Spring 基于 Java 的配置

## **基于 Java 的配置**

到目前为止，你已经看到如何使用 XML 配置文件来配置 Spring bean。如果你熟悉使用 XML 配置，那么我会说，不需要再学习如何进行基于 Java 的配置是，因为你要达到相同的结果，可以使用其他可用的配置。

基于 Java 的配置选项，可以使你在不用配置 XML 的情况下编写大多数的 Spring，但是一些有帮助的基于 Java 的注解，解释如下：

## **@Configuration 和 @Bean 注解**

带有 ****@Configuration**** 的注解类表示这个类可以使用 Spring IoC 容器作为 bean 定义的来源。****@Bean**** 注解告诉 Spring，一个带有 @Bean 的注解方法将返回一个对象，该对象应该被注册为在 Spring 应用程序上下文中的 bean。最简单可行的 @Configuration 类如下所示：

package com.tutorialspoint;

import org.springframework.context.annotation.\*;

@Configuration

public class HelloWorldConfig {

@Bean

public HelloWorld helloWorld(){

return new HelloWorld();

}

}

上面的代码将等同于下面的 XML 配置：

<beans>

<bean id="helloWorld" class="com.tutorialspoint.HelloWorld" />

</beans>

在这里，带有 @Bean 注解的方法名称作为 bean 的 ID，它创建并返回实际的 bean。你的配置类可以声明多个 @Bean。一旦定义了配置类，你就可以使用 *AnnotationConfigApplicationContext* 来加载并把他们提供给 Spring 容器，如下所示：

public static void main(String[] args) {

ApplicationContext ctx =

new AnnotationConfigApplicationContext(HelloWorldConfig.class);

HelloWorld helloWorld = ctx.getBean(HelloWorld.class);

helloWorld.setMessage("Hello World!");

helloWorld.getMessage();

}

你可以加载各种配置类，如下所示：

public static void main(String[] args) {

AnnotationConfigApplicationContext ctx =

new AnnotationConfigApplicationContext();

ctx.register(AppConfig.class, OtherConfig.class);

ctx.register(AdditionalConfig.class);

ctx.refresh();

MyService myService = ctx.getBean(MyService.class);

myService.doStuff();

}

## **例子**

让我们在这里是 ****HelloWorldConfig.java**** 文件的内容：

package com.tutorialspoint;

import org.springframework.context.annotation.\*;

@Configuration

public class HelloWorldConfig {

@Bean

public HelloWorld helloWorld(){

return new HelloWorld();

}

}

这里是 ****HelloWorld.java**** 文件的内容：

package com.tutorialspoint;

public class HelloWorld {

private String message;

public void setMessage(String message){

this.message = message;

}

public void getMessage(){

System.out.println("Your Message : " + message);

}

}

下面是 ****MainApp.java**** 文件的内容：

package com.tutorialspoint;

import org.springframework.context.ApplicationContext;

import org.springframework.context.annotation.\*;

public class MainApp {

public static void main(String[] args) {

ApplicationContext ctx =

new AnnotationConfigApplicationContext(HelloWorldConfig.class);

HelloWorld helloWorld = ctx.getBean(HelloWorld.class);

helloWorld.setMessage("Hello World!");

helloWorld.getMessage();

}

}

一旦你完成了创建所有的源文件并添加所需的额外的库后，我们就可以运行该应用程序。你应该注意这里不需要配置文件。如果你的应用程序一切都正常，将输出以下信息：

Your Message : Hello World!

## **注入 Bean 的依赖性**

当 @Beans 依赖对方时，表达这种依赖性非常简单，只要有一个 bean 方法调用另一个，如下所示：

package com.tutorialspoint;

import org.springframework.context.annotation.\*;

@Configuration

public class AppConfig {

@Bean

public Foo foo() {

return new Foo(bar());

}

@Bean

public Bar bar() {

return new Bar();

}

}

这里，foo Bean 通过构造函数注入来接收参考基准。现在，让我们看到一个正在执行的例子：

## **例子:**

这里是 ****TextEditorConfig.java**** 文件的内容：

package com.tutorialspoint;

import org.springframework.context.annotation.\*;

@Configuration

public class TextEditorConfig {

@Bean

public TextEditor textEditor(){

return new TextEditor( spellChecker() );

}

@Bean

public SpellChecker spellChecker(){

return new SpellChecker( );

}

}

这里是 TextEditor.java 文件的内容：

package com.tutorialspoint;

public class TextEditor {

private SpellChecker spellChecker;

public TextEditor(SpellChecker spellChecker){

System.out.println("Inside TextEditor constructor." );

this.spellChecker = spellChecker;

}

public void spellCheck(){

spellChecker.checkSpelling();

}

}

下面是另一个依赖的类文件 ****SpellChecker.java**** 的内容：

package com.tutorialspoint;

public class SpellChecker {

public SpellChecker(){

System.out.println("Inside SpellChecker constructor." );

}

public void checkSpelling(){

System.out.println("Inside checkSpelling." );

}

}

下面是 ****MainApp.java**** 文件的内容：

package com.tutorialspoint;

import org.springframework.context.ApplicationContext;

import org.springframework.context.annotation.\*;

public class MainApp {

public static void main(String[] args) {

ApplicationContext ctx =

new AnnotationConfigApplicationContext(TextEditorConfig.class);

TextEditor te = ctx.getBean(TextEditor.class);

te.spellCheck();

}

}

一旦你完成了创建所有的源文件并添加所需的额外的库后，我们就可以运行该应用程序。你应该注意这里不需要配置文件。如果你的应用程序一切都正常，将输出以下信息：

Inside SpellChecker constructor.

Inside TextEditor constructor.

Inside checkSpelling.

## **@Import 注解:**

****@import**** 注解允许从另一个配置类中加载 @Bean 定义。考虑 ConfigA 类，如下所示：

@Configuration

public class ConfigA {

@Bean

public A a() {

return new A();

}

}

你可以在另一个 Bean 声明中导入上述 Bean 声明，如下所示：

@Configuration@Import(ConfigA.class)

public class ConfigB {

@Bean

public B b() {

return new B();

}

}

现在，当实例化上下文时，不需要同时指定 ConfigA.class 和 ConfigB.class，只有 ConfigB 类需要提供，如下所示：

public static void main(String[] args) {

ApplicationContext ctx =

new AnnotationConfigApplicationContext(ConfigB.class);

// now both beans A and B will be available...

A a = ctx.getBean(A.class);

B b = ctx.getBean(B.class);

}

## **生命周期回调**

@Bean 注解支持指定任意的初始化和销毁的回调方法，就像在 bean 元素中 Spring 的 XML 的初始化方法和销毁方法的属性：

public class Foo {

public void init() {

// initialization logic

}

public void cleanup() {

// destruction logic

}

}

@Configuration

public class AppConfig {

@Bean(initMethod = "init", destroyMethod = "cleanup" )

public Foo foo() {

return new Foo();

}

}

指定 Bean 的范围：

默认范围是单实例，但是你可以重写带有 @Scope 注解的该方法，如下所示：

@Configuration

public class AppConfig {

@Bean

@Scope("prototype")

public Foo foo() {

return new Foo();

}

}