Chapitre 5
Programmation réactive

1. Flux réactifs (reactive streams)

Ce chapitre décrit les principaux aspects de la programmation réactive qui est à la base des nouveaux paradigmes réactifs de programmation. Il présente les implémentations réactives Akka, RxJava, RxJava 2, Reactor et Java 9.

Les architectures « réactives » visent à calquer le modèle des architectures basées sur des serveurs de messages à l'échelle d'un projet avec des streams en mémoire.

Dans un premier temps, Microsoft a créé la bibliothèque Reactive Extension (Rx) pour le framework .NET. La librairie RxJava est arrivée ensuite avec RxJS, suivi de RxJava, et a été utilisée sous Android avec RxAndroid.

Une normalisation a émergé au fil des années à travers l'initiative Reactive Streams, qui définit une spécification de l'ensemble des interfaces et des règles d'interaction pour les bibliothèques réactives.

Les librairies Reactive Stream permettent de faire des programmes asynchrones basés sur des événements via l'utilisation de séquences observables. Elles étendent le patron de conception (*design pattern*) observateur (*observer*).

Les observateurs reçoivent des messages des modules qu'ils observent. Cela permet de coupler les modules afin de réduire les dépendances aux seuls modules observés.

Comme nous l'avons évoqué, les paradigmes réactifs viennent de l'initiative Reactive Streams (http://www.reactive-streams.org/) qui a pour objectif de mettre au point une norme pour le traitement des flux non bloquants asynchrones avec contre-pression pour les environnements d'exécution JVM et JavaScript et pour les échanges via le réseau. L'initiative Reactive Streams se base sur le manifeste « Reactive Manifesto », libre de droits, publié en septembre 2014, qui est disponible sur : https://www.reactivemanifesto.org/fr.

Cet ouvrage se concentre sur la version JVM des implémentations réactives RxJava et Reactor. Pour cette plateforme, le projet Reactive Streams met à disposition des API qui sont ensuite surchargées pour les implémentations RxJava 2 et Reactor.

RxJava 1.x est sorti avant la mise au point de l'initiative Reactive Streams. Il n'en implémente donc pas tous les aspects, mais il en est très proche. Étant plus ancien, il est aussi beaucoup plus riche en fonctionnalités, qui ne sont pas encore disponibles avec RxJava 2.

Pour la partie JVM, le dépôt Git est accessible sur : https://github.com/reactive-streams/reactive-streams-jvm. Cette partie est composée d'une API très simple et d'un kit de test de compatibilité : le TCK (*Technology Compatibility Kit*).

Il n'est pas possible d'utiliser Reactive Streams seul, on utilise plutôt les implémentations. Par contre, si l'on veut étendre les opérateurs de RxJava ou de Reactor, il faut être compatible avec Reactive Streams et utiliser le TCK.

RxJava est l'implémentation pour la JVM de la bibliothèque ReactiveX, qui permet de faire de la programmation asynchrone avec différents langages, plateformes et frameworks. Elle n'implémente pas directement les API Reactive Streams, car RxJava 1.x existait déjà et il n'était pas possible de casser les API publiques existantes.

Le module `RxJavaReactiveStreams` (https://github.com/ReactiveX/RxJavaReactiveStreams) permet de faire un pont entre les deux API, avec une conversion de types entre `Publisher` et `Observable` que nous détaillerons par la suite.

La récente sortie de RxJava 2.x et de Reactor a inspiré la création de la nouvelle API du JDK 9 (java.util.concurrent.Flow) qui reprend point par point les spécifications de son homologue Reactive Streams. L'interopérabilité permet d'utiliser conjointement les différentes implémentations avec des convertisseurs pour passer d'une implémentation à une autre. Il est donc possible de mixer les librairies dans un même projet. L'étude de RxJava 1.x et de RxJava 2.x permet d'avoir une vision de l'implémentation Reactive Streams, mais aussi de mixer du RxJava avec Reactor pour les fonctionnalités qui ne sont que dans une des deux implémentations.

Historiquement, Spring 4.3 a été développé pour être compatible avec Java 6. Il n'est pas possible d'utiliser Spring 5 avec des versions Java antérieures à la version 8.

Les utilisateurs de RxJava 1.x peuvent choisir de passer en 2.x, de migrer sur Reactor ou de faire des applications mixtes avec des conversions de types, notamment pour migrer progressivement.

Certains frameworks sont couplés avec RxJava, comme Vert.x, Akka et Reactor.

L'API repose sur quatre classes : `Processor`, `Publisher`, `Subscriber` et `Subscription`.

Interfaces réactives :

```
//https://github.com/reactive-streams/reactive-streams-jvm
public interface Publisher<T> {
public void subscribe(Subscriber<? super T> s);
}

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

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

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

Un ensemble d'exemples d'implémentations minimales des interfaces des API est mis à disposition dans un des exemples téléchargeables sur le site des Éditions ENI.

2. Programmation asynchrone historique

Il existe principalement trois méthodes pour faire de la programmation asynchrone sous Java :

Méthode	Explication
`Callback`	Les méthodes asynchrones n'ont pas de valeur de retour, mais prennent un paramètre supplémentaire de callback (une expression Lambda ou une classe anonyme) qui est appelé lorsque le résultat est disponible. Un exemple bien connu est la hiérarchie `EventListener` de Swing.
`Future`	Les méthodes asynchrones retournent immédiatement un `Future<T>`. Le processus asynchrone calcule une valeur T, mais l'objet `Future` masque son accès. La valeur n'est pas immédiatement disponible et l'objet peut être interrogé jusqu'à ce que la valeur soit disponible. Par exemple, les tâches `ExecutorService` `Callable<T>` en cours d'exécution utilisent des objets `Future`.
`CompletableFuture`	Comme `Future`, avec des comportements supplémentaires.

Ces implémentations sont basées sur du langage "impératif" et utilisent des callbacks pour gérer les rappels en fin de traitement. Si nous utilisons ce type de programmation pour faire, par exemple, de l'orchestration d'appels de services web en parallèle avec de la logique métier pour conditionner les différents appels, nous sommes très rapidement confrontés à du code très complexe à déboguer et à maintenir. Pour pallier cet enfer des callbacks, la solution basée sur des API réactives encapsule et masque les callbacks en fournissant une API plus claire.

3. API réactives

Il existe aujourd'hui un certain nombre d'implémentations d'API réactives, issues de plusieurs générations.

API	Génération	Basée sur
Akka Streams 2.x	4e	Reactive Streams
RxJava 1.x	2e	Précurseur
Reactor 3.0	3e	Reactive API

Nous allons examiner, à travers des exemples, ces différentes implémentations afin de voir l'origine de certains concepts et leurs réutilisations.

3.1 Akka

Nous commencerons par Akka en présentant dans un premier temps un peu de théorie sur le framework, puis des exemples d'utilisation avec Spring. Nous ne traitons, ici, qu'une partie d'Akka car ce framework est très vaste.

Akka est une bibliothèque open source qui permet de développer facilement des applications concurrentes et distribuées. La bibliothèque est écrite en Scala et peut être utilisée en Java. Nous la rencontrons plus souvent dans l'écosystème Scala, qui est en avance sur Java pour la programmation fonctionnelle. Alors qu'elles ont été introduites récemment dans Java, les lambdas existent depuis longtemps dans Scala. Akka exploite le modèle d'acteur.

Sa force réside dans une gestion très propre des échanges asynchrones entre objets qui nous libère des contraintes liées aux threads et à la gestion des accès concurrents sur les données en mémoire. Nous expliquons dans ce qui suit ce que sont les acteurs, comment ils communiquent et comment nous pouvons les libérer. Nous introduisons quelques bonnes pratiques pour le travail avec Akka.

3.2 Modèle d'acteur

Le modèle d'acteur n'est pas nouveau. Il a été introduit par Carl Eddie Hewitt en 1973. Il est basé sur un modèle théorique pour le traitement des calculs concurrents. Il trouve son origine dans l'émergence des applications concurrentes et distribuées.

Un acteur représente une unité de calcul indépendante. Ces principales caractéristiques sont les suivantes : un acteur encapsule son état et une partie de la logique de l'application. Les acteurs interagissent uniquement via des messages asynchrones, et jamais via des appels de méthodes directs. Un acteur a une adresse unique et une file de messages propre dans laquelle les autres acteurs peuvent livrer des messages. Un acteur traite tous les messages de sa file dans un ordre séquentiel, qui est par défaut FIFO. Le système d'acteurs est organisé dans une hiérarchie arborescente. Un acteur peut créer d'autres acteurs enfants. Il peut envoyer des messages à tout autre acteur et s'arrêter et arrêter ses acteurs enfants.

Finalités du modèle d'acteur

Comme nous l'avons déjà évoqué, la difficulté des applications modernes est intimement liée aux aspects concurrents et asynchrones. Cela induit des verrous sur la mémoire pour gérer les accès concurrents sur la mémoire partagée et dans les systèmes de persistance des données, et des points de synchronisation entre les processus. Par ailleurs, le switch de contexte est coûteux, car il faut sauvegarder l'état des processus avant de les switcher. Avec les processeurs modernes, il faut gérer correctement les caches des processeurs et cela aussi est coûteux.