sử dụng để chuyển đổi một Runnable thành một Mono<Void>. Runnable là một giao diện trong Java được sử dụng để biểu diễn một khối mã không trả về kết quả.

Khi sử dụng phương thức fromRunnable, có thể chuyển đổi một Runnable thành một Mono<Void>, cho phép bạn tích hợp các khối mã không trả về kết quả vào luồng reactive.

```
public static void main(String[] args) {
    Mono.fromRunnable(timeConsumingProcess())
            .subscribe(Util.onNext(),
                    Util.onError(),
```

```
                    () -> {
                        System.out.println("process is done. Sending emails...");
                    });
}

private static Runnable timeConsumingProcess(){
    return () -> {
        Util.sleepSeconds(3);
        System.out.println("Operation completed");
    };
}
```

=> phương thức fromRunnable trong môi trường Reactive Webflux cho phép bạn chuyển đổi một Runnable thành một Mono<Void>, và từ đó bạn có thể xử lý sự kiện và hoàn tất của quá trình trong luồng reactive.

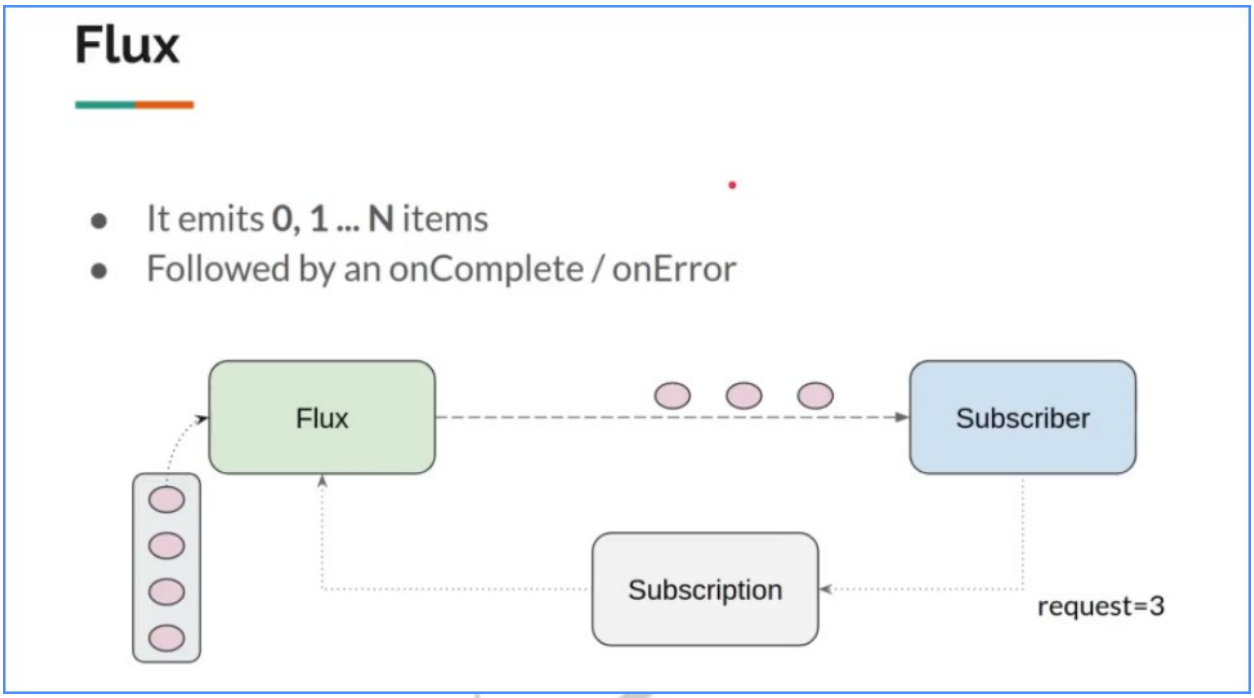
Mono

Type	Condition	What to use
Create Mono	Data already present	Mono.just(data)
Create Mono	Data to be calculated	• Mono.fromSupplier(() -> getData()) • Mono.fromCallable(() -> getData())
Create Mono	Data is coming from async completableFuture	Mono.fromFuture(future)
Create Mono	Emit empty once a given runnable is complete	Mono.fromRunnable(runnable)
Pass Mono as argument	Function accepts a Mono<Address>. But I do not have data.	Mono.empty()
Return Mono	Function needs to return a Mono.	• Mono.error(...) • Mono.empty() • above creation types

Chapter 4: Flux

Flux là một thành phần quan trọng trong mô hình lập trình reactive. Nó đại diện cho một luồng dữ liệu có thể phát ra một hoặc nhiều giá trị theo thời gian.

Flux được sử dụng để xử lý các phép tính bất đồng bộ và dữ liệu reactive trong ứng dụng. Nó cung cấp các phép biến đổi mạnh mẽ để xử lý, biến đổi và kết hợp các luồng dữ liệu.



1. [Flux]- Just

Phương thức just(T...) trong lớp Flux được sử dụng để tạo một Flux phát ra các giá trị đã cho. Nó cho phép bạn tạo một luồng dữ liệu chứa các giá trị cố định và phát ra chúng theo thứ tự đã chỉ định.

Dưới đây là một ví dụ minh họa sử dụng phương thức just:

```
Flux<Integer> numbers = Flux.just(1, 2, 3, 4, 5);
numbers.subscribe(System.out::println);
```

Trong ví dụ trên, Flux.just(1, 2, 3, 4, 5) tạo một Flux phát ra các giá trị từ 1 đến 5 theo thứ tự đã chỉ định. Sau đó, numbers.subscribe(System.out::println) đăng ký một người nghe (subscriber) để nhận các giá trị phát ra từ Flux và in chúng ra màn hình.

Kết quả sẽ là:

```
1
2
3
4
5
```

Phương thức just cũng cho phép bạn truyền vào các đối tượng khác nhau của cùng một kiểu dữ liệu. Ví dụ:

```
Flux<String> words = Flux.just("Hello", "World");
words.subscribe(System.out::println);
```

Kết quả sẽ là:

```
Hello
World
```

Ví dụ về onNext, onError, onComplete

```
public static void main(String[] args) {

    Flux<Object> flux = Flux.just(1,2, 3, "a", Util.faker().name().fullName());

    flux.subscribe(
            Util.onNext(),
            Util.onError(),
            Util.onComplete());
}
```

2. [Flux]- Multiple Subscribers

Trong WebFlux, Flux hỗ trợ nhiều người nghe (subscriber) đăng ký và nhận dữ liệu từ cùng một luồng dữ liệu. Điều này mang lại khả năng chia sẻ dữ liệu và xử lý dữ liệu đồng thời cho nhiều thành phần khác nhau trong ứng dụng.

Khi bạn có một Flux và nhiều subscriber đăng ký, mỗi subscriber sẽ nhận dữ liệu theo cơ chế push khi Flux phát ra giá trị. Mỗi subscriber sẽ nhận các giá trị theo thứ tự và tốc độ Flux phát ra.

Dưới đây là một ví dụ về việc sử dụng nhiều người nghe (subscriber) cho một Flux trong WebFlux:

```
import reactor.core.publisher.Flux;

public class MultipleSubscribersExample {
    public static void main(String[] args) {
        Flux<Integer> numbers = Flux.range(1, 5);

        // Subscriber 1
        numbers.subscribe(value -> System.out.println("Subscriber 1: " + value));

        // Subscriber 2
        numbers.subscribe(value -> System.out.println("Subscriber 2: " + value));
    }
}
```

Trong ví dụ trên, chúng ta tạo một Flux gồm các số từ 1 đến 5 bằng cách sử dụng Flux.range(1, 5). Sau đó, chúng ta đăng ký hai người nghe (subscriber) khác nhau cho Flux này bằng cách sử dụng phương thức subscribe.

3. [Flux]- From Array / List

Trong framework Reactor của Java, Flux cung cấp một số phương thức như fromArray() và fromIterable() để tạo Flux từ một mảng (Array) hoặc một danh sách (List). Điều này cho phép chúng ta tạo Flux từ các tập hợp dữ liệu có sẵn.

Ví dụ sử dụng Flux.fromArray():

```
import reactor.core.publisher.Flux;

public class FluxFromArrayExample {
    public static void main(String[] args) {
        Integer[] numbersArray = {1, 2, 3, 4, 5};

        Flux<Integer> flux = Flux.fromArray(numbersArray);

        flux.subscribe(number -> System.out.println(number));
    }
}
```

Trong ví dụ trên, chúng ta có một mảng số nguyên numbersArray chứa các số từ 1 đến 5. Chúng ta sử dụng phương thức Flux.fromArray() để tạo một Flux từ mảng này. Sau đó, chúng ta đăng ký một người nghe (subscriber) cho Flux và in ra mỗi số nguyên nhận được.

Kết quả khi chạy chương trình sẽ là:

```
1
2
3
4
5
```

Ví dụ sử dụng Flux.fromIterable():

```
import reactor.core.publisher.Flux;

import java.util.Arrays;
import java.util.List;

public class FluxFromIterableExample {
    public static void main(String[] args) {
        List<String> fruitsList = Arrays.asList("Apple", "Banana", "Orange");

        Flux<String> flux = Flux.fromIterable(fruitsList);

        flux.subscribe(fruit -> System.out.println(fruit));
    }
}
```

Trong ví dụ trên, chúng ta có một danh sách fruitsList chứa các chuỗi "Apple", "Banana" và "Orange". Chúng ta sử dụng phương thức Flux.fromIterable() để tạo một Flux từ danh sách này. Sau đó, chúng ta đăng ký một người nghe (subscriber) cho Flux và in ra mỗi chuỗi nhận được.

Kết quả khi chạy chương trình sẽ là:

```
Apple
Banana
Orange
```

Như vậy, việc sử dụng Flux.fromArray() và Flux.fromIterable() cho phép chúng ta tạo Flux từ các tập hợp dữ liệu có sẵn như mảng hoặc danh sách, và tiếp tục xử lý dữ liệu theo kiểu reactive trên các luồng dữ liệu đó.

4. [Flux]- From Stream

Trong framework Reactor của Java, Flux cung cấp phương thức fromStream() để tạo Flux từ một luồng (Stream) của Java. Điều này cho phép chúng ta tạo Flux từ các luồng dữ liệu có sẵn, như luồng từ một tập hợp hoặc một phương thức tạo luồng tùy chỉnh.

Ví dụ sử dụng Flux.fromStream():

```
import reactor.core.publisher.Flux;

import java.util.stream.Stream;

public class FluxFromStreamExample {
    public static void main(String[] args) {
        Stream<Integer> numberStream = Stream.of(1, 2, 3, 4, 5);

        Flux<Integer> flux = Flux.fromStream(numberStream);

        flux.subscribe(number -> System.out.println(number));
//        flux
//                .subscribe(
//                        Util.onNext(),
//                        Util.onError(),
//                        Util.onComplete()
//                );
    }
}
```

Trong ví dụ trên, chúng ta có một luồng số nguyên numberStream từ 1 đến 5 bằng cách sử dụng Stream.of(). Chúng ta sử dụng phương thức Flux.fromStream() để tạo một Flux từ luồng này. Sau đó, chúng ta đăng ký một người nghe (subscriber) cho Flux và in ra mỗi số nguyên nhận được.

Kết quả khi chạy chương trình sẽ là:

```
1
2
3
4
5
```

5. [Flux]- Range

Trong framework Reactor của Java, Flux cung cấp phương thức range() để tạo Flux chứa một dãy giá trị từ một giá trị bắt đầu đến một giá trị kết thúc. Phương thức này hữu ích khi chúng ta muốn tạo một luồng dữ liệu có giá trị tăng dần hoặc giảm dần theo một quy tắc nhất định.

Ví dụ sử dụng Flux.range():

```
import reactor.core.publisher.Flux;

public class FluxRangeExample {
    public static void main(String[] args) {
        Flux<Integer> flux = Flux.range(1, 5);

        flux.subscribe(number -> System.out.println(number));
    }
}
```

Trong ví dụ trên, chúng ta sử dụng phương thức Flux.range() để tạo một Flux chứa dãy số nguyên từ 1 đến 5. Đối số đầu tiên của phương thức (1) là giá trị bắt đầu và đối số thứ hai (5) là số lượng giá trị trong dãy. Sau đó, chúng ta đăng ký một người nghe (subscriber) cho Flux và in ra mỗi số nguyên nhận được.

Kết quả khi chạy chương trình sẽ là:

```
1
2
3
4
5
```

Chú ý rằng giá trị cuối cùng trong dãy được bao gồm trong Flux. Trong ví dụ trên, dãy số được tạo bắt đầu từ 1 và kết thúc tại 5.

Ngoài ra, chúng ta cũng có thể sử dụng Flux.rangeLong() để tạo Flux với kiểu dữ liệu long. Ví dụ:

```
import reactor.core.publisher.Flux;

public class FluxRangeLongExample {
```

```
    public static void main(String[] args) {
        Flux<Long> flux = Flux.rangeLong(10, 3);

        flux.subscribe(number -> System.out.println(number));
    }
}
```

6. [Flux]- Log

Trong framework Reactor của Java, Flux cung cấp phương thức log() để ghi lại các sự kiện trong quá trình xử lý Flux. Phương thức này hữu ích để gỡ lỗi và theo dõi luồng dữ liệu trong quá trình reactive.

Khi sử dụng phương thức log(), bạn có thể chỉ định một thông điệp hoặc một hàm để tạo thông điệp ghi lại. Các thông điệp ghi lại sẽ được gửi đến hệ thống ghi (logger) được cấu hình trong ứng dụng của bạn.

Ví dụ sử dụng Flux.log():

```
import reactor.core.publisher.Flux;

public class FluxLogExample {
    public static void main(String[] args) {
        Flux<Integer> flux = Flux.range(1, 5)
                .log("FluxEvents");

        flux.subscribe(number -> System.out.println(number));
    }
}
```

Trong ví dụ trên, chúng ta sử dụng phương thức log() để ghi lại các sự kiện trong Flux. Đối số của phương thức ("FluxEvents") là thông điệp ghi lại mà chúng ta muốn gửi đến hệ thống ghi. Sau đó, chúng ta đăng ký một người nghe (subscriber) cho Flux và in ra mỗi số nguyên nhận được.

Kết quả khi chạy chương trình sẽ bao gồm các thông điệp ghi lại như sau:

```
/*
FluxEvents - | onSubscribe([Synchronous Fuseable] FluxRange.RangeSubscription)
FluxEvents - | request(unbounded)
FluxEvents - | onNext(1)
1
FluxEvents - | onNext(2)
2
FluxEvents - | onNext(3)
```

```
3
FluxEvents - | onNext(4)
4
FluxEvents - | onNext(5)
5
FluxEvents - | onComplete()
*/
```

Các thông điệp ghi lại cung cấp thông tin về các sự kiện quan trọng trong Flux, bao gồm quá trình đăng ký, yêu cầu dữ liệu, giá trị nhận được và hoàn thành Flux.

Bên cạnh việc chỉ định thông điệp cụ thể, bạn cũng có thể sử dụng log() mà không cung cấp thông điệp, trong trường hợp này nó sẽ sử dụng một thông điệp mặc định.

```
import reactor.core.publisher.Flux;

public class FluxLogExample {
    public static void main(String[] args) {
        Flux<Integer> flux = Flux.range(1, 5)
                .log();

        flux.subscribe(number -> System.out.println(number));
    }
}
```

Kết quả khi chạy chương trình sẽ bao gồm các thông điệp ghi lại mặc định tương tự như ví dụ trước.

Phương thức log() giúp bạn theo dõi quá trình xử lý Flux và kiểm tra các sự kiện xảy ra, giúp gỡ rối và phân tích lỗi trong ứng dụng của bạn.

7. [Flux]- Custom Subscriber Implementation

Khi làm việc với Flux trong Reactor, bạn có thể triển khai một Subscriber tùy chỉnh để xử lý dữ liệu và sự kiện từ Flux theo cách riêng của bạn. Điều này cho phép bạn điều khiển việc xử lý dữ liệu, xử lý lỗi và cung cấp các hành vi tùy chỉnh cho quá trình reactive.

Để triển khai một Subscriber tùy chỉnh, bạn cần implement interface Subscriber<T> và cung cấp các phương thức để xử lý các sự kiện quan trọng trong quá trình reactive, bao gồm:

- onSubscribe(Subscription): Được gọi khi Subscriber đăng ký nhận dữ liệu từ Publisher.
- onNext(T): Được gọi khi một giá trị mới được phát ra từ Publisher.
- onError(Throwable): Được gọi khi xảy ra một lỗi trong quá trình reactive.

- onComplete(): Được gọi khi Publisher hoàn thành việc phát dữ liệu.

Dưới đây là một ví dụ về cách triển khai một Subscriber tùy chỉnh:

```
import org.reactivestreams.Subscriber;
import org.reactivestreams.Subscription;

public class CustomSubscriber<T> implements Subscriber<T> {
    private Subscription subscription;

    @Override
    public void onSubscribe(Subscription subscription) {
        this.subscription = subscription;
        subscription.request(1); // Yêu cầu lấy một giá trị từ Publisher
    }

    @Override
    public void onNext(T value) {
        // Xử lý giá trị nhận được từ Publisher
        System.out.println("Received value: " + value);
        subscription.request(1); // Yêu cầu lấy thêm một giá trị từ Publisher
    }

    @Override
    public void onError(Throwable throwable) {
        // Xử lý lỗi nếu có
        System.err.println("Error occurred: " + throwable.getMessage());
    }

    @Override
    public void onComplete() {
        // Xử lý khi Publisher hoàn thành
        System.out.println("Completed");
    }
}
```

Trong ví dụ trên, chúng ta triển khai một Subscriber tùy chỉnh bằng cách implement interface Subscriber<T>. Trong phương thức onSubscribe(), chúng ta lưu trữ đối tượng Subscription để có thể yêu cầu lấy dữ liệu từ Publisher. Trong phương thức onNext(), chúng ta xử lý giá trị nhận được từ Publisher và yêu cầu lấy thêm giá trị (nếu cần). Trong phương thức onError() và onComplete(), chúng ta xử lý lỗi (nếu có) và khi quá trình reactive hoàn thành.

Sau khi triển khai Subscriber tùy chỉnh, bạn có thể sử dụng nó để đăng ký nhận dữ liệu từ Flux:

```
import reactor.core.publisher.Flux;

public class CustomSubscriberExample {
    public static void main(String[] args) {
        Flux<Integer> flux = Flux.range(1, 5);

        CustomSubscriber<Integer> subscriber = new CustomSubscriber<>();
        flux.subscribe(subscriber);
    }
}
```

Trong ví dụ trên, chúng ta tạo một Flux chứa dãy số từ 1 đến 5. Sau đó, chúng ta tạo một đối tượng CustomSubscriber và đăng ký nó để nhận dữ liệu từ Flux. Khi Flux phát ra giá trị, các phương thức tương ứng trong Subscriber sẽ được gọi.

Đây là đầu ra mà bạn có thể mong đợi từ ví dụ trên:

```
/*
Received value: 1
Received value: 2
Received value: 3
Received value: 4
Received value: 5
Completed

 */
```

Bằng cách triển khai một Subscriber tùy chỉnh, bạn có thể kiểm soát quá trình xử lý dữ liệu và cung cấp các hành vi tùy chỉnh các sự kiện trong quá trình reactive.

Ví dụ khác:

```
public class Lec06Subscription {

    public static void main(String[] args) {

        AtomicReference<Subscription> atomicReference = new AtomicReference<>();
        Flux.range(1, 20)
                .log()
                .subscribeWith(new Subscriber<Integer>() {
                    @Override
                    public void onSubscribe(Subscription subscription) {
```

```
                        System.out.println("Received Sub : " + subscription);
                        atomicReference.set(subscription);
                    }

                    @Override
                    public void onNext(Integer integer) {
                        System.out.println("onNext : " + integer);
                    }

                    @Override
                    public void onError(Throwable throwable) {
                        System.out.println("onError : " +
throwable.getMessage());
                    }

                    @Override
                    public void onComplete() {
                        System.out.println("onComplete");
                    }
                });

        Util.sleepSeconds(3);
        atomicReference.get().request(3);
        Util.sleepSeconds(5);
        atomicReference.get().request(3);
        Util.sleepSeconds(5);
        System.out.println("going to cancel");
        atomicReference.get().cancel();
        Util.sleepSeconds(3);
        atomicReference.get().request(4);

        Util.sleepSeconds(3);
    }
}
```

8. Flux vs List

Flux và List là hai cấu trúc dữ liệu khác nhau và được sử dụng trong ngữ cảnh và mục đích khác nhau.

1. Phạm vi sử dụng:
   - List: Thường được sử dụng để lưu trữ và xử lý danh sách các phần tử theo thứ tự cụ thể. Thích hợp cho các tác vụ liên quan đến thêm, xóa, truy cập và tìm kiếm phần tử trong danh sách.
   - Flux: Thường được sử dụng trong lập trình reactive để xử lý dữ liệu theo cách không đồng bộ. Thích hợp cho các tác vụ xử lý dữ liệu và sự kiện theo kiểu reactive, như ánh xạ, lọc, gom nhóm và xử lý lỗi.
2. Tính chất dữ liệu:
   - List: Lưu trữ các phần tử theo thứ tự cụ thể và có thể chứa các phần tử trùng lặp.
   - Flux: Phát ra một chuỗi các phần tử theo kiểu "cold" hoặc "hot" và không có khái niệm vị trí hoặc phần tử trùng lặp.
3. Xử lý dữ liệu:
   - List: Cung cấp các phương thức để thêm, xóa, truy cập và tìm kiếm các phần tử trong danh sách. Có thể sử dụng các phương thức như add(), remove(), get(), contains(), size() và nhiều hơn nữa.
   - Flux: Cung cấp các phương thức để xử lý dữ liệu và sự kiện theo kiểu reactive. Có thể sử dụng các phương thức như map(), filter(), reduce(), flatMap(), zip() và nhiều hơn nữa.
4. Đặc điểm reactive:
   - List: Không có tính chất reactive tích hợp sẵn. Cần sử dụng các phương thức hoặc thư viện bên ngoài để xử lý dữ liệu theo cách reactive.
   - Flux: Là một phần của framework Reactor và hỗ trợ các phép xử lý dữ liệu theo kiểu reactive như ánh xạ, lọc, gom nhóm và xử lý lỗi.

9. [Flux]- Interval

Flux.interval() là một phương thức trong framework Reactor của Java, được sử dụng để tạo một Flux phát ra các phần tử trong một khoảng thời gian nhất định. Nó tạo ra một chuỗi các phần tử theo thời gian và thường được sử dụng trong các tác vụ liên quan đến xử lý dữ liệu theo chu kỳ hoặc thời gian.

Cú pháp phương thức interval() trong Reactor là:

```
public static Flux<Long> interval(Duration period)
```

Cú pháp trên tạo ra một Flux phát ra các giá trị kiểu Long mỗi period thời gian. period được xác định bằng đối tượng Duration, cho phép bạn chỉ định đơn vị thời gian như giây, mili-giây, nano-giây, v.v.

Dưới đây là một ví dụ sử dụng Flux.interval() để phát ra các số nguyên từ 1 đến 5 mỗi giây một lần:

```
import reactor.core.publisher.Flux;

import java.time.Duration;

public class IntervalExample {
    public static void main(String[] args) throws InterruptedException {
        Flux.interval(Duration.ofSeconds(1))
                .map(count -> count + 1)
                .take(5)
                .subscribe(System.out::println);

        Thread.sleep(6000);
    }
}
```

Kết quả của đoạn mã trên sẽ là:

```
/*
1
2
3
4
5
 */
```

Trong ví dụ trên, chúng ta sử dụng Flux.interval() để tạo một Flux phát ra các giá trị sau mỗi giây. Sau đó, chúng ta áp dụng phép ánh xạ (map()) để tăng giá trị của mỗi phần tử lên 1. Tiếp theo, chúng ta sử dụng phương thức take(5) để chỉ lấy 5 phần tử đầu tiên và sau đó đăng ký một Subscriber để in ra các phần tử.

Lưu ý rằng chúng ta sử dụng Thread.sleep(6000) để dừng chương trình trong 6 giây để cho phép Flux phát ra đủ 5 phần tử. Nếu không có dòng này, chương trình sẽ kết thúc trước khi Flux phát ra tất cả các phần tử.

10. [Flux]- From Mono / Publisher

Trong Reactor, Flux.from() và Flux.fromIterable() là hai phương thức được sử dụng để tạo một Flux từ một Mono hoặc một Publisher. Điều này cho phép chuyển đổi từ một Mono hoặc một Publisher thành một Flux để thực hiện các phép xử lý dữ liệu theo kiểu reactive.

1. Flux.from(Mono):
   - Phương thức from() trong Flux được sử dụng để tạo một Flux từ một Mono.
   - Mono là một loại Publisher trong Reactor, nhưng chỉ phát ra một giá trị hoặc một lỗi.
   - Khi sử dụng Flux.from(Mono), Mono sẽ được chuyển đổi thành Flux và phát ra giá trị của nó sau đó kết thúc.
   - Điều này hữu ích khi bạn muốn sử dụng các phép xử lý dữ liệu của Flux trên một Mono hoặc kết hợp nhiều Mono thành một Flux.
2. Flux.fromIterable(Iterable):
   - Phương thức fromIterable() trong Flux được sử dụng để tạo một Flux từ một Iterable.
   - Iterable là một giao diện trong Java, đại diện cho một tập hợp các phần tử có thể lặp lại.
   - Khi sử dụng Flux.fromIterable(Iterable), Iterable sẽ được chuyển thành Flux và phát ra các phần tử của nó theo thứ tự.
   - Điều này hữu ích khi bạn muốn sử dụng các phép xử lý dữ liệu của Flux trên một danh sách các phần tử có thể lặp lại.

Dưới đây là một ví dụ minh họa:

```
import reactor.core.publisher.Flux;
import reactor.core.publisher.Mono;

import java.util.Arrays;
import java.util.List;

public class FluxFromExample {
    public static void main(String[] args) {
        Mono<String> mono = Mono.just("Hello");
        Flux<String> fluxFromMono = Flux.from(mono);
        fluxFromMono.subscribe(System.out::println);

        List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);
        Flux<Integer> fluxFromIterable = Flux.fromIterable(numbers);
        fluxFromIterable.map(num -> num * 2)
                .subscribe(System.out::println);
    }
}
```

Kết quả của đoạn mã trên sẽ là:

```
/*
Hello
2
4
6
8
10

 */
```

Trong ví dụ trên, chúng ta sử dụng Flux.from() để tạo một Flux từ một Mono. Mono mono chứa giá trị "Hello", và sau khi chuyển đổi thành Flux, nó phát ra giá trị "Hello" và kết thúc.

Chúng ta cũng sử dụng Flux.fromIterable() để tạo một Flux từ một danh sách số nguyên. Flux này phát ra các phần tử của danh sách theo thứ tự và sau đó áp dụng phép ánh xạ (map()) để nhân mỗi số lên 2. Kết quả là Flux phát ra các số nhân lên 2 là 2, 4, 6, 8 và 10.

11. [Flux]- To Mono

Trong Reactor, phương thức Flux.to(Mono) được sử dụng để chuyển đổi một Flux thành một Mono. Điều này cho phép chuyển đổi từ một Flux chứa nhiều phần tử thành một Mono chỉ chứa một phần tử đầu tiên của Flux hoặc một giá trị mặc định.

Cú pháp phương thức to() trong Reactor là:

```
public final Mono<T> to(Mono<? extends T> other)
```

Dưới đây là một ví dụ minh họa:

```
import reactor.core.publisher.Flux;
import reactor.core.publisher.Mono;

public class FluxToMonoExample {
    public static void main(String[] args) {
        Flux<Integer> flux = Flux.just(1, 2, 3, 4, 5);
        Mono<Integer> mono = flux.reduce((a, b) -> a + b);
        mono.subscribe(System.out::println);
    }
}
```

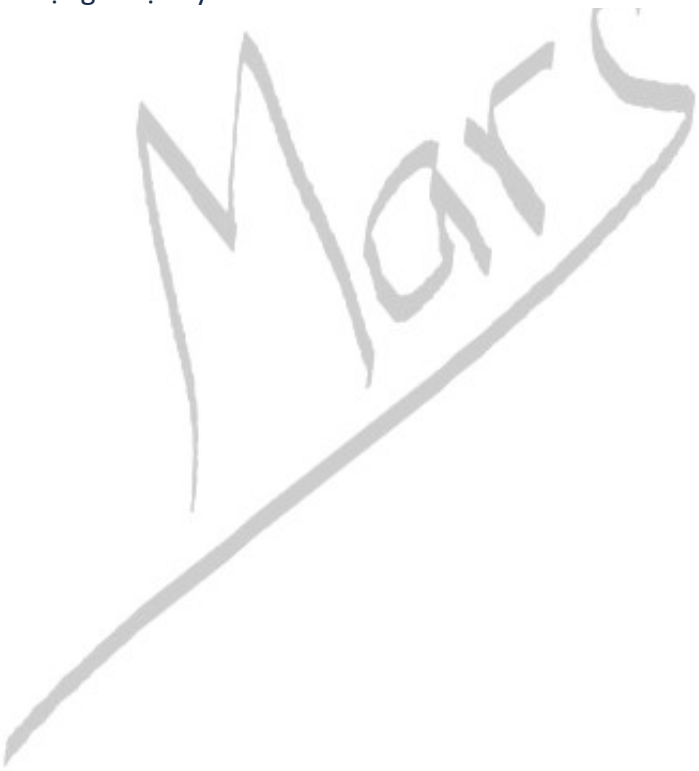
Kết quả của đoạn mã trên sẽ là:

```
/*
Copy
15
 */
```

Trong ví dụ trên, chúng ta có một Flux flux chứa các số nguyên từ 1 đến 5. Chúng ta sử dụng phương thức reduce() để tính tổng của tất cả các số trong Flux. Kết quả của phép tính toán được chuyển đổi thành Mono mono bằng cách sử dụng to().

Sau đó, chúng ta đăng ký một Subscriber với Mono mono để in ra kết quả tổng là 15.

Phương thức to(Mono) rất hữu ích khi bạn muốn chuyển đổi một Flux thành một Mono để thực hiện các phép xử lý dữ liệu theo kiểu reactive chỉ trên một phần tử đầu tiên của Flux hoặc kết hợp các phần tử thành một giá trị duy nhất.



Chapter 5. Flux- Emitting Items Programmatically

1. Default Subscriber Implementation

Trong Reactive Streams, triển khai một Subscriber mặc định có thể được sử dụng để cung cấp một triển khai chuẩn cho giao diện Subscriber. Một Subscriber được sử dụng để nhận dữ liệu từ Publisher và xử lý nó theo cách tùy chỉnh.

Mục đích của việc triển khai một Subscriber mặc định trong Reactive Streams là:

- Cung cấp một triển khai mặc định cho giao diện Subscriber: Một Subscriber mặc định có thể triển khai các phương thức của giao diện Subscriber và cung cấp hành vi mặc định cho chúng. Điều này giúp giảm công việc lặp lại khi triển khai các Subscriber riêng lẻ và cung cấp một triển khai chuẩn cho các tình huống thông thường.
- Định nghĩa hành vi mặc định cho các phương thức: Một Subscriber mặc định có thể cung cấp triển khai mặc định cho các phương thức trong giao diện Subscriber, bao gồm các phương thức như onNext(), onError(), onComplete(), và onSubscribe(). Điều này cho phép các lớp con kế thừa từ Subscriber mặc định và ghi đè các phương thức cụ thể mà nó quan tâm để xử lý dữ liệu hoặc thực hiện các hành động khác.

Dưới đây là một ví dụ đơn giản về cách triển khai một:

```
public class DefaultSubscriber<T> implements Subscriber<T> {
    @Override
    public void onSubscribe(Subscription subscription) {
        // Xử lý việc đăng ký Subscription
    }

    @Override
    public void onNext(T item) {
        // Xử lý giá trị dữ liệu tiếp theo
    }

    @Override
    public void onError(Throwable throwable) {
        // Xử lý lỗi
    }

    @Override
    public void onComplete() {
        // Xử lý khi hoàn thành
    }
}
```

Trong ví dụ trên, lớp DefaultSubscriber triển khai giao diện Subscriber và cung cấp triển khai mặc định cho các phương thức onSubscribe(), onNext(), onError(), onComplete(). Bạn có thể tạo các lớp con mới kế thừa từ DefaultSubscriber và ghi đè các phương thức cụ thể để xử lý dữ liệu hoặc thực hiện các hành động khác.

2. Flux Create

Trong Reactor, Flux.create() là một phương thức để tạo một Flux tùy chỉnh bằng cách triển khai logic xử lý và phát(emit) các giá trị. Phương thức này cho phép bạn định nghĩa logic phát ra dữ liệu và quản lý các sự kiện trong Flux.

Mục đích của Flux.create() là để tạo ra một Flux có thể tạo và emit dữ liệu theo logic tùy chỉnh của bạn. Bằng cách sử dụng Flux.create(), bạn có thể tạo Flux từ các nguồn dữ liệu không đồng bộ, ví dụ như gọi các API bên ngoài, đọc từ tệp tin, hoặc kết nối với các hệ thống khác.

Dưới đây là một ví dụ đơn giản về cách sử dụng Flux.create():

```
import reactor.core.publisher.Flux;

public class FluxCreateExample {
    public static void main(String[] args) {
        Flux<Integer> flux = Flux.create(sink -> {
            for (int i = 1; i <= 5; i++) {
                // Phát ra giá trị
                sink.next(i);
            }

            // Khi hoàn thành, đánh dấu Flux đã hoàn thành
            sink.complete();
        });

        flux.subscribe(value -> {
            // Xử lý giá trị từ Flux
            System.out.println("Received value: " + value);
        });
    }
}
```

Trong ví dụ trên, chúng ta sử dụng Flux.create() để tạo một Flux tùy chỉnh. Trong lambda expression, chúng ta triển khai sink để phát ra các giá trị từ 1 đến 5 bằng cách sử dụng sink.next(i). Sau đó, chúng ta gọi sink.complete() để đánh dấu rằng Flux đã hoàn thành.

Sau đó, chúng ta sử dụng flux.subscribe() để đăng ký một hàm callback để nhận và xử lý các giá trị từ Flux. Trong hàm callback, chúng ta in ra các giá trị nhận được từ Flux.

Kết quả sẽ là:

```
Received value: 1
Received value: 2
Received value: 3
Received value: 4
Received value: 5
```

Ví dụ khác:

```
public static void main(String[] args) {

    Flux.create(fluxSink -> {
        String country;
        do{
            country = Util.faker().country().name();
            fluxSink.next(country);
        }while (!country.toLowerCase().equals("canada"));
        fluxSink.complete();
    })
    .subscribe(Util.subscriber());

}
```

3. FluxSink- Sharing With Multiple Threads

Trong ngữ cảnh của Reactive Streams, FluxSink là một đối tượng được sử dụng để tạo và gửi dữ liệu trong một luồng (stream). Đối tượng FluxSink cho phép chia sẻ dữ liệu với nhiều luồng (threads) một cách an toàn.

Khi bạn muốn tạo một luồng dữ liệu đa luồng (multithreaded) và gửi dữ liệu từ nhiều luồng đến đối tượng Flux (ví dụ: để xử lý các tác vụ đồng thời), bạn có thể sử dụng FluxSink để chia sẻ và gửi dữ liệu một cách an toàn giữa các luồng.

Dưới đây là một ví dụ đơn giản về việc sử dụng FluxSink để chia sẻ dữ liệu với nhiều luồng:

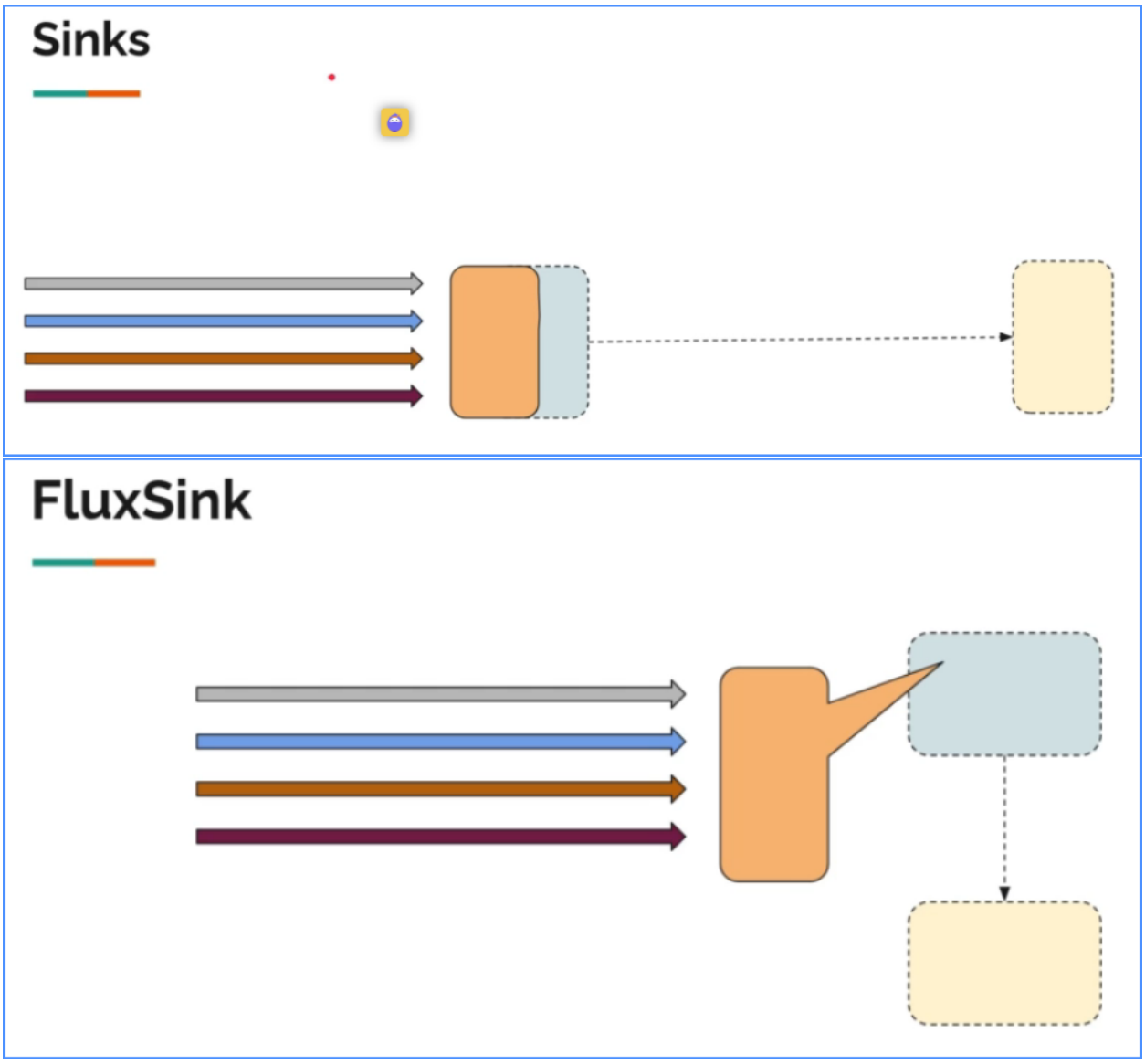
```
public class NameProducer implements Consumer<FluxSink<String>> {

    private FluxSink<String> fluxSink;

    @Override
    public void accept(FluxSink<String> stringFluxSink) {
        this.fluxSink = stringFluxSink;
    }

    public void produce(){
        String name = Util.faker().name().fullName();
        String thread = Thread.currentThread().getName();
        this.fluxSink.next(thread + " : " + name);
    }

}

public class Lec02FluxCreateRefactoring {

    public static void main(String[] args) {

        NameProducer nameProducer = new NameProducer();

        Flux.create(nameProducer)
                .subscribe(Util.subscriber());

        Runnable runnable = nameProducer::produce;

        for (int i = 0; i < 10; i++) {
            new Thread(runnable).start();
        }

        Util.sleepSeconds(2);

    }
}
```

Trong ví dụ trên, chúng ta tạo một đối tượng Flux bằng cách sử dụng phương thức tĩnh create() và truyền một lambda vào đó. Lambda này nhận một đối tượng FluxSink để gửi dữ liệu.

Output

```
/*
Received : Thread-5 : Aron Dickens
Received : Thread-2 : Siu Kertzmann
Received : Thread-7 : Gennie Runolfsdottir
Received : Thread-8 : Abraham Schaden
Received : Thread-0 : Juliet Abshire
Received : Thread-1 : Luana Williamson
Received : Thread-4 : Gregg Waters
Received : Thread-3 : Cary Zulauf
Received : Thread-6 : Javier Hagenes
Received : Thread-9 : Derrick Stoltenberg

 */
```

4. FluxSink vs Sinks



FluxSink và Sinks đều là các công cụ cung cấp bởi Reactor Core để tạo và quản lý Flux hoặc Mono. Tuy nhiên, chúng có một số khác biệt quan trọng:

- **FluxSink**: FluxSink là một đối tượng được sử dụng để gửi dữ liệu tới Flux. Bạn có thể sử dụng FluxSink để gửi các giá trị thông qua phương thức next() và kết thúc Flux bằng cách gọi complete(). FluxSink cung cấp một giao diện tiếp cận đơn giản và trực tiếp cho việc tạo và quản lý dữ liệu trong một Flux.
- **Sinks**: Sinks là một lớp trừu tượng hỗ trợ tạo và quản lý Flux và Mono. Nó cung cấp các phương thức tạo Flux (many()) và Mono (one()) thông qua các phương thức tạo tương ứng SinkMany và SinkMono. Sinks cũng hỗ trợ xử lý nhiều dữ liệu đầu vào thông qua SinkMany để gửi các giá trị thông qua phương thức emitNext(), tryEmitNext() và kết thúc Flux bằng cách gọi emitComplete().

- **Tính linh hoạt**: Với FluxSink, bạn có thể tạo một Flux và trực tiếp gửi dữ liệu từ một nguồn đơn tại thời điểm tạo Flux. Trong khi đó, với Sinks, bạn có thể tạo một SinkMany hoặc SinkMono và gửi dữ liệu từ nhiều nguồn hoặc trong khoảng thời gian dài sau khi tạo Sinks. Sinks cung cấp khả năng kiểm soát linh hoạt hơn về việc gửi dữ liệu theo nhu cầu.
- **Quản lý đa luồng**: FluxSink không cung cấp các tính năng xử lý đa luồng một cách tự động. Bạn cần tự quản lý việc sử dụng FluxSink trong các luồng khác nhau nếu muốn chia sẻ dữ liệu trên nhiều luồng. Trong khi đó, Sinks hỗ trợ xử lý đa luồng thông qua các phương thức thread-safe như emitNext() và tryEmitNext(), cho phép bạn gửi dữ liệu từ nhiều luồng một cách an toàn.

5. Take Operator

Toàn tử take() trong Reactor Core được sử dụng để giới hạn số lượng phần tử được phát ra từ một Flux ban đầu. Nó cho phép bạn chỉ định một số lượng cụ thể hoặc một điều kiện để giới hạn dữ liệu được phát ra.

Cú pháp chung của toán tử take() là:

```
public final Flux<T> take(long n)
```

Nơi n là số lượng phần tử bạn muốn giới hạn. Khi Flux phát ra n phần tử, nó sẽ kết thúc và không phát ra bất kỳ phần tử nào sau đó.

Dưới đây là một ví dụ minh họa:

```
Flux.range(1, 10)
    .take(5)
    .subscribe(System.out::println);
```

Trong ví dụ trên, chúng ta tạo một Flux bằng cách sử dụng Flux.range() để phát ra các số từ 1 đến 10. Sau đó, chúng ta sử dụng toán tử take(5) để giới hạn Flux chỉ phát ra 5 phần tử đầu tiên. Kết quả là chỉ có các số từ 1 đến 5 được in ra màn hình.

Output:

```
1
2
3
4
5
```

Toán tử take() cũng có thể được sử dụng với một số điều kiện khác nhau, chẳng hạn như giới hạn dựa trên một điều kiện predicate hoặc thời gian nhất định. Ví dụ:

```
Flux.interval(Duration.ofSeconds(1))
    .take(Duration.ofSeconds(5))
    .subscribe(System.out::println);
```

Trong ví dụ trên, chúng ta sử dụng Flux.interval() để phát ra các phần tử liên tục với khoảng thời gian 1 giây. Tuy nhiên, chúng ta sử dụng toán tử take(Duration.ofSeconds(5)) để chỉ cho Flux phát ra trong 5 giây. Kết quả là sau 5 giây, Flux sẽ kết thúc và không phát ra bất kỳ phần tử nào sau đó.

Output (sau 5 giây):

```
0
1
2
3
4
```

6. Flux Create- Check If Down Stream Is Cancelled

Phương thức Flux.create() trong Reactor Core cho phép bạn tạo một Flux tùy chỉnh bằng cách cung cấp một triển khai của giao diện FluxSink. Bằng cách sử dụng FluxSink, bạn có thể tạo và quản lý quá trình phát dữ liệu của Flux một cách linh hoạt.

Một trong những tình huống phổ biến khi sử dụng Flux.create() là kiểm tra xem down stream (subscriber) đã bị hủy bỏ hay chưa. Điều này có thể hữu ích trong việc ngừng sản xuất và phát dữ liệu khi không còn người tiêu dùng (subscriber) nào còn quan tâm tới nó nữa. Điều này giúp tiết kiệm tài nguyên và tránh việc phát ra dữ liệu không cần thiết.

```
Flux.create(fluxSink -> {
    String country;
    int counter = 0;
    do{
        country = Util.faker().country().name();
        System.out.println("emitting : " + country);
        fluxSink.next(country);
        counter++;
    }while (!country.toLowerCase().equals("canada") && !fluxSink.isCancelled() &&
counter < 10);
    fluxSink.complete();
})
.take(3)
.subscribe(Util.subscriber());
```

Trong ví dụ trên, chúng ta sử dụng Flux.create() và triển khai FluxSink trong lambda function. Trong quá trình sản xuất dữ liệu, chúng ta kiểm tra liên tục trạng thái hủy bỏ của downstream bằng cách sử dụng phương thức isCancelled() của FluxSink.

Trong ví dụ trên, chúng ta sử dụng Flux.create() và triển khai FluxSink trong lambda function. Trong quá trình sản xuất dữ liệu, chúng ta kiểm tra liên tục trạng thái hủy bỏ của downstream bằng cách sử dụng phương thức isCancelled() của FluxSink.

7. Flux Generate

Phương thức Flux.generate() trong Reactor's Flux được sử dụng để tạo ra một chuỗi dữ liệu tuần tự dựa trên một trạng thái ban đầu và một logic để tạo ra giá trị và cập nhật trạng thái. Nó có thể được sử dụng để tạo ra các chuỗi dữ liệu không đồng bộ theo logic tùy chỉnh.

Cú pháp của phương thức Flux.generate() như sau:

```
public static <S, T> Flux<T> generate(Supplier<S> initialState, BiFunction<S,
SynchronousSink<T>, S> generator)
```

- initialState: Một lambda expression hoặc một đối tượng Supplier, đại diện cho trạng thái ban đầu của quá trình tạo dữ liệu.
- generator: Một lambda expression hoặc một đối tượng BiFunction, đại diện cho logic để tạo ra các giá trị trong chuỗi dữ liệu và cập nhật trạng thái. Nó nhận vào hai đối số: trạng thái hiện tại và một đối tượng SynchronousSink để phát ra giá trị và điều khiển quá trình tạo dữ liệu.

Trong lambda expression của generator, chúng ta có thể thực hiện các hoạt động như phát ra giá trị bằng cách sử dụng synchronousSink.next(), kết thúc chuỗi dữ liệu bằng cách sử dụng synchronousSink.complete(), hoặc phát sinh lỗi bằng cách sử dụng synchronousSink.error().

Phương thức Flux.generate() cho phép chúng ta tạo ra các chuỗi dữ liệu theo logic tùy chỉnh và cập nhật trạng thái trong mỗi lần lặp. Nó rất hữu ích khi chúng ta muốn tạo ra các chuỗi dữ liệu không đồng bộ hoặc tùy chỉnh theo logic riêng của chúng ta.

```
Flux.generate(synchronousSink -> {
    System.out.println("emitting");
    synchronousSink.next(Util.faker().country().name()); // 1
    //synchronousSink.error(new RuntimeException("oops"));
})
.take(2)
.subscribe(Util.subscriber());
```

Trong đoạn code trên, chúng ta sử dụng phương thức Flux.generate() để tạo một chuỗi dữ liệu bằng cách sử dụng một đối tượng SynchronousSink. Hãy tìm hiểu cách nó hoạt động:

1. Đầu tiên, chúng ta gọi phương thức Flux.generate() và truyền vào một lambda expression. Lambda expression này nhận một đối tượng synchronousSink, đại diện cho sink (nơi phát ra giá trị) đồng bộ.

2. Trong thân của lambda expression, chúng ta có một số logic để tạo ra giá trị trong chuỗi dữ liệu. Trong ví dụ này, chúng ta sử dụng phương thức Util.faker().country().name() để lấy tên một quốc gia ngẫu nhiên và phát ra giá trị đó bằng cách sử dụng synchronousSink.next().
3. Trong ví dụ này, chúng ta sử dụng phương thức take(2) sau Flux.generate() để chỉ lấy 2 giá trị từ chuỗi dữ liệu được tạo ra.
4. Cuối cùng, chúng ta gọi phương thức subscribe() và truyền vào một người tiêu dùng, được đại diện bởi Util.subscriber(), để tiêu thụ các giá trị trong chuỗi dữ liệu.

Ví dụ tạo ra một chuỗi dữ liệu tuần tự dựa trên một trạng thái ban đầu và một logic để tạo ra giá trị và cập nhật trạng thái:

```
Flux.generate(
    () -> 0, // Trạng thái ban đầu
    (state, sink) -> {
        sink.next("Element " + state); // Sinh ra phần tử dựa trên trạng thái
        if (state == 9) {
            sink.complete(); // Kết thúc khi đạt đến trạng thái 9
        }
        return state + 1; // Trả về trạng thái mới
    })
    .subscribe(System.out::println);
```

Trong ví dụ trên, chúng ta sử dụng Flux.generate() để tạo một Flux sinh ra các phần tử "Element 0" đến "Element 9". Trạng thái ban đầu được cung cấp bởi () -> 0, và hàm sinh dữ liệu nhận trạng thái hiện tại và FluxSink để sinh ra phần tử tiếp theo.

Trong hàm sinh dữ liệu, chúng ta sử dụng sink.next() để sinh ra phần tử dựa trên trạng thái hiện tại. Nếu trạng thái là 9, chúng ta gọi sink.complete() để kết thúc Flux. Cuối cùng, chúng ta trả về trạng thái mới bằng cách tăng giá trị trạng thái hiện tại lên 1.

Output:

```
/*
Element 0
Element 1
Element 2
Element 3
Element 4
Element 5
Element 6
Element 7
```

```
Element 8
Element 9
 */
```

Thông qua Flux.generate(), bạn có thể tạo ra các Flux tùy chỉnh với quy trình sinh dữ liệu linh hoạt. Bạn có thể sử dụng các điều kiện, logic và luồng điều khiển phức tạp để tạo và phát ra các phần tử theo nhu cầu của ứng dụng của bạn.

8. Flux Push

Trong Reactor's Flux, phương thức push() được sử dụng để tạo ra một Flux từ một nguồn dữ liệu bên ngoài. Nó cho phép chúng ta đẩy các giá trị vào Flux một cách tùy ý từ mã nguồn bất đồng bộ.

Phương thức push() có thể được sử dụng để tạo ra Flux từ nhiều nguồn dữ liệu như sự kiện, callback, hoặc luồng dữ liệu không đồng bộ khác. Khi có dữ liệu mới, chúng ta có thể đẩy nó vào Flux bằng cách sử dụng Sink, và Flux sẽ phát ra các giá trị đó cho các bộ tiêu thụ đã đăng ký.

Dưới đây là một ví dụ minh họa về cách sử dụng phương thức push():

```
FluxSink<Integer> sink = Flux.push(sink -> {
    // Đẩy các giá trị vào Flux
    sink.next(1);
    sink.next(2);
    sink.next(3);
    sink.complete();
});

sink.subscribe(System.out::println); // In các giá trị từ Flux
public static void main(String[] args) {
        NameProducer nameProducer = new NameProducer();
        Flux.push(nameProducer)
                .subscribe(Util.subscriber());
        Runnable runnable = nameProducer::produce;
        for (int i = 0; i < 10; i++) {
            new Thread(runnable).start();
        }
        Util.sleepSeconds(2);

}
```

=> Sử dụng mỗi thread duy nhất cho mỗi người dùng, nếu dùng create thì sẽ tạo ra 1 luồng được sử dụng lại


Mars

Chapter 6. Operators

1. Operator- Handle

Trong WebFlux, operator handle được sử dụng để thực hiện xử lý tùy chỉnh trên từng phần tử của chuỗi dữ liệu và phát sinh các giá trị mới. Điều này cho phép chúng ta thực hiện xử lý logic phức tạp và tạo ra các giá trị tùy ý trong quá trình xử lý dữ liệu.

Cú pháp của operator handle trong WebFlux như sau:

```
Flux<T> handle(BiConsumer<T, SynchronousSink<R>> handler)
```

Trong đó:

- T là kiểu dữ liệu của các phần tử trong chuỗi dữ liệu ban đầu.
- R là kiểu dữ liệu của các phần tử được tạo ra trong quá trình xử lý.
- handler là một BiConsumer (hàm tiêu thụ hai tham số), nhận một phần tử từ chuỗi dữ liệu ban đầu và một SynchronousSink để phát sinh các giá trị mới.

Dưới đây là một ví dụ minh họa về cách sử dụng operator handle trong WebFlux:

```
 Flux.range(1, 5).handle((number, sink) -> {
    if (number % 2 == 0) {
        sink.next("Even: " + number);
    } else {
        sink.next("Odd: " + number);
    }
}).subscribe(System.out::println);
```

// Ouput

```
/*
Odd: 1
Even: 2
Odd: 3
Even: 4
Odd: 5
 */
```

Trong ví dụ trên, chúng ta tạo ra một Flux numbers bằng cách sử dụng Flux.range(1, 5) để tạo ra chuỗi các số từ 1 đến 5.

Sau đó, chúng ta sử dụng operator handle để xử lý từng phần tử trong Flux numbers. Trong hàm handle, chúng ta kiểm tra xem số đó có phải số chẵn hay số lẻ và sử dụng SynchronousSink để phát sinh các giá trị mới.

Cuối cùng, chúng ta đăng ký một bộ tiêu thụ với Flux transformedNumbers và in các giá trị đã được xử lý ra màn hình.

Kết quả sẽ là các chuỗi "Odd: 1", "Even: 2", "Odd: 3", "Even: 4" và "Odd: 5" được in ra màn hình, tương ứng với việc xử lý và phát sinh các giá trị mới dựa trên từng phần tử của Flux numbers.

Operator handle cho phép chúng ta thực hiện xử lý linh hoạt và tạo ra các giá trị mới trong quá trình xử lý dữ liệu.

2. Operator- Do Hooks

Trong WebFlux, operator doOn và các hooks tương ứng (doOnNext, doOnComplete, doOnError, doOnSubscribe, doOnRequest, doOnCancel, doOnTerminate, doFinally) được sử dụng để thực hiện các hành động phụ (side effects) trong quá trình xử lý dữ liệu trong chuỗi Flux hoặc Mono.

Các operator doOn và hooks tương ứng cho phép chúng ta gắn kết các hành động phụ vào các sự kiện xảy ra trong quá trình xử lý dữ liệu như khi một phần tử mới được phát ra, hoàn thành quá trình xử lý, xảy ra lỗi, và các sự kiện khác.

Dưới đây là một số ví dụ về cách sử dụng operator doOn và hooks tương ứng trong WebFlux:

```
Flux<Integer> numbers = Flux.range(1, 5);

numbers
    .doOnNext(number -> System.out.println("Processing number: " + number))
    .doOnComplete(() -> System.out.println("Processing completed"))
    .subscribe(System.out::println);
```

Trong ví dụ trên, chúng ta tạo ra một Flux numbers bằng cách sử dụng Flux.range(1, 5) để tạo ra một chuỗi các số từ 1 đến 5.

Sau đó, chúng ta sử dụng operator doOnNext để gắn kết một hành động phụ (System.out.println) khi một phần tử mới được phát ra từ Flux. Trong trường hợp này, chúng ta đơn giản in ra thông báo "Processing number" kèm theo giá trị của phần tử.

Chúng ta cũng sử dụng hook doOnComplete để gắn kết một hành động phụ để được thực thi khi quá trình xử lý hoàn thành. Trong trường hợp này, chúng ta in ra thông báo "Processing completed".

Cuối cùng, chúng ta đăng ký một bộ tiêu thụ với Flux và in các giá trị từ Flux ra màn hình.

Kết quả sẽ là các giá trị từ Flux được in ra màn hình, và các thông báo "Processing number" và "Processing completed" được in ra trước và sau quá trình xử lý dữ liệu.

Operator doOn và hooks tương ứng trong WebFlux cho phép chúng ta thực hiện các hành động phụ trong quá trình xử lý dữ liệu như ghi log, thống kê, gửi thông báo, và các hành động khác để theo dõi và kiểm soát quá trình xử lý.

Ví dụ đầy đủ:

```
Flux.create(fluxSink -> {
    System.out.println("inside create");
    for (int i = 0; i < 5; i++) {
        fluxSink.next(i);
    }
    // fluxSink.complete();
    fluxSink.error(new RuntimeException("oops"));
    System.out.println("--completed");
})
.doOnComplete(() -> System.out.println("doOnComplete"))
.doFirst(() -> System.out.println("doFirst"))
.doOnNext(o -> System.out.println("doOnNext : " + o))
.doOnSubscribe(s -> System.out.println("doOnSubscribe" + s))
.doOnRequest(l -> System.out.println("doOnRequest : " + l))
.doOnError(err -> System.out.println("doOnError : " + err.getMessage()))
.doOnTerminate(() -> System.out.println("doOnTerminate"))
.doOnCancel(() -> System.out.println("doOnCancel"))
.doFinally(signal -> System.out.println("doFinally 1 : " + signal))
.doOnDiscard(Object.class, o -> System.out.println("doOnDiscard : " + o))
.take(2)
        .doFinally(signal -> System.out.println("doFinally 2 : " + signal))
.subscribe(Util.subscriber());
```

Mã trên thực hiện các hoạt động liên quan đến Flux trong WebFlux. Hãy giải thích từng phần trong mã:

1. Flux.create(fluxSink -> { ... }): Tạo một Flux bằng cách sử dụng phương thức tĩnh create và truyền vào một FluxSink. FluxSink được sử dụng để phát ra các phần tử và điều khiển luồng dữ liệu.
2. System.out.println("inside create"): In ra thông báo "inside create" để xác nhận rằng mã đang ở trong phương thức create của Flux.

3. Vòng lặp for từ 0 đến 4: Trong vòng lặp, chúng ta sử dụng fluxSink.next(i) để phát ra các giá trị từ 0 đến 4.
4. fluxSink.error(new RuntimeException("oops")): Gây ra một lỗi bằng cách sử dụng fluxSink.error với một RuntimeException. Điều này sẽ kích hoạt việc xử lý lỗi trong Flux.
5. System.out.println("--completed"): In ra thông báo "--completed" sau khi vòng lặp hoàn thành, bất kể có lỗi xảy ra hay không.
6. Các operator doOn... được sử dụng để gắn kết các hành động phụ vào các sự kiện trong quá trình xử lý Flux. Ví dụ:
    - doOnComplete(): In ra thông báo "doOnComplete" khi Flux hoàn thành.
    - doFirst(): In ra thông báo "doFirst" trước khi bất kỳ phần tử nào được xử lý.
    - doOnNext(): In ra thông báo "doOnNext" kèm theo giá trị của từng phần tử trong Flux.
    - doOnSubscribe(): In ra thông báo "doOnSubscribe" khi một Subscriber đăng ký với Flux.
    - doOnRequest(): In ra thông báo "doOnRequest" kèm theo số lượng yêu cầu phần tử trong quá trình xử lý.
    - doOnError(): In ra thông báo "doOnError" kèm theo thông báo lỗi khi có lỗi xảy ra trong quá trình xử lý.
    - doOnTerminate(): In ra thông báo "doOnTerminate" khi Flux kết thúc (hoàn thành hoặc có lỗi).
    - doOnCancel(): In ra thông báo "doOnCancel" khi Subscriber hủy đăng ký với Flux.
    - doFinally(): In ra thông báo "doFinally" sau khi Flux kết thúc, bao gồm cả hoàn thành, lỗi, hoặc hủy đăng ký.
7. doOnDiscard(Object.class, o -> System.out.println("doOnDiscard : " + o)): In ra thông báo "doOnDiscard" khi có phần tử bị loại bỏ trong quá trình xử lý. Trong trường hợp này, chúng ta không chỉ định loại đối tượng cụ thể.
8. take(2): Giới hạn số lượng phần tử chỉ lấy 2 phần tử đầu tiên từ Flux. Điều này tạo ra một Flux mới chứa chỉ 2 phần tử.
9. subscribe(Util.subscriber()): Đăng ký một Subscriber với Flux để tiêu thụ dữ liệu. Trong trường hợp này, chúng ta sử dụng một Subscriber được cung cấp bởi lớp tiện ích Util. Subscriber này sẽ in ra các phần tử nhận được từ Flux.

```
/*  Output
doFirst
doOnSubscribereactor.core.publisher.FluxPeekFuseable$PeekConditionalSubscriber@3f
fc5af1
doOnRequest : 9223372036854775807
inside create
doOnNext : 0
Received : 0
doOnNext : 1
Received : 1
doOnNext : 2
```

```
Received : 2
doOnNext : 3
Received : 3
doOnCancel
doFinally 1 : cancel
Completed
doFinally 2 : onComplete
doOnDiscard : 4
--completed
*/
```

3. Operator- Delay

Operator delay trong WebFlux được sử dụng để tạo một độ trễ giữa các phần tử trong một chuỗi Flux. Nó cho phép kiểm soát tốc độ phát ra của các phần tử và tạo ra một khoảng thời gian chờ giữa các sự kiện trong quá trình xử lý Flux.

Khi sử dụng operator delay, chúng ta có thể chỉ định thời gian chờ giữa các phần tử trong Flux bằng cách sử dụng một khoảng thời gian cụ thể hoặc một hàm để tính toán thời gian chờ.

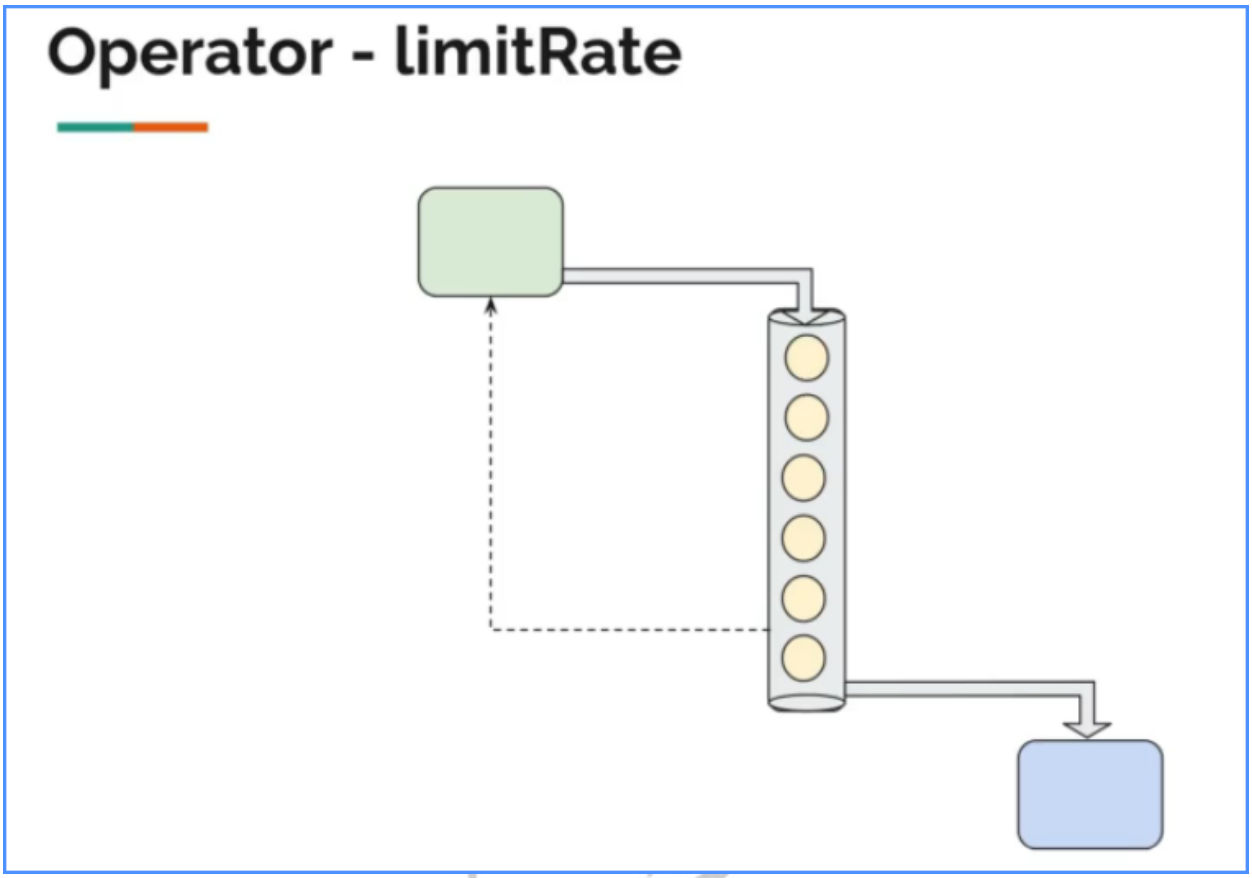
Dưới đây là một ví dụ minh họa về cách sử dụng operator delay trong WebFlux:

```
Flux.range(1, 5)
    .delayElements(Duration.ofSeconds(1))
    .subscribe(System.out::println);
```

Trong ví dụ trên, chúng ta tạo ra một Flux bằng cách sử dụng Flux.range để tạo ra một chuỗi các số từ 1 đến 5.

Sau đó, chúng ta sử dụng operator delayElements(Duration.ofSeconds(1)) để tạo một độ trễ 1 giây giữa các phần tử trong Flux. Điều này có nghĩa là mỗi phần tử sẽ được phát ra sau 1 giây kể từ phát ra phần tử trước đó.

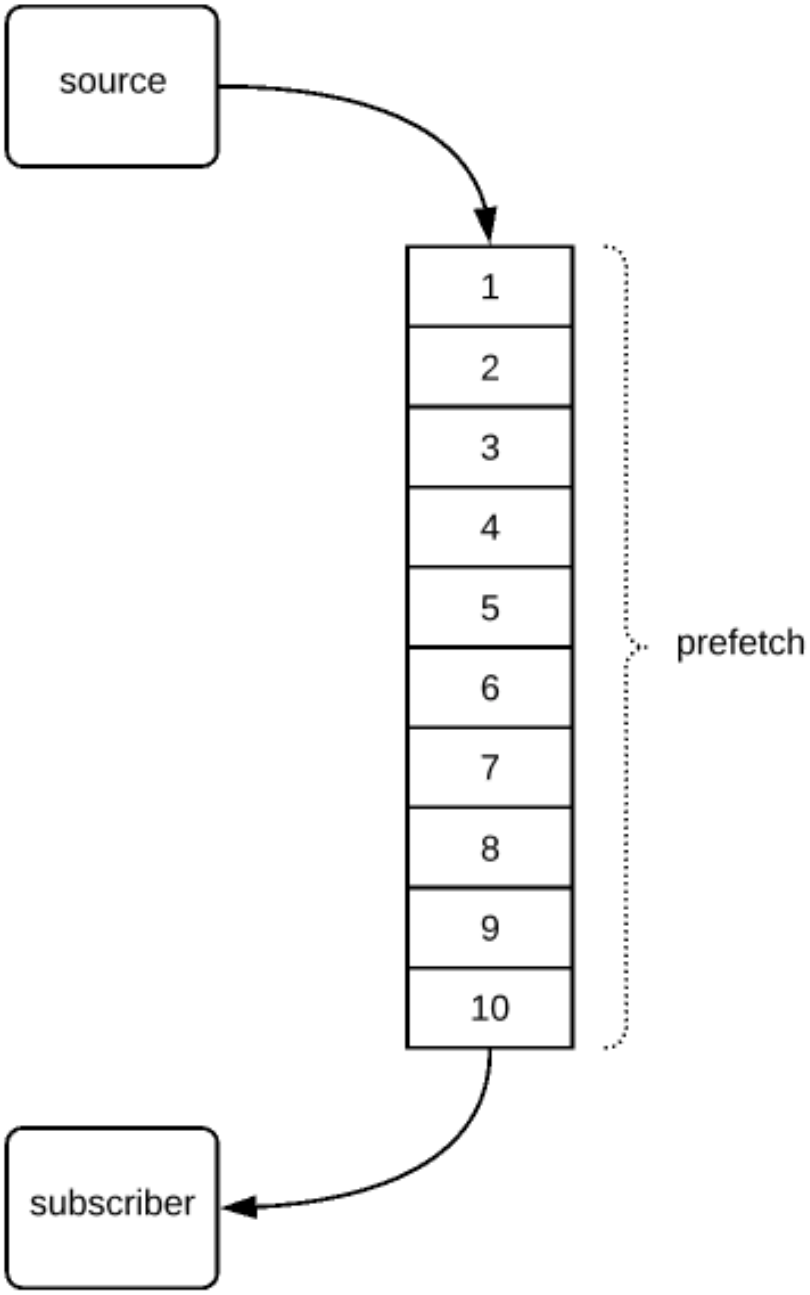
4. Operator- Limit Rate



a. Reactor limitRate

Trong lreactive programming, chúng ta có ***subscribers*** (downstream) ***publishers*** (upstream). Đôi khi, các downstream service có thể yêu cầu lượng dữ liệu vô hạn tới nguồn (upstream). Nó có thể ảnh hưởng đến hiệu suất của nguồn (giả sử nguồn là DB). Trong trường hợp này, sẽ tốt hơn nếu có một số toán tử ở giữa nguồn và downstream subscriber để hiểu nhu cầu từ downstream và tìm nạp trước dữ liệu dựa trên tốc độ xử lý downstream thay vì tìm nạp tất cả dữ liệu cùng một lúc.

Reactor limitRate có thể kiểm soát hành vi này bằng cách gửi yêu cầu nhu cầu cụ thể lên upstream. Trong trường hợp này, chỉ yêu cầu 10.




```
Flux.range(1, 100)
        .log()
        .limitRate(10)
        .delayElements(Duration.ofMillis(100))
        .subscribe(System.out::println);
```

Output

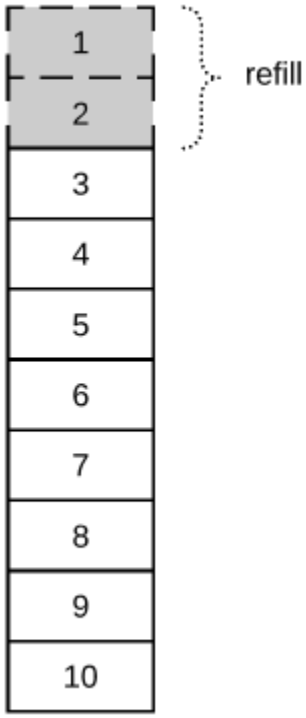
```
15:18:11.012 [main] INFO reactor.Flux.Range.1 - | onSubscribe([Synchronous
Fuseable] FluxRange.RangeSubscription)
15:18:11.017 [main] INFO reactor.Flux.Range.1 - | request(10)
15:18:11.021 [main] INFO reactor.Flux.Range.1 - | onNext(1)
...
15:18:11.079 [main] INFO reactor.Flux.Range.1 - | onNext(10)
1
2
3
4
5
6
7
15:18:11.778 [parallel-3] INFO reactor.Flux.Range.1 - | request(8)
15:18:11.778 [parallel-3] INFO reactor.Flux.Range.1 - | onNext(11)
...
...
...
...
15:18:19.005 [parallel-3] INFO reactor.Flux.Range.1 - | onNext(90)
80
...
87
15:18:19.808 [parallel-3] INFO reactor.Flux.Range.1 - | request(8)
15:18:19.808 [parallel-3] INFO reactor.Flux.Range.1 - | onNext(91)
...
15:18:19.808 [parallel-3] INFO reactor.Flux.Range.1 - | onNext(98)
88
...
95
15:18:20.610 [parallel-3] INFO reactor.Flux.Range.1 - | request(8)
15:18:20.611 [parallel-3] INFO reactor.Flux.Range.1 - | onNext(99)
15:18:20.611 [parallel-3] INFO reactor.Flux.Range.1 - | onNext(100)
15:18:20.612 [parallel-3] INFO reactor.Flux.Range.1 - | onComplete()
96
97
98
99
100
```

Sau khi 75% dữ liệu bị emit, nó sẽ tự động yêu cầu nạp lại số lượng. Đó là lý do tại sao thấy yêu cầu đầu tiên 10. Sau đó, thấy còn 8 lần sau đó.

b. High Tide / Low tide:

High Tide chỉ đơn giản là lượng tìm nạp trước ban đầu. Lượng nạp lại tiếp theo có thể được điều chỉnh bằng cách sử dụng **Low tide** như minh họa ở đây. **Low tide** sẽ thường xuyên yêu cầu nạp nước lên upstream.

```
Flux.range(1, 100)
        .log()
        .limitRate(10, 2)
        .delayElements(Duration.ofMillis(100))
        .subscribe(System.out::println);
```



Đầu ra:

```
17:48:20.077 [main] INFO reactor.Flux.Range.1 - | request(10)
17:48:20.079 [main] INFO reactor.Flux.Range.1 - | onNext(1)
...
17:48:20.124 [main] INFO reactor.Flux.Range.1 - | onNext(10)
1
17:48:20.223 [parallel-1] INFO reactor.Flux.Range.1 - | request(2)
17:48:20.225 [parallel-1] INFO reactor.Flux.Range.1 - | onNext(11)
17:48:20.225 [parallel-1] INFO reactor.Flux.Range.1 - | onNext(12)
2
```

```
3
17:48:20.426 [parallel-3] INFO reactor.Flux.Range.1 - | request(2)
17:48:20.426 [parallel-3] INFO reactor.Flux.Range.1 - | onNext(13)
17:48:20.426 [parallel-3] INFO reactor.Flux.Range.1 - | onNext(14)
4
5
17:48:20.627 [parallel-1] INFO reactor.Flux.Range.1 - | request(2)
...
...
...
98
99
17:48:30.065 [parallel-3] INFO reactor.Flux.Range.1 - | request(2)
100
```

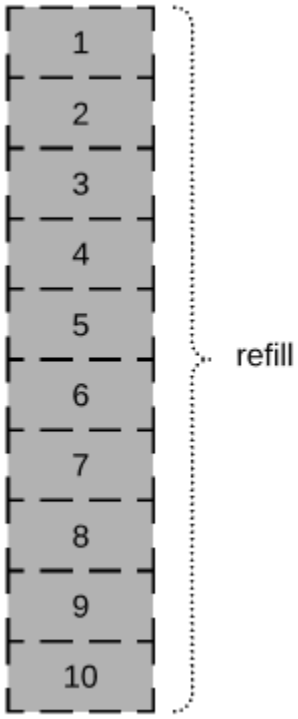
c. Low Tide = High Tide:

Khi **Low Tide = High Tide**, nó sẽ hoàn nguyên chiến lược bổ sung về mặc định là 75%.

d. Low tide = 0:

Khi Low tide = 0, nó sẽ vô hiệu hóa hoàn toàn yêu cầu sớm lên upstream. Thay vào đó, tất cả các dữ liệu sẽ cạn kiệt hoàn toàn trước khi yêu cầu nạp lại.

```
Flux.range(1, 100)
        .log()
        .limitRate(10, 0)
        .delayElements(Duration.ofMillis(100))
        .subscribe(System.out::println);
```



Output:

```
17:45:45.888 [main] INFO reactor.Flux.Range.1 - | request(10)
17:45:45.889 [main] INFO reactor.Flux.Range.1 - | onNext(1)
...
17:45:45.944 [main] INFO reactor.Flux.Range.1 - | onNext(10)
1
..
9
17:45:46.837 [parallel-1] INFO reactor.Flux.Range.1 - | request(10)
17:45:46.838 [parallel-1] INFO reactor.Flux.Range.1 - | onNext(11)
...
17:45:46.838 [parallel-1] INFO reactor.Flux.Range.1 - | onNext(20)
10
...
19
17:45:47.841 [parallel-3] INFO reactor.Flux.Range.1 - | request(10)
17:45:47.841 [parallel-3] INFO reactor.Flux.Range.1 - | onNext(21)
...
17:45:47.842 [parallel-3] INFO reactor.Flux.Range.1 - | onNext(29)
17:45:47.842 [parallel-3] INFO reactor.Flux.Range.1 - | onNext(30)
20
...
```

```
29
17:45:48.844 [parallel-1] INFO reactor.Flux.Range.1 - | request(10)
...
...
17:45:54.866 [parallel-1] INFO reactor.Flux.Range.1 - | onNext(99)
17:45:54.866 [parallel-1] INFO reactor.Flux.Range.1 - | onNext(100)
17:45:54.868 [parallel-1] INFO reactor.Flux.Range.1 - | onComplete()
90
...
99
17:45:55.871 [parallel-3] INFO reactor.Flux.Range.1 - | request(10)
100
```

5. Operator- onError

Operator onError trong WebFlux được sử dụng để xử lý các lỗi xảy ra trong quá trình xử lý Flux. Nó cho phép chúng ta đăng ký một hành động để được thực hiện khi một lỗi xảy ra trong Flux.

Khi sử dụng operator onError, chúng ta có thể xử lý lỗi bằng cách cung cấp một hàm consumer (hoặc một hàm lambda) nhận lỗi như tham số. Hành động này có thể là xử lý lỗi, ghi log, hoặc thực hiện bất kỳ hành động nào khác để đáp ứng với lỗi xảy ra.

Dưới đây là một ví dụ minh họa về cách sử dụng operator onError trong WebFlux:

```
public static void main(String[] args) {

    Flux.range(1, 10)
            .log()
            .map(i -> 10 / (5 - i))
            // .onErrorReturn(-1)
            //  .onErrorResume(e -> fallback())
            .onErrorContinue((err, obj) -> {

            })
            .subscribe(Util.subscriber());


}

private static Mono<Integer> fallback(){
    return Mono.fromSupplier(() -> Util.faker().random().nextInt(100, 200));
}
```

6. Operator- Timeout

Operator timeout trong WebFlux được sử dụng để đặt một thời gian giới hạn (timeout) cho việc xử lý một phần tử trong Flux. Nếu quá trình xử lý một phần tử mất quá nhiều thời gian so với giới hạn timeout đã định trước, nó sẽ tạo ra một lỗi timeout.

Khi sử dụng operator timeout, chúng ta có thể chỉ định một khoảng thời gian timeout cụ thể bằng cách sử dụng Duration hoặc sử dụng một hàm lambda để tính toán thời gian timeout dựa trên từng phần tử.

Dưới đây là một ví dụ minh họa về cách sử dụng operator timeout trong WebFlux:

```
Flux.range(1, 5)
    .delayElements(Duration.ofSeconds(2))
    .timeout(Duration.ofSeconds(1))
    .subscribe(System.out::println, error ->
        System.err.println("Timeout error: " + error.getMessage()));
```

Trong ví dụ trên, chúng ta tạo ra một Flux bằn

7. Operator- Default If Empty và Switch If Empty

Operator defaultIfEmpty trong WebFlux được sử dụng để cung cấp một giá trị mặc định trong trường hợp Flux không có phần tử nào. Nếu Flux không có phần tử, thì giá trị mặc định được cung cấp sẽ được phát ra thay vì Flux rỗng.

Dưới đây là một ví dụ minh họa về cách sử dụng operator defaultIfEmpty trong WebFlux:

```
Flux.empty()
    .defaultIfEmpty("No elements found")
    .subscribe(System.out::println);
```

Operator switchIfEmpty trong WebFlux được sử dụng để chuyển đổi sang một Flux khác trong trường hợp Flux ban đầu không có phần tử. Nếu Flux ban đầu không có phần tử, thì chúng ta có thể chuyển đổi sang một Flux khác để tiếp tục xử lý dữ liệu.

Dưới đây là một ví dụ minh họa về cách sử dụng operator switchIfEmpty trong WebFlux:

```
Flux.empty()
    .switchIfEmpty(Flux.just("No elements found, switching to backup Flux"))
    .subscribe(System.out::println);
```

```
================================================================================
===
public static void main(String[] args) {

        getOrderNumbers()
                .filter(i -> i > 10)
                .switchIfEmpty(fallback())
                .subscribe(Util.subscriber());

    }

    // redis cache / db
    private static Flux<Integer> getOrderNumbers(){
        return Flux.range(1, 10);
    }

    // db // cache
    private static Flux<Integer> fallback(){
        return Flux.range(20, 5);
    }
```

8. Operator- Transform

Operator transform trong WebFlux được sử dụng để biến đổi một Flux bằng cách áp dụng một Function (hàm) biến đổi cho nó. Điều này cho phép chúng ta thực hiện các biến đổi tùy chỉnh trên Flux và tạo ra một Flux mới dựa trên Flux ban đầu.

Dưới đây là một ví dụ minh họa về cách sử dụng operator transform trong WebFlux:

```
Flux.range(1, 5)
    .transform(flux -> flux.map(i -> i * 2))
    .subscribe(System.out::println);
```

Ví dụ khác:

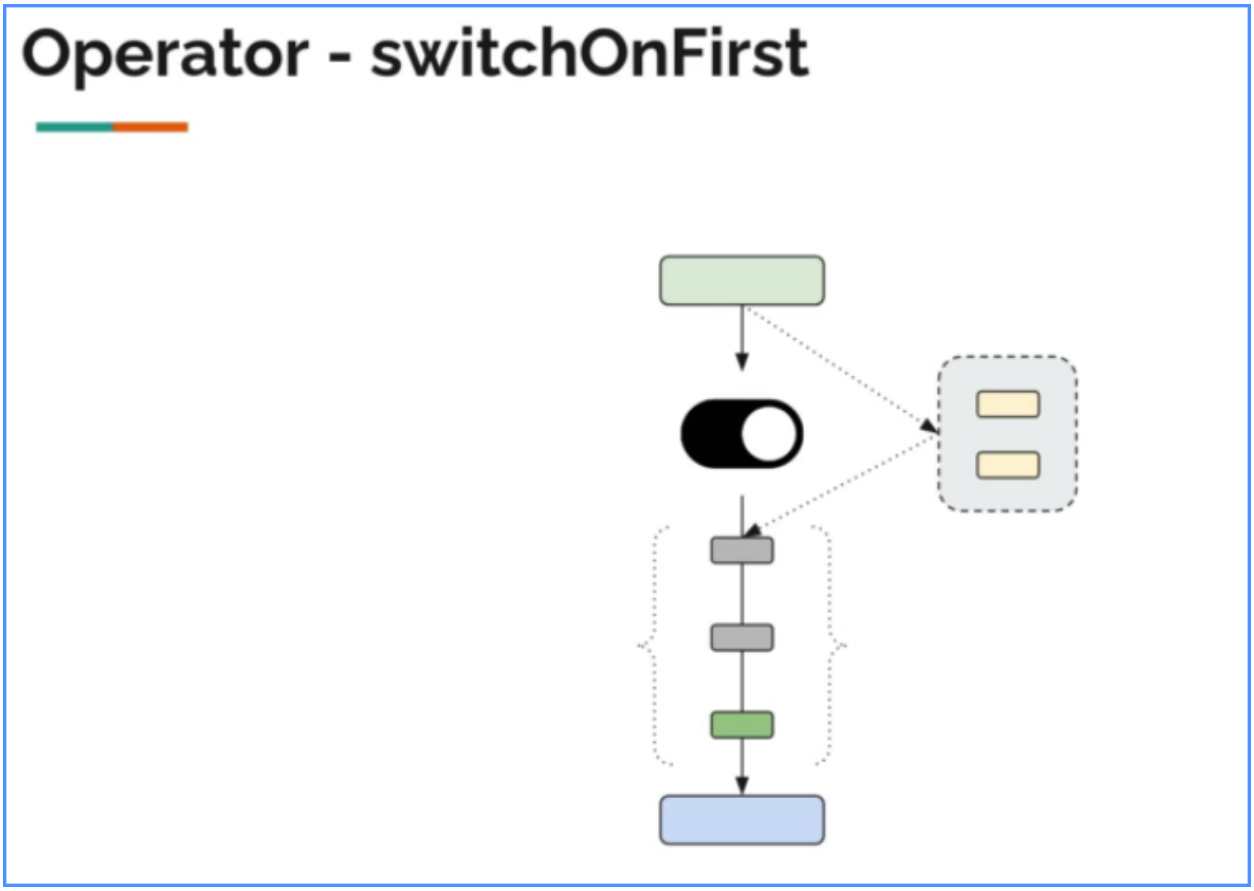
```
public static void main(String[] args) {

    getPerson()
            .transform(applyFilterMap())
            .subscribe(Util.subscriber());
}
```

```
public static Flux<Person> getPerson(){
return Flux.range(1, 10)
            .map(i -> new Person());
}

public static Function<Flux<Person>, Flux<Person>> applyFilterMap(){
return flux -> flux
                    .filter(p -> p.getAge() > 10)
                    .doOnNext(p -> p.setName(p.getName().toUpperCase()))
                    .doOnDiscard(Person.class, p -> System.out.println("Not
allowing : " + p));
}
```

Outputs:

```
/*
Received : Person(name=DAVID, age=23)
Received : Person(name=SHARAN, age=14)
Received : Person(name=ALFREDIA, age=29)
Not allowing : Person(name=Bobby, age=1)
Not allowing : Person(name=Jame, age=8)
Not allowing : Person(name=Paola, age=10)
Not allowing : Person(name=Kendall, age=8)
Not allowing : Person(name=Cordelia, age=1)
Received : Person(name=EMMA, age=27)
Not allowing : Person(name=Jeff, age=8)
Completed

 */
```

9. Operator- Switch On First



Operator switchOnFirst trong WebFlux được sử dụng để chuyển đổi từ một Flux đầu vào sang một Flux khác dựa trên sự kiện đầu tiên xảy ra trong Flux ban đầu. Nếu Flux ban đầu không có phần tử, chúng ta có thể chuyển đổi ngay lập tức sang một Flux khác. Nếu Flux ban đầu có một phần tử, chúng ta có thể áp dụng một biến đổi cho phần tử đó và tiếp tục với Flux biến đổi.

Dưới đây là một ví dụ minh họa về cách sử dụng operator switchOnFirst trong WebFlux:

```
Flux.just(1, 2, 3)
    .switchOnFirst((signal, flux) -> {
        if (signal.get() == 1) {
            return flux.map(i -> i * 2);
        } else {
            return Flux.just(0);
        }
    })
    .subscribe(System.out::println);
```

=> Kiểm tra phần tử đầu có bằng 1 hay không, nếu có thực hiện return flux.map(i -> i * 2);

Ví dụ:

```
public static void main(String[] args) {
    Flux.just(1, 2, 3)
            .switchOnFirst((signal, flux) -> {
                if (signal.get() == 2) {
                    return flux.map(i -> i * 2);
                } else {
                    return Flux.just(0);
                }
            })
            .subscribe(System.out::println);
}

public static Flux<Person> getPerson(){

    return Flux.range(1, 10)
            .map(i -> new Person());
}

public static Function<Flux<Person>, Flux<Person>> applyFilterMap(){
    System.out.println("get");
    return flux -> flux
            .filter(p -> p.getAge() > 10)
            .doOnNext(p -> p.setName(p.getName().toUpperCase()))
            .doOnDiscard(Person.class, p -> System.out.println("Not allowing : "
+ p));
}
```

10. Operator- Flat Map

Operator flatMap trong WebFlux được sử dụng để ánh xạ các phần tử trong một Flux sang một Flux khác và kết hợp chúng thành một Flux duy nhất. Điều này cho phép chúng ta thực hiện các hoạt động bất đồng bộ và kết hợp các kết quả thành một Flux duy nhất để tiếp tục xử lý.

Dưới đây là một ví dụ minh họa về cách sử dụng operator flatMap trong WebFlux:

```
Flux.just(1, 2, 3)
    .flatMap(i -> Flux.range(i, 2))
    .subscribe(System.out::println);
```

Output

```
1
2
2
3
3
4
```

Trong ví dụ trên, chúng ta có một Flux được tạo bằng cách sử dụng Flux.just với các phần tử 1, 2, 3.

Sau đó, chúng ta sử dụng operator flatMap để ánh xạ mỗi phần tử trong Flux ban đầu thành một Flux mới bằng cách sử dụng Flux.range(i, 2). Điều này tạo ra Flux mới với các phần tử từ i đến i + 1, trong đó i là giá trị của phần tử trong Flux ban đầu.

Cuối cùng, chúng ta đăng ký một bộ tiêu thụ với Flux và in ra các giá trị từ Flux ra màn hình. Kết quả in ra sẽ là 1, 2, 2, 3, 3, 4.

Trong ví dụ này, mỗi phần tử trong Flux ban đầu được ánh xạ thành một Flux mới bằng cách sử dụng flatMap. Sau đó, các phần tử trong các Flux con được kết hợp lại thành một Flux duy nhất để tiếp tục xử lý.

Operator flatMap cho phép chúng ta thực hiện các hoạt động bất đồng bộ trên mỗi phần tử trong Flux và kết hợp kết quả thành một Flux duy nhất. Điều này hữu ích để xử lý các tác vụ phức tạp và kết hợp kết quả từ nhiều Flux thành một Flux để tiếp tục xử lý.

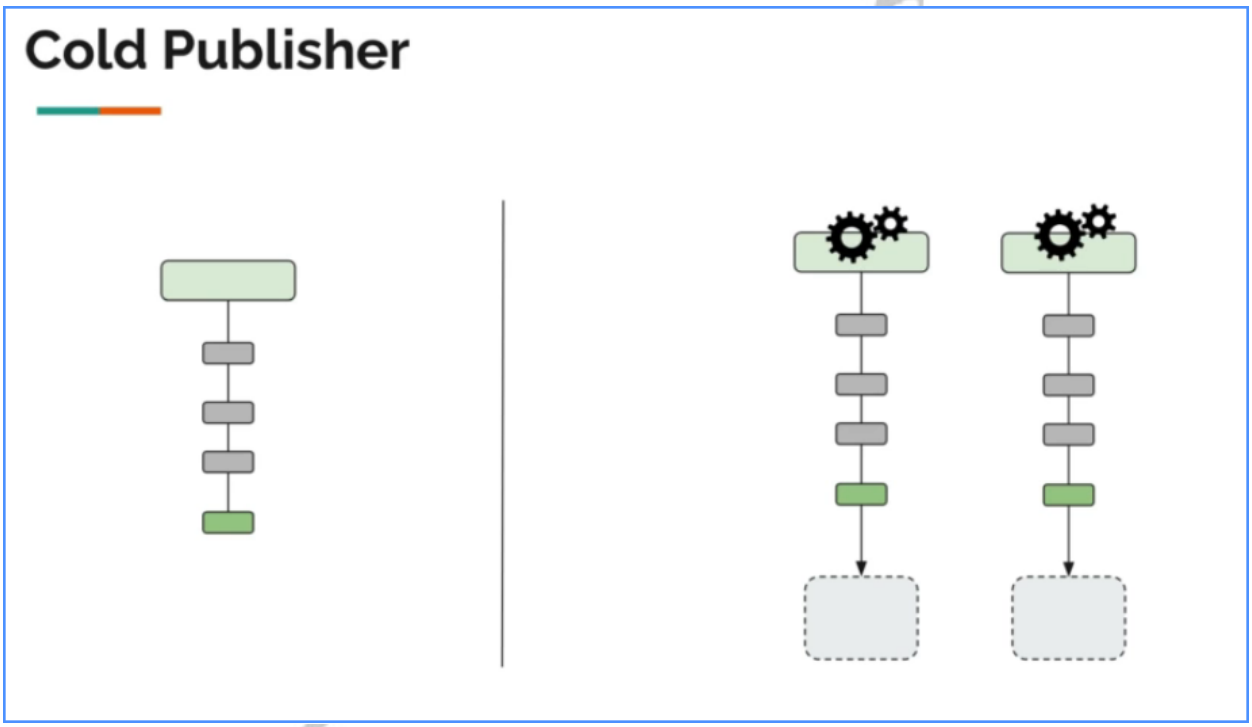
Chapter 7: Hot & Cold Publishers

1. Giới thiệu

a. Cold Publishers

Cold Publisher: Cold Publisher là một nguồn dữ liệu mà phát hành dữ liệu từ đầu mỗi khi có một Subscriber đăng ký. Mỗi Subscriber nhận được một luồng dữ liệu độc lập từ đầu. Khi một Subscriber mới đăng ký, quá trình sản xuất dữ liệu sẽ bắt đầu từ đầu, giống như một bộ phim được quay lại từ đầu mỗi khi có một khán giả mới.

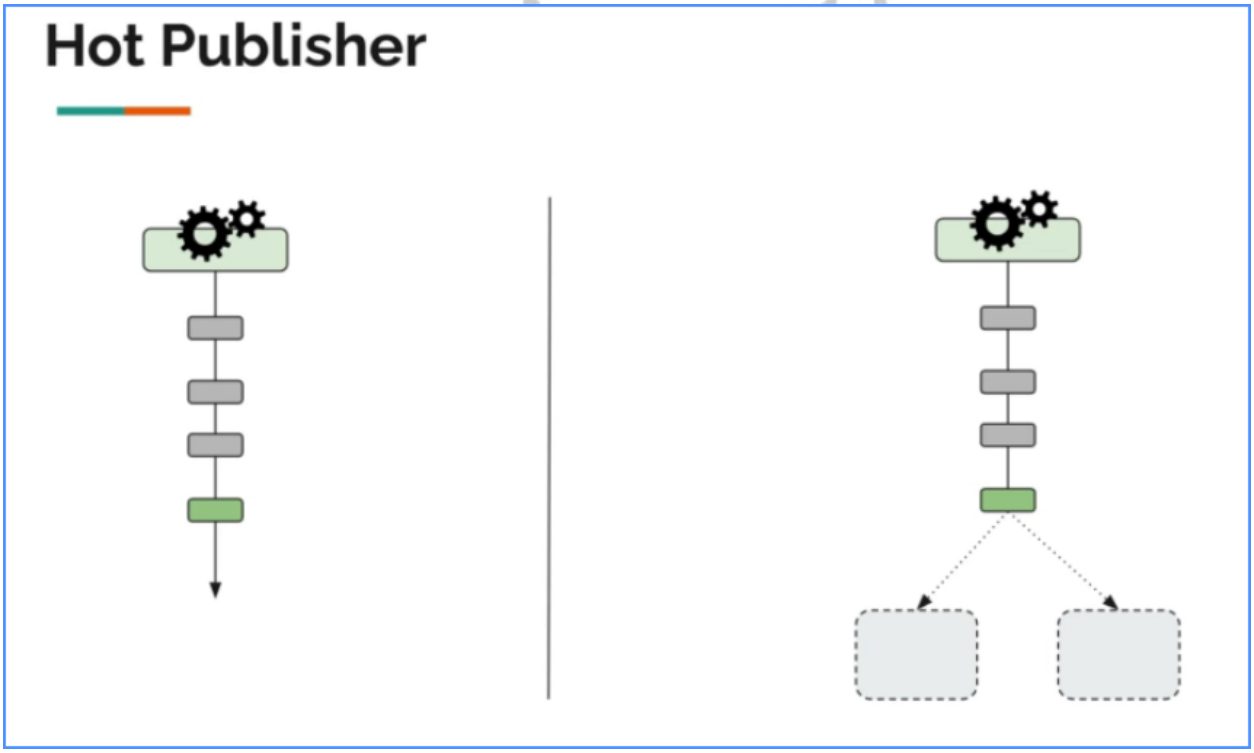
> Các Publisher theo mặc định không tạo ra bất kỳ giá trị nào trừ khi có ít nhất 1 observer đăng ký nó. Publisher tạo new data producer cho mỗi lần đăng ký mới (new subscription).



b. Host Publishers

Hot Publisher là một nguồn dữ liệu mà phát hành dữ liệu ngay cả khi không có Subscriber nào đăng ký. Dữ liệu được phát ra bất kể ai hoặc có bao nhiêu Subscriber đang lắng nghe. Khi một Subscriber mới đăng ký, nó chỉ nhận được dữ liệu từ thời điểm đăng ký trở đi. Hot Publisher tiếp tục phát dữ liệu cho tất cả các Subscriber hiện có mà không quan tâm đến việc có ai đang lắng nghe hay không.

> Hot Publishers không tạo nhà sản xuất dữ liệu mới (new data producer) cho mỗi đăng ký mới (như Cold Publisher làm). Thay vào đó sẽ chỉ có một data producer và tất cả những người quan sát sẽ lắng nghe dữ liệu do nhà sản xuất dữ liệu duy nhất tạo ra. Vì vậy, tất cả những người quan sát đều nhận được dữ liệu giống nhau.



2. Cold Publisher

Cold Publisher trong Reactive Programming được sử dụng để tạo và phát ra dữ liệu từ đầu mỗi khi có một Subscriber đăng ký. Nó thích hợp cho các tình huống khi bạn muốn mỗi Subscriber nhận được một luồng dữ liệu độc lập từ đầu, giống như việc quay lại từ đầu khi có một khán giả mới.

Dưới đây là một ví dụ về cách sử dụng Cold Publisher trong Java Reactive Programming bằng Project Reactor:

```
import reactor.core.publisher.Flux;

public class ColdPublisherExample {
    public static void main(String[] args) {
        // Tạo một Cold Publisher bằng Flux
        Flux<Integer> numbers = Flux.range(1, 5);

        // Đăng ký và tiêu thụ dữ liệu từ Cold Publisher
        numbers.subscribe(System.out::println); // Subscriber 1
        numbers.subscribe(System.out::println); // Subscriber 2
    }
}
```

Trong ví dụ trên, chúng ta tạo một Cold Publisher bằng cách sử dụng phương thức Flux.range(1, 5), tạo ra một luồng các số từ 1 đến 5. Khi mỗi Subscriber đăng ký, Cold Publisher sẽ phát ra dữ liệu từ đầu cho mỗi Subscriber.

Khi chúng ta chạy ví dụ này, kết quả sẽ là cả hai Subscriber nhận được cùng một luồng dữ liệu từ đầu:

```
/*
1
2
3
4
5
1
2
3
4
5

 */
```

Điều này cho thấy Cold Publisher bắt đầu phát dữ liệu từ đầu cho mỗi Subscriber khi họ đăng ký. Mỗi Subscriber nhận được một luồng dữ liệu độc lập từ đầu, giống như một bộ phim được quay lại từ đầu mỗi khi có một khán giả mới.

Ví dụ:

```
public class Lec01ColdPublisher {

    public static void main(String[] args) {
        Flux<String> movieStream = Flux.fromStream(() -> getMovie())
                .delayElements(Duration.ofSeconds(2));
        movieStream
                .subscribe(Util.subscriber("sam"));
        Util.sleepSeconds(5);
        movieStream
                .subscribe(Util.subscriber("mike"));
        Util.sleepSeconds(60);
    }

    // netflix
    private static Stream<String> getMovie(){
        System.out.println("Got the movie streaming req");
        return Stream.of(
                "Scene 1",
                "Scene 2",
                "Scene 3",
                "Scene 4",
                "Scene 5",
                "Scene 6",
                "Scene 7"
        );
    }
}
```

3. Hot Publisher- Share

a. Trường hợp 1:

Thêm phương thức ' ***share*** ' trong Flux để biến Netflux server thành Movie theater. Share biến Cold source thành Hot bằng cách truyền nhiều dữ liệu được phát tới nhiều subscriber

```
//our movie theatre
//each scene will play for 2 seconds
Flux<String> movieTheatre = Flux.fromStream(() -> getMovie())
                            .delayElements(Duration.ofSeconds(2)).share();

// you start watching the movie
movieTheatre.subscribe(scene -> System.out.println("You are watching " + scene));

//I join after sometime
Thread.sleep(5000);
movieTheatre.subscribe(scene -> System.out.println("Vinsguru is watching " +
scene));
```

Output:

```
Got the movie streaming request
You are watching scene 1
You are watching scene 2
You are watching scene 3
Vinsguru is watching scene 3
You are watching scene 4
Vinsguru is watching scene 4
You are watching scene 5
Vinsguru is watching scene 5
```

Từ đầu ra, đã bỏ lỡ 2 cảnh đầu tiên của bộ phim đó vì tham gia muộn. Tuy nhiên, có thể xem cảnh mới nhất được chiếu trong rạp cùng với những cảnh khác.

b. **Trường hợp 2**

Nhưng ở đây khi subscriber thứ hai tham gia, nguồn đã phát ra dữ liệu & hoàn thành. Vì vậy, subscription thứ hai lặp lại quá trình emit. Hãy tưởng tượng điều này giống như một ví dụ về rạp chiếu phim với buổi chiếu tiếp theo sau khi buổi chiếu đầu tiên hoàn thành.

```
//our movie theatre
//each scene will play for 2 seconds
Flux<String> movieTheatre = Flux.fromStream(() -> getMovie())
.delayElements(Duration.ofSeconds(2)).share();

// you start watching the movie
movieTheatre.subscribe(scene -> System.out.println("You are watching " + scene));

//I join after the source is completed
Thread.sleep(12000);
movieTheatre.subscribe(scene -> System.out.println("Vinsguru is watching " +
scene));
```

Output:

```
Got the movie streaming request
You are watching scene 1
You are watching scene 2
You are watching scene 3
You are watching scene 4
You are watching scene 5
Got the movie streaming request
Vinsguru is watching scene 1
Vinsguru is watching scene 2
Vinsguru is watching scene 3
Vinsguru is watching scene 4
Vinsguru is watching scene 5
```

4. Hot Publisher- Ref Count

Trong Reactive Streams và WebFlux, phương thức refCount() được sử dụng để quản lý số lượng đăng ký tới một Flux và tự động bắt đầu hoặc dừng việc phát ra sự kiện dựa trên số lượng đăng ký hiện tại.

Ví dụ: .refCount(1): Quản lý số lượng đăng ký tới Flux và tự động bắt đầu hoặc dừng việc phát ra sự kiện dựa trên số lượng đăng ký hiện tại.

```
public class Lec03HotPublish {

    public static void main(String[] args) {
        // share = publish().refCount(1)
        Flux<String> movieStream = Flux.fromStream(() -> getMovie())
                .delayElements(Duration.ofSeconds(1))
                .publish()
                .refCount(1);
        movieStream
                .subscribe(Util.subscriber("sam"));
        Util.sleepSeconds(10);
        movieStream
                .subscribe(Util.subscriber("mike"));
        Util.sleepSeconds(60);
    }

    // movie-theatre
    private static Stream<String> getMovie(){
        System.out.println("Got the movie streaming req");
        return Stream.of(
                "Scene 1",
                "Scene 2",
                "Scene 3",
                "Scene 4",
                "Scene 5",
                "Scene 6",
                "Scene 7"
        );
    }


}
```

Output:

```
/*
Got the movie streaming req
sam - Received : Scene 1
sam - Received : Scene 2
sam - Received : Scene 3
sam - Received : Scene 4
sam - Received : Scene 5
sam - Received : Scene 6
```

```
sam - Received : Scene 7
sam - Completed
Got the movie streaming req
mike - Received : Scene 1
mike - Received : Scene 2
mike - Received : Scene 3
mike - Received : Scene 4
mike - Received : Scene 5
mike - Received : Scene 6
mike - Received : Scene 7
mike - Completed

Process finished with exit code 0

 */
```

Ouput khi: **.refCount(2);**

```
/*
Got the movie streaming req
sam - Received : Scene 1
mike - Received : Scene 1
sam - Received : Scene 2
mike - Received : Scene 2
sam - Received : Scene 3
mike - Received : Scene 3
sam - Received : Scene 4
mike - Received : Scene 4
sam - Received : Scene 5
mike - Received : Scene 5
sam - Received : Scene 6
mike - Received : Scene 6
sam - Received : Scene 7
mike - Received : Scene 7
sam - Completed
mike - Completed

 */
```

5. Hot Publisher- Auto Connect

Trong Reactive Streams và WebFlux, phương thức autoConnect() được sử dụng để tự động kích hoạt việc phát ra sự kiện của một "Hot Publisher" khi một số lượng cụ thể của Subscriber đã đăng ký. Nó giúp đảm bảo rằng các sự kiện sẽ chỉ được phát ra khi đạt được điều kiện số lượng đăng ký mong muốn.

```
Flux<String> movieStream = Flux.fromStream(() -> getMovie())
.delayElements(Duration.ofSeconds(1))
.publish()
.autoConnect(2);
Util.sleepSeconds(3);
movieStream
.subscribe(Util.subscriber("sam"));
Util.sleepSeconds(10);
System.out.println("Mike is about to join");
movieStream
.subscribe(Util.subscriber("mike"));
Util.sleepSeconds(60);
```

=> Đợi đến khi có 2 Subscriber đăng ký mới emit dữ liệu, và sẽ đợi 13s

```
Mike is about to join
Got the movie streaming req
sam - Received : Scene 1
mike - Received : Scene 1
sam - Received : Scene 2
mike - Received : Scene 2
sam - Received : Scene 3
mike - Received : Scene 3
sam - Received : Scene 4
mike - Received : Scene 4
sam - Received : Scene 5
mike - Received : Scene 5
sam - Received : Scene 6
mike - Received : Scene 6
sam - Received : Scene 7
mike - Received : Scene 7
sam - Completed
mike - Completed
```

=> Nếu chuyển sang .autoConnect(1) và .autoConnect(0) Ouput sẽ thay đổi và subscriber: mike sẽ không được in ra

6. Hot Publisher- Cache

Trong Reactive Streams và WebFlux, phương thức cache() được sử dụng để tạo một "Hot Publisher" có khả năng lưu trữ (cache) các phần tử đã phát ra trước đó. Nó giúp đảm bảo rằng các Subscriber mới đăng ký sau khi có sự kiện phát ra vẫn nhận được các phần tử mà các Subscriber trước đó đã nhận.

```
Flux<String> movieStream = Flux.fromStream(() -> getMovie())
.delayElements(Duration.ofSeconds(1))
.cache(2);

Util.sleepSeconds(2);
movieStream
.subscribe(Util.subscriber("sam"));

Util.sleepSeconds(10);

System.out.println("Mike is about to join");
movieStream
.subscribe(Util.subscriber("mike"));
```

Trong ví dụ trên, chúng ta tạo một Flux bằng cách sử dụng getMovie(), một Flux phát ra các dữ liệu với độ trễ 1 giây giữa mỗi dữ liệu. Chúng ta áp dụng .cache(2) lên Flux để tạo một "Hot Publisher" có khả năng lưu trữ 2 phần tử đã phát ra.

```
/*
Got the movie streaming req
sam - Received : Scene 1
sam - Received : Scene 2
sam - Received : Scene 3
sam - Received : Scene 4
sam - Received : Scene 5
sam - Received : Scene 6
sam - Received : Scene 7
sam - Completed
Mike is about to join
mike - Received : Scene 6
mike - Received : Scene 7
mike - Completed
 */
```

=> Nếu không truyền giá trị sẽ lưu trữ toàn bộ phần tử

Chapter 8: Threading & Schedulers

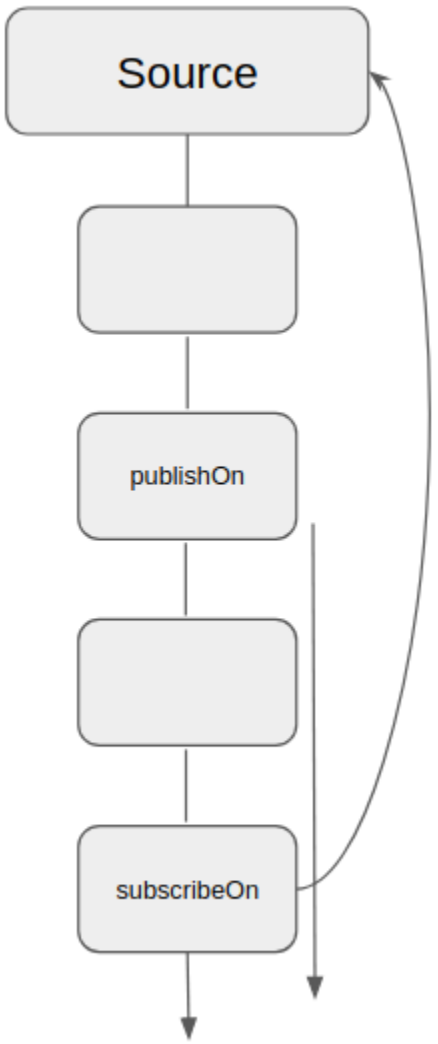
1. Reactor Schedulers:

Reactive Streams cung cấp tiêu chuẩn để xử lý luồng không đồng bộ. Để được hành vi asynchronous/non-blocking bằng cách lên lịch các tác vụ trên các worker threads (luồng công việc). Tự tạo và quản lý các task không phải là một nhiệm vụ dễ dàng. ***Project Reactor*** cung cấp dưới đây các phương pháp thuận tiện để sử dụng **workers thread** thông qua lớp **Schedulers**.

Scheduler Method	Usage
parallel	Đối với các tác vụ chuyên sâu về CPU (tính toán)
boundedElastic	Đối với các tác vụ chuyên sâu về IO (network calls)
immediate	Để giữ thực thi trong luồng hiện tại
single	Một luồng có thể tái sử dụng duy nhất cho tất cả người gọi

2. PublishOn vs SubscribeOn:

PublishOn & ***subscribeOn*** là các phương thức thuận tiện trong ***Project Reactor*** chấp nhận bất kỳ ***Schedulers*** *nào ở* trên để thay đổi bối cảnh thực thi tác vụ cho các hoạt động trong reactive pipeline. Trong khi ***subscribeOn*** buộc quá trình emit phải sử dụng Schedulers cụ thể, thì ***publishOn*** thay đổi Schedulers cho tất cả các hoạt động downstream trong quy trình như được hiển thị bên dưới.



3. Publisher

```
Flux<Integer> flux = Flux.range(0, 2)
  .map(i -> {
      System.out.println("Mapping for " + i + " is done by thread " +
Thread.currentThread().getName());
      return i;
  });
```

Map function chỉ trả về giá trị nhận được. Chỉ muốn thực hiện một số thao tác ghi nhật ký để xem tên luồng đang thực hiện map operation. Trong quá trình triển khai ***runnable***, chỉ cần subscribe cho flux trên! Xin lưu ý rằng trong reactive programming, sẽ không có gì xảy ra cho đến khi subscribe!

```
//create a runnable with flux subscription
Runnable r = () -> flux.subscribe(s -> {
    System.out.println("Received " + s + " via " +
Thread.currentThread().getName());
});
```

Tạo 2 luồng có tên t1 và t2 để thực thi runnable ở trên.

```
Thread t1 = new Thread(r, "t1");
Thread t2 = new Thread(r, "t2");

//lets start the threads. (this is when we are subscribing to the flux)
System.out.println("Program thread :: " + Thread.currentThread().getName());
t1.start();
t2.start();
```

Output:

```
/*
Program thread :: main
Mapping for 0 is done by thread t1
Mapping for 0 is done by thread t2
Received 0 via t1
Received 0 via t2
Mapping for 1 is done by thread t1
Mapping for 1 is done by thread t2
Received 1 via t1
Received 1 via t2
```

```
 */
```

Kết quả ở trên thì đối với mỗi subscription (cold subscription), Publisher sẽ publish các giá trị. Các hoạt động trên map đang được thực thi trong các thread tương ứng nơi diễn ra subscription. Đây là hành vi mặc định.

Ví dụ:

```
public static void main(String[] args) {

    Flux<Object> flux = Flux.create(fluxSink -> {
        printThreadName("create");
        fluxSink.next(1);
    })
            .doOnNext(i -> printThreadName("next " + i));
   Runnable runnable = () -> flux.subscribe(v -> printThreadName("sub " + v));

    for (int i = 0; i < 2; i++) {
        new Thread(runnable).start();
    }

    Util.sleepSeconds(5);

}

private static void printThreadName(String msg){
    System.out.println(msg + "\t\t: Thread : " +
Thread.currentThread().getName());
}
```

Output:

```
/*
create      : Thread : Thread-1
create      : Thread : Thread-0
next 1      : Thread : Thread-1
next 1      : Thread : Thread-0
sub 1       : Thread : Thread-0
sub 1       : Thread : Thread-1
 */
```

Tuy nhiên, Reactor cung cấp một cách dễ dàng để chuyển đổi việc thực thi task trong chuỗi reactive bằng các phương pháp bên dưới.

- publishOn
- subscribeOn

4. PublishOn:

publishOn chấp nhận **Scheduler** để thay cách thực thi task cho các hoạt động ở phía dưới. (đối với tất cả các hoạt động hoặc cho đến khi một ***publishOn*** khác chuyển ngữ cảnh trong chuỗi). Hãy xem các ví dụ dưới đây.

- **Scheulers.immediate():**

Để duy trì việc thực thi trong thread hiện tại.

```
Flux<Integer> flux = Flux.range(0, 2)
                      .publishOn(Schedulers.immediate())
                      .map(i -> {
                          System.out.println("Mapping for " + i + " is done by
thread " + Thread.currentThread().getName());
                          return i;
                      });
```

Đầu ra vẫn sẽ như cũ! Ở đây *Schedulers.immediate()* giữ việc thực thi trong thread hiện tại. Ở đây thread hiện tại không phải là chủ đề chính. **Thread** được gọi là subscribe method.

- **Schedulers.single():**

Một **thread** có thể tái sử dụng duy nhất. Khi chúng tôi sử dụng **thread** này, tất cả các hoạt động của chuỗi reactive sẽ được thực thi bằng **thread** này bởi tất cả người gọi.

```
Flux<Integer> flux = Flux.range(0, 2)
                          .publishOn(Schedulers.single())
                          .map(i -> {
                              System.out.println("Mapping for " + i + " is done
by thread " + Thread.currentThread().getName());
                              return i;
                          });
```

Output:

```
/*
Program thread :: main
Mapping for 0 is done by thread single-1
Received 0 via single-1
```

```
Mapping for 1 is done by thread single-1
Received 1 via single-1
Mapping for 0 is done by thread single-1
Received 0 via single-1
Mapping for 1 is done by thread single-1
Received 1 via single-1

 */
```

Kiểm tra đầu ra. Cả hai đăng ký được thực hiện bởi t1 và t2 đều được thực thi thông qua một **thread** duy nhất.

- **Schedulers.newSingle():**

Giống như trên. Nhưng một **thread** duy nhất dành riêng cho người gọi.

```
Flux<Integer> flux = Flux.range(0, 2)
                         .publishOn(Schedulers.newSingle("vinsguru"));
```

Output:

```
/*
Program thread :: main
Mapping for 0 is done by thread vinsguru-1
Received 0 via vinsguru-1
Mapping for 1 is done by thread vinsguru-1
Received 1 via vinsguru-1
Mapping for 0 is done by thread vinsguru-1
Received 0 via vinsguru-1
Mapping for 1 is done by thread vinsguru-1
Received 1 via vinsguru-1


 */
```

- **Schedulers.elastic():**

Đây là thread pool với các thread không giới hạn và không còn được ưa thích nữa. Vì vậy **KHÔNG SỬ DỤNG** tùy chọn này.

- **Schedulers.boundedElastic():**

Đây là một lựa chọn ưa thích thay vì elastic. Thread pool chứa 10 * number of CPU cores mà bạn có. Lựa chọn tốt cho hoạt động IO hoặc bất kỳ blocking call.

Schedulers.boundedElastic() được sử dụng để chỉ định rằng việc xử lý các phép tính reactive trong Flux sẽ được thực hiện trên một luồng có giới hạn (bounded) từ hồi quy (elastic) được cung cấp bởi lớp Schedulers trong Reactor.

Khi sử dụng Schedulers.boundedElastic(), bạn đang yêu cầu sử dụng một luồng hồi quy có giới hạn. Điều này có ý nghĩa là nó sẽ tạo ra một pool luồng được quản lý bởi hệ thống Reactor, giới hạn số luồng tối đa mà nó sử dụng. Việc giới hạn luồng giúp tránh tình trạng tạo quá nhiều luồng và tiêu thụ tài nguyên hệ thống quá mức.

```
Flux<Integer> flux = Flux.range(0, 2)
                         .publishOn(Schedulers.boundedElastic());
```

Output:

```
/*
Program thread :: main
Mapping for 0 is done by thread boundedElastic-1
Mapping for 0 is done by thread boundedElastic-2
Received 0 via boundedElastic-1
Mapping for 1 is done by thread boundedElastic-1
Received 1 via boundedElastic-1
Received 0 via boundedElastic-2
Mapping for 1 is done by thread boundedElastic-2
Received 1 via boundedElastic-2
 */
```

- **Schedulers.parallel():**

Một nhóm workers cố định được điều chỉnh để làm việc song song. Nó tạo ra nhiều worker như số lõi CPU của bạn. **Nên được sử dụng cho bất kỳ hoạt động CPU nào. Không dành cho IO hoặc blocking calls.**

```
Flux<Integer> flux = Flux.range(0, 2)
                            .publishOn(Schedulers.parallel());
```

Output:

```
/*
Program thread :: main
Mapping for 0 is done by thread parallel-1
Mapping for 0 is done by thread parallel-2
Received 0 via parallel-1
Mapping for 1 is done by thread parallel-1
```

```
Received 0 via parallel-2
Received 1 via parallel-1
Mapping for 1 is done by thread parallel-2
Received 1 via parallel-2
 */
```

Nếu bạn không muốn sử dụng parallel / boundedElastic pools, Schedulers có các phương pháp thuận tiện để tạo một nhóm luồng ***parallel*** và ***boundedElastic*** mới bằng các phương thức bên dưới:

- newParallel()
- newBoundedElastic()

5. Multiple PublishOn Methods:

```
Flux<Integer> flux = Flux.range(0, 2)
        .map(i -> {
            System.out.println("Mapping one for " + i + " is done by thread "
                + Thread.currentThread().getName());
            return i;
        })
        .publishOn(Schedulers.boundedElastic())
        .map(i -> {
            System.out.println("Mapping two for " + i + " is done by thread "
                + Thread.currentThread().getName());
            return i;
        })
        .publishOn(Schedulers.parallel())
        .map(i -> {
            System.out.println("Mapping three for " + i + " is done by thread "
                + Thread.currentThread().getName());
            return i;
        });
```

Output:

```
/*
Program thread :: main
Mapping one for 0 is done by thread t2
Mapping one for 0 is done by thread t1
Mapping one for 1 is done by thread t2
Mapping one for 1 is done by thread t1
```

```
Mapping two for 0 is done by thread boundedElastic-2
Mapping two for 0 is done by thread boundedElastic-1
Mapping two for 1 is done by thread boundedElastic-2
Mapping two for 1 is done by thread boundedElastic-1
Mapping three for 0 is done by thread parallel-1
Received 0 via parallel-1
Mapping three for 1 is done by thread parallel-1
Received 1 via parallel-1
Mapping three for 0 is done by thread parallel-2
Received 0 via parallel-2
Mapping three for 1 is done by thread parallel-2
Received 1 via parallel-2
 */
```

Kểm tra đầu ra ở trên.

- Ở đây map đầu tiên được thực thi trên subscription thread
- Map thứ hai được thực thi trên bounded Elastic
- Map thứ ba được thực thi trên parallel thread pool
- Vì không còn phương thức publish nào nữa nên ngay cả phương thức cuối cùng cũng được thực thi trên parallel thread pool

6. SubscribeOn:

SubscribeOn: Phương thức subscribeOn() được sử dụng để chỉ định luồng mà các phép tính reactive trong Flux được thực thi trên từ nguồn dữ liệu (Publisher) đến Subscriber cuối cùng. Nó ảnh hưởng đến toàn bộ chuỗi xử lý Flux. Khi sử dụng subscribeOn(), các phép tính reactive sẽ được thực thi trên luồng được chỉ định, và các luồng khác có thể tiếp tục công việc của chúng mà không bị chặn.

subscribeOn ảnh hưởng đến bối cảnh của source. Nghĩa là, không có gì xảy ra trong chuỗi phản ứng cho đến khi subscribe! Sau khi subscribe, pipeline sẽ được thực thi theo mặc định trên thread đã subscribe. Khi gặp phương thức PublishOn, nó sẽ chuyển ngữ cảnh cho các hoạt động downstream. Nhưng nguồn là Flux/Mono/hoặc bất kỳ publisher nào, luôn được thực thi trên thread hiện tại đã subscribe. ***Phương thức subscribeOn*** này sẽ thay đổi hành vi.

```
Runnable r = () -> flux
                        .subscribeOn(Schedulers.single())
                        .subscribe(s -> {
    System.out.println("Received " + s + " via " +
Thread.currentThread().getName());
});
```

Output:

```
/*
Program thread :: main
Mapping one for 0 is done by thread single-1
Mapping one for 1 is done by thread single-1
Mapping two for 0 is done by thread boundedElastic-1
Mapping two for 1 is done by thread boundedElastic-1
Mapping one for 0 is done by thread single-1
Mapping three for 0 is done by thread parallel-1
Received 0 via parallel-1
Mapping three for 1 is done by thread parallel-1
Received 1 via parallel-1
Mapping one for 1 is done by thread single-1
Mapping two for 0 is done by thread boundedElastic-2
Mapping two for 1 is done by thread boundedElastic-2
Mapping three for 0 is done by thread parallel-2
Received 0 via parallel-2
Mapping three for 1 is done by thread parallel-2
Received 1 via parallel-2

 */
```

Từ output,có thể thấy rõ rằng có thể kiểm soát thread pool nào sẽ được sử dụng cho nguồn . Xin lưu ý rằng có thể có nhiều phương thức ***publishOn*** sẽ tiếp tục chuyển đổi ngữ cảnh. Tuy nhiên phương thức ***subscribeOn*** không thể làm được điều đó. ***Chỉ phương thức subscribeOn*** đầu tiên gần nguồn mới được ưu tiên.

7. So sánh **publishOn** và **subscribeOn**

Trong WebFlux, publishOn và subscribeOn là hai phương thức được sử dụng để thay đổi luồng (thread) mà các phép biến đổi và xử lý trong luồng Reactive sẽ chạy trên. Dưới đây là một số điểm khác nhau giữa publishOn và subscribeOn:

1. **Vị trí ảnh hưởng**:
   - publishOn: Ảnh hưởng đến các phép biến đổi và xử lý sau phép publishOn trong chuỗi xử lý Reactive.
   - subscribeOn: Ảnh hưởng đến toàn bộ chuỗi xử lý Reactive từ phép biến đổi đầu tiên trở đi.
2. **Phạm vi ảnh hưởng**:
   - publishOn: Thay đổi luồng mà các phần tử trong luồng Reactive được xử lý trên.
   - subscribeOn: Thay đổi luồng mà toàn bộ chuỗi xử lý Reactive chạy trên.
3. **Thời điểm áp dụng**:

- publishOn: Áp dụng sau phép publishOn và trước phép biến đổi tiếp theo trong chuỗi xử lý Reactive.
- subscribeOn: Áp dụng cho toàn bộ chuỗi xử lý Reactive, từ phép biến đổi đầu tiên trở đi.

4. **Ưu điểm sử dụng**:
   - publishOn: Thường được sử dụng để thay đổi luồng xử lý cho các phép biến đổi và xử lý chậm, nhưng vẫn giữ nguyên luồng gốc cho các phần tử nhanh.
   - subscribeOn: Thường được sử dụng để thay đổi luồng xử lý cho toàn bộ chuỗi xử lý Reactive.

Ví dụ:

```
Flux.range(1, 10)
    .map(i -> {
        System.out.println("Mapping element " + i + " on thread " +
Thread.currentThread().getName());
        return i * 2;
    })
    .publishOn(Schedulers.parallel()) // Thay đổi luồng xử lý từ đây
    .map(i -> {
        System.out.println("Processing element " + i + " on thread " +
Thread.currentThread().getName());
        return i + 1;
    })
    .subscribeOn(Schedulers.single()) // Thay đổi luồng xử lý từ đây
    .subscribe(System.out::println);
```

Trong ví dụ trên:

- publishOn(Schedulers.parallel()) sẽ thay đổi luồng xử lý từ phép biến đổi publishOn trở đi, các phần tử sẽ được xử lý trên luồng song song (parallel).
- subscribeOn(Schedulers.single()) sẽ thay đổi luồng xử lý cho toàn bộ chuỗi xử lý Reactive, bao gồm cả phép biến đổi đầu tiên, và chúng sẽ chạy trên luồng đơn (single).

Tóm lại, publishOn và subscribeOn đều có vai trò quan trọng trong việc quản lý luồng trong WebFlux. Sự khác nhau chính giữa hai phương thức này là vị trí và phạm vi ảnh hưởng đến chuỗi xử lý Reactive.

8. Parallel Execution

Parallel Execution trong WebFlux được sử dụng để thực hiện các phép biến đổi và xử lý trên nhiều luồng song song, nhằm tăng hiệu suất và tận dụng tối đa khả năng xử lý đa luồng của hệ thống.

Việc sử dụng Parallel Execution có thể giúp tăng tốc độ xử lý cho các tác vụ tốn nhiều thời gian, giảm thời gian chờ đợi và tăng khả năng chịu tải của ứng dụng WebFlux.

```
Flux.range(1, 10)
    .parallel()
    .runOn(Schedulers.parallel())
    .map(i -> {
        System.out.println("Processing element " + i + " on thread " +
Thread.currentThread().getName());
        return i * 2;
    })
    .sequential()
    .subscribe(System.out::println);
```

Output:

```
/*
Processing element 7 on thread parallel-7
Processing element 10 on thread parallel-10
14
Processing element 3 on thread parallel-3
Processing element 1 on thread parallel-1
Processing element 8 on thread parallel-8
Processing element 6 on thread parallel-6
Processing element 2 on thread parallel-2
20
Processing element 9 on thread parallel-9
Processing element 5 on thread parallel-5
Processing element 4 on thread parallel-4
2
4
6
8
10
12
16
18
 */
```

Trong ví dụ trên, parallel() được sử dụng để chuyển đổi từ một Flux tuần tự thành một Flux song song. Sau đó, runOn(Schedulers.parallel()) được sử dụng để chỉ định luồng xử lý cho các phần tử trong Flux song song, trong trường hợp này là luồng song song. Tiếp theo, map() được sử dụng để thực hiện phép biến đổi trên các phần tử của Flux song song. Cuối cùng, sequential() được sử dụng để chuyển đổi từ Flux song song thành Flux tuần tự trước khi subscribe.

Kết quả là các phần tử trong Flux sẽ được xử lý trên các luồng song song, do đó, các phép biến đổi và xử lý có thể được thực hiện đồng thời trên nhiều luồng. Điều này giúp tăng tốc độ xử lý và cải thiện hiệu suất của ứng dụng.

Tuy nhiên, khi sử dụng Parallel Execution, cần lưu ý rằng việc xử lý đồng thời có thể tạo ra các kết quả không đúng thứ tự. Do đó, chỉ nên sử dụng Parallel Execution khi thứ tự không phải là yếu tố quan trọng trong quá trình xử lý.

Tóm lại, Parallel Execution trong WebFlux được sử dụng để thực hiện các phép biến đổi và xử lý trên nhiều luồng song song, nhằm tăng hiệu suất và tận dụng khả năng xử lý đa luồng.

Ví dụ khác:

```
Flux.range(1, 10)
.parallel(10)
.runOn(Schedulers.boundedElastic())
.doOnNext(i -> printThreadName("next " + i))
.sequential()
.subscribe(v -> printThreadName("sub " + v));
Util.sleepSeconds(5);
```

// Output

```
/*
next 8      : Thread : boundedElastic-8
next 1      : Thread : boundedElastic-1
next 5      : Thread : boundedElastic-5
next 6      : Thread : boundedElastic-6
next 7      : Thread : boundedElastic-7
next 2      : Thread : boundedElastic-2
next 4      : Thread : boundedElastic-4
sub 8       : Thread : boundedElastic-8
next 3      : Thread : boundedElastic-3
next 9      : Thread : boundedElastic-9
sub 1       : Thread : boundedElastic-8
next 10     : Thread : boundedElastic-10
sub 2       : Thread : boundedElastic-8
```

```
sub 3       : Thread : boundedElastic-8
sub 4       : Thread : boundedElastic-8
sub 5       : Thread : boundedElastic-8
sub 6       : Thread : boundedElastic-8
sub 7       : Thread : boundedElastic-8
sub 9       : Thread : boundedElastic-8
sub 10      : Thread : boundedElastic-8
 */
```

=> Nếu không có sequential thì sẽ không là luồng tuần tự và thứ tự next và sub khác nhau

9. Note On Flux Interval

Flux.interval() trong WebFlux được sử dụng để tạo ra một Flux phát ra các giá trị liên tục sau một khoảng thời gian nhất định. Nó thường được sử dụng để tạo ra các sự kiện định kỳ trong ứng dụng.

Ví dụ:

```
Flux.interval(Duration.ofSeconds(1))
                .log()
                .subscribe(Util.subscriber());
    Util.sleepSeconds(60);
```

Output:

```
/*
[ INFO] (main) onSubscribe(FluxConcatMap.ConcatMapImmediate)
[ INFO] (main) request(unbounded)
[ INFO] (parallel-1) onNext(1)
Received : 1
[ INFO] (parallel-2) onNext(2)
Received : 2
[ INFO] (parallel-3) onNext(3)
Received : 3
[ INFO] (parallel-4) onNext(4)
Received : 4

....
 */
```

Trong ví dụ trên, Flux.interval(Duration.ofSeconds(1)) sẽ tạo ra một Flux phát ra các giá trị liên tục sau mỗi giây. Phép biến đổi take(5) được sử dụng để giới hạn Flux chỉ phát ra 5 giá trị. Cuối cùng, subscribe(System.out::println) được sử dụng để đăng ký một người nghe (subscriber) để nhận các giá trị từ Flux và in chúng ra màn hình.

delayElements là một phép biến đổi trong WebFlux được sử dụng để trì hoãn việc phát ra các phần tử trong một Flux. Nó giúp tạo ra một khoảng thời gian chờ giữa các phần tử khi chúng được phát ra tới người nghe (subscriber).

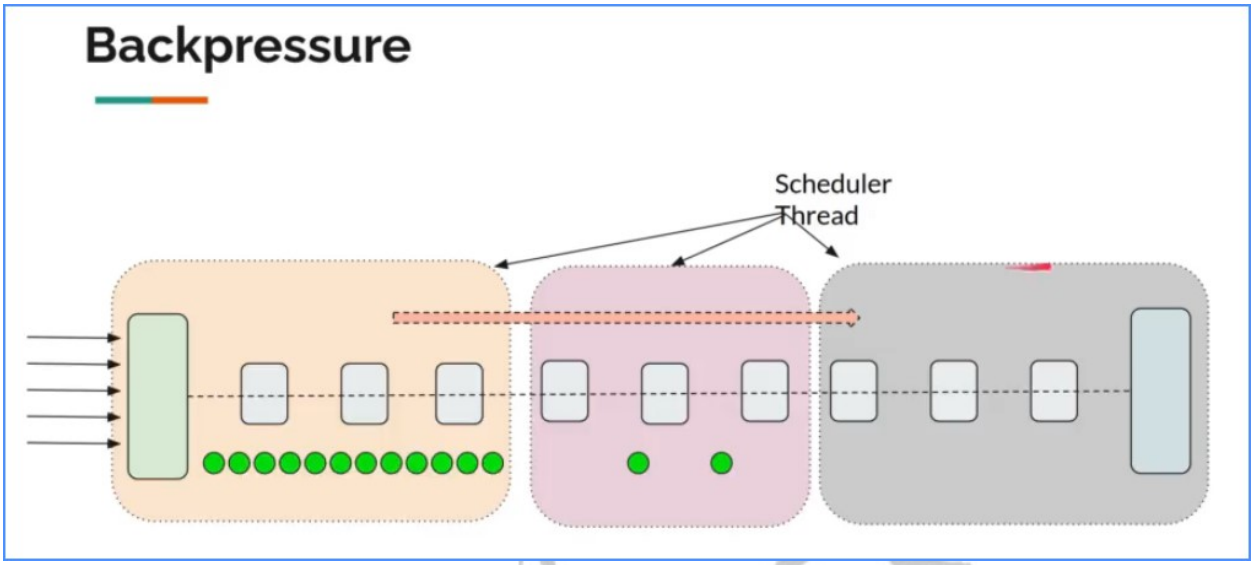
Ví dụ:

```
Flux.range(1, 10)
                .delayElements(Duration.ofSeconds(1))
                .log()
                .subscribe(Util.subscriber());
        Util.sleepSeconds(60);
```

Output:

```
/*
[ INFO] (main) onSubscribe(FluxConcatMap.ConcatMapImmediate)
[ INFO] (main) request(unbounded)
[ INFO] (parallel-1) onNext(1)
Received : 1
[ INFO] (parallel-2) onNext(2)
Received : 2
[ INFO] (parallel-3) onNext(3)
Received : 3
[ INFO] (parallel-4) onNext(4)
Received : 4
[ INFO] (parallel-5) onNext(5)

....

 */
```

Chapter 9: Backpressure / Overflow Strategy



Backpressure là một khái niệm quan trọng trong lập trình dựa trên luồng (stream-based programming) như Reactive Streams, bao gồm cả WebFlux. Nó liên quan đến cách quản lý tốc độ của nguồn dữ liệu (publisher) và khả năng tiếp nhận dữ liệu của người tiêu thụ (subscriber) trong một luồng dữ liệu.

Khi nguồn dữ liệu (publisher) phát ra dữ liệu nhanh hơn những gì người tiêu thụ (subscriber) có thể xử lý, sự mất cân đối này có thể gây ra những vấn đề như quá tải bộ nhớ, giảm hiệu suất, hoặc thậm chí là crash ứng dụng. Để giải quyết vấn đề này, Reactive Streams giới thiệu khái niệm backpressure.

Backpressure cho phép người tiêu thụ (subscriber) thông báo cho nguồn dữ liệu (publisher) về khả năng xử lý hiện tại của mình và yêu cầu nguồn dữ liệu giảm tốc độ phát ra dữ liệu nếu cần. Điều này đảm bảo rằng người tiêu thụ sẽ không bị quá tải và có thể xử lý dữ liệu một cách hiệu quả.

Overflow Strategy là một phần của hệ thống backpressure trong Reactive Streams, nó xác định cách xử lý khi có sự mất cân đối giữa nguồn dữ liệu (publisher) và người tiêu thụ (subscriber). Có một số Overflow Strategy phổ biến được sử dụng:

1. **BUFFER**: Dữ liệu sẽ được lưu trữ trong bộ đệm (buffer) khi người tiêu thụ không thể xử lý nhanh chóng. Điều này cho phép nguồn dữ liệu tiếp tục phát ra dữ liệu mà không bị chặn. Tuy nhiên, nếu bộ đệm tràn, nó có thể dẫn đến vấn đề về tài nguyên và hiệu suất. => Mặc định của hệ thống
2. **DROP**: Dữ liệu mới sẽ bị bỏ qua khi người tiêu thụ không thể xử lý nhanh chóng. Điều này có thể dẫn đến mất mát dữ liệu, nhưng nguồn dữ liệu vẫn tiếp tục hoạt động mà không bị chặn.

3. **LATEST**: Chỉ dữ liệu mới nhất sẽ được giữ lại trong bộ đệm, và dữ liệu cũ hơn sẽ bị loại bỏ. Điều này đảm bảo rằng người tiêu thụ chỉ nhận được dữ liệu mới nhất và không bị chặn.
4. **ERROR**: Nếu người tiêu thụ không thể xử lý nhanh chóng, dữ liệu mới sẽ không được chấp nhận và một lỗi (exception) sẽ được phát ra để thông báo về sự mất cân đối.

Strategy	Behavior
buffer	keep in memory
drop	Once the queue is full, new items will be dropped
latest	Once the queue is full, keep 1 latest item as and when it arrives. drop old
error	throw error to the downstream

1. Overflow Strategy- Drop

Overflow Strategy - Drop là một chiến lược xử lý tràn trong hệ thống backpressure của Reactive Streams, trong đó dữ liệu mới sẽ bị bỏ qua khi người tiêu thụ không thể xử lý nhanh chóng. Người tiêu thụ chỉ nhận được dữ liệu nhanh nhất và không lưu trữ dữ liệu không thể xử lý.

Ví dụ:

```
Flux.range(1, 10)
    .onBackpressureDrop()
    .subscribe(System.out::println);
```

Trong ví dụ trên, Flux.range(1, 10) tạo ra một Flux phát ra các số từ 1 đến 10. Phép biến đổi onBackpressureDrop() được sử dụng để áp dụng chiến lược Drop. Khi người tiêu thụ không thể xử lý nhanh chóng, dữ liệu mới sẽ bị bỏ qua và không được lưu trữ.

```
public static void main(String[] args) {
    // 75% 12
    System.setProperty("reactor.bufferSize.small", "16");

    List<Object> list = new ArrayList<>();

    Flux.create(fluxSink -> {
        for (int i = 1; i < 201; i++) {
            fluxSink.next(i);
            System.out.println("Pushed : " + i);
            Util.sleepMillis(1);
        }
        fluxSink.complete();
    })
            .onBackpressureDrop(list::add)
            .publishOn(Schedulers.boundedElastic())
            .doOnNext(i -> {
                Util.sleepMillis(10);
            })
            .subscribe(Util.subscriber());
    Util.sleepSeconds(10);
    System.out.println(list);

}
```

Output: (số lượng bản ghi bị loại bỏ)

```
/*
[17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29,
  30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42,
  43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55,
  56, 57, 58, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80,
  81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93,
  94, 95, 96, 97, 98, 99, 100, 101, 102, 103, 104, 105,
  106, 107, 108, 109, 110, 111, 112, 113, 114, 115, 116,
  117, 118, 119, 120, 121, 122, 123, 136, 137, 138, 139,
  140, 141, 142, 143, 144, 145, 146, 147, 148, 149, 150,
  151, 152, 153, 154, 155, 156, 157, 158, 159, 160, 161,
  162, 163, 164, 165, 166, 167, 168, 169, 170, 171, 172,
  173, 174, 175, 176, 177, 178, 179, 180, 193, 194, 195,
  196, 197, 198, 199, 200
]
 */
```

2. Overflow Strategy- Latest

Overflow Strategy - Latest là một chiến lược xử lý tràn trong hệ thống backpressure của Reactive Streams, trong đó người tiêu thụ chỉ nhận được dữ liệu mới nhất khi không thể xử lý nhanh chóng. Các dữ liệu cũ hơn sẽ bị ghi đè bởi dữ liệu mới nhất và không được lưu trữ.

Ví dụ:

```
Flux.range(1, 10)
    .onBackpressureLatest()
    .subscribe(System.out::println);
```

Trong ví dụ trên, Flux.range(1, 10) tạo ra một Flux phát ra các số từ 1 đến 10. Phép biến đổi onBackpressureLatest() được sử dụng để áp dụng chiến lược Latest. Khi người tiêu thụ không thể xử lý nhanh chóng, dữ liệu mới nhất sẽ được giữ lại và các dữ liệu cũ hơn sẽ bị ghi đè.

Ví dụ:

```
public static void main(String[] args) {
    // 75% 12
    System.setProperty("reactor.bufferSize.small", "16");

    Flux.create(fluxSink -> {
        for (int i = 1; i < 201; i++) {
            fluxSink.next(i);
            System.out.println("Pushed : " + i);
            Util.sleepMillis(1);
        }
        fluxSink.complete();
    })
            .onBackpressureLatest()
            .publishOn(Schedulers.boundedElastic())
            .doOnNext(i -> {
                Util.sleepMillis(10);
            })
            .subscribe(Util.subscriber());


    Util.sleepSeconds(10);


}
```

3. Overflow Strategy- Error

Overflow Strategy - Error là một chiến lược xử lý tràn trong hệ thống backpressure của Reactive Streams, trong đó một lỗi (Exception) sẽ được ném ra khi người tiêu thụ không thể xử lý nhanh chóng. Dữ liệu sẽ không được lưu trữ và quá trình phát ra dữ liệu sẽ bị dừng lại.

```
Flux.range(1, 10)
    .onBackpressureError()
    .subscribe(System.out::println);
```

Trong ví dụ trên, Flux.range(1, 10) tạo ra một Flux phát ra các số từ 1 đến 10. Phép biến đổi onBackpressureError() được sử dụng để áp dụng chiến lược Error. Khi người tiêu thụ không thể xử lý nhanh chóng, một lỗi (BackpressureException) sẽ được ném ra và quá trình phát ra dữ liệu sẽ bị dừng lại.

Khi chạy ví dụ trên, nếu người tiêu thụ không thể xử lý đủ nhanh để theo kịp tốc độ phát ra của Flux, một lỗi reactor.core.Exceptions$OverflowException: Could not emit buffer due to lack of requests sẽ được ném ra.

Overflow Strategy - Error là một lựa chọn khi bạn muốn biết và xử lý lỗi khi người tiêu thụ không thể xử lý nhanh chóng. Điều này có thể giúp bạn phát hiện và giải quyết các vấn đề hiệu suất hoặc lỗi xảy ra trong hệ thống.

```
public static void main(String[] args) {
    // 75% 12
    System.setProperty("reactor.bufferSize.small", "16");
    Flux.create(fluxSink -> {
        for (int i = 1; i < 201 && !fluxSink.isCancelled(); i++) {
            fluxSink.next(i);
            System.out.println("Pushed : " + i);
            Util.sleepMillis(1);
        }
        fluxSink.complete();
    })
            .onBackpressureError()
            .publishOn(Schedulers.boundedElastic())

            .doOnNext(i -> {
                Util.sleepMillis(10);
            })
            .subscribe(Util.subscriber());

    Util.sleepSeconds(10);
```

```
}
```

4. Overflow Strategy- Buffer

Overflow Strategy - Buffer With Size là một chiến lược xử lý tràn trong hệ thống backpressure của Reactive Streams, trong đó dữ liệu được lưu trữ trong một bộ đệm có kích thước cố định khi người tiêu thụ không thể xử lý nhanh chóng. Khi bộ đệm đạt đến kích thước tối đa, dữ liệu mới sẽ bị chặn và nguồn dữ liệu sẽ đợi cho đến khi có vị trí trống trong bộ đệm để ghi dữ liệu mới.

Ví dụ:

```
Flux.range(1, 10)
    .onBackpressureBuffer(5)
    .subscribe(System.out::println);
```

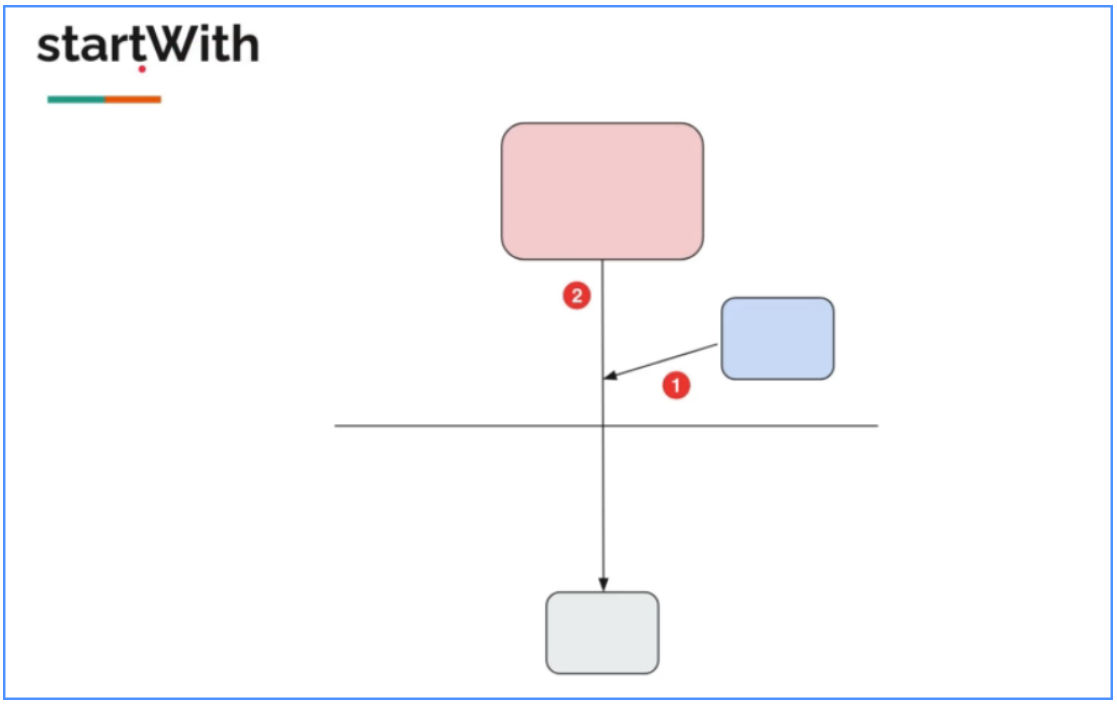
Trong ví dụ trên, Flux.range(1, 10) tạo ra một Flux phát ra các số từ 1 đến 10. Phép biến đổi onBackpressureBuffer(5) được sử dụng để áp dụng chiến lược Buffer With Size với kích thước bộ đệm là 5. Khi người tiêu thụ không thể xử lý nhanh chóng, dữ liệu sẽ được lưu trữ trong bộ đệm. Nếu bộ đệm đạt đến kích thước tối đa (5 phần tử), dữ liệu mới sẽ bị chặn cho đến khi có vị trí trống trong bộ đệm.

```
public static void main(String[] args) {
    // 75% 12
    System.setProperty("reactor.bufferSize.small", "16");

    Flux.create(fluxSink -> {
        for (int i = 1; i < 201 && !fluxSink.isCancelled(); i++) {
            fluxSink.next(i);
            System.out.println("Pushed : " + i);
            Util.sleepMillis(1);
        }
        fluxSink.complete();
    })
            .onBackpressureBuffer(50, o -> System.out.println("Dropped : "+ o))
            .publishOn(Schedulers.boundedElastic())

            .doOnNext(i -> {
                Util.sleepMillis(10);
            })
            .subscribe(Util.subscriber());
```

```
    Util.sleepSeconds(10);

}
```

Chapter 10: Combining Publishers

1. Start With



Trong Webflux, phương thức startWith được sử dụng để thêm một hoặc nhiều phần tử vào đầu của một Flux hoặc Mono. Điều này cho phép bạn bổ sung dữ liệu ban đầu trước khi phát ra các phần tử từ nguồn dữ liệu chính.

Dưới đây là một ví dụ về cách sử dụng startWith trong Webflux:

```
Flux<Integer> sourceFlux = Flux.just(3, 4, 5);
Flux<Integer> modifiedFlux = sourceFlux.startWith(1, 2);

modifiedFlux.subscribe(System.out::println);
```

Trong ví dụ trên, chúng ta có một Flux ban đầu sourceFlux chứa các số 3, 4, 5. Bằng cách sử dụng phương thức startWith, chúng ta thêm các số 1 và 2 vào đầu của Flux. Sau đó, chúng ta đăng ký một người tiêu thụ đơn giản để in ra các phần tử của Flux đã được sửa đổi.

Kết quả của ví dụ trên sẽ là:

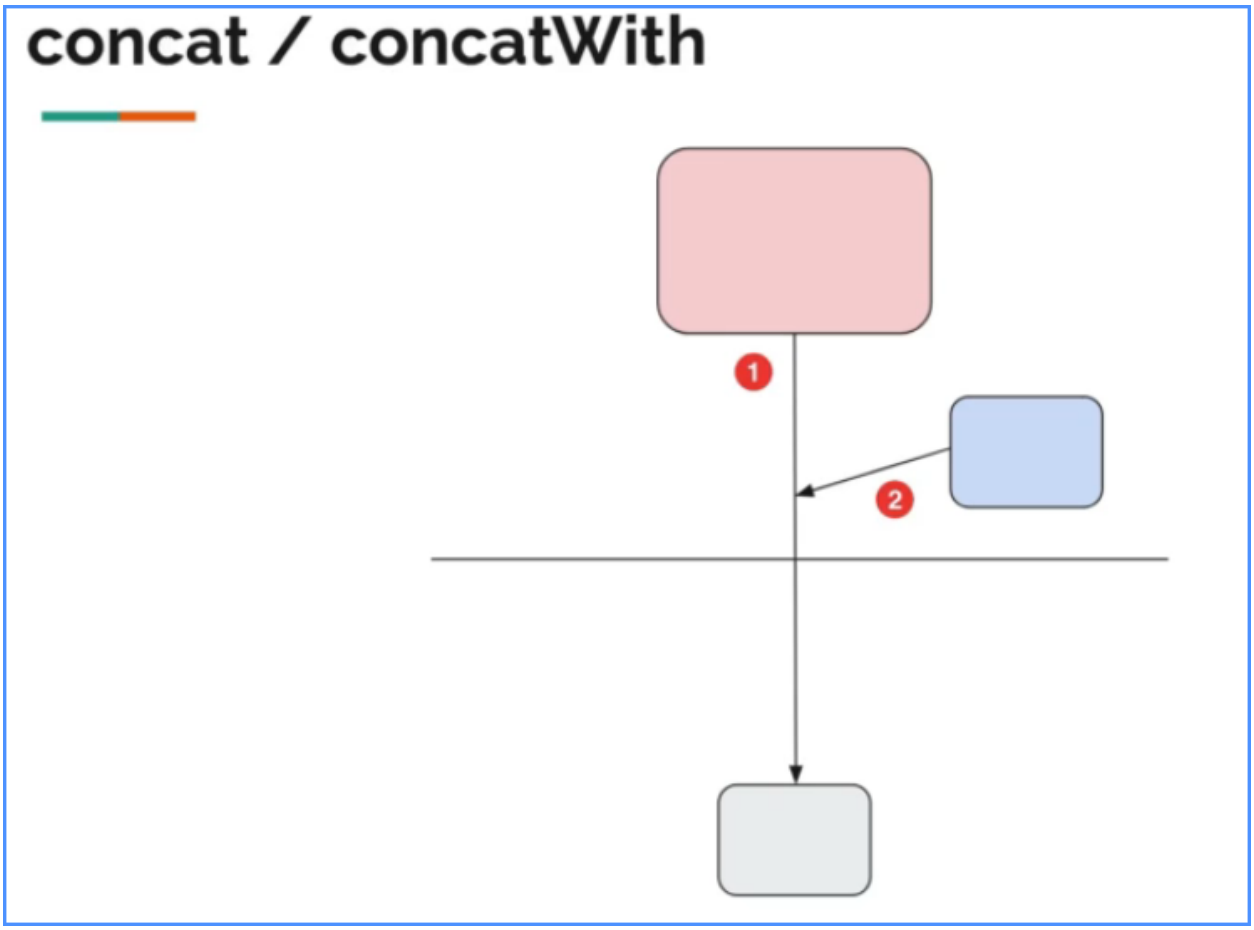
```
1
```

```
2
3
4
5
```

Các phần tử 1 và 2 được thêm vào đầu của Flux trước khi các phần tử từ nguồn dữ liệu ban đầu (3, 4, 5) được phát ra.

Phương thức startWith cũng có thể được sử dụng với Mono tương tự.

2. Concat



Trong Webflux, phương thức concat được sử dụng để kết hợp các Flux hoặc Mono lại với nhau theo thứ tự tuần tự. Nó giúp chúng ta kết nối các nguồn dữ liệu và phát ra các phần tử từng nguồn dữ liệu một sau khi hoàn thành nguồn dữ liệu trước.

Dưới đây là một ví dụ về cách sử dụng concat trong Webflux:

```
Flux<Integer> flux1 = Flux.just(1, 2, 3);
Flux<Integer> flux2 = Flux.just(4, 5, 6);
```

```
Flux<Integer> concatenatedFlux = Flux.concat(flux1, flux2);

concatenatedFlux.subscribe(System.out::println);
```

Trong ví dụ trên, chúng ta có hai Flux flux1 và flux2 chứa các số tương ứng là 1, 2, 3 và 4, 5, 6. Bằng cách sử dụng phương thức concat, chúng ta kết hợp hai Flux lại với nhau theo thứ tự tuần tự. Sau đó, chúng ta đăng ký một người tiêu thụ đơn giản để in ra các phần tử của Flux đã được kết hợp.

Kết quả của ví dụ trên sẽ là:

```
1
2
3
4
5
6
```

Các phần tử từ flux1 được phát ra trước, sau đó là các phần tử từ flux2. Các Flux được kết hợp lại thành một Flux duy nhất và phát ra các phần tử theo thứ tự tuần tự.

```
Flux<String> flux1 = Flux.just("a", "b");
Flux<String> flux2 = Flux.error(new RuntimeException("oops"));
Flux<String> flux3 = Flux.just("c", "d", "e");

Flux<String> flux = Flux.concatDelayError(flux1, flux2, flux3);

flux.subscribe(Util.subscriber());
```

Output:

```
/*
Received : a
Received : b
Received : c
Received : d
Received : e
ERROR : oops
 */
```

3. Merge

Trong Webflux, phương thức merge được sử dụng để kết hợp các Flux hoặc Mono lại với nhau mà không quan tâm đến thứ tự phát ra. Nó cho phép đồng thời phát ra các phần tử từ nhiều nguồn dữ liệu.

Dưới đây là một ví dụ về cách sử dụng merge trong Webflux:

```
Flux<Integer> flux1 = Flux.just(1, 2, 3);
Flux<Integer> flux2 = Flux.just(4, 5, 6);
Flux<Integer> mergedFlux = Flux.merge(flux1, flux2);

mergedFlux.subscribe(System.out::println);
```

Trong ví dụ trên, chúng ta có hai Flux flux1 và flux2 chứa các số tương ứng là 1, 2, 3 và 4, 5, 6. Bằng cách sử dụng phương thức merge, chúng ta kết hợp hai Flux lại với nhau mà không quan tâm đến thứ tự phát ra. Các phần tử từ các Flux sẽ được phát ra đồng thời mà không cần chờ Flux trước hoàn thành.

Kết quả của ví dụ trên có thể là:

```
/*
1
4
2
5
3
6
 */
```

Các phần tử từ flux1 và flux2 được phát ra đồng thời mà không quan tâm đến thứ tự ban đầu. Điều này cho phép các phần tử từ các nguồn dữ liệu được phát ra cùng lúc.

Phương thức merge cũng có thể được sử dụng với Mono để kết hợp các Mono lại với nhau mà không quan tâm đến thứ tự phát ra.

4. Zip

Trong Webflux, phương thức zip được sử dụng để kết hợp các Flux hoặc Mono lại với nhau bằng cách ghép cặp các phần tử tương ứng từ các nguồn dữ liệu. Nó tạo ra một Flux mới hoặc Mono mới chứa các cặp phần tử tương ứng từ các nguồn dữ liệu.

Dưới đây là một ví dụ về cách sử dụng zip trong Webflux:

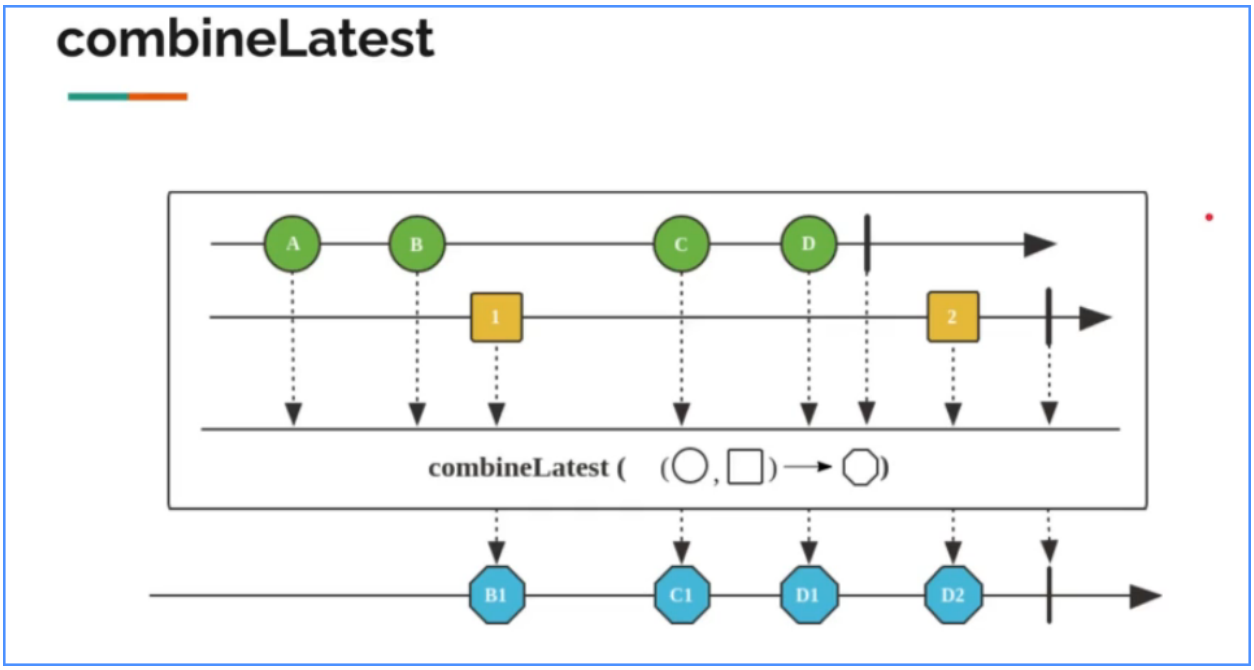
```
public static void main(String[] args) {
    Flux.zip(getBody(), getEngine(), getTires())
            .subscribe(Util.subscriber());

}

private static Flux<String> getBody(){
    return Flux.range(1, 5)
            .map(i -> "body");
}

private static Flux<String> getEngine(){
    return Flux.range(1, 3)
            .map(i -> "engine");
}

private static Flux<String> getTires(){
    return Flux.range(1, 6)
            .map(i -> "tires");
}
```

Bằng cách sử dụng phương thức zip, chúng ta kết hợp hai Flux lại với nhau và tạo ra một Flux mới chứa các cặp phần tử tương ứng từ hai nguồn dữ liệu.

Output:

```
/*
Received : [body,engine,tires]
Received : [body,engine,tires]
Received : [body,engine,tires]
 */
```

5. Combine Latest



Trong Webflux, phương thức combineLatest được sử dụng để kết hợp các Flux hoặc Mono lại với nhau bằng cách lấy giá trị mới nhất từ mỗi nguồn dữ liệu khi có sự thay đổi. Nó tạo ra một Flux mới hoặc Mono mới chứa các cặp giá trị mới nhất từ các nguồn dữ liệu.

Dưới đây là một ví dụ về cách sử dụng combineLatest trong Webflux:

```
public static void main(String[] args) {

    Flux.combineLatest(getString(), getNumber(), (s, i) -> s+i)
            .subscribe(Util.subscriber());

    Util.sleepSeconds(10);

}

private static Flux<String> getString(){
    return Flux.just("A", "B", "C", "D")
                .delayElements(Duration.ofSeconds(1));
}

private static Flux<Integer> getNumber(){
    return Flux.just(1, 2, 3)
            .delayElements(Duration.ofSeconds(3));
}
```


Mars

Chapter 11. Batching

1. Batching With Buffer

Trong WebFlux, phương thức buffer trong Flux được sử dụng để tạo ra các nhóm (batches) của các phần tử từ một Flux. Điều này cho phép xử lý một lượng lớn phần tử cùng lúc thay vì xử lý từng phần tử một.

Dưới đây là một ví dụ về cách sử dụng buffer trong WebFlux:

```
Flux.range(1, 10)
    .buffer(3)
    .subscribe(batch -> {
        System.out.println("Batch: " + batch);
        // Xử lý batch
    });
```

Trong ví dụ trên, chúng ta có một Flux từ 1 đến 10. Bằng cách sử dụng phương thức buffer(3), chúng ta tạo ra các nhóm có kích thước 3 phần tử. Khi các nhóm được tạo ra, chúng được truyền cho người tiêu thụ thông qua lambda expression trong subscribe. Trong ví dụ này, chúng ta đơn giản chỉ in các batch ra màn hình, nhưng bạn có thể thực hiện xử lý phức tạp hơn trên từng batch.

```
/* Output
Batch: [1, 2, 3]
Batch: [4, 5, 6]
Batch: [7, 8, 9]
Batch: [10]
 */
```

Các phần tử từ Flux được nhóm lại thành các batch có kích thước 3 phần tử. Các batch được truyền cho người tiêu thụ và in ra dưới dạng "Batch: [phần tử1, phần tử2, phần tử3]".

```
public static void main(String[] args) {
    eventStream()
            .buffer(5)
            .subscribe(Util.subscriber());
    Util.sleepSeconds(60);
}
private static Flux<String> eventStream(){
    return Flux.interval(Duration.ofMillis(800))
                .map(i -> "event"+i);
}
```

Output:

```
/*
Received : [event0, event1, event2, event3, event4]
Received : [event5, event6, event7, event8, event9]
Received : [event10, event11, event12, event13, event14]
...
 */
```

2. Batching With Buffer- When Complete Signal Is Emitted

```
3. public static void main(String[] args) {
4.     eventStream()
5.             .buffer(5)
6.             .subscribe(Util.subscriber());
7.     Util.sleepSeconds(60);
8. }
9.
10.private static Flux<String> eventStream(){
11.    return Flux.interval(Duration.ofMillis(800))
12.            .take(3)
13.                .map(i -> "event"+i);
14.}
15.
```

Output:

```
/*
Received : [event0, event1, event2]
Completed
 */
```

3. Batching With Buffer- Timeout

Trong WebFlux, bạn có thể sử dụng phương thức bufferTimeout trong Flux để tạo batch từ các phần tử và đặt một thời gian chờ (timeout). Khi đạt đến timeout, batch hiện tại sẽ được phát đi và một batch mới sẽ bắt đầu.

Dưới đây là một ví dụ về cách sử dụng bufferTimeout trong WebFlux:

```
Flux.range(1, 10)
    .bufferTimeout(3, Duration.ofSeconds(2))
    .subscribe(batch -> {
        System.out.println("Batch: " + batch);
        // Xử lý batch
    });
```

Trong ví dụ trên, chúng ta có một Flux từ 1 đến 10. Bằng cách sử dụng phương thức bufferTimeout(3, Duration.ofSeconds(2)), chúng ta tạo ra các batch có kích thước 3 phần tử và timeout là 2 giây. Điều này có nghĩa là nếu không có thêm phần tử mới trong khoảng thời gian 2 giây, batch hiện tại sẽ được phát đi và một batch mới sẽ bắt đầu.

Kết quả của ví dụ trên sẽ tùy thuộc vào tốc độ thực thi của Flux và timeout được đặt. Nếu Flux phát ra phần tử nhanh hơn timeout, các batch sẽ được tạo ra. Ví dụ, nếu các phần tử được phát ra mỗi giây, kết quả có thể là:

```
/*
Batch: [1, 2, 3]
Batch: [4, 5, 6]
Batch: [7, 8, 9]
Batch: [10]

 */
```

Trong trường hợp này, mỗi batch có kích thước 3 phần tử và timeout là 2 giây. Sau khi mỗi batch được tạo ra, nó sẽ được truyền cho người tiêu thụ và in ra màn hình.

Ví dụ tạo một Flux phát ra các phần tử theo khoảng thời gian cố định. Khi đạt đến timeout, batch hiện tại sẽ được phát đi và một batch mới sẽ bắt đầu.

```
public static void main(String[] args) {
    eventStream()
            .bufferTimeout(5, Duration.ofSeconds(2))
            .subscribe(Util.subscriber());
    Util.sleepSeconds(60);

}
private static Flux<String> eventStream(){
    return Flux.interval(Duration.ofMillis(800))
                .map(i -> "event"+i);
}
```

Output:

```
/*
Received : [event0, event1, event2]
Received : [event3, event4, event5]
Received : [event6, event7, event8]
Received : [event9, event10, event11]
Received : [event12, event13, event14]
Received : [event15, event16, event17]
Received : [event18, event19, event20]

 */
```

4. Batching With Buffer- Overlapping & Dropping

Trong WebFlux, khi sử dụng phương thức buffer để tạo batch từ một Flux, bạn có thể điều chỉnh cách xử lý các phần tử khi batch được tạo ra và lớn hơn kích thước batch tối đa. Hai tùy chọn phổ biến để xử lý trường hợp này là "overlapping" và "dropping".

1. Overlapping (chồng chéo): Khi sử dụng tùy chọn overlapping, các batch sẽ chồng chéo lên nhau và có các phần tử chung. Điều này có nghĩa là một phần tử có thể xuất hiện trong nhiều batch. Ví dụ:

```
Flux.range(1, 10)
    .buffer(3, 2) // Kích thước batch = 3, bước nhảy = 2
    .subscribe(batch -> {
        System.out.println("Batch: " + batch);
        // Xử lý batch
    });
```

Kết quả của ví dụ trên sẽ là:

```
/*
Batch: [1, 2, 3]
Batch: [3, 4, 5]
Batch: [5, 6, 7]
Batch: [7, 8, 9]
Batch: [9, 10]
 */
```

Trong ví dụ này, chúng ta có một Flux từ 1 đến 10. Bằng cách sử dụng buffer(3, 2), chúng ta tạo ra các batch có kích thước 3 phần tử và bước nhảy là 2. Điều này dẫn đến việc các batch chồng chéo lên nhau và có các phần tử chung. Ví dụ, phần tử "3" xuất hiện trong batch thứ nhất và batch thứ hai.

2. Dropping (loại bỏ): Khi sử dụng tùy chọn dropping, các phần tử vượt quá kích thước batch tối đa sẽ bị loại bỏ và không xuất hiện trong batch. Ví dụ:

```
Flux.range(1, 10)
    .buffer(3) // Kích thước batch = 3
    .subscribe(batch -> {
        System.out.println("Batch: " + batch);
        // Xử lý batch
    });
```

Kết quả của ví dụ trên sẽ là:

```
/*
Batch: [1, 2, 3]
Batch: [4, 5, 6]
Batch: [7, 8, 9]
Batch: [10]

 */
```

Trong ví dụ này, chúng ta cũng có một Flux từ 1 đến 10 và sử dụng buffer(3) để tạo batch có kích thước 3 phần tử. Tuy nhiên, trong trường hợp này, các phần tử vượt quá kích thước batch sẽ bị loại bỏ và không xuất hiện trong batch cuối cùng.

Tùy chọn overlapping và dropping cho phép bạn điều chỉnh cách xử lý các phần tử khi batch được tạo ra và lớn hơn kích thước batch tối đa. Bạn có thể chọn tùy chọn phù hợp với nhu cầu xử lý dữ liệu của bạn.

```
public static void main(String[] args) {
    eventStream()
            .buffer(3, 5)
            .subscribe(Util.subscriber());
    Util.sleepSeconds(60);
}


private static Flux<String> eventStream(){
    return Flux.interval(Duration.ofMillis(300))
            .map(i -> "event"+i);
}
```

Output:

```
/*
Received : [event0, event1, event2]
Received : [event5, event6, event7]
Received : [event10, event11, event12]
Received : [event15, event16, event17]
 */
```

5. Batching With Window

Trong Reactor và Spring WebFlux, bạn có thể sử dụng phương thức window để tạo các Flux con (windowed Flux) từ một Flux gốc dựa trên kích thước batch. Mỗi Flux con đại diện cho một batch của các phần tử trong Flux gốc.

Ví dụ, để tạo các windowed Flux với kích thước batch là 3, bạn có thể sử dụng phương thức window như sau:

```
Flux.range(1, 10)
    .window(3)
    .flatMap(windowFlux -> windowFlux.collectList())
    .subscribe(batch -> {
        System.out.println("Batch: " + batch);
        // Xử lý batch
    });
```

Kết quả của ví dụ trên sẽ là:

```
/*
Batch: [1, 2, 3]
Batch: [4, 5, 6]
Batch: [7, 8, 9]
Batch: [10]
 */
```

Trong ví dụ này, chúng ta sử dụng window(3) để tạo các windowed Flux với kích thước batch là


Mars

3. Mỗi windowed Flux sẽ chứa 3 phần tử từ Flux gốc. Chúng ta sử dụng phương thức flatMap để chuyển đổi mỗi windowed Flux thành một Flux con gồm một danh sách (List) các phần tử trong windowed Flux đó. Sau đó, chúng ta có thể xử lý batch bằng cách sử dụng phương thức subscribe.

Lưu ý rằng windowed Flux cuối cùng có thể có kích thước nhỏ hơn kích thước batch nếu số lượng phần tử trong Flux gốc không chia hết cho kích thước batch.

```
private static AtomicInteger atomicInteger = new AtomicInteger(1);

public static void main(String[] args) {

    eventStream()
            .window(3)
            .flatMap(flux -> saveEvents(flux))
            .subscribe(Util.subscriber());

    Util.sleepSeconds(60);

}

private static Flux<String> eventStream(){
    return Flux.interval(Duration.ofMillis(500))
            .map(i -> "event"+i);
}

private static Mono<Integer> saveEvents(Flux<String> flux){
    return flux
                .doOnNext(e -> System.out.println("saving " + e))
                .doOnComplete(() -> {
                    System.out.println("saved this batch");
                    System.out.println("-------------------");
                })
                .then(Mono.just(atomicInteger.getAndIncrement()));
}
```

Output:

```
/*
saving event0
saving event1
saving event2
saved this batch
-------------------
```

```
Received : 1
saving event3
saving event4
saving event5
saved this batch
--------------------
Received : 2
saving event6
saving event7
saving event8
saved this batch

 */
```

6. Batching With Group By

Trong Reactor và Spring WebFlux, bạn có thể sử dụng phương thức groupBy để tạo các Flux con (grouped Flux) từ một Flux gốc dựa trên một tiêu chí nhất định. Mỗi grouped Flux đại diện cho một nhóm các phần tử trong Flux gốc.

Ví dụ, để tạo các grouped Flux dựa trên giá trị chẵn/lẻ của các phần tử trong Flux gốc, bạn có thể sử dụng phương thức groupBy như sau:

```
Flux.range(1, 10)
    .groupBy(number -> number % 2 == 0 ? "even" : "odd")
    .subscribe(groupedFlux -> {
        groupedFlux.subscribe(number -> {
            System.out.println("Group: " + groupedFlux.key() + ", Value: " +
number);
            // Xử lý phần tử trong từng group
        });
    });
```

Kết quả của ví dụ trên sẽ là:

```
/*
Group: odd, Value: 1
Group: even, Value: 2
Group: odd, Value: 3
Group: even, Value: 4
Group: odd, Value: 5
```

```
Group: even, Value: 6
Group: odd, Value: 7
Group: even, Value: 8
Group: odd, Value: 9
Group: even, Value: 10

 */
```

Trong ví dụ này, chúng ta sử dụng groupBy để tạo các grouped Flux dựa trên giá trị chẵn/lẻ của các phần tử trong Flux gốc. Khi một phần tử trong Flux gốc được xử lý, nó sẽ được phân loại vào một trong hai group: "even" (chẵn) hoặc "odd" (lẻ). Chúng ta sử dụng phương thức subscribe trên grouped Flux để xử lý từng phần tử trong từng group. Bằng cách sử dụng groupedFlux.key(), chúng ta có thể truy cập vào giá trị của key (nhãn) của group hiện tại.

Chapter 12: Repeat & Retry

1. Repeat

Trong Reactor và Spring WebFlux, phương thức repeat được sử dụng để lặp lại việc phát các phần tử trong một Flux cho đến khi một điều kiện nhất định được đáp ứng. Nó cho phép bạn tạo một vòng lặp vô hạn hoặc lặp lại một số lần xác định.

Ví dụ, để tạo một Flux lặp lại vô hạn với các số nguyên từ 1 đến 5, bạn có thể sử dụng phương thức repeat như sau:

```
Flux.range(1, 5)
    .repeat()
    .subscribe(number -> System.out.println("Value: " + number));
```

Kết quả của ví dụ trên là một vòng lặp vô hạn với các số nguyên từ 1 đến 5:

```
/*
Value: 1
Value: 2
Value: 3
Value: 4
Value: 5
Value: 1
Value: 2
Value: 3
...
 */
```

Trong ví dụ này, chúng ta sử dụng repeat để lặp lại Flux gốc vô hạn. Khi Flux gốc phát hết các phần tử, nó sẽ được lặp lại và phát lại từ đầu. Quá trình này sẽ tiếp tục mãi mãi, tạo ra một vòng lặp vô hạn.

Ngoài việc lặp lại vô hạn, bạn cũng có thể sử dụng phương thức repeat với một đối số để xác định số lần lặp lại cụ thể. Ví dụ, để lặp lại Flux gốc 3 lần, bạn có thể sử dụng:

```
Flux.range(1, 5)
    .repeat(3)
    .subscribe(number -> System.out.println("Value: " + number));
```

Kết quả của ví dụ này sẽ là:

```
/*
```

```
Value: 1
Value: 2
Value: 3
Value: 4
Value: 5
Value: 1
Value: 2
Value: 3
Value: 4
Value: 5
Value: 1
Value: 2
Value: 3
Value: 4
Value: 5

 */
```

Trong ví dụ này, Flux gốc được lặp lại 3 lần, tạo ra một chuỗi phát các phần tử theo thứ tự từ Flux gốc.

Phương thức repeat trong Flux cho phép bạn lặp lại việc phát các phần tử để thực hiện các vòng lặp. Bằng cách sử dụng repeat với hoặc không có đối số, bạn có thể tạo ra các vòng lặp vô hạn hoặc lặp lại một số lần xác định.

2. Repeat With Condition

Trong Reactor và Spring WebFlux, bạn có thể sử dụng phương thức repeat kết hợp với một điều kiện để lặp lại việc phát các phần tử trong một Flux cho đến khi điều kiện đó không được đáp ứng nữa. Điều này cho phép bạn lặp lại Flux cho đến khi một điều kiện nhất định được thỏa mãn.

Ví dụ, để tạo một Flux lặp lại các số nguyên từ 1 đến 10 cho đến khi số đó là số chẵn, bạn có thể sử dụng phương thức repeat kết hợp với một điều kiện như sau:

```
Flux.range(1, 10)
    .repeat()
    .takeUntil(number -> number % 2 == 0)
    .subscribe(number -> System.out.println("Value: " + number));

// Cách khác
Flux.range(1, 10)
    .repeat(number -> number % 2 == 0)
    .subscribe(number -> System.out.println("Value: " + number));
```

Kết quả của ví dụ trên sẽ là:

```
/*
Value: 1
Value: 2
 */
```

Trong ví dụ này, chúng ta sử dụng repeat để lặp lại Flux gốc cho đến khi số chẵn xuất hiện. Sử dụng phương thức takeUntil, chúng ta kết thúc lặp lại khi một số chẵn được phát. Do đó, chỉ có hai phần tử đầu tiên của Flux gốc được phát ra trước khi lặp lại kết thúc.

3. Retry

Trong Reactor và Spring WebFlux, phương thức retry được sử dụng để thực hiện việc thử lại (retry) việc phát các phần tử trong một Flux khi một lỗi xảy ra. Nó cho phép bạn xử lý các tình huống lỗi và thử lại việc phát các phần tử để đảm bảo thành công.

Ví dụ, để thực hiện việc thử lại việc phát các phần tử trong một Flux với một số lần cố định, bạn có thể sử dụng phương thức retry như sau:

```
Flux.range(1, 5)
    .map(number -> {
        if (number == 3) {
            throw new RuntimeException("Error occurred");
        }
        return number;
    })
    .retry(3)
    .subscribe(
        number -> System.out.println("Value: " + number),
        error -> System.err.println("Error: " + error.getMessage())
    );
```

Trong ví dụ này, chúng ta sử dụng map để tạo một lỗi tại phần tử số 3 trong Flux gốc. Bằng cách sử dụng retry(3), chúng ta yêu cầu Flux thử lại việc phát các phần tử sau mỗi lần xảy ra lỗi, với một số lần cố định là 3. Nếu lỗi xảy ra, Flux sẽ thử lại việc phát các phần tử theo số lần được chỉ định. Nếu sau 3 lần thất bại, lỗi cuối cùng sẽ được truyền vào subscribe để xử lý.

Kết quả của ví dụ trên sẽ là:

```
/*
Value: 1
Value: 2
```

```
Value: 1
Value: 2
Value: 1
Value: 2
Error: Error occurred
 */
```

Trong ví dụ này, Flux cố gắng phát các phần tử 1 và 2, nhưng khi đến phần tử số 3, một lỗi xảy ra. Sau đó, Flux thử lại việc phát các phần tử 1 và 2 ba lần, và cuối cùng, khi lỗi vẫn xảy ra sau 3 lần thử, lỗi được truyền vào hàm xử lý lỗi trong subscribe.

Ngoài việc sử dụng số lần cố định, bạn cũng có thể sử dụng một lambda expression trong retry để xác định logic tùy chỉnh cho việc thử lại. Lambda expression sẽ nhận vào số lần đã thử lại và lỗi gốc và trả về một Flux mới để thử lại hoặc Mono.error để kết thúc việc thử lại.

```
public static void main(String[] args) {

    getIntegers()
            .retry(2)
            .subscribe(Util.subscriber());


}

private static Flux<Integer> getIntegers(){
    return Flux.range(1, 3)
            .doOnSubscribe(s -> System.out.println("Subscribed"))
            .doOnComplete(() -> System.out.println("--Completed"))
            .map(i -> i / (Util.faker().random().nextInt(1, 5) > 3 ? 0 : 1))
            .doOnError(err -> System.out.println("--error"));
}
```

4. Retry With Fixed Delay

Trong Reactor và Spring WebFlux, bạn có thể sử dụng phương thức retryWhen kết hợp với phương thức delayElements để thực hiện việc thử lại (retry) việc phát các phần tử trong một Flux với một độ trễ cố định giữa các lần thử lại. Điều này cho phép bạn xử lý các tình huống lỗi và tự động thực hiện việc thử lại với một độ trễ giữa các lần thử lại.

Ví dụ, để thực hiện việc thử lại việc phát các phần tử trong một Flux với một độ trễ cố định giữa các lần thử lại, bạn có thể sử dụng phương thức retryWhen kết hợp với delayElements như sau:

```
Flux.range(1, 5)
    .map(number -> {
        if (number == 3) {
            throw new RuntimeException("Error occurred");
        }
        return number;
    })
    .retryWhen(errors -> errors
        .zipWith(Flux.range(1, 3), (error, retryCount) -> retryCount)
        .flatMap(retryCount -> Mono.delay(Duration.ofSeconds(retryCount)))
    )
    .subscribe(
        number -> System.out.println("Value: " + number),
        error -> System.err.println("Error: " + error.getMessage())
    );
```

Trong ví dụ này, chúng ta sử dụng map để tạo một lỗi tại phần tử số 3 trong Flux gốc. Bằng cách sử dụng retryWhen, chúng ta xử lý luồng lỗi và tạo ra một Flux mới để thực hiện việc thử lại. Trong Flux thử lại này, chúng ta sử dụng zipWith để kết hợp lỗi gốc với một Flux chứa các số thứ tự từ 1 đến 3, đại diện cho số lần thử lại. Sau đó, chúng ta sử dụng flatMap để tạo một độ trễ sử dụng Mono.delay với một khoảng thời gian tăng dần dựa trên số lần thử lại. Cuối cùng, Flux thử lại này được sử dụng để thực hiện việc thử lại việc phát các phần tử.

Kết quả của ví dụ trên sẽ tùy thuộc vào độ trễ được xác định. Giả sử chúng ta sử dụng độ trễ trong đơn vị giây, kết quả có thể tương tự như sau:

```
/*
Value: 1
Value: 2
Value: 1
Value: 2
Value: 1
Value: 2
Value: 3

 */
```

Trong ví dụ này, Flux cố gắng phát các phần tử 1 và 2, nhưng khi đến phần tử số 3, một lỗi xảy ra. Sau đó, Flux thử lại việc phát các phần tử 1 và 2 với độ trễ tăng dần: lần thử lại thứ nhất có độ trễ 1 giây, lần thử lại thứ hai có độ trễ 2 giây, và lần thử lại thứ ba có độ trễ 3 giây. Cuối cùng, khi lỗi vẫn xảy ra sau 3 lần thử, lỗi cuối cùng được truyền vào hàm xử lý lỗi trong subscribe.

Bằng cách sử dụng retryWhen kết hợp với delayElements, bạn có thể thực hiện việc thử lại việc phát các phần tử trong Flux với một độ trễ cố định giữa các lần thử lại. Điều này cho phép bạn xử lý các tình huống lỗi và Điều này cho phép bạn xử lý các tình huống lỗi và tự động thực hiện việc thử lại với một độ trễ giữa các lần thử lại.

```
public static void main(String[] args) {

    getIntegers()
            .retryWhen(Retry.fixedDelay(2, Duration.ofSeconds(3)))
            .subscribe(Util.subscriber());

    Util.sleepSeconds(60);
}

private static Flux<Integer> getIntegers(){
    return Flux.range(1, 3)
            .doOnSubscribe(s -> System.out.println("Subscribed"))
            .doOnComplete(() -> System.out.println("--Completed"))
            .map(i -> i / (Util.faker().random().nextInt(1, 5) > 3 ? 0 : 1))
            .doOnError(err -> System.out.println("--error"));
}
```

5. Retry Spec

Trong Reactor và Spring WebFlux, bạn có thể sử dụng phương thức retryWhen kết hợp với các toán tử trong Reactor để thực hiện việc thử lại (retry) việc phát các phần tử trong một Flux dựa trên một logic tùy chỉnh. Điều này cho phép bạn xử lý các tình huống lỗi và xác định rõ hơn khi nào và cách thực hiện việc thử lại.

Ví dụ, để thực hiện việc thử lại việc phát các phần tử trong một Flux dựa trên một logic tùy chỉnh, bạn có thể sử dụng phương thức retryWhen kết hợp với các toán tử trong Reactor như flatMap và zipWith như sau:

```
Flux.range(1, 5)
    .map(number -> {
        if (number == 3) {
            throw new RuntimeException("Error occurred");
        }
        return number;
    })
    .retryWhen(errors -> errors
        .zipWith(Flux.range(1, 3), (error, retryCount) -> {
```

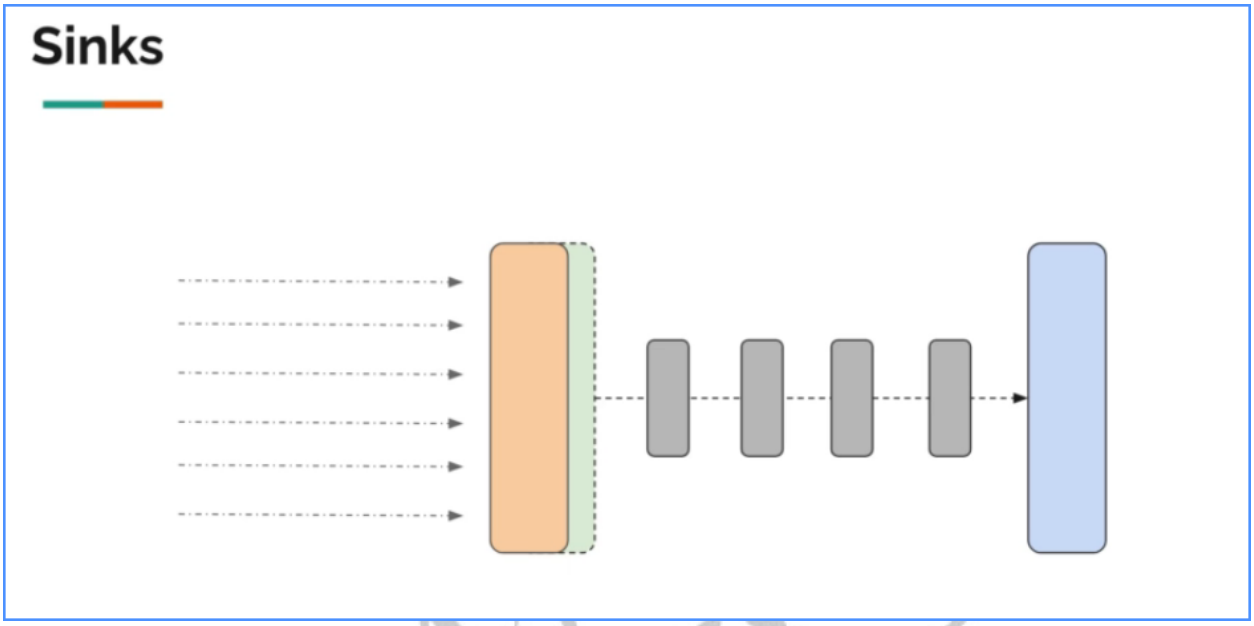
```
            if (retryCount < 3) {
                return retryCount;
            } else {
                throw Exceptions.propagate(error);
            }
        })
        .flatMap(retryCount -> Mono.delay(Duration.ofSeconds(retryCount)))
    )
    .subscribe(
        number -> System.out.println("Value: " + number),
        error -> System.err.println("Error: " + error.getMessage())
    );
```

Trong ví dụ này, chúng ta sử dụng map để tạo một lỗi tại phần tử số 3 trong Flux gốc. Bằng cách sử dụng retryWhen, chúng ta xử lý luồng lỗi và tạo ra một Flux mới để thực hiện việc thử lại dựa trên logic tùy chỉnh. Trong Flux thử lại này, chúng ta sử dụng zipWith để kết hợp lỗi gốc với một Flux chứa các số thứ tự từ 1 đến 3, đại diện cho số lần thử lại. Sau đó, chúng ta kiểm tra số lần thử lại và quyết định xem có tiếp tục thử lại hay không. Trong trường hợp này, nếu số lần thử lại nhỏ hơn 3, chúng ta trả về số lần thử lại để tiếp tục việc thử lại. Nếu số lần thử lại đạt đến 3, chúng ta sử dụng Exceptions.propagate để ném lỗi cuối cùng. Cuối cùng, chúng ta sử dụng flatMap để tạo một độ trễ sử dụng Mono.delay dựa trên số lần thử lại.

```
public static void main(String[] args) {

    orderService(Util.faker().business().creditCardNumber())
            .retryWhen(Retry.from(
                 flux -> flux
                            .doOnNext(rs -> {
                                System.out.println(rs.totalRetries());
                                System.out.println(rs.failure());
                            })
                            .handle((rs, synchronousSink) -> {
                                if(rs.failure().getMessage().equals("500"))
                                    synchronousSink.next(1);
                                else
                                    synchronousSink.error(rs.failure());
                            })
                            .delayElements(Duration.ofSeconds(1))
            ))
            .subscribe(Util.subscriber());

    Util.sleepSeconds(60);
```

```
}


// order service
private static Mono<String> orderService(String ccNumber){
    return Mono.fromSupplier(() -> {
        processPayment(ccNumber);
        return Util.faker().idNumber().valid();
    });
}

// payment service
private static void processPayment(String ccNumber){
    int random = Util.faker().random().nextInt(1, 10);
    if(random < 8)
        throw new RuntimeException("500");
    else if(random < 10)
        throw new RuntimeException("404");
}
```

Chapter 13. Sink



Trong Reactor và Spring WebFlux, một Sink là một cơ chế cho phép bạn thủ công phát giá trị vào một Flux. Nó cung cấp các phương thức để phát giá trị và quản lý luồng dữ liệu trong Flux.

Có một số loại Sink khác nhau trong Reactor, bao gồm:

1. Sinks.One: Đại diện cho một Sink có thể phát duy nhất một giá trị vào Flux. Bạn có thể sử dụng emitValue hoặc tryEmitValue để phát giá trị vào Flux.
2. Sinks.Many: Đại diện cho một Sink có thể phát nhiều giá trị vào Flux. Bạn có thể sử dụng emitNext hoặc tryEmitNext để phát từng giá trị vào Flux.
3. Sinks.Empty: Đại diện cho một Sink rỗng, không phát giá trị nào vào Flux. Bạn có thể sử dụng asMono để chuyển đổi nó thành một Mono.
4. Sinks.UnicastSpec: Đại diện cho một Sink hỗ trợ phát giá trị từ nhiều nguồn (multiple producers) vào Flux. Nó hỗ trợ các phương thức như onBackpressureBuffer, onBackpressureDrop, và onBackpressureLatest để quản lý dữ liệu khi có sự chênh lệch giữa tốc độ phát và tiêu thụ.

Với sử dụng của Sink, bạn có thể tạo và điều khiển sự phát giá trị vào Flux một cách tự do. Điều này rất hữu ích trong các trường hợp cần phát giá trị từ các nguồn không đồng bộ hoặc bên ngoài Flux, cho phép bạn tạo ra luồng dữ liệu theo ý muốn và kết hợp với các phương thức xử lý của Flux.

1. Sink One- Try Emit Value

Trong Reactor và Spring WebFlux, bạn có thể sử dụng Sink để thủ công phát một giá trị duy nhất vào một Flux. Sink cung cấp các phương thức để thực hiện việc này, và bạn có thể sử dụng tryEmitNext để thử phát một giá trị vào Flux và kiểm tra xem liệu việc phát có thành công hay không.

Dưới đây là một ví dụ minh họa về cách sử dụng Sink để thử phát một giá trị vào một Flux:

```
Sinks.One<Integer> sink = Sinks.one();

Flux<Integer> flux = sink.asMono()
    .concatWith(Flux.range(1, 5));

sink.tryEmitNext(10);

flux.subscribe(
    value -> System.out.println("Value: " + value),
    error -> System.err.println("Error: " + error),
    () -> System.out.println("Complete")
);
```

Trong ví dụ này, chúng ta tạo một Sinks.One<Integer> để đại diện cho một Sink có thể phát duy nhất một giá trị kiểu Integer. Sau đó, chúng ta tạo một Flux bằng cách sử dụng sink.asMono().concatWith(Flux.range(1, 5)). Điều này sẽ nối Mono của Sink với một Flux chứa các số từ 1 đến 5.

Tiếp theo, chúng ta sử dụng sink.tryEmitNext(10) để thử phát giá trị 10 vào Flux thông qua Sink. Phương thức tryEmitNext sẽ trả về true nếu việc phát thành công và false nếu không thành công. Bạn có thể kiểm tra giá trị trả về để biết việc phát có thành công hay không.

Cuối cùng, chúng ta đăng ký một Subscriber cho Flux và in ra các giá trị nhận được. Kết quả sẽ tương tự như sau:

```
/*
Value: 10
Value: 1
Value: 2
Value: 3
Value: 4
Value: 5
Complete
```

```
 */
```

Trong ví dụ này, giá trị 10 được phát thành công vào Flux thông qua Sink trước khi Flux phát các số từ 1 đến 5.

2. Sink One- Emit Failure Handler

Trong Reactor và Spring WebFlux, khi sử dụng Sink để phát giá trị vào một Flux, bạn có thể cung cấp một xử lý (handler) cho trường hợp phát lỗi (failure) thông qua phương thức asMono().onErrorResume. Điều này cho phép bạn định nghĩa một xử lý tùy chỉnh cho việc xử lý lỗi khi phát giá trị vào Flux.

Dưới đây là một ví dụ minh họa về cách sử dụng Sink và xử lý lỗi khi phát giá trị vào một Flux:

```
public static void main(String[] args) {

    // mono 1 value / empty / error
    Sinks.One<Object> sink = Sinks.one();

    Mono<Object> mono = sink.asMono();

    mono.subscribe(Util.subscriber("sam"));
    sink.emitValue("hi", (signalType, emitResult) -> {
        System.out.println(signalType.name());
        System.out.println(emitResult.name());
        return false;
    });
}
```

3. Sink One- Multiple Subscribers

Trong Reactor và Spring WebFlux, một Sink.One không hỗ trợ việc có nhiều Subscriber trực tiếp. Mỗi lần khi bạn đăng ký một Subscriber mới cho Flux từ Sink.One, nó sẽ tạo ra một Mono mới cho mỗi Subscriber. Điều này có nghĩa là mỗi Subscriber sẽ nhận được một luồng độc lập từ Flux, không chia sẻ dữ liệu với các Subscriber khác.

Dưới đây là một ví dụ minh họa về việc sử dụng Sink.One với nhiều Subscriber trên Flux:

```
public static void main(String[] args) {
    // mono 1 value / empty / error
    Sinks.One<Object> sink = Sinks.one();
    Mono<Object> mono = sink.asMono();
    mono.subscribe(Util.subscriber("sam"));
    mono.subscribe(Util.subscriber("mike"));
    sink.tryEmitValue("Hello");
}
```

4. Sinks Types

Sinks

Type	Behavior	Pub:Sub
one	Mono	1:N
many - unicast	Flux	1:1
many - multicast	Flux	1:N
many - replay	Flux	1:N (with replay of all values to late subscribers)

5. Sink Many- Unicast

Trong Reactor và Spring WebFlux, Sinks.Many và Sinks.UnicastSpec được sử dụng để tạo một Sink hỗ trợ phát nhiều giá trị từ nhiều nguồn (multiple producers) vào Flux. Điều này cho phép bạn phát giá trị từ các nguồn không đồng bộ và kết hợp chúng vào Flux.

Dưới đây là một ví dụ về việc sử dụng Sinks.Many và Sinks.UnicastSpec để tạo một Sink Unicast trong WebFlux:

```
public static void main(String[] args) {

    // handle through which we would push items
```

```
    Sinks.Many<Object> sink = Sinks.many().unicast().onBackpressureBuffer();

    // handle through which subscribers will receive items
    Flux<Object> flux = sink.asFlux();

    flux.subscribe(Util.subscriber("sam"));
    flux.subscribe(Util.subscriber("mike"));

    sink.tryEmitNext("hi");
    sink.tryEmitNext("how are you");
    sink.tryEmitNext("?");

}
```

Trong ví dụ này, chúng ta sử dụng Sinks.unsafe().many().unicast().onBackpressureBuffer() để tạo một Sinks.Many và Sinks.UnicastSpec để tạo một Sink Unicast hỗ trợ phát nhiều giá trị vào Flux.

Chúng ta sử dụng emitNext và tryEmitNext để phát các giá trị vào Sink. Chúng ta cũng sử dụng emitComplete để đánh dấu kết thúc của Sink.

Output:

```
/*
mike - ERROR : UnicastProcessor allows only a single Subscriber
sam - Received : hi
sam - Received : how are you
sam - Received : ?
 */
```

6. Are Sinks Thread Safe?

Trong Reactor và Spring WebFlux, các Sink như Sinks.One và Sinks.Many không được đảm bảo thread-safe. Điều này có nghĩa là không nên sử dụng chúng từ nhiều thread cùng một lúc mà không có sự đồng bộ hóa bên ngoài.

Nếu bạn cần sử dụng Sink từ nhiều thread, bạn cần đảm bảo rằng các hoạt động truy cập và sửa đổi Sink được đồng bộ hóa sử dụng các cơ chế như synchronized hoặc Lock. Điều này đảm bảo rằng các thao tác trên Sink không xảy ra cùng một lúc từ nhiều thread và giúp tránh các vấn đề như đọc/ghi không đồng bộ hoặc đua condition.

Ví dụ:

```
public static void main(String[] args) {


    // handle through which we would push items
    Sinks.Many<Object> sink = Sinks.many().unicast().onBackpressureBuffer();

    // handle through which subscribers will receive items
    Flux<Object> flux = sink.asFlux();
    List<Object> list = new ArrayList<>();

    flux.subscribe(list::add);
    for (int i = 0; i < 1000; i++) {
        final int j = i;
        CompletableFuture.runAsync(() -> {
            sink.emitNext(j, (s, e) -> true);
        });
    }

    Util.sleepSeconds(3);
    System.out.println(list.size());
}
```

---

7. Sink Many- Multicast

```
public static void main(String[] args) {

    // xử lý mà qua đó chúng ta sẽ push items
    Sinks.Many<Object> sink = Sinks.many().multicast().onBackpressureBuffer();

    // xử lý thông qua đó subscribers sẽ nhận được items
    Flux<Object> flux = sink.asFlux();

    sink.tryEmitNext("hi");
    sink.tryEmitNext("how are you");
    flux.subscribe(Util.subscriber("sam"));
    flux.subscribe(Util.subscriber("mike"));
    sink.tryEmitNext("?");
    flux.subscribe(Util.subscriber("jake"));
    sink.tryEmitNext("new msg");
}
```

Output:

```
/*
sam - Received : hi
sam - Received : how are you
sam - Received : ?
mike - Received : ?
sam - Received : new msg
mike - Received : new msg
jake - Received : new msg

 */
```

8. Sink Many- Multicast- Disable History

// handle through which we would push items

```
Sinks.Many<Object> sink = Sinks.many().multicast().directAllOrNothing();
```

Output:

```
/*
sam - Received : ?
mike - Received : ?
sam - Received : new msg
mike - Received : new msg
jake - Received : new msg
 */
```

9. Sink Many- Multicast- Direct Best Effort

Sinks.many().multicast().directBestEffort() được sử dụng để tạo một Sinks.Many với multicast và chế độ direct best effort.

Trong Reactor và Spring WebFlux, multicast cho phép phát dữ liệu đến nhiều Subscriber, trong khi direct best effort là một chế độ multicast mà không lưu trữ lịch sử dữ liệu. Điều này có nghĩa là khi một Subscriber mới đăng ký, nó chỉ nhận được các giá trị mới nhất và không có khả năng nhận các giá trị trước đó.

```
Sinks.Many<Object> sink = Sinks.many().multicast().directBestEffort();
```

Output:

```
/*
am - Received : 0
sam - Received : 1
sam - Received : 2
sam - Received : 3
sam - Received : 4
sam - Received : 5
.....

 */
```

10. Sink Many- Replay

Trong Reactor và Spring WebFlux, bạn có thể sử dụng Sinks.Many với Replay để tạo một multicast và lưu trữ lịch sử dữ liệu. Điều này cho phép các Subscriber mới đăng ký nhận được lịch sử dữ liệu đã được phát trước đó.

Dưới đây là một ví dụ về việc sử dụng Replay trong Sinks.Many trong WebFlux:

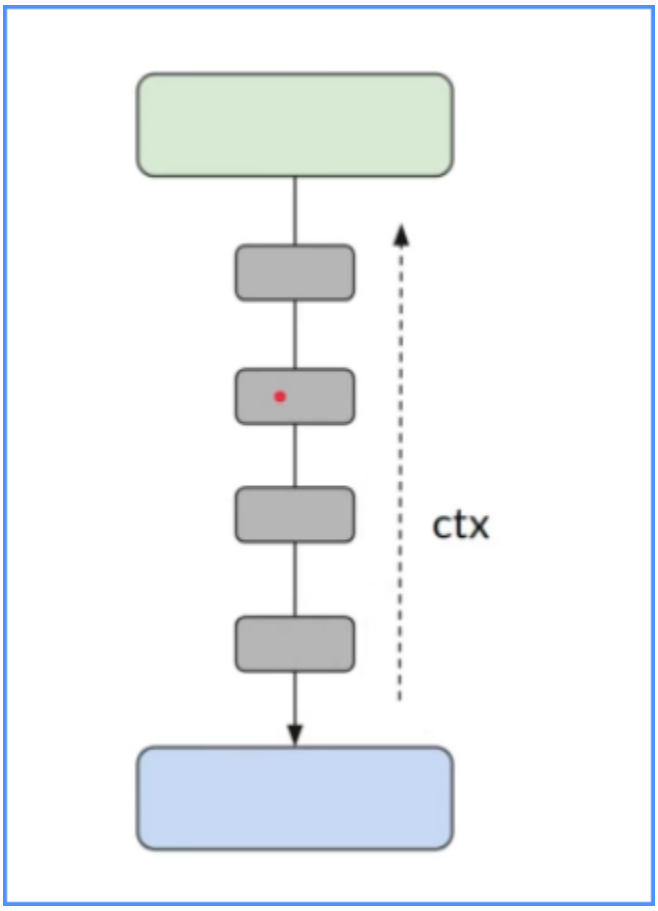
```
public static void main(String[] args) {

    // handle through which we would push items
    Sinks.Many<Object> sink = Sinks.many().replay().all();

    // handle through which subscribers will receive items
    Flux<Object> flux = sink.asFlux();


    sink.tryEmitNext("hi");
    sink.tryEmitNext("how are you");

    flux.subscribe(Util.subscriber("sam"));
    flux.subscribe(Util.subscriber("mike"));
    sink.tryEmitNext("?");
    flux.subscribe(Util.subscriber("jake"));

    sink.tryEmitNext("new msg");
```

```
}
```

Chapter 14. Context

Trong Spring WebFlux, Context là một cơ chế mạnh mẽ để chia sẻ thông tin giữa các phần của ứng dụng trong quá trình xử lý bất đồng bộ. Nó cho phép bạn lưu trữ và truy xuất các dữ liệu liên quan đến một nhiệm vụ cụ thể mà không cần truyền qua các tham số.

Context trong WebFlux được triển khai bằng cách sử dụng lớp reactor.util.context.Context. Đối tượng Context lưu trữ các cặp khóa-giá trị và có thể được truy xuất trong quá trình xử lý sự kiện bất đồng bộ bằng cách sử dụng phương thức Mono/Flux#subscriberContext() hoặc Mono/Flux#subscriberContext(Function<Context, Context>).



```
public static void main(String[] args) {

    getWelcomeMessage()
            .contextWrite(ctx -> ctx.put("user",
ctx.get("user").toString().toUpperCase()))
            .contextWrite(Context.of("user", "sam"))
            .subscribe(Util.subscriber());

}

private static Mono<String> getWelcomeMessage(){
    return Mono.deferContextual(ctx -> {
        if(ctx.hasKey("user")){
            return Mono.just("Welcome  " + ctx.get("user"));
        }else{
            return Mono.error(new RuntimeException("unauthenticated"));
        }
    });
}
```

1. Phương thức này sử dụng Mono.deferContextual() để truy xuất Context và thực hiện xử lý dựa trên nó. Nếu Context có khóa "user", nó trả về Mono với thông báo chào mừng "Welcome" và giá trị của "user". Nếu không, nó sẽ tạo một Mono lỗi với RuntimeException "unauthenticated".
2. .contextWrite(ctx -> ctx.put("user", ctx.get("user").toString().toUpperCase())): Đây là một phần của chuỗi gọi phương thức trên Mono và thực hiện một thay đổi trên Context. Nó sử dụng contextWrite() để thêm hoặc cập nhật giá trị của "user" trong Context. Trong trường hợp này, giá trị "user" được chuyển đổi thành chữ hoa.
3. .contextWrite(Context.of("user", "sam")): Đây là một phần khác của chuỗi gọi phương thức trên Mono và thực hiện một thay đổi khác trên Context. Nó sử dụng contextWrite() để thiết lập giá trị "user" trong Context thành "sam". Điều này có nghĩa là khi Mono được đăng ký, Context sẽ có thông tin "user" là "sam".
4. .subscribe(Util.subscriber()): Đây là phần cuối cùng của chuỗi gọi phương thức trên Mono và đăng ký một Subscriber để nhận kết quả. Trong ví dụ này, Util.subscriber() được sử dụng để đăng ký một Subscriber mặc định để in ra kết quả.

Chapter 15. Unit Testing

1. Step Verifier- Expect Next

Trong unit test của Spring WebFlux, StepVerifier cung cấp các phương thức để xác định và kiểm tra kết quả của một luồng Reactive. Một trong những phương thức quan trọng của StepVerifier là expectNext(), được sử dụng để xác định kết quả kỳ vọng của bước tiếp theo trong luồng Reactive.

Dưới đây là một ví dụ về việc sử dụng expectNext() trong unit test của WebFlux:

```
import org.junit.jupiter.api.Test;
import reactor.core.publisher.Flux;
import reactor.test.StepVerifier;

public class WebFluxUnitTest {

    @Test
    public void testFlux() {
        Flux<Integer> flux = Flux.just(1, 2, 3, 4, 5);

        StepVerifier.create(flux)
                .expectNext(1)
                .expectNext(2)
                .expectNext(3)
                .expectNext(4)
                .expectNext(5)
//              .expectNext(1, 2, 3)
                .verifyComplete();
    }
}
```

Trong ví dụ trên, chúng ta tạo một Flux với các giá trị từ 1 đến 5. Tiếp theo, chúng ta sử dụng StepVerifier.create() để tạo một StepVerifier cho Flux này. Với StepVerifier, chúng ta có thể định nghĩa các kỳ vọng của mình về luồng Reactive.

Với expectNext(), chúng ta xác định kết quả kỳ vọng của bước tiếp theo trong luồng Reactive. Trong ví dụ trên, chúng ta mong đợi các số từ 1 đến 5 lần lượt xuất hiện. Nếu các kết quả không khớp, StepVerifier sẽ báo lỗi.

Cuối cùng, chúng ta sử dụng verifyComplete() để kiểm tra xem luồng Reactive đã hoàn thành thành công hay không. Nếu luồng Reactive không hoàn thành, StepVerifier sẽ báo lỗi.

2. Step Verifier- Error

Trong unit test của Spring WebFlux, StepVerifier cung cấp phương thức expectError() để kiểm tra xem luồng Reactive có phát sinh lỗi như mong đợi hay không.

Dưới đây là một ví dụ về việc sử dụng expectError() trong unit test của WebFlux:

```
import org.junit.jupiter.api.Test;
import reactor.core.publisher.Flux;
import reactor.test.StepVerifier;

public class WebFluxUnitTest {

    @Test
    public void testFluxWithError() {
        Flux<Integer> flux = Flux.error(new RuntimeException("Error occurred"));

        StepVerifier.create(flux)
                .expectError(RuntimeException.class)
                .verify();
    }
    @Test
    public void test1(){
        Flux<Integer> just = Flux.just(1, 2, 3);
        Flux<Integer> error = Flux.error(new RuntimeException("oops"));
        Flux<Integer> concat = Flux.concat(just, error);

        StepVerifier.create(concat)
                .expectNext(1, 2, 3)
                .verifyError();
    }

    @Test
    public void test2(){
        Flux<Integer> just = Flux.just(1, 2, 3);
        Flux<Integer> error = Flux.error(new RuntimeException("oops"));
        Flux<Integer> concat = Flux.concat(just, error);

        StepVerifier.create(concat)
                .expectNext(1, 2, 3)
                .verifyError(RuntimeException.class);
    }

    @Test
```

```
    public void test3(){
        Flux<Integer> just = Flux.just(1, 2, 3);
        Flux<Integer> error = Flux.error(new RuntimeException("oops"));
        Flux<Integer> concat = Flux.concat(just, error);

        StepVerifier.create(concat)
                .expectNext(1, 2, 3)
                .verifyErrorMessage("oops");
    }
}
```

Trong ví dụ trên, chúng ta tạo một Flux với một lỗi được phát sinh bằng cách sử dụng Flux.error(). Chúng ta truyền một RuntimeException với thông báo "Error occurred" vào phương thức Flux.error().

Tiếp theo, chúng ta sử dụng StepVerifier.create() để tạo một StepVerifier cho Flux này. Với StepVerifier, chúng ta có thể định nghĩa các kỳ vọng của mình về luồng Reactive.

Với expectError(), chúng ta xác định kiểu lỗi mong đợi trong luồng Reactive. Trong ví dụ trên, chúng ta mong đợi một RuntimeException được phát sinh. Nếu không có lỗi hoặc loại lỗi không khớp, StepVerifier sẽ báo lỗi.

Cuối cùng, chúng ta sử dụng verify() để hoàn thành kiểm tra. Nếu luồng Reactive không phát sinh lỗi như mong đợi, StepVerifier sẽ báo lỗi.

3. Step Verifier- Expect Next Count & Consume While

Trong unit test của Spring WebFlux, StepVerifier cung cấp các phương thức expectNextCount() và consumeWhile() để kiểm tra số lượng phần tử và kiểm tra một điều kiện trong luồng Reactive.

Dưới đây là ví dụ về việc sử dụng expectNextCount() và consumeWhile() trong unit test của WebFlux:

```
@Test
public void test1(){
    Flux<Integer> range = Flux.range(1, 50);
    StepVerifier.create(range)
            .expectNextCount(50)
            .verifyComplete();
}

@Test
public void test2(){
```

```
    Flux<Integer> range = Flux.range(1, 50);
    StepVerifier.create(range)
            .thenConsumeWhile(i -> i < 100)
            .verifyComplete();
}
```

Bằng cách sử dụng expectNextCount(50), chúng ta xác định rằng chúng ta mong đợi Flux này phải có 50 phần tử. Nếu số lượng phần tử không khớp, StepVerifier sẽ báo lỗi.

Trong ví dụ thứ hai (test2()), chúng ta tiếp tục sử dụng Flux với các giá trị từ 1 đến 50. Bằng cách sử dụng consumeWhile(), chúng ta kiểm tra xem các phần tử trong Flux có thỏa mãn một điều kiện cụ thể hay không. Trong trường hợp này, chúng ta kiểm tra xem các phần tử có nhỏ hơn 100 không. Nếu điều kiện không được thỏa mãn, StepVerifier sẽ báo lỗi.

---

4. Step Verifier- Delay Test

Trong unit test của Spring WebFlux, StepVerifier cung cấp phương thức verify(Duration) để xử lý kiểm tra độ trễ trong luồng Reactive.

Dưới đây là một ví dụ về việc sử dụng verify(Duration) trong unit test của WebFlux:

```
@Test
public void test2(){

    Mono<BookOrder> mono = Mono.fromSupplier(() -> new BookOrder())
                               .delayElement(Duration.ofSeconds(3));

    StepVerifier.create(mono)
            .assertNext(b -> Assertions.assertNotNull(b.getAuthor()))
            .expectComplete()
            .verify(Duration.ofSeconds(4));
}
```

---

5. Step Verifier- Virtual Time Test

Phương thức StepVerifier.withVirtualTime() trong Spring WebFlux được sử dụng để tạo một StepVerifier hoạt động trong môi trường thời gian ảo (virtual time). Thời gian ảo cho phép bạn kiểm tra các luồng Reactive có sự tương tác với thời gian, chẳng hạn như độ trễ hoặc giới hạn thời gian, mà không cần đợi thực tế.

Khi sử dụng StepVerifier.withVirtualTime(), bạn có thể tùy chỉnh thời gian diễn ra trong luồng Reactive bằng cách sử dụng các phương thức như thenAwait(Duration) hoặc expectNoEvent(Duration) để chờ hoặc kiểm tra sự kiện trong khoảng thời gian ảo.

Dưới đây là một ví dụ về việc sử dụng StepVerifier.withVirtualTime() trong unit test của WebFlux:

```
import org.junit.jupiter.api.Test;
import reactor.core.publisher.Flux;
import reactor.test.StepVerifier;

import java.time.Duration;

public class WebFluxUnitTest {

    @Test
    public void testFluxWithVirtualTime() {
        StepVerifier.withVirtualTime(() ->
Flux.interval(Duration.ofSeconds(1)).take(5))
                .expectSubscription()
                .expectNoEvent(Duration.ofSeconds(1))
                .expectNext(0L)
                .thenAwait(Duration.ofSeconds(1))
                .expectNext(1L)
                .thenAwait(Duration.ofSeconds(1))
                .expectNext(2L)
                .thenAwait(Duration.ofSeconds(1))
                .expectNext(3L)
                .thenAwait(Duration.ofSeconds(1))
                .expectNext(4L)
                .verifyComplete();
    }
}
```

Trong ví dụ trên, chúng ta sử dụng StepVerifier.withVirtualTime() để tạo một StepVerifier hoạt động trong thời gian ảo. Trong StepVerifier này, chúng ta tạo một Flux sử dụng

Flux.interval(Duration.ofSeconds(1)).take(5), chẳng hạn như một dãy số từ 0 đến 4 được phát ra mỗi giây.

Sau đó, chúng ta định nghĩa các kỳ vọng của mình với các phương thức như expectNoEvent(Duration) và thenAwait(Duration) để kiểm tra sự kiện và đợi trong thời gian ảo. Ví dụ này mong đợi một đăng ký, sau đó chờ 1 giây, kiểm tra phần tử tiếp theo là 0, sau đó chờ 1 giây, kiểm tra phần tử tiếp theo là 1, và tiếp tục như vậy cho tới khi kiểm tra phần tử cuối cùng là 4. Cuối cùng, chúng ta sử dụng verifyComplete() để kiểm tra xem luồng Reactive đã hoàn thành thành công hay không.

Bằng cách sử dụng StepVerifier.withVirtualTime(), bạn có thể kiểm tra các tương tác thời gian trong luồng Reactive mà không phụ thuộc vào thực tế thời gian, cho phép bạn kiểm tra các kịch bản phức tạp và đảm bảo sự đồng bộ và sự kiểm soát đúng đắn trong các unit test của bạn.

```
@Test
public void test1(){
    StepVerifier.withVirtualTime(() -> timeConsumingFlux())
            .thenAwait(Duration.ofSeconds(30))
            .expectNext("1a", "2a", "3a", "4a")
            .verifyComplete();
}

@Test
public void test2(){
    StepVerifier.withVirtualTime(() -> timeConsumingFlux())
            .expectSubscription() // sub is an event
            .expectNoEvent(Duration.ofSeconds(4))
            .thenAwait(Duration.ofSeconds(20))
            .expectNext("1a", "2a", "3a", "4a")
            .verifyComplete();
}


private Flux<String> timeConsumingFlux(){
    return Flux.range(1, 4)
                .delayElements(Duration.ofSeconds(5))
                .map((i -> i + "a"));
}
```

6. Step Verifier- Scenario Name

Trong unit test của Spring WebFlux, bạn có thể sử dụng phương thức scenario(String) của StepVerifier để đặt tên cho các kịch bản (scenario) kiểm tra của mình. Điều này giúp bạn định danh và mô tả rõ ràng mục tiêu kiểm tra của mỗi kịch bản.

Dưới đây là một ví dụ về việc sử dụng scenario(String) trong unit test của WebFlux:

```
import org.junit.jupiter.api.Test;
import reactor.core.publisher.Flux;
import reactor.test.StepVerifier;

public class WebFluxUnitTest {

    @Test
    public void testFluxScenarioName() {
        Flux<Integer> flux = Flux.just(1, 2, 3, 4, 5);

        StepVerifier.create(flux)
                .scenario("Test Flux")
                .expectNext(1)
                .expectNext(2)
                .expectNext(3)
                .expectNext(4)
                .expectNext(5)
                .verifyComplete();
    }
}
```

Trong ví dụ trên, chúng ta sử dụng StepVerifier.create() để tạo một StepVerifier cho Flux. Tiếp theo, chúng ta sử dụng scenario("Test Flux") để đặt tên cho kịch bản kiểm tra của mình là "Test Flux".

Khi bạn chạy unit test, tên kịch bản sẽ được hiển thị trong kết quả của báo cáo kiểm tra, giúp bạn nhận biết rõ ràng mục tiêu kiểm tra của từng kịch bản. Điều này đặc biệt hữu ích khi có nhiều kịch bản kiểm tra và bạn cần xác định rõ ràng mỗi kịch bản làm gì.

Bằng cách sử dụng scenario(String), bạn có thể tạo các kịch bản kiểm tra đồng thời đặt tên cho chúng, giúp bạn quản lý và xác định rõ ràng mục tiêu kiểm tra của mỗi kịch bản trong unit test của bạn.

7. Step Verifier- Context

Trong unit test của Spring WebFlux, StepVerifier cung cấp một khả năng quan trọng là withInitialContext(). Phương thức withInitialContext(Context) cho phép bạn thiết lập và truy cập vào thông tin ngữ cảnh (context) trong quá trình kiểm tra luồng Reactive.

Bạn có thể sử dụng withInitialContext() để đặt thông tin ngữ cảnh (context) và sau đó sử dụng nó trong các bước kiểm tra hoặc các phép toán khác.

Dưới đây là một ví dụ về việc sử dụng withInitialContext() trong unit test của WebFlux:

```
@Test
public void test1(){
    StepVerifier.create(getWelcomeMessage())
            .verifyError(RuntimeException.class);

}

@Test
public void test2(){
    StepVerifierOptions options =
StepVerifierOptions.create().withInitialContext(Context.of("user", "sam"));
    StepVerifier.create(getWelcomeMessage(), options)
            .expectNext("Welcome sam")
            .verifyComplete();
}
```