I, [core] Khái niệm tight-coupling (liên kết ràng buộc) và cách loosely coupled

1, Giới thiệu

* tight-coupling (liên kết rằng buộc) là một khái niệm trong java ám chỉ việc mối quan hệ giữa các class quá chặt chẽ. Khi yêu cầu thay đổi logic hay một class bị lỗi dẫn tới ảnh hưởng tới toàn bộ các class liên quan đến nó.
* Loosely-coupled là cách ám chỉ việc làm giảm bớt sự phụ thuộc giữa các class với nhau.

VD:

1, Bạn có một Class thực thi một nhiệm vụ cực kỳ phức tạp, và một trong số đó là việc sắp xếp dữ liệu trước khi xử lý.

public class BubbleSortAlgorithm{

public void sort(int[] array) {

// TODO: Add your logic here

System.out.println("Đã sắp xếp bằng thuật toán sx nổi bọt");

}

}

public class VeryComplexService {

// tạo mới 1 thằng BubblesortAlgorithm

private BubbleSortAlgorithm bubbleSortAlgorithm = new BubbleSortAlgorithm();

public VeryComplexService(){

}

public void complexBusiness(int array[]){

bubbleSortAlgorithm.sort(array);

// TODO: logic here

}

}

* Với cách làm ở trên, VeryComplexService đã hoàn thiện được nhiệm vụ, tuy nhiên, khi có yêu cầu thay đổi thuật toán sắp xếp sang QuickSort thì nghe vẻ chúng ta sẽ phải sửa lại hoàn toàn 2 Class ở trên ( sửa logic ở BubbleSortAlgorithm và sửa lại biến ở VeryComplexService )
* Ngoài ra BubbleSortAlgorithm sẽ chỉ tồn tại nếu VeryComplexService tồn tại, vì VeryComplexService tạo đối tượng BubbleSortAlgorithm bên trong nó ( hay nói cách khác là sự sống chết của BubbleSortAlgorithm sẽ do VeryComplexService quyết định ), theo như cách impplement này, nó là liên kết rất chặt với nhau

2,

public interface SortAlgorithm {

/\*\*

\* Sắp xếp mảng đầu vào

\* @param array

\*/

public void sort(int array[]);

}

public class BubbleSortAlgorithm implements SortAlgorithm{

@Override

public void sort(int[] array) {

// TODO: Add your logic here

System.out.println("Đã sắp xếp bằng thuật toán sx nổi bọt");

}

}

public class VeryComplexService {

private SortAlgorithm sortAlgorithm;

public VeryComplexService(){

sortAlgorithm = new BubbleSortAlgorithm();

}

public void complexBusiness(int array[]){

sortAlgorithm.sort(array);

// TODO: more logic here

}

* Với cách làm này, VeryComplexService sẽ chỉ quan hệ với một interface là SortAlgorithm với cách này thì mối quan hệ giảm bớt sự liên kết, nhưng nó không thay đổi được việc thuật toán vẫn đang là BubbleSortAlgorithm

3, cách 3

public interface SortAlgorithm {

\* Sắp xếp mảng đầu vào

public void sort(int array[]);

}

public class BubbleSortAlgorithm implements SortAlgorithm{

@Override

public void sort(int[] array) {

// TODO: Add your logic here

System.out.println("Đã sắp xếp bằng thuật toán sx nổi bọt");

}}

public class QuicksortAlgorithm implements SortAlgorithm {

@Override

public void sort(int[] array) {

// TODO: Add your logic here

System.out.println("Đã sắp xếp bằng thuật sx nhanh");

}}

public class VeryComplexService {

private SortAlgorithm sortAlgorithm;

public VeryComplexService(SortAlgorithm sortAlgorithm){

this.sortAlgorithm = sortAlgorithm;

}

public void complexBusiness(int array[]){

sortAlgorithm.sort(array);

// TODO: more logic here

}

}

public static void main(String[] args) {

SortAlgorithm bubbleSortAlgorithm = new BubbleSortAlgorithm();

SortAlgorithm quickSortAlgorithm = new QuicksortAlgorithm();

VeryComplexService business1 = new VeryComplexService(bubbleSortAlgorithm);

VeryComplexService business2 = new VeryComplexService(quickSortAlgorithm);

}

* Cách thứ ba này cũng là cách phổ biến nhất. Mối liên hệ giữa 2 Class đã “lỏng lẻo” hơn trước rấ nhiều. VeryComplexService sẽ không quan tâm đến thuật toán sắp xếp là gì nữa, mà chỉ cần tập trung vào nghiệp vụ. Còn SortAlgorithm sẽ được đưa vào từ bên ngoài tùy theo nhu cầu sử dụng.

II, [core] Giải thích Dependency Injection (DI) và IoC bằng Ngọc Trinh

1, Giới thiệu

* Dependency Injection is a design pattern….
* Có thể hiểu nôm na rằng đó là một phương pháp lập trình, là một thiết kế, template để ta có thể dựa theo để code giúp code clear hơn, dễ hiểu và theo một chuẩn đã được nghiên cứu
* Vậy cuối cùng Dependency Injection là gì?

Vd:

public class Girl{

// tạo 1 class outfit, khai báo nó để sử dụng

private Bikini outfit; // mỗi cô gái sẽ có một bộ bikini khi ra ngoài

public Girl(){

// biến outfit ở đây đang gọi đến class bikini để sử dụng các thuộc tính, phương thức của lớp bikini

outfit = new Bikini(); // Khi bạn tạo ra 1 cô gái, bạn cho cô ta mặc Bikini chẳng hạn

}

}

* Trước hết, qua đoạn code này, bạn sẽ thấy là khi tạo ra một Girl, bạn sẽ tạo ra thêm 1 bộ bikini đi kèm với cô gái đó. Lúc này, Bikini tồn tại mang ý nghĩa là dependency (phụ thuộc) của Girl
* Khi khởi tạo thuộc tính như này, bạn vô tình tạo ra một điểm thắt nút trong chương trình của mình, giả sử, Girl muốn mặc một món đồ khác skirthWithTshirt (Váy + áo) hoặc Nake (Trần chuồng)? hoặc nguy hiểm hơn là Class Bikini bị hỏng ( code lớp Bikini không hoạt động) ?
* Mình sẽ phải sửa lại toàn bộ class or code mới hoàn toàn

2. vấn đề là

* Các Class không nên phụ thuộc vào các kế thừa cấp thấp, mà nên phụ thuộc vào Abtraction (lớp trừu tượng).
* Giải thích:

Vd:

// Một interface cho việc ăn mặc

public interface Outfit {

public void wear();

}

// Một object cấp thấp, implement của Outfits

public class Bikini implements Outfit {

public void wear() {

System.out.println("Đã mặc Bikini");

}

}

// Bây giờ Girl chỉ phụ thuộc vào Outfit. nếu muốn thay đổi đồ của cô gái, chúng ta chỉ cần cho Outfit một thể hiện mới.

// hiểu đơn giản: nếu muốn thay đổi Class bikini ta chỉ cần tạo ra 1 class mới implement từ Interface Outfit là được

public class Girl{

private Outfit outfit;

public Girl(){

outfit = new Bikini();

}

// có thể hiểu khi class girl được tạo ra đồng thời nó đã tự động new 1 lớp gì đó (bikini) như vậy chúng ta không thể thay đổi giá trị truyền vào của lớp girl nữa

}

* Tới đây, chúng ta mới chỉ Abtract hóa thuộc tính của Girl mà thôi, còn thực tế, Girl vẫn đang gắn với một bộ Bikini duy nhất, Vậy muốn thay đồ thì chúng ta phải sửa thêm 1 chút xíu nữa.

Sửa lại code:

public class Girl{

private Outfit outfit;

public Girl(Outfit anything){

this.outfit = anything // Tạo ra một cô gái, với một món đồ tùy biến

// Không bị phụ thuộc quá nhiều vào thời điểm khởi tạo, hay code.

// ở đây outfit là biến truyền vào nên ta có thể thay đổi tùy ý

}

}

public class Main {

public static void main(String[] args) {

Outfit bikini = new Bikini(); // Tạo ra đối tượng Bikini ở ngoài đối tượng

Girl ngocTrinh = new Girl(bikini); // Mặc nó vào cho cô gái khi tạo ra cô ấy.

// new Girl(bikini): đây chính là Injection

}

}

* Với đoạn code ở trên, chúng ta đã gần như tách được bikini ra hoàn toàn khỏi Girl. Điều này làm giảm sự phụ thuộc giữa Girl và Bikini. Mà tăng tính tùy biến, linh hoạt cho code. Bây giờ Girl sẽ hoạt động với Outfit mà thôi. Và Outfit ở đâu ra? Chúng ta tạo ra và đưa nó vào (Injection) cô gái Girl.
* Khái niệm Denpendency Injection từ đây mà ra =vvvvv
* Dependency Injection là việc các Object nên phụ thuộc vào các Abstract Class và thể hiện chi tiết của nó sẽ được Inject vào đối tượng lúc runtime.
* Bây giờ muốn Girl mặc gì khác, bạn chỉ cần tạo một Class kế thừa Outfit và Inject (tiêm) nó vào Girl là xong.
* Cách để Inject dependency vào một đối tượng có thể kể đến như sau:
* **Constructor Injection**: là cái ở ví dụ, Inject dependency ngay vào Contructor.
* **Setter Injection**: nó sẽ gần giống so với constructor. Vd: girl.setOutfit (new naked())
* **Interface Injection**: mỗi Class muốn inject cái gì, thì phải implement một Interface có chứa hàm inject(xx) (gần như thay thế cho setter). Nếu muốn inject gì đó thì gọi hàm inject(xx) ra. Cách này khó

3, Inversion of Control

* Dependency Injection giúp chúng ta dễ dàng mở rộng code và giảm sự phụ thuộc giữa các dependency với nhau. Tuy nhiên, lúc này, khi code bạn sẽ phải kiêm thêm nhiệm vụ Inject dependency (tiêm sự phụ thuộc). thử tưởng tượng một Class có vài chục dependency thì bạn sẽ phải tự tay inject bao nhiêu lần.
* Gây khó khăn trong quản lý code và denpendency cũng như lúc code

Vd:

public static void main(String[] args) {

Outfit bikini = new Bikini();

Accessories gucci = new GucciAccessories();

HairStyle hair = new KoreanHairStyle();

Girl ngocTrinh = new Girl(bikini, gucci, hair…);

}

* Giá mà có em nào đấy đến inject hộ thì tốt nhỉ :>>
* Bây giờ giả sử, chúng ta định nghĩa trước toàn bộ các dependency có trong Project, mô tả nó và tống nó vào 1 cái kho và giao cho 1 thằng framework quản lý. Bất kỳ các Class nào khi khởi tọa, nó cần dependency gì, thì cái framework đấy sẽ tìm trong kho rồi inject vào đối tượng thay chúng ta. Tiện hơn nhỉ?
* Đó chính là nguyên lý của Inversion of Control (IOC) – đảo chiều sự điều khiển
* Inversion of Control is a programming principle. Flow of control within the application is not controlled by the application it self, but rather by the underlying framework.
* Khi đó, code chúng ta sẽ chỉ cần như này, để lấy ra 1 đối tượng:

@Override

public void run(String... args) throws Exception {

Girl girl = context.getBean(Girl.class);

}

* Đó chính là lý do Spring framework ra đời để thực hiện ý tưởng Inversion of Control (IOC), tuy nhiên, theo thời gian, Spring lớn mạnh và trở thành một hệ sinh thái rộng lớn phục vụ nhiều chức năng trên nền tảng IoC này.

III, [Basic] @Component và @Autowired

1, project

* Cài đặt các gói thư viện của Spring Boot trong Maven bằng cách thêm gọi

spring-boot-starter-parent (là parent của toàn bộ project )

<parent>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-parent</artifactId>

<version>2.0.5.RELEASE</version>

<relativePath /> <!-- lookup parent from repository -->

</parent>

* Các thư viện cho lập trình web hoặc server side, chúng ta thêm:

<dependencies>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-web</artifactId>

</dependency>

</dependencies>

* File pom.xml

<?xml version="1.0" encoding="UTF-8"?>

<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://maven.apache.org/POM/4.0.0http://maven.apache.org/xsd/maven-4.0.0.xsd">

<modelVersion>4.0.0</modelVersion>

<packaging>pom</packaging>

<parent>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-parent</artifactId>

<version>2.0.5.RELEASE</version>

<relativePath /> <!-- lookup parent from repository -->

</parent>

<groupId>me.loda.spring</groupId>

<artifactId>spring-boot-learning</artifactId>

<version>0.0.1-SNAPSHOT</version>

<name>spring-boot-learning</name>

<description>Everything about Spring Boot</description>

<properties>

<java.version>1.8</java.version>

</properties>

<dependencies>

<!--spring mvc, rest-->

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-web</artifactId>

</dependency>

</dependencies>

<build>

<plugins>

<plugin>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-maven-plugin</artifactId>

</plugin>

</plugins>

</build>

</project>

2, cách chạy ứng dụng

* Nếu trong Java truyền thống, khi chạy cả một project, chúng ta sẽ phải định nghĩa một hàm main() và để nó chạy đầu tiên thì ở đây chúng ta cũng vẫn không thay đổi =>> !!
* Có khác là chúng ta sẽ chỉ cho Spring Boot biết nơi nó khởi chạy lần đầu, để nó cài đặt mọi thứ.
* Cách thực hiện là thêm annotation @SpringBootApplication trên class chính và gọi SpringApplication.run (App.class, args); để chạy project.

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

import org.springframework.context.ApplicationContext;

@SpringBootApplication

public class App {

public static void main(String[] args) {

SpringApplication.run(App.class, args); }}

* Nếu qua bài Dependency Injection (DI) và IoC. Thì có thể hiểu nhiệm vụ của Spring là tạo ra một cái Contianer chứa các Dependency cho chúng ta.
* SpringApplication.run (App.class, args) chính là câu lệnh để tạo ra container. Sau đó nó tìm toàn bộ các denpendency trong project của bạn và đưa vào đó.
* Spring đặt tên cho container là ApplicationContex và dependency là Bean
* Vậy làm sao để Spring biết đâu là dependency? Chúng ta tới với khái niệm @component

3, @Component

* Là một Annotation đánh dấu trên các Class để giúp Spring biết nó là một Bean.

Vd:

* Chúng ta có 1 interface Outfit

public interface Outfit {

public void wear();

}

* Implement nó là Class Bikini

/\*

Đánh dấu class bằng @Component

Class này sẽ được Spring Boot hiểu là một Bean (hoặc dependency)

Và sẽ được Spring Boot quản lý

\*/

@Component

public class Bikini implements Outfit {

@Override

public void wear() {

System.out.println("Mặc bikini");

}

}

* Chạy chương trình, xem kết quả:

@SpringBootApplication

public class App {

public static void main(String[] args) {

// ApplicationContext chính là container, chứa toàn bộ các Bean

ApplicationContext context = SpringApplication.run(App.class, args);

// Khi chạy xong, lúc này context sẽ chứa các Bean có đánh

// dấu @Component.

// Lấy Bean ra bằng cách

Outfit outfit = context.getBean(Outfit.class);

// In ra để xem thử nó là gì

System.out.println("Instance: " + outfit);

// xài hàm wear()

outfit.wear();

}

}

Output:

[1] Instance: me.loda.spring.helloworld.Bikini@1e1f6d9d

[2] Mặc bikini

* Bạn sẽ thấy Outfit lúc này chính là Bikini. Class được đánh dấu là @Component
* Spring Boot khi chạy sẽ dò tìm toàn bộ các Class cùng cấp được ở trong các package thấp hơn so với Class App mà bạn đã cung cấp cho Spring ( chúng ta có thể cấu hình việc tìm kiếm này, sẽ đề cập sau). Trong quá trình dò tìm này, khi gặp một class được đánh dấu @Component thì nó sẽ tạo ra một instance và đưa vào Applicationcontex để quản lý.

4, @Autowired

* Bây giờ mình tạo ra 1 class Girl và có một thuộc tính là Outfit.
* Mình cũng đánh dấu Girl là một @component. Tức Spring boot cần tạo ra một instance của girl để quản lý.

@Component

public class Girl {

@Autowired

Outfit outfit;

public Girl(Outfit outfit) {

this.outfit = outfit;

}

// GET

// SET

}

* Đánh dấu thuộc tính Outfit của Girl bởi annotation @Autowired. Điều này nói với Spring Boot hãy tự inject (tiêm) một instance của Outfit vào thuộc tính này khi khởi tạo Girl.
* Chạy chương trình:

@SpringBootApplication

public class App {

public static void main(String[] args) {

// ApplicationContext chính là container, chứa toàn bộ các Bean

ApplicationContext context = SpringApplication.run(App.class, args);

// Khi chạy xong, lúc này context sẽ chứa các Bean có đánh

// dấu @Component.

// Lấy Bean ra bằng cách

Outfit outfit = context.getBean(Outfit.class);

// In ra để xem thử nó là gì

System.out.println("Output Instance: " + outfit);

// xài hàm wear()

outfit.wear();

Girl girl = context.getBean(Girl.class);

System.out.println("Girl Instance: " + girl);

System.out.println("Girl Outfit: " + girl.outfit);

girl.outfit.wear();

}

}

Output:

[1] Output Instance: me.loda.spring.helloworld.Bikini@2e16f13a

[2] Mặc bikini

[3] Girl Instance: me.loda.spring.helloworld.Girl@353cb1cb

[4] Girl Outfit: me.loda.spring.helloworld.Bikini@2e16f13a

[5] Mặc bikini

* Spring boot đã tự tạo ra một Girl và trong quá tình tạo ra đó, nó truyền Outfit vào làm thuộc tính

5, Singleton

* Điều đặc biệt là các Bean đươc quản lý bên trong ApplicationContext đề là singleton
* Tức là các class này đều được tạo 1 lần duy nhất trong toàn bộ quá trình chạy

[1] Output Instance: me.loda.spring.helloworld.Bikini@2e16f13a

[4] Girl Outfit: me.loda.spring.helloworld.Bikini@2e16f13a

* Tất cả những Bean được quản lý trong ApplicationContext đều chỉ được tọa ra một lần duy nhất và khi có Class yêu cầu @Autowired thì nó sẽ lấy đối tượng đã được tạo sẵn trong ApplicationContext để inject vào.
* Trong trường hợp muốn mỗi lần sử dụng là một instance hoàn toàn mới. thì hãy đánh dấu @Component đó bằng @Scope(“Prototype”)

@Component

@Scope("prototype")

public class Bikini implements Outfit {

@Override

public void wear() {

System.out.println("Mặc bikini");

}

}