Reactor Core

Reactor Core é uma biblioteca Java 8 que implementa o modelo de programação reativa. Ele é desenvolvido com base na especificação Reactive Streams , um padrão para a criação de aplicativos reativos.

Especificação de Fluxos Reativos:

Antes de olharmos para o Reactor, devemos olhar para a Especificação de Fluxos Reativos. Isso é o que o Reactor implementa e estabelece as bases para a biblioteca.

Essencialmente, Reactive Streams é uma especificação para processamento de fluxo assíncrono.

Em outras palavras, um sistema onde muitos eventos estão sendo produzidos e consumidos de forma assíncrona. Pense em um fluxo de milhares de atualizações de estoque por segundo entrando em um aplicativo financeiro e em ter que responder a essas atualizações em tempo hábil.

Um dos principais objetivos disso é resolver o problema da contrapressão. Se tivermos um produtor que está emitindo eventos para um consumidor mais rápido do que pode processá-los, eventualmente o consumidor ficará sobrecarregado com eventos, ficando sem recursos do sistema.

Contrapressão significa que nosso consumidor deve ser capaz de dizer ao produtor quantos dados enviar para evitar isso, e é isso que está definido na especificação.

Produzindo um fluxo de dados:

Para que um aplicativo seja reativo, a primeira coisa que ele deve ser capaz de fazer é produzir um fluxo de dados.

Isso pode ser algo como o exemplo de atualização de estoque que demos anteriormente. Sem esses dados, não teríamos nada a que reagir, e é por isso que este é um primeiro passo lógico.

O Reactive Core nos dá dois tipos de dados que nos permitem fazer isso.

Flux: A primeira maneira de fazer isso é com *o Flux* . É um stream que pode emitir *0..n* elementos. Vamos tentar criar um simples:

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

Neste caso, temos um fluxo estático de quatro elementos.

Mono: A segunda maneira de fazer isso é com um *Mono* , que é um fluxo de *0..1* elementos. Vamos tentar instanciar um:

```
Mono<Integer> just = Mono.just(1);
```

Isso parece e se comporta quase exatamente igual ao *Flux* , só que desta vez estamos limitados a não mais de um elemento.

Por que não usar apenas flux?

Antes de experimentar mais, vale a pena destacar por que temos esses dois tipos de dados.

Primeiro, deve-se notar que tanto *o Flux* quanto *o Mono* são implementações da interface Reactive Streams *Publisher* . Ambas as classes são compatíveis com a especificação e poderíamos usar esta interface em seu lugar:

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

Mas, realmente, conhecer essa cardinalidade é útil. Isso ocorre porque algumas operações só fazem sentido para um dos dois tipos e porque pode ser mais expressivo (imagine *findOne()* em um repositório).

Coletando Elementos:

Vamos usar o método *Subscribe()* para coletar todos os elementos em um stream:

```
List<Integer> elements = new ArrayList<>();
Flux.just(1, 2, 3, 4)
  .log()
  .subscribe(elements::add);
assertThat(elements).containsExactly(1, 2, 3, 4);
```

Os dados não começarão a fluir até que nos inscrevamos. Observe que adicionamos alguns logs também, isso será útil quando olharmos para o que está acontecendo nos bastidores.

O fluxo de elementos:

Com o login ativado, podemos usá-lo para visualizar como os dados estão fluindo em nosso fluxo:

```
20:25:19.550 [main] INFO  reactor.Flux.Array.1 - | onSubscribe([Synchronous Fuseable] FluxArray.ArraySubscription)
20:25:19.553 [main] INFO  reactor.Flux.Array.1 - | request(unbounded)
20:25:19.553 [main] INFO  reactor.Flux.Array.1 - | onNext(1)
20:25:19.553 [main] INFO  reactor.Flux.Array.1 - | onNext(2)
20:25:19.553 [main] INFO  reactor.Flux.Array.1 - | onNext(3)
20:25:19.553 [main] INFO  reactor.Flux.Array.1 - | onNext(4)
20:25:19.553 [main] INFO  reactor.Flux.Array.1 - | onComplete()
```

Em primeiro lugar, tudo está sendo executado no thread principal. Não vamos entrar em detalhes sobre isso, pois veremos mais a fundo a simultaneidade mais adiante neste artigo. Isso torna as coisas simples, pois podemos lidar com tudo em ordem.

Agora vamos percorrer a sequência que registramos um por um:

1. *onSubscribe()* – Isso é chamado quando nos inscrevemos em nosso stream
2. *request(unbounded)* – Quando chamamos *subscrever* , nos bastidores estamos criando uma *Assinatura* . Esta assinatura solicita elementos do stream. Nesse caso, o padrão é *ilimitado,* o que significa que solicita todos os elementos disponíveis
3. *onNext()* – Isso é chamado em cada elemento
4. *onComplete()* – Isso é chamado por último, depois de receber o último elemento. Na verdade , também existe um *onError() , que seria chamado se houvesse uma exceção, mas, neste caso, não há*

Este é o fluxo definido na interface *do Assinante* como parte da Especificação de Streams Reativos e, na realidade, é o que foi instanciado nos bastidores em nossa chamada para *onSubscribe().* É um método útil, mas para entender melhor o que está acontecendo, vamos fornecer uma interface *de assinante diretamente:*

```
Flux.just(1, 2, 3, 4)
  .log()
  .subscribe(new Subscriber<Integer>() {
    @Override
    public void onSubscribe(Subscription s) {
      s.request(Long.MAX_VALUE);
    }
    @Override
    public void onNext(Integer integer) {
      elements.add(integer);
    }
    @Override
    public void onError(Throwable t) {}
    @Override
    public void onComplete() {}
});
```

Podemos ver que cada estágio possível no fluxo acima é mapeado para um método na implementação do *Assinante* . Acontece que *o Flux* nos forneceu um método auxiliar para reduzir essa verbosidade.

Contrapressão:

A próxima coisa que devemos considerar é a contrapressão. Em nosso exemplo, o assinante está dizendo ao produtor para enviar todos os elementos de uma vez. Isso pode acabar se tornando opressor para o assinante, consumindo todos os seus recursos.

Contrapressão é quando um downstream pode dizer a um upstream para enviar menos dados para evitar que ele seja sobrecarregado .

Podemos modificar nossa implementação do Assinante para aplicar contrapressão. Vamos dizer ao upstream para enviar apenas dois elementos por vez usando 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() {}
});
```

Agora, se executarmos nosso código novamente, veremos que *request(2)* é chamado, seguido por duas chamadas *onNext() e, em seguida, request(2)* novamente.

```
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()
```

Essencialmente, esta é a contrapressão de tração reativa. Estamos solicitando ao upstream que empurre apenas uma certa quantidade de elementos e somente quando estivermos prontos.
Se imaginarmos que estamos recebendo tweets transmitidos do Twitter, caberá ao upstream decidir o que fazer. Se os tweets estivessem chegando, mas não houvesse solicitações do downstream, o upstream poderia descartar itens, armazená-los em um buffer ou alguma outra estratégia.

Mapeando dados em um fluxo:

Uma operação simples que podemos realizar é aplicar uma transformação. Neste caso, vamos apenas dobrar todos os números em nosso stream:

```
Flux.just(1, 2, 3, 4)
  .log()
  .map(i -> i * 2)
  .subscribe(elements::add);
```

map() será aplicado quando *onNext()* for chamado.

Combinando Dois Fluxos:

Podemos então tornar as coisas mais interessantes combinando outro fluxo com este. Vamos tentar isso usando a função *zip():*

```
Flux.just(1, 2, 3, 4)
  .log()
  .map(i -> i * 2)
  .zipWith(Flux.range(0, Integer.MAX_VALUE),
    (one, two) -> String.format("First Flux: %d, Second Flux: %d", one, two))
  .subscribe(elements::add);
assertThat(elements).containsExactly(
  "First Flux: 2, Second Flux: 0",
  "First Flux: 4, Second Flux: 1",
  "First Flux: 6, Second Flux: 2",
  "First Flux: 8, Second Flux: 3");
```

Aqui, estamos criando outro *Flux* que continua incrementando em um e transmitindo-o junto com o original. Podemos ver como eles funcionam juntos inspecionando os logs:

```
20:04:38.064 [main] INFO  reactor.Flux.Array.1 - | onSubscribe([Synchronous Fuseable] FluxArray.ArraySubscription)
20:04:38.065 [main] INFO  reactor.Flux.Array.1 - | onNext(1)
20:04:38.066 [main] INFO  reactor.Flux.Range.2 - | onSubscribe([Synchronous Fuseable] FluxRange.RangeSubscription)
```

```
20:04:38.066 [main] INFO  reactor.Flux.Range.2 - | onNext(0)
20:04:38.067 [main] INFO  reactor.Flux.Array.1 - | onNext(2)
20:04:38.067 [main] INFO  reactor.Flux.Range.2 - | onNext(1)
20:04:38.067 [main] INFO  reactor.Flux.Array.1 - | onNext(3)
20:04:38.067 [main] INFO  reactor.Flux.Range.2 - | onNext(2)
20:04:38.067 [main] INFO  reactor.Flux.Array.1 - | onNext(4)
20:04:38.067 [main] INFO  reactor.Flux.Range.2 - | onNext(3)
20:04:38.067 [main] INFO  reactor.Flux.Array.1 - | onComplete()
20:04:38.067 [main] INFO  reactor.Flux.Array.1 - | cancel()
20:04:38.067 [main] INFO  reactor.Flux.Range.2 - | cancel()
```

Observe como agora temos uma assinatura por *Flux* . As chamadas *onNext()* também são alternadas, de modo que o índice de cada elemento no fluxo corresponderá quando aplicarmos a função *zip()*.

Fluxos Quentes (Hot Streams):

Atualmente, nos concentramos principalmente em correntes frias. Esses são fluxos estáticos e de comprimento fixo fáceis de lidar. Um caso de uso mais realista para reativo pode ser algo que acontece infinitamente.

Por exemplo, poderíamos ter um fluxo de movimentos do mouse que precisa ser constantemente reagido ou um feed do Twitter. Esses tipos de fluxos são chamados de fluxos quentes, pois estão sempre em execução e podem ser assinados a qualquer momento, perdendo o início dos dados.

Criando um *ConnectableFlux:*

Uma maneira de criar um fluxo quente é convertendo um fluxo frio em um. Vamos criar um *Flux* que dure para sempre, enviando os resultados para o console, que simularia um fluxo infinito de dados vindo de um recurso externo:

```
ConnectableFlux<Object> publish = Flux.create(fluxSink -> {
    while(true) {
        fluxSink.next(System.currentTimeMillis());
    }
})
  .publish();
```

Ao chamar *publish(),* recebemos um *ConnectableFlux* . Isso significa que chamar *Subscribe()* não fará com que ele comece a emitir, permitindo-nos adicionar várias assinaturas:

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

Se tentarmos executar este código, nada acontecerá. Não é até chamarmos *connect(),* que o *Flux* começará a emitir:

```
publish.connect();
```

Throttling:

Se executarmos nosso código, nosso console ficará sobrecarregado com o registro. Isso está simulando uma situação em que muitos dados estão sendo passados para nossos consumidores. Vamos tentar contornar isso com limitação:

```
ConnectableFlux<Object> publish = Flux.create(fluxSink -> {
    while(true) {
        fluxSink.next(System.currentTimeMillis());
    }
})
  .sample(ofSeconds(2))
  .publish();
```

Aqui, introduzimos um método *sample()* com um intervalo de dois segundos. Agora os valores só serão enviados para nosso assinante a cada dois segundos, o que significa que o console ficará muito menos agitado.

É claro que existem várias estratégias para reduzir a quantidade de dados enviados downstream, como janelamento e buffering, mas elas ficarão fora do escopo deste artigo.

Concurrency (Simultaneidade):

Todos os nossos exemplos acima foram executados no thread principal. No entanto, podemos controlar em qual thread nosso código é executado, se quisermos. A interface *do Scheduler* fornece uma abstração em torno do código assíncrono, para o qual muitas implementações são fornecidas para nós. Vamos tentar assinar um tópico diferente do principal:

```
Flux.just(1, 2, 3, 4)
  .log()
  .map(i -> i * 2)
  .subscribeOn(Schedulers.parallel())
  .subscribe(elements::add);
```

O agendador *Parallel* fará com que nossa assinatura seja executada em um thread diferente, o que podemos provar observando os logs. Vemos que a primeira entrada vem do thread *principal* e o Flux está sendo executado em outro thread chamado *parallel-1*.

```
20:03:27.505 [main] DEBUG reactor.util.Loggers$LoggerFactory - Using Slf4j logging framework
20:03:27.529 [parallel-1] INFO  reactor.Flux.Array.1 - | onSubscribe([Synchronous Fuseable] FluxArray.ArraySubscription)
20:03:27.531 [parallel-1] INFO  reactor.Flux.Array.1 - | request(unbounded)
20:03:27.531 [parallel-1] INFO  reactor.Flux.Array.1 - | onNext(1)
20:03:27.531 [parallel-1] INFO  reactor.Flux.Array.1 - | onNext(2)
20:03:27.531 [parallel-1] INFO  reactor.Flux.Array.1 - | onNext(3)
20:03:27.531 [parallel-1] INFO  reactor.Flux.Array.1 - | onNext(4)
20:03:27.531 [parallel-1] INFO  reactor.Flux.Array.1 - | onComplete()
```

A simultaneidade é mais interessante do que isso e vale a pena explorá-la em outro artigo.