1

Inversion de contrôle et conteneurs

Au sommaire de ce chapitre

- Utiliser un conteneur pour gérer des composants
- Utiliser un localisateur de service pour simplifier la recherche
- Appliquer l'inversion de contrôle et l'injection de dépendance
- Comprendre les différents types d'injections de dépendance
- Configurer un conteneur à partir d'un fichier de configuration
- En résumé

Dans ce chapitre, vous allez faire connaissance avec le principe de conception *Inversion de contrôle* (IoC), que de nombreux conteneurs modernes emploient pour découpler les dépendances entre composants. Le framework Spring fournit un conteneur IoC puissant et extensible pour la gestion des composants. Il se situe au cœur du framework et s'intègre parfaitement aux autres modules de Spring. L'objectif de ce chapitre est de vous apporter les connaissances de base qui vous permettront de démarrer avec Spring.

Pour la plupart des programmeurs, dans le contexte d'une plate-forme Java EE, les composants sont des EJB (*Enterprise JavaBean*). La spécification des EJB définit clairement le contrat qui lie les composants et les conteneurs EJB. En s'exécutant dans un conteneur EJB, les composants EJB bénéficient des services de gestion du cycle de vie, de gestion des transactions et de sécurité. Toutefois, dans les versions antérieures à la version 3.0 des EJB, un seul composant EJB exige une interface Remote/Local, une interface Home et une classe d'implémentation du bean. En raison de leur complexité, ces EJB sont appelés composants lourds.

Par ailleurs, dans ces versions des EJB, un composant EJB peut s'exécuter uniquement au sein d'un conteneur EJB et doit rechercher les autres EJB à l'aide de JNDI (*Java*

Naming and Directory Interface). Les composants EJB sont dépendants de la technologie et ne peuvent pas être réutilisés ou testés en dehors d'un conteneur EJB.

De nombreux conteneurs légers sont conçus pour pallier les défauts des EJB. Ils sont légers car les objets Java simples peuvent être employés comme des composants. Toutefois, ces conteneurs posent leur propre défi : comment découpler les dépendances entre les composants. L'IoC est une solution efficace à ce problème.

Alors que l'IoC est un principe général de conception, l'injection de dépendance (DI, *Dependency Injection*) est un design pattern concret qui incarne ce principe. Puisque l'injection de dépendance constitue la mise en œuvre type de l'inversion de contrôle, si ce n'est la seule, les termes IoC et DI sont fréquemment employés de manière interchangeable.

À la fin de ce chapitre, vous serez capable d'écrire un conteneur IoC simple qui sera conceptuellement équivalent au conteneur Spring IoC. Si l'inversion de contrôle n'a pas de secret pour vous, n'hésitez pas à passer directement au Chapitre 2, qui présente l'architecture globale et la configuration du framework Spring.

1.1 Utiliser un conteneur pour gérer des composants

Problème

Derrière la conception orientée objet se cache l'idée de décomposer le système en un ensemble d'objets réutilisables. Sans un module central de gestion des objets, ces derniers doivent créer et gérer leurs propres dépendances. Par conséquent, ils sont fortement couplés.

Solution

Nous avons besoin d'un *conteneur* de gestion des objets qui composent notre système. Un conteneur centralise la création des objets et joue le rôle de registre pour les services de recherche. Un conteneur prend également en charge le cycle de vie des objets et leur fournit une plate-forme d'exécution.

Les objets qui s'exécutent à l'intérieur d'un conteneur sont appelés *composants*. Ils doivent se conformer aux spécifications définies par le conteneur.

Explications

Séparer l'interface de son implémentation

Supposons que notre objectif soit de développer un système qui génère différents types de rapports au format HTML ou PDF. Conformément au principe de "séparation de

l'interface de l'implémentation" qui prévaut dans une conception orientée objet, nous devons créer une interface commune pour la génération des rapports. Supposons que le contenu d'un rapport soit fourni par une table d'enregistrements donnée sous la forme d'un tableau de chaînes de caractères à deux dimensions.

```
package com.apress.springrecipes.report;

public interface ReportGenerator {

    public void generate(String[][] table);
}
```

Nous créons alors deux classes, `HtmlReportGenerator` et `PdfReportGenerator`, qui implémentent cette interface pour les rapports HTML et les rapports PDF. Dans le cadre de cet exemple, le squelette des méthodes suffit.

```
package com.apress.springrecipes.report;

public class HtmlReportGenerator implements ReportGenerator {

    public void generate(String[][] table) {
        System.out.println("Génération d'un rapport HTML...");
    }
}

---

package com.apress.springrecipes.report;

public class PdfReportGenerator implements ReportGenerator {

    public void generate(String[][] table) {
        System.out.println("Génération d'un rapport PDF...");
    }
}
```

Les instructions `println` présentes dans le corps des méthodes nous permettent d'être informés de l'exécution de chacune d'elles.

Les classes des générateurs de rapports étant prêtes, nous pouvons débuter la création de la classe `ReportService`, qui joue le rôle de fournisseur de service pour la génération des différents types de rapports. Ses méthodes, comme `generateAnnualReport()`, `generateMonthlyReport()` et `generateDailyReport()`, permettent de générer des rapports fondés sur les données des différentes périodes.

```
package com.apress.springrecipes.report;

public class ReportService {

    private ReportGenerator reportGenerator = new PdfReportGenerator();

    public void generateAnnualReport(int year) {
        String[][] statistics = null;
```

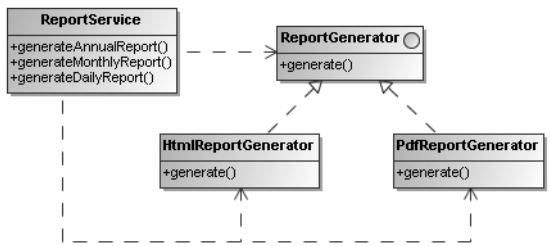
```
        //
        // Collecter les statistiques pour l'année...
        //
        reportGenerator.generate(statistics);
    }

    public void generateMonthlyReport(int year, int month) {
        String[][] statistics = null;
        //
        // Collecter les statistiques pour le mois...
        //
        reportGenerator.generate(statistics);
    }

    public void generateDailyReport(int year, int month, int day) {
        String[][] statistics = null;
        //
        // Collecter les statistiques pour la journée...
        //
        reportGenerator.generate(statistics);
    }
}
```

Puisque la logique de génération d'un rapport est déjà implémentée dans les classes des générateurs, nous pouvons créer une instance de l'une de ces classes dans une variable privée et l'invoquer dès que nous devons générer un rapport. Le format de sortie du rapport dépend de la classe instanciée.

Le diagramme de classes UML de la Figure 1.1 présente les dépendances entre `ReportService` et les différentes implémentations de `ReportGenerator`.

Figure 1.1

Dépendances entre `ReportService` et les différentes implémentations de `ReportGenerator`.



Pour le moment, `ReportService` crée l'instance de `ReportGenerator` de manière interne et doit donc savoir quelle classe concrète de `ReportGenerator` utiliser. Cela crée une dépendance directe entre `ReportService` et l'une des implémentations de `ReportGenerator`. Par la suite, nous supprimerons totalement les lignes de dépendance avec les implémentations de `ReportGenerator`.

Employer un conteneur

Supposons que notre système de génération de rapports soit destiné à plusieurs entreprises. Certains des utilisateurs préféreront les rapports au format HTML, tandis que d'autres opteront pour le format PDF. Nous devons gérer deux versions différentes de `ReportService` pour les deux formats de rapport. L'une crée une instance de `HtmlReportGenerator`, l'autre, une instance de `PdfReportGenerator`.

Si cette conception n'est pas vraiment souple, c'est parce que nous avons créé l'instance de `ReportGenerator` directement à l'intérieur de `ReportService` et, par conséquent, que cette classe doit savoir quelle implémentation de `ReportGenerator` utiliser. Rappelez-vous les lignes de dépendance partant de `ReportService` vers `HtmlReportGenerator` et `PdfReportGenerator` dans le diagramme de classe de la Figure 1.1. Tout changement d'implémentation du générateur de rapports conduit à une modification de `ReportService`.

Pour résoudre ce problème, nous avons besoin d'un conteneur qui gère les composants du système. Un conteneur complet est extrêmement complexe, mais nous pouvons commencer par en créer une version très simple :

```
package com.apress.springrecipes.report;
...
public class Container {

    // L'instance globale de cette classe Container afin que les composants
    // puissent la trouver.
    public static Container instance;

    // Un Map pour stocker les composants. L'identifiant du composant
    // sert de clé.
    private Map<String, Object> components;

    public Container() {
        components = new HashMap<String, Object>();
        instance = this;

        ReportGenerator reportGenerator = new PdfReportGenerator();
        components.put("reportGenerator", reportGenerator);

        ReportService reportService = new ReportService();
        components.put("reportService", reportService);
    }

    public Object getComponent(String id) {
        return components.get(id);
    }
}
```

Dans cet exemple de conteneur, un `Map` enregistre les composants, dont les identifiants servent de clés. Le constructeur du conteneur initialise les composants et les place dans

le `Map`. Pour le moment, il n'existe que deux composants dans notre système : `ReportGenerator` et `ReportService`. La méthode `getComponent()` retrouve un composant à partir de l'identifiant indiqué. La variable statique publique `instance` contient l'instance globale de cette classe `Container`. Les composants peuvent ainsi retrouver ce conteneur et rechercher d'autres composants.

Grâce au conteneur qui gère nos composants, nous pouvons remplacer la création de l'instance de `ReportGenerator` dans `ReportService` par une instruction de recherche d'un composant.

```
package com.apress.springrecipes.report;

public class ReportService {

    private ReportGenerator reportGenerator =
        (ReportGenerator) Container.instance.getComponent("reportGenerator");

    public void generateAnnualReport(int year) {
        ...
    }

    public void generateMonthlyReport(int year, int month) {
        ...
    }

    public void generateDailyReport(int year, int month, int day) {
        ...
    }
}
```

Ainsi, `ReportService` ne choisit plus l'implémentation de `ReportGenerator` qu'elle doit employer. Il n'est donc plus nécessaire de modifier `ReportService` lorsque nous voulons changer d'implémentation du générateur de rapports.

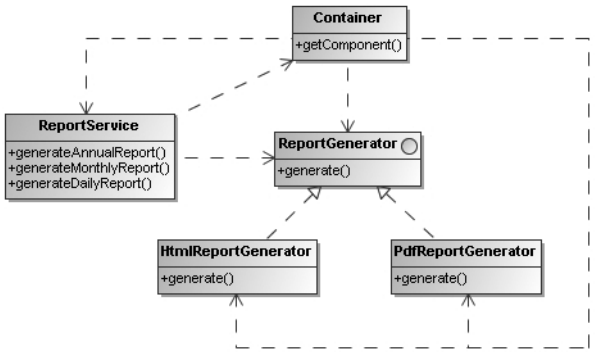
En recherchant un générateur de rapports au travers du conteneur, nous améliorons la réutilisabilité de `ReportService` car cette classe n'a plus de dépendance directe avec une implémentation de `ReportGenerator`. Nous pouvons configurer et déployer différents conteneurs pour les diverses entreprises sans modifier `ReportService`.

La Figure 1.2 présente le diagramme de classes UML lorsque les composants sont gérés par un conteneur.

La classe centrale `Container` présente des dépendances avec tous les composants sous sa responsabilité. Les dépendances entre `ReportService` et les deux implémentations de `ReportGenerator` ont été supprimées. Elles sont remplacées par une ligne de dépendance entre `ReportService` et `Container` car `ReportService` obtient un générateur de rapports à partir de `Container`.

Figure 1.2

Employer un conteneur pour gérer les composants.



Nous pouvons à présent écrire une classe `Main` pour tester notre conteneur et des composants.

```
package com.apress.springrecipes.report;

public class Main {

    public static void main(String[] args) {
        Container container = new Container();
        ReportService reportService =
            (ReportService) container.getComponent("reportService");
        reportService.generateAnnualReport(2007);
    }
}
```

Dans la méthode `main()`, nous commençons par créer une instance du conteneur, à partir de laquelle nous obtenons ensuite le composant `ReportService`. Puis, lorsque nous invoquons la méthode `generateAnnualReport()` sur `ReportService`, la requête de génération du rapport est prise en charge par `PdfReportGenerator`, comme cela a été fixé par le conteneur.

Pour résumer, l'emploi d'un conteneur permet de diminuer le couplage entre les différents composants d'un système et, par conséquent, d'améliorer leur indépendance et leur réutilisabilité. Cette approche permet de séparer la configuration (par exemple, le type de générateur de rapports à utiliser) de la logique de programmation (par exemple, comment générer un rapport au format PDF) et ainsi de promouvoir la réutilisabilité du système global. Pour améliorer encore notre conteneur, nous pouvons lire un fichier de configuration qui définit les composants (voir la section 1.5).

1.2 Utiliser un localisateur de service pour simplifier la recherche

Problème

Lorsque la gestion des composants est assurée par un conteneur, leurs dépendances se placent au niveau de leur interface, non de leur implémentation. Cependant, ils ne peuvent consulter le conteneur qu'en utilisant un code propriétaire complexe.

Solution

Pour simplifier la recherche de nos composants, nous pouvons appliquer le design pattern *Service Locator* (localisateur de service) proposé par Sun dans Java EE. L'idée sous-jacente à ce pattern est simple : utiliser un localisateur de service pour encapsuler la logique complexe de recherche, tout en présentant des méthodes simples pour la consultation. N'importe quel composant peut ensuite déléguer les requêtes de recherche à ce localisateur de services.

Explications

Supposons que nous devions réutiliser les composants `ReportGenerator` et `ReportService` dans d'autres conteneurs, avec des mécanismes de recherche différents, comme JNDI. `ReportGenerator` ne présente aucune difficulté. En revanche, ce n'est pas le cas pour `ReportService` car nous avons incorporé la logique de recherche dans le composant lui-même. Nous devons la modifier avant de pouvoir réutiliser cette classe.

```
package com.apress.springrecipes.report;

public class ReportService {

    private ReportGenerator reportGenerator =
        (ReportGenerator) Container.instance.getComponent("reportGenerator");
    ...
}
```

Un localisateur de service peut être mis en œuvre par une simple classe qui encapsule la logique de recherche et présente des méthodes simples pour la consultation.

```
package com.apress.springrecipes.report;

public class ServiceLocator {

    private static Container container = Container.instance;

    public static ReportGenerator getReportGenerator() {
        return (ReportGenerator) container.getComponent("reportGenerator");
    }
}
```

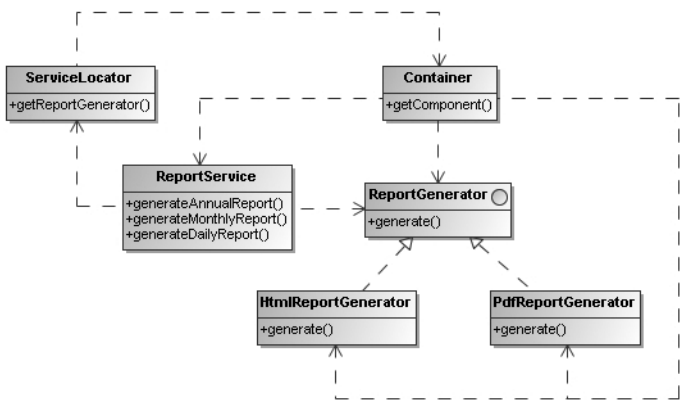
Ensuite, dans `ReportService`, il suffit d'invoquer `ServiceLocator` pour obtenir un générateur de rapports, au lieu d'effectuer directement la recherche.

```
package com.apress.springrecipes.report;

public class ReportService {

    private ReportGenerator reportGenerator =
        ServiceLocator.getReportGenerator();

    public void generateAnnualReport(int year) {
        ...
    }

    public void generateMonthlyReport(int year, int month) {
        ...
    }

    public void generateDailyReport(int year, int month, int day) {
        ...
    }
}
```

La Figure 1.3 présente le diagramme de classes UML après application du pattern Service Locator. La ligne de dépendance qui allait initialement de `ReportService` à `Container` passe à présent par `ServiceLocator`.

Figure 1.3

Appliquer le pattern Service Locator de manière à réduire la complexité de la recherche.



En appliquant le pattern Service Locator, nous séparons la logique de recherche de nos composants et simplifions ainsi cette recherche. Ce pattern améliore également les possibilités de réutilisation des composants dans des environnements fondés sur des mécanismes de recherche différents. N'oubliez pas que ce design pattern est utilisé non pas seulement dans la recherche de composants, mais également dans celle de ressources.

1.3 Appliquer l'inversion de contrôle et l'injection de dépendance

Problème

Lorsqu'un composant a besoin d'une ressource externe, comme une source de données ou une référence à un autre composant, l'approche la plus directe et la plus judicieuse consiste à effectuer une recherche. Nous disons que cette opération est une recherche *active*. Elle a pour inconvénient d'obliger le composant à connaître le fonctionnement de la recherche des ressources, même si la logique est encapsulée dans un localisateur de service.

Solution

Pour la recherche de ressources, une meilleure solution consiste à appliquer l'inversion de contrôle (IoC). L'idée de ce principe est d'inverser le sens de la recherche des ressources. Dans une recherche traditionnelle, les composants dénichent les ressources en consultant un conteneur qui renvoie dûment les ressources en question. Avec l'IoC, le conteneur délivre lui-même des ressources aux composants qu'il gère. Ces derniers doivent simplement choisir une manière d'accepter les ressources. Nous qualifions cette approche de recherche *passive*.

L'IoC est un principe général, tandis que l'injection de dépendance (DI) est un design pattern concret qui incarne ce principe. Dans le pattern DI, la responsabilité d'injection des ressources appropriées dans chaque composant, en respectant un mécanisme prédéfini, par exemple au travers d'un mutateur (*setter*), est dévolue au conteneur.

Explications

Pour appliquer le pattern DI, notre `ReportService` expose un mutateur qui accepte une propriété du type `ReportGenerator`.

```
package com.apress.springrecipes.report;

public class ReportService {

    private ReportGenerator reportGenerator; // Recherche active inutile.

    public void setReportGenerator(ReportGenerator reportGenerator) {
        this.reportGenerator = reportGenerator;
    }

    public void generateAnnualReport(int year) {
        ...
    }

    public void generateMonthlyReport(int year, int month) {
        ...
    }
```

```
    public void generateDailyReport(int year, int month, int day) {
        ...
    }
}
```

Le conteneur se charge d'injecter les ressources nécessaires dans chaque composant. Puisque la recherche active n'existe plus, nous pouvons retirer la variable statique `instance` dans `Container` et supprimer la classe `ServiceLocator`.

```
package com.apress.springrecipes.report;
...
public class Container {

    // Il est inutile de s'exposer pour être localisé par les composants.
    // public static Container instance;

    private Map<String, Object> components;

    public Container() {
        components = new HashMap<String, Object>();

        // Il est inutile d'exposer l'instance du conteneur.
        // instance = this;

        ReportGenerator reportGenerator = new PdfReportGenerator();
        components.put("reportGenerator", reportGenerator);

        ReportService reportService = new ReportService();
        reportService.setReportGenerator(reportGenerator);
        components.put("reportService", reportService);
    }

    public Object getComponent(String id) {
        return components.get(id);
    }
}
```

La Figure 1.4 présente le diagramme de classes UML après application de l'inversion de contrôle. La ligne de dépendance qui va de `ReportService` à `Container` (voir Figure 1.2) peut être retirée même sans l'aide de `ServiceLocator`.

Figure 1.4

Appliquer le principe IoC à l'obtention de ressources.