Chapitre 3
Reactor Core

1. Introduction

Le principal cas d'usage de l'utilisation de Reactor Core est la création d'une application web, intranet ou extranet, avec Spring WebFlux, le pendant de Spring MVC en mode réactif.

Cependant, il peut être intéressant dans certains cas d'utiliser directement Reactor Core pour :

– programmer un serveur comme nous le ferions avec Vert.x
– programmer plusieurs serveurs web dans une même application
– programmer des serveurs qui ne servent pas des pages HTML via HTTP
– avoir des serveurs de mocks pour les tests d'intégration

Nous allons aborder dans ce chapitre des éléments plutôt théoriques, car la librairie Spring Reactor Core est rarement utilisée seule pour plusieurs raisons :

– Complexité : Spring Reactor Core est une bibliothèque puissante mais complexe qui implémente le paradigme de programmation réactive. Utiliser Reactor Core seule peut demander une courbe d'apprentissage assez raide, car elle nécessite une compréhension approfondie des concepts de flux, de mono et de l'opérateur de composition fonctionnelle.

– Manque de facilité d'intégration : Spring Reactor Core fournit principalement une base pour la programmation réactive et ne propose pas directement des fonctionnalités spécifiques aux applications, telles que le traitement des requêtes HTTP, la gestion des dépendances, l'injection de dépendances, etc. Pour bénéficier d'une application web réactive complète, les développeurs préfèrent utiliser les autres modules de Spring WebFlux, qui offrent une intégration facile avec Spring Boot et le reste de l'écosystème Spring.
– Présence de Spring WebFlux : Spring WebFlux est une extension de Spring qui offre un support complet pour la programmation réactive. Il utilise Spring Reactor Core sous le capot et fournit une API plus simple et plus intuitive pour développer des applications web réactives.
– Les développeurs préfèrent généralement utiliser Spring WebFlux, car il offre une solution plus complète et simplifiée pour développer des applications web réactives avec l'écosystème Spring. Cependant, il est important de bien comprendre son fonctionnement pour en tirer pleinement parti, notamment pour utiliser les extensions Netty, RabbitMQ et Kafka de Reactor.

Reactor est une bibliothèque de programmation réactive entièrement non bloquante pour la JVM, offrant une gestion efficace de la demande grâce à la contre-pression. Elle s'intègre parfaitement aux API fonctionnelles de Java 8, telles que `CompletableFuture`, `Stream` et `Duration`. Cette bibliothèque propose des API de séquences asynchrones et composables : `Flux` (pour les éléments [N]) et `Mono` (pour les éléments [0 | 1]), et elle met en œuvre de manière approfondie la spécification `Reactive Streams`.

La prise en charge de Reactor Core débute à partir de la version 8 de java. Dans nos exemples, nous utilisons Spring 6 qui nécessite au moins Java 17. Reactor Core lui-même dépend de `org.reactivestreams:reactive-streams:1.0.3` pour son fonctionnement.

Pour l'utiliser, nous aurons recours à une nomenclature (*Bill of Materials*, BOM) :

```
<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>io.projectreactor</groupId>
            <artifactId>reactor-bom</artifactId>
            <version>2022.0.9</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
    </dependencies>
</dependencyManagement>
<dependencies>
    <dependency>
        <groupId>io.projectreactor</groupId>
        <artifactId>reactor-core</artifactId>
    </dependency>
    <dependency>
        <groupId>io.projectreactor</groupId>
        <artifactId>reactor-test</artifactId>
        <scope>test</scope>
    </dependency>
</dependencies>
```

2. Reactor Core

La programmation réactive est un concept spécial. Elle implique la propagation d'événements ou de signaux, comme des boules de flipper interagissant les unes avec les autres. En utilisant un style de programmation fonctionnelle, nous définissons les parcours possibles des boules, puis les lançons en parallèle pour qu'elles se déplacent simultanément. Chaque boule change de comportement lorsqu'elle interagit ou entre en collision avec d'autres éléments, ou lorsqu'elle atteint un point spécifique dans le temps. Le rôle du moteur est de gérer les interactions en appelant le code approprié pour chaque événement. Il les met en file d'attente dans une boucle d'événement et les exécute séquentiellement, ce qui donne l'impression d'une exécution parallèle depuis l'extérieur.

Les briques de base des Reactive Streams sont :

– `Subscriber` : l'abonné (*subscriber*) est l'objet qui reçoit les éléments émis par l'éditeur *(publisher)*. Il consomme les éléments et les traite selon la logique définie dans ses méthodes. Lorsqu'un `Publisher` émet un nouvel élément, il est transmis au `Subscriber` via la méthode `onNext()`. Le Subscriber peut également gérer les erreurs en utilisant la méthode `onError()` et indiquer la fin du flux de données avec la méthode `onComplete()`.
– `Subscription` : l'abonnement (*subscription*) entre le `Publisher` et le `Subscriber`. Lorsqu'un `Subscriber` s'abonne à un `Publisher`, le `Publisher` envoie une `Subscription` au `Subscriber`. Cette `Subscription` permet au `Subscriber` de demander un certain nombre d'éléments au `Publisher` en utilisant la méthode `request(long n)`. Cette méthode spécifie combien d'éléments le `Subscriber` souhaite recevoir.
– `Processor` : le processeur est une interface qui combine les rôles de `Publisher` et de `Subscriber`. Il peut être vu comme un élément intermédiaire dans le flux de données, où les éléments sont émis par un `Publisher`, traités par le `Processor`, puis consommés par un `Subscriber`. Il permet de créer des opérations de transformation ou de filtrage des données entre le `Publisher` et le `Subscriber`.

Ces trois briques — `Publisher`, `Subscriber` et `Subscription` — forment la base des Reactive Streams et permettent de créer des flux de données réactifs asynchrones et non bloquants. L'interface `Processor`, quant à elle, permet de réaliser des opérations plus complexes et de gérer la transformation des données entre les différentes étapes du flux de données. Ensemble, ces modules permettent une gestion efficace des flux de données réactifs dans un environnement réactif.

Ces briques sont habillées par différents frameworks qui vont privilégier certains aspects.

Spring a choisi d'introduire deux types réactifs différenciants et composables, qui implémentent l'interface `Publisher : Mono` et `Flux`. Ces types sont comparables à la classe `Observable` de RxJava et RxJS, mais avec des spécificités propres. Ils ont été créés de manière à répondre à des besoins différents en fonction de leur cardinalité et de leurs méthodes spécifiques :

– `Flux` est utilisé pour représenter un flux de données asynchrone avec un nombre d'éléments allant de 0 à n. Il permet de gérer une séquence de zéro, un ou plusieurs éléments.
– `Mono`, quant à lui, est conçu pour gérer une séquence de zéro ou un seul élément. Il représente un flux asynchrone avec une cardinalité allant de 0 à 1.

Ces types, réactifs, intègrent nativement les fonctionnalités de gestion des erreurs, de la synchronisation et de la backpressure, ce qui les rend naturellement adaptés aux cas d'utilisation réactifs.

En programmation réactive, nous rencontrons deux types de traitements :

– Des traitements classiques tels que des calculs ou des transtypages, qui sont généralement instantanés et ne nécessitent pas de flux de données asynchrones.
– Des traitements sur des flux, qui prennent un flux (ou plusieurs flux) en entrée et retournent un flux (ou plusieurs flux) en sortie.

Les flux en entrée peuvent être des `Mono` ou des `Flux`. Ceux-ci sont tous deux des implémentations de l'interface `Publisher`. Les objets `Publisher` réagissent à des stimuli en déclenchant de façon automatique des traitements sous la forme d'un signal. Une fois à l'intérieur d'un de ces traitements, ils peuvent déclencher d'autres traitements et/ou émettre à leur tour de nouveaux éléments de flux.

Ces traitements sur des flux permettent de manipuler les données de manière réactive et asynchrone, en autorisant l'émission des éléments du flux un par un, au fur et à mesure de leur disponibilité, sans attendre que l'ensemble du flux soit disponible. Cela améliore l'efficacité et la performance dans les cas où les données sont produites ou consommées de manière asynchrone.

Pour le `Mono` :

Signal	Signal
`onNext`	L'élément qu'on attendait est arrivé et il n'y en aura pas d'autre.
`onComplete`	On a reçu l'élément et cela s'est passé sans erreur.
`onError`	Il y a eu une erreur.

Pour le `Flux` :

Signal	Signification
`onNext`	Un des éléments qu'on attendait est arrivé et il y en aura peut-être d'autres.
`onComplete`	On a reçu tous les éléments attendus.
`onError`	Il y a eu une erreur. Le flux est interrompu.

Pour les exemples suivants, nous allons mixer la programmation classique avec la programmation réactive. En programmation réactive, nous pouvons combiner la programmation réactive et la programmation impérative en utilisant les méthodes `just()`, `block()` ainsi que d'autres méthodes fournies par Reactor.

– Méthode `just()` : la méthode `just()` permet de créer un flux réactif contenant une ou plusieurs valeurs données en argument. Par exemple, `Flux.just(1, 2, 3)` crée un flux contenant les valeurs 1, 2 et 3.

– Méthode `block()` : la méthode `block()` est une méthode bloquante qui permet de récupérer la valeur émise par un flux réactif. Elle est principalement utilisée dans le contexte de tests ou dans la méthode `main()` d'une application pour obtenir le résultat d'un flux avant que l'application ne se termine.

Exemple d'utilisation de `just()` et `block()` :

```
public static void main(String[] args) {
    // Création d'un flux réactif avec la méthode just()
    Flux<Integer> flux = Flux.just(1, 2, 3, 4, 5);

    // Récupération de la valeur émise par le flux
    // avec la méthode block()
    int result = flux.blockFirst();

    // Affichage du résultat
    System.out.println("Résultat : " + result);
}
```

Dans cet exemple, nous créons un flux réactif avec la méthode `just()` contenant les valeurs 1, 2, 3, 4 et 5. Ensuite, nous utilisons la méthode `blockFirst()` pour bloquer le thread courant et attendre que le flux émette sa première valeur. Une fois que la valeur est émise, nous la récupérons dans la variable `result` et l'affichons à l'écran.

Il est important de noter que l'utilisation de `block()` dans une application réelle est déconseillée, car elle bloque le thread courant, ce qui peut entraîner des problèmes de performances et de scalabilité. Dans une application réelle, nous préférons généralement utiliser des opérations asynchrones pour traiter les flux réactifs sans bloquer les threads. Cependant, dans certains cas de tests unitaires ou d'utilisation spécifiques, l'utilisation de `block()` peut être utile.

Il existe d'autres méthodes :

– `fromIterable()` : la méthode `fromIterable()` permet de créer un flux réactif à partir d'une collection ou d'une liste d'éléments. Elle est utile lorsque nous voulons émettre une séquence de valeurs à partir d'une collection existante.

```
List<Integer> list = Arrays.asList(1, 2, 3, 4, 5);
Flux<Integer> flux = Flux.fromIterable(list);
```

– `fromStream()` : la méthode `fromStream()` permet de créer un flux réactif à partir d'un flux Java Stream. Cela peut être utile lorsque nous voulons combiner des opérations réactives avec des opérations impératives sur des collections.

```
Stream<Integer> stream = Stream.of(1, 2, 3, 4, 5);
Flux<Integer> flux = Flux.fromStream(stream);
```

– `blockFirst()` : la méthode `blockFirst()` est une méthode bloquante qui permet de récupérer la première valeur émise par un flux réactif. Elle est souvent utilisée dans les tests unitaires ou dans la méthode `main()` d'une application pour obtenir rapidement le résultat d'un flux.

```
Flux<Integer> flux = Flux.just(1, 2, 3, 4, 5);
int result = flux.blockFirst();
System.out.println("Première valeur : " + result);
```

– `blockLast()` : la méthode `blockLast()` est similaire à `blockFirst()`, mais elle permet de récupérer la dernière valeur émise par un flux réactif.

```
Flux<Integer> flux = Flux.just(1, 2, 3, 4, 5);
int result = flux.blockLast();
System.out.println("Dernière valeur : " + result);
```

– `toIterable()` : la méthode `toIterable()` permet de convertir un flux réactif en itérable. Cela peut être utile lorsque nous voulons utiliser des méthodes impératives qui attendent un itérable en entrée.

```
Flux<Integer> flux = Flux.just(1, 2, 3, 4, 5);
Iterable<Integer> iterable = flux.toIterable();
```

– `collectList()` : la méthode `collectList()` permet de collecter toutes les valeurs émises par un flux réactif dans une liste. Cela peut être utile pour un traitement ultérieur de ces valeurs.

```
Flux<Integer> flux = Flux.just(1, 2, 3, 4, 5);
Mono<List<Integer>> listMono = flux.collectList();
```